An Algorithmic And Computational Approach To Local Computations

Abstract : In this thesis, we study the algorithmic aspects of local computations in the area of distributed algorithms. We focus our attention on two special domains. These are network synchronization and execution control. We present new results concerning the synchronization of networks under the assumption that processes are aware of some network properties. The results presented here make use, amongst others, of randomization and of the SSP protocol introduced by Y. Shi, B. Szymanski and N. Prywest.

On the other hand, we take advantage of graph reduction algorithms to present an algorithmically solution for recognizing graph properties by local computations. To this end, we introduce the notion of handy reduction systems and prove, specially, that all properties on graphs of bounded treewidth, definable in the monadic second order logic, can be recognized by local computations. We also demonstrate that any handy reduction systems corresponds to a labeled graph recognizer with structural knowledge and, with the help of special kind of graphs morphisms : coverings, we derive necessary conditions for the size-recognizability of a graph property characterized by a handy reduction systems.

At the end, we introduce the programming language for implementing local computations (Lidia). This language is built on a two levels transition system and on the logic $\mathcal{L}^*_\infty$ used to describe the preconditions of each transition system. We exploit results from the finite model theory to prove the descriptive power of $\mathcal{L}^*_\infty$ and to show that any algorithm encoded by local computations can be implemented in Lidia.

Discipline : Computer-Science

Keywords : Distributed systems, graph reduction algorithms, formal languages, infinite logics.

Aspects Algorithmiques et Implémentation des Calculs Locaux

Résumé : Dans cette thèse nous nous intéressons aux aspects algorithmiques des calculs locaux dans les domaines de la synchronisation et du contrôle de l'exécution. Nous introduisons, entre autre, deux synchroniseurs dont l'un se base sur l'utilisation de l'algorithme de Y. Shi, B. Szymanski et N. Prywest et l'autre sur l'application des marches aléatoires dans les graphes.

Ensuite, nous utilisons le concept des réductions de graphes pour présenter un algorithme capable de reconnaître des propriétés de graphes à l'aide des calculs locaux. Cette étude introduit la notion de systèmes de réduction pratiques (handy reduction systems) qui nous permet de démontrer que toutes les propriétés de graphes de largeur arborescente bornée, définissable en logique monadique du second ordre, peuvent être reconnues par les calculs locaux. Nous établissons un lien directe entre les systèmes de réduction pratiques et le concept de reconnaisseurs de graphes avec connaissance structurelle (labeled graph recognizers). Ce lien nous permet de donner les conditions nécessaires pour qu'une propriété caractérisée par un système de réduction pratique soit reconnue au moyen de calculs locaux. Enfin, nous introduisons le langage de programmation des calculs locaux (Lidia). Ce langage est basé sur un système de transition à deux niveaux où les préconditions de chaque transition sont exprimées par la logique $\mathcal{L}^*_\infty$. En se servant des propriétés descriptives de $\mathcal{L}^*_\infty$, nous établissons la complétude du langage Lidia.

Discipline : Informatique

Mots-Clefs : Systèmes distribués, synchroniseurs, réduction de graphes, langages formels, logique.

LaBRI,
Université Bordeaux 1,
351 cours de la Libération,
33405 Talence Cedex (FRANCE).

N° d'ordre : 3067

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ BORDEAUX I

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

Par **Rodrigue Bertrand OSSAMY**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

An Algorithmic and Computational Approach to Local Computations

---

Soutenue le : 01 Décembre 2005

Après avis des rapporteurs :

Annegret HABEL ...	Professeur	
Wieslaw ZIELONKA	Professeur	

Devant la commission d'examen composée de :

Yves MÉTIVIER ...	Professeur	Directeur de thèse
Mohamed MOSBAH	Professeur	Examinateur
Vincent VILLAIN ...	Professeur	Président
Wieslaw ZIELONKA	Professeur	Rapporteur

– 2005 –

Remerciements

A journey is easier when you travel together. Interdependence is certainly more valuable than independence. This thesis is the result of three years of work whereby I have been accompanied and supported by many people. It is a pleasant aspect that I have now the opportunity to express my gratitude for all of them.

Mes remerciements s'adressent en premier leiu à mes deux directeurs de thèse : Yves Métivier et Mohamed Mosbah. Ils ont été présents tout au long de ces années. Je leur suis particulièrement reconnaissant d'avoir eu confiance en moi en acceptant de me guider dans mes recherches. Qu'ils trouvent ici toute l'expression de ma reconnaissance.

Je tiens à remercier chaleureusement Wieslaw Zielonka et Annegret Habel pour m'avoir fait l'honneur de lire attentivement mon mémoire. Leurs commentaires sur ce mémoire ont été très enrichissants. Je tiens à exprimer ma gratitude pour l'intérêt qu'ils ont porté à l'égard de mon travail.

Je remercie Vincent Villain d'avoir accepté de présider mon jury.

Je remercie Annegret Habel ainsi que toute son équipe de recherche pour l'acceuil qu'ils m'ont reservé lors de ma visite à Oldenburg (R. F. A.). La forme et l'organisation générale de ce mémoire sont en grande partie dues aux différentes remarques issues de nos entretiens. Ce fut un réel plaisir de travailler avec Annegret. Je suis heureux de pouvoir lui exprimer ici toute ma gratitude.
Les discussions avec Bruno Courcelles et Philippe Duchon, qui ne s'en souviendra peut être pas, m'ont apporté énormément de réponses dans les diverses thématiques abordées tout au long de ma thèse. Merci d'avoir pris le temps de m'apporter cette aide précieuse.

Je tiens également à remercier :

- Monsieur LEPO dont l'indéfectible soutien et les multiples conseils m'ont permis de mener à bien ce projet de recherche. Merci d'avoir toujours été là pour moi, d'avoir cru en moi même lorsque cela n'était pas évident.
- Matsy et tous mes enfants, dont l'exhaustive énumération donnerait certainement lieu à un nouveau mémoire. Merci pour votre patience.
- Tous les membres du projet ViSiDiA: Afif, Bilel, Hamid et Jérémie.

- Pierre Hanna, Rui Chen, Pascal Grange, Fabrice et tous les membres de l'équipe de foot de l'AFoDiB. L'esprit de «gagne» de cette équipe me manquera.
- David Renault, Pascal Ochem, Nicolas Bonichon et Michael Montassier pour avoir généralement pris le temps de partager leur expérience avec moi. Un grand merci à Nicolas dont le tapis de souris m'a permis de rédiger cette thèse. Merci aussi à David dont les connaissances en LaTeX sont «infinies».
- Aziz, Nader et Ben Mammar avec lesquels j'ai eu des discussions «extra recherches» très passionnantes.
- Marcien Mackaya, Michael Adelaide et Mouba Joan qui sont vraiment des types biens. Merci pour l'aide que vous m'avez apportée durant ces années. Les rires de Marcien et Mickael et surtout les remarques perfides de Joan ont généralement eu pour conséquence d'enlever un peu de tension à mon quotidien.

FOR YOU NOTHING SHOULD BE GOOD ENOUGH

RIEN NE SERA ASSEZ CONVENABLE POUR VOUS

EINMAL MEHR MOECHTE ICH MICH BEI DIR BEDANKEN

TO FULFILL OUR DREAMS AND EXPECTATIONS THAT IS WHAT WE HAVE TO

ZEIT IS MANGELHAFT ABER DER TAG WIRD KOMMEN

Merci!

Aspects algorithmiques et implémentation des calculs locaux

Résumé: Dans le modèle des calculs locaux, un système distribué est représenté par un graphe étiqueté dont les sommets correspondent aux processeurs et les arêtes aux liens de communication. Un algorithme distribué est alors décrit par un système de règles de transition local où l'étiquette suivante d'un sommet est fonction de son étiquette actuelle et de celles de ses voisins. Les réétiquetages opérant sur des voisinages disjoints peuvent se dérouler en parallèle, de manière synchrone ou asynchrone. Dans cette thèse nous nous intéressons aux aspects algorithmiques des calculs locaux dans les domaines de la synchronisation d'algorithmes et du contrôle de l'exécution. Dans un premier temps nous proposons différents protocoles de synchronisation qui ont besoin, pour certains, d'avoir une connaissance structurelle du graphe (diamètre, nombre de processeurs, etc...). Nous introduisons, entre autre, deux synchroniseurs dont l'un se base sur l'utilisation de l'algorithme SSP de Y. Shi, B. Szymanski et N. Prywest et l'autre sur l'application des marches aléatoires dans les graphes.

Ensuite, nous utilisons le concept des réductions de graphes pour présenter un algorithme capable de reconnaître des propriétés de graphes à l'aide des calculs locaux. Cette étude introduit la notion de systèmes de réduction pratiques (handy reduction systems) qui nous permet de démontrer que toutes les propriétés de graphes de largeur arborescente bornée, définissable en logique monadique du second ordre, peuvent être reconnues par les calculs locaux. Nous établissons un lien directe entre les systèmes de réduction pratiques et le concept de reconnaisseurs de graphes avec connaissance structurelle (labeled graph recognizers). Ce lien nous permet de donner les conditions nécessaires pour qu'une propriété caractérisée par un système de réduction pratique soit reconnue au moyen de calculs locaux.

Enfin, nous introduisons le langage de programmation des calculs locaux (Lidia). Ce langage est basé sur un système de transition à deux niveaux où les préconditions de chaque transition sont exprimées par la logique $\mathcal{L}^*_\infty$. En se servant des propriétés descriptives de $\mathcal{L}^*_\infty$, nous établissons la complétude du langage Lidia.

Mots-clés: Systèmes distribués, synchroniseurs, réduction de graphes, reconaisseur de graphes, election locale, langages de programmation, logiques infinies, théorie des modèles, snapshots

Discipline: Informatique

LaBRI,
Université Bordeaux 1,
351, cours de la libération
33405 Talence Cedex (FRANCE)

An Algorithmic and computational approach to local computations

Abstract : Local computations in graphs have been proved to be a powerful model for encoding distributed algorithms, for proving their correctness and for understanding their power. In this representation, a network is represented as a connected undirected graph where vertices denote processes and edges denote communication links. Within this framework, an algorithm is encoded by means of local relabellings. Labels attached to vertices and edges are modified locally, that is on a subgraph of fixed radius k of the input graph. In this thesis, we study the algorithmic aspects of local computations in the area of distributed algorithms. We focus our attention on two special domains. These are network synchronization and execution control. We present new results concerning the synchronization of networks under the assumption that processes are aware of some network properties. The results presented here make use, amongst others, of randomization and of the SSP protocol introduced by Y. Shi, B. Szymanski and N. Prywest.

On the other hand, we take advantage of graph reduction algorithms to present an algorithmically solution for recognizing graph properties by local computations. To this end, we introduce the notion of handy reduction systems and prove, specially, that all properties on graphs of bounded treewidth, definable in the monadic second order logic, can be recognized by local computations. We also demonstrate that any handy reduction systems corresponds to a labeled graph recognizer with structural knowledge and, with the help of special kind of graphs morphisms: coverings, we derive necessary conditions for the size-recognizability of a graph property characterized by a handy reduction systems.

At the end, we introduce the programming language for implementing local computations (Lidia). This language is built on a two levels transition system and on the logic $\mathcal{L}^*_\infty$ used to describe the preconditions of each transition system. We exploit results from the finite model theory to prove the descriptive power of $\mathcal{L}^*_\infty$ and to show that any algorithm encoded by local computations can be implemented in Lidia.

Keywords: Distributed systems, synchronizers, graph reduction algorithms, graph recognizers, local election, distributed languages, infinite and counting logics, model theory, snapshots based algorithms

Discipline: Computer Science

LaBRI,
Université Bordeaux 1,
351, cours de la libération
33405 Talence Cedex (FRANCE)

Contents

List of Figures

Introduction

Distributed Systems

A *distributed system* is a collection of individual computing devices (i.e. computers, processors, processes) that are interconnected by a network allowing them to communicate with each other. This very general definition covers a wide range of modern day computer systems, ranging from a VLSI chip, to a tightly-coupled shared memory multiprocessor, to a local-area cluster of workstations, to the Internet. Gerard Tel [Tel91] presents an overview of general topics related to the field of distributed systems. For further information concerning the simulation of distributed systems and their algorithmic approaches it is referred to the works of Nancy Lynch [Lyn96] and Attiya et al. [AW98].

There are two major *interprocess communication methods* in the field of distributed computations:

Communication by shared memory: Every two connected processes share a register on which they can performed read and write operations.

Communication by message passing: Each process can exchange informations with its neighbors. This is done by using the communication primitives *send* and *receive* that respectively allow a process to mail messages to its neighbors and to read its incoming mails.

The simplest, and certainly the oldest, example of a distributed system is an operating system for a conventional sequential computer. In this case processes on the same hardware communicate using the same software, either by exchanging messages or through a common address space. In order to time share a single CPU among multiple processes, as is done in most contemporary operating systems, issues relating to the (virtual) concurrency of the processes must be addressed. Many of the problems faced by an operating system also arise in other distributed systems, such as mutual exclusion and deadlock detection and prevention.

Within the scope of this thesis we will deal, unless stated otherwise, with the communication model based on message passing. This model is composed of two modes of operation depending on timing requirements. In fact, several different assumptions can be made about the timing of events in the system, reflecting the different types of timing information that might be used by algorithms. At one extreme, processes can be completely *synchronous*, performing communication and computation in perfect lock-step synchrony. At the other extreme, they can be completely *asynchronous*, taking steps at arbitrary speeds and in arbitrary

orders. More generally, if no timing assumptions at all are made, the system is said asynchronous. In a synchronous distributed system, the timing behavior is constrained by the following assumptions [HT94]:

- there is a known finite upper bound for the message transmission delay, and
- there is a known finite upper bound for the time required by any process to execute a step, and
- on every process, there is a local clock with a known finite upper bound for clock drift with respect to real time.

Modeling of distributed algorithms

An algorithm executed in a distributed system is called a *distributed algorithm*. A possible modeling of an asynchronous distributed algorithm consists in representing the network as a connected undirected graph where vertices denote processes and edges denote direct communication links. Labels attached to vertices and edges represent respectively the states of processes and communication links. The labels attached to vertices could also contain network informations such as processes identifiers, a distinguished process, the number of processes, the diameter of the graph or its topology. Weights, marks for encoding a spanning tree or the sense of direction are examples of labels attached to edges. Within this model, the execution of a distributed algorithm is then subdivided in computation steps. At each step, labels are modified locally, that is, on a subgraph of fixed radius k of the given graph, according to certain rules depending on the subgraph only (*local computations*). The relabeling is performed until no more transformation is possible, i.e., until a normal form is obtained.

This framework will be used as the standard model throughout the rest of this work. It has several interests:

- it gives an abstract model to think about some problems in the field of distributed computing independently of the wide variety of models used to represent distributed systems [LL90].
- as classical models in programming, it enables to build and to prove complex systems, and so, to get them right,
- it is easier to understand and to explain problems, to obtain solutions or impossibility results which remain true in weaker models,
- any positive solution in this model may guide the research of a solution in a weaker model or be implemented in a weaker model using randomized algorithms,
- this model gives nice properties and examples using classical combinatorial material.

The formalism pictured by the above model is similar to the models addressed by *P. Rosensthiel* or *D. Angluin*. In [RFH72] *Rosensthiel et al.* have introduced the concept of *intelligent graphs*. In their model, a distributed network system is pictured by a graph, having

maximal degree d, where each process is represented by a vertex acting as a finite automata. All automata execute the same underlying algorithm. The next state of a process is computed depending on its actual state and on the actual states of all its neighbors. Basically, this model is synchronous in the sense that each computation step changes simultaneously the states of each process. The model introduced by *D. Angluin* [Ang80] works on any type of graphs. In this representation, a computation step consists, for two neighboring process, in exchanging messages and computing their new states. This model is basically asynchronous.

Research domains in the area of distributed algorithms

Gerard Tel, in *Topics in Distributed Algorithms* [Tel91], has defined four major research domains in the area of distributed algorithms:

Fault tolerance: This is the study of the behavior of a distributed system in presence of processes failures. Generally, fault tolerance deals with the ability of a distributed system to continue normal operation despite the presence of hardware or software faults. A software fault occurs when a process does not break down, but executes a wrong protocol.

Communication: The task of this domain is to improve the efficiency of data transmission through the network. This includes the development of efficient routing algorithms in the distributed environment.

Synchronization: The goal here is to design protocols that are able to transform algorithms for the synchronous network model into algorithms for the fully asynchronous network model.

Execution control: In this area, one try to solve problems such as the *leader election in a network* or the termination detection that should allow processes to be aware of the end of the distributed computation. The objective is to have a mean to monitor each step of the execution of an asynchronous algorithm.

Topics of this thesis

In the range of this thesis, we focus our study on the last two domains. This is primary motivated by the importance of the synchronization paradigm in the study of distributed algorithms. As a matter of fact, the asynchronous network model has so much uncertainty (unknown number of processes, unknown network topology, independent inputs at different locations, process nondeterminism, uncertain message delivery times,...) that it is usually tedious to program directly in this environment. It is, therefore, desirable to have simpler distributed models that can be programmed more easily and whose programs can be translated into programs for the general asynchronous network.

