A Course on Deterministic Distributed Algorithms

Jukka Suomela

8 April 2014

Contents

Foreword

This course is a brief introduction to the theory of distributed algorithms, more specifically, *deterministic, synchronous network algorithms*. The topics covered in this course include algorithmic techniques that can be used to solve graph problems efficiently in extremely large networks, as well as fundamental impossibility results that put limitations on distributed computing.

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.

Acknowledgements. Many thanks to Mika Göös, Juho Hirvonen, Teemu Kuusisto, Tuomo Lempiäinen, and Jussi Väisänen for comments, and to Juho Hirvonen, Joel Kaasinen, and Joel Rybicki for helping me with the arrangements of this course. This work was supported by the Academy of Finland, Grant 252018. For updates and additional material, see

http://www.cs.helsinki.fi/jukka.suomela/dda

Chapter 1

Introduction and Preliminaries

1.1 Scope

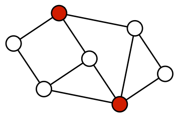
This course focuses on the *theoretical foundations of distributed systems*. Our approach is similar to typical courses on models of computation, computational complexity, and design and analysis of algorithms. The main difference is in the models of computation that we study: instead of traditional models, such as finite state machines, Turing machines, RAM machines, or Boolean circuits, our model of choice is a *distributed system*.

1.1.1 Distributed Systems as a Model of Computation

A distributed system consists of multiple machines that are connected to each other through communication links. We usually view a distributed system as a (simple, undirected) graph $G = (V, E)$: each node $v \in V$ represents a machine and an edge $\{u, v\} \in E$ represents a communication link between machines u and v.

To understand the key difference between distributed systems and more familiar models of computation, let us consider an illustrative example: the problem of finding a maximal independent set.

An *independent set* for a graph $G = (V, E)$ is a set $I \subseteq V$ such that for each edge $\{u, v\} \in E$ at most one of u and v is in I. An independent set I is *maximal* if it cannot be extended, i.e., it is not a proper subset of another independent set.



Now given any model of computation X we can pose the familiar question:

- *Computability:* is it *possible* to find a maximal independent set in model X?
- *Computational complexity:* can we find a maximal independent set *efficiently* in model X?

We are familiar with such questions in the context of Turing machines, but it is not immediately obvious what these questions mean in the context of distributed systems. The following informal comparison illustrates the key differences.

Input. The input is a graph G.

Turing machines: We assume that the structure of G is encoded as a string and given to the Turing machine on the input tape.

Distributed systems: We assume that the structure of the input graph G is the same as the structure of the distributed system. Initially, each machine $v \in V$ only knows some local information related to v (for example, the degree of v and the unique identifier of v). To acquire more information about G, the nodes need to exchange messages.

Output. The output is an independent set $I \subseteq V$.

Turing machines: We require that the machine prints an encoding of I on the output tape.

Distributed systems: We require that each node $v \in V$ produces one bit of output: if $v \in I$, node v has to output 1, and if $v \notin I$, node v has to output 0.

Algorithm. We say that an algorithm solves the problem if it produces a valid output for any valid input.

Turing machines: The algorithm designer chooses the state transitions of the Turing machine.

Distributed systems: The algorithm designer writes one program. The same program is installed in each $v \in V$.

Complexity measures. There are many possible complexity measures, but perhaps the most commonly used is the time complexity.

Turing machines: Time = number of elementary steps. In each time unit, (1) the machine moves the tape heads, (2) performs a state transition that depends on the contents of the tapes, and (3) possibly halts.

Distributed systems: Time = number of synchronous communication rounds. In each time unit, all machines in parallel (1) exchange messages with their neighbours, (2) perform state transitions that depend on the messages that they received, and (3) possibly halt.

To oversimplify a bit, distributed computation is not really about *computation* — it is all about *communication*. Throughout this course, we will see striking examples of the implications of this change of perspective.

1.1.2 Outside the Scope

The term "distributed computing" is overloaded, and it means very different things to different people.

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 such as SETI@Home and Folding@Home. However, this is *not* the definition that we use, and our course is in no way related to large-scale number crunching.

In general, our focus is on theory, not practice. For our purposes, a communication network is an idealised abstraction. We are not interested in any implementation details or engineering aspects. For example, the following topics are *not* covered on this course: physical properties of wired or wireless media, modulation techniques, communication protocols, standards, software architectures, programming languages, software libraries, privacy, and security.

Within the field of the theory of distributed computing, there are also numerous topics that we are not going to cover. We will conclude this course with a brief overview of other research areas within the field in Section 7.1.

1.2 Graphs

As we already saw in Section 1.1.1, 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 graph-theoretic concepts that we will use.

1.2.1 Terminology

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

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 *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

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

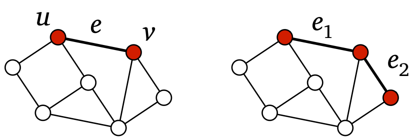


Figure 1.1: Node u is adjacent to node v. Nodes u and v are incident to edge e. Edge e_1 is adjacent to edge e_2.

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 $v \in V$.

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 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 edges E_2.

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 1.2. 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 $1 \leq i < j \leq \ell$; it follows that the length of a cycle is at least 3.

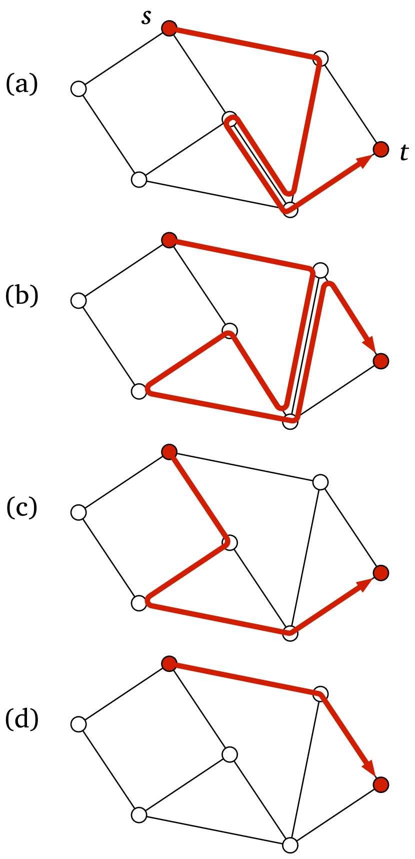


Figure 1.2: (a) A walk of length 5 from s to t. (b) A non-backtracking walk. (c) A path of length 4. (d) A path of length 2; this is a shortest path and hence $\text{dist}_G(s,t) = 2$.

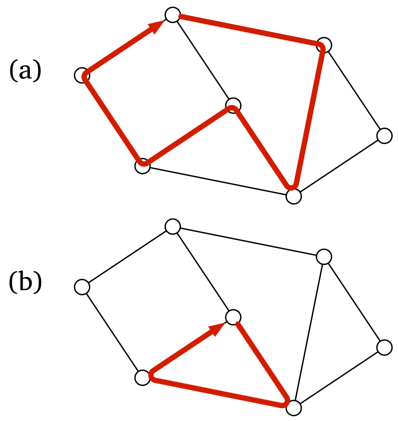


Figure 1.3: (a) A cycle of length 6. (b) A cycle of length 3; this is a shortest cycle and hence the girth of the graph is 3.

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 $\operatorname{dist}_G(u, v) = \ell$. If u and v are not in the same connected component, we define $\operatorname{dist}_G(u, v) = \infty$. Note that $\operatorname{dist}_G(u, u) = 0$ 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:

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

A graph is *connected* if it consists of one connected component. The *diameter* of graph G, in notation $\operatorname{diam}(G)$, is the length of a longest shortest path, that is, the maximum of $\operatorname{dist}_G(u, v)$ over all $u, v \in V$; we have $\operatorname{diam}(G) = \infty$ if the graph is not connected.

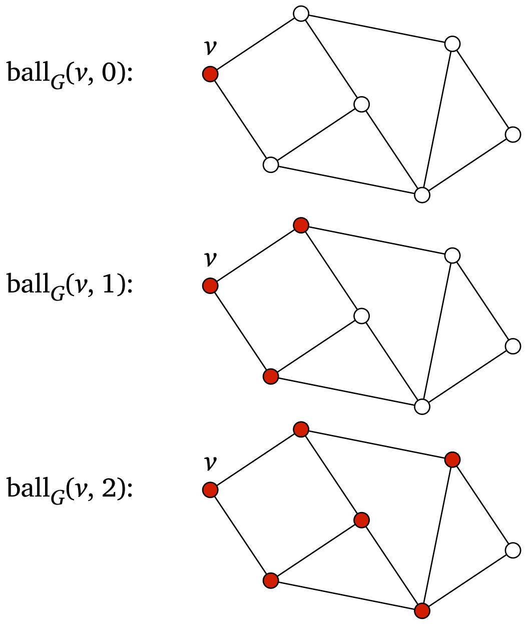


Figure 1.4: Neighbourhoods.

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

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

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

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.

1.2.2 Packing and Covering

A subset of nodes $X \subseteq V$ is

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

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

(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

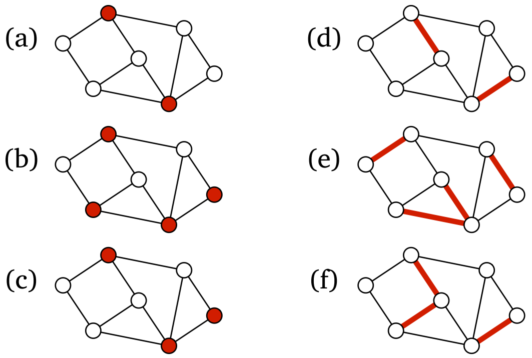


Figure 1.5: Packing and covering problems; see Section 1.2.2.

(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 1.5 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 1.5a 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 1.5d 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.

1.2.3 Labellings and Partitions

We will often encounter functions of the form

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

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

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

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 $f : V \to \{1, 2, \dots, k\}$ 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 $f : E \to \{1, 2, \dots, k\}$ 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 1.6 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 to a proper vertex colouring with k colours; the term *k-edge colouring* is defined in an analogous manner.

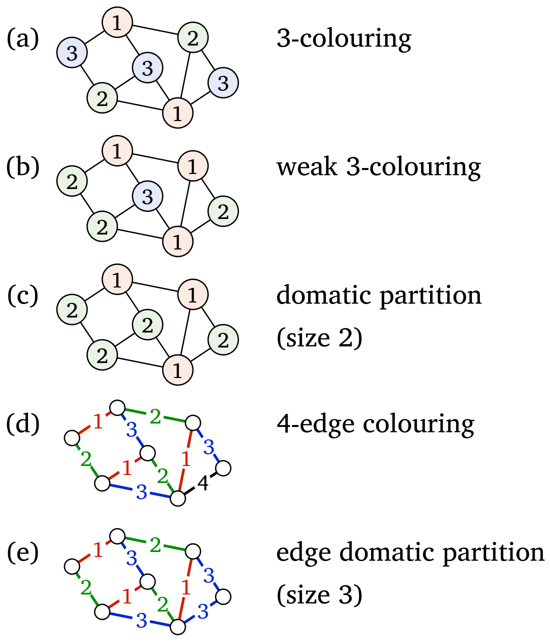


Figure 1.6: Partition problems; see Section 1.2.3.

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.

1.2.4 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 $f : E \to \{1, 2, \dots, k\}$ is a *d-factorisation* of G if $f^{-1}(i)$ is a d-factor for each i. See Figure 1.7 for examples.

We make the following observations:

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

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

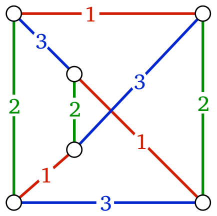
(c) The subgraph induced by a 2-factor consists of disjoint cycles.

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

1.2.5 Approximations

So far we have encountered a number of maximisation problems and minimisation problems. More formally, the definition of a maximisation

(a)



(b)

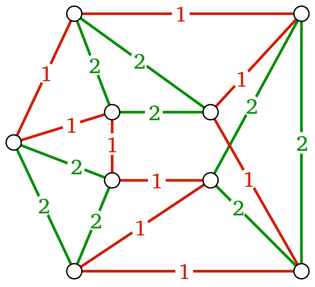


Figure 1.7: (a) A 1-factorisation of a 3-regular graph. (b) A 2-factorisation of a 4-regular graph.

problem consists of two parts: a set of *feasible solutions* $\mathscr{S}$ and an *objective function* $g\colon \mathscr{S} \to \mathbb{R}$. In a maximisation problem, the goal is to find a feasible solution $X \in \mathscr{S}$ that maximises $g(X)$. A minimisation problem is analogous: the goal is to find a feasible solution $X \in \mathscr{S}$ 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 $\mathscr{S}$ consists of all matchings in G, and we simply define $g(M) = |M|$ for each matching $M \in \mathscr{S}$.

As another example, the problem of finding an optimal colouring is a minimisation problem. The set of feasible solutions $\mathscr{S}$ consists of all proper vertex colourings, and $g(f)$ is the number of colours in $f \in \mathscr{S}$.

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

Similarly, if $(\mathscr{S}, g)$ is a minimisation problem, we say that a solution X is an α-approximation if $X \in \mathscr{S}$, and we have $g(X) \le \alpha g(Y)$ for all $Y \in \mathscr{S}$. 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.

1.2.6 Directed Graphs and Orientations

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

1.3 Exercises

Exercise 1.1 (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 1.2 (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 1.3 (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 1.4 (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 1.5 (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: Assume that D is an edge dominating set; show that you can construct a maximal matching M with $|M| \le |D|$.

Exercise 1.6 (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 1.7 (line graphs)**.** Look up the definition of a *line graph*. Whenever possible, use line graphs to explain

(a) the connection between matchings and independent sets,

(b) the connection between dominating sets and edge dominating sets,

(c) the connection between node colourings and edge colourings.

Exercise 1.8 (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.

Exercise 1.9 (Petersen 1891)**.** Show that any $2d$-regular graph has a 2-factorisation.

Exercise 1.10 (orientations)**.** Using the result of Exercise 1.9, 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 $v \in V$.

Chapter 2

Port-Numbering Model

2.1 Introduction

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: distributed algorithm in the "port-numbering model". The basic idea 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.

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) More precisely, a node of degree d corresponds to a computer with d communication ports that are labelled with numbers $1, 2, \dots, d$. Each port is connected to precisely one neighbour.

(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 2.2 and 2.3 below go through the effort of formalising this idea. While at it, we also address the following issues:

(a) It is easy to encode a subset of nodes using local outputs — but how should we encode, for example, a subset of edges?

(b) Often it is useful to have not only local outputs but also a *local input* for each computer. Then we could compose algorithms: first, algorithm A_1 solves a problem Π_1; then algorithm A_2 uses the solution of Π_1 to solve a problem Π_2.

(c) Often we will focus our attention to certain families of graphs — it is too much to expect that an algorithm could solve a problem in *any* undirected graph G.

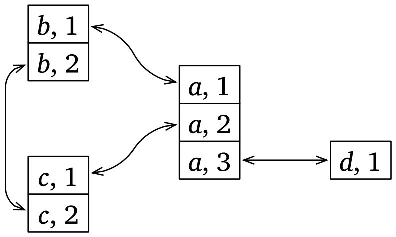


Figure 2.1: A port-numbered network $N = (V, P, p)$. There are four nodes, $V = \{a, b, c, d\}$; the degree of node a is 3, the degrees of nodes b and c are 2, and the degree of node d is 1. The connection function p is illustrated with arrows — for example, $p(a, 3) = (d, 1)$ and conversely $p(d, 1) = (a, 3)$. This network is simple.

