Symmetry in Port-Labelled Anonymous Networks

by

Yongzhen Ren

A thesis submitted to

The Faculty of Graduate Studies of

The University of Manitoba

in partial fulfillment of the requirements

for the degree of

Master of Science

Department of Computer Science

The University of Manitoba

Winnipeg, Manitoba, Canada

July 2021

Thesis Advisor
Dr. Avery Miller

Author
Yongzhen Ren

Symmetry in Port-Labelled Anonymous Networks

Abstract

Our work investigates symmetry in a port-labelled anonymous network, a relatively weak but yet useful model of computation in distributed computing. In such a network, nodes have no identifiers, and for each node v, its incident links are labelled bijectively with the integers $\{ 1, 2, \ldots, \deg(v) \}$ called 'ports'. Each node has access to its 'view', a mathematical object completely representing the information that the node can learn from communicating with its neighbours. The question that arises is to understand how many nodes have their identities uniquely determined by such a view and what parameters for the number of nodes sharing a similar view are possible.

This thesis provides a detailed survey in this area and addresses a number of new questions (detailed in Chapter 3). Applications and related problems of port-labelled anonymous networks are explained in Chapter 4. The new results are presented in Chapters 5, 6, and 7. Some open problems are given in Chapter 8 and Appendix B.

Keywords: Algorithm, Degree Tree, Distributed Computing, Graph Theory, Multiplicity, Symmetricity, View

Contents

Title Page i

Abstract ii

Contents iii

List of Figures vii

List of Tables x

List of Algorithms xii

Acknowledgements xiii

Dedication xv

1 Introduction 1

2 Terminology and Preliminaries **6**

2.1 Port-Labelled Anonymous Networks 7

2.1.1 Views 10

2.2 From Multiplicity to Symmetricity 13

2.2.1 Preliminary Results About Symmetricity 14

2.3 Data Types 18

2.4 Number-Theoretic Definitions 21

3 Problem Statements **22**

4 Related Work **26**

4.1 Anonymous Networks 27

4.1.1 Views 28

4.1.2 Distributed Computing Problems 29

4.1.3 Computable Functions and Memory Use 30

4.1.4 Advice 32

4.1.5 Mobile Agents 33

4.1.6 Families of Topology 35

4.2 Mathematical Techniques 37

4.2.1 Graph Automorphism 37

4.2.2 Edge Labellings with a Sense of Direction 39

4.2.3 Incidence Colouring 40

5 Producing Symmetry **43**

5.1 Labelling an Anonymous Network 44
5.2 {1, 2}-Factorable Graphs Under a Fully Symmetric Labelling . 46
5.3 The Multiplicity Gap for the Petersen Graph 54
5.4 Port-Labelling Complete Graphs 56
5.4.1 Factorizations of Complete Graphs 56
5.4.2 An Algorithm for Producing a Desired Multiplicity . . 57
5.5 Rooted Product 69

6 Computing Symmetry **74**
6.1 Computing Views 75
6.2 Computing Multiplicities 79
6.3 Degree Trees: Motivation and Notation 83
6.4 Some Results About Degree Trees 86
6.5 Computing and Comparing Degree Trees 93
6.6 An Upper Bound on Symmetricity 99
6.7 Symmetricity of Trees 103

7 Producing Asymmetry **111**
7.1 Asymmetric Partial Labellings 113
7.2 Three Cases of Step 3 118
7.2.1 Case 1 118
7.2.2 Case 2 118
7.2.3 Case 3 120
7.3 The Final Algorithm 120

8 Future Work **122**

Appendix A The Graph Construction **126**

Appendix B Two Unsolved Cases **128**

B.1 Biregular Graphs 128

B.1.1 Designs 132

B.1.2 Hypergraphs 133

B.2 Non-{1, 2}-Factorable Regular Graphs 135

Bibliography **136**

List of Figures

2.1 A labelling on P_5 10

2.2 $\langle G, \mathbf{f} \rangle$ and $T_{\mathbf{f}}(v_1)$ 12

2.3 A vertex-labelled graph, along with one of its subgraphs and its only induced subgraph from $\{ v_1, v_2, v_3, v_5 \}$ 15

4.1 The Frucht graph 38

4.2 All four edge labellings with a SoD 40

4.3 The configurations of three neighbouring incidences 41

4.4 A legal 5-incidence colouring and a port-labelling on the Petersen graph 42

5.1 Creating an anonymous network 44

5.2 An example of a cubic graph that is not $\{1, 2\}$-factorable . . . 47

5.3 A visualization of lines 10 to 13 of Algorithm 5.3 52

5.4 A labelling on the Petersen graph whose multiplicity is 5 . . . 54

5.5 The vertex-labelled complete graph K_{12} 65
5.6 The partial labellings on three cliques of size four (Step 1) . . 66
5.7 The total labelling on a 1-factorization of $G_{0,1}$ (Step 2) 66
5.8 The total labelling on a 1-factorization of $G_{0,2}$ (Step 2) 67
5.9 The total labelling on a 1-factorization of $G_{1,2}$ (Step 2) 67
5.10 Port-switching the vertices in V_{G_0} (Step 3) 68
5.11 An illustration of $G \odot R$ 70
5.12 The rooted product of a port-labelled graph C_4 and a port-labelled rooted graph P_3 73

6.1 The data changing in ν and ν' when a directed edge (ν, ν') in a view is converted 76
6.2 Two graphs G_1 (left) and G_2 (right) under a fully symmetric labelling 84
6.3 The view $T_{\mathbf{f}}(v_1)$ and its corresponding degree tree $D_G(v_1)$ . . . 85
6.4 The degree tree up to level 2 for a vertex from a regular graph 88
6.5 The difference between a 2-degree graph and a biregular graph 89
6.6 $D(u)$ and $D(v)$ at level 1 (Proposition 6.3) 91
6.7 A non-biregular graph with $\kappa = 2$ 91
6.8 A graph G whose symmetricity is 1 100
6.9 Two 6-vertex graphs G_1 and G_2 102
6.10 Case 2, subcase (ii) of Lemma 6.3 104
6.11 The path between two vertices v_r and v'_r 106

6.12 Two isomorphic subtrees with a bridge connecting two roots . 110

7.1 The port change of v_{unique} after line 8 118

B.1 A failed attempt to create a graphic biregular graph 130

B.2 An example of Conjecture B.3 for a $(2, 6)$-biregular graph with 16 vertices 132

B.3 A 3-uniform non-regular hypergraph with seven vertices and four hyperedges 134

List of Tables

5.1 The adjacency matrix of the vertex-labelled graph G in Figure 2.2 45

5.2 The labelling matrix (representing the labelling $\mathbf{f}$) in Figure 2.2 46

5.3 Four divisions of all regular graphs 49

6.1 The 2-dimensional `array` $A_{T_\mathbf{f}}$ 77

6.2 $M_{\mathcal{D}:\mathcal{V}}$ associating $\mathcal{D}$ with $\mathcal{V}$ 97

List of Algorithms

5.1 Check if a regular graph is $\{1, 2\}$-factorable 49

5.2 Do a partial labelling on a 1-factor of a graph 50

5.3 Do a partial labelling on a 2-factor of a graph 51

5.4 Do a fully symmetric labelling on a $\{1, 2\}$-factorable graph . . 53

5.5 Return two neighbours of a vertex in a graph with the minimum and maximum ports 57

5.6 Switch the minimum and maximum ports of each vertex from a vertex subset 58

5.7 Port-label a complete graph totally whose multiplicity is d . . 62

5.8 Port-label a graph arbitrarily and totally 71

5.9 Return the rooted product of two port-labelled graphs 72

6.1 Generate the view of a vertex in a graph under a labelling . . 76

6.2 Fill A_{T_f} 77

6.3 Check if two views are similar 78
6.4 Return all integer factors of n 80
6.5 Compute the multiplicity of a graph under a labelling 81
6.6 Generate the degree tree of a vertex in a graph 93
6.7 Fill A_D 94
6.8 Assign its canonical name for a degree tree 95
6.9 Check if two degree trees are similar 96
6.10 Generate the degree tree collection $\mathcal{D}$ 96
6.11 Generate the degree tree vertex partition $\mathcal{V}$ 97
6.12 Generate $\mathcal{D}$, $\mathcal{V}$ and $M_{\mathcal{D}:\mathcal{V}}$ altogether 98
6.13 Compute $\mu(G)$ 101

7.1 Select $\widehat{V_D}$ from $\mathcal{V}$ 114
7.2 Return $\mathscr{N}_G(v, D)$ for a vertex v and a degree tree D 115
7.3 Do a partial labelling on the vertices from $\widehat{V_D}$ 116
7.4 Do an asymmetric partial labelling on a graph 117
7.5 Check if a graph with $\kappa = 2$ is biregular 119
7.6 Do an asymmetric partial labelling on any graph 120

Acknowledgements

First and foremost, I would like to express my gratitude to my advisor Dr. Avery Miller for his incredible patience and extensive expertise in both distributed computing and graph theory. Although we did not meet face-to-face for more than one year due to the COVID-19 pandemic, he still devoted enormous time to assist me in finishing the thesis. I also took his course on lower bounds and impossibility results back in 2019, and it turned out to be rather quite profound — with his dry sense of humour and a right blend of high teaching proficiency and Internet memes, it was much easier for me to survive the journey.

I really appreciated the detailed thesis feedback from Dr. Karen Gunderson; with her considerable effort, I formulated the solid mathematical arguments in the thesis. As the other member of my thesis committee, Dr. Shahin Kamali offered a set of valuable suggestion on improving the thesis; he shared immense knowledge both on cutting-edge research of online algorithms and

oral presentation skills with me in his amazing lectures.

Dr. Parimala Thulasiraman, as the current Associate Head (Graduate) for Department of Computer Science in the U of M, helped me overcome the biggest crisis since I joined the U of M community. I discovered a new world of parallel computing and high-performance computing due to her intensive lectures, which expanded my horizons a lot.

Without consistent love from my parents, I would not have enough courage to pursue the master degree in the first place. Whenever I felt upset about my academic life, they video-called me immediately from mainland China, delivered the warmest support to me and melted down the darkness inside me. My cousin Qianyi "Lucy" Liu and her husband Xu Zhang were always here for me when I was in need. Also, my kind-hearted landlady Hong "Sunshine" Xia, a cheerful new grandma, invited me over for dinner occasionally.

Last summer, I spent plenty of quality time with Lei "Tina" Wang. May she become a successful aerosol scientist and a happy wife one day, as she wished all the time.

Finally, two albums *Human* and *Outsider* by Three Days Grace (3DG), an Ontarian rock band, saved my boring life literally. Dr. Jordan B. Peterson, a world-renowned clinical psychologist and an unbelievably popular YouTube personality, rekindled my life and taught me how to act as a responsible man in the modern Western society.

The thesis is dedicated to my beloved parents.

This page is intentionally left blank.

1 Introduction

> To define is to lose. The essence
> of all things is the Nameless.
> The Nameless is unknowable,
> mightier even than Brahma.
>
> *Lord of Light*
> Roger Zelazny

For networks widely occurring in the real world, we design our connection systems so that each node (a computer, a server, a mobile device, an unmanned aerial vehicle, etc.) has an identity (ID) to distinguish itself from others in the same network, represented by a 6-byte hardware-manufacturer-generated MAC (Media Access Control) address burned into its Ethernet card or an IPv4/IPv6 (Internet Protocol version 4/6) address assigned administratively [49, pp. 23–24]. Along with such a digital label, these networks have a great amount of outgoing/incoming ports as communication endpoints. Two core communication protocols, implemented as the transport layer and comprising

the Internet protocol suite, both choose to use 16-bit unsigned integers (thus ranging from 0 to 65535) as ports, respectively the connection-oriented Transmission Control Protocol (TCP) and the connectionless User Datagram Protocol (UDP) [83, pp. 453–454]. In terms of distributed computing research, a network, often treated as a mathematical object called graph, is eponymous if all nodes in it possess unique identifiers [74, p. 380]. In contrast, our work concerns port-labelled anonymous networks, in which nodes have the same identifiers or have no identifiers at all, but the ports at each node do have identifiers [90, p. 70].

The importance of our research on properties of port-labelled anonymous networks, especially their symmetry, analyzed by graph-theoretic methods, consists of both practical and theoretical aspects.

In practice, the model simplifies network designs by avoiding complicated MAC/IP addresses and leaving relatively short-length distinct ports only, in order to improve network operational efficiency and rule out possible protocol redundancy. Also, considering all other settings being equal, a port-labelled anonymous network where every node runs its algorithms in the same way is likely to be easier to troubleshoot than an eponymous one where all nodes execute entirely different sequences of steps, depending on their identities. Potentially saving resources (such as memory usage and process time) is another focal point, embedded in nearly all areas of computer networks. Occasionally, it is not economically feasible to equip every node with a unique identifier, such as installing resource-limited temperature sensors for

an ever-moving bird flock [6, pp. 235–236]. In other contexts, the nodes may want to refrain from revealing their identities out of privacy concerns to enhance security in a sensitive environment. For instance, to opt out of notorious ongoing global data surveillance programs run by government intelligence agencies including, but not limited to, PRISM, Tempora and XKeyscore[1], the public may gradually switch to anonymity-based alternatives of currently available Internet, such as The Onion Routing (Tor), the Invisible Internet Project (I2P) [29], and various forms of anonymous P2P filesharing systems [27].

However, we cannot simply remove identifiers from our networks, since the underlying algorithms rely on them. This motivates us to try to understand what is algorithmically possible or impossible in port-labelled anonymous networks, i.e., the theoretical aspects. While plenty of researchers in the area concentrated on feasibility results with regard to distributed problems running on such a network, there is still much to understand about characterizing its symmetry beyond two fundamental works from Yamashita and Kameda [90,91]. Network symmetry, often measured by 'symmetricity' (see Definition 2.17), is a crucial factor when it comes to figuring out possibility/impossibility in port-labelled anonymous networks (see Subsection 2.2.1). When running an algorithm within a fixed amount of time, each node can only obtain a certain amount of information about other nearby nodes in the network, represented by 'views' (see Definition 2.11). If two nodes have the exact same information,

[1]See the website PRISM Break for more information.

they will perform the exact same actions — they are all algorithmically indistinguishable in terms of deterministic distributed computing [8, p. 3]. If a node in such a network fails to distinguish itself from other nodes, the algorithm performed by these anonymous nodes then is not able to accomplish computationally important tasks, such as leader election (see Theorem 2.3), edge election (see Theorem 2.4), vertex colouring, finding an independent set, clustering etc.

Previous works have had to cope with this in some way: when designing algorithms, the researchers are restricting their attention to networks where the task is actually possible to solve (e.g., networks that are sufficiently asymmetric); when proving impossibility results or lower bounds, their proofs are building specific networks (or classes of networks) that have a certain amount of network symmetry. Accordingly, as researchers consider certain problems which they want to solve, they have to understand in which scenarios their problem is solvable or not solvable, and this all comes down to symmetry. Unfortunately, many existing tools to study symmetry in graphs are insufficient (see Section 4.2). Additionally, connecting the theoretical with more practical considerations, engineering work has already begun on designing debugging/monitoring platforms for anonymous distributed computing systems like ViSiDiA [1].

The goal of this thesis is to study various problems arising from symmetry in port-labelled anonymous networks. There are two types of results that we are interested in, specified later in Chapter 3 based on the foundational

definitions from Chapter 2:

1. For a desired amount of symmetry, can we construct a port-labelled anonymous network demonstrating it?

2. Given an (either totally or partially) port-labelled anonymous network, can we characterize its symmetry?

2 Terminology and Preliminaries

> Wisdom was not at the top of the graduate school mountain, but there in the sandpile at Sunday school.
>
> *All I Really Need To Know I Learned In Kindergarten*
> Robert Fulghum

In our work, we adopt and modify the naming conventions used by Yamashita and Kameda [90], who inherited implicitly the similar notation [5]. We first present the port-labelled anonymous variant of the classical synchronous message-passing model and define 'multiplicity' and 'symmetricity' in such a model, using 'views'. Then we introduce several primitive/abstract data types occurred later in the algorithms, ending the whole chapter with several number-theoretic definitions.

The symbol ■ is placed to indicate the end of a proof. Text in **boldface** indicates a new term definition or a reserved keyword in algorithms, while text

in *italics* indicates a notion that needs to be emphasized. Unless otherwise stated, all numbers below are natural numbers; all mathematical indices start from 1 and all algorithmic ones start from 0. The subscripts may be dropped when no ambiguity can arise.

2.1 Port-Labelled Anonymous Networks

In a synchronous message-passing model in distributed computing, a **processor** is an entity with unbounded computational power, i.e. it can compute any function in negligible time. In each synchronous communication round, each exchanges arbitrary-length binary strings bidirectionally with others to which it is connected by a **communication link**, which is a fault-free bidirectional wire connecting a pair of processors. A **network** is a combination of a set of processors executing the same deterministic algorithm and their links.

Definition 2.1. A (finite) **undirected graph** $G = (V, E)$ is an ordered pair, where V is a set of n vertices and E is a set of edges (v, v') between two different vertices v and v', with $|E| = m$. A graph containing no loops or multiple edges is called **simple**. A graph is **connected** when there exists a path[1] between every pair of vertices in such a graph. A **directed graph** is a graph where each edge is replaced by an edge with a direction from one vertex to another.

[1] A path is a sequence of distinct vertices, with every consecutive pair in the sequence forming an edge.

All the graphs hereinafter have at least two vertices ($n \geq 2$ and $m \geq 1$), and they are simple and connected. A network is modelled by a graph. In what follows, the terms *networks* and *graphs* are used interchangeably. Naturally, so do *processors* and *vertices*, and *links* and *edges*. In a typical distributed computing configuration, each processor owns a unique identifier (normally an integer or an alphabet letter), in order to compute values or make algorithmic decisions. By contrast, processors in a network are all **anonymous** provided that they do not possess such identifiers, or they all have the same label; correspondingly a network with anonymous processors is called a **purely anonymous network**. A purely anonymous network, i.e. a graph without extra labels attached to vertices or edges, belongs to SB class [65, p. 31] and is located at the bottom of the computability hierarchy of weak models in distributed computing; this type of network cannot carry out basic tasks. Instead, we focus on port-labelled anonymous networks, which is a variant belonging to $\mathtt{VV_c}$ class [65, p. 37] instead.

Definition 2.2. A **neighbour** of a vertex v in a graph G is a vertex adjacent to v; the (open) **neighbourhood** of v is the set of all its neighbours excluding v itself, denoted as $N_G(v)$.

Definition 2.3. The **degree** of a vertex v in a graph G is the size of $N_G(v)$, denoted by $\deg(v)$; Δ_G represents the maximum degree among all vertex degrees in G.

Definition 2.4. A **port label** (or just **port**) of a vertex v is an integer

attaching to one of the edges that v is incident on.

In our work, ports for a vertex v range from 1 to $\deg(v)$ by default.

Definition 2.5. A **local labelling function** f_v on a vertex v of a graph G is a bijection $\psi\colon N_G(v) \to \{\, 1, 2, \ldots, \deg(v) \,\}$, mapping v's neighbourhood to the set of its ports. If p is the port at v_i's end on the edge (v_i, v_j), then $f_{v_i}(v_j) = p$.

The definition of local labelling function allows two ports at the two endpoints of an edge to be assigned independently.

Definition 2.6. A **(port-)labelling** on a graph $G = (V, E)$ is a set of local labelling functions for the vertices in G, denoted as $\mathbf{f}$. That is to say, for a graph $G = (V, E)$, $\mathbf{f} = \{\, f_v \mid v \in V \,\}$.

A labelling function at a specific vertex v is **total** if all of v's incident edges have been mapped; a labelling $\mathbf{f}$ is **total** if every vertex in a graph has an associated total labelling function. In contrast, a **partial** labelling $\mathfrak{f}$ on a graph is one that is not total. In our work, any labelling function must be total; a labelling is assumed to be total, unless otherwise stated, since partial labellings will be used as steps towards the construction of the total labelling in our algorithms.

Example 2.1. In Figure 2.1, there is a path graph P_5[2] under a labelling. The elements of local labelling function f_{v_3} are marked red: the vertex v_3 connects

[2] A path graph P_n is a connected graph with two vertices of degree 1 and the other $n-2$ ones of degree 2, different from a path in a graph (see page 7).

its two neighbours v_2 and v_4 via ports 2 and 1 respectively, i.e. $f_{v_3}(v_2) = 2$ and $f_{v_3}(v_4) = 1$.

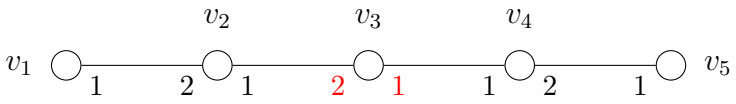


Figure 2.1: A labelling on P_5

A **port-labelled anonymous network** $\langle G, \mathbf{f} \rangle$ is a graph G under a labelling $\mathbf{f}$. In such an anonymous network, a vertex may gather all available information and transmits it through port p, including the port itself; correspondingly, while receiving a message through port p', a vertex is aware of p'. It is simply referred to as **anonymous networks** or **port-labelled graphs** in the rest of our work. As demonstrated in Figure 2.1, vertices in a graph are occasionally labelled for descriptive purposes only, and such a graph is called **vertex-labelled** — an algorithm being executed by the vertices does not have access to such labels.

2.1.1 Views

We proceed to define the terms needed to understand views in such an anonymous network.

Definition 2.7. A **rooted graph** is a graph $G = (V, E)$ where one vertex $v \in V$ has been designated as the root.

Definition 2.8. A (free) **tree** is a graph where any two vertices are connected by exactly one path.

Every finite tree has n vertices and $n-1$ edges.

Definition 2.9. The **distance** between two vertices v and v' in a graph is the number of edges in a shortest path connecting them, denoted as $d(v, v')$; the **level** of a vertex v in a rooted tree whose root is v_r is defined as $d(v_r, v)$.

Definition 2.10. An **arborescence** is a directed rooted tree where all its edges point away from the root, i.e. where there exists a unique directed path from the root to each vertex.

Every arborescence is a directed acyclic graph (DAG) but not vice versa. The **in-degree** (resp., **out-degree**) of each vertex in a directed graph is the number of directed edges pointing towards (resp., leaving from) it.

Definition 2.11. The **view** $T_{\mathbf{f}}(v)$ of a vertex v in a graph G under a labelling $\mathbf{f}$ is a labelled arborescence with infinite levels, defined recursively as follows: the root of $T_{\mathbf{f}}(v)$ is v itself; for each neighbour of v in G, $T_{\mathbf{f}}(v)$ contains a vertex v_i and a directed edge from v to v_i with ports $f_v(v_i)$ and $f_{v_i}(v)$ at its source and destination respectively; the vertex v_i is the root of $T_{\mathbf{f}}(v_i)$.

It should be stressed that view $T_{\mathbf{f}}(v)$ itself is directed, while the underlying graph is undirected.

Example 2.2. On the left of Figure 2.2, the graph without red vertex labels is a graph G under a labelling $\mathbf{f}$. These additional labels just help us construct

the view, but do not appear in the view itself. The view of a vertex v_1 in G is displayed on the right: the black-coloured vertex is the root of $T_{\mathbf{f}}(v_1)$ and the dashed boxes indicate the views from all v_1's neighbours appended to the nodes at level 1 of v_1's view.

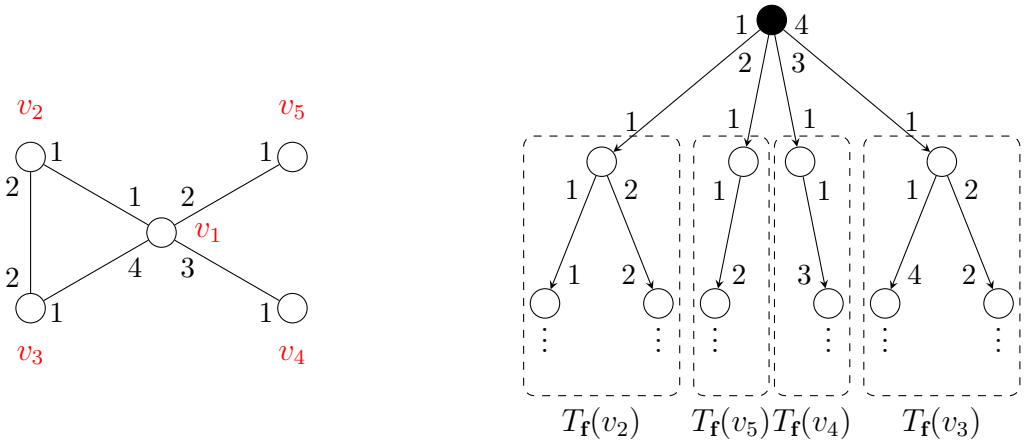


Figure 2.2: $\langle G, \mathbf{f} \rangle$ and $T_{\mathbf{f}}(v_1)$

Definition 2.12 (Graph Isomorphism)**.** Let $G = (V, E)$ and $G' = (V', E')$ be two graphs. G and G' are **isomorphic**, if there exists a bijection $\psi \colon V \to V'$ such that $(v_i, v_j) \in E$ iff $(\psi(v_i), \psi(v_j)) \in E'$. Two directed graphs are isomorphic if their underlying undirected graphs are isomorphic and are oriented the same.

Definition 2.13 (View Similarity/Dissimilarity)**.** Two views $T_{\mathbf{f}}$ and $T'_{\mathbf{f}}$ are said to be **similar** if there exists a directed graph isomorphism between them, preserving the roots and all edge labels — we then write $T_{\mathbf{f}} \equiv T'_{\mathbf{f}}$; otherwise, $T_{\mathbf{f}}$ and $T'_{\mathbf{f}}$ are **dissimilar**, written as $T_{\mathbf{f}} \not\equiv T'_{\mathbf{f}}$.

It is worth noting that two vertices v and v' from different graphs can have $T_{\mathbf{f}}(v) \equiv T_{\mathbf{f}}(v')$.

The view up to level ℓ for a vertex v in a graph G under a labelling $\mathbf{f}$ is $T_{\mathbf{f}}(v)$ truncated to the first ℓ levels, denoted by $T^{\ell}_{\mathbf{f}}(v)$. For convenience, the notation $T_{\mathbf{f}}(v)$ will refer to $T^{n-1}_{\mathbf{f}}(v)$ from now on, since Norris [78] already proved that $T_{\mathbf{f}}(v) \equiv T_{\mathbf{f}}(v')$ iff $T^{n-1}_{\mathbf{f}}(v) \equiv T^{n-1}_{\mathbf{f}}(v')$.

Definition 2.14. The set of all dissimilar views from a graph G under a labelling $\mathbf{f}$ is denoted as $\mathcal{T}_{\mathbf{f}} = \{\, T_{\mathbf{f}}(v) \mid v \in V \,\}$.

Definition 2.15. The set of all vertices with a view that is similar to $T_{\mathbf{f}}$ is denoted as $V_{T_{\mathbf{f}}} = \{\, v \mid T_{\mathbf{f}}(v) \equiv T_{\mathbf{f}} \,\}$.

2.2 From Multiplicity to Symmetricity

As a foundation of defining multiplicity, we start from Proposition 2.1 [90, p. 72].

Proposition 2.1. *For any graph $G = (V, E)$ under any labelling $\boldsymbol{f}$, the size of $V_{T_{\boldsymbol{f}}}$ is the same for all $T_{\boldsymbol{f}} \in \mathcal{T}_{\boldsymbol{f}}$.*

In other words, for any graph with n vertices and any labelling $\mathbf{f}$ on it, it is guaranteed that there exist fixed $x, y \geq 1$ with $xy = n$ such that there are x dissimilar views in total and each dissimilar view is shared by exactly y vertices, which gives rise to the subsequent definition.

Definition 2.16. The **multiplicity** $s_{\mathbf{f}}$ is the number of vertices sharing similar views of a graph under a labelling $\mathbf{f}$, defined as

$$s_{\mathbf{f}} = \frac{n}{|\mathcal{T}_{\mathbf{f}}|}.$$

By definition, for any $T_{\mathbf{f}} \in \mathcal{T}_{\mathbf{f}}$, we have $|V_{T_{\mathbf{f}}}| = s_{\mathbf{f}}$. Using multiplicities, we can now define the symmetricity of a graph.

Definition 2.17. The **symmetricity** $\sigma(G)$ of a graph G is the largest multiplicity which can be achieved among *all* possible labellings on G, i.e.,

$$\sigma(G) = \max \{\, s_{\mathbf{f}} \mid \mathbf{f} \text{ is a labelling on } G \,\}.$$

The symmetricity $\sigma(G)$ is a property solely determined by the topological structure of G itself, which indicates the maximum number of vertices in G sharing similar views in the worst-case labelling.

Observation 2.1. For a path graph P_n,

$$\sigma(P_n) = \begin{cases} 2, & n \text{ is even} \\ 1, & n \text{ is odd.} \end{cases}$$

2.2.1 Preliminary Results About Symmetricity

Several additional graph-theoretic definitions are required first, before some intriguing results are shown.