Over and above, paying attention to the execution control of a distributed system is strongly related to problems concerning the modeling of distributed algorithms and the proof of their correctness. With respect to the modeling of distributed algorithms, we have introduced a programming language, called Lidia (*Language for implementing distributed algorithms*), for implementing algorithms encoded by means of local computations. The encoding of these

algorithms uses a set of computation rules (relabeling rules) that can be executes concurrently on disjoint parts of the network. Due to this concurrency, it can be tedious to show the correctness of the execution of such an algorithm. To do that, it is common to exhibit associated *invariant properties*. These properties generally allow to derive the correctness of the system. Hence, we have enhanced the Lidia platform with an invariants-checker based tool that is able to perform online or offline tests of the validity of given properties.

Another aspect of the domain of execution control we have considered is related to the distributed detection of network properties. We have developed a methodology (based on graph reduction rules) that, under certain conditions, says if a given network satisfies a given property or not. The importance of this paradigm is a direct consequence of the lack of global solutions in asynchronous distributed systems. That is, for a problem P, there could exist different algorithms for solving P and each of these solutions can be related to a unique networks class. As an instance, there exist two different algorithms for solving the election problem on tree-shaped networks and on networks represented by complete graphs. Detecting network properties is an important feature that permits to chose the adapted solution of a given distributed problem.

In the followings, we present how the rest of the thesis is organized.

Chapter 1: Preliminaries

This chapter presents an overview of the basic theoretical material that is useful for the understanding of the present work. First of all, we introduce the notion of *graph* and its related concepts. Thereafter, we define the terms of *labeled graphs* and *coverings* and take advantage of their characteristics to give a formal definition of the basics of local computations in graphs.

Part I: Synchronizers in the local computations framework

This part deals with the presentation of protocols which transform synchronous network algorithms into asynchronous algorithms. These protocols, called *synchronizers*, are of two types. The first one contains all synchronizers that work properly if and only if processes have knowledge about some network properties such as the size or the diameter. The second type includes the synchronizers that work faithfully without knowing anything about the underlying network. We will show that this kind of synchronizers can not guarantee the synchronization of two processes at distance at least 2 from each other. Nevertheless, we will state a wide variety of protocols, of the first type, that ensure a synchronization in the whole network. These synchronizers take advantage of various network properties and use algorithmic features such as randomization, or the SSP algorithm presented by Szymanski et al. [SSP85]. It will be stated that in tree-shaped networks, it is possible to construct a synchronizer that does not need more network knowledge. The end of this chapter will be devoted to the presentation of a new methodology that, given a synchronous algorithm and a synchronizer, constructs the asynchronous algorithm that simulates the execution of the synchronous network algorithm. All the protocols presented in this chapter are encoded by local computations.

Part II: Reduction based recognition of graphs properties with local computations

In this part we turn our attention to the distributed recognition of graph properties by local computations. We propose a solution based on the use of graph reduction algorithms in the local computation environment. The basic idea of graph reduction algorithms is to reduce the size of the input graph by an adequate set of graph transformation rules. Once the graph is irreducible, it is checked if the so obtained graph belongs to a given class of irreducible graphs. If this is the case, then it is deduced that the initial graph satisfies the property characterized by the set of reduction rules and the set of irreducible graphs. This part is covered by two chapters.

Chapter 4: A probabilistic algorithm for local election

A reduction rule consists in replacing a graph G through a graph G' having a smaller size. In a distributed environment we could face the case where parallel executions of reduction rules occur on overlapping subgraphs. To go around this problem, we take advantage of a k-local election algorithm to reach faithful reductions. To this end, we have developed an algorithm that is able to solve the k-local election problem. It is known that there is no Las Vegas algorithm for solving the k-local election problem for $k \geq 3$. For this reason we use a probabilistic protocol based on the distributed computation of rooted trees of minimal paths. The correctness of the election algorithm will be stated and we will introduce some computational heuristics that increase the success rate of the election algorithm.

Chapter 5: Recognition of graph properties with local computations

One of the most important constraint in the field of local computations resides in the fact that they never change the internal structure of the basic underlying graph. On account of this, we present in this chapter a way to encode basic graph reduction rules (vertex deletion, edge deletion, edge contraction and edge addition) in a distributed system with respect to the requirements of the local computations framework. This leads to the concept of *handy reduction rule*. Starting from this encoding system, we will present a methodology to encode reduction algorithms in our distributed environment. As a direct consequence, we will show that all properties of graphs of bounded treewidth definable in the monadic second order logic can be decided in a distributed way and within the framework of local computations. We will also state a straightforwards relationship between our encoding of reduction algorithms and the concept of *labeled graph recognizers* initiated by Godard, Métivier and Muscholl [GMM04]. At last we will present some improvements of *handy reduction rules* that allow, for certain classes of graph problems, to simultaneously solve a graph decision problem and compute a solution in order to reinforce the answer of the decision problem.

Chapter 6: Checking properties in distributed systems

In the last chapter we present a distributed algorithm that checks the validity of given invariants properties. The proposed algorithm takes advantage of an improved version of vector clocks [Lam78] to perform a partial order of *local snapshots*. Starting from these snapshots, the algorithm is able to generates all *states* through which the system has passed during the computation. This algorithm represents a tool that allows to automatically check the correctness of program executions in Lidia.

Part III: Implementing local computations: Lidia

This part is devoted to the presentation of the programming language Lidia. This language is based on a two-level transition system. The first level contains a number of transition systems, that define the behavior of a single process. The second level transition system consists of a single transition system that models the interactions among first-level transition systems. Each transition system is defined in a precondition-effect style, where the preconditions parts in Lidia are exclusively expressed in the logic $\mathcal{L}^*_\infty$. This logic is an extension of the first order logic by means of counting terms, counting quantifiers and infinite disjunctions and conjunctions.

Chapter 7: A language for implementing local computations

In this chapter we present the main concepts of Lidia. We focus our attention on the logic $\mathcal{L}^*_\infty$ and on the basic constructs that allow Lidia to fulfill the requirements of our computation model. We will also point out the implementation of the communication level and show that within this framework we can easily compute distributed algorithms encoded by means of local computations.

Chapter 8: Computational characteristics of Lidia

The main topic of this chapter is to state a relationship between the logical definability in $\mathcal{L}^*_\infty$ and the computational power of Lidia. Using results of the *finite model theory*, we show that if the rules preconditions of an algorithm encoded by local computations can be expressed in $\mathcal{L}^*_\infty$, then this algorithm can be implemented in Lidia. We will first state the descriptive complexity of $\mathcal{L}^*_\infty$ and derive the main result of this chapter. That is, all algorithms encoded by local computations and whose preconditions are evaluated in PTIME can be implemented in Lidia. We will also prove that in presence of *user defined functions*, Lidia is able to implement any kind of algorithms encoded in the local computations environment.

Chapter 1

Preliminaries

Contents

1.1 Graphs Properties

In this section we give an overview of the terminology that is used in this thesis. We assume that the reader is familiar with graph theory and algorithms. More background information can be found in e.g. Harary [Har69] for graph theory and Cormen, Leiserson and Rivest [CLR69] for algorithms. Finally, we will presented our distributed computation model and show how it can be encoded using graph relabeling systems.

1.1.1 Undirected Graphs

Definition 1.1 *A simple undirected graph G is a pair (V, E), where V is a set of vertices, and E is a set of edges. Each edge is an unordered pair of distinct vertices u and v, denoted by $\{u, v\}$. The sets of vertices and edges of a graph G are respectively denoted by $V(G)$ and $E(G)$. A multigraph G is a pair (V, E), where V is a set of vertices, and E is a multiset of edges. A graph is either a simple graph or a multigraph. The cardinality of $V(G)$ is usually denoted by n, the cardinality of $E(G)$ by m.*

In this chapter, the term *graph* refers to both simple graphs and multigraphs. In the remaining chapters of the thesis, unless stated otherwise, we use the term graph for simple connected undirected graph. In some cases, we use *directed graphs* (either simple of multigraphs): in a directed graph, each edge is an ordered pair of vertices, and an edge from vertex u to v is denoted (u, v).

Let $G = (V, E)$ be a graph. For any edge $e = \{u, v\} \in E$, u and v are called the *end points* of e, and e is called an edge between u and v, or connecting u and v. Two vertices $u, v \in E$ are *adjacent* (or *neighbors*) if there is an edge $\{u, v\} \in E$. If two vertices u and v are adjacent, we also say that u is a *neighbor* of v, and vice versa. A vertex $v \in V$ and an edge $e \in E$ are called incident if $e = \{u, v\}$ for some $u \in V$. The *degree* of a vertex v in G is the number of edges that are incident with v, and is denoted by $deg(v)$.

Definition 1.2 *A graph G' is a subgraph of a graph G if $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$. If G' is a subgraph of G, then G is called s supergraph of G'. A graph G' is the subgraph of G induced by W, where $W \subseteq V(G)$, if $V(G') = W$ and $E(G') = \{\{u, v\} \in E \mid u, v \in W\}$. We also say G' is an induced subgraph of G. For any $W \subseteq V(G)$, the subgraph induced by W is denoted by $G[W]$. A graph $G_1 = (V_1, E_1)$ is a spanning subgraph of $G = (V, E)$ if V_1 spans all vertices of G, i.e., $V_1 = V$.*

A *walk* W in a graph G is an alternating sequence $(v_1, e_1, v_2, e_2, ..., e_p, v_{p+1})$ of vertices and edges ($p \geq 0$), starting and ending with a vertex, such that for each i, $v_i \in V$, and $e_i \in E$, and $e_i = \{v_i, v_{i+1}\}$. The walk W is also called a walk from v_1 to v_{p+1}. Vertices v_1 and v_{p+1} are called the end points of the walk, all other vertices are inner vertices. We also called v_1 the first vertex and v_{p+1} the last vertex of the walk. The *length* of a walk is the number of edges in the walk. We say a walk W *goes through* a vertex v if $v = v_i$ for some i with $1 \leq i \leq p+1$, and W *avoids* v if W does not go through v.

Definition 1.3 *A path in a graph G is a walk in which all vertices are distinct (and hence all edges are distinct). A cycle C in G is a walk in which all edges are distinct, and all vertices are distinct, except for the first and the last vertex, which are equal.*

A walk, path or cycle H in a graph can also be seen as a subgraph of G, and we denote the set of vertices in H by $V(H)$, and the set of edges by $E(H)$.

Definition 1.4 *The distance between two vertices v and w in G is the length of a shortest path from v to w in G. The diameter of a graph G, denoted $\Delta(G)$, is the greatest distance between any two vertices of G.*

Two vertices are *connected* in a graph G if there is a path between them. A graph G is connected if every pair of vertices of G is connected. A (connected) component C of G is a maximal connected subgraph of G, i.e., C is a subgraph of G which is connected, and there is no subgraph of G which properly contains C and is also connected. An edge $e = \{u, v\}$ is said to be *pendant* if $deg(u) = 1$.

Definition 1.5 *A tree is a simple connected graph without cycles. A forest is a simple graph without cycles, i.e., a graph is a forest if and only if each of its connected components is a tree. Note that in a tree, there is a unique path between each pair of vertices. A spanning tree of a graph G is a tree that spans all the vertices of G. A spanning forest of a graph G is a forest that spans all the vertices of G.*

A *rooted tree* is a tree T with a distinguished vertex $r \in V(T)$ called the *root* of T. In a rooted tree T, the *descendants* of a vertex $v \in V(T)$ are the vertices of which the path to the root goes through v. The *children* of v are the descendants of v which have distance one to v. If v is not the root, then the parent of v is the unique vertex of which v is a child. The *leaves* of a rooted tree are the vertices without children. The vertices which are not leaves are called *internal vertices*. The *depth* of a rooted tree T is the maximum distance of any vertex in T to the root. The *level* of a vertex v in a rooted tree T equals the depth of T minus the distance of v to the root. Hence the root has level d, where d is the depth, and the vertices on level zero are leaves which have distance d to the root.

Definition 1.6 *A complete graph or clique is a simple graph in which every two vertices are adjacent. The complete graph on k vertices, denoted by K_k, is also called a k-clique. A clique in a graph G is a subgraph of G which is a clique. The maximum clique size of a graph G is the maximum number of vertices of any clique in G.*

Definition 1.7 *The neighborhood of a vertex v in $G = (V, E)$ is the set of all vertices which are neighbors of u. This set is denoted by $N_G(u) = \{v \in V \mid \{u, v\} \in E\}$. Obviously, $|N_G(u)| = deg(u)$. $I_G(u)$ will stand for the set of all the edges incident to the vertex u.*

Definition 1.8 *For two vertices $u, v \in V(G)$, $d(u, v)$ represents the distance between u and v in G. Moreover, we denote by $B_G(v, k)$ the ball of radius k with center v, that is the graph with vertices set $V_v = \{v_0 \in V | d(v, v_0) \leq k\}$ and edges set $\{e = (u_0, v_0) | u_0, v_0 \in V_v\}$. If $k = 1$ we usually use $B_G(v)$ instead of $B_G(v, 1)$.*

Definition 1.9 *Two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are said to be isomorphic if there are bijections $f : V_1 \to V_2$ and $g : E_1 \to E_2$ such that for each $v \in V_1$ and $e \in E_1$, v is incident with e in G_1 if and only if $f(v)$ is incident with $g(e)$ in G_2. The pair (f, g) is called an isomorphism from G_1 to G_2.*

If G_1 and G_2 are simple graphs, then it suffices to give the bijection between the vertices of G_1 and G_2, i.e., G_1 and G_2 are isomorphic if there is a bijection $f : V_1 \to V_2$ such that for each $u, v \in V_1, \{u, v\} \in E_1$ if and only if $\{f(u), f(v)\} \in E_2$. For simple graphs, we also say that f is an isomorphism from G_1 to G_2.

Definition 1.10 *A homomorphism between graphs $G = (V(G), E(G))$ and $H = (V(H), E(H))$ is a mapping $\gamma \colon V(G) \to V(H)$ such that if $\{u, v\} \in E(G)$ then $\{\gamma(u), \gamma(v)\} \in E(H)$. If γ is bijective and γ^{-1} is a homomorphism, then γ is an isomorphism.*

1.1.2 Graphs and Logics

A graph can also be considered as a logical structure of a certain type. Hence, properties of graphs can be written in first-order logic or in second-order logic. It turns out that *monadic second-order logic*, where quantifications over sets of vertices and sets of edges are used, is a reasonably powerful logical language in which many usual graph properties can be written and for which one can obtain decidability results [Cou90b]. These decidability results do not hold for second-order logic, where quantifications over binary relations can also be used. The goal of this section is to give present a simple presentation of the monadic second-order logic of graphs (MSOL).

Monadic Second-order Logic. MSOL for graphs $G = (V, E)$ consists of a language in which predicates can be built with

- the logic connectives $\wedge$, $\vee$, $\neg$, $\Rightarrow$, $\Leftrightarrow$ (with their usual meanings),
- individual variables which may be vertex variables (with domain V), edge variables (with domain E), vertex set variables (with domain $\mathcal{P}(V)$, the power set of V), and edge set variables (with domain $\mathcal{P}(E)$), the existence and universal quantifiers ranging over variables ($\exists$ and $\forall$, respectively), and
- the following binary relations:
  - $v \in W$, where v is a vertex variable and W a vertex set variable,
  - $e \in F$, where e is an edge variable and F an edge set variable,
  - "v and w are adjacent in G", where v and w are vertex variables,
  - "v is incident with e in G", where v is a vertex variable, and e an edge variable,
  - equality for variables.

Definition 1.11 (MS-definable) *Graph properties that can be defined by an MSOL predicate are called MS-definable graph properties.*

1.1.3 Contractions and Minors

Let $e = (x, y)$ be an edge of a graph $G = (V, E)$. By G/e we denote the graph obtained from G by *contracting* the edge e into a new vertex v_e, which becomes adjacent to all the former neighbors of x and of y. Formally, G/e is a graph (V', E') with vertex set $V' := (V - \{x, y\} \cup \{v_e\}$ and edge set $E' := \{\{v, w\} \in E | \{v, w\} \cap \{x, y\} = \emptyset\} \cup \{\{v_e, w\} | \{x, w\} \in E - e \vee \{y, w\} \in E - e\}$ (see Figure 1). More generally, if X is another graph and $\{V_x | x \in V(X)\}$

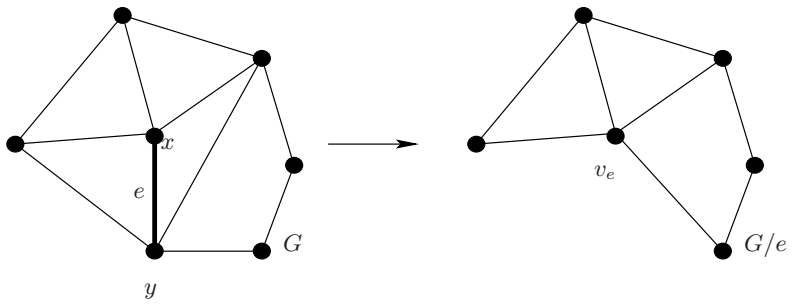


Figure 1: Contracting the edge $e = \{x, y\}$

is a partition of V into connected subsets such that, for any two vertices $x, y \in X$, there is a $V_x - V_y$ edge in G if and only if $\{x, y\} \in E(X)$, we call G an MX and write $G = \text{MX}$. The sets V_x are the *branch sets* of this MX. Intuitively, we obtain X from G by contracting every branch set to a single vertex and deleting any parallel edges or loops that may arise. In infinite graphs, branch sets are allowed to be infinite.