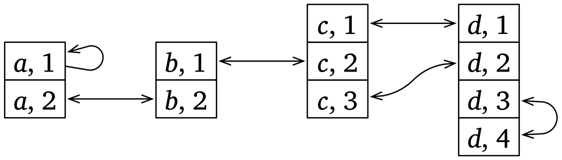


Figure 2.2: A port-numbered network $N = (V, P, p)$. There is a loop at node a, as $p(a, 1) = (a, 1)$, and another loop at node d, as $p(d, 3) = (d, 4)$. There are also multiple connections between c and d. Hence the network is not simple.

2.2 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 2.1 and 2.2 for illustrations.

2.2.1 Terminology

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

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

If $(v, i) \in P$, 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.

2.2.2 Intuition

The intuitive idea behind the definition is that a simple port-numbered network N is a model of a physical, real-world communication network:

(a) each node $v \in V$ is a physical device (e.g., a computer or a router),

(b) node v has $\deg_N(v)$ communication ports, labelled with integers $1, 2, \dots, \deg_N(v)$,

(c) $p(u, i) = (v, j)$ indicates that there is a cable that connects the port number i in device u with the port number j in device v.

2.2.3 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 2.3 for an illustration.

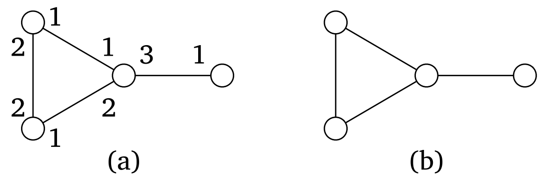


Figure 2.3: (a) An alternative drawing of the simple port-numbered network N from Figure 2.1. (b) The underlying graph G of N.

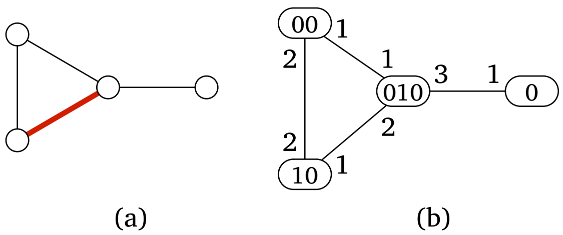


Figure 2.4: (a) A graph $G = (V, E)$ and a matching $M \subseteq E$. (b) A port-numbered network N; graph G is the underlying graph of N. The node labelling $f : V \to \{0, 1\}^*$ is an encoding of matching M.

2.2.4 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 that a node labelling $f : V \to Y$ can be used to represent the following graph-theoretic structures; see Figure 2.4 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.

Note that the above encoding is natural from the perspective of distributed systems. For example, assume that we have used a node labelling $f: V \to Y$ to encode a matching $M \subseteq E$. Now the label $f(v)$ of a node $v \in V$ effectively describes M in the immediate neighbourhood of v. In particular, $f(v)$ indicates whether v is matched, i.e., whether there is a node u such that $\{u, v\} \in M$, and if this is the case, which of the ports is connected to u.

2.2.5 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 2.2.4, 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.

2.3 Distributed Algorithms in the Port-Numbering Model

We proceed to give a formal definition of a distributed algorithm in the port-numbering model. In essence, a distributed algorithm is a 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.

It should be noted that 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 a higher-level pseudocode to define algorithms and omit the tedious details of translating the high-level description into a state machine.

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

(iv) Msg_A is the set of possible messages.

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

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

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

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

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

2.3.2 Execution

Let A be a distributed algorithm, let $N = (V, P, p)$ be a port-numbered network, and let $f : V \to \mathrm{Input}_A$ be a labelling of the nodes.

A *state vector* is a function $x \colon V \to \mathrm{States}_A$. The *execution* of A on (N, f) is a sequence of state vectors $x_0, x_1, \dots$ defined recursively as follows.

The initial state vector x_0 is defined by

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

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

Now assume that we have defined state vector x_{t-1}. Define $m_t\colon P \to \mathrm{Msg}_A$ 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

$$m_t(u) = \bigl(m_t(u,1), m_t(u,2), \dots, m_t(u,d)\bigr).$$

Finally, we define the new state vector x_t by

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

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.

2.3.3 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 2.2.5). 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 $T\colon \mathbb{N} \to \mathbb{N}$, we say that A solves the problem *in time* T.

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

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 $T : n \mapsto k$.

2.4 Examples

In this section, we will give two examples of distributed algorithms that solve distributed graph problems. We will give an *informal* presentation of the algorithms — formalising the algorithms as state machines is left as an exercise.

2.4.1 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 $M \subseteq E$.

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 MS(i) are stopping states. If the state of a node v is 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

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

The algorithm is presented in Table 2.1; see Figure 2.5 for an illustration.

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

Lemma 2.1. *Assume that u is a white node, v is a black node, and $(u,i) = p(v,j)$. Then at least one of the following holds:*

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

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

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

Lemma 2.2. *Algorithm BMM stops in time $2\Delta + 1$, where Δ is the maximum degree of N.*

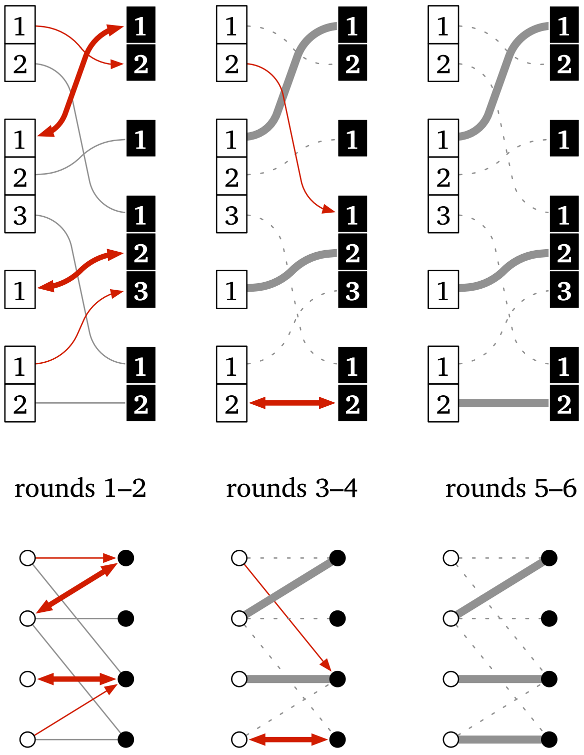


Figure 2.5: Algorithm BMM; the illustration shows the algorithm both from the perspective of the port-numbered network N and from the perspective of the underlying graph G. Arrows pointing right are proposals, and arrows pointing left are acceptances. Wide grey edges have been added to matching M.

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 2.1: Algorithm BMM; here $k = 1, 2, \ldots$.

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 2.1, 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 2.3. *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 MS(i) if and only if v stops in state MS(j). If u stops in state MS(i), it has received an '*accept*' from v, and v stops in state MS(j). Conversely, if v stops in state MS(j), it has received a '*proposal*' from u and it sends an '*accept*' to u, after which u stops in state MS(i).

Let us then verify that M is indeed maximal. If this was not the case, there would be an unmatched white node u that is connected to an unmatched black node v. However, Lemma 2.1 implies that at least one of them becomes matched before or during round 2Δ. □

2.4.2 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.

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 2.6 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':

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

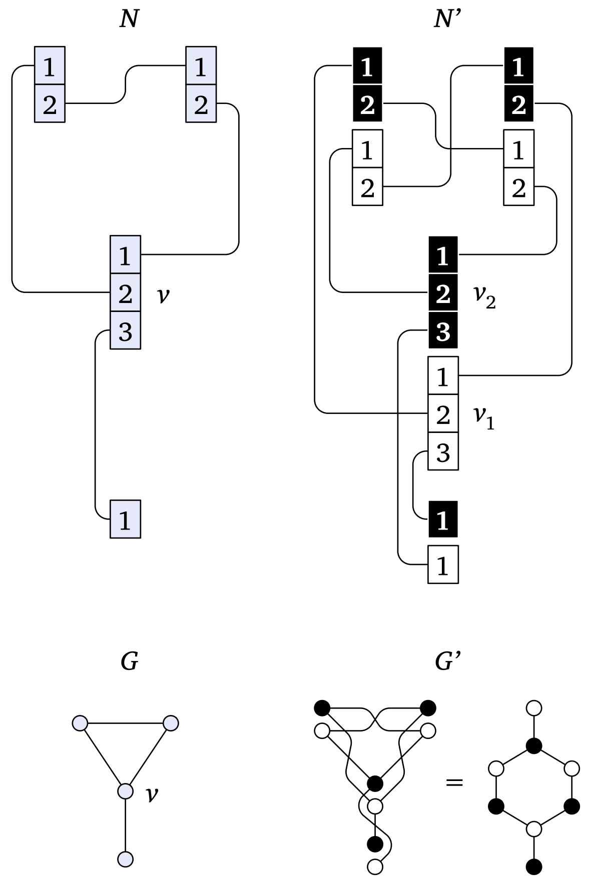


Figure 2.6: Construction of the virtual network N' in algorithm VC3.

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*.

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 2.4.1 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.

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.

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:

$$M = \{\, \{u, v\} \in E : \{u_1, v_2\} \in M' \text{ or } \{u_2, v_1\} \in M' \,\}. \tag{2.1}$$

See Figure 2.7 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 2.4. *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 maximum degree at most 2. Set C consists of all nodes in H. We will then argue that any vertex cover

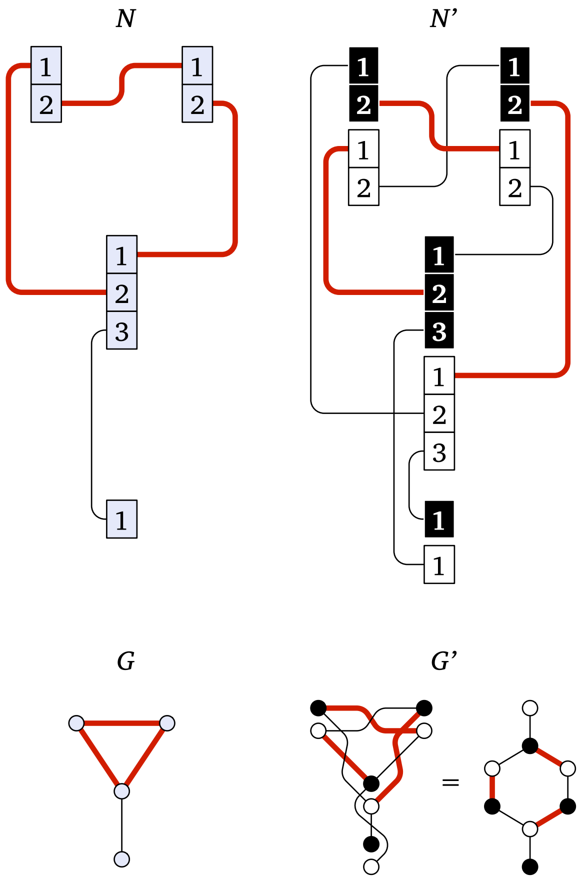


Figure 2.7: Set $M \subseteq E$ (left) and matching $M' \subseteq E'$ (right).

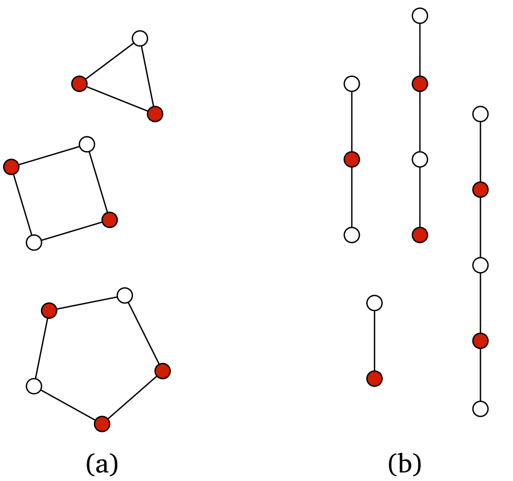


Figure 2.8: (a) In a cycle with n nodes, any vertex cover contains at least $n/2$ nodes. (b) In a path with n nodes, any vertex cover contains at least $n/3$ nodes.

C^* contains at least a fraction $1/3$ of the nodes in H; see Figure 2.8 for an example. Then it follows that C is at most 3 times as large as a minimum vertex cover.

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

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

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

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

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

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

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

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

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

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

In summary, 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.

2.5 Exercises

Exercise 2.1 (stopped nodes). In the formalism of this section, 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 2.2 (formalising BMM). Present algorithm BMM from Section 2.4.1 in a formally precise manner, using the definitions of Sections 2.2 and 2.3. Try to make Msg_A as small as possible.

Exercise 2.3 (formalising VC3). Present algorithm VC3 from Section 2.4.2 in a formally precise manner, using the definitions of Sections 2.2 and 2.3. Try to make both Msg_A and States_A as small as possible.

Hint: 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.

Exercise 2.4 (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 2.5 (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 2.6 (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 2.7 (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: 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.

Chapter 3

Impossibility Results

3.1 Introduction

In the previous chapter, we have seen examples of problems that can be solved with a distributed algorithm in the port-numbering model. However, there are many problems that cannot be solved.

As a very simple example, let $N = (V, P, p)$ be a port-numbered network with two nodes, u and v, that are connected to each other:

$$\begin{tikzcd} \boxed{u, 1} \arrow[r, leftrightarrow] & \boxed{v, 1} \end{tikzcd}$$

Assume that we are given a labelling $f(u) = f(v) = 0$. Now let A be any distributed algorithm, and consider the execution of A on (N, f). As the local inputs of u and v are identical, we will have $x_0(u) = x_0(v)$ after the initialisation, that is, nodes u and v have *identical states before round* 1. It follows that the message sent by u to v in round 1 is the same as the message sent by v to u in round 1. Therefore we will have $x_1(u) = x_1(v)$, that is, nodes u and v have *identical states after round* 1. By induction, we have $x_t(u) = x_t(v)$ for any round t. In particular, if A stops in time T, we will have $x_T(u) = x_T(v)$, i.e., both u and v produce the same local output.

This reasoning already shows that A cannot produce a proper colouring, a maximal independent set, a minimum vertex cover, etc. — in each of these cases nodes u and v would have to produce distinct outputs. We generalise this observation in Section 3.2, when we introduce a very useful graph-theoretic tool, covering maps.

There are also many problems that can be solved with a distributed algorithm, but it requires a lot of time. Techniques that are useful in proving time lower bounds will be introduced in Section 3.3.

3.2 Covering Maps

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.

3.2.1 Definition

Let $N = (V, P, p)$ and $N' = (V', P', p')$ be port-numbered networks, and let $\phi : V \to V'$. We say that ϕ is a *covering map from N to N'* if the following holds:

(a) ϕ is a surjection: $\phi(V) = V'$.

(b) ϕ preserves degrees: $\deg_N(v) = \deg_{N'}(\phi(v))$ for all $v \in V$.

(c) ϕ preserves connections and port numbers: $p(u,i) = (v,j)$ implies $p'(\phi(u),i) = (\phi(v),j)$.