Definition 2.18. A graph $G' = (V', E')$ is a **subgraph** of another graph $G = (V, E)$ iff $V' \subseteq V$ and $E' \subseteq E$. A subgraph $G'' = (V'', E'')$ is said to be

vertex-induced, often simply called **induced** from G, if $V'' \subseteq V$, $E'' \subseteq E$ and E'' consists of all the edges in E both of whose endpoints are in V''.

Example 2.3. Figure 2.3 shows the difference between a subgraph and an induced subgraph of the same graph with the same vertex subset $\{ v_1, v_2, v_3, v_5 \}$. The leftmost graph comprises five vertices and the graph in the middle is one of its subgraphs with three edges. The subgraph induced by these vertices is drawn to the right, and includes all six edges with both endpoints in that set.

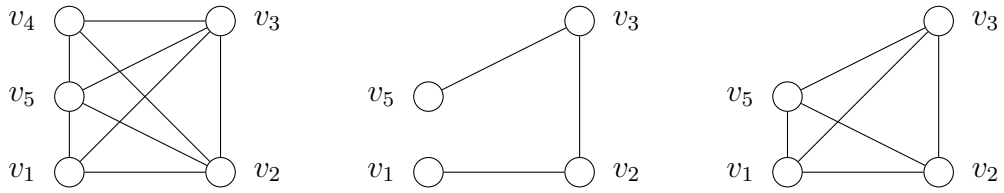


Figure 2.3: A vertex-labelled graph, along with one of its subgraphs and its only induced subgraph from $\{ v_1, v_2, v_3, v_5 \}$

Definition 2.19. A graph is **regular** if each vertex in the graph has the same degree; a regular graph with all vertices of degree k is **k-regular**. A 3-regular graph is also called **cubic**.

Definition 2.20. A **factor** of a graph G is a subgraph with the same vertex set as G, also known as a **spanning subgraph**. A graph G is the **sum** of factors if G is their edge-disjoint union, and the set of factors involved in this sum is known as a **factorization**. A **k-factor** of a graph is a spanning k-regular subgraph and a **k-factorization** is a partition of all edges in a

graph into disjoint k-factors. A graph G is **k-factorable** if it admits a k-factorization. An **$\{x, y\}$-factorization** is a partition of all edges into disjoint factors where each factor is either an x-factor or a y-factor. A graph G is **$\{x, y\}$-factorable** if it admits an $\{x, y\}$-factorization.

Notably, a 2-factor is a collection of disjoint cycles[3].

Definition 2.21. A graph $G = (U \cup V, E)$ is **bipartite**, if for two disjoint vertex sets U and V, every edge from E connects a vertex in U to one in V.

Now we have the next definition and the corresponding theorem [90, p. 78], which relate an anonymous network's symmetricity to the factorizations of the underlying graph.

Definition 2.22. A graph $G = (V, E)$ with n vertices satisfies the **factorization condition** with an integer g, if there exists a disjoint partition for vertex set V such that $V = \bigcup_{i=1}^{g} V_i$ with $|V_i| = n/g$, meeting both of the following conditions:

1. For $i \in \{ 1, 2, \ldots, g \}$, the subgraph of G induced by V_i is $\{1, 2\}$-factorable;

2. For $i, j \in \{ 1, 2, \ldots, g \}$ $(i \neq j)$, each bipartite graph $(V_i \cup V_j, E \cap \{ (v_i, v_j) \mid v_i \in V_i, v_j \in V_j \})$ is regular.

Theorem 2.1. *For any graph $G = (V, E)$,*

$$\sigma(G) = n / \min \{ g \mid G \text{ satisfies the factorization condition with } g \}.$$

[3] A cycle graph C_n is a 2-regular connected graph with n vertices.

Let $g = 1$ in the theorem above, and we have Corollary 2.1.

Corollary 2.1. *A graph G with n vertices is $\{1, 2\}$-factorable iff $\sigma(G) = n$.*

Here are two results with regard to symmetricity [91, p. 90 and p. 92].

Lemma 2.1. *No regular graph G with n vertices has $\sigma(G) = n/2$.*

Theorem 2.2. *For a graph $G = (V, E)$ with $|V| = n$ vertices and $|E| = m$ edges,*

1. *Determining if $\sigma(G) = n$ or $n/2$ is solvable in $O(m\sqrt{n})$ time;*
2. *For any $g \geq 3$, determining if $\sigma(G) = n/g$ is $\mathcal{NP}$-complete;*
3. *Determining if $\sigma(G) = 1$ is co-$\mathcal{NP}$-complete.*

Before showing two more theorems [90, pp. 74–75] on task solvability, we need to introduce two classical distributed computing problems briefly. The **leader election** (usually abbreviated to LE) is to elect a vertex v as the leader after finite rounds, where v knows that it has been elected and other $n - 1$ vertices are aware that they have not. Meanwhile, the **edge election** is to select an edge $e = (v, v')$ in the sense that two vertices v and v' know the ports at each end of e and the rest of the vertices know that they are not incident to e.

Theorem 2.3. *LE is guaranteed to be solvable on a graph G iff $\sigma(G) = 1$.*

Proof sketch. Each vertex can build its own view within $n - 1$ rounds [78], and finds out others' views using extra $n-1$ communication rounds. Then we

determine the smallest dissimilar view (with respect to lexicographic order) in such a graph. The view of vertex v up to level $2n-2$ is exactly the information that v can acquire in $2n-2$ rounds of a deterministic algorithm. In one direction, if there is a vertex with a dissimilar view after $2n-2$ rounds, it can be elected as a leader using such an algorithm. In the other direction, if no vertex has a dissimilar view after $2n-2$ rounds, then every vertex has a "twin" that will output the same value — a leader cannot be elected in $2n-2$ rounds. ■

Theorem 2.4. *The edge election is guaranteed to be solvable on a graph G iff one of the following conditions are held:*

1. $\sigma(G) = 1$ *or*
2. $\sigma(G) = 2$ *and there exists an edge (u, v) such that $T_{\boldsymbol{f}}(u) \equiv T_{\boldsymbol{f}}(v)$ for any labelling $\boldsymbol{f}$ with $s_{\boldsymbol{f}} = 2$.*

2.3 Data Types

The primitive data types used in our algorithms include integers, Boolean values (ones with two possible values True and False) and strings. Particularly, a **string** is a fixed sequence of symbols from a fixed alphabet. For two strings s and s', the non-commutative operation $s + s'$ is to return a string consisting of the concatenation of s and s'; the comparison operation $s = s'$, when used in conditional statements, returns a Boolean value based on whether s and s'

are identical character-by-character.

The abstract data types[4] listed below are central to build the algorithms in Chapters 5, 6 and 7. The letter $\hat{e}$ represents a generalized element in the following part and N stands for the number of elements stored in each ADT.

Definition 2.23. A `set` is an ADT storing unique values without any particular order. For a `set` S, the operation $\hat{e} \in S$ checks if S contains $\hat{e}$ or not and returns a Boolean value. The operation $S \leftarrow S \cup \{ \hat{e} \}$ adds $\hat{e}$ into S if $\hat{e}$ is not already in S. The size of S is returned by $|S|$.

We implement `sets` by applying the union-find data structure, since it can achieve $O(N)$ space complexity and $O(\alpha(N))$ searching performance in the worst case, where $\alpha(N)$ is the exceedingly slow-growing inverse Ackermann function and hence treated as essentially constant [30, p. 585].

Definition 2.24. An `array` is an ADT containing a number of elements, each accessed by specifying a non-empty sequence of indices. The **dimension** is the number of indices needed to specify an element. For an `array` A, $A[x, y, \dots]$ accesses the element by sequence $(x, y, \dots)$ and $A[x, y, \dots] \leftarrow \hat{e}$ writes $\hat{e}$ into A as the element that can be subsequently accessed using sequence $(x, y, \dots)$. For two `arrays` A and A', the operation $A \leftarrow A'$ updates the elements of A with A': for each at $(x, y, \dots)$ in A, it is replaced with the one at the same indices in A', if there exists such one.

[4] An abstract data type (ADT) is a mathematical model for data types whose behaviours (semantics) are defined by a set of values and a set of corresponding operations without expatiating on the concrete representation of the data.

All `arrays` are static, which means that the size of an `array` must be declared in advance. `Arrays` here are implemented as standard arrays, with $O(1)$ storing/accessing time and linear $O(N)$ space, even in the worst case.

Definition 2.25. A `dictionary` is an ADT consisting of key-value pairs such that each key appears without repetition. For any key-value pair $(\hat{e}_k, \hat{e}_v)$ in $M_{X:Y}$ with key set X and value set Y, the operation $M_{X:Y}[\hat{e}_k]$ accesses the value $\hat{e}_v$. The operation $M_{X:Y}[\hat{e}_k] \leftarrow \hat{e}'_v$ replaces the pair $(\hat{e}_k, \hat{e}_v)$ with $(\hat{e}_k, \hat{e}'_v)$.

`Dictionaries` are powered by hash tables due to their outstanding *average* performance: the lookup, insertion and deletion time is $O(1)$ for average cases and $O(N)$ for the worst case, while its storage requirement is only $\Theta(N)$ [30, pp. 253–285].

Definition 2.26. A `list` is a variable-length ADT representing a finite sequence of values. A new element $\hat{e}$ is appended to a `list`, right after the last pre-existed ones. The other basic operations on a `list` include accessing and deletion of a specific element.

A `list` here is implemented as a doubly linked list, so the running time of appending is $O(1)$, and $\Theta(N)$ time is required for deletion in the worst case [30, p. 238].

Definition 2.27. A `tree`[5] is a recursive hierarchical data structure composed of a collection of nodes. There is exactly one designated **root**, and all other

[5] See Definition 2.8 for the graph-theoretic definition with the same name.

nodes are non-root nodes. Each node ν has one variable that store its value, which can be empty sometimes; meanwhile, ν has a **children** list A_ν, which contains references to all nodes that have ν as their **parent**. A node with an empty children list is a **leaf**.

For simplicity, we may use its root to represent the whole `tree`, especially in a **return** statement of the functions from our algorithms.

2.4 Number-Theoretic Definitions

We use $x \mid y$ to represent that x is a divisor of y or y is a multiple of x; if x is not a divisor of y, then $x \nmid y$.

Definition 2.28. An integer n is **prime** if its only divisors are 1 and n.

Definition 2.29. The **greatest common divisor (gcd)** of a sequence of two or more non-negative integers, which are not all 0, is the largest positive integer that is a divisor of all elements in the sequence, denoted as $\gcd(x, y, \dots)$.

3 Problem Statements

> The power to question is the basis of all human progress.
>
> — Indira Gandhi

In this chapter, we define five general problems by considering four parameters: a graph G, a labelling $\mathbf{f}$, the number of vertices n and a divisor d of n ($1 \leq d \leq n$ and $d \mid n$), which is a potential candidate for the multiplicity $s_{\mathbf{f}}$. When n is fixed, all possible values for d are ensured; n can also be derived implicitly from $\mathbf{f}$; n is dependent on the underlying graph G and so is $\mathbf{f}$.

These five problems, generated via rearranging four parameters above, can be classified into two types, as mentioned in Chapter 1:

1. Given some parameters, can we come up with examples of graphs and labellings that have a certain amount of symmetry? (Problems 1, 2 and 3 below)

2. Given a graph (with a labelling), can we determine how much symmetry

it has? (Problems 4 and 5 below)

Both types are relevant to our understanding of when problems are solvable or unsolvable in port-labelled anonymous networks.

Problem 1

Given n, provide a graph G with n vertices such that for all $d \mid n$, there exists a labelling $\mathbf{f}$ on G where $s_{\mathbf{f}} = d$.

We show that there exists such a labelling on a graph whose multiplicity is d, by later presenting a labelling scheme (Algorithm 5.7) on a complete graph (see Definition 5.3) with a technique called 'port-switching' (defined after Algorithm 5.5) in Section 5.4.

Problem 2

Given any $d \mid n$, provide a graph G with n vertices and a labelling $\mathbf{f}$ on G such that $s_{\mathbf{f}} = d$.

If Problem 1 has a solution, then the same graph would solve Problem 2.

Problem 3

Given n, list all $d \mid n$ such that there exists a graph with n vertices under a labelling $\mathbf{f}$, such that $s_{\mathbf{f}} = d$.

A solution to Problem 1 provides a solution to Problem 3: a complete graph under a certain labelling can be used as the answer. Also in Section 5.5, we provide an alternative way of generating other qualified graphs systematically.

Problem 4

Given a graph under a labelling $\mathbf{f}$, compute $s_{\mathbf{f}}$.

An algorithm for calculating multiplicity is presented in Section 6.1.

Problem 5

Given a graph G, list all possible multiplicities $s_{\mathbf{f}}$ and a corresponding labelling $\mathbf{f}$ on G, for each $s_{\mathbf{f}}$.

Problem 5 can be split into three independent subproblems.

Subproblem 1 of Problem 5

Given a graph, determine lower/upper bounds on its multiplicities.

Firstly, the range of $s_{\mathbf{f}}$ has to be decided. From Section 2.2, we already knew that the maximum of all possible multiplicities for a graph G is defined as the symmetricity $\sigma(G)$, $s_{\mathbf{f}}$ is always an integer for any labelling $\mathbf{f}$, and $0 < s_{\mathbf{f}} \leq \sigma(G)$ holds for any graph G. With regard to the minimum multiplicity, we demonstrate in Chapter 7 that for most classes of graphs G with more than two vertices, there always exists a labelling $\mathbf{f}$ such that $s_{\mathbf{f}} = 1$ by applying port-switching again. Due to its $\mathcal{NP}$-completeness, shown in Theorem 2.2, it is quite unwieldy to computing the symmetricity of a graph. Hence, a tight enough upper bound for $\sigma(G)$ (for its multiplicities as well) is sometimes more appropriate in practice. In Section 6.6, we offer such a bound for by

introducing a new concept called 'degree trees' (see Definition 6.1), which are basically unlabelled views.

Subproblem 2 of Problem 5

Given a graph G with n vertices, determine if every $d \mid n$ can be a valid multiplicity $s_{\mathbf{f}}$ for some labelling $\mathbf{f}$ on G.

Given a graph G with n vertices, all divisors of n are potential values for multiplicities $s_{\mathbf{f}}$. There are two mutually exclusive possible answers to this problem:

1. At least one divisor less than $\sigma(G)$ cannot be a valid multiplicity for any labelling $\mathbf{f}$;
2. All divisors of n less than $\sigma(G)$ can be a valid multiplicity for some labelling $\mathbf{f}$.

Although Subproblem 2 of Problem 5 is not fully solved, in Section 5.3, we show that the Petersen graph falls into the first case. Meanwhile, in Section 5.4, we show that the complete graph falls into the second one.

Subproblem 3 of Problem 5

Given a graph, list all possible multiplicities and at least one of their corresponding labellings.

We have not answered Subproblem 3 of Problem 5 in this thesis, and we leave that for future work.

4 Related Work

> If you copy from one book, that's plagiarism; if you copy from many books, that's research.
>
> Wallace Notestein

In this chapter, we summarize relevant research results from the literature, and we present them in two major categories. The first category is distributed computing in anonymous networks, including solvability/computability of various classical problems, and for specific classes of graphs. The second one includes results about a classical form of graph symmetry ('automorphism') and two other types of edge labellings, which turn out to be quite different from the port-labelling that we study in this thesis.

4.1 Anonymous Networks

The formal study of anonymous networks was started by Angluin [5], who set out to study which network functions (such as constructing a spanning tree) could be carried out by nodes without identities. More than a decade later, Yamashita and Kameda [90] came up with a formal model and framework for studying anonymous networks, and gave a full characterization of types of networks where LE and map construction are solvable in a deterministic fashion.

We restrict attention to deterministic algorithms rather than probabilistic ones. It has been shown by Emek et al. [50] that randomized Las Vegas algorithms (probabilistic but always correct) have the same computational power as deterministic algorithms, as long as the deterministic algorithm is executed after the nodes of the network have been labelled with a distance-2 colouring. Also, we assume that all anonymous networks henceforth are static and fault-free, where links remain consistent during the whole execution time. All the anonymous networks mentioned in the chapter are connected, while disconnected ones are usually investigated for concurrent systems [68]. We do not cover results about node-labelled networks, e.g., networks where each node possesses a unique identifier, as well as networks where node labels are not necessarily unique [24, 38, 48, 74, 88].

In the literature on anonymous distributed computing, three pillars of models of computation are the message-passing model (bidirectional commu-

nication), the shared-memory model (using atomic read/write registers) and the local computation model[1]. Other models that have been studied include quantum networks [70] and anonymous radio networks [77]. We only consider the message-passing model without bandwidth restriction.

4.1.1 Views

Klasing et al. [69] proved that, for some port-labellings, it is possible for each node to distinguish its own view from other nodes by only considering its truncated view up to level $O(\min(\mathbb{D}, \log n))$, where is $\mathbb{D}$ the diameter of a graph. However, looking at the truncated view up to level $\Omega(\mathbb{D} \log (n/\mathbb{D}))$ is always sufficient [39, 66].

Fraigniaud and Pelc [57] proved that if two nodes have similar views to depth $\hat{n} - 1$, then their whole views must be similar, where $\hat{n}$ is the number of nodes with dissimilar views (equivalently, $\hat{n}$ is in essence $|T_{\mathbf{f}}|$ or the size of its corresponding quotient graph [90]).

Within at most $2n$ rounds, Tani [84] proposed that a compressed view (called **folded view** or **f-view** for shorthand) can be constructed with polynomial $O(\Delta m n^3 \log \Delta)$ bit-complexity (the total number of bits transmitted); there is also an algorithm for counting non-isomorphic f-views in an anonymous network in $O(\Delta n^5 \log n)$ time complexity, working for any network

[1] Local computation in the context of anonymous networks is a general term indicating that nodes initialized with the same state interact *locally* with graph rewriting/relabelling rules on individual edges or star-shaped subgraphs, until no rules can be applied on any node [12].

topology.

Although the view acts as the central concept among many of the papers related to anonymous networks, there are still some exceptions that do not apply views directly [26, 34].

4.1.2 Distributed Computing Problems

Distributed computing problems were first considerably studied for networks with distinct labels, and then later studied for their anonymous counterparts. As we explained in Subsection 2.2.1, LE, the process of designating a single node as the leader, may be one of the most fundamental problems in distributed computing. With unlabelled nodes, LE is impossible to solve in symmetric networks [5]. Attiya et al. [10] verified further that no deterministic algorithm can break symmetry to solve LE in purely anonymous networks within bounded time without error, even if it is given the number of nodes as advice[2]. So it is necessary to create LE algorithms without depending on node identities but exploit asymmetries of the network itself based on the port-labelling or its topological structure. After LE is solved, it is possible to assign short unique identifiers to the nodes of an anonymous network using a distributed algorithm [58]. Fusco and Pelc [61] showed that the number of rounds needed to declare LE infeasible is equal to $\Theta(\mathbb{D} + \lambda)$, irrespective of which kinds of LE (strong or weak), where the level of symmetry λ is the smallest level where some node has a dissimilar view when LE is possible,

[2]Its formal definition can be found in Subsection 4.1.4.

also called the election index [42].

Other distributed computing problems were examined including sorting multisets [51], the k-grouping and pairing problem (a generalized version of traditional LE) [23], vertex/set cover [7], and the enumeration problem (in asynchronous anonymous networks) [14].

4.1.3 Computable Functions and Memory Use

In light of restricted computational power of an anonymous network, two key problems aroused the interest of researchers: what kinds of functions are guaranteed to be computable on such a network , and what is their complexity? In terms of computing a function h, each node v_i in an anonymous network receives an input value I_i as an initial configuration; later all nodes halt and reach a single final state, which is $h(I_1, I_2, \ldots, I_n)$.

The Boolean functions (the ones with $\{0, 1\}$-valued inputs and outputs) such as AND and SUM, and -specific functions like orientation (where all nodes agree on what is left/right consistently) are computable in a synchronous anonymous n-node cycle with a total of $O(n \log n)$ messages in the worst case; $O(n^2)$ messages are required under an asynchronous model [10]. The results remain valid even if nodes know the precise structure of the network but do not know their own locations within it. Later on, Attiya and Snir [9] proved that an asynchronous deterministic algorithm can compute any computable function with $O(n \log n)$ messages on average. The bit-complexity of computing Boolean functions on arbitrary anonymous networks is polynomial

$O(n^3\mathbb{D}\Delta^2 \log n)$ for computable Boolean functions; further, all symmetric functions (the ones with commutative property) can be computed based on under different activation models (synchronous, asynchronous and interleaved) with bit-complexity $O(n^2\mathbb{D}\Delta^2 \log^2 n)$ [72, p. 230]. As a significant milestone, Yamashita and Kameda [89] gave a characterization of asynchronous anonymous networks in which any arbitrary function is computable on an asynchronous anonymous network when each node is assumed to know the topology. Halpern and Petride [63] generalized the works of Attiya and Snir [9] and Yamashita and Kameda [89], and then borrowed a framework called the knowledge-based (kb) program, to solve global function computation on (edge-weighted) anonymous networks whenever possible.

In terms of memory use, Ando et al. [4] investigated space complexity of LE in anonymous networks, and proved that $\Omega(\Delta + \log c)$ bits are necessary, where c is the maximum size (in bits) of a message; $O(n \log c)$ bits are sufficient to solve LE on arbitrary anonymous n-node networks. While most of the literature about LE was concerned with time and message complexity, Fusco and Pelc [60] instead showed that the minimum memory size at nodes in an arbitrary anonymous network, whose nodes are identical automata, is logarithmic, which is optimal for two versions of LE. Datta et al. [37] proposed a self-stabilizing algorithm in a message-passing model for the weak LE in an anonymous network with $O(1)$ bits of memory per edge.

4.1.4 Advice

Advice in distributed computing is a binary string *a priori* shared by all nodes in a network as global information and the length of the string is the size of the advice; and the term *informative labelling schemes* is used instead when different nodes get different information [62, p. 6]. In most cases, there is a negative correlation between the size of the advice and the running time of the algorithm (called tradeoff), i.e., the more information given, the less allocated time needed. Following a characterization of four types of initial information about the attributes of an anonymous network [90], Sakamoto [81] built up a computational hierarchy for even more types of initial conditions given to all nodes and a transformation algorithm from one to another. Glacet et al. [62] figured out the tradeoff between time and information of LE when it comes to anonymous trees. More recently, Dieudonné and Pelc [42] aimed at establishing tradeoffs between the allocated time and the amount of information given as advice for an anonymous network where LE is actually possible to solve.

When it comes to the classic consensus problem, Fusco and Pelc [59] worked on its communication complexity instead: without knowing the topology structure, an optimal consensus algorithm on anonymous networks may use $O(n^2)$ messages; once the topology is known by all nodes (provided as advice), it drops to $O(n^{3/2} \log^2 n)$.

4.1.5 Mobile Agents

A **mobile entity/agent/robot/walker/automaton** is an autonomous movable object in a network. When there are multiple agents in a network, they are often considered to be identical (i.e., mutually indistinguishable). Since a mobile-agents algorithm and a traditional message-passing algorithm are proved to be computationally equivalent [11, 25] and Das et al. [35] built the computational equivalence between them by letting a mobile-agents algorithm simulate the (fault-tolerant) execution of a message-passing algorithm, it can be possibly extended on anonymous networks to the topology recognition, the naming problem, the spanning tree construction, etc. Even LE for anonymous asynchronous agents is possible [41].

Map drawing/construction, also called topology recognition (deterministically finding an isomorphic copy of it including port-labelling), by a mobile agent is infeasible unless the anonymous network is a tree, therefore extra advice is of necessity and its minimum size is $\Theta(ms_{\mathbf{f}})$ [40]; this result also holds even under the assumption that a stationary token is fixed at the starting node of the agent. In a model where each node contains a limited-size bulletin board, it is possible for p identical anonymous agents to construct a labelled map when $\gcd(n, p) = 1$ and the value of n or p is provided as advice [33].

The problem of searching and exploring an unknown environment is a principal problem with applications ranging from robot navigation to searching the Internet. In such a problem, a mobile agent has to visit all nodes and traverse all edges. Exploration of anonymous networks is possible in trees, but

otherwise is only possible if the agents are allowed to mark the nodes in some way [33]. In order to perform anonymous graph exploration, we have to allow non-terminating algorithms [44] or use a variant of the problem to ask it to go back to a marked starting position [22]. Reingold [80] proved that deterministic exploration of arbitrary anonymous networks can be performed in log-space with the help of universal traversal (exploration) sequences, while an agent is equipped with $O(\log n)$ bits of memory. If there are no marks/tokens left, then a mobile agent cannot achieve exploration and stop at the last node. A finite automaton using three states can perform a periodic graph exploration on a port-labelled network within $4n - 2$ rounds, independently of its starting position and initial state [67]. For an oblivious one (using only one state), the period is expanded up to $10n$ if right-hand-on-the-wall walk is involved and starting the exploration by the edge with port 1 is required [47].

The mobile-agents rendezvous problem (two anonymous mobile agents navigate synchronously in an anonymous network and have to meet at a node, using a deterministic algorithm) has been studied as a symmetry breaking task. The information gathered by a mobile agent traversing an anonymous graph, which does not have the ability to write to its environment, is simply a subtree of the view from its starting node. Hence, for example, rendezvous of deterministic agents is only possible if they start from positions with different views. For anonymous networks, rendezvous is deterministically achievable by two identical mobile agents when they are allowed to leave a token to mark the nodes they stand on currently, even when some tokens

may disappear unexpectedly and without knowing the topology [36]. The deterministic rendezvous problem is known to be not easier than graph exploration [32]: two identical anonymous mobile agents must be equipped with $\Theta(\log n)$ memory bits regardless of the delay between the starting times of the agents, in order to solve deterministic rendezvous on arbitrary feasible n-node networks. Another way of breaking symmetry, available even when agents are anonymous, is by exploiting either non-symmetries of the network itself, or the differences of the initial positions of the agents. It is known that agents can always meet if their initial positions are non-symmetric, and that if they are symmetric and agents start simultaneously then rendezvous is impossible. Even with arbitrary start-up times, and in the case where the agents have unique identifiers, rendezvous is solvable on any anonymous n-node tree with $O(n + \log \varsigma)$ steps for both agents, where ς is the smaller of the two identifiers [79], and Pelc also showed the optimal cost on cycles.

4.1.6 Families of Topology

Many papers started studying distributed computing problems by first considering fixed-sized anonymous cycles. Diks et al. [45] proved that the anonymous wireless and hardware cycles have the same computational power for Boolean functions. Flocchini et al. [52] solved the sorting and LE in anonymous asynchronous cycles.

Distance-regular graphs contains complete bipartite graphs, hypercubes etc. If anonymous networks are distance-regular, the bit-complexity of com-

puting Boolean functions on them is then improved to $O(n\mathbb{D}\Delta \log n)$ [72, pp. 232–234]; for an unoriented hypercube network, there is an algorithm with bit-complexity $O(n^3 \log^4 n)$ that computes Boolean functions [72, pp. 223–226]. There exists a broadcast algorithm, that works in anonymous hypercubes which use only $O(n)$ messages of size $O(\log n)$ [43].

There is no deterministic distributed verification algorithm on anonymous tori [82]. The bit-complexity of computing Boolean functions is $O(n\sqrt{n})$ on an anonymous $\sqrt{n}$-by-$\sqrt{n}$-torus [13]; with each node knowing the topology and the size of the torus as advice, there is a broadcast algorithm using $2n + O(\sqrt{n})$ messages and a lower bound of $1.04n - O(\sqrt{n})$ messages was given [46]. There also exists an LE algorithm using $\Theta(n)$ messages on both anonymous tori and chordal rings [73].

Every function on a ϱ-dimensional n-node grid/mesh with boundary connections can be computed with bit-complexity $O(n^{1+1/\varrho})$ [13]. Kranakis and Krizanc [71] presented a group-theoretic characterization of computable Boolean functions on an anonymous Cayley network (many network topologies, including hypercubes, cycles, tori, stars, pancakes, bubble-sort networks etc., can be obtained by different choices of groups and generating sets in Cayley graphs) and a complexity bound for its bit-complexity as well, along with an efficient algorithm.

4.2 Mathematical Techniques

In this section, we outline three previously-known tools for describing network symmetry, and we differentiate them from the concept of 'port-labellings' used in the remainder of this thesis.

4.2.1 Graph Automorphism

In graph theory, symmetry is often studied using graph automorphisms. The set of automorphisms characterizes the symmetry of a graph, seems related to the set of dissimilar views, as we defined previously.

Definition 4.1. An **automorphism** of a graph $G = (V, E)$ is a graph isomorphism with G itself, that is, a bijective mapping $\psi\colon V \to V$ such that the edge $(u, v) \in E$ for $u, v \in V$ iff the edge $(\psi(u), \psi(v)) \in E$. The set of G's all distinct automorphisms is denoted as $\mathrm{Aut}(G)$.