If $V_x = U \subseteq V$ is one of the branch sets above and every other branch set consists just of a single vertex, we also write G/U for the graph X and v_U for the vertex $x \in X$ to which U

contracts, and think of the rest of X as an induced subgraph of G. The contraction of a single edge $\{u, u'\}$ defined earlier can then be viewed as the special case of $U := \{u, u'\}$. Figure 2 shows the branch sets contraction of a graph G.

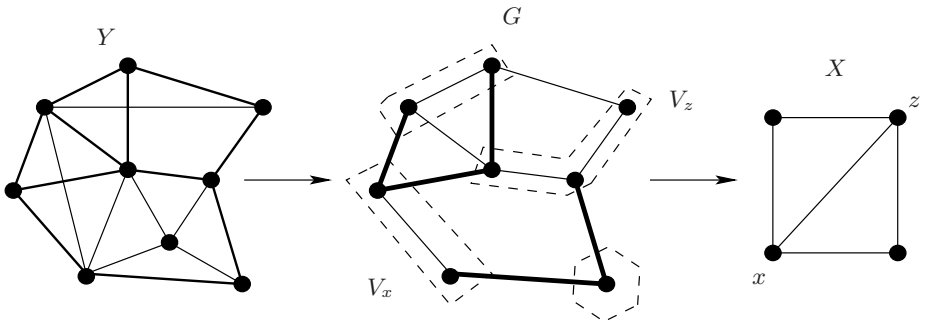


Figure 2: $Y \supseteq G = \text{MX}$, so X is a minor of Y

Proposition 1.1 *G is an MX if and only if X can be obtained from G by a series of edge contractions, i.e., if and only if there are graphs $G_0, ..., G_n$ and edges $e_i \in G_i$ such that $G_0 = G$, $G_n = X$, and $G_i + 1 = G_i/e_i$ for all $i < n$.*

Proof. The proof of this proposition can be done by a simpler induction on $k = |G| - |X|$. □

Definition 1.12 *If $G = MX$ is a subgraph of another graph Y, we call X a minor of Y and write $X \preccurlyeq Y$.*

Remark 1.1 *Every subgraph of a graph is also its minor; in particular, every graph is its own minor. By Proposition 1.1, any minor of a graph can be obtained from it by first deleting some vertices and edges, and then contracting some further edges. Conversely, any graph obtained from another by repeated deletions and contractions (in any order), is its minor: this is clear for one deletion or contraction, and follows for several from the transitivity of the minor relation (Proposition 1.1).*

1.1.4 Treewidth and Pathwidth

In this section, we give some background information on the treewidth of a graph. The notion of treewidth was introduced by Robertson and Seymour [RS83, RS86].

Definition 1.13 (Tree decomposition and Treewidth) . *Let $G = (V, E)$ be a graph. A tree decomposition TD of G is a pair $(T, \mathcal{X})$, where $T = (I, F)$ is a tree, and $\mathcal{X} = \{X_i | i \in I\}$ is a family of subsets of V, one for each vertex of T, such that*

- $\bigcup_{i \in I} X_i = V$,
- *for each edge $\{v, w\} \in E$, there is an $i \in I$ with $v \in X_i$ and $w \in X_i$, and*
- *for all $i, j, k \in I$, if j is on the path from i to k in T, then $X_i \cap X_k \subseteq X_j$.*

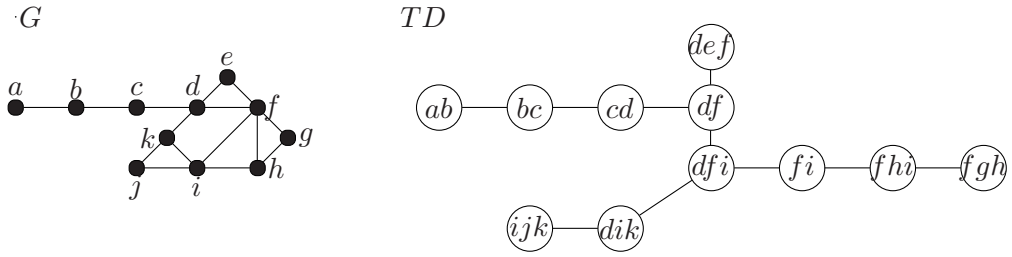


Figure 3: A graph G of treewidth two, and a tree decomposition TD of width two of G.

The treewidth or width of a tree decomposition $((I, F), \{X_i | i \in I\})$ *is* $max_{i \in I} |X_i| - 1$. *The treewidth of a graph* G*, denoted* $tw(G)$*, is the minimum width over all possible tree decompositions of* G.

The vertices of a tree in a tree decomposition are usually called *nodes* to avoid confusion with the vertices of a graph. If a vertex v or the end points of an edge e are contained in X_i for some node i of a tree decomposition, we also say node i *contains* v or e. An example of a graph of treewidth two and a tree decomposition of width two of the graph is given in Figure 3. A tree decomposition is usually depicted as a tree in which each node i contains the vertices of X_i.

Definition 1.14 (Path Decomposition and Pathwidth) . *A path decomposition PD of a graph* G *is a tree decomposition* $(T, \mathcal{X})$ *of* G *in which the tree* T *is a path (i.e. the nodes of* T *have degree at most two). The pathwidth of a graph* G *is the minimum width over all possible path decompositions of the graph, and is denoted by* $pw(G)$.

Let k be a positive integer. Graphs of treewidth at most k are also called *partial* k*-trees* (as they are exactly the subgraphs of k-trees, see e.g. Kloks [Klo94] for definitions and proofs). In the literature, many other notions have been defined which turned out to be equivalent to the notions of treewidth or pathwidth. Bodlaender [BdF96] gave a list of these notions. There are also many classes of graphs which have a constant bound on the treewidth or pathwidth, or which are closely related to classes of graphs of bounded treewidth or bounded pathwidth. For example, the forest are exactly the simple graphs of treewidth at most one. Series-parallel graphs have a treewidth at most two, k-outerplanar graphs have treewidth at most $3k - 1$ [BdF96].

Now we give a number of well-known properties of tree and path decompositions and of graphs of bounded treewidth or pathwidth. Most of these properties have already been noted by many authors (see e.g. Robertson and Seymour [RS83, RS86], Scheffler [Sch89] and Bodlaender [BdF96]). We present some results in the area of tree and path decompositions that are of interest.

Lemma 1.1 (Scheffler [Sch89], Bodlaender [BdF96])
Let G *be a graph.*

1. *The treewidth (or pathwidth) of any subgraph of G is at most the treewidth (or pathwidth) of G.*
2. *The treewidth of G is the maximum treewidth over all components of G.*
3. *The pathwidth of G is the maximum pathwidth over all components of G.*
4. *The treewidth of G is the maximum treewidth over all blocks of G.*

Lemma 1.2 ([vAdF97]) *Let G be a graph and $TD = (T, \mathcal{X})$ a tree decomposition of G.*

1. *Let $u, v \in V(G)$, and let $i, j \in I$ be such that $u \in X_i$ and $v \in X_j$. Then each node on the path from i to j in T contains a vertex of every path from u to v in G.*
2. *For each connected subgraph G' of G, the nodes in T which contain a vertex of G' induce a subtree of T.*

A *rooted binary* tree decomposition of a graph G is a tree decomposition $(T, \mathcal{X})$ of G in which T is a rooted binary tree.

Lemma 1.3 ([vAdF97]) *Let G be a graph. There is a rooted binary tree decomposition of minimum width of G with $O(n)$ nodes.*

Lemma 1.4 ([vAdF97]) *Let $G = (V, E)$ be a simple graph, let $k \geq 1$ a natural number, and suppose $tw(G) = k$. Then $|E| \leq k|V| - \frac{1}{2}(k+1)$.*

Lemma 1.5 *Let G be a graph and let H be a minor of G. Then $tw(H) \leq tw(G)$ and $pw(H) \leq pw(G)$.*

1.2 Randomized Algorithms and Random Walks

1.2.1 Randomized Algorithms

A *randomized algorithm* or *probabilistic algorithm* is an algorithm which is allowed to flip a truly random coin. In common practice, this means that the machine implementing the algorithm has access to a pseudo-random number generator. The algorithm typically uses the random bits as an auxiliary input to guide its behavior, in the hope of achieving good performance in the "average case". Formally, the algorithm's performance will be a random variable determined by the random bits, with (hopefully) good expected value; this expected value is called the expected runtime. The "worst case" is typically so unlikely to occur that it can be ignored.

Definition 1.15 (Las Vegas Algorithm) *A Las Vegas algorithm is a randomized algorithm which is correct; that is, it always produces the correct result. Thus, the used randomization only influence the resources used by the algorithm. A simple example is randomized quicksort, where the pivot is chosen randomly, but the result is always a sorted sequence.*

An alternative definition of a Las Vegas algorithm includes the restriction that the average-case running-time must be finite.

1.2.2 Random Walks

Let G be a connected graph on n vertices, and let v be a fixed vertex of G. A *random walk* on G, beginning at v, is a stochastic process whose state at any time t is given by a vertex of G; at time 0 it is at vertex v, and if at time t it is at vertex u, then at time $t+1$ it will be at one of the neighbors of u, each neighbor having been chosen with equal probability. The random walk thus constitutes a Markov chain, with state transition probability $p_{u,v} = 0$ if v is not adjacent to u and $p_{u,v} = \frac{1}{d(u)}$ if v is adjacent to u and u has degree $d(u)$.

When dealing with randomness some probability space $(\Omega, \mathcal{A}, \mathcal{P})$ is usually involved. In our case, we define Ω to be the *sample space*, $\mathcal{A}$ the set of all possible events and $\mathcal{P}$ is some *probability measure* on Ω. In this work we assume further, that the state space is always finite, $\mathcal{S} = V$. The function $\mathcal{P} : \mathcal{S} \times \mathcal{S} \to \mathbb{R}$ with

$$\mathcal{P}(u, v) = \mathcal{P}[X_1 = v | X_0 = u]$$

is called the transition function; its values $\mathcal{P}(u, v)$ are called the (conditional) transition probabilities from u to v.

Definition 1.16 *A probability distribution π satisfying*

$$\begin{aligned} \pi &= \pi\mathcal{P} \quad \textit{this means that,} \\ \pi(v) &= \sum_{u \in \mathcal{S}} \pi(u)\mathcal{P}(u, v) \quad \textit{for all } v \in \mathcal{S}. \end{aligned}$$

is called a **stationary distribution** *of the Markov chain $\{X_k\}_{k\in\mathbb{N}}$. With $\mathbb{P}$ representing the probability transition matrix whose entries are all non negative and whose rows sum to 1. We denote by $\mathcal{P}_\pi$ the transition probability with respect to the distribution π.*

Definition 1.17 *[Cha99] We say that a Markov chain $\{X_k\}_{k\in\mathbb{N}}$ is* **reversible** *with respect to the probability distribution π, if*

$$\mathcal{P}_\pi[X_0 = u_0, ..., X_m = u_m] = \mathcal{P}_\pi[X_0 = u_m, ..., X_m = u_0]$$

for every $m \in \mathbb{N}$ and every $u_0, ..., u_m \in \mathcal{S}$.

J. Chang [Cha99] gave a simple condition for reversibility that we now state as a proposition. For a complete proof of this proposition the manuscript of Wilhelm Huisinga [Hui03] can be recommended.

Proposition 1.2 *Let $\{X_k\}_{k\in\mathbb{N}}$ denote some Markov chain with transition matrix $\mathbb{P}$, and let π denote some probability distribution. Then the Markov chain is reversible with respect to π, if and only if the relation*

$$\pi(u)\mathbb{P}(u, v) = \pi(v)\mathbb{P}(v, u)$$

holds for every $u, v \in \mathcal{S}$. In either case, π is a stationary distribution of $\{X_k\}_{k\in\mathbb{N}}$.

Definition 1.18 *Let u and v be two states. We say that v* **is accessible from** *u if it is possible (with positive probability) for the chain ever to visit state v if the chain starts in state u, or in other words,*

$$\mathbb{P}\{\bigcup_{n=0}^{\infty} \{X_n = v\} | X_0 = u\} > 0.$$

- *We say u* **communicates with** *v if v is accessible from u and u is accessible from v.*
- *We say that the Markov chain is* **irreducible** *if all pairs of states communicate.*

Theorem 1.1 *[Cha99] Every irreducible Markov chain with finite state space admits a unique stationary distribution that is positive everywhere.*

Definition 1.19 *The period $D(u)$ of some state $u \in \mathcal{S}$ is, by definition,*

$$D(u) = gcd\{k \geq 1 : \mathbb{P}[X_k = u | X_0 = u] = \mathbb{P}^k(u,u) > 0\},$$

with the convention $D(u) = \infty$, if there is no $k \geq 1$ with $\mathbb{P}^k > 0$. If $D(u) = 1$, then the state u is called aperiodic.

Definition 1.20 *[Hui03] An irreducible Markov chain $\{X_k\}_{k\in\mathbb{N}}$ is* **aperiodic**, *if there exists at least one state $u \in \mathcal{S}$ such that $\mathbb{P}(u,u) > 0$.*

Theorem 1.2 (Basic Limit Theorem) *[Sen80, Cha99, Hui03] Let $\{X_k\}_{k\in\mathbb{N}}$ be an irreducible, aperiodic Markov chain having a stationary distribution π. Then for all initial distributions π_0,*

$$\lim_{n\to\infty} \mathbb{P}_{\pi_0}\{X_n = i\} = \pi(i) \text{ for all } i \in \mathcal{S}.$$

Theorem 1.2 states that for large n, the Markov chain X_n at time n is approximately distributed like π, and moreover it is approximately independent of its history, in particular of X_{n-1} and X_0. Thus the distribution of X_n for $n >> 0$ is almost the same, namely π, regardless of whether the Markov chain started at $X_0 = u$ or $X_0 = v$ for some initial states $u, v \in \mathcal{S}$.

1.3 Labeled Graphs and Coverings

1.3.1 Labeled Graphs

In this thesis we will consider graphs whose vertices are labeled with labels from a recursive set L. A graph labeled over L will be denoted by (G, λ), where $G = (V(G), E(G))$ is a graph and $\lambda\colon V(G) \to L$ is the vertex labeling function. The graph G is called the *underlying graph* and the mapping λ is a labeling of G. Labeled graphs will be designated by bold letters like $\mathbf{G}$, $\mathbf{H}, \ldots$ If $\mathbf{G}$ is a labeled graph, then G denotes the underlying graph.

The class of labeled graphs over some fixed alphabet L will be denoted by $\mathcal{G}_L$ ($\mathcal{G}$ if there is no ambiguity). Let (G, λ) and (G', λ') be two labeled graphs. Then (G, λ) is a subgraph of (G', λ'), denoted by $(G, \lambda) \subseteq (G', \lambda')$, if G is a subgraph of G' and λ is the restriction of the labeling λ' to $V(G) \cup E(G)$.

Definition 1.21 *A mapping $\gamma\colon V(G) \to V(G')$ is a homomorphism from (G, λ) to (G', λ') if γ is a graph homomorphism from G to G' which preserves the labeling, i.e., such that $\lambda'(\gamma(v)) = \lambda(v)$ holds for every $v \in V(G)$.*

1.3.2 Coverings

The notion of *Covering* is known from algebraic topology [Mas91]. Coverings have been used for simulation [BL86] and for proving impossibility results on distributed computing [Ang80, FLM86].

Definition 1.22 (Covering) *We say that a graph G is a* covering *of a graph H via γ if γ is a surjective homomorphism from G onto H such that for every vertex v of $V(G)$ the restriction of γ to $B_G(v)$ is a bijection onto $B_H(\gamma(v))$.*

Naturally, coverings of labeled graphs are just coverings of underlying graphs preserving the labeling.