See Figures 3.1–3.3 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:

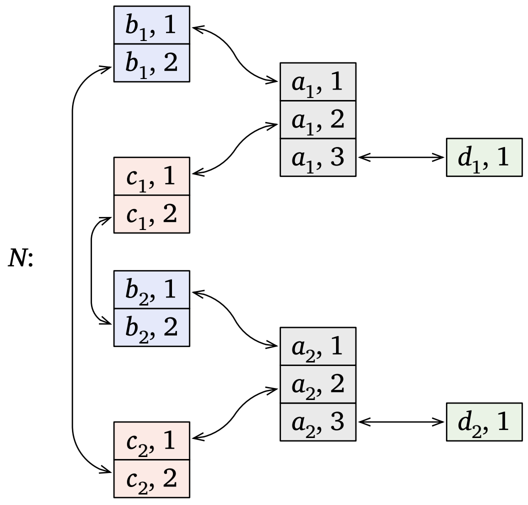
(d) ϕ preserves labels: $f(v) = f'(\phi(v))$ for all $v \in V$.

3.2.2 Covers and Executions

Now we will study covering maps from the perspective of distributed algorithms. The basic idea is that a covering map ϕ from N to N' fools any distributed 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 3.1. *Assume that*



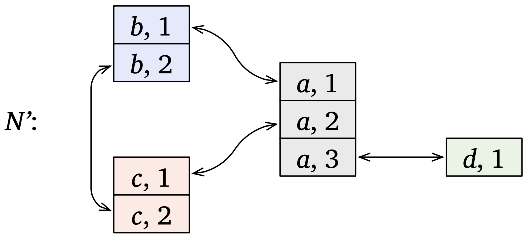


Figure 3.1: There is a covering map ϕ from N to N' that maps $a_i \mapsto a$, $b_i \mapsto b$, $c_i \mapsto c$, and $d_i \mapsto d$ for each $i \in \{1, 2\}$.

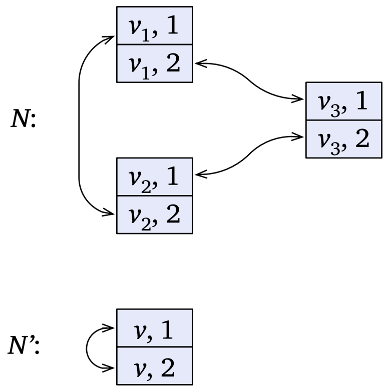


Figure 3.2: There is a covering map ϕ from N to N' that maps $v_i \mapsto v$ for each $i \in \{1,2,3\}$. Here N is a simple port-numbered network but N' is not.

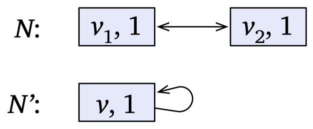


Figure 3.3: There is a covering map ϕ from N to N' that maps $v_i \mapsto v$ for each $i \in \{1,2\}$. Again, N is a simple port-numbered network but N' is not.

(a) *A is a distributed 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 2.3.2; 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 $v \in V$.

3.2.3 Examples

We will give representative examples of negative results that we can easily derive from Theorem 3.1. First, we will observe that a distributed algorithm cannot break symmetry in a cycle — unless we provide some symmetry-breaking information in local inputs.

Lemma 3.2. *Let $G = (V, E)$ be a cycle graph, let A be a distributed 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 $V = \{ v_1, v_2, \dots, v_n \}$ along the cycle so that the edges are

$$E = \{ \{v_1, v_2\}, \ \{v_2, v_3\}, \ \dots, \ \{v_{n-1}, v_n\}, \ \{v_n, v_1\} \}.$$

Choose the port numbering p as follows:

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

See Figure 3.2 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 3.1, 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 3.1 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 3.2 has many immediate corollaries.

Corollary 3.3. *Let $\mathscr{F}$ be the family of cycle graphs. Then there is no distributed 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 $G \in \mathscr{F}$ such that a constant function is not a feasible solution in the network N that we constructed in Lemma 3.2.

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 3.4. *There is no 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 3.1.

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 3.4. 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 3.2; 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 3.4.

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 3.4. 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.

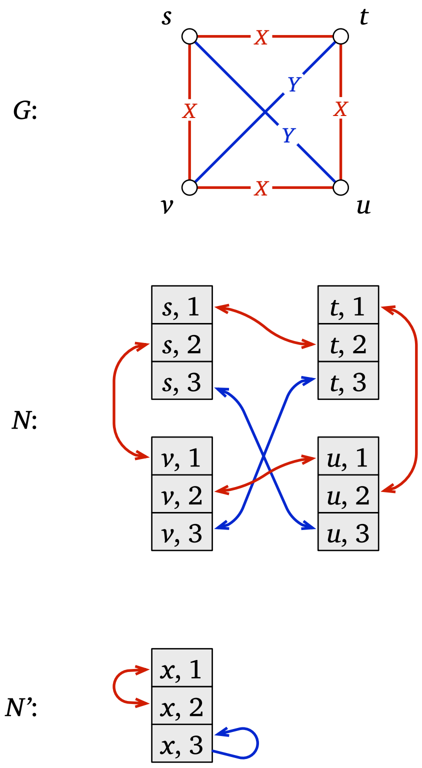


Figure 3.4: Graph G is the complete graph on four nodes. The edges of G can be partitioned into a 2-factor X and a 1-factor Y. Network N has G as its underlying graph, and there is a covering map ϕ from N to N'

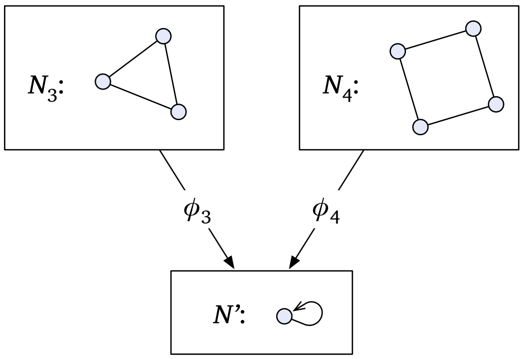


Figure 3.5: The structure of the proof of Lemma 3.5.

So far we have used only one covering map in our proofs; the following lemma gives an example of the use of more than one covering map.

Lemma 3.5. *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 distributed 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 3.2; 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 3.2, 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 3.5. The local inputs are assumed to be all zeroes.

Let A be a distributed algorithm, and let c be the output of the only node of N'. If we apply Theorem 3.1 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 3.1 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 distributed algorithm cannot determine the length of a cycle. In particular, a distributed algorithm cannot determine if a graph is bipartite.

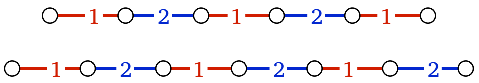
3.3 Local Neighbourhoods

Covering maps can be used to argue that a problem cannot be solved at all. Now we will study a technique that can be used to argue that a problem cannot be solved *fast*.

Some problems can be solved very quickly with a distributed algorithm. For example, algorithm VC3 from Section 2.4.2 runs in time $O(\Delta)$, where Δ is the maximum degree of the graph. If we focus on a family of bounded-degree graphs, i.e., $\Delta = O(1)$, this is a constant-time algorithm — the running time of the algorithm is independent of the size of the graph.

3.3.1 An Introductory Example

However, some problems cannot be solved quickly with any distributed algorithm. As an introductory example, let $\mathscr{F}$ consist of all path graphs, and let Π be the problem of finding a 2-edge colouring.



With a little thought, we can design a distributed algorithm A that solves Π on $\mathscr{F}$. Informally, algorithm A proceeds as follows. First, we find the midpoint of the graph. This is possible if nodes of degree 1 generate a token that is forwarded by nodes of degree 2. Eventually, the two tokens meet at the midpoint of the graph. There are two cases:

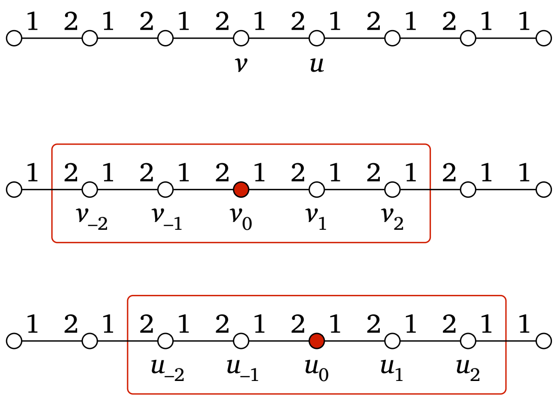


Figure 3.6: Nodes u and v have isomorphic radius-r neighbourhoods in a path of length $2r+3$; in this illustration, $r=2$.

(a) The midpoint is a node v, i.e., we have an even path. Then we can use the port numbers of v to break symmetry: the edge connected to port i is labelled with colour i. Then we can assign alternating colours to all other edges, starting from v.

(b) The midpoint is an edge $\{u,v\}$, i.e., we have an odd path. Then we can assign colour 1 to $\{u,v\}$ and alternating colours to all other edges, starting from both u and v.

The algorithm certainly finds a correct solution — in any path graph, the edges will be properly coloured with colours 1 and 2. However, the running time of the algorithm is $\Theta(n)$, where n is the number of nodes.

We will now argue that no algorithm can find a 2-edge colouring in time $o(n)$. To this end, assume that G is a path of length $2r+3$, and let N be a simple port-numbered network that has G as the underlying graph; choose the port numbers as shown in Figure 3.6.

Now let u and v be the two nodes that are incident to the midpoint of the path. Let us label the nodes in the radius-r neighbourhoods of u

and v as we have shown in Figure 3.6:

$$\begin{aligned}\operatorname{ball}_G(u,r) &= \{\, u_{-r}, u_{-r+1}, \dots, u_r \,\},\\ \operatorname{ball}_G(v,r) &= \{\, v_{-r}, v_{-r+1}, \dots, v_r \,\}.\end{aligned}$$

In particular, $v = v_0$ and $u = u_0$.

Now assume that we have a distributed algorithm A, and we apply it to N. Initially, we have

$$x_0(u_i) = x_0(v_i) \quad \text{for all} \quad -r \le i \le r.$$

It follows that the messages sent by u_i and v_i on round 1 are identical for all $-r \le i \le r$. Therefore the messages received by u_i and v_i on round 1 are identical for all $-r+1 \le i \le r-1$ (note that u_r and v_r may receive different messages). It follows that after round 1 we have

$$x_1(u_i) = x_1(v_i) \quad \text{for all} \quad -r+1 \le i \le r-1.$$

By induction, after round $t \le r$ we have

$$x_t(u_i) = x_t(v_i) \quad \text{for all} \quad -r+t \le i \le r-t.$$

In particular,

$$x_r(u) = x_r(u_0) = x_r(v_0) = x_r(v).$$

Hence if A stops in time r, both u and v produce the same output. However, this contradicts with the definition of problem Π. Therefore the running time of A has to be larger than r in a graph with $2r+4$ nodes.

In what follows, we will formalise and generalise the ideas that we used in this example.

3.3.2 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

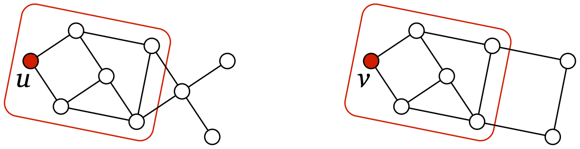


Figure 3.7: Nodes u and v have isomorphic radius-2 neighbourhoods, provided that we choose the port numbers appropriately. Therefore in any algorithm A the state of u equals the state of v at time $t = 0, 1, 2$. However, at time $t = 3, 4, \dots$ this does not necessarily hold.

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

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

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 $v \in U$.

(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 3.7 for an example.

3.3.3 Local Neighbourhoods and Executions

Theorem 3.6. *Assume that*

(a) A is a distributed 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 $u \in \text{ball}_G(v, r - t)$.

To prove the base case $t = 0$, let $u \in \text{ball}_G(v, r)$, $d = \deg_N(u)$, and $u' = \psi(u)$; we have

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

For the inductive step, assume that $t \geq 1$ and

$$u \in \text{ball}_G(v, r - t).$$

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

$$\text{dist}_G(s, v) \leq \text{dist}_G(s, u) + \text{dist}_G(u, v) \leq 1 + r - t.$$

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

$$x'_t(u') = x_t(u). \qquad \square$$

We will often consider the case that $N = N'$ but $v \neq v'$ when we apply Theorem 3.6; we have already seen an example of this in Section 3.3.1.

3.4 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 3.2 shows that all cycle graphs are homogeneous.

Exercise 3.1 (finding port numbers). Consider the graph G and network N' illustrated in Figure 3.8. 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 3.2 (homogeneity). Assume that G is homogeneous and it contains a node with degree at least two. Give several examples of graph problems that cannot be solved with any distributed algorithm in any family of graphs that contains G.

Exercise 3.3 (4-regular and homogeneous). Show that the graph illustrated in Figure 3.9 is homogeneous.

Hint: Apply the result of Exercise 1.9.

Exercise 3.4 (3-regular and homogeneous). Show that the graph illustrated in Figure 3.8 is homogeneous.

Hint: Find a 1-factor.

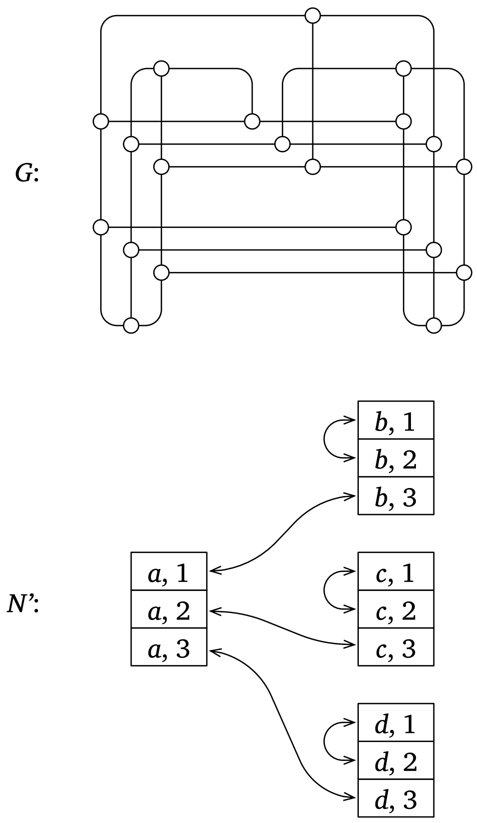


Figure 3.8: Graph G and network N' for Exercises 3.1 and 3.4.

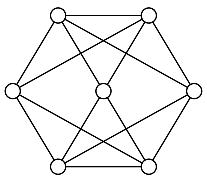


Figure 3.9: Graph for Exercise 3.3.

Exercise 3.5 (even degrees)**.** Show that any $2k$-regular graph is homogeneous, for any positive integer k.

Exercise 3.6 (complete graphs)**.** Show that any complete graph is homogeneous.

Hint: Show that if we have a complete graph with an even number of nodes, there is a 1-factorisation.

Exercise 3.7 (path graphs)**.** In this exercise, the graph family $\mathscr{F}$ consists of *path graphs*.

(a) Show that it is possible to find a maximum matching in time $3n + O(1)$.

(b) Show that it is not possible to find a maximum matching in time $n/3 + O(1)$.

(c) Show that it is not possible to find a 2-colouring.

(d) Show that it is not possible to find a weak 2-colouring.

(e) Is it possible to find a minimum vertex cover? If yes, how fast?

(f) Is it possible to find a minimum dominating set? If yes, how fast?

(g) Is it possible to find a minimum edge dominating set? If yes, how fast?