It appears that there are relationships between the symmetricity of a graph and its automorphism: for example, for any graph G with $\sigma(G) = 2$, there exists a fixed-point-free automorphism on G. They are however quite independent, since graph automorphism only considers vertex-labelled graphs with no port labels. The following two propositions were proved by Yamashita and Kameda [90, p. 86], and these propositions show that graph symmetry according to symmetricity $\sigma(G)$ can be rather different from $|\mathrm{Aut}(G)|$.

Proposition 4.1. *For any star graph S_n[3] with at least three vertices in total,*

[3] A star graph S_n is a tree of n vertices with one vertex having degree $n - 1$ and the

$\sigma(S_n) = 1$. *Meanwhile,* $|Aut(S_n)| = n!$ *for* $n \geq 2$.

Definition 4.2. A **bridge** is a single edge whose deletion disconnects a connected graph, and increases the number of its connected components[4]. Equivalently, an edge is a bridge iff it is not contained in any cycle. A graph is **bridgeless** if it contains no bridges.

Proposition 4.2. *For any cubic bridgeless graph* G *with* n *vertices, we have* $\sigma(G) = n$*; however, there exists a cubic bridgeless graph* G' *having* $|Aut(G')| = 1$.

Example 4.1. The Frucht graph G in Figure 4.1, cubic and bridgeless, has $|\mathrm{Aut}(G)| = 1$ but $\sigma(G) = 12$, which instantiates Proposition 4.2.

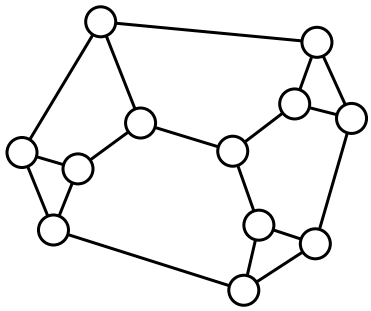


Figure 4.1: The Frucht graph[5]

others having degree 1.

[4] A connected component is a maximal connected subgraph.

[5] The drawing was originally plotted by Dr. David Eppstein in the Wikipedia entry *Frucht graph* and modified to fit the thesis style.

4.2.2 Edge Labellings with a Sense of Direction

Informally speaking, a **sense of direction** (SoD) means that each vertex v can determine whether or not two different walks[6] starting at v (given by sequences of port labels) end at the same vertex. The formal definition of SoD and the classification of four different classes of that were first posed by Flocchini et al. [53] and the group-theoretic alternative definition was provided by Tel [85, p. 54]. It is a helpful global property since a network with SoD is strictly more powerful than a network where each vertex is given the entire topology as advice [56]. An edge labelling of an underlying graph based on a SoD bears a resemblance to our port-labelling. Such an edge labelling is **minimal** if it only uses in total Δ different ports for all vertices [54, p. 30].

Example 4.2. The first three labellings in Figure 4.2 all possess a SoD. The leftmost edge labelling is a legal but not minimal edge labelling with a SoD. It uses eight different ports, ranging from 1 to 8, while $\Delta = 2$. The second and third one are both minimal edge labellings, although the third one is also a total port-labelling with multiplicity 1 as well. The major difference between them is that for an edge labelling, its labelling function can be an *injective* mapping instead of a bijective one, a port on a vertex v larger than $\deg(v)$ is allowed. Compared to that, ours are much stricter and offers less flexibility in terms of distinguishing vertices from one another in an anonymous network. In the context of 'backward SoD', computationally

[6] A walk in a graph is a finite alternating sequence of edges where the endpoint of one edge is the starting point of the next edge.

equivalent to SoD [55], in such an edge labelling, some vertices can have duplicated ports, as demonstrated in the rightmost one.

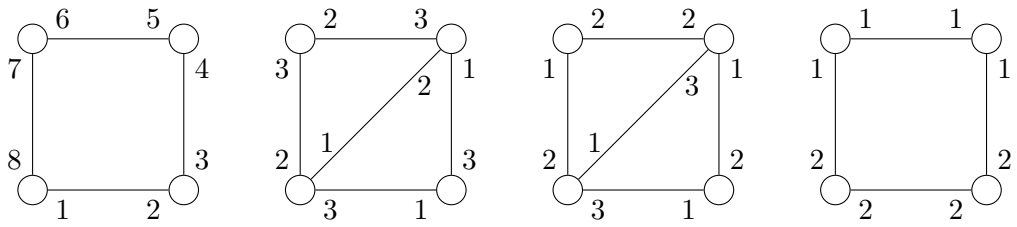


Figure 4.2: All four edge labellings with a SoD

4.2.3 Incidence Colouring

In graph theory, incidence colouring was introduced by Brualdi and Massey [18], which seems to have connections with what we study in the thesis but is based on different motivations.

Definition 4.3. An **incidence** of a graph $G = (V, E)$ is defined as a pair $\langle v, e\rangle$ where $v \in V$ is an endpoint of $e \in E$. Two incidences $\langle v_1, e_1\rangle$ and $\langle v_2, e_2\rangle$ are said to be **neighbouring** if one of the following holds:

1. $v_1 = v_2$ and $e_1 \neq e_2$;
2. $e_1 = e_2$ and $v_1 \neq v_2$;
3. $e_1 = (v_1, v_2)$, $e_2 = (v_1, v_3)$ and $v_2 \neq v_3$.

Example 4.3. Figure 4.3 contains configurations of neighbouring incidences associated with the three cases of Definition 4.3, in order from left to right in order. A star beside an edge closest to a vertex represents an incidence.

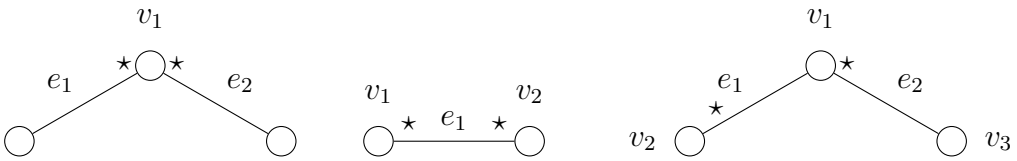


Figure 4.3: The configurations of three neighbouring incidences

Definition 4.4. Let $\iota(G)$ be the set of all incidences of a graph G. An i-**incidence colouring** of G is a function mapping $\iota(G)$ to $\{ 1, 2, \ldots, i \}$ such that each neighbouring incidence is mapped to a distinct value.

Example 4.4. Figure 4.4 validates a legal 5-incidence colouring (left) and a port-labelling (right) of the Petersen graph. Noticeably, the labels whose ends are vertex v (filled in black) do not necessarily start from 1 and labels larger than $\deg(v)$ are allowed in such a colouring. According to Definition 4.3, any edge here cannot have the same label on its two ends. The vertex v' (filled in black) in the right-hand graph is incident to 3 edges with the same ports on their both ends, which is different from its counterpart in the left-hand graph.

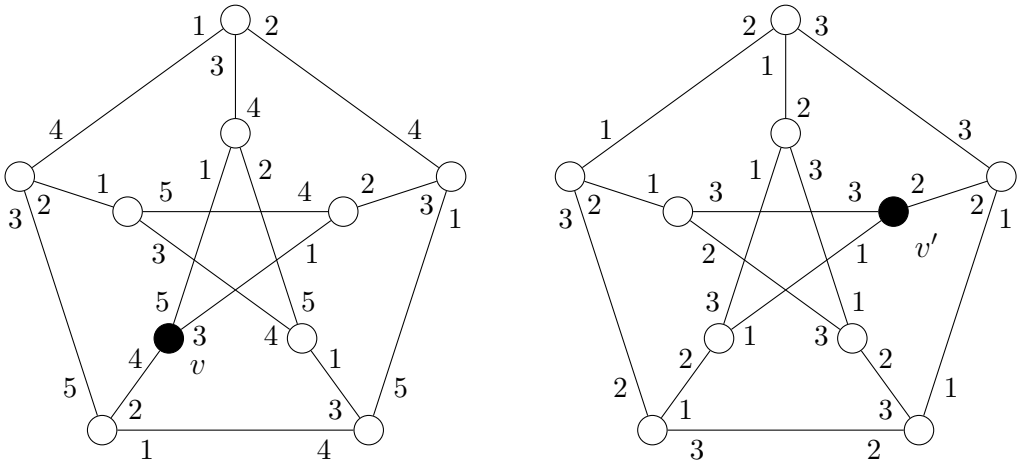


Figure 4.4: A legal 5-incidence colouring and a port-labelling on the Petersen graph

5

Producing Symmetry

Tyger Tyger burning bright,
In the forests of the night:
What immortal hand or eye,
Dare frame thy fearful
symmetry?

The Tyger
William Blake

The goal of this chapter is to devise a method to port-label graphs to produce a desired multiplicity. In Section 5.2, we demonstrate how to port-label any $\{1, 2\}$-factorable n-vertex graph so that the resulting multiplicity is n. Next, in Section 5.3, we provide an example of a regular graph with symmetricity n such that multiplicity d is not attainable for some $d \mid n$. In Section 5.4, we solve Problem 1 by showing, for any $d \mid n$, how to port-label the complete graph with n vertices such that the resulting multiplicity is d. Note that this immediately provides a solution to Problems 2 and 3. Finally,

in Section 5.5, we provide a method based on rooted graph products for generating additional examples that solve Problems 2 and 3.

5.1 Labelling an Anonymous Network

In this section, we discuss the process of taking an arbitrary graph G and assigning a labelling. One approach is to label ports from a *local* perspective. Each vertex v assigns port labels to each incident edge with a distinct integer from $\{ 1, 2, \ldots, \deg(v) \}$ and without any knowledge about the graph other than its own degree, which generates an arbitrary labelling. Another approach, which we will take in this thesis, is to label ports from a *global* perspective. To avoid ambiguity when describing the labelling of a graph, we first assign a unique identifier in the range $\left\{ v_1, v_2, \ldots, v_{|V|} \right\}$ to each vertex. This allows us to fully represent the labelling using a matrix (we elaborate on this below). After the port labels have been assigned, the graph is made an anonymous network by simply removing the vertex labels, or, by blocking each vertex's access to its own vertex label. Figure 5.1 demonstrates the above process.

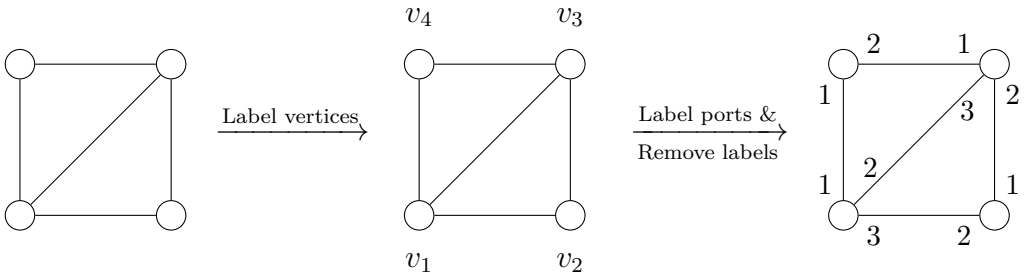


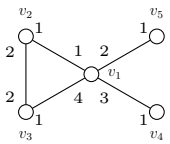
Figure 5.1: Creating an anonymous network

Definition 5.1. The **adjacency matrix** of a graph is a square $(0,1)$-matrix with rows and columns labelled by vertices, where 1 is in row v_i and column v_j when pairs of v_i and v_j are adjacent, and 0 when there is no edge between v_i and v_j.

Extending the adjacency matrix idea, we define the labelling matrix of a labelling on a graph.

Definition 5.2. The **labelling matrix** of a labelling on a graph is a square matrix with rows and columns labelled by vertices, where an outgoing port p on the edge from v_i to v_j (i.e. $p = f_{v_i}(v_j)$) is in row v_i and column v_j, and 0 if no edge exists.

Example 5.1. In Table 5.1, we provide the adjacency matrix of the vertex-labelled graph G in Figure 2.2 on page 12 (reproduced here as a margin note), which is symmetric. In Table 5.2, we give the corresponding labelling matrix, which we note is asymmetric.



Vertex v_i \ **Vertex** v_j	v_1	v_2	v_3	v_4	v_5
v_1	0	1	1	1	1
v_2	1	0	1	0	0
v_3	1	1	0	0	0
v_4	1	0	0	0	0
v_5	1	0	0	0	0

Table 5.1: The adjacency matrix of the vertex-labelled graph G in Figure 2.2

The labelling matrix on its own is sufficient to completely describe an anonymous network, which makes them useful in our algorithm implementa-

Vertex v_i \ **Vertex** v_j	v_1	v_2	v_3	v_4	v_5
v_1	0	1	4	3	2
v_2	1	0	2	0	0
v_3	1	2	0	0	0
v_4	1	0	0	0	0
v_5	1	0	0	0	0

Table 5.2: The labelling matrix (representing the labelling **f**) in Figure 2.2

tions. Both the adjacency matrix and the labelling matrix of a vertex-labelled graph with n vertices are treated as n-by-n `arrays` in all the algorithms of our work. Although a labelling is a set containing local labelling functions by Definition 2.6, we implement it using a labelling matrix. The notation $f_{v_i}(v_j)$ in the code is equivalent to accessing the entry in row v_i and column v_j of a labelling matrix.

5.2 {1, 2}-Factorable Graphs Under a Fully Symmetric Labelling

If a total labelling on a graph G with n vertices has multiplicity n, then we say that it is **fully symmetric**. In this section, our goal is to provide an algorithm that will label the ports of any $\{1, 2\}$-factorable graph such that its multiplicity is n, or in a fully symmetric way. According to Corollary 2.1, a fully symmetric labelling can be achieved only on a $\{1, 2\}$-factorable graph, so the remainder of this section focuses only on such graphs.

In the early stages of our work, we conjectured that all regular graphs

could be port-labelled in a fully symmetric way, but it turns out that this is not the case. This motivated us to investigate the relationship between regular graphs and $\{1, 2\}$-factorable graphs. A $\{1, 2\}$-factorable graph must be regular (since it is the sum of 1-regular and 2-regular spanning subgraphs), but a regular graph is not necessarily $\{1, 2\}$-factorable: the example in Figure 5.2 is the smallest cubic graph without 1-factors or 2-factors [16, p. 417; 87], thus being not $\{1, 2\}$-factorable.

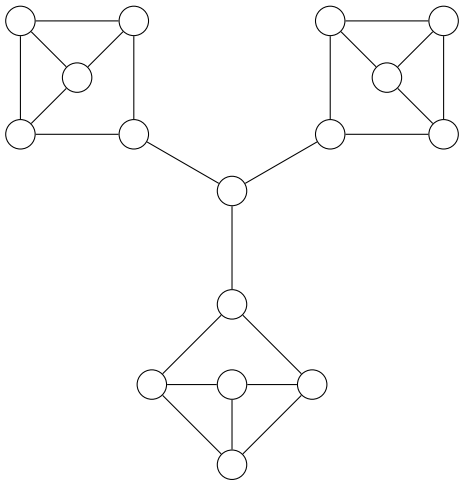


Figure 5.2: An example of a cubic graph that is not $\{1, 2\}$-factorable

For a regular graph of even degree, it is $\{1, 2\}$-factorable owing to Theorem 5.1 below.

Theorem 5.1 (Petersen's 2-Factor Theorem (1891))**.** *A graph is 2-factorable iff it is regular of even degree.*

When considering a regular graph with odd degree, whether or not it is $\{1, 2\}$-factorable solely depends on the existence of one 1-factor.

Corollary 5.1. *For a k-regular graph G of odd degree, G has i edge-disjoint 2-factors iff G has a 1-factor, where $k = 2i + 1$ and $i \geq 1$.*

Proof. The proof consists of two parts.

The "if" part. We remove such a 1-factor from G to obtain a new graph G', decreasing each vertex's degree by one. Now G' is regular of even degree, thus being 2-factorable according to Theorem 5.1.

The "only if" part. We remove i edge-disjoint 2-factors from G, making all the vertices become degree 1 — the leftover part must be a 1-factor.

■

Table 5.3 defines a partition of all k-regular graphs into four divisions, based on the existence of 1-factors and the parity of their degree. All the graphs from division I, II and III comprise $\{1, 2\}$-factorable graphs and Figure 5.2 on page 47 is an example of division IV. Besides, a regular graph with an odd n (and an even k) belongs to the set of graphs in division II since 1-factors can only exist in a graph with an even n by definition; a complete graph K_n with an odd n is clearly one of them. The graphs from division I and II altogether are equivalent to 2-factorable ones in light of Theorem 5.1. On the other hand, the 1-factorable graph is a proper subset of division I and III. For instance, a complete graph K_n with an even n (and an odd k) falls into division III and is 1-factorable; in contrast, the Petersen graph is in division III but not 1-factorable. A cycle graph C_n with an even n (and

an even k) is 1-factorable from division I. There also exist non-1-factorable k-regular n-vertex graphs with at least one 1-factor (see the construction in Appendix A).

Degree \ **1-Factor**	With	Without
Even	I	II
Odd	III	IV

Table 5.3: Four divisions of all regular graphs

The function IF12FACTORABLE from Algorithm 5.1 is used to check whether a regular graph is $\{1, 2\}$-factorable or not (applied later in Algorithm 7.4 from Section 7.1): if each vertex has the same degree and the degree of an arbitrary one is even, then it is 2-factorable (by Theorem 5.1); if its degree is odd, we then examine whether the graph G contains at least one 1-factor (according to Corollary 5.1). For the existence of 1-factors, we apply the function IFCONTAIN1FACTOR (in line 6), which is a modified matching algorithm implemented by Micali and Vazirani with time complexity $O(m\sqrt{n})$ [76] (their algorithm was also applied in the proof of Theorem 2.2).

Algorithm 5.1 Check if a regular graph is $\{1, 2\}$-factorable

Require: G is a regular graph

```
1: function IF12FACTORABLE(G)
2:     d ← the degree of an arbitrary vertex in V_G
3:     if d mod 2 = 0 then
4:         return True
5:     else
6:         if IFCONTAIN1FACTOR(G) then
```

```
7:            return True
8:        else
9:            return False
10:       end if
11:   end if
12: end function
```

To conclude this section, we provide a systematic approach in Algorithm 5.4 to generate a fully symmetric labelling on any $\{1, 2\}$-factorable graph with two helper functions from Algorithms 5.2 and 5.3.

The function PortLabel1Factor in Algorithm 5.2 assigns the two ports of each edge in a 1-factor G_F of a graph G to integer p that was provided as a parameter. In most cases, G_F is disconnected, but it is still passed as one adjacency matrix. Line 6 writes p into the entry in row v_i and column v of $\mathfrak{f}$.

Algorithm 5.2 Do a partial labelling on a 1-factor of a graph

Require: G_F is a 1-factor of a graph G and $1 \leq p \leq \Delta_G$

```
1: function PortLabel1Factor(G_F, p)
2:     n ← |V_{G_F}|
3:     f ← an empty 2-dimensional n-by-n array
4:     for all v_i ∈ V_{G_F} do
5:         v ← the only neighbour of v_i          ▶ deg(v_i) = 1
6:         f_{v_i}(v) ← p
7:     end for
8:     return f
9: end function
```

The second helper function is the function PortLabel2Factor in Algorithm 5.3. After it is invoked with parameters G_F and p, all the edges

of a 2-factor G_F are port-labelled with p and $p+1$ at two ends respectively. Again, a 2-factor of a graph may contain several disconnected cycles. The variable *traversed* in line 2 is initialized to be empty (represented by the symbol $\varnothing$) and keeps tracking the vertices whose ports were assigned already. The code in lines 10 to 13 updates the values in $\mathfrak{f}$ and is exemplified in Figure 5.3, where the arrows indicate the labelling orientation in the next iteration and vertex v_i (in black) is both the starting point and later the ending point. In line 14, the new variable v_s is the candidate for the next round of the labelling. After a new edge is port-labelled, we update v_s with the waiting-for-port-labelling neighbour in line 21. When the **break** statement in line 22 is encountered, the **for** loop is terminated immediately and the control of the program is passed to the first statement after the loop, which is line 24.

Algorithm 5.3 Do a partial labelling on a 2-factor of a graph

Require: G_F is a 2-factor of a graph G and $1 \leq p \leq \Delta_G$

1: **function** PortLabel2Factor(G_F, p)
2: $\quad$ *traversed* $\leftarrow \varnothing$
3: $\quad$ $n \leftarrow |V_{G_F}|$
4: $\quad$ $\mathfrak{f} \leftarrow$ an empty 2-dimensional n-by-n `array`
5: $\quad$ **for all** $v_i \in V_{G_F}$ **do**
6: $\quad\quad$ **if** $v_i \notin$ *traversed* **then** ▶ A new cycle subgraph appears
7: $\quad\quad\quad$ *traversed* $\leftarrow$ *traversed* $\cup \{ v_i \}$
8: $\quad\quad\quad$ $v' \leftarrow$ one neighbour of v_i
9: $\quad\quad\quad$ $v'' \leftarrow$ the other neighbour of v_i ▶ $\deg(v_i) = 2$
10: $\quad\quad\quad$ $f_{v_i}(v') \leftarrow p$
11: $\quad\quad\quad$ $f_{v'}(v_i) \leftarrow p+1$
12: $\quad\quad\quad$ $f_{v''}(v_i) \leftarrow p$

13: $f_{v_i}(v'') \leftarrow p + 1$
14: $v_s \leftarrow v'$
15: **while** $v_s \notin$ *traversed* **do**
16: $traversed \leftarrow traversed \cup \{ v_s \}$
17: **for all** $v_j \in N_{G_F}(v_s)$ **do**
18: **if** $v_j \notin$ *traversed* **then**
19: $f_{v_s}(v_j) \leftarrow p$
20: $f_{v_j}(v_s) \leftarrow p + 1$
21: $v_s \leftarrow v_j$
22: **break**
23: **end if**
24: **end for**
25: **end while**
26: **end if**
27: **end for**
28: **return** $\mathfrak{f}$
29: **end function**

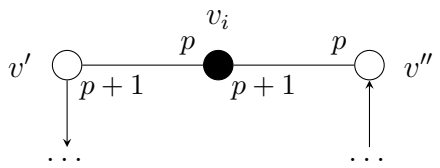


Figure 5.3: A visualization of lines 10 to 13 of Algorithm 5.3

With two building blocks above, we then set up an algorithm for doing a fully symmetric labelling on a $\{1, 2\}$-factorable graph. The parameter p in Algorithm 5.4 line 1 is the smallest port to use in the labelling. When constructing a fully symmetric labelling for a $\{1, 2\}$-factorable graph G, we would call the FullySymmetricLabel function with parameter $p = 1$. The

function FACTORIZEGRAPH (by Meijer et al. [75]) in line 5 takes the graph G as an argument and decomposes it into a `list` of 1-factors and 2-factors deterministically.

Algorithm 5.4 Do a fully symmetric labelling on a $\{1, 2\}$-factorable graph

Require: G is a vertex-labelled $\{1, 2\}$-factorable graph

```
1:  function FullySymmetricLabel(G, p)
2:      n ← |V_G|
3:      f ← an empty n-by-n array
4:      factorization ← an empty list
5:      factorization ← FactorizeGraph(G)
6:      for all G_{F_i} ∈ factorization do
7:          v ← an arbitrary vertex in V_{G_{F_i}}
8:          if deg(v) = 1 then                    ▶ G_{F_i} is a 1-factor
9:              f ← PortLabel1Factor(G_{F_i}, p)
10:             p ← p + 1
11:         else                                  ▶ G_{F_i} is a 2-factor
12:             f ← PortLabel2Factor(G_{F_i}, p)
13:             p ← p + 2
14:         end if
15:     end for
16:     return f
17: end function
```

5.3 The Multiplicity Gap for the Petersen Graph

Here we use the Petersen graph to show that Problem 1 is not trivial, since there exists at least one n-vertex graph such that for some $d \mid n$, d cannot be its valid multiplicity.

The Petersen graph is a $\{1, 2\}$-factorable cubic graph with 10 vertices, and naturally a fully symmetric labelling $\mathbf{f}$ can be done on it by the function FullySymmetricLabel from Algorithm 5.4, making $s_{\mathbf{f}} = 10$. The multiplicity of a labelling on it can also be 5, as illustrated in Figure 5.4 (vertices with the same colour have similar views). However, as the next result shows, there is no labelling for the Petersen graph that results in multiplicity 2.

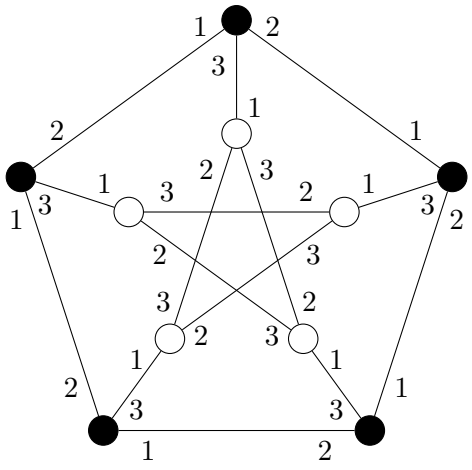


Figure 5.4: A labelling on the Petersen graph whose multiplicity is 5

Proposition 5.1. *For any labelling $\boldsymbol{f}$ on the Petersen graph, $s_{\boldsymbol{f}} \neq 2$.*

Proof by contradiction. Suppose that there are five dissimilar views in total ($s_{\mathbf{f}} = 2$), and each view is held by two vertices. The Petersen graph has diameter 2, so the two vertices with the similar views are within distance 2 from each other.

Case 1. There exists a view $T_{\mathbf{f}}$ such that two vertices with view $T_{\mathbf{f}}$ are exactly distance 2 away from each other. Let v_1 and v_2 be the vertices with view $T_{\mathbf{f}}$, and let u_1 be a vertex with v_1 and v_2 as neighbours. Let $T'_{\mathbf{f}}$ be the view of u_1 and $T_{\mathbf{f}} \not\equiv T'_{\mathbf{f}}$. Then the other vertex u_2 with view $T'_{\mathbf{f}}$ also must have two neighbours with view $T_{\mathbf{f}}$. This gives a 4-cycle $(v_1, u_1, v_2, u_2, v_1)$. However, the Petersen graph does not contain a 4-cycle, on the grounds that the Petersen graph has girth[1] 5 [16, p. 44].

Case 2. For each view, the two vertices with similar views are adjacent to each other. Consider the vertices v_1 and v_2 with view $T_{\mathbf{f}}$ adjacent to each other. Since each vertex has degree 3, the vertex v_1 has another neighbour u_1 that does not have view $T_{\mathbf{f}}$. We assume that u_1 has view $T'_{\mathbf{f}}$. The other vertex u_2 with view $T'_{\mathbf{f}}$ is adjacent to u_1, by assumption. Since u_1 has a neighbour with view $T_{\mathbf{f}}$, u_2 must also have a neighbour with view $T_{\mathbf{f}}$.

Subcase (i): v_1 is adjacent to u_2. This gives a 3-cycle (v_1, u_1, u_2, v_1).

Subcase (ii): v_2 is adjacent to u_2. This gives a 4-cycle $(v_1, u_1, u_2, v_2, v_1)$.

Both of these subcases are impossible because the Petersen graph has girth 5. A contradiction occurs in all cases, which proves the desired result.

■

[1] The girth of a graph is the length of a shortest cycle.

5.4 Port-Labelling Complete Graphs

In this section, we provide an algorithm to show that for an n-vertex complete graph and any $d \mid n$, all possible divisors d can become its multiplicity, and hence solve Problem 1.

Definition 5.3. A **complete graph** K_n is a graph with n vertices where every pair of vertices is connected by an edge. A **complete bipartite graph** $K_{a,b}$ is a bipartite graph $(U \cup V, E)$ with $|U| = a$ and $|V| = b$ such that for every two vertices $u \in U$ and $v \in V$, $(u, v) \in E$.

If $a = b$, the complete bipartite graph $K_{a,b}$ is regular.

5.4.1 Factorizations of Complete Graphs

Our approach will make use of the following theorems about factorizations of complete graphs from Harary [64, pp. 84–90]. The first two was attributed to Walecki and published by Lucas according to Alspach [3].

Theorem 5.2. *Every complete graph K_n with an even n is 1-factorable and the sum of a 1-factor and $n/2 - 1$ spanning cycles C_n.*

Theorem 5.3. *Every complete graph K_n with an odd n is the sum of $(n-1)/2$ spanning cycles C_n.*

The core idea of Algorithm 5.7 in Subsection 5.4.2 heavily relies on Theorem 5.4.

Theorem 5.4 (Kőnig's Theorem (1931))**.** *Each regular bipartite graph is 1-factorable.*

In summary, every complete graph K_n with an even n is 1-factorable (by Theorem 5.2) and every complete graph K_n with an odd n is 2-factorable (by Theorem 5.3), which indicates that the complete graph K_n admits a $\{1, 2\}$-factorization. Together with Corollary 2.1, it implies that its symmetricity is always n, i.e. $\sigma(K_n) = n$. Similarly, by Theorem 5.4 and Corollary 2.1, every regular bipartite graph has symmetricity n.