Definition 1.23 (Proper Covering) *A graph G is a* proper covering *of a graph H via γ if G is a* covering *of H via γ and G and H are not isomorphic.*

Definition 1.24 (Covering-Minimal) *A graph G is called* covering-minimal *if there is no graph H such that G is a proper covering of H via a morphism γ. That is, every covering from G to some H is a bijection.*

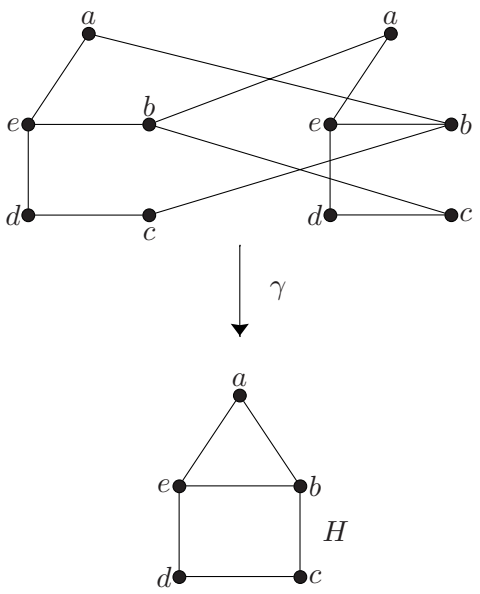


Figure 4: The morphism γ is a covering from G to H.

Lemma 1.6 ([Rei32]) *Suppose that G is a covering of H via γ. Let T be a subgraph of H. If T is a tree then $\gamma^{-1}(T)$ is a set of disjoint trees, each isomorphic to T.*

By considering simple paths between two vertices, the previous lemma implies:

Lemma 1.7 *Suppose that* **G** *is a covering of* **H** *via a morphism γ. There exists an integer q such that for every vertex $v \in V(H)$, $|\gamma^{-1}(v)| = q$.*

The integer q in the previous lemma is called the number of *sheets* of the covering. We also refer to γ as a *q-sheeted covering*. If $q = 1$, then G and H are isomorphic.

Example 1.1 *A simple example of 2-sheeted covering is given in Figure 4. The image of each vertex of G is given by the corresponding Roman letter. Furthermore, we note that the image of each vertex is also given by its position on the H pattern.*

Figure 5 depicts the results of the following lemma.

Lemma 1.8 *Let G be a covering of H via γ and let $v_1, v_2 \in V(G)$ be such that $v_1 \neq v_2$ and $\gamma(v_1) = \gamma(v_2)$. Then we have $B_G(v_1) \cap B_G(v_2) = \emptyset$.*

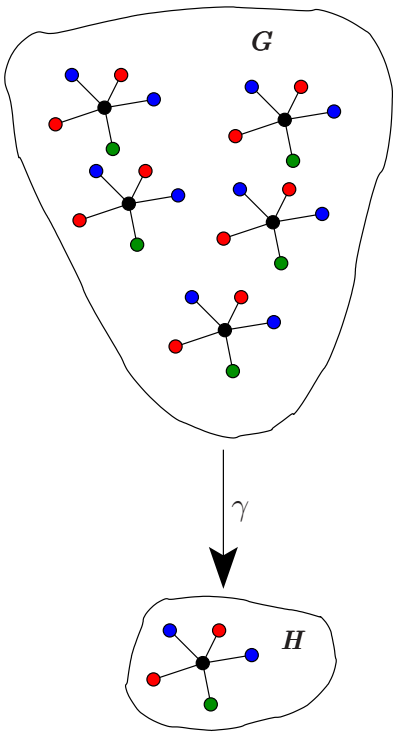


Figure 5: The preimages of a ball B_H of H build a collection of disjoint balls of G

1.3.3 Quasi-coverings

Quasi-coverings have been introduced in [MMW97] to obtain impossibility proofs for local detection of global termination. These are applications that, partially, behave like coverings. The idea behind this notion is to enable the simulation of local computations on a given graph in a restricted area of a larger graph, such that the simulation can lead to false conclusion. The restricted area where we can perform the simulation will shrink while the number of simulated steps increases.

Definition 1.25 *Let K, H be two graphs and let δ be a partial function on $V(K)$ that assigns to each element of a subset of $V(K)$ exactly one element of $V(H)$. Then K is a quasi-covering of H of size r if there exists a finite or infinite covering G of H via γ, vertices*

$z_K \in V(K)$, $z_G \in V(G)$, and an integer $r > 0$ such that:

1. *$B_{\mathbf{K}}(z_K, r)$ is isomorphic via φ to $B_{\mathbf{G}}(z_G, r)$,*
2. *$|V(B_{\mathbf{K}}(z_K, r))| \geq s$,*
3. *the domain of definition of δ contains $B_{\mathbf{K}}(z_K, r))$, and*
4. *$\delta = \gamma \circ \varphi$ when restricted to $V(B_{\mathbf{K}}(z_K, r))$.*

The basic ideas of Definition 1.25 are represented in Figure 6.

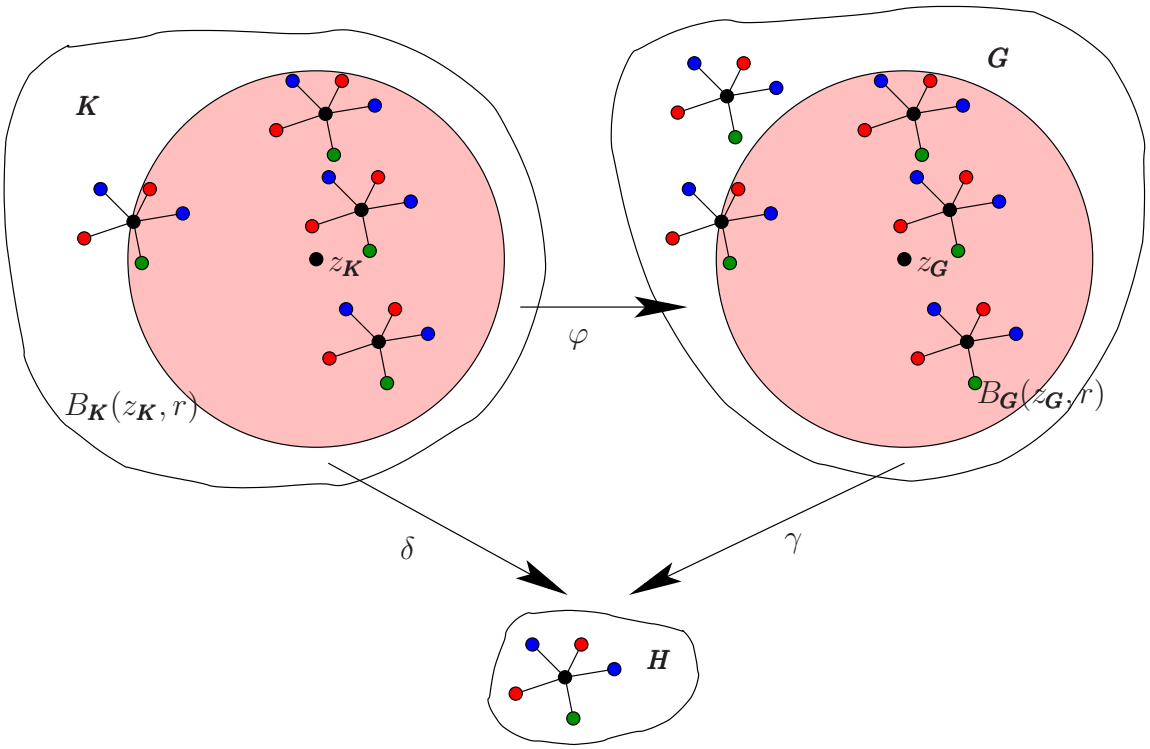


Figure 6: $\delta : V(K) \to V(H)$ is a quasi-covering of radius r. $\gamma : G \to H$ represents the associated covering.

Chapter 2

Local Computations and Graph Relabeling Systems

Contents

In this chapter we give formal definitions of the concepts of graph relabeling systems and local computations. For a better understanding of these concepts we refer to the works of Bauderon et al. [BMMS02] and Litovsky et al. [LMS99].

2.1 The Computation Model

Graph relabeling systems and, more generally, local computations in graphs are models which provide general tools for encoding distributed algorithms, for proving their correctness and for understanding their power [LMS99]. Standard considerations on distributed algorithms are presented in [AW98, Bar96, Lav95, Lyn96, Tel00].

We consider a network of processors with arbitrary topology. It is represented as a connected, undirected graph where vertices denote processors, and edges denote direct communication links. An algorithm is encoded by means of local relabellings. Labels attached to

vertices and edges are modified *locally*, that is on a subgraph of fixed radius k ($k \in \mathbb{N}$) of the given graph, according to certain rules depending on the subgraph only (*local computations*). The relabeling is performed until no more transformation is possible. The corresponding configuration is said to be in normal form. Two sequential relabeling steps are said to be *independent* if they are applied on disjoint subgraphs. In this case they may be applied in any order or even concurrently.

Related Models. Among models related to the model described above there are local computation systems as defined by Rosenstiehl et al. [RFH72], Angluin [Ang80], Yamashita and Kameda [KY96] and Boldi and Vigna [BV99]. In [RFH72] a synchronous model is considered, where vertices represent (identical) deterministic finite automata. The basic computation step is to compute the next state of each processor according to its state and the states of its neighbors. In [Ang80] an asynchronous model is considered within which a basic computation step stands for the fact that two adjacent vertices exchange their labels and then compute new ones. In [KY96] another asynchronous model is studied where a basic computation step means that a processor either changes its state and sends a message or it receives a message. In [BV99], networks are directed graphs colored on their arcs; each processor changes its state depending on its previous state and on the states of its in-neighbors. Activation of processors may be synchronous, asynchronous or interleaved.

2.2 Graph Relabeling Systems

Definition 2.1 *A* (graph) relabeling rule *is a triple* $R = (G_R, \lambda_R, \lambda'_R)$ *such that* (G_R, λ_R) *and* (G_R, λ'_R) *are two labeled graphs. The labeled graph* (G_R, λ_R) *is the* left-hand side *and the labeled graph* (G_R, λ'_R) *is the* right-hand side *of* R.

Definition 2.2 *A* graph relabeling system *(GRS for short) is a triple* $\mathcal{R} = (L, I, P)$ *where* L *is a set of labels,* I *a subset of* L *called the set of* initial labels *and* P *a finite set of relabeling rules.*

The intuitive notion of computation step will then correspond to the notion of relabeling step:

Definition 2.3 *An* $\mathcal{R}-$relabeling step *is a 5-tuple* $(G, \lambda, R, \varphi, \lambda')$ *such that* R *is a relabeling rule in* P *and* φ *is both an occurrence of* (G_R, λ_R) *in* (G, λ) *and an occurrence of* (G_R, λ'_R) *in* (G, λ').

Intuitively speaking, the labeling λ' of G is obtained from λ by modifying all the labels of the elements of $\varphi(G_R, \lambda_R)$ according to the labeling λ'_R. Such a relabeling step will be denoted by $(G, \lambda) \longrightarrow_{R,\varphi} (G, \lambda')$.

The notion of computation then corresponds to the notion of relabeling sequence:

Definition 2.4 *A* $\mathcal{R}-$relabeling sequence *is a tuple* $(G, \lambda_0, R_0, \varphi_0, \lambda_1, R_1, \varphi_1, \lambda_2, \ldots, \lambda_{n-1}, R_{n-1}, \varphi_{n-1}, \lambda_n)$ *such that for every* i, $0 \leq i < n$, $(G, \lambda_i, R_i, \varphi_i, \lambda_{i+1})$ *is a* $\mathcal{R}-$*relabeling step. The existence of such a relabeling sequence will be denoted by* $(G, \lambda_0) \longrightarrow^*_{\mathcal{R}} (G, \lambda_n)$.

The computation stops when the graph is labeled in such a way that no relabeling rule can be applied:

Definition 2.5 *A labeled graph (G, λ) is said to be* $\mathcal{R}-$irreducible *if there exists no occurrence of (G_R, λ_R) in (G, λ) for every relabeling rule R in P.*

For every labeled graph $\mathbf{G}$ in $\mathcal{G}_{\mathcal{I}}$ we denote by $Irred_{\mathcal{R}}(\mathbf{G})$ the set of all $\mathcal{R}-$irreducible labeled graphs $\mathbf{G}'$ such that $\mathbf{G} \longrightarrow_{\mathcal{R}}^{*} \mathbf{G}'$. Intuitively speaking, the set $Irred_{\mathcal{R}}(\mathbf{G})$ contains all the final labellings that can be obtained from an $I-$labeled graph $\mathbf{G}$ by applying relabeling rules in P and may be viewed as the set of all the possible results of the computation encoded by the system $\mathcal{R}$.

2.2.1 Graph Relabeling Systems with Priorities

The first mechanism we will consider is obtained by introducing some priority relation on the set of relabeling rules. It is obtained by introducing some priority relation, denoted $>$, on the set of relabeling rules. A relabeling rule R is applied if there exists no occurrence of a relabeling rule R' with $R' > R$ which intersects R (i.e. R and R' are applied on subgraphs having vertices in common).

Definition 2.6 *A graph relabeling system* with priorities *(PGRS for short) is a 4-tuple $\mathcal{R} = (L, I, P, >)$ such that (L, I, P) is a graph relabeling system and $>$ is a partial order defined on the set P, called the* priority relation.

An $\mathcal{R}-$relabeling step *is then defined as a 5-tuple $(G, \lambda, R, \varphi, \lambda')$ such that R is a relabeling rule in P, φ is both an occurrence of (G_R, λ_R) in (G, λ) and an occurrence of (G_R, λ'_R) in (G, λ') and there exists no occurrence φ' of a relabeling rule R' in P with $R' > R$ such that $\varphi(G_R)$ and $\varphi(G_{R'})$ intersect in G (that is $V(\varphi(G_R)) \cap V(\varphi(G_{R'})) = \emptyset$).*

The notion of relabeling sequence is defined as previously.

2.2.2 Graph Relabeling Systems with Forbidden Contexts

The idea developed here is to prevent the application of a relabeling rule whenever the corresponding occurrence is "included" in some special configuration, called a context. More formally, we have:

Definition 2.7 *Let (G, λ) be a labeled graph. A* context *of (G, λ) is a triple (H, μ, ψ) such that (H, μ) is a labeled graph and ψ an occurrence of (G, λ) in (H, μ).*

Definition 2.8 *A* relabeling rule with forbidden contexts *is a 4-tuple $R = (G_R, \lambda_R, \lambda'_R, F_R)$ such that $(G_R, \lambda_R, \lambda'_R)$ is a relabeling rule and F_R is a finite set of contexts of (G_R, λ_R).*

Definition 2.9 *A* graph relabeling system with forbidden contexts *(FCGRS for short) is a triple $\mathcal{R} = (L, I, P)$ defined as a GRS except that the set P is a set of relabeling rules with forbidden contexts.*

A relabeling rule with forbidden contexts may be applied on some occurrence if and only if this occurrence is not "included" in an occurrence of some of its forbidden contexts. More formally:

Definition 2.10 *An $\mathcal{R}-$*relabeling step *is a 5-tuple $(G, \lambda, R, \varphi, \lambda')$ such that R is a relabeling rule with forbidden contexts in P, φ is both an occurrence of (G_R, λ_R) in (G, λ) and an occurrence of (G_R, λ'_R) in (G, λ'), and for every context (H_i, μ_i, ψ_i) of (G_R, λ_R), there is no occurrence φ_i of (H_i, μ_i) in (G, λ) such that $\varphi_i(\psi_i(G_R, \lambda_R)) = \varphi(G_R, \lambda_R)$.*

The comparison between the expressive power of PGRSs and FCGRSs, together with some other types of GRSs, has been done in [LMS95]. In particular, it has been proved that PGRSs and FCGRSs are equivalent: for every PGRS (resp. FCGRS) there exists a FCGRS (resp. PGRS) achieving the same computation.

2.3 Local Computations

Graph relabeling systems, as introduced in the previous section, are in fact an illustration of a more general mechanism called local computations. Local computations as considered here can be described in the following general framework. For simplicity we consider local computations on labeled balls of radius 1, although the definitions are similar for balls of radius k for a constant k.

Labels are modified locally, that is, on a subgraph of fixed radius k of the given graph, according to certain rules depending on the subgraph only. The relabeling is performed until no more transformation is possible, i.e., until a normal form is obtained.

In this part we give formal definitions of local computations. Intuitively, local computations are characterized by applications of rules such that: an application of a rule to a ball depends exclusively on the labels appearing in the ball and changes only these labels.

Definition 2.11 *A graph rewriting relation is a binary relation $\mathcal{R} \subseteq \mathcal{G}_L \times \mathcal{G}_L$ closed under isomorphism. The transitive closure of $\mathcal{R}$ is denoted $\mathcal{R}^*$.*
An $\mathcal{R}-$rewriting chain is a sequence $\mathbf{G}_1, \mathbf{G}_2, \ldots, \mathbf{G}_n$ such that for every i, $1 \leq i < n$, $\mathbf{G}_i \ \mathcal{R} \ \mathbf{G}_{i+1}$. A sequence of length 1 is called a $\mathcal{R}-$rewriting step (a step for short).

By "closed under isomorphism" we mean that if $\mathbf{G}_1 \simeq \mathbf{G}$ and $\mathbf{G} \ \mathcal{R} \ \mathbf{G}'$, then there exists a labeled graph $\mathbf{G}'_1$ such that $\mathbf{G}_1 \ \mathcal{R} \ \mathbf{G}'_1$ and $\mathbf{G}'_1 \simeq \mathbf{G}'$.