(h) How fast is it possible to find a 2-approximation of a minimum vertex cover?

(i) How fast is it possible to find a 2-approximation of a minimum dominating set?

Exercise 3.8 (path graphs with auxiliary information)**.** In this exercise, the graph family $\mathscr{F}$ consists of *path graphs*.

(a) Assume that we are given a 4-colouring. Show that it is possible to find a 3-colouring in time 1.

(b) Assume that we are given a 4-colouring. Show that it is not possible to find a 3-colouring in time 0.

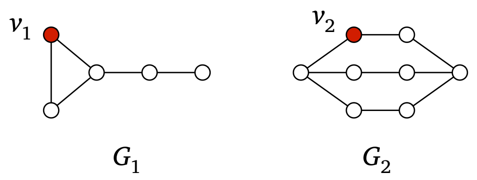


Figure 3.10: Graphs for Exercise 3.9.

(c) Assume that we are given a 4-colouring. Show that it is possible to find a 2-colouring in time $3n + O(1)$.

(d) Assume that we are given a 4-colouring. Show that it is not possible to find a 2-colouring in time $n/3 + O(1)$.

(e) Assume that we are given a 4-colouring. How fast is it possible to find a weak 2-colouring?

(f) Assume that we are given an orientation. Show that it is possible to find a 2-colouring in time $3n + O(1)$.

(g) Assume that we are given an orientation. Show that it is not possible to find a 2-colouring in time $n/3 + O(1)$.

Exercise 3.9 (combining techniques)**.** Consider the graphs G_1 and G_2 illustrated in Figure 3.10. Show that there are simple port-numbered networks N_1 and N_2 such that N_i has G_i as the underlying graph, and in any distributed algorithm with running time 2 the output of v_1 in N_1 equals the output of v_2 in N_2.

Hint: We need to combine the results of Theorems 3.1 and 3.6. 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.

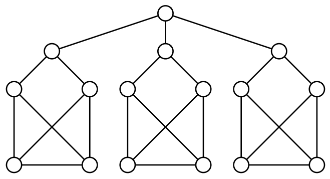


Figure 3.11: Graph G for Exercise 3.10.

Exercise 3.10 (3-regular and not homogeneous)**.** Consider the graph G illustrated in Figure 3.11.

(a) Show that G is not homogeneous.

(b) Present a distributed 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: Show that if a 3-regular graph is homogeneous, then it has a 1-factor. Show that G does not have any 1-factor.

Exercise 3.11 (covers with covers)**.** What is the connection between covering maps and algorithm VC3 of Section 2.4.2?

Exercise 3.12 (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 3.13 (k-fold covers)**.** Assume that $N = (V, P, p)$ and $N' = (V', P', p')$ are simple port-numbered networks, assume 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.

Exercise 3.14 (isomorphisms)**.** Construct port-numbered networks $N_1 = (V_1, P_1, p_1)$ and $N_2 = (V_2, P_2, p_2)$ such that $|V_1| = |V_2|$, both N_1 and N_2 are simple, the underlying graphs of N_1 and N_2 are connected, the underlying graphs of N_1 and N_2 are *not* isomorphic, and the following holds:

(a) There is a port-numbered network N, a covering map ϕ_1 from N_1 to N, and a covering map ϕ_2 from N_2 to N.

(b) There is a port-numbered network N', a covering map ϕ_1' from N' to N_1, and a covering map ϕ_2' from N' to N_2.

Exercise 3.15 (3-regular graphs)**.** Is it possible to construct connected 3-regular graphs $G = (V, E)$ and $G' = (V', E')$ with $|V| = |V'|$ such that the following holds: if N and N' are simple port-numbered networks that have G and G' as their underlying graphs, then there is no covering map from N to N'?

Chapter 4

Combinatorial Optimisation

4.1 Introduction

In this section, we will have a closer look at two optimisation problems: vertex covers and edge dominating sets.

In Section 2.4.2 we have already seen that it is possible to find a 3-approximation of a minimum vertex cover with a distributed algorithm. In Section 4.2, we will present a better algorithm that achieves the approximation factor of 2. Recall that this is optimal: Corollary 3.3 shows that it is not possible to find a 1.999-approximation with any distributed algorithm.

Once we have presented the vertex cover algorithm, we will turn our attention to the edge dominating set problem. This is the focus of the exercises in Section 4.3. Among others, we will design an algorithm that finds a 4-approximation of a minimum edge dominating set.

Throughout this chapter, we will design algorithms for *bounded-degree graphs*: we show that for each value of Δ, we can design an algorithm A_Δ that solves the problem in any graph of maximum degree at most Δ. The general case is left as an exercise.

4.2 Vertex Cover

In Exercise 1.3 we saw that if we are given a maximal matching, it is easy to find a 2-approximation of a minimum vertex cover. Unfortunately, Corollary 3.3 shows that we cannot find a maximal matching with a distributed algorithm.

In this section we will study so-called *maximal edge packings*. Maximal edge packings are closely related to maximal matchings — in particular, given a maximal edge packing, it is easy to find a 2-approximation of a minimum vertex cover. However, there is one crucial difference: while it is impossible to find maximal matchings with distributed algorithms, there is a distributed algorithm MEP that is able to find maximal edge packings.

To design algorithm MEP, we first introduce the concept of a *half-saturating edge packing* in Section 4.2.4. We design a distributed algorithm HSEP that finds a half-saturating edge packing. Then we use HSEP as a subroutine in algorithm MEP. Finally, algorithm VC2 uses algorithm MEP as a subroutine to find a 2-approximation of a minimum vertex cover.

4.2.1 Edge Packings

Let $G = (V, E)$ be a graph and let $f : E \to [0,1]$ be a function that assigns a real number $f(e)$ to each edge $e \in E$. We define the shorthand notation

$$f[v] = \sum_{e \in E :\, v \in e} f(e).$$

That is, $f[v]$ is the sum of values $f(e)$ over all edges e that are incident to v.

We say that f is an *edge packing* if $f[v] \leq 1$ for all $v \in V$. A node $v \in V$ is *saturated* if $f[v] = 1$, and an edge $e = \{u, v\} \in E$ is *saturated* if at least one of the nodes u and v is saturated. An edge packing f is *maximal* if all edges are saturated — see Figures 4.1 and 4.2 for examples.

4.2.2 Properties

The following facts are easy to verify:

(a) The constant function $f : e \mapsto 0$ is an edge packing. However, it is not a maximal edge packing unless $E = \varnothing$.

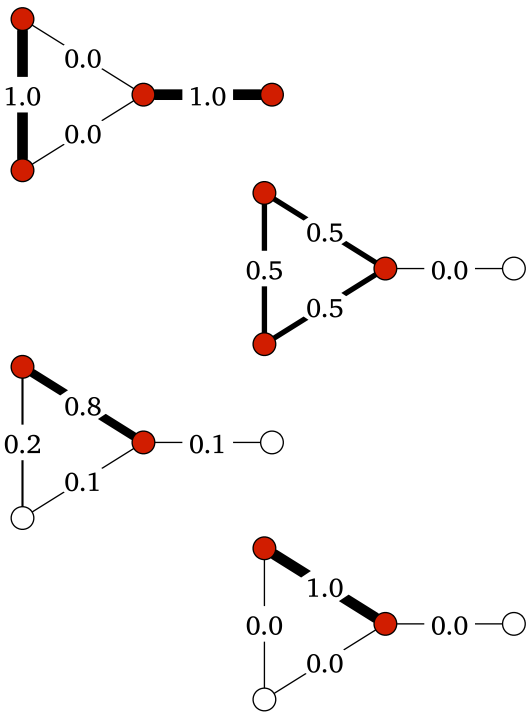


Figure 4.1: Maximal edge packings. Saturated nodes have been high-lighted.

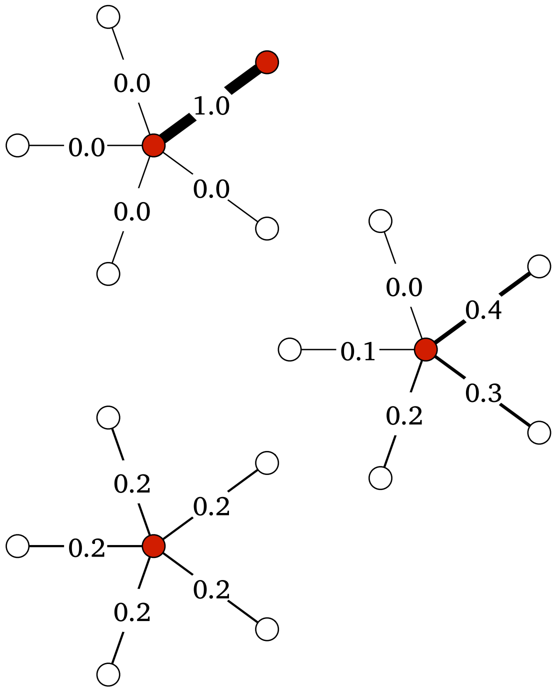


Figure 4.2: Maximal edge packings. Saturated nodes have been highlighted.

(b) If G is a d-regular graph, then the constant function $f: e \mapsto 1/d$ is a maximal edge packing. We will have $f[v] = 1$ for all nodes, that is, all nodes are saturated.

(c) Let $M \subseteq E$ be a subset of edges and let $f: E \to \{0,1\}$ be the indicator function of M, that is, $f(e) = 1$ if and only if $e \in M$. Now f is an edge packing if and only if M is a matching. Moreover, f is a maximal edge packing if and only if M is a maximal matching. A node v is saturated if and only if it is incident to an edge of M.

(d) Assume that f is an edge packing and f is not maximal. Then there is an edge $e_0 = \{u, v\} \in E$ such that neither u nor v is saturated. Let

$$\epsilon = \min\{1 - f[u],\, 1 - f[v]\}.$$

We have $\epsilon > 0$. Define the function

$$g(e) = \begin{cases} f(e) + \epsilon & \text{if } e = e_0, \\ f(e) & \text{otherwise.} \end{cases}$$

Now g is also an edge packing, and edge e_0 is saturated in g.

We will need the following technical lemma shortly.

Lemma 4.1. *Let $G = (V, E)$ be a graph, let $f: E \to [0,1]$ be an edge packing, and let $X \subseteq V$ be a subset of nodes. Then*

$$\sum_{v \in X} f[v] = \sum_{e \in E} f(e)|e \cap X|.$$

Proof. By definition, we have

$$\sum_{v \in X} f[v] = \sum_{v \in X} \sum_{e \in E:\, v \in e} f(e).$$

Now it is easy to verify that in the double sum, each edge $e \in E$ is counted precisely $|e \cap X|$ times. □

4.2.3 Edge Packings and Vertex Covers

Let $G = (V, E)$ be a graph, and let f be a maximal edge packing in G. Let $C \subseteq V$ consist of all saturated nodes.

Lemma 4.2. *Set C is a vertex cover.*

Proof. Let $e \in E$. By assumption, f is maximal, and therefore e is saturated, i.e., at least one endpoint of e is in C. $\square$

Lemma 4.3. *Set C is a 2-approximation of a minimum vertex cover.*

Proof. Let C^* be a minimum vertex cover; we will prove that $|C| \le 2|C^*|$. By definition, we have $f[v] = 1$ for all $v \in C$. Therefore

$$|C| = \sum_{v \in C} f[v].$$

By Lemma 4.1, we have

$$\sum_{v \in C} f[v] = \sum_{e \in E} f(e)|e \cap C|.$$

As C contains at most two endpoints of each edge and C^* contains at least one endpoint of each edge, we have

$$\sum_{e \in E} f(e)|e \cap C| \le 2 \sum_{e \in E} f(e)|e \cap C^*|.$$

Now we can apply Lemma 4.1 again to obtain

$$2 \sum_{e \in E} f(e)|e \cap C^*| = 2 \sum_{v \in C^*} f[v].$$

Finally, as f is an edge packing, we have $f[v] \le 1$, which implies

$$2 \sum_{v \in C^*} f[v] \le 2|C^*|.$$ $\square$

Informally, we have shown that maximal edge packings are as useful as maximal matching from the perspective of the vertex cover problem: both yield a 2-approximation of a minimum vertex cover.

Moreover, it also appears that maximal edge packings could be easier to find in a distributed setting. After all, we know that we cannot find a maximal matching in a cycle, while it is trivial to find a maximal edge packing in a cycle — set $f(e) = 1/2$ for each edge e.

In the following sections, we show that this is indeed the case: there is a distributed algorithm that finds a maximal edge packing in any graph. One such algorithm is a recursive scheme that is based on what we call half-saturating edge packings.

4.2.4 Half-Saturating Edge Packings

Let $G = (V, E)$ be a graph and let $f : E \to [0, 1]$ be an edge packing. We say that f is *half-saturating* if all of the following hold:

(a) $f(e) \in \{0,\ 1/2,\ 1\}$ for each $e \in E$,

(b) $f[v] = 0$ implies that $f[u] = 1$ for all neighbours u of v,

(c) $f[v] = 1/2$ implies that $f[u] = 1$ for at least one neighbour u of v.

Note that in a half-saturating edge packing we have $f[v] \in \{0,\ 1/2,\ 1\}$ for each node $v \in V$; see Figure 4.3 for an example.

The definition of a half-saturating edge packing may sound artificial and pointless. However, we will soon see that (i) it is easy to find half-saturating edge packings, and (ii) if we have an algorithm A that finds a half-saturating edge packing, we can find a maximal edge packing by a recursive application of A.

Half-saturating edge packings are not necessarily maximal edge packings. However, unsaturated edges have very specific properties.

Lemma 4.4. *If $f : E \to [0, 1]$ is a half-saturating edge packing, and an edge $e = \{u, v\} \in E$ is not saturated, then we have $f[u] = f[v] = 1/2$.*

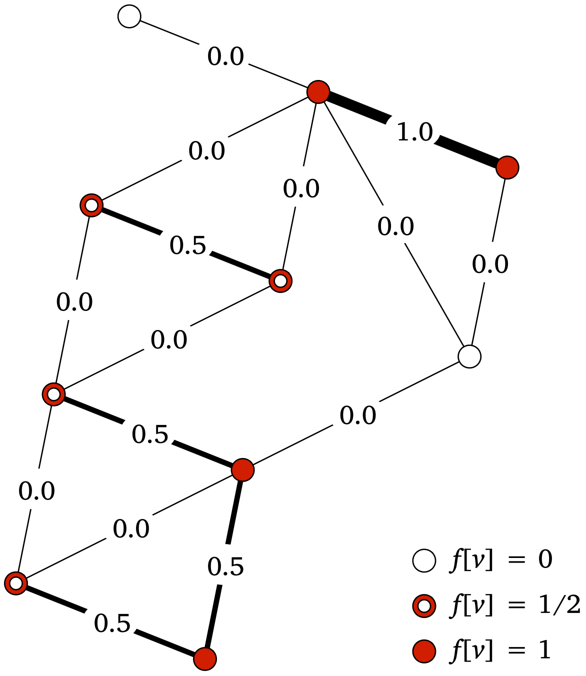


Figure 4.3: Graph G and a half-saturating edge packing f.