5.4.2 An Algorithm for Producing a Desired Multiplicity

Based on factorization facts in the last subsection, we provide a class of graphs to solve Problem 1 by coming up with Algorithm 5.7, before two helper functions from Algorithms 5.5 and 5.6 are explained.

Algorithm 5.5 contains a function of returning two neighbours of a vertex v in a graph G under a labelling $\mathbf{f}$ with the minimum and maximum ports, which requires $\deg(v) \geq 2$. It examines all the arbitrarily ordered ports on v's end one by one until the neighbours on the edges whose v's end is port-labelled with the minimum and maximum ones are found.

Algorithm 5.5 Return two neighbours of a vertex in a graph with the minimum and maximum ports

Require: $v \in V_G$ and $\deg(v) \geq 2$

1: **function** ReturnTwoNeighbours($\langle G, \mathbf{f} \rangle, v$)

```
2:      counter ← 0
3:      for all v_i ∈ N_G(v) do
4:          if f_v(v_i) = 1 then
5:              v_min ← v_i
6:              counter ← counter + 1
7:              if counter = 2 then
8:                  return v_min, v_max
9:              end if
10:         else if f_v(v_i) = deg(v) then
11:             v_max ← v_i
12:             counter ← counter + 1
13:             if counter = 2 then
14:                 return v_min, v_max
15:             end if
16:         end if
17:     end for
18: end function
```

In Algorithm 5.6, the function SwitchPorts first finds the minimum and maximum possible outgoing ports (namely 1 and $\deg(v)$) for each vertex $v \in V'$ and their corresponding neighbours, respectively $v_{\min}$ and $v_{\max}$. We call the function ReturnTwoNeighbours in Algorithm 5.6 line 3 from Algorithm 5.5. Then it makes the port of v_i on the edge $(v_i, v_{\min})$ become $\deg(v)$ and that of v_i on the edge $(v_i, v_{\max})$ become 1. In lines 4 and 5, the previously-stored values in $\mathbf{f}$ are substituted with the newly-assigned ones. In what follows, this operation will be referred to as *port-switching*.

Algorithm 5.6 Switch the minimum and maximum ports of each vertex from a vertex subset

Require: $V' \subseteq V_G$
1: **function** SwitchPorts($\langle G, \mathbf{f} \rangle, V'$)
2: **for all** $v_i \in V'$ **do**
3: $v_{\min}, v_{\max} \leftarrow$ ReturnTwoNeighbours($\langle G, \mathbf{f} \rangle, v_i$)
4: $f_{v_i}(v_{\min}) \leftarrow \deg(v_i)$
5: $f_{v_i}(v_{\max}) \leftarrow 1$
6: **end for**
7: **return f**
8: **end function**

There is one more definition and its corollary as prerequisite knowledge before the introduction of Algorithm 5.7.

Definition 5.4. A **clique** of a graph is its complete induced subgraph. More precisely, for a graph $G = (V, E)$ and a specific subset of vertices $V' \subseteq V$, a clique is a subgraph $G' = (V', E')$ such that $E' = \{ (v_i, v_j) \mid v_i, v_j \in V' \}$, where $v_i \neq v_j$ and $E' \subseteq E$.

The following lemma can be treated as a special case of the factorization condition clarified in Definition 2.22 as well.

Lemma 5.1. *Consider any $n \geq 3$, any $d > 1$ such that $d \mid n$, and let $g = n/d$. The complete graph K_n can be decomposed into g d-vertex cliques and $\binom{g}{2}$ complete bipartite induced subgraphs.*

Proof. We partition the vertex set $\{ v_1, v_2, \ldots, v_n \}$ into g disjoint subsets of size d and each one becomes the vertex set of an edge-disjoint clique G_i

$(1 \leq i \leq g)$:

$$V_{G_1} = \{\, v_1, v_2, \ldots, v_d \,\},$$

$$V_{G_2} = \{\, v_{d+1}, v_{d+2}, \ldots, v_{2d} \,\},$$

$$\ldots$$

$$V_{G_i} = \left\{\, v_{(i-1)d+1}, v_{(i-1)d+2}, \ldots, v_{id} \,\right\},$$

$$\ldots$$

$$V_{G_g} = \left\{\, v_{(g-1)d+1}, v_{(g-1)d+2}, \ldots, v_{gd} \,\right\}.$$

Next, each complete bipartite induced subgraph $G_{i,j}$ is formed by the vertex sets of two different cliques and all the possible edges between these two, i.e., $G_{i,j} = (V_{G_i} \cup V_{G_j}, \left\{\, (v_i, v_j) \;\middle|\; v_i \in V_{G_i}, v_j \in V_{G_j} \,\right\})$, for $i, j \in \{\, 1, 2, \ldots, g \,\}$ and $i \neq j$. Finally, consider any two vertices in $\{\, v_1, v_2, \ldots, v_n \,\}$. If they are in the same G_i, then there is an edge between them in G_i. Otherwise, they belong to two different G_i and G_j, in which case there is an edge between them in $G_{i,j}$. This proves that the union of all G_i and $G_{i,j}$ forms K_n. ■

At last, we claim that a complete graph under a specially-designed total labelling is always an answer for Problem 1 by presenting the following algorithm. The function PortLabelKn in Algorithm 5.7 below involves three steps. According to Lemma 5.1, we can always decompose a complete graph K_n into g d-vertex cliques G_i and $\binom{g}{2}$ complete bipartite induced subgraphs $G_{i,k}$, for $g = n/d$. The first step is to do a fully symmetric labelling on these cliques one by one. As the second step, all $G_{i,k}$ are port-labelled, and the result is a labelling of K_n with multiplicity n. In the final step, we

select the vertex set of the clique G_0, and port-switch each vertex in V_{G_0} such that only these d vertices have similar view $T'_{\mathbf{f}}$, and none of the other $n-d$ vertices have a view similar to $T'_{\mathbf{f}}$. Based on Proposition 2.1, the multiplicity for the new total labelling must be d.

We now provide a more detailed explanation of how each step is implemented in Algorithm 5.7. For Step 1, the variable K_n in line 2 is represented as an n-by-n `array`. In line 4, $M_{\mathbb{N}_0:\mathbb{V}}$ maps the clique indices to $\mathbb{V}$, where $\mathbb{N}_0$ is the set of natural numbers including 0 and $\mathbb{V}$ is the power set[2] of V_{K_n}; we store g vertex subsets in $M_{\mathbb{N}_0:\mathbb{V}}$ to build g cliques $G_0, G_1, \ldots, G_{g-1}$. The integers ranging from 0 to $g-1$ (not from 1 to g) in line 6 are used to represent g indices of cliques, simply because it is much easier to apply the modulo operation appeared later in line 16 of Step 2. From line 11 onward, $M_{\mathbb{N}_0:\mathbb{V}}[i] = V_{G_i}$. Line 12 labels all the ports within each clique using a fully-symmetric labelling using ports $1, 2, \ldots, d-1$.

Step 2 is to port-label all the leftover edges. For each i in line 14, a complete bipartite induced subgraph $G_{i,k}$ is formed by two vertex subsets V_{G_i} itself and V_{G_k}, where the variable k in line 16 is acted as a "wrap-around" to return the values from $g-1$ indices in order, namely $i+1, i+2, \ldots, g-1, 0, 1, \ldots, i-1$ of $M_{\mathbb{N}_0:\mathbb{V}}$. For line 19, we assume that the function FactorizeGraph returns a `list` of d 1-factors by factorizing $G_{i,k}$, which is possible due to Theorem 5.4. The variable *unused* in line 20 is the smallest unused port during the process. Since the ports up to $d-1$ are unavailable after line 12, *unused* is

[2] A power set of a set is the set of all its subsets including $\varnothing$ and the original set itself.

initialized to d. Unlike the function PortLabel1Factor from Algorithm 5.2, lines 22 to 25 only label the ports whose ends are the vertices from V_{G_i} within each iteration on the variable i in line 14.

The reason why we cannot just get rid of the first two steps by directly calling the function FullySymmetricLabel from Algorithm 5.4 on a complete graph K_n is that it is nearly impossible to then pick d vertices out of n ones and do port-switching on them without affecting others' views correspondingly, since FullySymmetricLabel is not designed to do so and views are interdependent in most cases.

In Step 3, each vertex $v_i \in V_{G_0}$ is port-switched — the maximum and minimum ports are exchanged such that each vertex's view is dissimilar from the ones outside V_{G_0} up to level 1 of their view, i.e., $T^1_{\mathbf{f}}(v_i) \not\equiv T^1_{\mathbf{f}}(v_j)$ for $v_j \in \bigcup_{j=1}^{g-1} V_{G_j}$.

Algorithm 5.7 Port-label a complete graph totally whose multiplicity is d

Require: $n \geq 3$, $d \mid n$ and $1 < d < n$

1: **function** PortLabelKn(n, d)
2: $K_n \leftarrow$ a vertex-labelled complete graph with n vertices
3: $\mathbf{f} \leftarrow$ an empty 2-dimensional n-by-n `array`
4: $M_{\mathbb{N}_0:\mathbb{V}} \leftarrow$ an empty `dictionary`
5: $g \leftarrow n/d$
▶ Step 1: port-label cliques one by one
6: **for** $i \leftarrow 0$ **to** $g - 1$ **do**
7: $M_{\mathbb{N}_0:\mathbb{V}}[i] \leftarrow \varnothing$
8: **for** $j \leftarrow 1$ **to** d **do**

9: $M_{\mathbb{N}_0:\mathbb{V}}[i] \leftarrow M_{\mathbb{N}_0:\mathbb{V}}[i] \cup \{ v_{id+j} \}$ ▶ The final value of $M_{\mathbb{N}_0:\mathbb{V}}[i]$ will be $\{ v_{id+1}, \ldots, v_{id+d} \}$

10: **end for**

11: $G_i \leftarrow$ a clique of K_n with vertex set $M_{\mathbb{N}_0:\mathbb{V}}[i]$

12: $\mathbf{f} \leftarrow$ FullySymmetricLabel$(G_i, 1)$

13: **end for**

▶ Step 2: port-label complete bipartite induced subgraphs

14: **for** $i \leftarrow 0$ **to** $g - 1$ **do**

15: **for** $j \leftarrow 1$ **to** $g - 1$ **do**

16: $k \leftarrow (i + j) \bmod g$

17: $G_{i,k} \leftarrow$ a complete bipartite induced subgraph of K_n with parts V_{G_i} and V_{G_k}

18: *factorization* $\leftarrow$ an empty `list`

19: *factorization* $\leftarrow$ FactorizeGraph$(G_{i,k})$

20: *unused* $\leftarrow d$

21: **for all** $G_{F_x} \in$ *factorization* **do**

22: **for all** $v_y \in V_{G_{F_x}} \cap V_{G_i}$ **do**

23: $v \leftarrow$ the only neighbour of v_y

24: $f_{v_y}(v) \leftarrow$ *unused*

25: **end for**

26: *unused* $\leftarrow$ *unused* $+ 1$

27: **end for**

28: **end for**

29: **end for**

▶ Step 3: port-switch all vertices in G_0 and return the result

30: $\mathbf{f} \leftarrow$ SwitchPorts$(\langle K_n, \mathbf{f} \rangle, V_{G_0})$

31: **return f**
32: **end function**

Algorithm 5.7 is unable to handle two special cases: $d = n$ and $d = 1$. We can produce these multiplicities using the following approaches. If $d = n$, we just need to invoke the function FullySymmetricLabel from Algorithm 5.4 solely with parameters K_n and port 1, which returns a port-labelled graph under a fully symmetric labelling where all the vertices sharing similar views. If $d = 1$, for a complete graph under a fully symmetric labelling (created also by FullySymmetricLabel), we port-switch any vertex v in such a graph by calling SwitchPorts on $\{ v \}$, giving rise to v with the unique view among others; then its multiplicity drops from n to 1 instantly.

The example below visualizes the total labelling (Steps 1 and 2) and the subsequent port-switching (Step 3) described above.

Example 5.2. For $n = 12$, $d = 4$ and $g = n/d = 3$, there must exist four vertices sharing similar views after the function PortLabelKn is executed. We let v_1, v_2, v_3, v_4 be such vertices. G_0 is a clique of the complete graph K_{12} with vertex set $\{ v_1, v_2, v_3, v_4 \}$; similarly, G_1 is a clique with vertex set $\{ v_5, v_6, v_7, v_8 \}$ and G_2 is a clique with vertex set $\{ v_9, v_{10}, v_{11}, v_{12} \}$. These three cliques are marked red in Figure 5.5. For Step 1, we port-label all the edges from G_0, G_1 and G_2 individually, using up ports 1 to $d - 1 = 3$, and the finished work is displayed in Figure 5.6. When $i = 0$ in Step 2, the ports (from $d = 4$ to $n - 1 = 11$) whose ends are the vertices from V_{G_0} are

orange in Figures 5.7 ($G_{0,1}$) and 5.8 ($G_{0,2}$). When $i = 1$, the ones whose ends are the vertices from V_{G_1} are blue in Figures 5.7 ($G_{0,1}$) and 5.9 ($G_{1,2}$); when $i = g-1 = 2$, the ones whose ends are the vertices from V_{G_2} are teal in Figures 5.8 ($G_{0,2}$) and 5.9 ($G_{1,2}$). After all the vertices from V_{G_0} are port-switched as the last step, the updated labelling affects 8 ports (all in magenta) involving 6 edges in Figure 5.10 and the rest of labelling remains unchanged since Step 2. At v_1, v_2, v_3, v_4, outgoing port 1 is now associated with an edge whose other port is 7, whereas at all other vertices, outgoing port 1 is still associated with an edge whose other port is 2; at $v_9, v_{10}, v_{11}, v_{12}$, outgoing port 7 is now associated with an edge whose other port is 1, whereas at all other vertices, outgoing port 7 is still associated with an edge whose other port is 11.

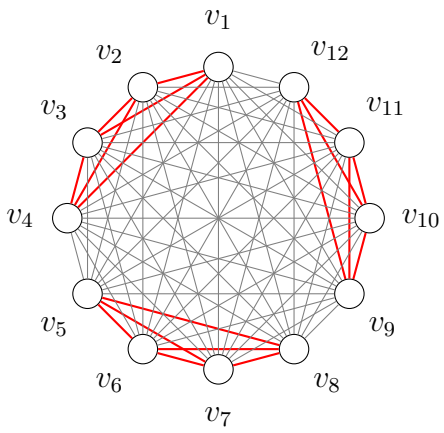


Figure 5.5: The vertex-labelled complete graph K_{12}

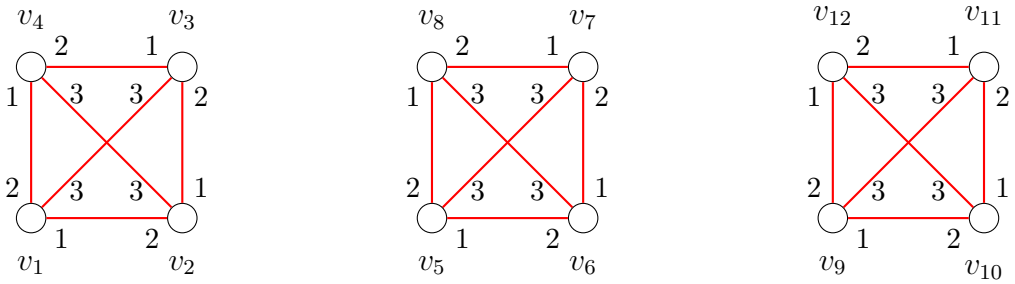


Figure 5.6: The partial labellings on three cliques of size four (Step 1)

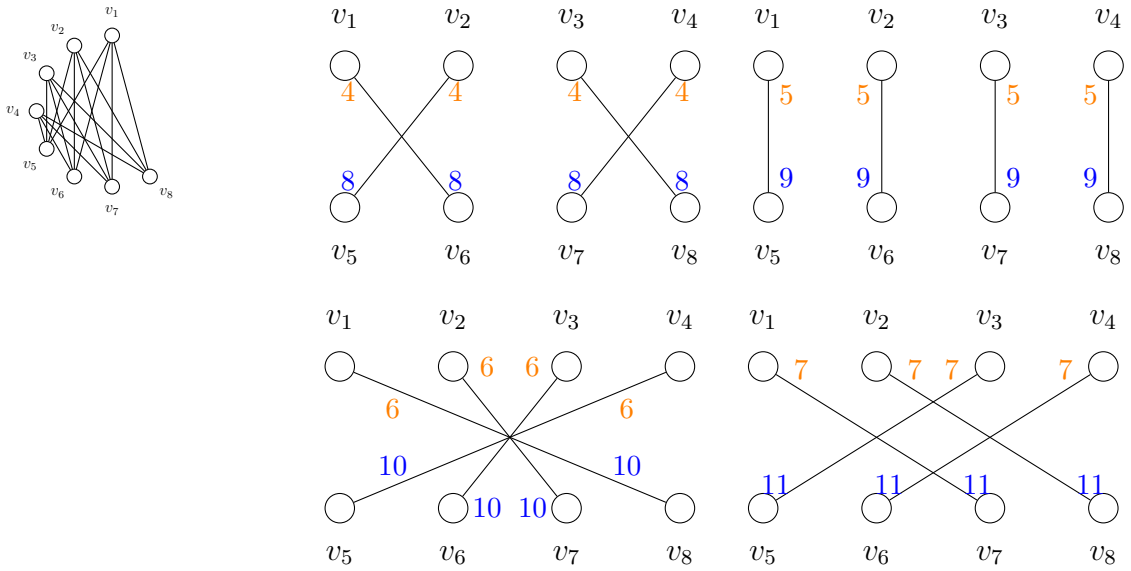
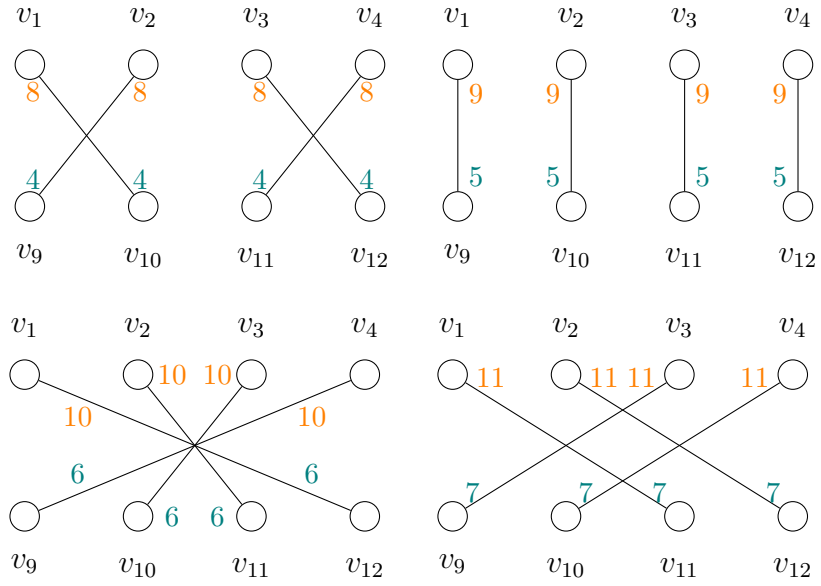


Figure 5.7: The total labelling on a 1-factorization of $G_{0,1}$ (Step 2)



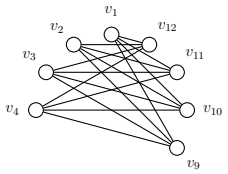
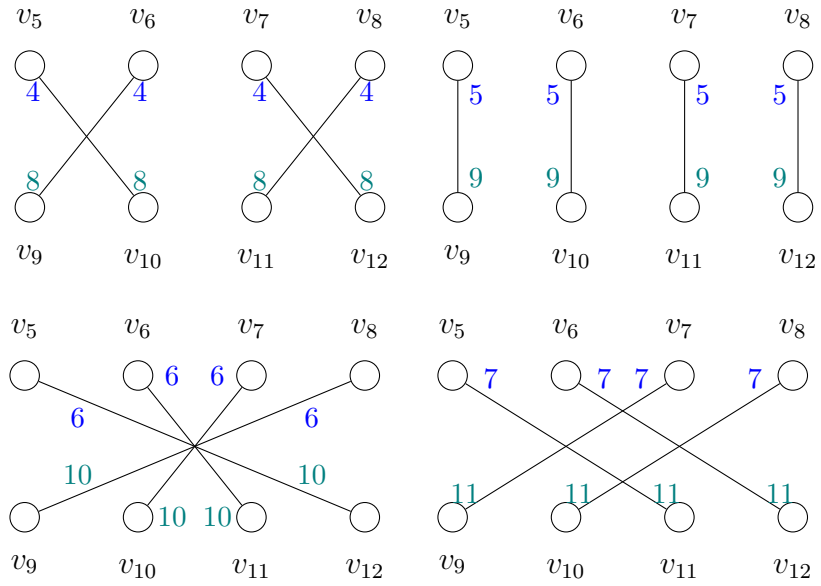


Figure 5.8: The total labelling on a 1-factorization of $G_{0,2}$ (Step 2)



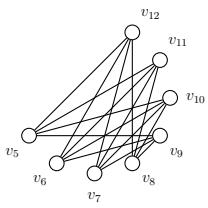


Figure 5.9: The total labelling on a 1-factorization of $G_{1,2}$ (Step 2)

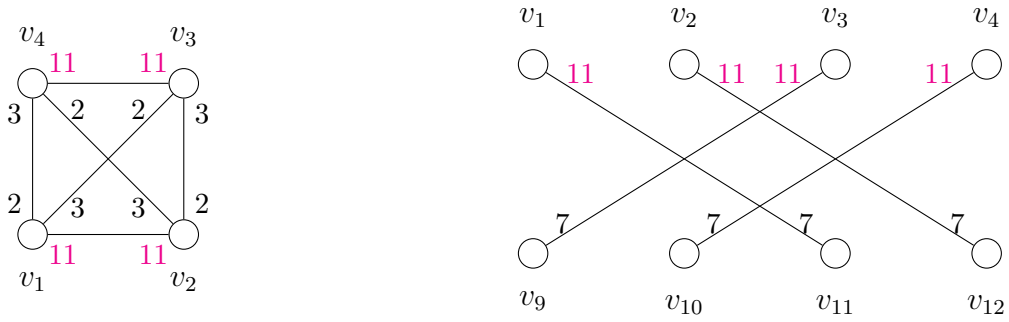


Figure 5.10: Port-switching the vertices in V_{G_0} (Step 3)

5.5 Rooted Product

Problem 2 becomes trivial instantly when Problem 1 is solved — we can always come up with a complete graph K_n under a labelling $\mathbf{f}$ whose multiplicity is d by applying the function FullySymmetricLabel (Algorithm 5.7). For Problem 3, given $n \geq 3$, all its divisors can be the multiplicity of a labelling on a complete graph K_n according to Problem 1 as well, which implies that Problem 3 can be solved in the affirmative.

We hope to generate more labellings on other classes of graphs with n vertices and $s_{\mathbf{f}} = d$, for $1 \leq d \leq n$ and $d \mid n$. For $d = 1$, any labelling on a graph with a vertex of a unique degree such as a star graph S_n or a wheel graph W_n[3] will suffice, since their symmetricity must be 1 (under any labelling, the universal vertex[4] with the unique degree in such graphs has a dissimilar view from others). For $d = n$, all $\{1, 2\}$-factorable graphs under a fully symmetric labelling have the desired property. Next, for values of d strictly between 1 and n, we provide a systematic way of generating examples. We will make use of the rooted product, which can be informally described as: take any graph $G = (V, E)$ and any rooted graph R, create $|V_G|$ copies of R, and replace each $v_i \in V_G$ with the root of the i-th copy of R. This operation is formally defined in Definition 5.5 and illustrated in Example 5.3.

Definition 5.5. Given a graph $G = (V_G, E_G)$ with $V_G = \{ v_1, v_2, \ldots, v_{n_1} \}$

[3] A wheel graph W_n is a graph with n vertices containing a cycle graph C_{n-1} and a vertex adjacent to every vertex from C_{n-1}.

[4] A universal vertex of a graph is a vertex adjacent to all other vertices in the graph.

and a rooted graph $R = (V_R, E_R)$ with $V_R = \{ u_1, u_2, \ldots, u_{n_2} \}$, whose root is u_1, the **rooted product** of G and R is defined as $G \odot R \coloneqq (V', E')$, where

$$V' = \{ w_{i,j} \mid 1 \leq i \leq n_1, 1 \leq j \leq n_2 \}$$

and

$$E' = \{ (w_{i,1}, w_{k,1}) \mid (v_i, v_k) \in E_G \} \cup \bigcup_{i=1}^{n_1} \{ (w_{i,j}, w_{i,k}) \mid (u_j, u_k) \in E_R \}.$$

Example 5.3. In Figure 5.11, there is a graph G with five vertices and a rooted graph R with four vertices, whose root is filled in black. The result of $G \odot R$ is a newly-generated graph with 20 vertices (rightmost).

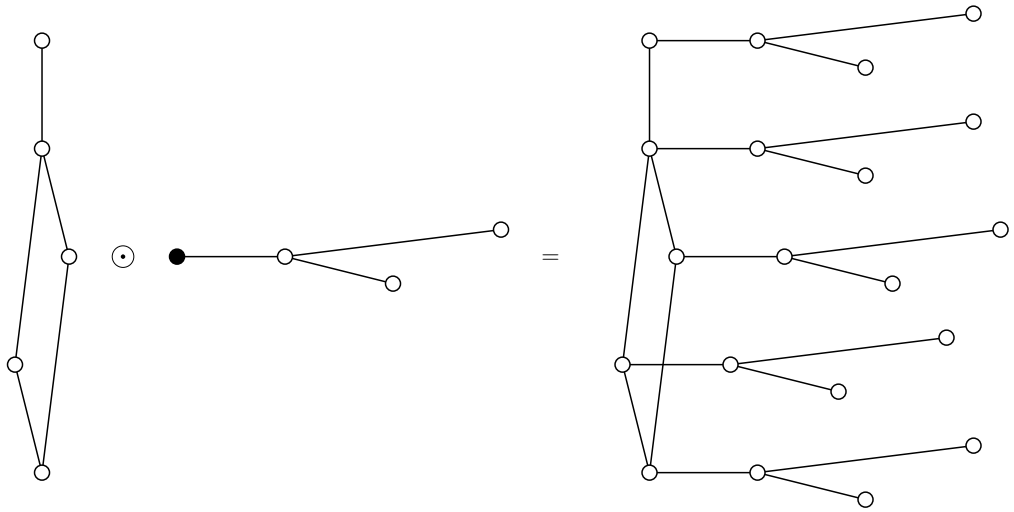


Figure 5.11: An illustration of $G \odot R$[5]

Below is the helper function for Algorithm 5.9, which port-labels a graph in an arbitrary and total fashion by traversing each vertex.

[5]This diagram is based on Dr. David Eppstein's image at the Wikipedia entry *rooted product of graphs*.

Algorithm 5.8 Port-label a graph arbitrarily and totally

```
1: function ArbitraryLabel(G)
2:     n ← |V_G|
3:     f ← an empty 2-dimensional n-by-n array
4:     for all v_i ∈ V_G do
5:         p ← 1
6:         for all v_j ∈ N_G(v_i) do
7:             f_{v_i}(v_j) ← p
8:             p ← p + 1
9:         end for
10:    end for
11:    return f
12: end function
```

Based on Corollary 2.1 and the rooted product operation, the algorithm for returning a totally port-labelled graph with n vertices whose multiplicity is d is proposed below, taking two integers n and d as inputs. The idea behind our algorithm is to compute the rooted product of a $\{1, 2\}$-factorable graph G with d vertices under a fully symmetric labelling and a totally (and arbitrarily) port-labelled rooted graph R with $g = n/d$ vertices. The resulting graph will have n vertices under a labelling of multiplicity d. The root of R can be any vertex with maximum degree Δ_R, which will ensure that d vertices (created by duplicating the root for d times) in the graph have degree $\Delta_G + \Delta_R$ (and none of the other vertices will have this degree). These d vertices now are connected by the edges originated from G, which is $\{1, 2\}$-factorable, therefore they will have similar views. The function FullySymmetricLabel in Algorithm

5.9 line 7 is taken from Algorithm 5.4. In line 8, a graph with g vertices is port-labelled arbitrarily by the function ArbitraryLabel taken from Algorithm 5.8. The input graphs G and R come with port labelling functions $\mathbf{f}_G$ and $\mathbf{f}_R$ in line 9; here we define the labelling $\mathbf{f}$ for the rooted product as the following: for each $i \in \{1, 2, \ldots, n_1\}$ and each edge $(v_i, v_k) \in E_G$, we set $\mathbf{f}_{w_{i,1}}(w_{k,1}) = \mathbf{f}_{G,v_i}(v_k)$, and, for each $j \in \{1, 2, \ldots, n_2\}$ and each edge $(u_j, u_k) \in E_R$, we set $\mathbf{f}_{w_{i,j}}(w_{i,k}) = \mathbf{f}_{R,u_j}(u_k)$.