Definition 2.12 *Let $\mathcal{R} \subseteq \mathcal{G}_L \times \mathcal{G}_L$ be a graph rewriting relation.*

1. *$\mathcal{R}$ is a relabeling relation if whenever two labeled graphs are in relation then their underlying graphs are equal (not only isomorphic):*

$$\mathbf{G} \ \mathcal{R} \ \mathbf{H} \quad \Longrightarrow \quad G = H.$$

When $\mathcal{R}$ is a relabeling relation we will speak about $\mathcal{R}-$relabeling chains (resp. step) instead of $\mathcal{R}-$rewriting chains (resp. step).

2. *A relabeling relation $\mathcal{R}$ is local if whenever $(G,\lambda)\ \mathcal{R}\ (G,\lambda')$, the labellings λ and λ' only differ on some ball of radius* 1 :

$$\exists\ v \in V(G) \text{ such that } \forall\ x \notin V(B_G(v,1)) \cup E(B_G(v,1)),\ \lambda(x) = \lambda'(x).$$

We say that the step changes labels in $B_G(v,1)$.

3. *An $\mathcal{R}-$normal form of $\mathbf{G} \in \mathcal{G}_L$ is a labeled graph $\mathbf{G}'$ such that $\mathbf{G}\ \mathcal{R}^*\ \mathbf{G}'$, and $\mathbf{G}'\ \mathcal{R}\ \mathbf{G}''$ holds for no $\mathbf{G}''$ in $\mathcal{G}_L$. We say that $\mathcal{R}$ is noetherian if for every graph $\mathbf{G}$ in $\mathcal{G}_L$ there exists no infinite $\mathcal{R}-$relabeling chain starting from $\mathbf{G}$. Thus, if a relabeling relation $\mathcal{R}$ is noetherian, then every labeled graph has an $\mathcal{R}-$normal form.*

Roughly speaking, a relabeling relation $\mathcal{R}$ is locally generated if the knowledge of its restriction on centered balls suffices to completely determine $\mathcal{R}$. In other words, the relabeling of a ball does not depend on the rest of the graph:

Definition 2.13 *Let $\mathcal{R}$ be a relabeling relation. The relation $\mathcal{R}$ is locally generated if for every labeled graphs (G,λ), (G,λ'), (H,η), (H,η') and every vertices $v \in V(G)$, $w \in V(H)$ such that the balls $B_G(v,1)$ and $B_H(w,1)$ are isomorphic via $\varphi\colon V(B_G(v,1)) \longrightarrow V(B_H(w,1))$ and $\varphi(v) = w$, the following three conditions:*

1. $\forall\ x \in V(B_G(v,1)) \cup E(B_G(v,1)),\ \lambda(x) = \eta(\varphi(x))$ *and* $\lambda'(x) = \eta'(\varphi(x))$,

2. $\forall\ x \notin V(B_G(v,1)) \cup E(B_G(v,1)),\ \lambda(x) = \lambda'(x)$,

3. $\forall\ x \notin V(B_H(w,1)) \cup E(B_H(w,1)),\ \eta(x) = \eta'(x)$,

imply that $(G,\lambda)\ \mathcal{R}\ (G,\lambda')$ if and only if $(H,\eta)\ \mathcal{R}\ (H,\eta')$.

Finally, local computations are the computations defined by a relation locally generated.

2.3.1 Distributed Computations of Local Computations

The notion of relabeling sequence corresponds to a notion of *sequential* computation. Let us also note that a locally generated relabeling relation allows parallel rewritings, since non-overlapping balls may be relabeled independently. Thus we can define a distributed way of computing by saying that two consecutive relabeling steps concerning non-overlapping balls may be applied in any order. We say that such relabeling steps *commute* and they may be applied concurrently. More generally, every two relabeling sequences such that the latter one may be obtained from the former one by a succession of such commutations lead to the same resulting labeled graph. Hence, our notion of relabeling sequence may be regarded as a *serialization* [Maz87] of some distributed computation. This model is clearly asynchronous: several relabeling steps *may* be done at the same time but we do not require that all of them have to be performed. In the sequel we will essentially deal with sequential relabeling sequences but the reader should keep in mind that such sequences may be done in a distributed way.

2.3.2 Proof Techniques

Graph relabeling systems provide a formal model for expressing distributed algorithms. The aim of this section is to show that this model is suitable for studying and proving properties of distributed algorithms.

Definition 2.14 *A graph relabeling system $\mathcal{R}$ is* noetherian *if there is no infinite $\mathcal{R}$-relabeling sequence starting from a graph with initial labels in $\mathcal{I}$.*

From the above definition it can be deduced that, if a distributed algorithm is encoded by a noetherian graph relabeling system then this algorithm always terminates. In order to prove that a given system is noetherian we generally use the following technique.

Let $(S, <)$ be a partially ordered set with no infinite decreasing chain. That is, every decreasing chain $x_1 > x_2 > \cdots > x_n > \ldots$ in S is finite.

Definition 2.15 *We say that $<$ is a* noetherian order compatible with $\mathcal{R}$ *if there exists a mapping f from $\mathcal{G}_\mathcal{L}$ to S such that for every $\mathcal{R}$-relabeling step $(G, \lambda, R, \varphi, \lambda')$ we have $f(G, \lambda) > f(G, \lambda')$.*

It is not difficult to see that if such an order exists then the system $\mathcal{R}$ is noetherian: since there is no infinite decreasing chain in S, there cannot exist any infinite $\mathcal{R}$-relabeling sequence.

In order to prove the correctness of a graph relabeling system, that is the correctness of an algorithm encoded by such a system, it is useful to exhibit

(i) some *invariant properties* associated with the system (by invariant property, we mean here some property of the graph labeling that is satisfied by the initial labeling and that is preserved by the application of every relabeling rule).

(ii) some properties of irreducible graphs. These properties generally allow to derive the correctness of the system.

We illustrate these techniques by considering the following simple graph relabeling system $\mathcal{R}_1 = \{\mathcal{L}_1, \mathcal{I}_1, P_1\}$. We define $\mathcal{L}_1 = \{N, A, 0, 1\}$, $\mathcal{I}_1 = \{N, A, 0\}$ and $P = \{R\}$ where R is the relabeling rule defined below. We assume that initially all edges have label 0, a unique vertex is labeled A and all other vertices are N-labeled.

At each step of the computation, an A-labeled vertex u may activate any of its neutral neighbors, say v. In that case, u keeps its label, v becomes A-labeled and the edge $\{u, v\}$ becomes 1-labeled. Hence, several vertices may be active at the same time. Concurrent steps will be allowed provided that two such steps involve distinct vertices. The computation stops as soon as all the vertices have been activated. Thereafter, the spanning tree is given by all the 1-labeled edges. Figure 7 describes a sample computation using this algorithm. According to the previous discussion, the reader should keep in mind that some of the relabeling steps *may* be applied concurrently.

Termination: Let f be the mapping from $\mathcal{G}_{\mathcal{L}_1}$ to the set of natural integers $\mathbb{N}$ which associates with each $\mathcal{L}_1$-labeled graph the number of its **N**-labeled vertices. Observing that this number

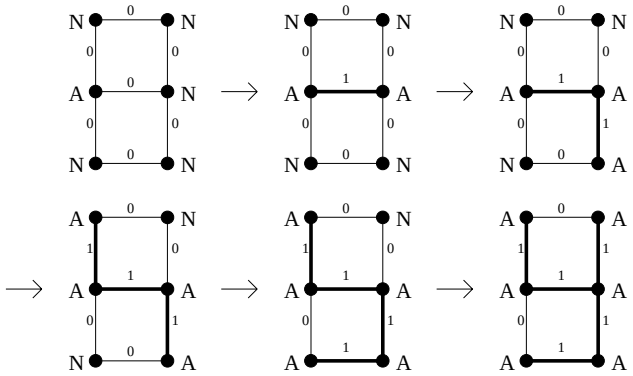


Figure 7: Distributed computation of a spanning tree

strictly decreases when we apply the relabeling rule R_1 we get that $(\mathbb{N}, >)$ is a noetherian order compatible with the system $\mathcal{R}_1$. Thus $\mathcal{R}_1$ is a noetherian system.

Correctness: Let $\mathbf{G}$ be a $\mathcal{L}_1$-labeled graph and P_1, P_2 be the following properties:

P_1 : Every **1**-labeled edge is incident with two **A**-labeled vertices,

P_2 : The subgraph of $\mathbf{G}$ made of the **1**-labeled edges and the **A**-labeled vertices has no cycle.

Every $\mathcal{I}_1$-labeled graph satisfies P_1 and P_2 since it has no **1**-labeled edge. Moreover, these two properties are clearly preserved when we apply the rule R_1. Thus, P_1 and P_2 are invariant with respect to $\mathcal{R}_1$.

Let now $\mathbf{G}$ be any $\mathcal{I}_1$-labeled graph having at least one **A**-labeled vertex and $\mathbf{G}'$ be a labeled graph in $Irred_{\mathcal{R}_1}(\mathbf{G})$. Considering the relabeling rule R_1, $\mathbf{G}'$ cannot have any **N**-labeled vertex. From property P_2, we get that the subgraph of $\mathbf{G}'$ induced by the **1**-labeled edges has no cycle. If $\mathbf{G}$ has exactly one **A**-labeled vertex we thus obtain a spanning tree of G. If $\mathbf{G}$ has more than one **A**-labeled vertex we obtain a spanning forest having as many components as the number of these initially **A**-labeled vertices.

The complexity of a distributed algorithm encoded by a graph relabeling system can also be studied by using classical techniques from rewriting theory. The space complexity is well-captured by the number of labels that are used, and the (sequential) time complexity by the length of a relabeling sequence. The degree of parallelism may also be measured by considering the ratio between the length of a parallel relabeling sequence and the length of a sequential relabeling sequence. Of course, this ratio strongly depends on the specific topology of the graph under consideration.

2.3.3 Notations

We explain the convention under which we will describe graph relabeling systems through out the rest of this thesis. If the number of rules is finite then we will describe all rules by their preconditions and relabellings. We will also describe a family of rules by a generic rule scheme ("meta-rule"). In this case, we will consider a generic star-graph of generic center v_0 and of generic set of vertices $B(v_0, 1)$. If $\lambda(v)$ is the label of v in the precondition, then $\lambda'(v)$ will be

its label in the relabeling. We will omit in the description labels that are not modified by the rule. This means that if $\lambda(v)$ is a label such that $\lambda'(v)$ is not explicitly described in the rule for a given v, then $\lambda'(v) = \lambda(v)$. The same definitions also hold for the relabeling of edges. The following relabeling rule depicts the above described rule $\mathcal{R}$ used for the computation of a spanning tree in a connected anonymous graph G.

R1 : **Find and Mark**

Precondition :

- $\lambda(v_0) = \mathrm{A}$,
- $\exists v \in B(v_0, 1)(v \neq v_0 \wedge \lambda(v) = \mathrm{N} \ \wedge \lambda(v_0, v)) = 0$.

Relabeling :

- $\lambda'(v_0, v) := 1$,
- $\lambda'(v) := A$.

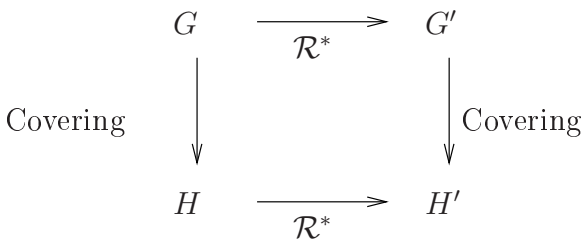
2.4 Local Computations and Coverings

We now present the fundamental result connecting coverings and locally generated relabeling relations. Its states that whenever $\mathbf{G}$ is a covering of $\mathbf{H}$, then every local computation in $\mathbf{H}$ can be lifted to a local computation in $\mathbf{G}$, which is compatible with the covering relation.

Lemma 2.1 (Lifting Lemma) *Let $\mathcal{R}$ be a locally generated relabeling relation and let $\mathbf{G}$ be a covering of $\mathbf{H}$ via γ. Moreover, let $\mathbf{H}\,\mathcal{R}^*\,\mathbf{H}'$. Then there exists a labeled graph $\mathbf{G}'$ such that*

- $\mathbf{G}\,\mathcal{R}^*\,\mathbf{G}'$ *and*
- $\mathbf{G}'$ *is a covering of* $\mathbf{H}'$.

Proof. It suffices to show the claim for the case $\mathbf{H}\,\mathcal{R}\,\mathbf{H}'$. Suppose that the relabeling step changes the labels of the ball $B_H(v)$, for some vertex $v \in V(H)$. We may apply this relabeling step to each of the disjoint labeled balls of $\gamma^{-1}(B_H(v))$, since they are isomorphic to $B_H(v)$. This yields the labeled graph $\mathbf{G}'$ which satisfies the claim. This result is depicted in the following commutative diagram:



□

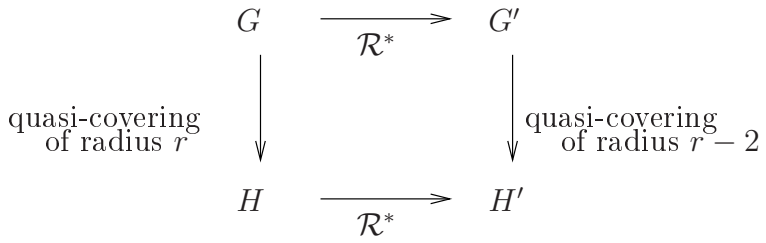
2.5 Local Computations and Quasi-Coverings

In contrast with the situation described in Section 2.4, we use quasi-coverings to depict configurations where only relabeling chains of bounded length can be simulated.

Lemma 2.2 (Quasi-Lifting Lemma) *Let $\mathcal{R}$ be a locally generated relabeling relation and let $\mathbf{G}$ be a quasi-covering of $\mathbf{H}$ of radius r via γ. Moreover, let $\mathbf{H}\,\mathcal{R}\,\mathbf{H}'$; Then there exists $\mathbf{G}'$ such that*

- *$\mathbf{G}\,\mathcal{R}^*\,\mathbf{G}'$ and*
- *$\mathbf{G}'$ is a quasi-covering of radius $r-2$ of $\mathbf{H}'$.*

Proof. Let $\mathbf{G}_0$ be the associated covering, z be the center of the ball of radius r and δ be defined as introduced in Definition 1.25. Suppose now that the relabeling step $\mathbf{H}\,\mathcal{R}\,\mathbf{H}'$ applies rule R_0 and modifies labels in the ball $B_H(v)$, for some vertex $v \in V(H)$. R_0 can also be applied to all the balls $\delta^{-1}(B_H(v))$ yielding $\mathbf{G}'_0$ and δ'. It applied also to the balls $\gamma^{-1}(B_{\mathbf{H}}(v))$ that are included in $B_G(z,r)$, since they are also isomorphic to $B_{\mathbf{H}}(v)$. We get $\mathbf{G}'$ and γ' satisfying the quasi-covering properties with radius $r-2$: consider w in $B_{\mathbf{G}}(z,r-2)$: since any ball containing w is included in $B_{\mathbf{G}}(z,r)$, w and $\gamma'(w)$ have the same label. This result is depicted in the following commutative diagram:



□

We illustrate the use of the covering concepts in the local computations framework by considering the election problem in asynchronous distributed systems.

The Election Problem. The election problem is one of the paradigms of the theory of distributed computing [Tel00]. Considering a network of processors we say that a given processor p has been *elected* when the network is in some global state such that the processor p knows that it is the elected processor and all other processors know that they are not. Using our terminology, it means that we get a labeling of the graph in which a unique vertex has some distinguished label.

This problem may be considered under various assumptions [Tel00]: the network may be directed or not, the network may be anonymous (all vertices have the same initial label) or not (every two distinct vertices have distinct initial labels), all vertices, or some of them, may have some specific knowledge on the network or not. This knowledge concerns, for instance, the diameter of the network, the total number of vertices or simply an upper bound of these parameters. A general impossibility result which summarize previous results has been obtained in [GMM00]. We say that v knows the topology of G if v's label encodes the incidence matrix of a graph $G' \simeq G$, but no information enables v to know which vertex of G' corresponds to v.

Let G be a graph which is not covering-minimal and let H be such that G is a proper covering of H via the morphism γ. A subgraph K of G is *free modulo* γ if $\gamma^{-1}(\gamma(K))$ is a disjoint union of graphs isomorphic to K. We say that a labeling λ is $\gamma-$*lifted* if

$$\gamma(x) = \gamma(y) \Longrightarrow \lambda(x) = \lambda(y)\,.$$

We say that an algorithm *operates on subgraphs from a given set* S if every relabeling step is performed only on subgraphs of the given graph, which belong to S. Using coverings Godard et al. [GMM00] have proved the next proposition.