Proof. If we had $f[u] = 1$ or $f[v] = 1$, edge e would be saturated. If we had $f[u] = 0$, the definition of a half-saturated edge packing would imply $f[v] = 1$ and vice versa. Hence the only remaining case is $f[u] = f[v] = 1/2$. □

Motivated by the above lemma, let us focus on the subgraph G_f induced by the unsaturated edges. More formally, define

$$\begin{aligned} G_f &= (V_f, E_f), \\ E_f &= \big\{\{u,v\} \in E : f[u] = f[v] = 1/2\big\}, \\ V_f &= \bigcup E_f. \end{aligned}$$

Now E_f is the set of unsaturated edges and G_f is the subgraph of G induced by E_f; see Figure 4.4 for an illustration.

We will now make two observations: (i) the maximum degree of G_f is strictly smaller than the maximum degree of G, and (ii) if we can find a maximal edge packing for the subgraph G_f, we can easily construct a maximal edge packing for the original graph G.

Lemma 4.5. *If E_f is non-empty, the maximum degree of G_f is strictly smaller than the maximum degree of G.*

Proof. Let $u \in V_f$. Then we have $f[u] = 1/2$. By the definition of a half-saturating edge packing, there is an edge $e = \{u,v\} \in E$ with $f[v] = 1$. That is, $e \notin E_f$. Hence the degree of u in G_f is strictly smaller than the degree of u in G. In particular, if the maximum degree of G is at most Δ, the maximum degree of G_f is at most $\Delta - 1$. □

Lemma 4.6. *Assume that $g\colon E_f \to [0,1]$ is a maximal edge packing for G_f. Define the function $h\colon E \to [0,1]$ by*

$$h(e) = \begin{cases} f(e) + g(e)/2 & \text{if } e \in E_f, \\ f(e) & \text{otherwise.} \end{cases}$$

Now h is a maximal edge packing for G.

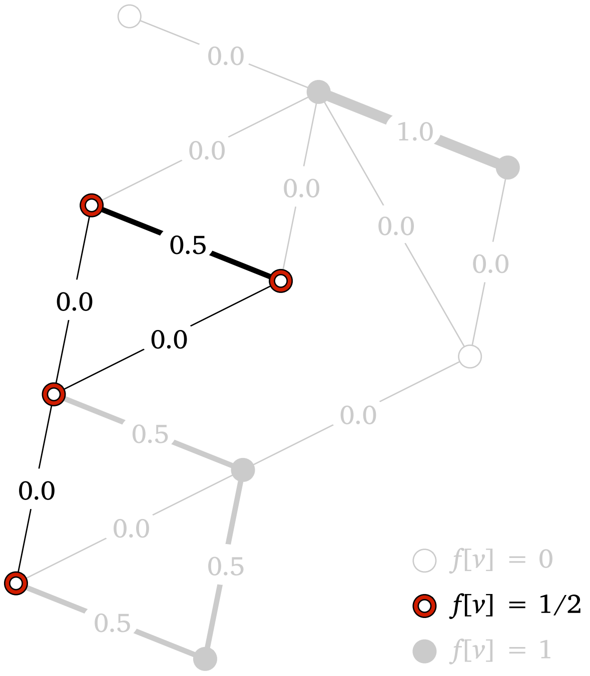


Figure 4.4: Subgraph G_f induced by the unsaturated edges; cf. Figure 4.3.

Proof. Let us first show that h is indeed an edge packing. Consider a node $v \in V$. If $v \notin V_f$, then v is not incident to any edge of E_f, and we have

$$h[v] = f[v] \le 1.$$

Otherwise $v \in V_f$, in which case $f[v] = 1/2$. We have

$$h[v] = f[v] + g[v]/2 = 1/2 + g[v]/2 \le 1/2 + 1/2 = 1.$$

Now let us prove that h is maximal. To this end, let $e \in E$. There are two cases:

(a) If $e \notin E_f$, then e is saturated by f in G. That is, there is an endpoint $v \in e$ with $f[v] = 1$, which implies $v \notin V_f$ and $h[v] = f[v] = 1$. Hence e is saturated by h in G.

(b) If $e \in E_f$, then e is saturated by g in G_f. That is, there is an endpoint $v \in e$ with $g[v] = 1$. Moreover, $v \in V_f$, which implies $f[v] = 1/2$. We have $h[v] = f[v] + g[v]/2 = 1/2 + 1/2 = 1$. Hence e is saturated by h in G.

In conclusion, h is a maximal edge packing for G. □

4.2.5 Finding Half-Saturating Edge Packings

Now we present algorithm HSEP that finds a half-saturating edge packing in any graph. It turns out that we are already familiar with all the key ingredients — in essence, algorithm HSEP uses the same idea as algorithm VC3 from Section 2.4.2.

Let $N = (V, P, p)$ be a port-numbered network. We construct a virtual port-numbered network $N' = (V', P', p')$ and a 2-colouring precisely as we did in Section 2.4.2. Let $G = (V, E)$ be the underlying graph of N, and let $G' = (V', E')$ be the underlying graph of N'. Recall that we used the symbols $v_1 \in V'$ and $v_2 \in V'$ to refer to the two virtual copies of a node $v \in V$.

Algorithm HSEP first simulates the execution of BMM on N' in order to find a maximal matching M' for G'. Given a maximal matching

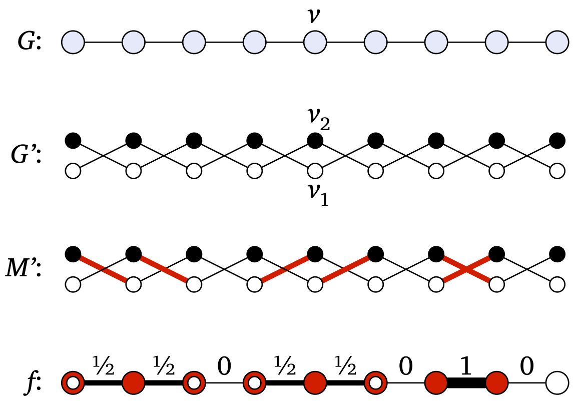


Figure 4.5: Algorithm HSEP. Note that f is a half-saturating edge packing for G, but it is not a maximal edge packing.

M', we construct a maximal edge packing $f'\colon E' \to [0,1]$ for G': we set $f'(e') = 1$ if $e' \in M'$ and $f'(e') = 0$ otherwise. Finally, we use the maximal edge packing f' to construct an edge packing $f : E \to [0,1]$ for G as follows:

$$f(\{u,v\}) = \frac{f'(\{u_1,v_2\}) + f'(\{u_2,v_1\})}{2}.$$

Algorithm HSEP outputs f and stops. See Figure 4.5 for an illustration.

Let us now prove that the output f is a half-saturating edge packing for G. It is straightforward to verify that

$$\begin{aligned} 2f[u] &= \sum_{v\colon \{u,v\}\in E} f'(\{u_1,v_2\}) + \sum_{v\colon \{u,v\}\in E} f'(\{u_2,v_1\}) \\ &= f'[u_1] + f'[u_2]. \end{aligned}$$

Now we can make the following observations; recall that f' is a maximal edge packing for G'.

(a) For each node $v \in V$, we have $f'[v_1] + f'[v_2] \le 1 + 1$ which implies $f[v] \le 1$.

(b) By construction, we have $f(e) \in \{0, 1/2, 1\}$.

(c) Assume that $f[v] = 0$, and let u be a neighbour of v in G. Then $f'[v_1] = f'[v_2] = 0$, i.e., neither v_1 nor v_2 are saturated. In graph G', node u_2 is a neighbour of v_1 and u_1 is a neighbour of v_2. As f' is maximal, both u_2 and u_1 have to be saturated. That is, $f'[u_2] = f'[u_1] = 1$, which implies $f[u] = 1$.

(d) Assume that $f[v] = 1/2$. Then one of the virtual copies of v is saturated; both cases are symmetric, so w.l.o.g. we will assume that $f'[v_1] = 1$ and $f'[v_2] = 0$. It follows that there is a neighbour u of v in G such that

$$f'(\{u_1, v_2\}) = 0,$$
$$f'(\{u_2, v_1\}) = 1.$$

By definition, we have $f'[u_2] = 1$. By the maximality of f', node u_1 has to be saturated, as v_2 is not saturated. In summary, $f'[u_2] = f'[u_1] = 1$, which implies $f[u] = 1$.

We conclude that f is a half-saturating edge packing for G. Hence algorithm HSEP works correctly. By Lemma 2.2 the running time of the algorithm is at most $2\Delta + 1$ rounds in a graph of maximum degree at most Δ.

4.2.6 Finding Maximal Edge Packings

Now we are ready to present algorithm MEP_Δ that finds a maximal edge packing h for any graph $G = (V, E)$ of maximum degree at most Δ. The algorithm has a recursive structure, and its running time is

$$T(\Delta) = \sum_{i=1}^{\Delta} 2(i+1) = \Delta(\Delta+3)$$

communication rounds.

Let us first assume that $\Delta \leq 1$. The case of $\Delta = 0$ is trivial, as there are no edges in the graph. For the case of $\Delta = 1$, algorithm MEP_1 returns the maximal edge packing

$$h\colon e \mapsto 1.$$

Clearly this can be done in $T(1)$ rounds.

Now assume that $\Delta > 1$, and assume that we have already defined $\mathsf{MEP}_{\Delta-1}$. Algorithm MEP_Δ proceeds as follows.

(a) We use $2\Delta + 1$ rounds to find a half-saturating edge packing f with algorithm HSEP. Now each node $v \in V$ knows $f(e)$ for each edge e incident to v; in particular, v knows the value $f[v]$.

(b) We use 1 round to exchange the values $f[v]$. Now each node v knows the value $f[u]$ for each neighbour u. In particular, node v knows which of its incident edges are saturated — put otherwise, v knows which of its incident edges are in the subgraph $G_f = (V_f, E_f)$.

(c) Next we have the recursive step. By Lemma 4.5, the maximum degree of G_f is at most $\Delta - 1$. Hence we can simulate the execution of $\mathsf{MEP}_{\Delta-1}$ in the subgraph $G_f = (V_f, E_f)$. After $T(\Delta - 1)$ rounds, algorithm $\mathsf{MEP}_{\Delta-1}$ outputs a maximal edge packing g for G_f.

(d) Now f is a half-saturating edge packing for G, and g is a maximal edge packing for the subgraph G_f. Each node knows the values of f and g for each incident edge. We use Lemma 4.6 to construct a maximal edge packing $h = f + g/2$ for G; this only requires local computation.

In summary, the algorithm takes $2\Delta + 1 + 1 + T(\Delta - 1) = T(\Delta)$ rounds; the correctness of the algorithm follows from Lemmas 4.5 and 4.6.

Now it is easy to design algorithm $\mathsf{VC2}_\Delta$ that finds a 2-approximation of a minimum vertex cover in any graph of maximum degree at most Δ: we first run MEP_Δ, and then each node outputs 1 if it is saturated. The correctness of the algorithm follows from Lemma 4.3.

4.3 Exercises

Exercise 4.1 (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 $\mathscr{F}$. Show that it is not possible to find a $(\Delta + 1 - \epsilon)$-approximation.

Hint: For the lower bound, use the result of Exercise 3.6.

Exercise 4.2 (implementation). In Exercise 2.6, we implemented a simulator and some simple distributed algorithms, including algorithm VC3. Now implement algorithm VC2 from Section 4.2, and compare its performance with VC3. Try out both algorithms with the instance from Exercise 2.5.

Exercise 4.3 (general case). Design a distributed algorithm that finds a 2-approximation of a minimum vertex cover in any graph. In particular, you cannot assume that there is a known upper bound Δ on the maximum degree of the graph.

Hint: The edge packing algorithm of Section 4.2.6 has the following high-level structure: run algorithm HSEP, remove saturated edges, and repeat. A node can stop as soon as all incident edges become saturated. In essence, we have a situation that we already studied in Exercise 2.7: our algorithm consist of several phases, and the output of phase i is needed as the input of phase $i + 1$.

Exercise 4.4 (centralised algorithms). In this chapter, we have seen an efficient distributed algorithm that finds a 2-approximation of a minimum-size vertex cover. What is known about efficient *centralised* approximation algorithms for the vertex cover problem?

* * *

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.

Exercise 4.5 (general case). Design a distributed algorithm that finds a 4-approximation of MEDS.

Hint: Use the idea of Section 2.4.2. Show that the edge set $M \subseteq E$ defined in (2.1) 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.

Exercise 4.6 (2-regular). Show that it is possible to find a 3-approximation of MEDS in 2-regular graphs, in constant time. Show that it is not possible to find a 2.999-approximation of MEDS in 2-regular graphs.

Exercise 4.7 (4-regular, upper bound). Show that it is possible to find a 3.5-approximation of MEDS in 4-regular graphs, in constant time.

Hint: 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 $|V|$.

Exercise 4.8 (4-regular, lower bound). Show that it is not possible to find a 3.499-approximation of MEDS in 4-regular graphs.

Hint: Use the construction of Exercise 3.3.

Exercise 4.9 (3-regular, lower bound). Show that it is not possible to find a 2.499-approximation of MEDS in 3-regular graphs.

Hint: Use the construction of Exercise 3.1.

Exercise 4.10 (3-regular, upper bound). Show that it is possible to find a 2.5-approximation of MEDS in 3-regular graphs, in constant time.

Hint: 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.

Chapter 5

Unique Identifiers

5.1 Introduction

So far we have studied deterministic distributed algorithms in port-numbered networks. Now we will introduce another model of distributed computing: deterministic distributed algorithms in *networks with unique identifiers*.

In the model of unique identifiers, we assume that we are given a node labelling $\mathrm{id}\colon V \to \mathbb{N}$ such that each node v has a unique label $\mathrm{id}(v)$; see Figure 5.1 for an example. We will assume that the labels are reasonably small — in an n-node network, the labels are $O(\log n)$-bit integers.

As such, the model does not seem to be a major deviation from what we have studied so far. We have already encountered various extensions of the port-numbering model — for example, we have studied distributed algorithms that assume that we are given a colouring of the nodes or an orientation of the edges.

However, once we have unique identifiers, we can no longer apply techniques based on covering graphs (see Section 3.2) to prove impossibility results. It turns out that *any* computable graph problem on connected graphs can be solved if we are given unique identifiers. Hence we are no longer interested in what can be solved; the key question is what can be solved *fast*.

5.2 Networks 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

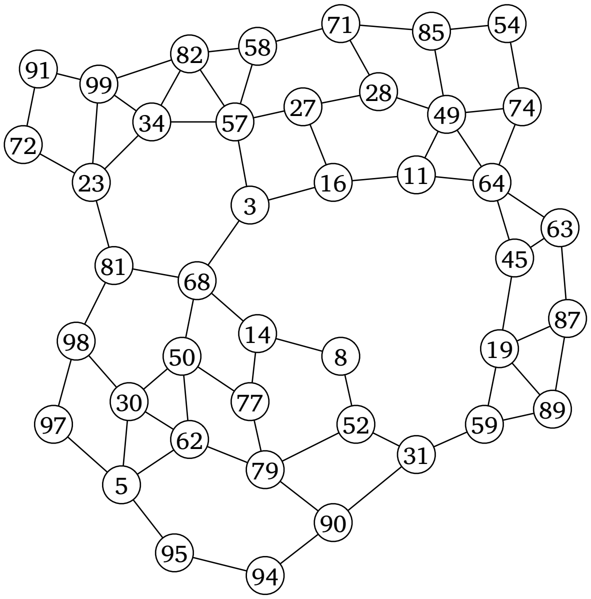


Figure 5.1: A network with unique identifiers.