Algorithm 5.9 Return the rooted product of two port-labelled graphs

Require: $n \geq 3$, $d \mid n$ and $1 < d < n$

1: **function** ReturnRootedProduct(n, d)
2: $g \leftarrow n/d$
3: $G \leftarrow$ an arbitrary vertex-labelled $\{1, 2\}$-factorable graph with d vertices
4: $R \leftarrow$ an arbitrary vertex-labelled rooted graph with g vertices whose root is the vertex with maximum degree Δ_R
5: $\mathbf{f}_G \leftarrow$ an empty 2-dimensional n-by-n `array`
6: $\mathbf{f}_R \leftarrow$ an empty 2-dimensional n-by-n `array`
7: $\mathbf{f}_G \leftarrow$ FullySymmetricLabel($G, \Delta_R + 1$)
8: $\mathbf{f}_R \leftarrow$ ArbitraryLabel(R)
9: **return** $\langle G, \mathbf{f}_G \rangle \odot \langle R, \mathbf{f}_R \rangle$
10: **end function**

Example 5.4. To generate a totally port-labelled graph with 12 vertices whose multiplicity is 4, in Figure 5.12, we first do a fully symmetric labelling on an arbitrarily chosen $\{1, 2\}$-factorable graph with four vertices (in this example, we choose C_4); then an arbitrary graph with three vertices is port-

labelled arbitrarily (here we choose P_3) and the vertex with the maximum degree is assigned to the root (marked green). The result of the rooted product $\langle C_4, \mathbf{f}_{C_4} \rangle \odot \langle P_3, \mathbf{f}_{P_3} \rangle$ is $\langle G, \mathbf{f}_G \rangle$, i.e., the rightmost graph in Figure 5.12, where the same colour means that the vertices share similar views. Clearly its multiplicity is 4.

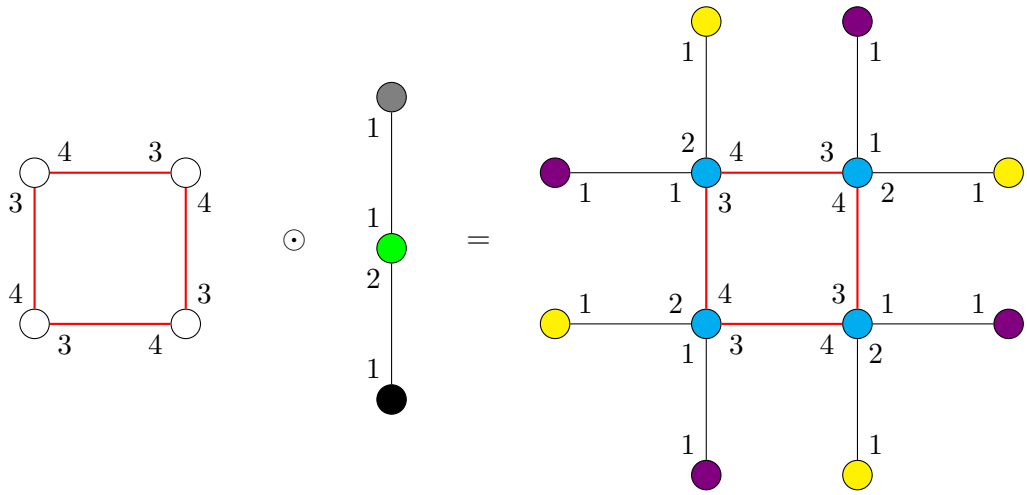


Figure 5.12: The rooted product of a port-labelled graph C_4 and a port-labelled rooted graph P_3

6 Computing Symmetry

> Everything you see or hear or experience in any way at all is specific to you. You create a universe by perceiving it, so everything in the universe you perceive is specific to you.
>
> *Mostly Harmless*
> Douglas Adams

In this chapter, we aim to calculate symmetry for views and 'degree trees' (formally defined in Section 6.3). We first show how to compute a view in Section 6.1 and solve Problem 4; based on that, in Section 6.2, a general approach is provided to compute the multiplicity of a totally port-labelled graph. In Sections 6.3 and 6.4, we then explain why degree trees in anonymous networks are introduced, and define them along with some mathematical results. Later on, the algorithms on generating degree trees are demonstrated in Section 6.5. In Section 6.6, we give an application of degree trees on

offering an upper bound on symmetricity. Furthermore, Section 6.7 is rather independent of the previous sections: we prove a sufficient and necessary condition on symmetricity of a tree.

6.1 Computing Views

Before building algorithms on views, we discuss how to represent and store a view. For each directed edge (ν, ν') in a view, with p_{parent} as the outgoing port from ν and with p_{child} as the incoming port to ν', we group them into an ordered pair $(p_{\text{parent}}, p_{\text{child}})$ and put it as the value stored in ν' — we can think of each node storing an outgoing-incoming pair of ports coming from its unique "parent" node in the view. The children list of each node ν in a `tree` hereinafter is implemented as a 1-dimensional `array` A_ν whose size is equal to the out-degree of ν (we use $\deg(\nu)$ to represent): each child node ν' of a parent node ν with outgoing port p_{parent} is stored in A_ν accessed by index p_{parent}, i.e., the entry $A_\nu[p_{\text{parent}}]$. The process is illustrated in Figure 6.1. From now on, we assume that all the views are encoded in this way.

Next, we describe a naïve recursive algorithm to generate the view of a fixed vertex in a graph under a total labelling. In Algorithm 6.1 line 1, the function GenerateView is called by assigning $n - 1$ to ℓ, where n is $|V_G|$. In line 2, the variable ν_r acts as the root of a `tree`, and its children list A_{ν_r} is implicitly initialized to be an empty 1-dimensional `array` of size $\deg(v) = \deg(\nu_r)$. The neighbourhood of v, i.e., $N_G(v)$, is implemented as a

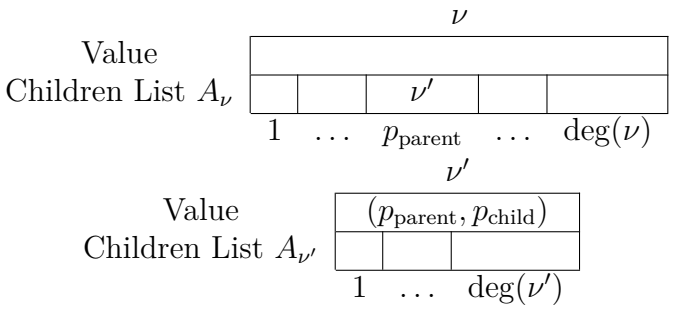


Figure 6.1: The data changing in ν and ν' when a directed edge (ν, ν') in a view is converted

`set` in line 3. In line 5, if $\ell > 1$, the recursion continues; if $\ell = 1$, then ν_i, as a leaf (with an empty children list), is added into A_ν directly.

Algorithm 6.1 Generate the view of a vertex in a graph under a labelling

Require: $v \in V_G$ and $\ell \geq 1$

1: **function** GENERATEVIEW($\langle G, \mathbf{f} \rangle, v, \ell$)
2: $\quad \nu_r \leftarrow$ a node with no value and an empty children list A_{ν_r}
3: $\quad$ **for all** $v_i \in N_G(v)$ **do**
4: $\quad\quad \nu_i \leftarrow$ a node with no value and an empty children list A_{ν_i}
5: $\quad\quad$ **if** $\ell > 1$ **then**
6: $\quad\quad\quad \nu_i \leftarrow$ GENERATEVIEW($\langle G, \mathbf{f} \rangle, v_i, \ell - 1$)
7: $\quad\quad$ **end if**
8: $\quad\quad p_{\text{parent}} \leftarrow f_v(v_i)$
9: $\quad\quad p_{\text{child}} \leftarrow f_{v_i}(v)$
10: $\quad\quad \nu_i$'s value $\leftarrow (p_{\text{parent}}, p_{\text{child}})$
11: $\quad\quad A_{\nu_r}[p_{\text{parent}}] \leftarrow \nu_i$
12: $\quad$ **end for**
13: $\quad$ **return** ν_r
14: **end function**

The issue with the naïve approach is that, in the worst case, its running time is exponential in the size of the graph. The reason is that there are many recursive calls that re-compute views unnecessarily. By applying dynamic programming (Algorithm 6.2), generating all n (not necessarily mutually dissimilar) views for a graph can be done within $O(n^2\Delta_G)$ time, using a 2-dimensional n-by-$(n-1)$ `array` $A_{T_\mathbf{f}}$. As Table 6.1 illustrates, each entry $A_{T_\mathbf{f}}[v_i, \ell]$ contains v_i's view truncated to level ℓ.

Vertex v_i \ **Level** ℓ	1	2	...	$n-1$
v_1	$T_\mathbf{f}^1(v_1)$	$T_\mathbf{f}^2(v_1)$	...	$T_\mathbf{f}^{n-1}(v_1)$
v_2	$T_\mathbf{f}^1(v_2)$	$T_\mathbf{f}^2(v_2)$	...	$T_\mathbf{f}^{n-1}(v_2)$
...	...	...	...	...
v_n	$T_\mathbf{f}^1(v_n)$	$T_\mathbf{f}^2(v_n)$	...	$T_\mathbf{f}^{n-1}(v_n)$

Table 6.1: The 2-dimensional `array` $A_{T_\mathbf{f}}$

Algorithm 6.2 Fill $A_{T_\mathbf{f}}$

```
1:  function FillATf(⟨G, f⟩)
2:      n ← |V_G|
3:      A_Tf ← an empty 2-dimensional n-by-(n − 1) array
4:      for ℓ ← 1 to n − 1 do
5:          for all v_i ∈ V_G do
6:              ν_i ← a node with no value and an empty children list A_νi
7:              for all v_j ∈ N_G(v_i) do
8:                  ν_j ← a node with no value and an empty children list A_νj
9:                  p_parent ← f_vi(v_j)
10:                 p_child ← f_vj(v_i)
11:                 if ℓ > 1 then
12:                     ν_j ← A_Tf[v_j, ℓ − 1]
```

```
13:                 end if
14:                 ν_j's value ← (p_parent, p_child)
15:                 A_{ν_i}[p_parent] ← ν_j
16:             end for
17:             A_{T_f}[v_i, ℓ] ← ν_i
18:         end for
19:     end for
20:     return A_{T_f}
21: end function
```

Each element accessed by indices v_i and $n-1$ in $A_{T_\mathbf{f}}$ represents $T_\mathbf{f}^{n-1}(v_i)$. But, for the purpose of determining if two views are similar, we recall from Subsection 2.1.1 that it is sufficient to compare the views up to level $n-1$. Hence, three variables $A_{T_\mathbf{f}}[v_i, n-1]$, $T_\mathbf{f}^{n-1}(v_i)$ and $T_\mathbf{f}(v_i)$ are synonymous from now on and can be used interchangeably.

If we want to compare if two views are similar, a modified breadth-first search (BFS) can be applied, with an added feature that the procedure terminates early as soon as it detects that the two views are dissimilar. For line 8 of Algorithm 6.3, we guarantee that $queue_1$ and $queue_2$ will have the same size all the time. In line 13, two nodes are equal if their own values are equivalent and the sizes of their children lists are the same.

Algorithm 6.3 Check if two views are similar

```
1: function IfTwoViewsSimilar(T_1, T_2)
2:     queue_1 ← an empty list
3:     queue_2 ← an empty list
4:     ν_1 ← a node with no value and an empty children list A_{ν_1}
```

```
5:      ν2 ← a node with no value and an empty children list A_ν2
6:      Append the root of T1 to queue1
7:      Append the root of T2 to queue2
8:      while queue1 is not empty ∧ queue2 is not empty do
9:          ν1 ← the first element in queue1
10:         Delete the first element from queue1
11:         ν2 ← the first element in queue2
12:         Delete the first element from queue2
13:         if ν1 = ν2 then
14:             for all ν_i, ν_j ∈ A_ν1, A_ν2 do
15:                 Append ν_i to queue1
16:                 Append ν_j to queue2
17:             end for
18:         else
19:             return False
20:         end if
21:     end while
22:     return True
23: end function
```

6.2 Computing Multiplicities

Now that we have the algorithms for generating and comparing views, we are ready to describe an algorithm to calculate multiplicity. We will use the following helper function that returns all integer factors of n. Since the factors of an odd number are always odd, all even possible numbers are skipped in terms of an odd n (from line 2 to line 6 of Algorithm 6.4). In line 9, $\lceil n^{1/2} \rceil$ can be replaced by $\lceil n/2 \rceil$ with nearly no side effects if the square

root operation is considered slow in the target environment. Lines 11 and 12 indicate that integer factors always come in pairs (i and n/i may be the same if n is a perfect square) and the size of *factors* is at least two. In line 16, the function SORT takes a `set` and returns a `list` of all the elements of the `set` in ascending order.

Algorithm 6.4 Return all integer factors of n

Require: $n \geq 2$

```
1:  function ReturnAllIntegerFactors(n)
2:      if n mod 2 = 0 then
3:          step ← 1
4:      else
5:          step ← 2
6:      end if
7:      i ← 1
8:      factors ← ∅
9:      while i ≤ ⌈n^(1/2)⌉ do
10:         if n mod i = 0 then
11:             factors ← factors ∪ { i }
12:             factors ← factors ∪ { n/i }
13:         end if
14:         i ← i + step
15:     end while
16:     return Sort(factors)
17: end function
```

Finally, we introduce the algorithm for calculating the multiplicity $s_{\mathbf{f}}$ of a graph G under a labelling $\mathbf{f}$ using the previous results. The function COMPUTEMULTIPLICITY in Algorithm 6.5 is deeply dependent on Proposition

2.1: for any multiplicity $s_{\mathbf{f}}$ of a graph with n vertices, there are $x = n/s_{\mathbf{f}}$ dissimilar views shared by $y = s_{\mathbf{f}}$ vertices each. This fact can drastically reduce the number of vertices that the function has to visit. The empty `dictionary` $M_{\mathcal{T}:\mathbb{V}}$ is declared in line 4, mapping the set of all dissimilar views $\mathcal{T}$ to the power set of the vertex set. The variable x in line 7 stands for the number of the dissimilar views already stored in $M_{\mathcal{T}:\mathbb{V}}$ as keys and y in line 8 is the maximum number of vertices sharing similar views among x ones. The function IfTwoViewsSimilar from Algorithm 6.3 is used implicitly in line 10 to compare $T_{\mathbf{f}}(v_i)$ with pre-stored keys of $M_{\mathcal{T}:\mathbb{V}}$. The functions Next and Previous (in lines 14 and 20 respectively) act as an iterator, returning the next/previous element from the last probed position of a `list` if there exists one. The worst-case running time of the function is bounded by the second largest factor of n: for an even n, the algorithm traverses at most $n/2 + 1$ vertices with their views; especially, for a graph with a prime n, it only takes two vertices to decide its multiplicity.

Algorithm 6.5 Compute the multiplicity of a graph under a labelling

1: **function** ComputeMultiplicity($\langle G, \mathbf{f} \rangle, A_{T_{\mathbf{f}}}$)
2: $n \leftarrow |V_G|$
3: *factors* $\leftarrow$ ReturnAllIntegerFactors(n)
4: $M_{\mathcal{T}:\mathbb{V}} \leftarrow$ an empty `dictionary`
5: *atleast* $\leftarrow$ the first element of *factors*
6: *atmost* $\leftarrow$ the last element of *factors*
7: $x \leftarrow 0$
8: $y \leftarrow 0$
9: **for all** $v_i \in V_G$ **do**

```
10:         if T_f(v_i) ∉ M_{T:V} then
11:             M_{T:V}[T_f(v_i)] ← { v_i }
12:             x ← x + 1
13:             if atleast < x then
14:                 atleast ← NEXT(factors)
15:             end if
16:         else
17:             M_{T:V}[T_f(v_i)] ← M_{T:V}[T_f(v_i)] ∪ { v_i }
18:             y ← max(y, |M_{T:V}[T_f(v_i)]|)
19:             if atmost > n/y then
20:                 atmost ← PREVIOUS(factors)
21:             end if
22:         end if
23:         if atleast = atmost then
24:             return atleast
25:         end if
26:     end for
27: end function

28: s_f ← COMPUTEMULTIPLICITY(⟨G, f⟩ , A_{T_f})
```

Example 6.1. Imagine that the function ComputeMultiplicity is invoked on a fully port-labelled graph with 36 vertices. After 12 out of 36 vertices are examined, suppose that the algorithm has found four dissimilar views so far: view T_1 is shared by two vertices, T_2 by two, T_3 by three and T_4 by five. Since 5 is not an integer factor of 36 and 6 is the smallest factor larger than 5, now we are certain that $6 \leq s_{\mathbf{f}} \leq 36/4 = 9$, narrowing the multiplicity to be either 6 or 9. If the 13th vertex turns out to have a view T_5 that was

not encountered before, we then conclude immediately that $s_{\mathbf{f}} = 6$ without going further — in order to make $s_{\mathbf{f}} = 9$, we can only allow *exactly* four views within 36 vertices.

6.3 Degree Trees: Motivation and Notation

In this section, we first illustrate an example to motivate the use of degree trees, and then proceed to formally define them.

Example 6.2. The left-hand side vertex-labelled graph G_1 in Figure 6.2 has been port-labelled with $\mathbf{f}_1$ and the right-hand side graph G_2 has been port-labelled with $\mathbf{f}_2$ such that $T_{\mathbf{f}_1}(v_i) \equiv T_{\mathbf{f}_2}(u_i)$ for each $i \in \{\, 1, 2, \ldots, 6 \,\}$. Under this special labelling, $T_{\mathbf{f}_1}(v_1) \equiv T_{\mathbf{f}_2}(u_1) \equiv T_{\mathbf{f}_1}(v_4) \equiv T_{\mathbf{f}_2}(u_4)$. In particular, even if v_1 was given a map of G_1 as advice, and if u_2 was given a map of G_2 as advice, they could not determine where they were located within the given map. Moreover, based only on their views (i.e., no additional advice given), v_1 in G_1 and u_2 in G_2 are not able to tell if they are actually located in G_1 or G_2 under these labellings, although G_1 and G_2 are not even isomorphic.

There are situations where the ports are assigned in a very symmetric way or even worse, in a fully symmetric way, and accordingly a vertex fails to learn much from its view besides information that was already available using just the topology of a graph. This motivates us to consider a simpler version of views, i.e., with the port information stripped away. We derive Definition 6.1 from Definition 2.11.

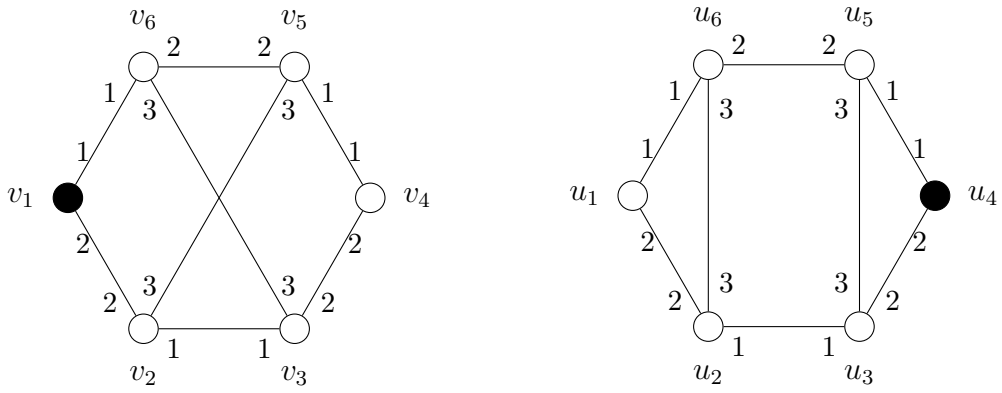


Figure 6.2: Two graphs G_1 (left) and G_2 (right) under a fully symmetric labelling

Definition 6.1. The **degree tree** $D_G(v)$ of a vertex v in a graph G is an arborescence with infinite number of levels, defined recursively as follows: $D_G(v)$ contains a unique vertex called root, represented by v itself; for each neighbour of v in G, $D_G(v)$ contains a vertex v_i and a directed edge from v to v_i; the vertex v_i is the root of $D_G(v_i)$.

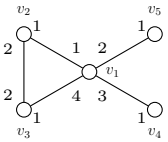
Compared to the view, the degree tree is the arborescence itself and purely extracted from the topological structure of a graph.

Definition 6.2 (Degree Tree Similarity/Dissimilarity)**.** Two degree trees D and D' are said to be **similar** if there exists a directed graph isomorphism between them, and we then write $D \equiv D'$; otherwise, D and D' are **dissimilar**, written as $D \not\equiv D'$.

It is possible for two vertices v and v' to have $D_G(v) \equiv D_{G'}(v')$ for $G \neq G'$.

Example 6.3. In Figure 6.3, the tree on the right-hand side is the degree tree $D_G(v_1)$, where v_1 is from Figure 2.2 on page 12 (as the margin note). The

root representing v_1 is filled in black. It is worth mentioning that $D_G(v_1)$ is highly symmetric: for these subtrees, $D_G(v_2) \equiv D_G(v_3)$ and $D_G(v_4) \equiv D_G(v_5)$, whereas $T_{\mathbf{f}}(v_2) \not\equiv T_{\mathbf{f}}(v_3)$ and $T_{\mathbf{f}}(v_4) \not\equiv T_{\mathbf{f}}(v_5)$.



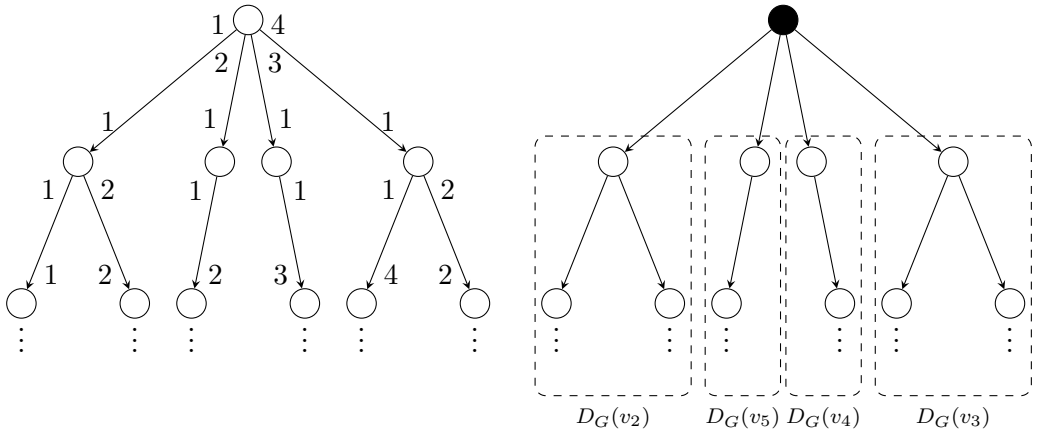


Figure 6.3: The view $T_{\mathbf{f}}(v_1)$ and its corresponding degree tree $D_G(v_1)$

The degree tree up to level ℓ for a vertex v in a graph G is $D_G(v)$ truncated to the first ℓ levels, denoted by $D_G^{\ell}(v)$. The truncated degree tree $D_G^{n-1}(v)$ of a vertex v is computationally equivalent to $D_G(v)$ [78], i.e., $D_G^{n-1}(v_1) \equiv D_G^{n-1}(v_2)$ iff $D_G(v_1) \equiv D_G(v_2)$ — now checking if two (infinite) degree trees are similar can be done in a finite amount of time.

Definition 6.3. The **degree tree collection** $\mathcal{D} = \{\, D(v) \mid v \in V \,\}$ is the set of all dissimilar degree trees from a graph $G = (V, E)$.

Definition 6.4. The **degree tree class** $V_D = \{\, v \mid D(v) \equiv D \,\}$ is the set of all vertices with a degree tree that is similar to D; correspondingly, the **degree tree vertex partition** $\mathcal{V} = \{\, V_D \mid D \in \mathcal{D} \,\}$ denotes the partition of V into its degree tree classes.

Unlike Proposition 2.1 for views, it is not guaranteed that each degree tree class has the same size. We let $|\mathcal{V}| = |\mathcal{D}| = \kappa$ represent the size of the degree tree collection, which is also equal to the number of classes in the degree tree vertex partition.

6.4 Some Results About Degree Trees

Since degree trees are developed directly from views, almost all properties from views can be inherited by degree trees with minor modifications. We assume all the vertices mentioned below are located in the same graph G.

Lemma 6.1. *If $T(v_1) \equiv T(v_2)$, then $D(v_1) \equiv D(v_2)$.*

Proof. A degree tree for a vertex is its view (under any labelling) without its edge labelling and remaining the structure of the original arborescence. ■

Lemma 6.2. *If $D(v_1) \equiv D(v_2)$, $\deg(v_1) = \deg(v_2)$.*

Proof. Since the degree trees of v_1 and v_2 are isomorphic, the root of $D(v_1)$ has the same number of nodes as that of $D(v_2)$, which is equal to $\deg(v_1)$ and also $\deg(v_2)$. ■

If two vertices have similar views, they must have similar degree trees, implying that their degrees are equal.

Proposition 6.1. *If there exists a vertex in a graph G with degree tree D and $|V_D| = 1$, then $\sigma(G) = 1$.*

Proof. Under any labelling, the vertex with such a degree tree D cannot have a view that is similar to any other vertices in G since edge labelling cannot change the topological structure of the arborescence. According to Proposition 2.1, $\sigma(G) = 1$. ■

The following theorem says that, if two vertices belong to the same V_D, i.e., $D(v_1) \equiv D(v_2)$, then they are each adjacent to the same number of vertices from any fixed $V_{D'}$; it holds even when $D \equiv D'$.

Theorem 6.1. *If* $v_1, v_2 \in V_D$, $|N_G(v_1) \cap V_{D'}| = |N_G(v_2) \cap V_{D'}|$ *for any degree tree* D'.

Proof. Since $v_1, v_2 \in V_D$, $D(v_1) \equiv D(v_2)$. There exists a degree tree similarity between $D(v_1)$ and $D(v_2)$, so there is an isomorphism $\psi\colon D(v_1) \to D(v_2)$. This isomorphism must map v_1 (the root of $D(v_1)$) to v_2 (the root of $D(v_2)$), since they are the only two nodes with in-degree 0. It also maps the nodes at level 1 of $D(v_1)$ (children of v_1) to the nodes at level 1 of $D(v_2)$ (children of v_2). However, each child of v_1 with degree tree D' must be mapped to a child of v_2 with degree tree D' and vice versa, since ψ is an isomorphism that must preserve the structure of their subtrees. This proves that the number of children of v_1 with degree tree D' is equal to that of children of v_2 with degree tree D'. ■

Proposition 6.2. *A graph is regular iff* $\kappa = 1$.

Proof. The proof consists of two parts.

The "if" part. If $\kappa = 1$, which indicates that there is only one similar degree tree, then the out-degree of the root of each degree tree must be the same. Hence, each vertex v_i in the graph G has degree $\deg(v_i) = \Delta_G$. By Definition 2.19, G is regular.

The "only if" part. If a graph $G = (V, E)$ is regular, then $\deg(v_i) = \deg(v_j) = \Delta_G$, for any $v_i, v_j \in V$. Each vertex's degree tree truncated to level ℓ is an ℓ-level perfect Δ_G-ary arborescence[1] (such a degree tree up to level 2 is shown in Figure 6.4). Therefore $\kappa = 1$.

■

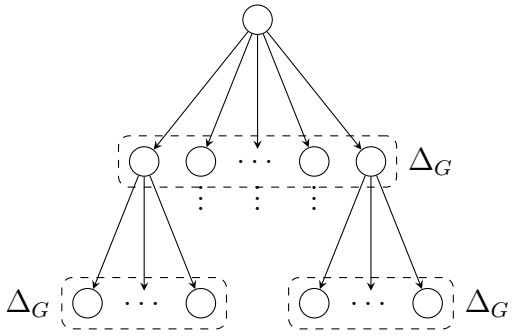


Figure 6.4: The degree tree up to level 2 for a vertex from a regular graph

Definition 6.5. A **biregular** graph is a bipartite graph $G = (U \cup V, E)$ where the vertices in each bipartition U or V have the same degree. An (x, y)**-biregular** graph is a biregular graph $G = (U \cup V, E)$ where all the vertices in U have degree x and the vertices in V have degree y.

[1] A perfect k-ary arborescence is a $(2^{k+1} - 1)$-node arborescence and all its internal nodes have out-degree k.

Corollary 6.1. *An (x, y)-biregular graph $G = (U \cup V, E)$ satisfies $x|U| = y|V|$.*

Proof. It can be verified by a simple double counting argument: the number of endpoints of edges in U is $x|U|$, that of endpoints of edges in V is $y|V|$, and each edge contributes the same amount to both numbers. ■

In some of the literature, biregular graphs (aka 2-degree/valency graphs) are defined as graphs in which there are precisely two different vertex degrees, i.e., there is no requirement that the graph is bipartite and that the degrees are the same within each set of the bipartition. Take Figure 6.5 as an example: the left-hand side graph P_5 is bipartite and 2-degree, but not biregular (by our definition), since the vertices within each bipartition have different degrees; the graph on the right-hand side satisfies our stronger definition of biregular. Every complete bipartite graph $K_{a,b}$ is (a, b)-biregular, and every k-regular bipartite graph is biregular (or (k, k)-biregular).