Proposition 2.1 *Let G be a graph which is not covering-minimal and let γ be a covering from G onto some graph $H \not\simeq G$. Then there is no election algorithm for G which operates on subgraphs free modulo γ, even if the topology of G is known by each vertex of G.*

In [Maz97], Mazurkiewicz gives an election algorithm for the family of graphs which are minimal for the covering relation when we know the size; a characterization of families of graphs for which there exists an election algorithm has been obtained in [GM02]. This characterization relies on the following theorem.

Theorem 2.1 *Let $\mathcal{G}$ be a class of connected labeled graphs. There exists an election algorithm for $\mathcal{G}$ if and only if elements of $\mathcal{G}$ are minimal for the covering relation and $\forall \mathbf{G} \in \mathcal{G}\ \exists h_{\mathbf{G}} \geq 0$ such that $\mathbf{G}$ has not quasi-coverings of size greater than $h_{\mathbf{G}}$ in $\mathcal{G}$.*

Many other works have been performed in order to get a better characterization of several distributed computation models. Yamashita and Kameda [YK96], Boldi et al. [BCG+96b] and Chalopin et al. [CM04, CMZ04] characterize families of graphs in which election is possible under different models of distributed computations. All these works took advantage of the covering concepts or of other mathematical notions such as *symmetricity* or *fibration*, that are closely related to coverings.

First Part

Synchronizers in the Local Computations Framework

A network synchronization consists in taking measures to define a structure which allows to control the computation steps of different processes. In this sequel, we are specially interested in structures that enable to divide the computation of an asynchronous distributed system in rounds such that there exists a small upper bound (generally 1) on the round difference between any two processes in the whole network. Once a network synchronization has been reached, it is possible to run any synchronous distributed algorithm in the so constructed distributed framework.

In this part, we will first present the notion of *network synchronizer* and state the most important properties that are inherent to any network synchronizer. Thereafter, we will depict the example of a simple synchronizer that does not need any knowledge on the underlying network and ensures for any process a synchronization at distance at most 1.

Throughout the rest of this part, we will point out various synchronization protocols encoded by local computations. These are :

- Synchronizer based on the algorithm from Shi et al. [SSP85] (Section 3.3).
- Randomized synchronization (Section 3.4).
- Synchronization in networks with distinguished process (Section 3.5).
- Synchronization in tree shaped networks (Section 3.6).

In the last section, we will introduce a general methodology that enables the construction of synchronizers in fully asynchronous distributed systems. This methodology will be backed up by an example showing a synchronizer using the SSP procedure. All the results of this part appear in the Proceeding of the Second International Conference on Graph Transformations [MMOS04].

Chapter 3

Synchronizing Distributed Algorithms

Contents

3.1 The Synchronizers

Distributed algorithms are studied under various models. One fundamental criterion is synchronous or asynchronous [AW98, CM88, Lyn96, Tel00].

In the synchronous model, we assume that there is a global clock and that the operations of components take place simultaneously: at each clock tick an action takes place on each process. This is an ideal timing model and this is not what happens in most distributed systems.

In fact, synchronization in networks may be viewed as a control structure which enables to control relative steps of different processes; it may be illustrated by the following examples:

1. the processors execute actions in lock-steps called pulses or rounds. In a pulse, a process p executes the following sequence of discrete steps:

   (a) p sends a message,

   (b) p receives some messages
   (c) p performs local computations;

2. another assumption is that computation events of pulse p appear after computation events of pulse $p-1$ and all messages sent in pulse p are delivered before computations events of pulse $p+1$;

3. if each process is equipped with a counter for local computations, we may assume that the difference between two counters is at most 1; and more generally, for a given non negative integer k we may assume that the difference between any two counters is at most k.

4. some algorithms need some synchronization barriers applied to a group of processes, it means that all group members are blocked until all processes of the group have reached this barrier.

In the asynchronous model there is no global clock, separate components take steps at arbitrary relative speeds. It is assumed that messages are delivered, processes perform local computations and send messages, but no assumption is made about how long it may take.

There exists also intermediate models like models assuming the knowledge of bounds on the relative speeds of processors or links.

In this chapter we present several methods for the simulation of synchrony on asynchronous distributed systems by means of local computations. These methods try to reduce the variation of pulses between the processes of the network. Synchronizations have been already used in the context of local computations in [BCG+96a, RFH72, KY96] they enable, for example, the computation of the view (the local knowledge) of a vertex.

Concepts of a synchronizer. A synchronous distributed system is organized as a sequence of pulses: in a pulse each processor performs a local computation. In an asynchronous system the speed of processors can vary, there is no bounded delay between consecutive steps of a processor. A synchronizer is a mechanism that transforms an algorithm for synchronous systems into an algorithm for asynchronous systems.

As the non-determinism in synchronous systems is weaker, in general, algorithms for synchronous systems are easier to design and to analyze than those for asynchronous networks. In asynchronous systems, it is difficult to deal with the absence of global synchronization of processes. Consequently, it is useful to have a general method to transform an algorithm for synchronous networks into an algorithm for asynchronous networks. Therefore, it becomes possible to design a synchronous algorithm, test it and analyze it and then use the standard method to implement it on an asynchronous network. A synchronizer operates by generating a sequence of local clock *pulses* at each processor. An introduction and the main results about synchronizers may be found in [AW98, CM88, Lyn96, Pel00, Tel00].

3.2 Synchronizer Properties

Throughout the rest of this chapter, several distributed synchronization protocols will be presented as well was the proofs related to their correctness.

In this framework, the operations of processes take place in a sequence of discrete steps called pulses: we represent a pulse by a counter, then we associate to each process a pulse number which is initialized to 0 or 1. At each step, a process goes from pulse i to pulse $i+1$.

All of these protocols involve synchronizing the system at *every synchronous round*. This is necessary because the protocols are designed to work for arbitrary synchronous algorithms. All the synchronizers we will build are "global", in the sense that they involve synchronization among arbitrary vertices in the whole network. To preserve this "global" synchronization, each synchronizer has to satisfy some properties. The essential property we seek to preserve in translating a generic synchronous algorithm $\mathcal{A}_s$ into an asynchronous algorithm $\mathcal{A}_{as}$ is that the pulse difference between two arbitrary vertices is at most 1. In order to ensure the overall and anytime satisfaction of this property, we begin by requiring *pulse compatibility* in the network. This means that a vertex can only increase its pulse, when it is sure that there is no vertex in the network that is still in a lower pulse. This is guaranteed by the validity of Property 2. Furthermore, we strengthen our synchronization assumption by forcing *pulse convergence* at any time. By *pulse convergence* we mean the fact that all the vertices of a network have simultaneously to be in pulse π before any vertex starts the pulse $\pi+1$. Pulse convergence is stated as Property 1. Thus, the correctness of Property 3 is directly deduced from the pulse compatibility and the pulse convergence properties.

Property 1 (Convergence) *Let $\pi \quad (\pi > 0)$ be the maximum pulse that has been reached so far. After a finite number of steps $T_\pi \geq 0$, all the vertices of G are in the same pulse π.*

Property 2 (Pulse compatibility) *A vertex u in G changes its pulse $p(u)$ only when there is no vertex v in G such that $v \neq u$ and $p(v) < p(u)$.*

Property 3 (Speed Limitation) *At any time t, the pulse difference between two vertices v and u of a network G is at most 1.*

Definition 3.1 (Correctness) *A synchronizer is correct if the convergence, pulse compatibility and the speed limitation properties are satisfied.*

A simple Synchronizer. We recall here the synchronization as presented and used in [RFH72]. On each vertex v of a graph there is a counter $p(v)$, the initial value of $p(v)$ is 0. At each step the value of the counter $p(v_0)$ depends on the value of the counters of the neighbors of v_0 more precisely if $p(v_0) = i$ and if for each neighbor v of v_0 $p(v) = i$ or $p(v) = i+1$ then v_0 is considered as *safe* and the new value of $p(v_0)$ is $i+1$.

R1 : **The synchronization rule**

Precondition :

- $p(v_0) = i$,
- $\forall v \in B(v_0, 1)\ p(v) = i$ or $p(v) = i+1$.

Relabeling :

- $p(v_0) := i+1$.

Proposition 3.1 *For all vertices v_1 and v_2*

$$|p(v_1) - p(v_2)| \leq d(v_1, v_2).$$

Proof. According to rule R1, this proposition can be shown by a simple induction on the distance between vertices □

Remark 3.1 *This synchronization does not need any knowledge on the graph, in particular on its size.*

This synchronization is equivalent to the synchronization described in the introduction where processors execute actions in lock-steps called pulse (or round): in a pulse a process sends a message, receives some messages and performs a local computation.

In fact, to implement this synchronization, a counter modulo 3 is sufficient: each process needs to compare the value of its counter to the value of each neighbor. More precisely, for each process v_0 and for each neighbor v of v_0 we determine if : $p(v_0) = p(v) - 1$, or $p(v_0) = p(v)$, or $p(v_0) = p(v) + 1$. Finally, the synchronization may encoded by:

R'1 : **The synchronization rule**

Precondition :
- $p(v_0) = i$,
- $\forall v \in B(v_0, 1)\ p(v) = i \bmod 3$ or $p(v) = (i+1) \bmod 3$.

Relabeling :
- $p(v_0) := (i+1) \bmod 3$.

The α synchronizer [Tel00] is similar to the synchronizer presented in this section. In fact, the precondition of rule R'1 expresses that the vertex v_0 is safe. In contrast to the α synchronizer, a vertex v_0 generates the next pulse as soon as it is safe. It does not wait until all its neighbors become safe. This synchronizer does not guarantee the *pulse compatibility* and the *Convergence* properties. Thus, it does not satisfy Property 3. It is therefore not appropriated to perform a "global" synchronization in a network. The next section will be devoted to the presentation of synchronizers that are able to preserve a "global" synchronization at every pulse.

3.3 A Synchronizer Protocol Based on the SSP Algorithm

3.3.1 The SSP Algorithm

We describe in our framework the algorithm by Szymanski, Shi and Prywes (the SSP algorithm for short) and then the synchronizer.

We consider a distributed algorithm which terminates when all processes reach their local termination conditions. Each process is able to determine only its own termination condition. The SSP algorithm detects an instant in which the entire computation is achieved.

Let G be a graph such that a boolean predicate $P(v)$ and an integer $a(v)$ is associated with each vertex v in G. Initially $P(v)$ is *false*, the local termination condition is not reached, and $a(v)$ is equal to -1. Transformations of the value of $a(v)$ are defined by the following rules.

Each local computation acts on the integer $a(v_0)$ associated to the vertex v_0; the new value of $a(v_0)$ depends on values associated to vertices of $B(v_0, 1)$. More precisely, let v_0 be a vertex and let $\{v_1, ..., v_d\}$ the set of vertices adjacent to v_0.

- If $P(v_0) = false$ then $a(v_0) = -1$;
- if $P(v_0) = true$ then $a(v_0) = 1 + Min\{a(v_k) \mid 0 \leq k \leq d\}$.

We consider the following assumption: for each vertex v the value of $P(v)$ eventually becomes true and remains true for ever. We will use the following notation.

Definition 3.2 (Associated relabeling chain) *Let $(\mathbf{G}_i)_{0 \leq i}$ be a relabeling chain associated to SSP's algorithm. We denote by $a_i(v)$ (resp. $P_i(v)$) the integer (resp. the boolean) associated to the vertex v of $\mathbf{G}_i$.*

Proposition 3.2 ([SSP85]) *Let $(\mathbf{G}_i)_{0 \leq i \leq n}$ be a relabeling chain associated to SSP's algorithm; let v be a vertex of G, we suppose that $h = a_i(v) \geq 0$. Then :*

$$\forall w \in V(G) \quad d(v, w) \leq h \Rightarrow a_i(w) \geq 0.$$

Let P^G be a global predicate that is satisfies if and only if $P(v)$ is true for all vertices v of the graph G. From the above property we deduce that a vertex v can detect the validity of P^G if it has knowledge of one of the following network parameters:

1. The size n of the graph G
2. An upper bound m of the size n
3. The diameter $\Delta(G)$ of the graph G
4. An upper bound of the diameter $\Delta(G)$

3.3.2 The SSP Synchronizer

In this subsection we introduce a new synchronization protocol. This protocol is based on the SSP algorithm.

Let u be a vertex, the integer $p(u)$ denotes the value of the pulse associated to the vertex u. In our assumption, a vertex v satisfies the stable properties, if there is no vertex $u \in B_G(v, 1)$ such that $p(u) \neq p(v)$. Before giving a formal description of this algorithm, we have to explain the way it works.

Let G be a graph with diameter $D = \Delta(G)$. To each vertex v of G, we associate two integers $p(v)$ and $a(v)$, where $p(v)$ denotes the pulse and $a(v)$ denotes the SSP value for the vertex v. Initially, $p(v)$ and $a(v)$ are respectively set to 1 and 0.

A vertex v can start the next pulse ($p(v) = p(v) + 1$) and its counter $a(v)$ is reset to 0 when it detects that the value of the pulse of all the vertices is equal to $p(v)$. Now, we give a formal description of the above arguments.

Consider a labeling function λ, where $\lambda : V \to [1..\infty] \times [0..D]$. Initially all vertices are labeled $(1, 0)$. The graph relabeling system is $\mathcal{R}_1 = (L_1, I_1, P_1)$ defined by $L_1 = \{[1..\infty] \times [0..D]\}$, $I_1 = \{(1, 0)\}$, $P_1 = \{R1, R2\}$ where $R1$ and $R2$ are the relabeling rules described below. Consider a vertex v_0, the SSP synchronization is defined by:

R1 : **The observation rule**

Precondition :

- $\lambda(v_0) = (p(v_0), a(v_0))$,
- $a(v_0) < D$,
- $\forall v \in B(v_0, 1)\ p(v) \geq p(v_0)$,
- $a(v_0) = Min\{a(v) \mid v \in B(v_0, 1) \text{ and } p(v) = p(v_0)\}$.

Relabeling :

- $\lambda'(v_0) := (p(v_0), a(v_0) + 1)$.

R2 : **The changing phase rule**

Precondition :

- $\lambda(v_0) = (p(v_0), D)$.

Relabeling :

- $\lambda'(v_0) := (p(v_0) + 1, 0)$.

We denote by $p_i(v)$ (resp. $a_i(v)$) the pulse (resp. the integer) associated to the vertex v of $\mathbf{G}_i$.

For the correctness of this algorithm we state some invariants.

Fact 3.1

$$p_{i+1}(v) \geq p_i(v).$$

Fact 3.2 *If $p_{i+1}(v) = p_i(v) + 1$ then $a_i(v) = D$.*

Lemma 3.1 *If $p_i(v) = \pi$ and $a_i(v) = h$ then:*

$$\forall w \in V(G) \quad d(v, w) \leq h \Rightarrow p_{i-h}(w) \geq \pi.$$

Proof. We show the lemma by induction on i. If $i = 0$ the property is obvious.

First we assume that $p_{i+1}(v) = p_i(v) = \pi$ and $a_{i+1}(v) = a_i(v) = h$. By the inductive hypothesis: $d(v, w) \leq h \Rightarrow p_{i-h}(w) \geq \pi$. From Fact 3.1, $p_{i-h+1}(w) \geq p_{i-h}(w) \geq \pi$. Thus $p_{(i+1)-h}(w) \geq \pi$.

Now we assume that $p_{i+1}(v) = p_i(v) = \pi$ and $a_{i+1}(v) = a_i(v) + 1 = h$. If $d(v,w) \leq h = a_i(v) + 1$ then let u be such that $d(v,u) = 1$ and $d(u,w) = a_i(v) = h - 1$.

We have $a_{i+1}(v) = a_i(v)+1 \Rightarrow p_i(u) \geq p_i(v) \geq \pi$ and $a_i(u) \geq h-1$ (by the precondition of the observation rule). By the inductive hypothesis applied to the vertex u, $p_{i-(h-1)}(w) \geq \pi$ and finally $p_{(i+1)-h}(w) \geq \pi$.

The last case is $p_{i+1}(v) = p_i(v) + 1$, necessary $a_{i+1}(v) = 0$ and this achieves the proof. □
From this lemma and Fact 3.1, it follows:

Corollary 3.1 *If $p_i(v) = \pi$ and $a_i(v) = h$ then $d(v,w) \leq h \Rightarrow p_i(w) \geq \pi$.*

Lemma 3.2 *If $p_i(v) = \pi$ and $p_i(w) = \pi + 1$ then $\forall u \in V(G)$ $(p_i(u) = \pi$ or $p_i(u) = \pi + 1)$.*

Proof. Let j be such that $p_j(w) = \pi$ and $p_{j+1}(w) = \pi + 1$, by the precondition of the phase rule and the previous corollary:

$$\forall u \in V(G) \quad p_j(u) \geq \pi.$$

For the same reasons, as $p_i(v) = \pi$, there does not exist u such that $p_i(u) > \pi + 1$. □

Theorem 3.1 (Correctness) *The synchronization protocol obtained by using the graph relabeling system $\mathcal{R}_1$ is correct.*

Proof. The proof of this theorem can easily be deduced from Lemma 3.1 and Lemma 3.1 □

Remark 3.2 *A similar correct synchronization protocol can be obtained in the same way, using an upper bound of the diameter, the size $n = |V(G)|$ of the graph or an upper bound of the size of the graph.*

3.4 Randomized Synchronization Algorithm (RS Algorithm)

We present a randomized synchronization algorithm. The main idea of this algorithm is based on the use of *random walks* in a network. Initially, each vertex (processor) gets a *token*. At each step, a vertex that has a token passes it randomly to one of its neighbors. When more than one token meet at one vertex, they merge to one token. In a connected undirected graph, with high probability, all tokens will merge to one token and the vertex, that gets it, starts the next synchronization pulse.