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 (supposed to be) globally unique, and the numbers can be interpreted as natural numbers. Moreover, the numbers are relatively small but not as small as possible: in a network with millions of devices we may have allocated a space of billions of possible identifiers. In particular, there is no guarantee that a device with identifier "1" exists in the network at any given time.

We will now give the formal definition that aims at capturing these properties of real-world networks.

5.2.1 Definitions

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

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

That is, each node $v \in V$ is labelled with a unique integer, and the labels are assumed to be relatively small (in comparison with the number of nodes in network N).

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

5.2.2 Nodes and Their Names

For the sake of convenience, when we discuss networks with unique identifiers, we will assume that

$$v = \mathrm{id}(v) \text{ for all } v \in V.$$

Put otherwise, we assume that the set V is a subset of natural numbers, and $\max V \le |V|^c$.

5.2.3 Gathering Everything

In the model of unique identifiers, if the underlying graph $G = (V, E)$ is connected, all nodes can learn everything about G in time $O(\text{diam}(G))$. 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

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

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

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.

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

(c) More generally, if G_v is the connected component of G that contains v, and $r \geq \text{diam}(G_v) + 1$, 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:

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

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

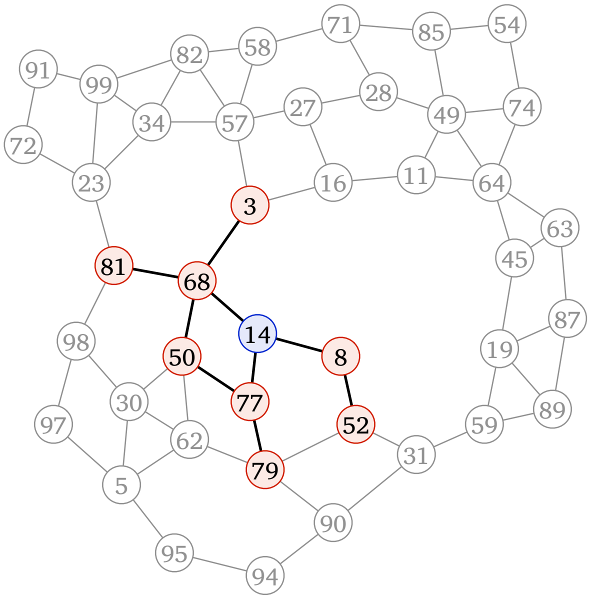


Figure 5.2: Subgraph $G(v,r)$ defined in (5.3), for $v = 14$ and $r = 2$.

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

5.2.4 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 model of unique identifiers. 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 5.2.3.

(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 $O(\operatorname{diam}(G))$.

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.

5.2.5 Focus on Complexity

The above discussion highlights the striking difference between the port-numbering model and the model of unique identifiers. While we saw in Section 3.2 plenty of examples of seemingly simple graph problems that cannot be solved at all in the port-numbering model, we have learned that with the help of unique identifiers all computable graph problems become solvable.

Hence our focus shifts from computability to computational complexity. While it is trivial to determine if a problem can be solved in the model of unique identifiers, 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 $\operatorname{diam}(G)$. One such problem is graph colouring.

5.3 Graph Colouring

Let $G = (V, E)$ be a graph with unique identifiers. We will use the shorthand notation $\chi = |V|^c$, that is, the unique identifiers are integers from $\{1, 2, \ldots, \chi\}$.

The unique identifiers form a proper vertex colouring with χ colours: certainly adjacent nodes have distinct identifiers if the identifiers are globally unique. Hence, in a sense, we have already solved the graph colouring problem — however, the number of colours χ is far too large for our purposes.

Our focus is therefore on *colour reduction*: given a graph colouring $f : V \to \{1, 2, \ldots, x\}$ with a large number x of colours, the goal is to find a new graph colouring $g : V \to \{1, 2, \ldots, y\}$ with a smaller number $y < x$ of colours.

5.3.1 Greedy Colour Reduction

Let $x \in \mathbb{N}$. There is a simple algorithm Greedy that reduces the number of colours from x to

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

where Δ is the maximum degree of the graph. The running time of the algorithm is one communication round.

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

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

be the set of *free colours* in the neighbourhood of v.

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

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

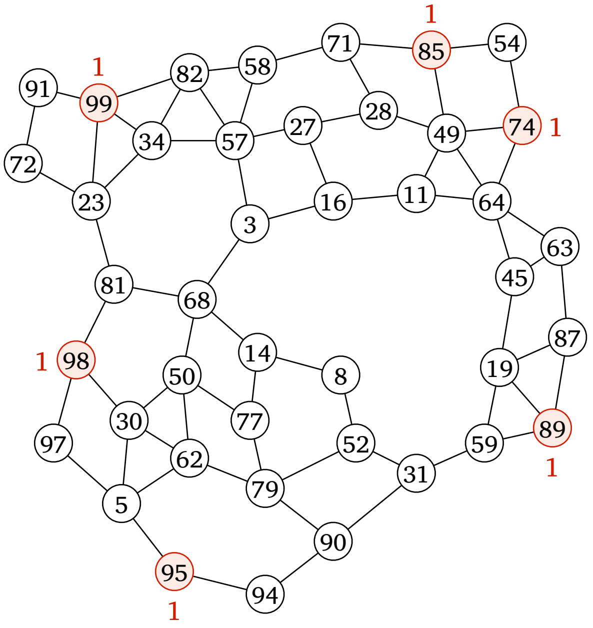


Figure 5.3: Greedy colour reduction. The active nodes have been highlighted. In this example, each active node can choose 1 as its new colour. Note that in the original colouring f, the largest colour was 99, while in the new colouring, the largest colour is strictly smaller than 99 — we successfully reduced the number of colours in the graph.

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

Lemma 5.1. *Algorithm Greedy 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

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

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

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

There is at least one value $i \in \{1, 2, \dots, |C(v)| + 1\}$ 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 $g(u) \neq g(v)$. □

A key observation in understanding the algorithm 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.

Note that algorithm Greedy does not need to know the number of colours x or the maximum degree Δ; we only used them in the analysis. We can simply take any graph, blindly apply algorithm Greedy, 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 Greedy 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.

In principle, we could use this strategy in the model of unique identifiers to find a $(\Delta + 1)$-colouring of any graph. However, such an algorithm would be extremely slow. In the worst case, we may have a long path of nodes, with increasing identifiers (colours) along the path, and in such a graph the running time of the greedy strategy would be linear in $|V|$: in each iteration, only one of the nodes is a local maximum.

In the next sections, we will develop an algorithm that is much faster — at least in low-degree graphs.

5.3.2 Directed Pseudoforests

We will first study fast colour reduction algorithms in a seemingly simple special case: we are given a pseudoforest with a particular orientation. Once we have solved the special case, we turn our attention to the more general case of colouring bounded-degree graphs.

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

5.3.3 Greedy Colouring in Pseudoforests

We will soon see that we can do colour reduction in directed pseudoforests quickly. However, let us first show that we can find a colouring

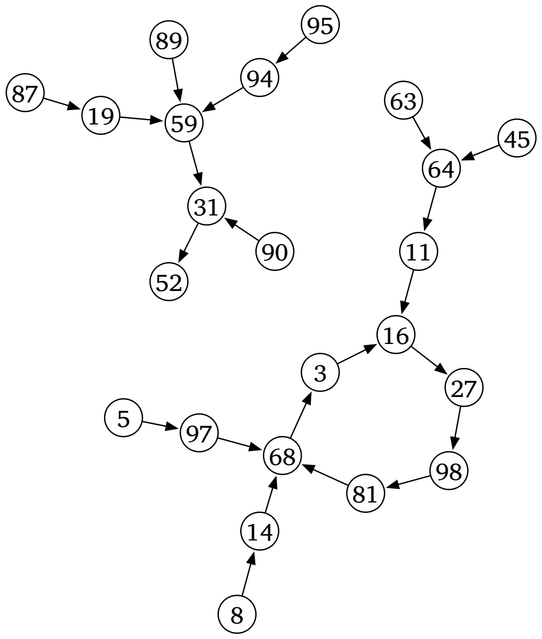


Figure 5.4: A directed pseudoforest with a colouring f.

with a very small number of colours with a modified version of algorithm Greedy.

Let $G = (V, E)$ be a directed pseudoforest, and let

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

be a proper vertex colouring of G, 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.

First, for each node $v \in V$, define $s(v)$ as follows:

(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 $s(v) = 2$.

By construction, we have $s(v) \neq f(v)$. Note that we can compute the values $s(v)$ for all nodes $v \in V$ with a simple distributed algorithm in one communication round.

We will now prove that the values $s(v)$ form a proper x-colouring of G. Moreover, we 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 $s(v) \in \{1, 2, \dots, x\}$.

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

$$s(u) = f(v) \neq s(v). \qquad \square$$

Lemma 5.3. *Define*

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

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

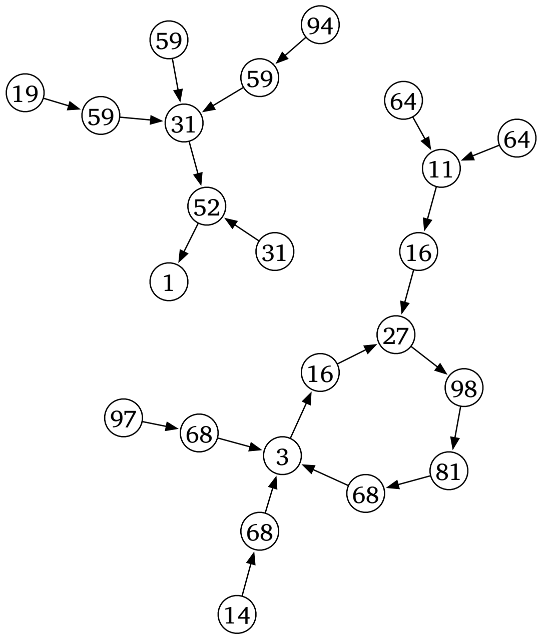


Figure 5.5: A directed pseudoforest with colouring s; compare with Figure 5.4. In colouring s, all predecessors of a node have the same colour; hence each node is adjacent to nodes of only two different colours.

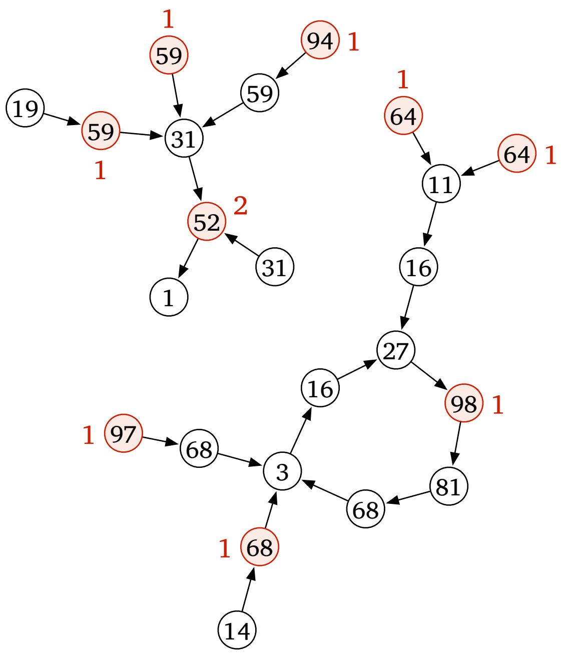


Figure 5.6: Algorithm Greedy applied to a directed pseudoforest with colouring s. The active nodes are highlighted.

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 we apply algorithm Greedy to colouring s; see Figure 5.6. Observe that 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 $x - 1$.

Let us summarise the above observations. 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:

(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 Greedy (Figure 5.6).

In particular, we can reduce the number of colours from any number $x \geq 3$ to 3 in $2(x-3)$ rounds by iterating the above steps.

Figure 5.7 demonstrates that the additional step of constructing colouring s is necessary.

5.3.4 Fast Colouring in Pseudoforests

So far we have only seen algorithms that reduce the number of colours by one in each iteration. This is by far too slow if, for example, we are given a colouring that is formed by 128-bit IPv6 addresses. In this section we will present an algorithm that is *much* faster.

In particular, 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.

Before presenting algorithm DPBit, we will give a practical example of its performance. Assume that the initial colouring is derived from 128-bit unique identifiers, that is, the number of colours is 2^{128}. If we iterate algorithm DPBit, we can reduce the number of colours as

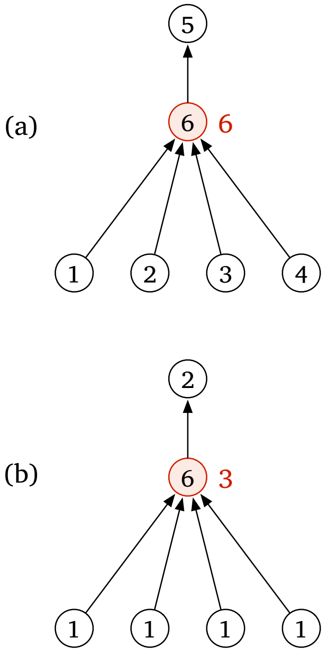


Figure 5.7: (a) If we tried to apply algorithm Greedy directly in any given colouring f, the active nodes would not be able to pick new colours from the set $\{1, 2, 3\}$. (b) In colouring s, algorithm Greedy will always find a new colour from the set $\{1, 2, 3\}$.

follows:

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

That is, given a 2^{128}-colouring, in only 4 communication rounds, we can find a 6-colouring. We cannot reduce the number of colours below 6 with DPBit; however, once we have reached such a low number of colours, we can resort to DPGreedy, which is able to reduce the number of colours from 6 to 3 in 6 communication rounds. In summary, we can reduce the number of colours from 2^{128} to 3 in only $4+6=10$ rounds, in any directed pseudoforest.

Let us now present algorithm DPBit. We assume that we are given a proper vertex colouring

$$f : V \rightarrow \{1,2,\dots,2^x\}$$

of a directed pseudoforest $G=(V,E)$. We will use the values $s(v)$ defined in Section 5.3.3 — 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 $j\in\{1,2,\dots,2^x\}$ 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

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

If $i\in\{0,1,\dots,x-1\}$, 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 $0 \leq b(v) \leq 1$.

The key observation is that the pairs $\bigl(i(v), b(v)\bigr)$ 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, $b(u) \neq b(v)$. □

We can now encode the pair $\bigl(i(v), b(v)\bigr)$ as a colour

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

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

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. Communication is only needed in order to discover the value $s(v)$ for each node v; the derivation of the values $i(v)$, $b(v)$, and $g(v)$ only needs local computation.

5.3.5 Fast Colouring in General Graphs

In this section, we will present algorithm Colour that reduces the number of colours from any number x to $\Delta+1$ in any graph of maximum degree at most Δ much faster than an iterated application of algorithm Greedy. Throughout this section, we will assume that the values of x and Δ are known to all nodes.

Let A be an algorithm that reduces the number of colours in a directed pseudoforest from x to 3 in time $T(x)$. For example, we can let A be the combination of the iterated DPBit (reduces the number of colours from any x to 6) followed by the iterated DPGreedy (reduces the number of colours from 6 to 3). As we will see in Exercise 5.4, the running time of A is then $T(x) = O(\log^* x)$.

Algorithm Colour uses A as a subroutine, and the running time of Colour will be $O(\Delta^2) + T(x)$. For example, with the above choice of A, the running time of Colour is $O(\Delta^2 + \log^* x)$.