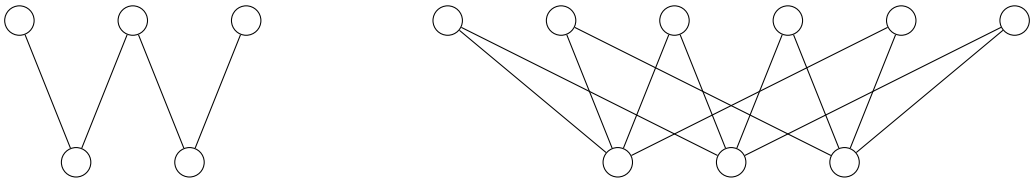


Figure 6.5: The difference between a 2-degree graph and a biregular graph

Since we disallow disconnected graphs, the following observation is always the case.

Observation 6.1. If $G = (U \cup V, E)$ is biregular with $|U| = 2$ or $|V| = 2$, then G is a complete bipartite graph.

Proposition 6.3. *If a graph is (x, y)-biregular for $x \neq y$, then $\kappa = 2$.*

Proof by induction. Let $G = (U \cup V, E)$ be an (x, y)-biregular graph and ℓ be the particular level of a degree tree.

Base case. When $\ell = 1$, according to Corollary 6.1, any vertex $u \in U$ is adjacent to x vertices from V and is not adjacent to any vertices from U; similarly, any vertex $v \in V$ is adjacent to y vertices from U and is not adjacent to any vertices from V. In other words, for $u \in U$ and $v \in V$, the out-degree of each node at level 1 of $D(u)$ is x and that of each node at level 1 of $D(v)$ is y (see Figure 6.6).

Inductive step. We then argue that, if the out-degree of each node at level ℓ of D is x, then the out-degree of each node at level $\ell + 1$ of D is y; likewise, if the out-degree of each node at level ℓ of D is y, then the out-degree of each node at level $\ell + 1$ of D is x. Consider any vertex $u \in U$: for all children (at level $\ell + 1$) of a node at level ℓ of its degree tree $D(u)$, they all have a similar degree tree $D(v)$ for $v \in V$ by the induction hypothesis. Since this holds for arbitrary vertex $u \in U$, it proves that all vertices in U have a similar degree tree up to level ℓ. A similar proof shows that all vertices in V have a similar degree tree up to level ℓ.

This means that there are at most two dissimilar degree trees. We know that there are at least two dissimilar degree trees, since $u \in U$ and $v \in V$ have different degrees. This proves there are exactly two degree trees and $\kappa = 2$. ■

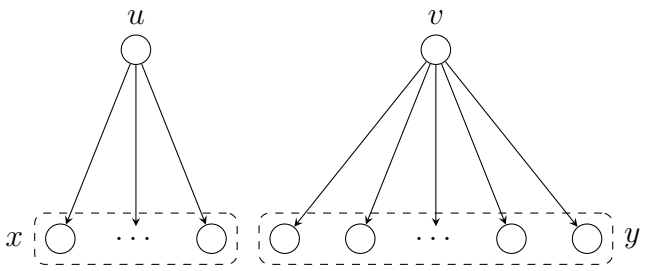


Figure 6.6: $D(u)$ and $D(v)$ at level 1 (Proposition 6.3)

Unlike Proposition 6.2, Proposition 6.3 is not biconditional: if $\kappa = 2$, then a graph is not necessarily biregular. For example, the vertices in Figure 6.7 with the same colour share similar degree trees: the two black vertices with degree 3 cannot belong to the same part in the bipartition as they are adjacent. Therefore, the graph here is definitely not biregular.

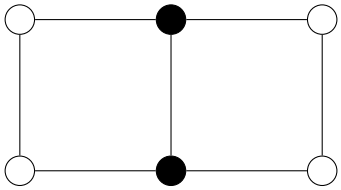


Figure 6.7: A non-biregular graph with $\kappa = 2$

Proposition 6.4 tells us that if G is a non-biregular graph with $\kappa = 2$, then there exist at least one vertex whose neighbours have two dissimilar degree trees. In Figure 6.7, each black vertex is an example of such a vertex.

Proposition 6.4. *For a non-biregular graph $G = (V, E)$ with $\kappa = 2$, there exist $v \in V$ and $\mathcal{D}^* = \{\, D(v_i) \mid v_i \in N_G(v) \,\}$ such that $|\mathcal{D}^*| = 2$.*

Proof. When $\kappa = 2$ for a non-biregular graph $G = (V, E)$, it simply means

that vertex set V can be divided into two disjoint degree tree classes V_D and $V_{D'}$ $(D \not\equiv D')$.

Case 1. If each vertex $v \in V_D$ is only adjacent to the vertices from V_D and each vertex $v' \in V_{D'}$ is only adjacent to vertices from $V_{D'}$, then G must be disconnected, which contradicts the assumption that we only consider connected graphs.

Case 2. If each vertex $v \in V_D$ is only adjacent to the vertices from V'_D and each vertex $v' \in V_{D'}$ is only adjacent to vertices from V_D, then G must be biregular by definition, which contradicts that G is non-biregular.

Since the two above cases are impossible, it follows that there exists a vertex that is adjacent to at least one vertex from V_D and at least one vertex from $V_{D'}$, which gives the desired result. ■

Proposition 6.5 shows that for a graph G with $\kappa \geq 3$, there exists at least one vertex whose neighbours have two or more dissimilar degree trees.

Proposition 6.5. *For a graph $G = (V, E)$ with $\kappa \geq 3$, there exist $v \in V$ and $\mathcal{D}^* = \{ D(v_i) \mid v_i \in N_G(v) \}$ such that $|\mathcal{D}^*| \geq 2$.*

Proof by contradiction. Assume that there exists no such a vertex. Then for each vertex v in G, the degree trees of all of v's neighbours are similar.

Case 1. All vertices in V_D only have neighbours in V_D: G must be either a regular graph or a disconnected one (which is disallowed). According to Proposition 6.2, all vertices in a regular graph have similar degree trees, i.e., $\kappa = 1$, which contradicts that $\kappa \geq 3$.

Case 2. All vertices in V_D only have neighbours in $V_{D'}$ ($D \not\equiv D'$): then G must be biregular. According to Proposition 6.3, G has two dissimilar degree trees, i.e., $\kappa = 2$, which contradicts that $\kappa \geq 3$.

■

6.5 Computing and Comparing Degree Trees

Using a brute-force approach resembling Algorithm 6.1, the following algorithm computes the degree tree for a single given vertex v. Besides, all the nodes here always contain no value unlike Algorithm 6.1 and only the tree structure matters.

Algorithm 6.6 Generate the degree tree of a vertex in a graph

Require: $v \in V_G$ and $\ell \geq 1$

```
1: function GenerateDegreeTree(G, v, ℓ)
2:     ν_r ← a node with no value and an empty children list A_{ν_r}
3:     index ← 1
4:     for all v_i ∈ N_G(v) do
5:         ν_i ← a node with no value and an empty children list A_{ν_i}
6:         if ℓ > 1 then
7:             ν_i ← GenerateDegreeTree(G, v, ℓ)
8:         end if
9:         A_{ν_r}[index] ← ν_i
10:        index ← index + 1
11:    end for
12:    return ν_r
13: end function
```

As discussed in Section 6.1, the brute-force approach can be very inefficient (exponential time), and a dynamic programming approach can compute all the degree trees simultaneously in polynomial time, as given in Algorithm 6.7. The variable *index* in line 7 is just a placeholder and its value does not have any special meaning. As was the case with views, the degree tree of a vertex v can be represented synonymously using any of $A_D[v, n-1]$, $D^{n-1}(v)$ and $D(v)$.

Algorithm 6.7 Fill A_D

```
1:  function FillAD(G)
2:      n ← |V_G|
3:      A_D ← an empty 2-dimensional n-by-(n − 1) array
4:      for ℓ ← 1 to n − 1 do
5:          for all v_i ∈ V_G do
6:              ν_i ← a node with no value and an empty children list A_{ν_i}
7:              index ← 1
8:              for all v_j ∈ N_G(v_i) do
9:                  ν_j ← a node with no value and an empty children list A_{ν_j}
10:                 if ℓ > 1 then
11:                     ν_j ← A_D[v_j, ℓ − 1]
12:                 end if
13:                 A_{ν_i}[index] ← ν_j
14:             end for
15:             A_D[v_i, ℓ] ← ν_i
16:         end for
17:     end for
18:     return A_D
19: end function
```

Our next goal is to provide algorithms that compute the degree tree collection $\mathcal{D}$ and the degree tree vertex partition $\mathcal{V}$. To help us do this, we first need an algorithm that checks if two degree trees are similar. Although this involves algorithmically solving graph isomorphism (whose complexity status is unresolved, and no polynomial-time algorithm is known), we are fortunate that the graphs involved are trees. There exists a linear-time AHU algorithm (1974) proposed by Aho, Hopcroft, and Ullman [2, pp. 84–85]: they claimed that two rooted trees are isomorphic iff their roots have identical canonical names (aka the Knuth parenthetical tuples); furthermore, two trees are isomorphic iff for all ℓ levels, the canonical level names of two trees are identical.

To check that two degrees up to level $n-1$ are similar, Algorithm 6.9 below is a modified $O(N^2)$ version [21] of the original AHU algorithm, where N is the number of nodes in a degree tree. Algorithm 6.8 is a helper function that computes the canonical name of the tree's root. The statement in line 8 is to yield each element ν_i from ν_r's children list A_{ν_r} one by one. The function RadixSort that appears in line 11 below sorts a `list` of binary strings lexicographically (see the least significant digit (LSD) radix sort [30, pp. 197–200]). Also, the function ToString in line 12 converts a `list` to a string by concatenating together the strings stored in the `list`.

Algorithm 6.8 Assign its canonical name for a degree tree

1: **function** AssignName(D)
2: $\quad\nu_r \leftarrow$ a node with no value and an empty children list A_{ν_r}
3: $\quad\nu_r \leftarrow$ the root of D

```
4:     if A_{ν_r} is empty then                          ▶ ν_r is a leaf
5:         return "()"
6:     else
7:         temp ← an empty list
8:         for all ν_i ∈ A_{ν_r} do
9:             Append AssignName(ν_i) to temp
10:        end for
11:        temp ← RadixSort(temp)
12:        return "(" + ToString(temp) + ")"
13:    end if
14: end function
```

Algorithm 6.9 Check if two degree trees are similar

```
1: function IfTwoDegreeTreesSimilar(D_1, D_2)
2:     if AssignName(D_1) = AssignName(D_2) then
3:         return True
4:     else
5:         return False
6:     end if
7: end function
```

With the help of two functions FillAD in Algorithm 6.7 and IfTwoDegreeTreesSimilar in Algorithm 6.9, we are able to build $\mathcal{D}$ (Algorithm 6.10) and $\mathcal{V}$ (Algorithm 6.11).

Algorithm 6.10 line 4, as a matter of fact, implicitly checks if a degree tree $D(v_i)$ is already in the collection by: iterating through the previously-existed degree trees, and, checking if each is similar to $D(v_i)$ (using Algorithm 6.9).

Algorithm 6.10 Generate the degree tree collection $\mathcal{D}$

```
1: function GenerateDegreeTreeCollection(G, A_D)
2:     D ← ∅
3:     for all v_i ∈ V do
4:         if D(v_i) ∉ D then
5:             D ← D ∪ { D(v_i) }
6:         end if
7:     end for
8:     return D
9: end function
```

With $\mathcal{D}$ generated by Algorithm 6.10, we can generate the corresponding $\mathcal{V}$ with $M_{\mathcal{D}:\mathcal{V}}$ in Algorithm 6.11. $M_{\mathcal{D}:\mathcal{V}}$ (line 2) is a one-to-one (injective) function mapping the degree tree collection $\mathcal{D}$ to the corresponding degree tree vertex partition $\mathcal{V}$ and implemented as a `dictionary` in Algorithm 6.11. Table 6.2 illustrates the data structure. The variable $\mathcal{V}$ in line 9 is implemented as a `set` of `sets`.

Degree Tree D_i (Key)	D_1	D_2	...	D_κ
Degree Tree Class V_{D_i} (Value)	V_{D_1}	V_{D_2}	...	V_{D_κ}

Table 6.2: $M_{\mathcal{D}:\mathcal{V}}$ associating $\mathcal{D}$ with $\mathcal{V}$

Algorithm 6.11 Generate the degree tree vertex partition $\mathcal{V}$

Ensure: $|\mathcal{V}| = |\mathcal{D}| = \kappa$

```
1: function GenerateDegreeTreeVertexPartition(G, D, A_D)
2:     M_{D:V} ← an empty dictionary
3:     for all D_i ∈ D do
4:         M_{D:V}[D_i] ← ∅
5:     end for
```

```
    for all v_i ∈ V_G do
        M_{D:V}[D(v_i)] ← M_{D:V}[D(v_i)] ∪ { v_i }
    end for
    V ← { ∅ }                          ▶ V is a set of disjoint degree tree classes
    for all D_i ∈ D do
        V ← V ∪ { M_{D:V}[D_i] }        ▶ M_{D:V}[D_i] = V_{D_i}
    end for
    return V
end function
```

In Algorithm 6.12, we combine the above ideas to generate $\mathcal{D}$, $\mathcal{V}$ and $M_{\mathcal{D}:\mathcal{V}}$ simultaneously in one pass.

Algorithm 6.12 Generate $\mathcal{D}$, $\mathcal{V}$ and $M_{\mathcal{D}:\mathcal{V}}$ altogether

Ensure: $|\mathcal{V}| = |\mathcal{D}| = \kappa$

```
function GenerateDVM(G, A_D)
    D ← ∅
    M_{D:V} ← an empty dictionary
    for all v_i ∈ V_G do
        if D(v_i) ∉ D then
            D ← D ∪ { D(v_i) }
            M_{D:V}[D(v_i)] ← { v_i }
        else
            M_{D:V}[D(v_i)] ← M_{D:V}[D(v_i)] ∪ { v_i }
        end if
    end for
    V ← { ∅ }
    for all D_i ∈ D do
        V ← V ∪ { M_{D:V}[D_i] }
    end for
```

16: **return** $\mathcal{D}, \mathcal{V}, M_{\mathcal{D}:\mathcal{V}}$
17: **end function**

6.6 An Upper Bound on Symmetricity

This section is an application of using degree trees to obtain an upper bound on the symmetricity of a graph G. Our upper bound is easier to compute than determining the symmetricity directly, since our technique does not need to consider any particular labelling $\mathbf{f}$ of the graph G.

Definition 6.6. The number of vertices of a graph G with degree x is denoted as $\#_x(G)$.

Below is a proposition from Yamashita and Kameda [90, p. 73] that gives a relationship between the multiplicity of a labelling and the number of vertices of any fixed degree d.

Proposition 6.6. *For any labelling* $\boldsymbol{f}$ *on any graph* G, $s_{\boldsymbol{f}} \mid \#_d(G)$, *for* $d \in \{ 1, 2, \ldots, n-1 \}$.

Example 6.4. In Figure 6.8, $\#_1(G) = 0$, $\#_2(G) = 0$, $\#_3(G) = 2$, and $\#_4(G) = 3$, as there are 5 vertices in G. Note that Proposition 6.6 implicitly implies that $s_{\mathbf{f}} = 1$ for any labelling $\mathbf{f}$, since 1 is the only integer that divides each $\#_d(G)$. This also indicates that $\sigma(G) = 1$ in this case — no matter which labelling is used, each vertex will have a dissimilar view.

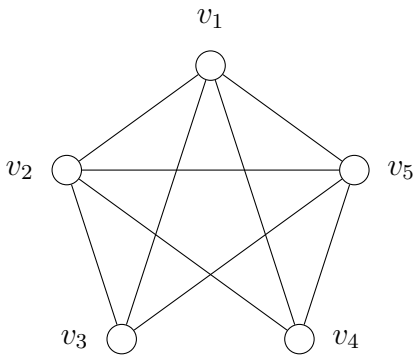


Figure 6.8: A graph G whose symmetricity is 1

Proposition 6.6 further implies that $s_{\mathbf{f}}$ divides the gcd of each $\#_d(G)$ for $d \in \{\, 1, 2, \ldots, n-1 \,\}$, and we define

$$\gamma(G) = \gcd(\#_1(G), \#_2(G), \ldots, \#_{n-1}(G)).$$

Therefore, $s_{\mathbf{f}} \mid \gamma(G)$ for any labelling $\mathbf{f}$; naturally $\sigma(G) \mid \gamma(G)$ since $\sigma(G)$ is the largest possible $s_{\mathbf{f}}$ by definition. Although $\gamma(G)$ itself can be used as an upper bound on the precise value of $\sigma(G)$, we next introduce a better one based on degree trees.

Definition 6.7. The number of vertices of a graph G with degree tree D is denoted as $\#_D(G)$, which is equal to $|V_D|$ for $V_D \in \mathcal{V}$.

Similarly, we define

$$\mu(G) = \gcd(\#_{D_1}(G), \#_{D_2}(G), \ldots, \#_{D_\kappa}(G)) = \gcd(|V_{D_1}|, |V_{D_2}|, \ldots, |V_{D_\kappa}|),$$

where $1 \leq \kappa \leq n$.

Below is the algorithm to calculate $\mu(G)$.

Algorithm 6.13 Compute $\mu(G)$

Require: $\mathcal{V}$ is generated from a graph G

```
function ComputeMuG(V)
    μ ← |V_{D_1}|
    for all V_{D_i} ∈ V do
        μ ← gcd(μ, |V_{D_i}|)
    end for
    return μ
end function
```

Since a degree tree D of a vertex contains more information than merely a degree of that, $\mu(G)$ performs at least as well as $\gamma(G)$ when it comes to an upper bound of a graph G's symmetricity $\sigma(G)$.

Proposition 6.7. *For a graph G under any labelling $\boldsymbol{f}$, we have $s_{\boldsymbol{f}} \leq \sigma(G) \leq \mu(G) \leq \gamma(G) \leq n$.*

Proof. By Definition 2.17, $s_{\mathbf{f}} \leq \sigma(G)$. Then we argue $\sigma(G) \leq \mu(G)$. Let $\#_D$ be the number of vertices with some degree tree $D_i \in \mathcal{D} = \{ D_1, D_2, \ldots, D_\kappa \}$, and we can write $\#_{D_i} = \sum |V_{T_i}|$, where the summation is taken over all $T_i \in \mathcal{T}$ such that T_i is a view under $\mathbf{f}$ isomorphic to D_i (when all port labels are ignored). According to Proposition 2.1, each term $|V_{T_i}|$ in that sum is equal to $s_{\mathbf{f}}$, so $s_{\mathbf{f}} \mid \#_{D_i}$. By definition, there exists a labelling $\mathbf{f}$ that makes $s_{\mathbf{f}} = \sigma(G)$. Since $\sigma(G) \mid \#_{D_i}$ for any degree tree D_i, it follows that $\sigma(G)$ divides the gcd of $\#_{D_1}, \ldots, \#_{D_i}, \ldots, \#_{D_\kappa}$. Therefore, $\sigma(G) \mid \mu(G)$ and $\sigma(G) \leq \mu(G)$.

By a similar argument, we show $\mu(G) \leq \gamma(G)$. Let $\#_x$ be the number of vertices with degree x. We have $\#_x = \sum \#_{D_i}$, where the summation is taken

over all $D_i \in \mathcal{D}$ such that the root of D_i has degree x. Since each term in the sum is divisible by $\mu(G)$ (by definition of $\mu(G)$), $\mu(G) \mid \#_x$. This indicates that $\mu(G)$ divides the gcd of $\#_1, \ldots, \#_x, \ldots, \#_{n-1}$, which implies $\mu(G) \mid \gamma(G)$ and $\mu(G) \leq \gamma(G)$. ■

Corollary 6.2. *If* $\mu(G) = 1$*, then* $\sigma(G) = 1$.

The chain of inequalities in Proposition 6.7 demonstrates that $\mu(G)$ is at least as tight an upper bound on $\sigma(G)$ as $\gamma(G)$ is. Example 6.5 gives a situation where the bound is strictly tighter.

Example 6.5. In Figure 6.9, both graphs have six vertices. In the left-hand side one G_1, $\#_2(G_1) = 4$, $\#_3(G_1) = 2$ and $\gamma(G_1) = \gcd(4, 2) = 2$; meanwhile, we have $\sigma(G_1) = \mu(G_1) = 1 \leq \gamma(G_1)$. For the right-hand side one G_2, $\#_2(G_2) = 2$, $\#_3(G_2) = 4$ and $\gamma(G_2) = \gcd(4, 2) = 2$; we have $\sigma(G_2) = \mu(G_2) = 1 \leq \gamma(G_2)$ as well.

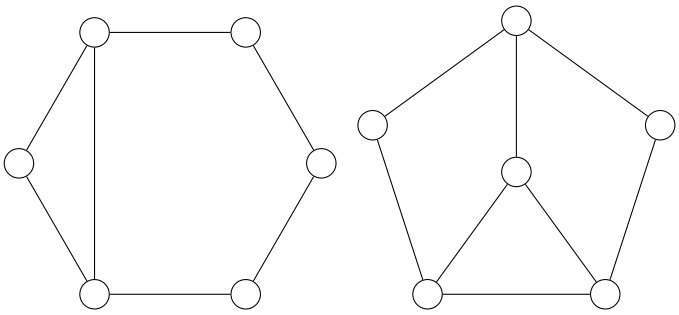


Figure 6.9: Two 6-vertex graphs G_1 and G_2

6.7 Symmetricity of Trees

This section is dedicated to proving a theorem that describes the symmetricity of trees, based on Proposition 6.8 [90, p. 76].

Proposition 6.8. *If a graph G is a tree, then $\sigma(G) \leq 2$.*

Our main result is the following theorem that characterizes exactly when $\sigma(G) = 1$ versus $\sigma(G) = 2$.

Theorem 6.2. *A tree G has $\sigma(G) = 2$ iff G consists of two isomorphic rooted subtrees whose roots are v_r and v'_r with a bridge (v_r, v'_r).*

These subtrees mentioned in Theorem 6.2 are induced. For any tree, all its $n - 1$ edges are in fact its bridges. We set up a series of lemmas below to prove the "only if" part of Theorem 6.2.

Lemma 6.3. *Given a tree $G = (V, E)$ under a labelling $\boldsymbol{f}$, for each vertex $v \in V$ and any two of its neighbours $v_i, v_j \in N_G(v)$, $T_{\boldsymbol{f}}(v_i) \not\equiv T_{\boldsymbol{f}}(v_j)$.*

In other words, a vertex in a port-labelled tree cannot have two neighbours which have similar views.

Proof by contradiction. Given such a tree, its multiplicity can only either be 1 or 2 according to Proposition 6.8.

Case 1. Suppose $s_{\mathbf{f}} = 1$ for some labelling $\mathbf{f}$. By definition, all the vertices in such a tree have the dissimilar views.

Case 2. Suppose $s_{\mathbf{f}} = 2$ for some labelling $\mathbf{f}$. There exists such a vertex v connecting two neighbours v_i and v_j with similar view T, and then there must exist another vertex v' ($v' \neq v$) connecting two neighbours v'_i and v'_j with similar view T.

Subcase (i): If $v_i \neq v'_i$ or $v_j \neq v'_j$. Then we have at least three vertices with similar view T, which contradicts our assumption.

Subcase (ii): If $v_i = v'_i$ and $v_j = v'_j$, i.e., v and v' are adjacent to both v_i and v_j. Then G must contain a cycle, which contradicts the fact that G is a tree, as shown in Figure 6.10, where each colour (black or white) represents a dissimilar view.

■

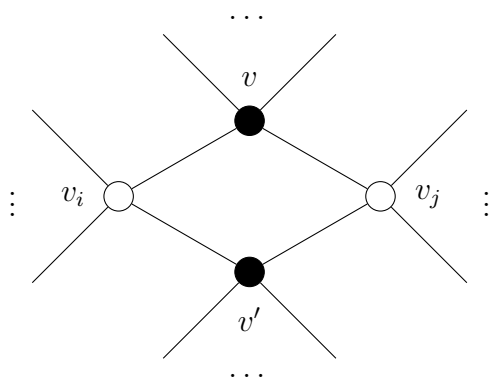


Figure 6.10: Case 2, subcase (ii) of Lemma 6.3

Lemma 6.4. *For a tree under a labelling $\boldsymbol{f}$ with $s_{\boldsymbol{f}} = 2$, there exists one and only one bridge (v_r, v'_r), such that $T_{\boldsymbol{f}}(v_r) \equiv T_{\boldsymbol{f}}(v'_r)$.*

Proof by contradiction. Based on Proposition 6.8, the tree's symmetricity

must be 2 and the number of vertices in such a tree must be even. We have $|\mathcal{T}_{\mathbf{f}}| = n/2$ and for any $T_i \in \mathcal{T}_{\mathbf{f}}$, $|V_{T_i}| = 2$.

Case 1. Suppose that there exist more than one bridge whose two endpoints have similar views. Let $e_1 = (v_r, v'_r)$ and $e_2 = (v, v')$ be two such bridges, where $v_r, v'_r \in V_{T_1}$, $v, v' \in V_{T_2}$ and $T_1 \not\equiv T_2$. Without loss of generality, let v'_r be the endpoint of e_1 and let v' be the endpoint of e_2 such that v_r and v lie on the path between v'_r and v'. In order to share a similar view with v_r, v'_r must have another path containing a vertex whose view is equivalent to T_2.

By the choice of v'_r, there is a path $(v'_r, v_r, v_w, \ldots, v, v')$ in the graph. Since v_r's view is similar to v'_r's view, there needs to be a path $(v_r, v_x, \ldots, v_y, v_z)$ where v_x's view is T_1, and v_y and v_z have view T_2. Then v_x must be v'_r since there are only two vertices with view T_1; v_y and v_z must be v and v' because they are the only vertices with view T_2. This proves that there exists a path between v_r and v where v_r's neighbour is v_w and there exists a path between v_r and v where v_r's neighbour is v'_r — since v_w and v'_r cannot be the same vertex (which would imply three vertices with view T_1), it contradicts that the graph is a tree.

Case 2. Suppose that there exists no such a bridge whose two endpoints have similar views. Let $v_r, v'_r \in V_T$ and $T(v_r) \equiv T(v'_r)$. By assumption, v_r and v'_r are not neighbours to one another. Since G is a tree, there is exactly one path between v_r and v'_r, which is $(v_r, u_0, u_1, \ldots, u_k, v'_r)$. For each u_i, let T_i be the view of u_i, and a sequence of the views mapping the vertices on the path between v_r and v'_r is $(T(v_r), T_0, T_1, \ldots, T_k, T(v'_r))$, shown in Figure 6.11. Since

the views of v_r and v'_r are similar, there must be a path between v'_r and v_r such that the sequence of the views along this path is $(T(v'_r), T_0, T_1, \ldots, T_k, T(v_r))$. These two paths must be the same; otherwise, there exists a cycle in the tree and a contradiction appears. Hence,

$$(T(v_r), T_0, T_1, \ldots, T_k, T(v'_r)) = (T(v'_r), T_0, T_1, \ldots, T_k, T(v_r)).$$

More specifically, $T_i \equiv T_{k-i}$ for all i.

Subcase (i): If k is odd, then the middle edge $(v_{(k-1)/2}, v_{(k+1)/2})$ has two endpoints with similar view $T_{(k-1)/2} \equiv T_{(k+1)/2}$, which contradicts our assumption.

Subcase (ii): If k is even, then the two neighbours of $v_{k/2}$ have similar views, which contradicts Lemma 6.3.

■

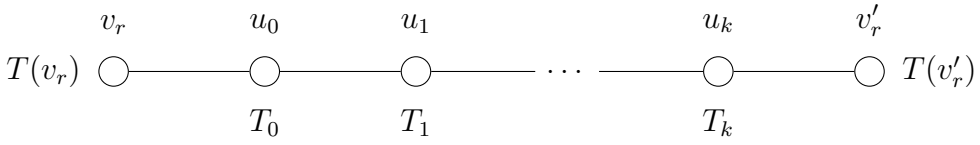


Figure 6.11: The path between two vertices v_r and v'_r

The next result shows that the only edge (v_r, v'_r) connecting two vertices with similar views, has the same port on its two endpoints.