For our purpose, we have slightly modified the above described algorithm. The motivation of this departure from the main idea is well grounded since it can be quite laborious for a vertex, that has a token, to know that there is no other token in the whole network. In order to avoid this kind of problem, we represent our tokens as natural numbers and the action of merging tokens is now done by adding the numbers corresponding to these tokens. Moreover, a vertex does not only pass one token to one of its neighbors. Rather, it first merges all the tokens of its neighborhood to one and passes the resulting number to one of its neighbors. At the beginning of each pulse, each vertex produces a new token with value 1. As soon as we have one token left, we try (if possible) to broadcast this information through the whole

graph (rule $R3$). Thus, more than one vertex could start the next pulse. If a vertex u has a token $c(u)$ such that $c(u) = |V|$, then u is allowed to start the next synchronization pulse. We assume that each vertex knows the size of the network. A formal description of the algorithm is done below.

Let G be a graph with n vertices. Consider a labeling function λ, where $\lambda : V \to [1..\infty] \times [0..n]$. The first item of the label of a vertex u represents the pulse of u and the second represents the minimum number of vertices that are in the same pulse as u. Initially, at least one vertex is labeled $(1,1)$ and the others are $(0,0)$-labeled. The graph relabeling system is $\mathcal{R}_2 = (L_2, I_2, P_2)$ defined by $L_2 = \{[1..\infty] \times [0..n]\}$, $I_2 = \{\{(1,1)\}\}$, $P_2 = \{R1, R2, R3, R4\}$ where $R1$, $R2$, $R3$ and $R4$ are the relabeling rules listed below. Let v_0 be a vertex, the randomized synchronization algorithm (RS algorithm in the sequel) is described as follow.

R1 : **The convergence rule**

Precondition :

- $\lambda(v_0) = (p(v_0), c(v_0))$,
- $c(v_0) < |V|$,
- $\exists v \in B(v_0, 1)\ \ p(v) = p(v_0) + 1$.

Relabeling :

- $\lambda'(v_0) := (p(v_0), |V|)$.

In the *convergence rule*, each vertex v_0 sets the value of its token $c(v_0)$ to $|V|$ as soon as there exists one vertex $v \in B_G(v_0, 1)$ that is in a greater pulse than $p(v_0)$.

R2 : **The phase rule**

Precondition :

- $\lambda(v_0) = (p(v_0), c(v_0))$,
- $c(v_0) = |V|$.

Relabeling :

- $\lambda'(v_0) := (p(v_0) + 1, 1)$.

In the *phase rule*, each vertex v_0 that has a token $c(v_0)$ such that $c(v_0) = |V|$ increases its pulse number and sets the value of its token to 1.

R3 : **The propagation rule**

Precondition :

- $\lambda(v_0) = (p(v_0), c(v_0))$,
- $c(v_0) < |V|$,
- $\exists v \in B(v_0, 1)\ \ c(v) = |V|$ and $p(v) = p(v_0)$.

Relabeling :

- $\lambda'(v_0) := (p(v_0), |V|)$.

In the *propagation rule*, a vertex v in the neighborhood of v_0 is in the same pulse $p(v_0)$ and $c(v) = |V|$ then the token of v_0 is set to $|V|$. This rule has only the final aim to broadcast the token information $c(v) = |V|$.

R4 : **The collecting rule**

Precondition :

- $\lambda(v_0) = (p(v_0), c(v_0))$,
- $c(v_0) < |V| \wedge c(v_0) > 0$,
- $\forall v \in B(v_0, 1)\ \ p(v) = p(v_0)$.

Relabeling :

- $w_0 :=$ *choose at random* $v \in B(v_0, 1)(v \neq v_0)$,
- $\forall v \in B(v_0, 1)(v \neq v_0)\lambda'(v) := (p(v), 0)$,
- $\mathcal{S} := \sum_{v \in B(v_0,1)} c(v)$,
- $\lambda'(w_0) := (p(w_0), \mathcal{S})$.

In the *collecting rule*, each vertex in the neighborhood of v_0 is in the same pulse $p(v_0)$. If v_0 has a token of value $c(v_0) < |V|$, then the sum of all the tokens in $B_G(v_0, 1)$ is computed and the resulting token is send to one neighbor w_0 of v_0. Note that the choice of w_0 is done randomly. The tokens of all the vertices $v \in B_G(v_0, 1)$ such that $v \neq w_0$ are set to 0.

We now turn our attention to the correctness of the described algorithm. Therefore, we first state some useful properties that will help us to show the validity of our synchronization assumption. One has also to notice that a vertex that does not have a token gets a *pseudo-token* with value 0. Let $(\mathbf{G}_i)_{0 \leq i}$ be a relabeling chain associated to a run of the RS Algorithm. We denote by $p_i(v)$ (resp. $c_i(v)$) the pulse (resp. the integer) associated to the vertex v of $\mathbf{G}_i$. We assume that initially exactly one vertex of $\mathbf{G}_0$ is labeled $(1, 1)$ and all the other vertices are labeled $(0, 0)$. As from now, we are going to pay attention to the validity of the properties of our algorithm.

First we have the two following facts.

Fact 3.3

$$\forall v \in V(G) \quad \forall i \qquad p_{i+1}(v) \geq p_i(v).$$

$$\forall v \in V(G) \quad \forall i \qquad p_{i+1}(v) = p_i(v) + 1 \Rightarrow c_i(v) = |V|.$$

Lemma 3.3 *Let π be a pulse, we have:*

1. *If there exists v_0 such that $0 < c_i(v_0) < |V|$ and $p_i(v_0) = \pi$ then:*

$$\sum_{\{v | p_i(v) = \pi\}} c_i(v) = Card\{v \mid p_i(v) = \pi\}.$$

2. *If there does not exist* v_0 *such that* $0 < c_i(v_0) < |V|$ *and* $p_i(v_0) = \pi$ *then:*

$$\forall v \quad \textit{such that} \quad p_i(v) = \pi \quad \textit{either} \quad c_i(v) = 0 \quad \textit{or} \quad c_i(v) = |V|.$$

Furthermore if for all v *such that* $p_i(v) = \pi$ *we have* $c_i(v) = 0$ *then there exists* w *such that* $p_i(w) = \pi + 1$.

3. $(\exists v, w$ *such that* $p_i(v) = p_i(w) + 1) \Rightarrow (c_i(w) = 0$ *or* $c_i(w) = |V|)$.

4. *if* $\exists v, w$ *such that* $p_i(v) = p_i(w) + 1$ *then*

$$\sum_{\{u | p_i(u) = p_i(v)\}} c_i(u) = Card\{u \mid p_i(u) = p_i(v)\}.$$

5. *If there exists a vertex* v *such that* $c_i(v) = |V|$ *then for all vertex* u *we have:* $p_i(u) \geq p_i(v)$.

6.

$$\forall u, v \quad |p_i(u) - p_i(v)| \leq 1.$$

Proof. The proof is by induction on i. We assume that all the properties are true at step i and then we examine what happens according to the relabeling rule which is applied. □

For the validity of our synchronization assumption we have to show that each vertex of G has enough knowledge of the whole graph to decide, if necessary, to change its pulse. This knowledge can only be achieved if we are able to ensure that after a finite time, and with high probability, there is only one token left in the whole network. We are now going to undertake the task of demonstrating that one can make this probability as close to 1 as desired.

3.4.1 Correctness of the Algorithm

To state the correctness of the RS Algorithm, we introduce some well-known topics related to the use of random walks and Markov chains [Bre99] in graphs. The idea is to represent the successive moves of any token as a random walk.

Consider the random walk due to successive applications of the fourth rule of our randomized algorithm. This random walk also constitutes a *Markov chain* with the transition probability

$$\mathcal{P}(u, v) = \begin{cases} \frac{1}{d(u)+1} & : \quad v \in B_G(u, 1) \\ 0 & : \quad \text{otherwise} \end{cases}$$

Now we intend to show that the so defined Markov chain $\{Y_k\}_{k \in \mathbb{N}}$ satisfies the conditions of the *Basic Limit Theorem* (see Theorem 1.2). To this end we have to show that $\{Y_k\}_{k \in \mathbb{N}}$ is irreducible, aperiodic and that we have a stationary distribution π. Because of the fact that the graph G is connected and undirected, it is quite simply to show that $\{Y_k\}_{k \in \mathbb{N}}$ is irreducible. From Claim 1.20 we can deduce that our Markov chain is also aperiodic, since for all vertices $u \in V$ we have $\mathcal{P}(u, u) > 0$. The non trivial part of this proof is to show the existence of a stationary distribution π. For this purpose we can take advantage of Proposition 1.2 and state the following claim.

Claim 3.1 *$\{Y_k\}_{k\in\mathbb{N}}$ is reversible with respect to*

$$\pi(u) = \frac{d(u)+1}{2|E|+|V|}$$

Proof. To see the validity of the above claim, we have to check two cases: Let $u, v \in V$ be two vertices from G.

- if $v \notin B_G(u,1)$ then $\pi(u)\mathcal{P}(u,v) = \pi(v)\mathcal{P}(v,u) = 0$.
- if $v \in B_G(u,1)$ then

$$\pi(u)\mathcal{P}(u,v) = \frac{d(u)+1}{2|E|+|V|}\frac{1}{d(u)+1} = \frac{1}{2|E|+|V|} = \pi(v)\mathcal{P}(v,u)$$

The value of $\pi(u)\mathcal{P}(u,v)$ is independent of u. Thus $\{Y_k\}_{k\in\mathbb{N}}$ is reversible and Theorem 1.1 allows us to concede that π is the unique stationary distribution of the Markov chain. □

Now it is clear that the *Basic limit Theorem* is applicable to our defined Markov chain. Thus, we have that in the limit, the probability of being at any particular vertex is proportional to its degree, regardless of the structure of G. This remarkable fact is the key to the proof of the next Lemma.

Lemma 3.4 *After a probably finite number of steps, there is at least one vertex u in G that satisfies $c(u) = |V|$.*

Proof. It clearly suffices to consider the case where the system begins with just two tokens. These tokens perform two *distinct* random walks in G and produce two Markov chains that start from two different vertices. According to the *Basic limit Theorem*, the probability, from these tokens to merge, will be very close to 1 when they reach their common stationary distribution π. In fact, it is like (for a graph with six vertices) repeated rolls of two dice (simultaneously) with the expectation of having two identical faces. Thus, it is clear that, in the limit, the merging probability becomes closer to 1.

Furthermore, Coppersmith, Tetali and Winkler [CTW93] have obtained a polynomial upper bound for the meeting time of two random walks on a graph G. They have shown that in the worst case, two random walks meet after an expected number of moves M with $M \leq \frac{4}{27}n^3$. □

Lemma 3.5 *The RS algorithm satisfies the conditions of Property 1 (Convergence).*

Proof. For the proof of Lemma 3.5 we first assume that after a finite number of computation steps, there is at least one vertex u in G that satisfies $c(u) = |V|$. The validity of this fact is a direct consequence of Lemma 3.4.

Let u be any vertex that satisfies $p(u) = \pi$. Only the application of rule $R2$ can increment the pulse of vertex u. This means that $c(u) = |V|$ and that all the other vertices of G have sent their tokens for the pulse π. Thus, all the vertices of G are in the same pulse π.

Let now v be a vertex of G such that $p(v) < \pi$. The rules $R1$ and $R2$ are applicable on v until $p(v) = \pi$. But as soon as the value of $p(v)$ reaches π, only the rule $R2$ is able to

increment $p(v)$. Therefore, all the vertices of G must have sent their tokens for the pulse π. Thus, all the vertices of G are also in the pulse π. □

We now go further in our demonstration and prove the validity of Property 2 concerning the *pulse compatibility* of the RS Algorithm.

Lemma 3.6 *The RS algorithm satisfies the conditions of Property 2 (Pulse compatibility).*

Proof.
A vertex u can only changes its pulse if it applies one of the rules $R1$ and $R2$. Further on, the use of $R1$ ensures that there exists a vertex $v \neq u$ with $p(v) = p(u) + 1$. Due to the initial configuration and to Property 1 we can claim that Property 2 is satisfied. The use of $R2$ combined with *Property* 3 of Lemma 3.3 also asserts the claim of this lemma. □

From Lemma 3.5 and Property 2 we can immediately deduce the correctness of Property 3 (Speed limitation).

3.5 Synchronization in Graphs with a Distinguished Vertex

In this section, a new synchronization protocol is introduced. The approach of the new methodology is based on two well-known paradigms in distributed computations. While the first one is the computation of a spanning tree in a graph, the election in anonymous trees represents the second one. The algorithms for the election and for the computation of a spanning tree use local computation techniques with graph relabeling systems to solve both problems with optimality.
The new protocol works in two steps. In the first step, a spanning tree T is computed in a given graph G. The goal of the second step is then to elect a vertex in T that will start the next pulse of our synchronization protocol. Thereafter, the algorithm enters a new cycle by executing a new computation of a spanning tree in G. There are no restrictions about the way we carry the both steps out. This means that while the tree T is constructed, the election algorithm can be simultaneously performed. Before going on with the presentation of the protocol, let us first give some definitions and formalize the above description in terms of graph relabeling systems.

Definition 3.3
The label of each vertex is represented by a two tuple (S, i) where $S \in \{N, A\}$ and $i \in \mathbb{N}$.

- *A vertex v is N-labeled if the first item of its label is N.*
- *A vertex v is A-labeled if the first item of its label is A.*
- *An edge e is i-labeled if its label is i.*
- *A pendant vertex v is a vertex that has exactly one N-labeled neighbor.*

- *A subgraph G_i is called N-labeled (respectively A-labeled) if all its vertices are N-labeled(respectively A-labeled).*

Let now G be a graph with a distinguished vertex u. Initially all edges are 0-labeled, a vertex u is labeled $(A,1)$, and excepted u, all the vertices are labeled $(N,0)$. The corresponding graph relabeling system is given by: $\mathcal{R}_3 = (L_3, I_3, P_3)$ where the components L_3, I_3 and P_3 are defined as follows:
$L_3 = \{\{A,N\} \times [0..\infty] \cup \{0,1\}\}$, $I_3 = \{\{(A,1),(N,0)\} \cup \{0\}\}$, and $P_3 = \{R1, R2, R3\}$ where $R1$, $R2$ and $R3$ are the relabeling rules described by Procedure 3.1.

Procedure 3.1 (Protocol for graphs with a distinguished vertex)

R*1* : **Tree computation rule**

Precondition :
- $\lambda(v_0) = (label(v_0), p(v_0))$,
- $label(v_0) = N$,
- $\exists v \in B(v_0,1)\ p(v) > p(v_0), label(v) = A$,

Relabeling :
- $\lambda'(v_0) := (A, p(v_0)+1)$,
- $\lambda'(v_0, v) = 1$.

R*2* : **Leaf elimination rule**

Precondition :
- $\lambda(v_0) = (label(v_0), p(v_0))$,
- $label(v_0) = A$,
- $\exists v \in B(v_0,1)\ label(v) = A,\ p(v) = p(v_0),\ \lambda(v_0, v) = 1$,
- $\forall w \in B(v_0,1)/v\ p(w) = p(v_0),\ \lambda(v_0, w) = 0$,

Relabeling :
- $\lambda'(v_0) := (N, p(v_0))$,
- $\lambda'(v_0, v) = 0$.

R*3* : **Tree election rule**

Precondition :
- $\lambda(v_0) = (label(v_0), p(v_0))$,
- $label(v_0) = A$,
- $\forall v \in B(v_0,1)\ p(v) = p(v_0),\ label(v) = N$,

Relabeling :
- $\lambda'(v_0) := (A, p(v_0)+1)$.

Rule R1 generates a spanning tree T that only contains A-labeled vertices and 1-labeled edges, while rules R2 and R3 perform the election algorithm in T.

Let now $G(V, E, \lambda)$ be a connected labeled graph such that a distinguished vertex v is labeled $(A, 1)$ and all other vertices are labeled $(N, 0)$. At the beginning, all the edges are labeled 0. Let $G'(V, E, \lambda')$ be a graph such that: $G(V, E, \lambda) \xrightarrow[\mathcal{R}_3]{*} G'(V, E, \lambda')$.
We are going to present some invariants that are satisfied by the graph $G'(V, E, \lambda')$. These invariants are given in form of five lemma whose proofs should simplify the comprehension of the results we will state later. The rest of this section will be dedicated to the correctness of the above presented synchronization methodology.