Let $G = (V, E)$ be a graph of maximum degree at most Δ, and let $f : V \to \{1, 2, \dots, x\}$ be an x-colouring of G. Let N be a port-numbered network with G as the underlying graph. Algorithm Colour constructs a $(\Delta+1)$-colouring g of G as follows.

Preliminaries. For each node v and each port number i, node v sends the pair $(f(v), i)$ to port i. This way a node u learns the following information about each node v that is adjacent to u: what is the old colour 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.

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 $f(u) < f(v)$. That is, we use the old colours of the nodes to orient the edges from a smaller colour to a larger colour; see Figure 5.8.

In the distributed algorithm, each node only needs to know the orientation of its incident edges. This step requires zero communication rounds.

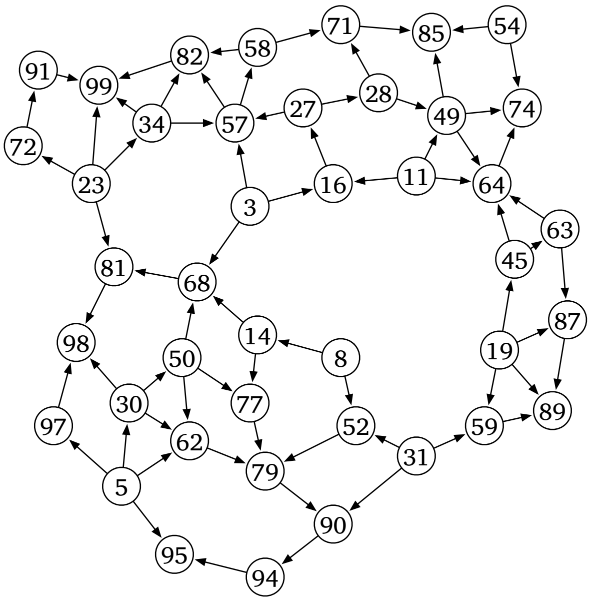


Figure 5.8: Orientation G' derived from the old colours — in this example, the old colours were unique identifiers.

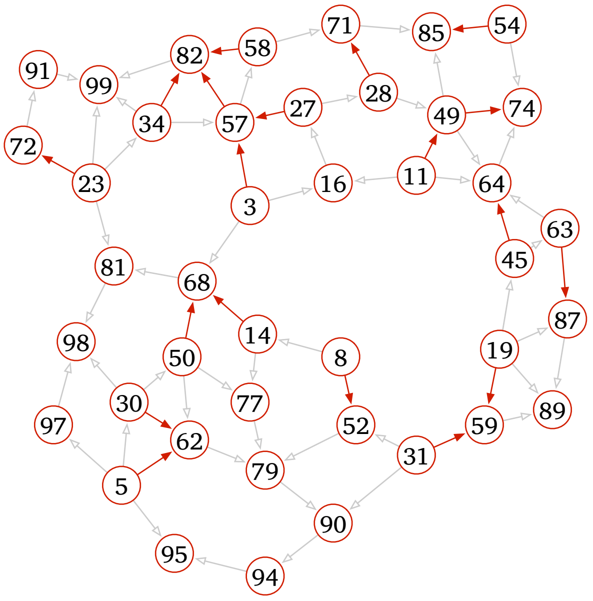


Figure 5.9: Subgraph G_i of G'. Each node has outdegree at most one.

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 5.9.

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*. Function f is an x-colouring of G_i for all i.

In the distributed algorithm, each node only needs to know which of its incident edges are in which subset E_i. This step requires zero communication rounds.

Parallel Colouring of Pseudoforests. For each i, we use algorithm A to construct a 3-colouring g_i of G_i.

In the distributed algorithm, each node $v \in V$ needs to know the value $g_i(v)$ for each i. This step takes only $T(x)$ 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.

Merging Colourings. For each $j = 0, 1, \dots, \Delta$, 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 $g'_0, g'_1, \dots, g'_\Delta$ 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 5.10. 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 Greedy 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 $O(\Delta)$.

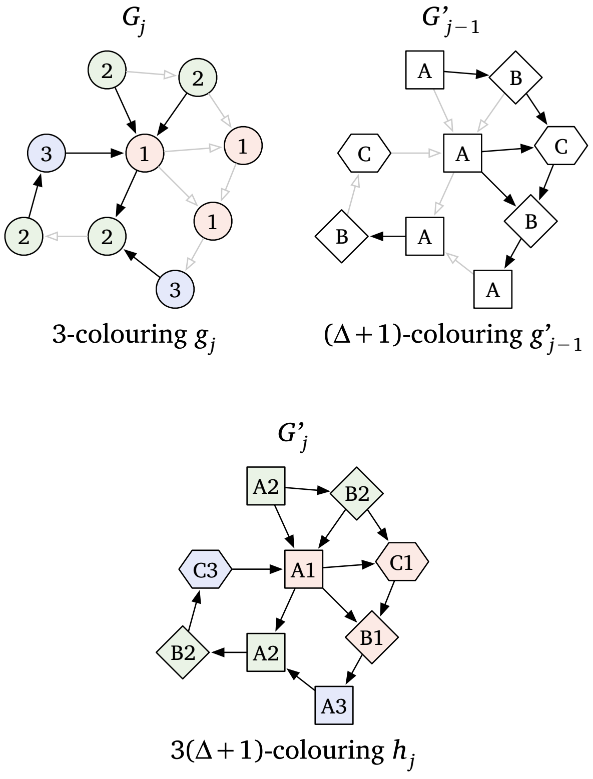

G_j
3-colouring g_j
G'_{j-1}
$(\Delta+1)$-colouring g'_{j-1}
G'_j
$3(\Delta+1)$-colouring h_j
A1
A2
A3
B1
B2
C1
C3

Figure 5.10: 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.

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.

5.4 Exercises

Exercise 5.1 (counting). The *counting problem* Π is defined as follows: if $N = (V, P, p)$ is a port-numbered network, then $g \in \Pi(N)$ if and only if $g(v) = |V|$ for all $v \in V$. That is, in the counting problem each node has to output the value $|V|$, i.e., it has to indicate how many nodes there are in the network.

Let $\mathscr{F}$ consist of all cycle graphs, and let $\mathscr{F}'$ consist of all graphs of maximum degree 2.

(a) Prove that the counting problem cannot be solved on $\mathscr{F}$ in the port-numbering model.

(b) Design an algorithm that solves the counting problem on $\mathscr{F}$ in the model of unique identifiers in time $O(|V|)$. Present the algorithm in a formally precise manner, using the definitions of Sections 2.2 and 2.3.

(c) Prove that the counting problem cannot be solved in time $o(|V|)$ on $\mathscr{F}$ in the model of unique identifiers.

(d) Prove that the counting problem cannot be solved on $\mathscr{F}'$ in the model of unique identifiers.

Exercise 5.2 (leader election). The *leader election problem* Π is defined as follows: if $N = (V, P, p)$ is a port-numbered network, then $g \in \Pi(N)$ if and only if there is precisely one node $u \in V$ such that

$$g(v) = \begin{cases} 1 & \text{if } v = u, \\ 0 & \text{otherwise.} \end{cases}$$

Let $\mathscr{F}$ consist of all connected graphs.

(a) Prove that the leader election problem cannot be solved on $\mathscr{F}$ in the port-numbering model.

(b) Design an algorithm that solves the leader election problem on $\mathscr{F}$ in the model of unique identifiers.

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: Adapt the basic idea of algorithm Greedy — 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.

Exercise 5.4 (log-star)**.** The *iterated logarithm* of x, in notation $\log^* x$, is defined recursively as follows:

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

This is a function that grows extremely slowly; for example

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

Prove that algorithm DPBit can be used to reduce the number of colours from x to 6 in $\log^* x$ communication rounds in any directed pseudoforest, for any $x \geq 6$. You can assume that the value of x is known in advance.

Hint: Consider the following cases separately:

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

In case (iii), prove that after $\log^*(x) - 3$ iterations of DPBit, the number of colours is at most 64.

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 5.4). 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

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

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 $g\colon V \to \{1, 2, \ldots, 2x\}$ 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 5.4?

Exercise 5.7 (cycles)**.** 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 model of unique identifiers.

Hint: 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.

Exercise 5.8 (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 model of unique identifiers.

(a) Maximal independent set.

(b) Maximal matching.

(c) Edge colouring with $O(\Delta)$ colours.

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

Exercise 5.9 (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:

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

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 model of unique identifiers.

Hint: 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 $\operatorname{dist}_G(u, v) \leq 2$. Prove that the maximum degree of G^2 is $O(\Delta^2)$. Simulate a fast graph colouring algorithm on G^2.

Exercise 5.10 (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 model of unique identifiers.

Hint: 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 $\operatorname{ball}_G(v, 1)$. For more details, see a textbook on approximation algorithms, e.g., Vazirani [28].

Second, show that you can *simulate* the centralised greedy algorithm in a distributed setting. Use the algorithm of Exercise 5.9 to construct a distance-2 colouring. Prove that the following strategy is a faithful simulation of the centralised greedy algorithm:

- For each possible value $i = \Delta + 1, \Delta, \dots, 2, 1$:
  - For each colour $j = 1, 2, \dots, O(\Delta^2)$:
    - 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.

Chapter 6

Ramsey Theory

6.1 Introduction

As a running example in this chapter, we will use the following task: find a 3-colouring of a directed cycle in the model of unique identifiers.

In a *directed cycle*, we assume that we are given a graph $G = (V, E)$ that is an orientation of a cycle graph. In particular, we assume that each node $v \in V$ has

$$\text{outdegree}_G(v) = \text{indegree}_G(v) = 1,$$

that is, there is precisely one incoming edge and one outgoing edge. Without loss of generality, we will assume that the incoming edge is connected to port number 1 and the outgoing edge is connected to port number 2 in each node — if this was not the case, each node could renumber its ports locally. See Figure 6.1 for an illustration.

Clearly, directed cycles are a special case of directed pseudoforests, and we already know how to find a 3-colouring of a directed pseudo-

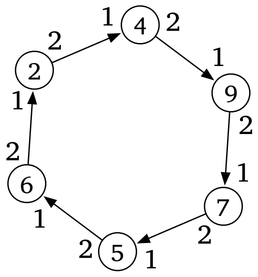


Figure 6.1: A directed cycle with unique identifiers.

forest in the model of unique identifiers. Indeed, there are several possible strategies.

- The greedy algorithm is simple but slow; in the case of directed cycles, it requires $\Omega(n)$ rounds in the worst case.
- Algorithm DPBit is much faster — as we saw in Exercise 5.4, algorithm DPBit finds a 6-colouring in $O(\log^* n)$ rounds, and we can then use the greedy algorithm to reduce the number of colours from 6 to 3 in constant time.
- Algorithm DPBit is in no way unique, and there are many alternative strategies that we can use to 3-colour a directed pseudoforest. Exercises 5.5 and 5.6 explore some possible ideas.

Moreover, directed cycles are a simple special case of directed pseudoforests, and whenever we have an algorithm that finds a 3-colouring in any directed pseudoforest, we can construct a slightly faster algorithm that finds a 3-colouring in directed cycles — for example, we can easily speed up algorithm DPGreedy by a factor of two in directed cycles, as the construction of intermediate colouring s becomes unnecessary.

However, no matter what combination of algorithm ideas we use, it appears that the worst-case running time of the algorithm is always $\Omega(\log^* n)$. That is, the running time *slightly* increases as the number of nodes n increases.

In this chapter we will prove that this is indeed necessary. We show that there is no $O(1)$-time algorithm that 3-colours any directed cycle in the model of unique identifiers. Our proof uses Ramsey's theorem, which is a fundamental result in combinatorics.

6.2 Ramsey's Theorem

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

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

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

Figure 6.2: 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.

for the collection of all k-subsets of Y.

6.2.1 Monochromatic Subsets

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 $A, B \in X^{(k)}$,

(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 6.2 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.

6.2.2 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 6.1 (Ramsey's theorem). *Numbers $R_c(n;k)$ are finite for all positive integers c, n, and k.*

We will prove Theorem 6.1 in Section 6.2.4; let us first have a look at an application.

6.2.3 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 6.2. *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 6.1, 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. □

6.2.4 Proof

Let us now prove Theorem 6.1. 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 6.3: $R_c(n;1)$ is finite for all n.

(b) Corollary 6.7: 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 6.5 — if $R_c(n;k-1)$ is finite for all n, then $\bar{R}_c(n;k)$ is finite for all n.

 (ii) Lemma 6.6 — 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 6.3. *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 $R_c(n;1) \le N$.

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 $\{1,2,\dots,c\}$, 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 6.4. *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 $W \subseteq Y$.

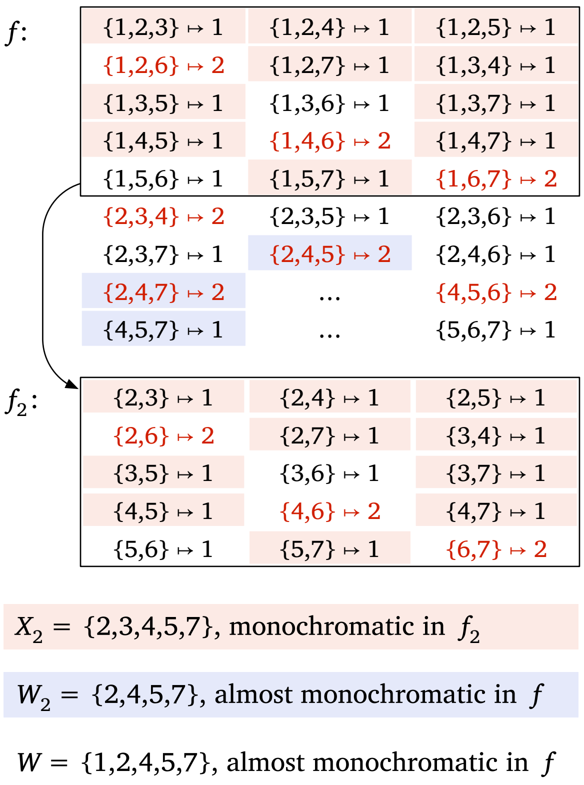
To this end, let $Y_2 = \{2,3,\dots,N\}$, and define a c-labelling f_2 of $Y_2^{(k-1)}$ as follows; see Figure 6.3 for an illustration:

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

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.



f:

$\{1,2,3\} \mapsto 1$	$\{1,2,4\} \mapsto 1$	$\{1,2,5\} \mapsto 1$
$\{1,2,6\} \mapsto 2$	$\{1,2,7\} \mapsto 1$	$\{1,3,4\} \mapsto 1$
$\{1,3,5\} \mapsto 1$	$\{1,3,6\} \mapsto 1$	$\{1,3,7\} \mapsto 1$
$\{1,4,5\} \mapsto 1$	$\{1,4,6\} \mapsto 2$	$\{1,4,7\} \mapsto 1$
$\{1,5,6\} \mapsto 1$	$\{1,5,7\} \mapsto 1$	$\{1,6,7\} \mapsto 2$
$\{2,3,4\} \mapsto 2$	$\{2,3,5\} \mapsto 1$	$\{2,3,6\} \mapsto 1$
$\{2,3,7\} \mapsto 1$	$\{2,4,5\} \mapsto 2$	$\{2,4,6\} \mapsto 1$
$\{2,4,7\} \mapsto 2$	$\dots$	$\{4,5,6\} \mapsto 2$
$\{4,5,7\} \mapsto 1$	$\dots$	$\{5,6,7\} \mapsto 1$

f_2:

$\{2,3\} \mapsto 1$	$\{2,4\} \mapsto 1$	$\{2,5\} \mapsto 1$
$\{2,6\} \mapsto 2$	$\{2,7\} \mapsto 1$	$\{3,4\} \mapsto 1$
$\{3,5\} \mapsto 1$	$\{3,6\} \mapsto 1$	$\{3,7\} \mapsto 1$
$\{4,5\} \mapsto 1$	$\{4,6\} \mapsto 2$	$\{4,7\} \mapsto 1$
$\{5,6\} \mapsto 1$	$\{5,7\} \mapsto 1$	$\{6,7\} \mapsto 2$

$X_2 = \{2,3,4,5,7\}$, 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 6.3: The proof of Lemma 6.4, for the case of $c = 2$, $k = 3$, and $n = 5$, assuming completely fictional values $M = 5$ and $N = 7$.