Corollary 6.3. *For any tree under a labelling $\boldsymbol{f}$ with $s_{\boldsymbol{f}} = 2$, and any edge (v_r, v'_r) such that $T_{\boldsymbol{f}}(v_r) \equiv T_{\boldsymbol{f}}(v'_r)$, we have that $f_{v_r}(v'_r) = f_{v'_r}(v_r)$.*

Proof by contradiction. Let $x = f_{v_r}(v'_r)$, $y = f_{v'_r}(v_r)$ and $T \equiv T_{\mathbf{f}}(v_r) \equiv T_{\mathbf{f}}(v'_r)$. Assume that $x \neq y$. In $T_{\mathbf{f}}(v_r)$, the root has one child with view T, and the directed edge are labelled with the ordered pair of ports (x, y). In $T_{\mathbf{f}}(v'_r)$, the root has one child with view T, and the outgoing and incoming ends of the directed edge are labelled with (y, x). Since v_r and v'_r are the only two vertices with view T, this means that there is no other directed edge from the root of $T_{\mathbf{f}}(v'_r)$ to a node with view T (and hence there is no such edge labelled as (x, y)). This demonstrates that $T_{\mathbf{f}}(v_r) \not\equiv T_{\mathbf{f}}(v'_r)$, which contradicts Lemma 6.4. ■

Corollary 6.4. *Consider any tree under a labelling $\boldsymbol{f}$ with $s_{\boldsymbol{f}} = 2$, and any edge (v_r, v'_r) such that $T_{\boldsymbol{f}}(v_r) \equiv T_{\boldsymbol{f}}(v'_r)$. For any v, v' other than v_r, v'_r such that $T_{\boldsymbol{f}}(v) \equiv T_{\boldsymbol{f}}(v')$, a path between v and v' always go through the edge (v_r, v'_r).*

Proof. From Lemma 6.3, two vertices with similar views cannot be neighbours of the same vertex. And as we did in Case 2 inside the proof of Lemma 6.4, we can prove there is an edge in the middle of the path between v and v' such that both endpoints have similar views. By Lemma 6.4, we conclude that these two vertices are v_r and v'_r, which proves that the path between v and v' contains the edge (v_r, v'_r). ■

The following lemma is the "only if" direction of Theorem 6.2.

Lemma 6.5. *If a tree G has $\sigma(G) = 2$, then G consists of two isomorphic rooted subtrees whose roots are v_r and v'_r with a bridge (v_r, v'_r).*

Proof by induction. Since $\sigma(G) = 2$, we assume that there is a labelling $\mathbf{f}$ on G with $s_{\mathbf{f}} = 2$. By Lemma 6.4, there is exactly one bridge (v_r, v'_r) such that $T_{\mathbf{f}}(v_r) \equiv T_{\mathbf{f}}(v'_r)$. In total n vertices in such a tree are divided into two disjoint sets: let V_r be the set of the vertices that are closer to v_r than v'_r, and, let V'_r be the set of the vertices that are closer to v'_r than v_r. By doing so, G is divided into three components: a subtree G_1 with vertex set V_r whose root is v_r, a subtree G_2 with vertex set V'_r whose root is v'_r, and a bridge (v_r, v'_r) connecting G_1 and G_2. Then we redefine the level of each vertex v here to be $\min(d(v, v_r), d(v, v'_r))$. There exists an isomorphism $\psi\colon V_r \to V'_r$ such that, if $\psi(v_x) = v_y$, then the sequence of ports from v_r to v_x is identical to the sequence of ports from v'_r to v_y. Let ℓ be the particular level of a subtree.

Base case. When $\ell = 0$, there is an isomorphism ψ such that $\psi(v_r) = v'_r$, since both of the sequences are empty.

Inductive step. We try to prove that if there exists an isomorphism ψ from all vertices in V_r at level ℓ to all vertices in V'_r at level ℓ, then the same isomorphism ψ maps all vertices in V_r at level $\ell + 1$ to all vertices in V'_r at level $\ell + 1$. Let v_w be a vertex in V_r at level $\ell + 1$ and v_x be the parent of v_w (i.e., the only neighbour of v_w at level ℓ). Let $p = f_{v_x}(v_w)$ and there must exist $v_y = \psi(v_x)$ in V'_r at level ℓ. Now, port p from v_y must lead to a vertex at level $\ell + 1$ in V'_r due to the induction hypothesis: the exact same port sequence from v_r to v_x appears on the path from v'_r to v_y. Especially, if port p leads from v_y to level $\ell - 1$, then the same port p would lead from v_x to

level $\ell - 1$. So we let v'_w be a vertex at level $\ell + 1$ in V'_r that can be reached by following port p from v_y and define $\psi(v_w) = v'_w$.

Next, it remains to show that the edge (v_x, v_w) is labelled in the same way as (v_y, v'_w). First, $f_{v_x}(v_w) = f_{v_y}(v'_w) = p$, because of how we chose v'_w. We then try to prove by contradiction that $f_{v_w}(v_x) = f_{v'_w}(v_y)$. Assume $f_{v_w}(v_x) \neq f_{v'_w}(v_y)$. Let S represent the sequence of ports from v_r to v_x. By the induction hypothesis, the same sequence S is the port sequence from v'_r to v_y. So the port sequence from v_r to v_w is S concatenating the ordered pair of ports (p, i), i.e., $S \cdot (p, i)$, where $i = f_{v_w}(v_x)$. And the port sequence from v'_r to v'_w is $S \cdot (p, i')$, where $i' = f_{v'_w}(v_y)$. If $i \neq i'$, then there is a port sequence in $T_{\mathbf{f}}(v_r)$ that does not exist in $T_{\mathbf{f}}(v'_r)$, which contradicts $T_{\mathbf{f}}(v_r) \equiv T_{\mathbf{f}}(v'_r)$.

In conclusion, two rooted subtrees G_1 and G_2 of G are isomorphic. ■

To finish the proof of Theorem 6.2, we prove its "if" direction, which is the following lemma.

Lemma 6.6. *If a tree G consists of two isomorphic rooted subtrees whose roots are v_r and v'_r with a bridge (v_r, v'_r), then G has $\sigma(G) = 2$.*

Proof. Let G_1 and G_2 be two isomorphic rooted subtrees of a tree G. We use the function ArbitraryLabel from Algorithm 5.8 on G_1 to give it an arbitrary labelling. The same labelling can be applied on G_2 in view of their isomorphism. Then there remains an unlabelled bridge connecting G_1 and G_2 whose two ends v_r and v'_r are the roots of G_1 and G_2. The ports for two

ends of the bridge (v_r, v'_r) are $\deg(v_r) = \deg(v'_r)$. As a total labelling done symmetrically, its multiplicity must be 2. ■

Example 6.6. In Figure 6.12, the red-coloured bridge divides two isomorphic components whose roots are v_r and v'_r and the tree has symmetricity 2.

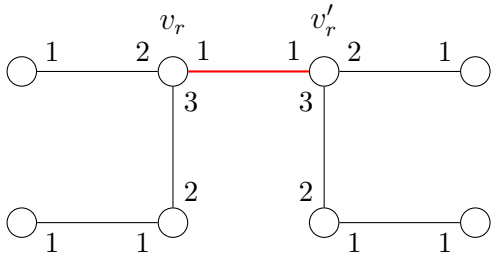


Figure 6.12: Two isomorphic subtrees with a bridge connecting two roots

7

Producing Asymmetry

> No two trees are the same to Raven. No two branches are the same to Wren.
>
> *Lost*
> David Wagoner

Provided that a total labelling has multiplicity 1 or a partial one is guaranteed to have multiplicity 1, we say that it is **asymmetric**. In this chapter, we partially prove that there is always a labelling that allows the graph to be completely asymmetric, i.e., the resulting multiplicity is 1. This will help us answer Subproblem 1 of Problem 5.

When the number of vertices for a graph is equal to 2, there is only one possible labelling on such a graph (i.e. the smallest complete graph K_2) and the port-labelled graph has multiplicity 2. The rest of this chapter considers the case where $n > 2$, and we make progress towards solving the following conjecture proposed by ourselves.

Conjecture 7.1. *For any graph G with $n > 2$, there always exists a labelling $\boldsymbol{f}$ such that $s_{\boldsymbol{f}} = 1$.*

Informally speaking, we attempt to offer a specially-designed labelling on any graph such that where all vertices have dissimilar views, or equivalently, prove that LE is solvable on such a port-labelled graph. The core idea behind it is to transform a partial labelling $\mathfrak{f}$ on G whose multiplicity is *potentially* larger than 1, into a slightly different partial labelling $\mathfrak{f}'$ whose multiplicity is *guaranteed* to be 1, by switching two ports on one carefully-selected vertex. In order to achieve that, each major step of our labelling algorithm (given in Algorithm 7.6 from Section 7.3) is listed below:

1. For a graph $G = (V, E)$ with $|V| = n$, we generate degree trees of each vertex using the function FillAD from Algorithm 6.7.

2. Then we generate degree tree collection $\mathcal{D} = \{ D_1, D_2, \ldots, D_\kappa \}$ and degree tree vertex partition $\mathcal{V} = \{ V_{D_1}, V_{D_2}, \ldots, V_{D_\kappa} \}$, where $1 \leq \kappa \leq n$, using the function GenerateDVM from Algorithm 6.12.

3. There are three mutually exclusive cases:

   Case 1. $\kappa \geq 3$.

   Case 2. $\kappa = 2$.

   Subcase (i): G is biregular.

   Subcase (ii): G is non-biregular.

   Case 3. $\kappa = 1$.

Subcase (i): G is $\{1,2\}$-factorable.

Subcase (ii): G is non-$\{1,2\}$-factorable.

Since the first two steps have already been implemented previously in Chapter 6, we now separately consider each case of Step 3. The following approach for an asymmetric partial labelling with port-switching will be used below when handling several of the cases. The three cases of Step 3 will be handled separately in Section 7.2.

7.1 Asymmetric Partial Labellings

In this section, we describe an algorithm that will be used in cases where there is a degree tree class V_D such that every vertex $v \in V_D$ (with degree tree D) has at least two neighbours with dissimilar degree trees. We choose such a $\widehat{V_D}$, and port-label all the vertices in $\widehat{V_D}$ identically, except exactly one of them (called v_{unique}). We then switch its ports on two edges connecting two neighbours with dissimilar degree trees. This process will result in an asymmetric labelling because switching the ports will make the view of v_{unique} dissimilar at level 1 from all other vertices in $\widehat{V_D}$, let alone the rest of the vertices in the same graph. From Proposition 2.1, the multiplicity is 1.

In Algorithm 7.1, the first-encountered degree tree class satisfying the condition above is returned. Line 3 indicates that port-switching is feasible when V_{D_i} contains at least two vertices. If there exist at least two dissimilar degree trees among v_j's neighbours ($v_j \in V_{D_i}$), V_{D_i} itself is returned as $\widehat{V_D}$

(in line 8).

Algorithm 7.1 Select $\widehat{V_D}$ from $\mathcal{V}$

```
1: function SelectVD(G, A_D, 𝒱)
2:     for all V_{D_i} ∈ 𝒱 do
3:         if |V_{D_i}| > 1 then
4:             v ← an arbitrary vertex in V_{D_i}
5:             temp ← ∅
6:             for all v_j ∈ N_G(v) do
7:                 if D(v_j) ∉ temp then
8:                     if temp ≠ ∅ then
9:                         return V_{D_i}
10:                    else
11:                        temp ← temp ∪ { D(v_j) }
12:                    end if
13:                end if
14:            end for
15:        end if
16:    end for
17: end function
```

Once it is achieved, we can do an asymmetric partial labelling to make sure that the multiplicity of any total labelling based on such a labelling (in Algorithm 7.3 mentioned later) is fixed to be 1. We define a new function.

Definition 7.1. The number of v's neighbours in a graph $G = (V, E)$ belonging to a degree tree class V_D is denoted as $\mathscr{N}_G(v, D)$, or in other words, the number of its neighbours with degree tree D.

Corollary 7.1. *For a graph $G = (V, E)$ and every $v \in V$, $\sum_{i=1}^{\kappa} \mathscr{N}_G(v, D_i) = \deg(v)$, for $\kappa = |\mathcal{D}|$ and $D_i \in \mathcal{D}$.*

Proof. There are κ dissimilar degree trees in total and each vertex can only own exactly one degree tree. The notation $\mathscr{N}_G(v, D_i)$ represents the number of v's neighbours with a specific degree tree D (which may be zero sometimes) and the sum of each $\mathscr{N}_G(v, D_i)$ is the number of its neighbours. By definition, it is equal to $\deg(v)$. ■

To compute $\mathscr{N}_G(v, D)$, one option is to create a `dictionary` for each vertex v whose key is a degree tree D and value is the number of v's neighbours with D for optimal execution time. However, to simplify the code, the function ReturnNG from Algorithm 7.2 is invoked implicitly every time when the expression $\mathscr{N}_G(v, D)$ occurs.

Algorithm 7.2 Return $\mathscr{N}_G(v, D)$ for a vertex v and a degree tree D

Require: $v \in V_G$ and $D \in \mathcal{D}$

```
1: function ReturnNG(G, A_D, v, D)
2:     num ← 0
3:     for all v_i ∈ N_G(v) do
4:         if D(v_i) = D then
5:             num ← num + 1
6:         end if
7:     end for
8:     return num
9: end function
```

We have Algorithm 7.3 to do a partial labelling for each vertex $v \in \widehat{V_D}$, such that their (incomplete) views up to level 1 are similar (although we are far from finishing the total labelling); its practicality follows from Theorem 6.1. In line 7, for each $D_j \in \{ D_1, D_2, \ldots, D_\kappa \}$, if the number of v_i's neighbours with a specific degree tree D_j is not zero, then we assign the ports from $\left\{ \sum_{x=1}^{j-1} \mathscr{N}_G(v_i, D_x) + 1, \ldots, \sum_{x=1}^{j} \mathscr{N}_G(v_i, D_x) \right\}$ without repetition to the edges incident to the neighbours with degree tree D_j. V_{D_j} is returned by referencing $M_{\mathcal{D}:\mathcal{V}}[D_j]$ while v_k is a neighbour of v_i with degree tree D_j (line 8).

Algorithm 7.3 Do a partial labelling on the vertices from $\widehat{V_D}$

```
1:  function PartialLabelVD(G, V̂_D, 𝒟, M_{𝒟:𝒱})
2:      n ← |V̂_D|
3:      f ← an empty 2-dimensional n-by-n array
4:      for all v_i ∈ V̂_D do
5:          unused ← 1
6:          for all D_j ∈ 𝒟 do
7:              if 𝒩_G(v_i, D_j) > 0 then
8:                  for all v_k ∈ N_G(v_i) ∩ M_{𝒟:𝒱}[D_j] do
9:                      f_{v_i}(v_k) ← unused
10:                     unused ← unused + 1
11:                 end for
12:             end if
13:         end for
14:     end for
15:     return f
16: end function
```

After Algorithm 7.3 has port-labelled the vertices in $\widehat{V_D}$, we make a vertex

arbitrarily (here, $v_{\text{unique}} \in \widehat{V_D}$) "unique" among any other vertices in $\widehat{V_D}$ by switching its ports on the edges incident to the neighbours from different degree tree classes. Immediately, the view of v_{unique} becomes dissimilar compared to its counterparts in $\widehat{V_D}$, even if it is just a partial labelling for the vertices in $\widehat{V_D}$ — suppose that a graph has been partially port-labelled, and that for some degree tree D, no vertex with degree tree D has had its ports labelled yet. As a consequence, the resulting multiplicity must be 1 regardless of how the rest of the labelling is defined. This is accomplished using the following algorithm, which makes use of the functions SelectVD (Algorithm 7.1), PartialLabelVD (Algorithm 7.3), and SwitchPorts (swapping the smallest port and the largest port, taken from Algorithm 5.6) defined before. The effect of the code in Algorithm 7.4 line 8 is shown in Figure 7.1 (the same colour represents the vertices with similar degree trees and the question marks indicate yet unlabelled ports): before the execution of line 8, $f_{v_{\text{unique}}}(v) = 1$ and $f_{v_{\text{unique}}}(v') = \delta$, for $D(v) \not\equiv D(v')$ and $\delta = \deg(v_{\text{unique}})$; after port-switching, $f_{v_{\text{unique}}}(v) = \delta$ and $f_{v_{\text{unique}}}(v') = 1$.

Algorithm 7.4 Do an asymmetric partial labelling on a graph

1: **function** AsymmetricPartialLabel($G, A_D, \mathcal{D}, \mathcal{V}, M_{\mathcal{D}:\mathcal{V}}$)
2: $\quad\widehat{V_D} \leftarrow \varnothing$
3: $\quad\widehat{V_D} \leftarrow$ SelectVD($G, A_D, \mathcal{V}$)
4: $\quad n \leftarrow |V_G|$
5: $\quad\mathfrak{f} \leftarrow$ an empty 2-dimensional n-by-n `array`
6: $\quad\mathfrak{f} \leftarrow$ PartialLabelVD($G, \widehat{V_D}, \mathcal{D}, M_{\mathcal{D}:\mathcal{V}}$)
7: $\quad v_{\text{unique}} \leftarrow$ an arbitrary vertex in $\widehat{V_D}$
8: $\quad\mathfrak{f} \leftarrow$ SwitchPorts($\langle G, \mathfrak{f} \rangle, \{ v_{\text{unique}} \}$)

9: **return** f
10: **end function**

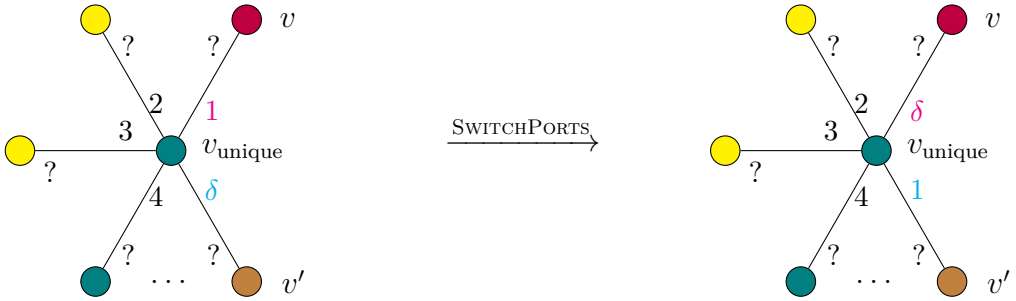


Figure 7.1: The port change of v_{unique} after line 8

7.2 Three Cases of Step 3

7.2.1 Case 1

According to Proposition 6.5, we can always call the function AsymmetricPartialLabel from Algorithm 7.4 on G with $\kappa \geq 3$.

7.2.2 Case 2

When $\kappa = 2$ (and $\mathcal{V} = \{ V_{D_1}, V_{D_2} \}$), G can either be biregular (Proposition 6.3) or non-biregular. For G with $\kappa = 2$ to be biregular, all the vertices in V_{D_1} connect only the ones in V_{D_2} and vice versa. We have Algorithm 7.5 to check if such G is biregular or not. As we are assuming that $\kappa = 2$ in this case, the algorithm picks one arbitrary vertex v from two degree tree classes V_{D_1} and V_{D_2}, and examines if v has any neighbour from the same degree tree class.

Algorithm 7.5 Check if a graph with $\kappa = 2$ is biregular

Require: $\kappa = 2$, i.e., $\mathcal{V} = \{ V_{D_1}, V_{D_2} \}$

```
1: function IfBiregular(G, 𝒱)
2:     for all V_{D_i} ∈ 𝒱 do
3:         v ← an arbitrary vertex in V_{D_i}
4:         for all v_i ∈ N_G(v) do
5:             if v_i ∈ V_{D_i} then
6:                 return False
7:             end if
8:         end for
9:     end for
10:    return True
11: end function
```

Furthermore, if G is (x, y)-biregular (the function IfBiregular on it returns True) with $\gcd(|V_{D_1}|, |V_{D_2}|) = 1$, then $\gcd(x, y) = 1$ (Corollary 6.1) and its symmetricity is actually 1. Although we can use the function ArbitraryLabel from Algorithm 5.8 to give such a graph an arbitrarily total labelling, here we do a partial labelling on it by applying the function AsymmetricPartialLabel from Algorithm 7.4 for consistency. The case of $\gcd(|V_{D_1}|, |V_{D_2}|) \neq 1$ ($\gcd(x, y) \neq 1$) remains unsolved (see Section B.1).

If G is non-biregular (the function IfBiregular on it returns False instead), there exists at least one vertex whose neighbours have two dissimilar degree trees due to Proposition 6.4 — so applying the function AsymmetricPartialLabel is sufficient to create a graph with multiplicity 1.

7.2.3 Case 3

When $\kappa = 1$, G must be regular, which is guaranteed by Proposition 6.2: it can be either $\{1, 2\}$-factorable or non-$\{1, 2\}$-factorable.

If G is $\{1, 2\}$-factorable, there always exists at least one labelling $\mathbf{f}$ whose multiplicity is n (Corollary 2.1). For such a subcase, one possible solution is to invoke the function FullySymmetricLabel from Algorithm 5.4 and switch two ports of one of its vertices by using the function SwitchPorts from Algorithm 5.6. Instead, we again use the function AsymmetricPartialLabel.

If G is non-$\{1, 2\}$-factorable (regular but without 1-factors; see Corollary 5.1), we do not currently have a solution, so we exclude this case from our final algorithm. There are some elementary results in Section B.2.

7.3 The Final Algorithm

We combine the above observations to produce Algorithm 7.6, which will ensure that the multiplicity becomes 1 in the cases we know how to solve. The function If12Factorable in Algorithm 7.6 line 23 is from Algorithm 5.1. Except the biregular graphs whose size of two bipartitions are not co-prime and the non-$\{1, 2\}$-factorable regular graphs, Conjecture 7.1 holds.

Algorithm 7.6 Do an asymmetric partial labelling on any graph

```
1: function AsymmetricPartialLabelAny(G)
2:     n ← |V_G|
```

```
    A_D ← an empty 2-dimensional n-by-(n − 1) array
    A_D ← FillAD(G)
    𝒟 ← ∅
    𝒱 ← { ∅ }
    M_{𝒟:𝒱} ← an empty dictionary
    𝒟, 𝒱, M_{𝒟:𝒱} ← GenerateDVM(G, A_D)
    κ ← |𝒱|                                   ▶ |𝒱| = |𝒟|
    ƒ ← an empty 2-dimensional n-by-n array
    if κ ≥ 3 then
        ƒ ← AsymmetricPartialLabel(G, A_D, 𝒟, 𝒱, M_{𝒟:𝒱})
    else
        if κ = 2 then
            if IfBiregular(G, 𝒱) then
                if gcd(|V_{D_1}|, |V_{D_2}|) = 1 then
                    ƒ ← AsymmetricPartialLabel(G, A_D, 𝒟, 𝒱, M_{𝒟:𝒱})
                end if
            else
                ƒ ← AsymmetricPartialLabel(G, A_D, 𝒟, 𝒱, M_{𝒟:𝒱})
            end if
        else                                  ▶ G is regular
            if If12Factorable(G) then
                ƒ ← AsymmetricPartialLabel(G, A_D, 𝒟, 𝒱, M_{𝒟:𝒱})
            end if
        end if
    end if
    return ƒ
end function
```

8 Future Work

> You can never plan the future by the past.
>
> Edmund Burke

In this thesis, we considered problems of two types:

1. Can we provide a port-labelled graph to satisfy certain symmetry requirements? (Problems 1, 2 and 3)

2. For a port-labelled graph, can we quantify its symmetry? (Problems 4 and 5)

In Chapter 5, we solved all three problems of the first type by creating an algorithm that produces a fully symmetric labelling (Algorithm 5.4), an algorithm that produces a total labelling on a complete graph with a desired multiplicity (Algorithm 5.7), and an algorithm that generates more instances with rooted products (Algorithm 5.12). For the second type, we provided

an algorithm that computes the multiplicity of a graph (Algorithm 6.5) as a solution of Problem 4 in Section 6.1. Unfortunately, Problem 5 was only partially solved. We made partial progress on its Subproblem 1: in terms of the minimum multiplicity of a graph, in Chapter 7, we came up with an algorithm that asymmetrically port-labels most families of graphs such that their multiplicities become 1 (Algorithm 7.6); for the maximum multiplicity (symmetricity), we provided an upper bound using degree trees in Section 6.6. For its Subproblem 2, we offered two examples falling into two cases in Sections 5.3 and 5.4 respectively. Additionally, the trees with symmetricity 2 are characterized precisely in Section 6.7. And we failed to make any progress on its Subproblem 3.

Subproblems 1, 2 and 3 of Problem 5

1. Given a graph, determine lower/upper bounds on its multiplicities.
2. Given a graph G, determine if every d between the extrema can be a valid multiplicity $s_{\mathbf{f}}$ for some labelling $\mathbf{f}$ on G.
3. Given a graph, list all possible multiplicities and at least one of their corresponding labellings.

Along with what we mentioned above, here are two more conjectures which can potentially be our next step.

Conjecture 8.1. *For a graph G and its* connected *complement* $\overline{G}$[1], $\mu(G) =$

[1]The complement of a graph $G = (V, E)$ is a graph H with the same vertex set V such

$\mu(\overline{G})$ *and* $\sigma(G) = \sigma(\overline{G})$.

Definition 8.1. A graph is **planar** if it can be drawn in the plane such that none of its edges intersects each other except their ends.

Definition 8.2. A graph G is **vertex-transitive** if given any two vertices v_1 and v_2, there exists a bijection $\psi\colon G \to G$ such that $\psi(v_1) = v_2$.

A graph is vertex-transitive iff its graph complement is; every vertex-transitive graph is regular, but not visa versa (e.g., the Frucht graph mentioned in Figure 4.1 is not).

Conjecture 8.2. *If a graph G is either planar or vertex-transitive, $\mu(G) = \sigma(G)$.*

Also, in order to decrease the running time of algorithms, there are three approaches to reduce considerably the complexity of the graph that we are operating on (without losing crucial graph invariants): covering graphs [5, p. 86] (later evolved into graph fibrations [15]), quotient graphs [90, pp. 76–82] and graph amalgamation (mainly appeared in graph embedding and factorization research works). Like the f-view [84] mentioned on page 28, we may apply the same technique to prune the degree tree directly, removing redundant nodes from it.

Besides, we would like to find a relatively large gap between the various quantities in the chain of inequalities $s_{\mathbf{f}} \leq \sigma(G) \leq \mu(G) \leq \gamma(G) \leq n$ from Proposition 6.7.

---

that its edges are the pair of nonadjacent vertices in G.

Finally, the reconstruction problem (proposed by K. Gunderson, personal communication, June 18, 2021) is still open to answer: can a vertex-unlabelled graph under a labelling always be constructed from its views? The answer is negative when the multiplicity is n. If the multiplicity is instead 1, then it is highly possible that a graph can be reconstructed from its collection of views: each vertex can be labelled according to its dissimilar view and then adjacency can be extracted from the views. If reconstruction is possible, it might be interesting to know the minimum number of views required to reconstruct.

A

The Graph Construction

> Ever tried. Ever failed. No matter. Try again. Fail again. Fail better.
>
> *Worstward Ho*
> Samuel Beckett

We here prove the existence of non-1-factorable k-regular n-vertex graph with at least one 1-factor by directly constructing such an example (as demonstrated by K. Gunderson, personal communication, June 18, 2021). For a k-regular graph with $k = 2$, the existence of a 1-factor implies the existence of a 1-factorization. The following proof is based on the fact that for a sufficient large n and $k \geq 2\lceil n/4 \rceil - 1$, every k-regular graph with n vertices has a 1-factorization [31].

Let $k \geq 4$ be even and $n = 4r + 2$, which is sufficiently large; let H be any k-regular Hamiltonian graph with $n/2 = 2r + 1$ vertices. We define a new graph G by taking two copies of H: H_1 and H_2. Then we fix an edge

$e = (v, w)$ on the Hamiltonian cycle of H. We remove the edge $e_1 = (v_1, w_1)$ and $e_2 = (v_2, w_2)$ (the copies of e in each of the copies of H) from G, and we add the new edges (v_1, v_2) and (w_1, w_2) to G. Now, the graph G is k-regular and has a perfect matching — if we remove the edge (v_1, v_2), then every second edge along the two Hamiltonian cycles forms a perfect matching.

Claim A.1. The graph G has no 1-factorization.

Proof by contradiction. Assume that G does have a 1-factorization. Let M_1 and M_2 be the perfect matchings containing the edges e_1 and e_2 respectively. The two matchings are not equal since the graph $H \setminus (v, w)$ has an odd number of vertices (and so no perfect matching of the remaining vertices). The graph $G - (M_1 \cup M_2)$ is a disconnected graph that has at least two components, in which every vertex has degree $k - 2 \geq 2$. Since each of the components is contained either in the vertices of H_1 or H_2 (both have an odd number of vertices), one of the components is odd and hence the graph has no perfect matching. ■

B

Two Unsolved Cases

Trifles make perfection, and
perfection is no trifle.

Lacon
Charles Caleb Colton

The appendix contains our unfinished work for Conjecture 7.1 — for any graph G with $n > 2$, there always exists a labelling $\mathbf{f}$ such that $s_{\mathbf{f}} = 1$. Its two unsolved cases are (x, y)-biregular graphs with $\gcd(x, y) \neq 1$, and all non-$\{1, 2\}$-factorable regular graphs.