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 $f(A) = f_2(A_2) = f_2(B_2) = f(B)$.

(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 6.5. *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 $n \le k$ 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 6.4, which implies that $\bar{R}_c(n;k)$ is finite. □

Lemma 6.6. *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 6.3, M is finite. By assumption, $\bar{R}_c(M;k)$ is also finite. We will show that

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

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

Figure 6.4: The proof of Lemma 6.6. 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.

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

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

for each k-subset $A \subseteq X$; see Figure 6.4. 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 6.5 and 6.6 have the following corollary.

Corollary 6.7. *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 6.3, and the inductive step is Corollary 6.7.

6.3 Speed Limits

We will now use Ramsey's theorem to prove that directed cycles cannot be 3-coloured in constant time.

Theorem 6.8. *Assume that A is a distributed algorithm for the model of unique identifiers. 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 $g\colon V \to \{1, 2, 3\}$. Then there exists a directed cycle G such that if we execute A on G, the output of A is not a proper vertex colouring of G.*

To prove Theorem 6.8, 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 $V = \{1, 2, \dots, N\}$. 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.

6.3.1 Subsets and Cycles

For each subset $X \subseteq V$, we define a directed cycle $G_X = (V, E_X)$ as follows; see Figure 6.5. Let $\ell = |X|$. Label the nodes by $x_1, x_2, \dots, x_N$ such that

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

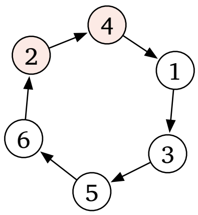


Figure 6.5: Construction of G_X. Here $N = 6$ and $X = \{2, 4\}$.

Then choose the edges

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

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.

6.3.2 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 6.6 for an illustration.

6.3.3 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 in f. Let us label the nodes of X by

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

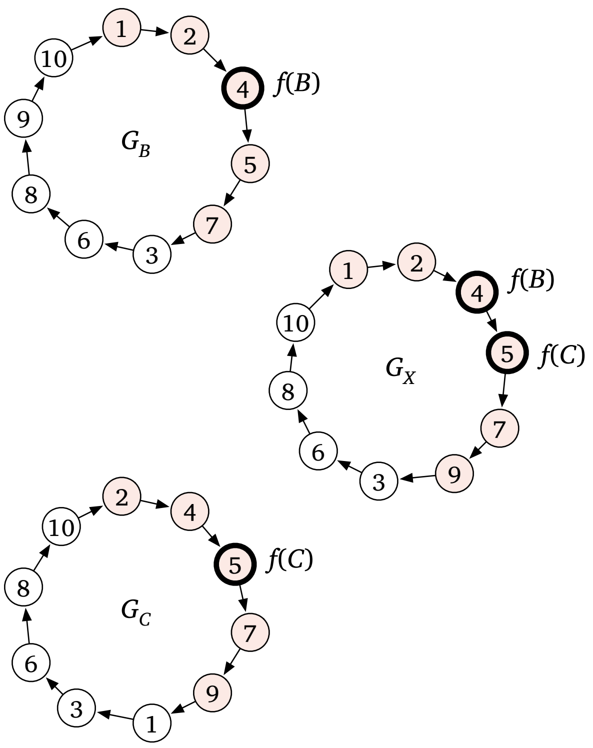


Figure 6.6: In this example, $N = 10$ and $T = 2$. Let $B = \{1, 2, 4, 5, 7\}$, $C = \{2, 4, 5, 7, 9\}$, and $X = \{1, 2, 4, 5, 7, 9\}$. The label $f(B)$ is defined as follows: we construct G_B, execute algorithm A, and take the output of the internal node $i(B) = 4$. Similarly, the label $f(C)$ is the 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 $f(B) = f(C)$.

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

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

See Figure 6.6 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 $x_0, x_1, \dots, x_{k-1}$ in this order. Hence by Theorem 3.6, 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.

6.4 Exercises

Exercise 6.1. Prove that $R_c(n; 1) = c(n-1) + 1$.

Hint: The proof of Lemma 6.3 shows that

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

You need to show that

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

Exercise 6.2. Prove that $R_2(3; 2) = 6$.

Exercise 6.3. Prove that it is not possible to find a proper vertex colouring with at most 100 colours in any directed cycle in constant time.

Hint: You can modify the proof of Theorem 6.8. Alternatively, you can show that if you could find a 100-colouring in constant time, you could also find a 3-colouring in constant time.

Exercise 6.4. Prove that it is not possible to find a maximal independent set in any directed cycle in constant time.

Hint: Assume that algorithm A finds an independent set in time T in any directed cycle. Follow the basic idea of the proof of Theorem 6.8. Choose $n = 2T + 3$, $k = 2T + 1$, and $c = 2$. Show that you can construct a cycle in which a node and *both* of its neighbours produce the same output. Argue that if the output is a valid independent set, it cannot be a maximal independent set.

Exercise 6.5. Prove that it is not possible to find a maximal matching in any directed cycle in constant time.

Exercise 6.6. Prove that it is not possible to find a 100-approximation of a maximum independent set in any directed cycle in constant time.

Hint: 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.

Chapter 7

What Next?

7.1 Other Stuff Exists

Distributed computing is a vast topic. We conclude this course by mentioning perspectives that we have not covered; we also provide pointers to more in-depth information.

7.1.1 Models of Computing

Many models of distributed computing can be seen as extensions of the models that we have studied. The following extensions are familiar from the context of classical computational complexity and Turing machines.

Randomised algorithms. Each node has access to a stream of random bits. A good example is Luby's [17] randomised algorithm for finding a maximal independent set — the algorithm uses the random bits for symmetry breaking.

Nondeterministic algorithms. It is sufficient that there exists a *proof* that can be verified efficiently in a distributed setting; we do not need to construct the proof. This research direction was introduced by Korman et al. [15].

7.1.2 Variants

There are many variants of the model that we described.

Asynchronous systems. Computers do not necessarily operate in a synchronous manner. In particular, the propagation delays of the messages may vary.

Message passing vs. 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.

The above aspects were irrelevant for our purposes, as we were only interested in the number of communication rounds; for example, asynchronous systems can be "synchronised" efficiently [5]. However, if we consider other complexity measures or fault tolerance, such details become important.

Our model of computing is primarily intended to capture the specifics of *wired* networks — communication links can be seen as cables that connect the computers. There are also numerous models that are designed with *wireless* networks in mind. A simple graph is no longer an appropriate model: a single radio transmission can be received by multiple nodes, and multiple simultaneous radio transmissions can interfere with each other. Radio propagation is closely connected with physical distances; hence in the context of wireless networks one often makes assumptions about *physical locations* of the nodes.

7.1.3 Complexity Measures

For us, the main complexity measure has been the number of synchronous communication rounds. Naturally, other possibilities exist.

Space. How many bits of memory do we need per node?

Number of messages. How many messages do we need to send in total?

Message size. We did not limit the size of a message. However, it is common to assume that the size of each message is $O(\log n)$ bits; how many communication rounds do we need in that case?

7.1.4 Fault Tolerance and Dynamics

Fault tolerance in general is an important topic in any large-scale distributed system. In the theory of distributed computing, fault tolerance has been studied from many different and complementary perspectives, of which we mention three representative examples.

Dynamic networks. Nodes can join and leave; edges can be removed and added. The system is expected to correct the output quickly after each change.

Byzantine failures. A fraction of nodes can be malicious and they may try to actively disturb the algorithm. Nevertheless, non-malicious nodes must be able to produce a correct output.

Self-stabilising systems. The initial state of each node can be arbitrary — an adversary may have corrupted the memory of each node. Nevertheless, the system must eventually recover and produce a correct output. Note that a self-stabilising system can never stop; all nodes have to keep communicating with each other indefinitely. See Dolev's [11] textbook for more information.

7.1.5 Problems

In this course we have studied *input/output problems*: we are given an input, we expect the system to do some computation, and eventually the system has to produce a correct output.

We assumed that the input is equal to the structure of the communication graph. This is not the only possibility: in general, one can solve arbitrary input/output problems in a distributed manner.

However, there are also many problems that are *not* input/output problems. In the context of distributed algorithms, there are also problems that are related to *controlling* an autonomous entity. Often we will use the metaphor of robot navigation: the graph is a map of an environment, and we need to control "robots" that navigate in the graph — however, instead of a physical robot, we can equally well study a logical entity such as a data packet or a token that is routed throughout a network. Some examples of robot navigation tasks include the following.

Graph exploration. A robot needs to visit all nodes of a graph.

Rendezvous. There are two robots who need to meet each other at a single node.

7.2 Further Reading

Nancy Lynch's textbook [18] provides an excellent overview of the field of distributed algorithms. Diestel's book [10] is a good source for graph-theoretic background, and Vazirani's book [28] provides further information on approximation algorithms from the perspective of non-distributed computing.

For more online material on distributed algorithms, see the following web page:

Principles of Distributed Computing,
Distributed Computing Group, ETH Zurich

http://dcg.ethz.ch/lectures/podc_allstars/

7.3 Bibliographic Notes

Many parts of this course have been directly influenced by numerous papers and textbooks; here is a brief summary of the key references.

Graph-theoretic Foundations. The connection between minimum maximal matchings and minimum edge dominating sets (Exercise 1.5) is due to Allan and Laskar [1] and Yannakakis and Gavril [30], and the connection between maximal edge packings and approximations of vertex covers (Lemma 4.3) was identified by Bar-Yehuda and Even [6]. The connection between maximal matchings and approximations of vertex covers (Exercise 1.3) is commonly attributed to Gavril and Yannakakis (see, e.g., Papadimitriou and Steiglitz [22]). Exercise 1.9 is a 120-year-old result due to Petersen [23]. The definition of a weak colouring is from Naor and Stockmeyer [19]. Ramsey's theorem dates back to 1930s [26]; our proof follows Nešetřil [20], and the notation is from Radziszowski [25].

Model of Computing. The model of computing that we use throughout this course — running time equals the number of synchronous communication rounds — is from Linial's [16] seminal paper, while the concept of a port numbering is from Angluin's [2] work.

Algorithms. Algorithm DPBit is based on the idea originally introduced by Cole and Vishkin [8] and further refined by Goldberg et al. [13]. The idea of algorithm DPSet is from Naor and Stockmeyer [19]. Algorithm Colour is from Goldberg et al. [13] and Panconesi and Rizzi [21]. Algorithm BMM is due to Hańćkowiak et al. [14]. Algorithm of Exercise 5.10 is from Friedman and Kogan [12].

Lower Bounds. The use of covering maps in the context of distributed algorithm was introduced by Angluin [2], and local neighbourhoods were studied by, among others, by Linial [16]. The general idea of Exercise 3.10 can be traced back to Yamashita and Kameda [29], while the specific construction in Figure 3.11 is from Bondy and Murty's textbook [7, Figure 5.10]. Lower bounds on graph colouring in the model of unique identifiers are from Linial's seminal work [16]; our presentation in Section 6.3 uses an alternative proof based on Ramsey's theorem, following, e.g., Naor and Stockmeyer [19] and Czygrinow et al. [9]. In particular, the idea of Exercise 6.6 is from Czygrinow et al. [9].

Local Work. Recent work by our research group is represented in algorithms VC3 [24] and VC2 [3]. Many exercises are also inspired by our work, including Exercises 3.1 and 3.3 [27], Exercise 3.6 [4], Exercise 4.1 [4], and Exercises 4.5–4.10 [27].

Index

Notation

$\lvert X\rvert$	the number of elements in set X
$f^{-1}(y)$	preimage of y, i.e., $f^{-1}(y) = \{x : f(x) = y\}$
$\deg_G(v)$	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, $\{0, 1, 2, \ldots\}$
$\mathbb{R}$	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
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	graphs, $G = (V, E)$
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)
U	subset of nodes
V	set of nodes
c, d	natural numbers
e	edges, elements of E
f, g, h	functions
i, j, k, ℓ	natural numbers
m_t	message
n	number of nodes, $n = \lvert V \rvert$
p	connection function, involution $p\colon P \to P$
r	natural numbers
s, t, u, v	nodes, elements of V
t	time step (round), $t = 0, 1, \dots, T$
w	walk
x_t	state

Algorithms

BMM	maximal matching in 2-coloured graphs, Section 2.4.1.
Colour	fast colour reduction in bounded-degree graphs, Section 5.3.5.
DPBit	fast colour reduction in directed pseudoforests, Section 5.3.4.
DPGreedy	greedy colour reduction in directed pseudoforests, Section 5.3.3.
DPSet	fast colour reduction in directed pseudoforests, Exercise 5.6.
Gather	gathering information in port-numbered graphs, Section 5.2.3.
Greedy	greedy colour reduction, Section 5.3.1.
HSEP	half-saturating edge packings, Section 4.2.5.
MEP	maximal edge packings, Section 4.2.6.
VC2	2-approximation of minimum vertex cover, Section 4.2.6.
VC3	3-approximation of minimum vertex cover, Section 2.4.2.

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, Patrik Floréen, Valentin Polishchuk, Joel Rybicki, Jukka Suomela, and Jara Uitto. A local 2-approximation algorithm for the vertex cover problem. In *Proc. 23rd International Symposium on Distributed Computing (DISC 2009)*, volume 5805 of *Lecture Notes in Computer Science*, pages 191–205. Springer, 2009. doi:10.1007/978-3-642-04355-0_21.

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

[5] Baruch Awerbuch. Complexity of network synchronization. *Journal of the ACM*, 32(4):804–823, 1985. doi:10.1145/4221.4227.

[6] Reuven Bar-Yehuda and Shimon Even. A linear-time approximation algorithm for the weighted vertex cover problem. *Journal of Algorithms*, 2(2):198–203, 1981. doi:10.1016/0196-6774(81)90020-1.

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

[8] 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.

[9] 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.

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

[11] Shlomi Dolev. *Self-Stabilization*. The MIT Press, Cambridge, MA, 2000.

[12] 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.

[13] 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.

[14] 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.

[15] Amos Korman, Shay Kutten, and David Peleg. Proof labeling schemes. In *Proc. 24th Annual ACM Symposium on Principles of Distributed Computing (PODC 2005)*, pages 9–18. ACM Press, 2005. doi:10.1145/1073814.1073817.

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

[17] Michael Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM Journal on Computing*, 15(4):1036–1053, 1986. doi:10.1137/0215074.

[18] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, San Francisco, 1996.

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

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

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

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

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

[24] 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.

[25] 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.

[26] 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.

[27] 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.

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

[29] 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.

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