B.1 Biregular Graphs

When a graph $G = (U \cup V, E)$ is (x, y)-biregular and $\gcd(x, y) > 1$, there are two remaining cases to discuss its minimum multiplicity according to De Morgan's laws when $\gcd(|U|, |V|) > 1$:

Case 1. $|U| \mid |V|$ or $|V| \mid |U|$;

Case 2. $|U| \nmid |V|$ and $|V| \nmid |U|$.

There are two possible ways to analyze these two cases: either fix $|U|$ and $|V|$, or fix x and y. Both of them will cover all possibilities, but here we prefer the former one on the ground that: when we fix x and y, we actually have no idea about $|U|$ and $|V|$, let alone $n = |U| + |V|$ itself. In a better way to say, $|U|$ and $|V|$ are unbounded and in some situation, are unreachable. For example, for a $(4, 6)$-biregular graph, we know that $4|U| = 6|V|$ from Corollary 6.1; $|U| = 3$ and $|V| = 2$ apparently fit the equation. However, when $|U| = 3$ and $|V| = 2$, the corresponding *complete* bipartite graph is only $(2, 3)$-biregular. If we fix $|U|$ and $|V|$ instead, all possible x and y are available, the degree sequences of all generated graphs are graphic.

Example B.1. Given an (x, y)-biregular graph $G = (U \cup V, E)$ where $|U| = a = 12$ and $|V| = b = 4$, the pairs of (x, y) can be $(2, 6)$, $(3, 9)$ or $(4, 12)$. Now make (x, y) be $(3, 9)$ for instance and each vertex from U connects three out of four vertices from V. Here we are showing a failed attempt in Figure B.1 to make a graphic $(3, 9)$-biregular graph with in total 16 vertices by adding edges one by one. We let $u_1, u_2, \ldots, u_9$ only connect v_1, v_2 and v_3. Now all u_1 to u_9 have degree 3 and v_1, v_2 and v_3 have degree 9. Under the situation, u_{10}, u_{11} and u_{12} cannot have degree 3 and v_4 cannot have degree 9 since there do not exist enough vertices to do so. And in fact, the only possible way to make this $(3, 9)$-biregular graph valid is to assign three vertices from U to connect v_1, v_2 and v_3, three ones to connect v_2, v_3 and v_4, three ones to connect v_1, v_3 and v_4 and the remaining three ones to connect v_1, v_2 and v_4.

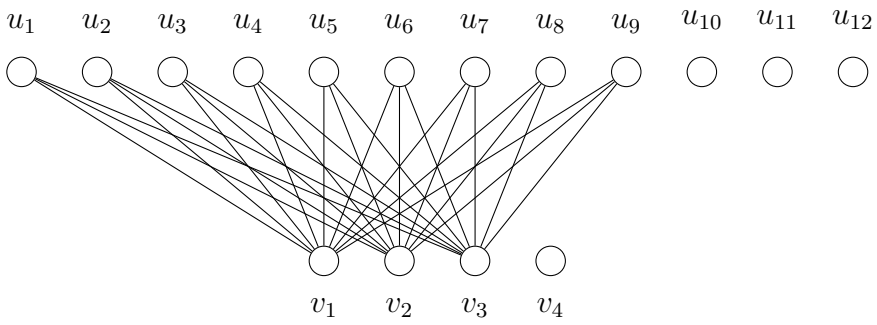


Figure B.1: A failed attempt to create a graphic biregular graph

For Case 1 (e.g. $n = 9$ when $|U| = 6$ and $|V| = 3$), we have the following conjecture.

Conjecture B.1. *If G is a biregular graph $G = (U \cup V, E)$ with n vertices and either $|U| \mid |V|$ or $|V| \mid |U|$, then $\mu(G) = \sigma(G) = \min(|U|, |V|)$.*

Theorem B.1. *Every subgraph G' of a bipartite graph G is, itself, bipartite.*

Theorem B.2. *If $G = (U \cup V, E)$ is a k-regular bipartite graph, then the number of vertices in U is equal to that of vertices in V.*

According to the theorems above along with Theorem 5.4, we know that 1-factors require even n and given a bipartite graph G with two bipartitions U and V — if $|U| \neq |V|$, then there does not exist a 1-factor for G. Since for a regular bipartite graph, we can always call the function PortLabel1Factor from Algorithm 5.2 to make it fully symmetric, we can derive Conjecture B.2 from Conjecture B.1.

Conjecture B.2. *For an (x, bx)-biregular graph $G = (U \cup V, E)$, there exists a bipartite x-regular subgraph in G.*

Here we use a simple-word example to instantiate Conjecture B.2: suppose that there are ab people ($a > 1$ and $b > 1$) and a bins; each person has x coins to put into x out of a bins ($2 \leq x \leq a$) without repetition such that after every one finishes, each bin has exactly bx (or y) coins. And Conjecture B.2 is transferred to that we can always pick a group of a people from ab ones, such that they, as a whole, put the same number of coins (which is x) in each a bins. And the beauty of Conjecture B.2 is that if it holds, we are guaranteed to keep applying it recursively by removing such a subgraph from the original graph. For example, if we have 18 people and six bins ($a = 6$ and $b = 3$), and such a group of six people do exist, then we remove these six people from 18 ones to leave 12 ones now ($b' = b - 1 = 2$ now) and repeat the process. Therefore, from Conjecture B.2, we imply the following one further.

Conjecture B.3. *An (x, bx)-biregular graph $G = (U \cup V, E)$ is the union of b edge-disjoint bipartite x-regular subgraphs.*

We have two potential directions to solve Conjecture B.3: design theory and hypergraph theory.

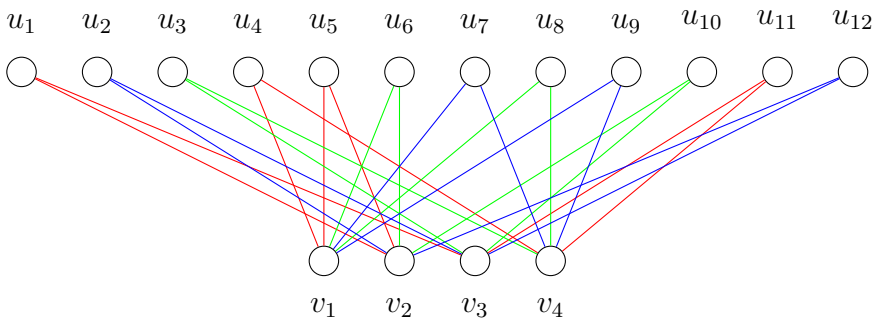


Figure B.2: An example of Conjecture B.3 for a $(2, 6)$-biregular graph with 16 vertices

B.1.1 Designs

Let t, k, v and λ be integers with $0 < t \leq k < v$ and $\lambda > 0$. And a t-(v, k, λ) **design**, or t-**design**, an ordered pair $(X, \mathcal{B})$, where X is a finite set of v points and $\mathcal{B}$ is a family of k-subsets of X (called **blocks** of the design), such that each t-subsets of X appears in preciously λ blocks [19, p. 257].

If we treat blocks $\mathcal{B}$ as the vertex set U of a bipartite graph $(U \cup V, E)$, X as the vertex set V (correspondingly v as the size of V), k as the degree of vertices in U, λ as the degree of vertices in V, and more importantly make $t = 1$, then we convert our graph decomposition problem (Conjecture B.3) into a combinatorial-design-theoretic one: any 1-(a, x, bx) design can be partitioned into b 1-(a, x, x) sub-designs, which are all **symmetric** (the number of points equals the number of blocks) and not necessarily **simple** (no repeated blocks allowed). The 1-(a, x, bx) design here is also **non-trivial** $(2 \leq k < v)$, **regular** (every point appears in the same number of blocks) and

uniform (every block contains the same number of points, k).

Obviously, a biregular graph is regarded as a design whose points and blocks are its vertices and edges. It is a 1-$(n, 2, k)$ design if it has n vertices and its valency is k [20, p. 57]. A **parallel class** or **resolution class** in a design is a set of blocks that partition the point set. Let α be a positive integer. An α**-parallel** class or α**-resolution** class in a design is a set of blocks containing every point of the design exactly α times [28, p. 130]. In other words, we try to prove that all non-simple 1-(a, x, bx) designs are x-resolvable.

Next are some potentially useful results [28, p. 25].

Proposition B.1. *If a* (v, k, λ)*-design has a proper* (ω, k, λ)*-subdesign, then* $\omega \leq \frac{v-1}{k-1}$.

Proposition B.2. *There exists a* 1*-*(v, k, λ) *design iff* $v\lambda \equiv 0 \pmod{k}$.

B.1.2 Hypergraphs

A hypergraph can be viewed as a generalization of a graph [86, pp. 135–136; 17, pp. 1–3], which we focused on previously. A **hypergraph** is an ordered pair $H = (V, \mathfrak{E})$ where V is a set of n vertices and $\mathfrak{E}$ is a set of m non-empty subsets of V called **hyperedges**. Noticeably, $\mathfrak{E}$ may contain the same set more than once, and the **multiplicity** of $\mathfrak{e} \in \mathfrak{E}$ is denoted by $m_H(\mathfrak{e})$. If H allows repeated hyperedges, H is **multiple**. A hypergraph is k**-uniform** if all its hyperedges have size k and it is d**-regular** if all its

vertices have degree d. Any hypergraph $H' = (V', \mathfrak{E}')$ such that $V' \subseteq V$ and $\mathfrak{E}' \subseteq \mathfrak{E}$ is called a **subhypergraph** of H. And for a hypergraph $H = (V, \mathfrak{E})$, any subhypergraph $H' \subseteq H$ such that $H' = (V, \mathfrak{E}')$ is called a **partial subhypergraph** (or spanning subhypergraph). A hypergraph H is called an **intersecting family** if all of its edges pairwise intersect.

Let $V = \{ v_1, v_2, \ldots, v_n \}$ and $\mathfrak{E} = \{ \mathfrak{e}_1, \mathfrak{e}_2, \ldots, \mathfrak{e}_m \}$. The n-by-m **incidence matrix** of a hypergraph $H = (V, \mathfrak{E})$ is a $(0,1)$-matrix $A = (a_{i,j})$ where

$$a_{i,j} = \begin{cases} 1, & \text{if } v_i \in \mathfrak{e}_j \\ 0, & \text{otherwise.} \end{cases}$$

And easily we can see that the incidence matrix of H is just the biadjacency matrix of the original graph.

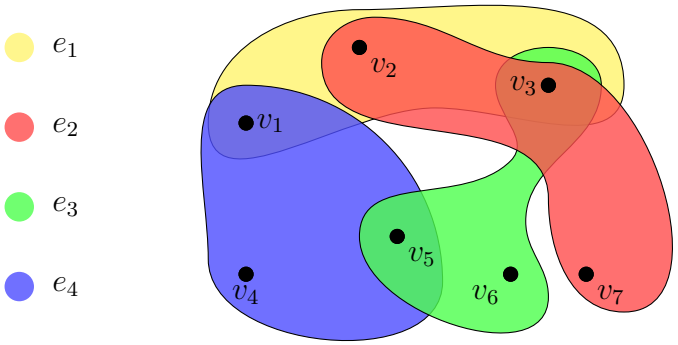


Figure B.3: A 3-uniform non-regular hypergraph with seven vertices and four hyperedges

Hence, Conjecture B.3 becomes the following form in the context of hypergraph theory: all multiple x-uniform bx-regular hypergraphs with abx

hyperedges contain at least one x-uniform x-regular partial subhypergraph with a hyperedges.

B.2 Non-{1, 2}-Factorable Regular Graphs

We have to explain more on non-$\{1,2\}$-factorable regular graphs. The following theorem is a characterization of graphs with 1-factors.

Theorem B.3 (Tutte's Theorem (1947))**.** *A graph $G = (V, E)$ has a 1-factor iff for each subset $U \subseteq V$, the induced subgraph with the vertex set $V - U$ has at most $|U|$ connected components with an odd number of vertices.*

Corollary B.1. *A k-regular graph is non-$\{1,2\}$-factorable iff k is odd (and the number of vertices is even) and it contains no 1-factors.*

Conjecture B.4. *If G is a k-regular graph with n vertices without 1-factors, where k is odd and n is even, then $\sigma(G) = 1$.*

From Lemma 2.1, for such a graph G, $\sigma(G) \neq n/2$. Therefore, $\sigma(G) < n/2$. Here is another related theorem [87].

Theorem B.4. *If G is a k-regular graph with n vertices without 1-factors and no odd components, where k is odd, then $n \geq 3k + 7$.*

Bibliography

[1] C. Aguerre, T. Morsellino, and M. Mosbah. Fully-distributed debugging and visualization of distributed systems in anonymous networks. In P. Richard, M. Kraus, R. S. Laramee, and J. Braz, editors, *GRAPP & IVAPP 2012: Proceedings of the International Conference on Computer Graphics Theory and Applications and International Conference on Information Visualization Theory and Applications, Rome, Italy, 24-26 February, 2012*, pages 764–767. SciTePress, 2012. 4

[2] A. V. Aho and J. E. Hopcroft. *The design and analysis of computer algorithms*. Pearson Education India, 1974. 95

[3] B. Alspach. The wonderful walecki construction. *Bull. Inst. Combin. Appl*, 52(52):7–20, 2008. 56

[4] E. Ando, H. Ono, K. Sadakane, and M. Yamashita. The space complexity of leader election in anonymous networks. *International Journal of*

Foundations of Computer Science, 21(03):427–440, 2010. 31

[5] D. Angluin. Local and global properties in networks of processors. In *Proceedings of the twelfth annual ACM symposium on Theory of computing*, pages 82–93. Acm, 1980. 6, 27, 29, 124

[6] D. Angluin, J. Aspnes, Z. Diamadi, M. J. Fischer, and R. Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed computing*, 18(4):235–253, 2006. 3

[7] M. Åstrand and J. Suomela. Fast distributed approximation algorithms for vertex cover and set cover in anonymous networks. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, pages 294–302, 2010. 30

[8] H. Attiya and F. Ellen. Impossibility results for distributed computing. *Synthesis Lectures on Distributed Computing Theory*, 5(1):1–162, 2014. 4

[9] H. Attiya and M. Snir. Better computing on the anonymous ring. *Journal of Algorithms*, 12(2):204–238, 1991. 30, 31

[10] H. Attiya, M. Snir, and M. K. Warmuth. Computing on an anonymous ring. *Journal of the ACM (JACM)*, 35(4):845–875, 1988. 29, 30

[11] L. Barrière, P. Flocchin, P. Fraigniaud, and N. Santor. Can we elect if we cannot compare? In *Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, pages 324–332, 2003. 33

[12] M. Bauderon, Y. Métivier, M. Mosbah, and A. Sellami. Graph relabelling systems: a tool for encoding, proving, studying and visualizing distributed algorithms. *Electronic Notes in Theoretical Computer Science*, 51:93–107, 2002. 28

[13] P. W. Beame and H. L. Bodlaender. Distributed computing on transitive networks: the torus. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 294–303. Springer, 1989. 36

[14] P. Blanchard and R. Guerraoui. On the smallest grain of salt to get a unique identity. In *International Colloquium on Structural Information and Communication Complexity*, pages 106–121. Springer, 2017. 30

[15] P. Boldi and S. Vigna. Fibrations of graphs. *Discrete Mathematics*, 243(1-3):21–66, 2002. 124

[16] J. A. Bondy and U. S. R. Murty. *Graph Theory*, volume 244. Springer-Verlag London, 2008. 47, 55

[17] A. Bretto. *Hypergraph theory*. Springer, 2013. 133

[18] R. A. Brualdi and J. J. Q. Massey. Incidence and strong edge colorings of graphs. *Discrete Mathematics*, 122(1-3):51–58, 1993. 40

[19] P. J. Cameron. *Combinatorics: topics, techniques, algorithms*. Cambridge University Press, 1994. 132

[20] P. J. Cameron, J. H. Van Lint, and P. J. Cameron. *Designs, graphs, codes and their links*, volume 3. Cambridge University Press, 1991. 133

[21] D. M. Campbell and D. Radford. Tree isomorphism algorithms: Speed vs. clarity. *Mathematics Magazine*, 64(4):252–261, 1991. 95

[22] J. Chalopin, S. Das, and A. Kosowski. Constructing a map of an anonymous graph: Applications of universal sequences. In *International Conference On Principles Of Distributed Systems*, pages 119–134. Springer, 2010. 34

[23] J. Chalopin, S. Das, and N. Santoro. Groupings and pairings in anonymous networks. In *International Symposium on Distributed Computing*, pages 105–119. Springer, 2006. 30

[24] J. Chalopin, E. Godard, and Y. Métivier. Election in partially anonymous networks with arbitrary knowledge in message passing systems. *Distributed Computing*, 25(4):297–311, 2012. 27

[25] J. Chalopin, E. Godard, Y. Métivier, and R. Ossamy. Mobile agent algorithms versus message passing algorithms. In *International Conference On Principles Of Distributed Systems*, pages 187–201. Springer, 2006. 33

[26] J. Chalopin and Y. Métivier. An efficient message passing election algorithm based on mazurkiewicz's algorithm. *Fundamenta Informaticae*, 80(1-3):221–246, 2007. 29

[27] T. Chothia and K. Chatzikokolakis. A survey of anonymous peer-to-peer file-sharing. In *International Conference on Embedded and Ubiquitous Computing*, pages 744–755. Springer, 2005. 3

[28] C. J. Colbourn and J. H. Dinitz. *Handbook of combinatorial designs*. CRC press, 2006. 133

[29] B. Conrad and F. Shirazi. A survey on tor and i2p. In *Ninth International Conference on Internet Monitoring and Protection (ICIMP2014)*, pages 22–28, 2014. 3

[30] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press, 2009. 19, 20, 95

[31] B. Csaba, D. Kühn, A. Lo, D. Osthus, and A. Treglown. *Proof of the 1-factorization and Hamilton decomposition conjectures*, volume 244. American Mathematical Society, 2016. 126

[32] J. Czyzowicz, A. Kosowski, and A. Pelc. How to meet when you forget: log-space rendezvous in arbitrary graphs. *Distributed Computing*, 25(2):165–178, 2012. 35

[33] S. Das, P. Flocchini, S. Kutten, A. Nayak, and N. Santoro. Map construction of unknown graphs by multiple agents. *Theoretical Computer Science*, 385(1-3):34–48, 2007. 33, 34

[34] S. Das, P. Flocchini, A. Nayak, and N. Santoro. Effective elections for anonymous mobile agents. In *International Symposium on Algorithms and Computation*, pages 732–743. Springer, 2006. 29

[35] S. Das, P. Flocchini, N. Santoro, and M. Yamashita. Fault-tolerant simulation of message-passing algorithms by mobile agents. In *International Colloquium on Structural Information and Communication Complexity*, pages 289–303. Springer, 2007. 33

[36] S. Das, M. Mihalák, R. Šrámek, E. Vicari, and P. Widmayer. Rendezvous of mobile agents when tokens fail anytime. In *International Conference On Principles Of Distributed Systems*, pages 463–480. Springer, 2008. 35

[37] A. K. Datta, S. Devismes, L. L. Larmore, and V. Villain. Self-stabilizing weak leader election in anonymous trees using constant memory per edge. *Parallel Processing Letters*, 27(02):1750002, 2017. 31

[38] C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, A.-M. Kermarrec, E. Ruppert, and H. Tran-The. Byzantine agreement with homonyms. In *Proceedings of the 30th annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 21–30, 2011. 27

[39] D. Dereniowski, A. Kosowski, and D. Pająk. Distinguishing views in symmetric networks: A tight lower bound. *Theoretical Computer Science*, 582:27–34, 2015. 28

[40] D. Dereniowski and A. Pelc. Drawing maps with advice. *Journal of Parallel and Distributed Computing*, 72(2):132–143, 2012. 33

[41] D. Dereniowski and A. Pelc. Leader election for anonymous asynchronous agents in arbitrary networks. *Distributed Computing*, 27(1):21–38, 2014. 33

[42] Y. Dieudonné and A. Pelc. Impact of knowledge on election time in anonymous networks. *Algorithmica*, 81(1):238–288, 2019. 30, 32

[43] K. Diks, S. Dobrev, E. Kranakis, A. Pelc, and P. Ružička. Broadcasting in unlabeled hypercubes with a linear number of messages. *Information Processing Letters*, 66(4):181–186, 1998. 36

[44] K. Diks, P. Fraigniaud, E. Kranakis, and A. Pelc. Tree exploration with little memory. *Journal of Algorithms*, 51(1):38–63, 2004. 34

[45] K. Diks, E. Kranakis, A. Malinowski, and A. Pelc. Anonymous wireless rings. *Theoretical Computer Science*, 145(1-2):95–109, 1995. 35

[46] K. Diks, E. Kranakis, and A. Pelc. Broadcasting in unlabeled tori. *Parallel Processing Letters*, 8(02):177–188, 1998. 36

[47] S. Dobrev, J. Jansson, K. Sadakane, and W.-K. Sung. Finding short right-hand-on-the-wall walks in graphs. In *International Colloquium on Structural Information and Communication Complexity*, pages 127–139. Springer, 2005. 34

[48] S. Dobrev and A. Pelc. Leader election in rings with nonunique labels. *Fundamenta Informaticae*, 59(4):333–347, 2004. 27

[49] P. L. Dordal. An introduction to computer networks, 2021. 1

[50] Y. Emek, C. Pfister, J. Seidel, and R. Wattenhofer. Anonymous networks: randomization = 2-hop coloring. In *Proceedings of the 2014 ACM symposium on Principles of distributed computing*, pages 96–105, 2014. 27

[51] P. Flocchini, E. Kranakis, D. Krizanc, F. L. Luccio, and N. Santoro. Sorting multisets in anonymous rings. In *Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000*, pages 275–280. Ieee, 2000. 30

[52] P. Flocchini, E. Kranakis, D. Krizanc, F. L. Luccio, and N. Santoro. Sorting and election in anonymous asynchronous rings. *Journal of Parallel and Distributed Computing*, 64(2):254–265, 2004. 35

[53] P. Flocchini, B. Mans, and N. Santoro. Sense of direction: Definitions, properties, and classes. *Networks: An International Journal*, 32(3):165–180, 1998. 39

[54] P. Flocchini, B. Mans, and N. Santoro. Sense of direction in distributed computing. *Theoretical Computer Science*, 291(1):29–53, 2003. 39

[55] P. Flocchini, A. Roncato, and N. Santoro. Backward consistency and sense of direction in advanced distributed systems. *SIAM Journal on Computing*, 32(2):281–306, 2003. 40

[56] P. Flocchini, A. Roncato, and N. Santoro. Computing on anonymous networks with sense of direction. *Theoretical Computer Science*, 301(1-3):355–379, 2003. 39

[57] P. Fraigniaud and A. Pelc. Decidability classes for mobile agents computing. In *Latin American Symposium on Theoretical Informatics*, pages 362–374. Springer, 2012. 28

[58] P. Fraigniaud, A. Pelc, D. Peleg, and S. Pérennes. Assigning labels in an unknown anonymous network with a leader. *Distributed Computing*, 14(3):163–183, 2001. 29

[59] E. G. Fusco and A. Pelc. Communication complexity of consensus in anonymous message passing systems. In *International Conference On Principles Of Distributed Systems*, pages 191–206. Springer, 2011. 32

[60] E. G. Fusco and A. Pelc. How much memory is needed for leader election. *Distributed Computing*, 24(2):65, 2011. 31

[61] E. G. Fusco and A. Pelc. Knowledge, level of symmetry, and time of leader election. *Distributed Computing*, 28(4):221–232, 2015. 29

[62] C. Glacet, A. Miller, and A. Pelc. Time vs. information tradeoffs for leader election in anonymous trees. *ACM Transactions on Algorithms (TALG)*, 13(3):1–41, 2017. 32

[63] J. Y. Halpern and S. Petride. A knowledge-based analysis of global function computation. *Distributed Computing*, 23(3):197–224, 2010. 31

[64] F. Harary. *Graph Theory (on Demand Printing of 02787)*. Taylor & Francis, 2018. 56

[65] L. Hella, M. Järvisalo, A. Kuusisto, J. Laurinharju, T. Lempiäinen, K. Luosto, J. Suomela, and J. Virtema. Weak models of distributed computing, with connections to modal logic. *Distributed Computing*, 28(1):31–53, 2015. 8

[66] J. M. Hendrickx. Views in a graph: to which depth must equality be checked? *IEEE Transactions on Parallel and Distributed Systems*, 25(7):1907–1912, 2013. 28

[67] D. Ilcinkas. Setting port numbers for fast graph exploration. *Theoretical Computer Science*, 401(1-3):236–242, 2008. 34

[68] R. E. Johnson and F. B. Schneider. Symmetry and similarity in distributed systems. In *Proceedings of the fourth annual ACM symposium on Principles of distributed computing*, pages 13–22, 1985. 27

[69] R. Klasing, A. Kosowski, and D. Pajak. Setting ports in an anonymous network: How to reduce the level of symmetry? In *International*

Colloquium on Structural Information and Communication Complexity, pages 35–48. Springer, 2016. 28

[70] H. Kobayashi, K. Matsumoto, and S. Tani. Simpler exact leader election via quantum reduction. *Chicago Journal of Theoretical Computer Science*, 10:2014, 2014. 28

[71] E. Kranakis and D. Krizanc. Distributed computing on cayley networks. In *1992 Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing*, pages 222–229. Ieee, 1992. 36

[72] E. Kranakis, D. Krizanc, and J. Vandenberg. Computing boolean functions on anonymous networks. *Information and Computation*, 114(2):214–236, 1994. 31, 36

[73] B. Mans. Optimal distributed algorithms in unlabeled tori and chordal rings. *Journal of Parallel and Distributed Computing*, 46(1):80–90, 1997. 36

[74] M. Mavronicolas, L. Michael, and P. G. Spirakis. Computing on a partially eponymous ring. In *International Conference On Principles Of Distributed Systems*, pages 380–394. Springer, 2006. 2, 27

[75] H. Meijer, Y. Núñez-Rodríguez, and D. Rappaport. An algorithm for computing simple k-factors. *Information processing letters*, 109(12):620–625, 2009. 53

[76] S. Micali and V. V. Vazirani. An $O(\sqrt{|v|} \cdot |E|)$ algorithm for finding maximum matching in general graphs. In *21st Annual Symposium on Foundations of Computer Science (sfcs 1980)*, pages 17–27. Ieee, 1980. 49

[77] A. Miller, A. Pelc, and R. N. Yadav. Deterministic leader election in anonymous radio networks. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 407–417, 2020. 28

[78] N. Norris. Universal covers of edge-labeled digraphs: Isomorphism to depth n-1 implies isomorphism to all depths. *Discrete Applied Mathematics*, 56(1):61–74, 1995. 13, 17, 85

[79] A. Pelc. Deterministic rendezvous algorithms. In *Distributed Computing by Mobile Entities*, pages 423–454. Springer, 2019. 35

[80] O. Reingold. Undirected connectivity in log-space. *Journal of the ACM (JACM)*, 55(4):1–24, 2008. 34

[81] N. Sakamoto. Comparison of initial conditions for distributed algorithms on anonymous networks. In *Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, pages 173–179, 1999. 32

[82] V. R. Syrotiuk, C. J. Colbourn, and J. Pachl. Wang tilings and distributed verification on anonymous torus networks. *Theory of Computing Systems*, 30(2):145–163, 1997. 36

[83] A. S. Tanenbaum and D. J. Wetherall. *Computer networks*. Prentice Hall, 5th edition, 2011. 2

[84] S. Tani. Compression of view on anonymous networks–folded view–. *IEEE Transactions on Parallel and Distributed Systems*, 23(2):255–262, 2011. 28, 124

[85] G. Tel. Sense of direction in processor networks. In *International Conference on Current Trends in Theory and Practice of Computer Science*, pages 50–82. Springer, 1995. 39

[86] V. I. Voloshin. *Introduction to graph and hypergraph theory*. Nova Science Publishers, 2009. 133

[87] W. Wallis. The smallest regular graphs without one-factors. *Ars Combin*, 11:295–300, 1981. 47, 135

[88] M. Yamashita and T. Kameda. Electing a leader when processor identity numbers are not distinct. In *International Workshop on Distributed Algorithms*, pages 303–314. Springer, 1989. 27

[89] M. Yamashita and T. Kameda. Computing functions on asynchronous anonymous networks. *Mathematical Systems Theory*, 29(4):331–356, 1996. 31

[90] M. Yamashita and T. Kameda. Computing on anonymous networks: part i—characterizing the solvable cases. *IEEE Transactions on parallel*

and distributed systems, 7(1):69–89, 1996. 2, 3, 6, 13, 16, 17, 27, 28, 32, 37, 99, 103, 124

[91] M. Yamashita and T. Kameda. Computing on anonymous networks: part ii—decision and membership problems. *IEEE Transactions on parallel and distributed systems*, 7(1):90–96, 1996. 3, 17