Lemma 3.7 *There exists at least one A-labeled vertex in G.*

Proof. The validity of this property depends entirely on rule R2. As a matter of fact, rule R2 ensures that a A-labeled vertex v can only change its label, when there is exactly one A-labeled vertex u in its neighborhood, such that v and u are incident to the same 1-labeled edge. Furthermore, rule R3 ensures that if a A-labeled vertex v has only N-labeled vertices in its neighborhood, then v continues to be A-labeled until the next pulse is started. Thus, at any time, there is at least one A-labeled vertex in the network. □

Lemma 3.8 *All edges incident to a (N, i)-labeled vertex are 0-labeled.*

Proof. We assume that v is any N-labeled vertex. Further, let S be the set of N-labeled vertices u such that v and u are incident to the same 1-labeled edge. The precondition of rule R2 ensures that v can not have a A-labeled vertex u such that v and u are incident to the same 1-labeled edge. Thus, for the validity of this invariant we only have to show that the set S is empty.

We assume further, that the set S is not empty. Let u be an element of S such that there exists an edge e satisfying $e = (v, u)$. Consider that at time t_1, v was N-labeled and u was A-labeled. This means that at time $t_0 < t_1$, v and u were both A-labeled and that the edge e was 1-labeled. This means that the execution of rule R2 on v at time t_0 was not allowed. Vertex v can therefore not be N-labeled at time t_1. This represents a contradiction to the assumption we made above. □

Lemma 3.9 *A N-labeled vertex v has no neighbor u such that $p(u) < p(v)$.*

Proof. Let v be a N-labeled vertex. All the edges incident to v are 0-labeled (Lemma 3.8). Let u be any neighbor of v. There are two interesting cases:

u is N-labeled: In this case, the execution of rule R2 (on u or on v) guarantees the fact that $p(v) = p(u)$.

u is A-labeled: This implies two possibilities:

- The execution of R2 on v guarantees that $p(v) = p(u)$.
- A vertex w was elected (rule R3) and u was reached by the pulse propagation rule R1. This means that $p(v) + 1 = p(u)$.

□

Lemma 3.10 *The subgraph G_T induced by all the A-labeled vertices is connected by all the 1-labeled edges and all the vertices of G_T are in the same pulse P_A.*

Proof. After the election phase (R3), there is exactly one A-labeled vertex in the neighborhood of the elected vertex v. Rule R3 also guarantees that all the neighbors u of v are in the same pulse $P_N = p(v) - 1$. We can easily generalize this fact and state that all the N-labeled vertices of the graph G are at this time in pulse $p(v) - 1$. This is deduced from successive executions of R2. After each execution of R2, two A-labeled vertices are incident to the same 1-labeled edge. Rule R1 is the only possibility to generate 1-labeled edges. Furthermore, each generated 1-labeled edge is incident to two A-labeled vertices. And because of Lemma 3.7, we know that there is no N-labeled vertex incident to a 1-labeled edge. Thus, the subgraph induced by all the A-labeled vertices is connected by all the 1-labeled vertices. Because of R1, we can also deduce from the above arguments that all the A-labeled vertices are always in the same pulse P_A. □

Lemma 3.11 *The subgraph induced by all the 1-labeled edges is a tree.*

Proof. Let G_T be the subgraph induced by all A-labeled vertices. Without loss of generality, we assume that there are three vertices u, v and w that belong to G_T and form a 1-labeled cycle C_T (see Figure 8). We further assume that at a time t_p, u was A-labeled while v and w were N-labeled. Thus, at t_p, all the edges of C_T were 0-labeled (Lemma 3.7). In order to change the label of the edges of C_T, rule R1 must, since t_p, have been executed on u, v and w. From the above argumentation, we deduce that u was N-labeled at time t_p. This represents a contradiction of the assumption we made about the label of u. □

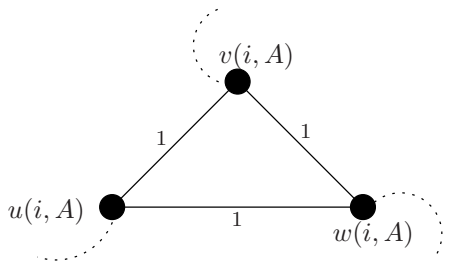


Figure 8: Cycle C_T in the A-labeled subgraph.

Proposition 3.3 *There exists a time $t_0 > 0$, where all the N-labeled vertices are in the same pulse $P_N > 0$.*

Proof. Let v_i be the elected vertex at the beginning of pulse i ($p(v_i) = i$). Let t_{i-1} be the time before the execution of rule R3 on v_i. At this time, there is only one A-labeled vertex left in the whole network (election principle), and all the N-labeled vertices are in the same pulse $p(v_i) - 1$ (*see the proof of Lemma 3.10). Thus,* $t_0 = t_{i-1}$. □

Our attention is now turned to the validity of Property 1.

Lemma 3.12 *Procedure 3.1 satisfies the conditions of Property 1.*

Proof. Let t_0 and v_i be defined as in the proof of Proposition 3.3. Let $u \neq v_i$ be the last A-labeled vertex that became N-labeled. It is clear that u belongs to the neighborhood of v_i. Let t_u be the time when R2 was executed on u. We know that $p(u) = p(v_i)$ at time t_u. Thus, all the N-labeled vertices of G were in pulse $p(u) = p(v_i)$ at time t_u. This implies that all the vertices of G were in pulse P_u at this time. □

We go further in our demonstration and prove the validity of Property 2.

Lemma 3.13 *Procedure 3.1 satisfies the conditions of Property 2.*

Proof. There are only two possibilities for a vertex v to change its pulse. The first one is by executing rule R3. This rule ensures that v is A-labeled. Due to Lemma 3.9 and Lemma 3.10, we know that the relation $p(v) \geq P_N$ yields. Where P_N is the pulse of all the N-labeled vertices of G.

The second possibility for a vertex v to change its pulse, is to execute R1. This rule is always executed after a vertex u has started the pulse $i + 1$. At the beginning of this pulse, all the N-labeled vertices are in pulse i (see Proposition 3.3). Because of Lemma 3.10, we know that all the A-labeled vertices are in pulse $i + 1$ and that all the N-labeled vertices are at least in pulse i. Thus, v can take for sure that there is no vertex w in G such that $p(w) < p(v)$. □

Theorem 3.2 (Correctness) *The synchronization protocol obtained by using the graph relabeling system $\mathcal{R}_3$ is correct.*

Proof. From Lemma 3.12 and Lemma 3.13 we can immediately deduce the correctness of the Property 3. □

Example: The β synchronizer [Tel00] is a particular case of the synchronizer $\mathcal{R}$. While in the synchronizer $\mathcal{R}$ the root is not known in advance and is determined by an election algorithm, the β synchronizer assumes a spanning tree with the same root. Before increasing the pulse, the root waits for "tree safe" messages from its sons, which wait in their turn for "tree messages" from their sons and so on. In other words, the root ensures that all vertices of the spanning tree are safe before going to the next pulse.

3.6 Synchronization in Trees

All the synchronization protocols developed so far were dedicated to any type of graphs. These protocols need adequate knowledge of some graph characteristics to be exact and faultless. Now we introduce a new methodology devoted to trees. Although this methodology is restricted to trees, it has the advantage that we do not need to have more knowledge about the network size, the network diameter or the existence of a distinguished vertex.
The main idea of this protocol resides in the use of an election algorithm to decide which

vertex should start the next pulse. The election algorithm [BMMS02] we used is the same as the one introduced in Section 3.5. This algorithm solves the problem of the election in anonymous trees using graph relabeling systems. At the beginning of the synchronization protocol, all vertices are in the same pulse. Their labels have two items. The first one is needed for the election algorithm and the second one represents the pulse number. We exploit the election algorithm to choose a vertex u that starts the next pulse. Furthermore, all the vertices v, that have a vertex w in their neighborhood such that $p(w) > p(v)$, increase their pulse. When a vertex increases its pulse, its *election*-label is set back to the initial value. A formalized description of this synchronization protocol is given below.

Let T be a tree. Initially all vertices are labeled $(N, 1)$. The graph relabeling system is $\mathcal{R}_4 = (L_4, I_4, P_4)$ defined by $L_4 = \{\{L, N\} \times [1..\infty]\}$, $I_4 = \{(N, 1)\}$, $P_4 = \{R1, R2, R3\}$ where $R1$, $R2$ and $R3$ are the relabeling rules given in Procedure 3.2.

Procedure 3.2 (protocol for tree-shaped networks)

R*1* : **Leaf elimination rule**

Precondition :

- $\lambda(v_0) = (label(v_0), p(v_0))$,
- $label(v_0) = N$,
- $\exists v \in B_T(v_0, 1)\ label(v) = N,\ p(v) = p(v_0)$,
- $\forall w \in B_T(v_0, 1)/v\ p(w) = p(v_0),\ label(w) = L$,

Relabeling :

- $\lambda'(v_0) := (L, p(v_0))$.

R*2* : **Tree election rule**

Precondition :

- $\lambda(v_0) = (label(v_0), p(v_0))$,
- $label(v_0) = N$,
- $\forall v \in B_T(v_0, 1)\ p(v) = p(v_0),\ label(v) = L$,

Relabeling :

- $\lambda'(v_0) := (N, p(v_0) + 1)$.

R*3* : **The propagation rule**

Precondition :

- $\lambda(v_0) = (label(v_0), p(v_0))$,
- $label(v_0) = L$,
- $\exists v \in B_T(v_0, 1)\ p(v) > p(v_0), label(v) = N$,

Relabeling :

– $\lambda'(v_0) := (N, p(v_0) + 1)$.

In order to prove the correctness of Procedure 3.2, we need to introduce some definitions that should simplify the proofs we will state later.

Definition 3.4

- *A vertex v is N-labeled if the second item of its label is N.*
- *A vertex v is L-labeled if the second item of its label is L.*
- *A pendant vertex v is a vertex that has exactly one N-labeled neighbor.*
- *A subgraph T_i is called N-labeled (respectively L-labeled) if all its vertices are N-labeled (respectively L-labeled).*

Before trying to demonstrate the validity of these lemma, we first have to state some invariants that will be helpful for the proofs we will started thereafter.
Let $T(V, E, \lambda)$ be a tree such that all its vertices are labeled $(N, 1)$. Let $T'(V, E, \lambda')$ be a tree such that: $T(V, E, \lambda) \xrightarrow[\mathcal{R}_4]{*} T'(V, E, \lambda')$.
The tree $T'(V, E, \lambda')$ satisfies the following properties:

Lemma 3.14 *A L-labeled vertex v has at most one neighbor w such that $p(w) \geq p(v)$ and w is N-labeled.*

Proof. We first show that a L-labeled vertex has at most one N-labeled neighbor.
Let v be any L-labeled vertex. Only the rule R1 can generate a L-labeled vertex. After the application of R1 on v, the precondition ensures that v has exactly one N-labeled vertex. Let w be the N-labeled vertex that belongs to the neighborhood of v. If w is elected, then there is no changes in the neighborhood of v. Otherwise, w will also become L-labeled. Thus, the only possibility that could increase the amount of N-labeled vertices in the neighborhood of v is the use of rule R3.
We can assume that the tree T can be decomposed in two subtrees T_1 and T_2 such that T_1 is L-labeled and T_2 contains the elected vertex (*see Figure 3.6*). After a vertex of T_2 has been elected, successive applications of R3 (propagation phase) will first change the label of w and v before transforming any vertex u of T_1 in a N-labeled vertex. This means that the maximal amount of N-labeled vertex in the neighborhood of a L-labeled vertex v is 1.
Let now v and w be defined as above. $p(v)$ and $p(w)$ are respectively the pulse of v and w. At the beginning, R1 ensures that $p(v) = p(w)$. From the above discussion, we derive that v is only able to change its label (pulse) after $p(w)$ has changed. Thus $0 \leq (p(w) - p(v)) \leq 1$. □

Lemma 3.15 *A L-labeled vertex v has no neighbor w such that w is N-labeled and $p(w) < p(v)$.*

Proof. The validity of this property can be derived from Lemma 3.14. □

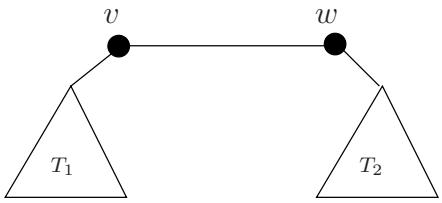


Figure 9: Representation of the propagation phase

Lemma 3.16 *The subgraph induced by all N-labeled vertices is connected.*

Proof. The aim of the used election algorithm is to "cut" the *pendant* vertices (rule R1). Let now consider all L-labeled vertices as vertices that have been removed from T. At the beginning of the election phase, the *pendant* vertices are the leafs of T.
Without loss of generality, let T_k be a tree. It is a well-known fact that if we remove (or add) some leafs of (to) T_k, the resulting graph will still be a tree. Thus, applications of R1 (remove) and R3 (add) on T will generate a subtree T' that is N-labeled. □

Lemma 3.17 *All N-labeled vertices are in the same pulse P_N.*

Proof. Let T_N be the subtree induced by all N-labeled vertices. Let now consider the T_N before and after the execution of R2. Previous to the execution of R2, there is one N-labeled vertex v in T_N (election main principle). The execution of R2 increase the pulse of v. In the propagation phase (rule R3), the pulse of all the new N-labeled vertices is set to $p(v)$. Hence, all the vertices of T_N are in the same pulse $P_N = p(v)$. □

Lemma 3.18 *All the vertices of a L-labeled subtree T_i are in the same pulse P_{T_i}.*

Proof. Let T be composed as described in Figure 10. The existence of the subtrees T_i is guaranteed by the fact that all the "cuts" of *pendant* vertices converge to the same N-labeled vertex v that will be elected. Thus, if T_0 is connected to a L-labeled vertex v_i, then v_i is the root of the L-labeled subtree T_i. Successive executions of rule R1 guarantee the fact that T_i is a connected set of vertices that are all in the same pulse $P_{T_i} = p(v_i)$. We have to notice that all the L-labeled vertices of T are not always in the same pulse. □

The following lemma is also of prime importance for the comprehension and the validity of our synchronization protocol.

Theorem 3.3 *At a time t_0, all L-labeled vertices are in the same pulse P_L.*

Proof. At a specific time t, the election algorithm will use rule R2 on a vertex v. Thereafter, v is elected. We now set $t_0 = t$. At time t_0, the tree T is in the configuration defined in Figure 10. In this configuration T_0 only contains one N-labeled vertex : v. Consider that the vertex

All $T_i (i = 1, 2, 3)$ are L-labeled, T_0 is N-labeled.

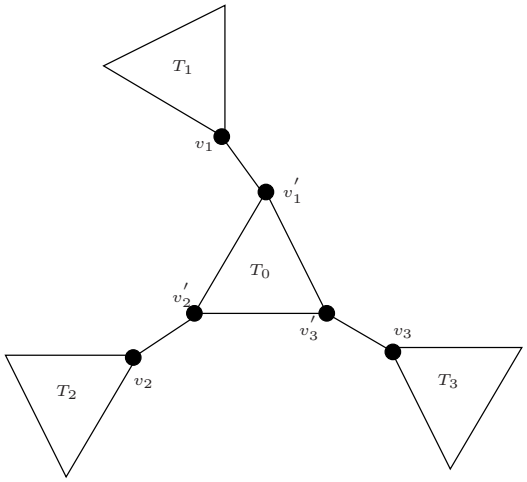


Figure 10: Decomposition of T

v has exactly three neighbors v_1, v_2 and v_3. Thus, we have the relation $v_1' = v_2' = v_3' = v$. From the preconditions of rule R1, we deduce that $p(v) = p(v_1) = p(v_2) = p(v_3)$. Because of *Lemma 3.18*, we know that all the vertices of the subtree T_i are in pulse $p(v_i)$. All L-labeled vertices are then in the same pulse $P_L = p(v)$. Hence, at the time t_0, all the vertices of T are in pulse P_L. □

Now, we can turn our attention to the proofs of Property 1 and Property 2.

Lemma 3.19
Fact 1: Procedure 3.2 satisfies the conditions of Property 1.
Fact 2: Procedure 3.2 satisfies the conditions of Property 2.

Proof.
Fact 1: It is obvious to see that Property 1 is a corollary of Theorem 3.3.
Fact 2: Only the rules R2 and R3 are able to allow a vertex v to change its pulse. The execution of R2 takes place under the terms of Theorem 3.3. Thus, all the vertices of T are in the same pulse.
The rule R3 is always applied after the time t_0, when rule R2 started a new pulse $i + 1$. Hence, all the N-labeled are in the same pulse $P_N = i + 1$ (*see Lemma 3.17*) and Theorem 3.3 ensures that all L-labeled vertices are at least in pulse $P_L = i$. This will suffice to state the validity of Property 2. □

Theorem 3.4 (Corectness) *The synchronization protocol obtained by using the graph relabeling system $\mathcal{R}_4$ is correct.*

Proof. From Lemma 3.19 we can immediately deduce the correctness of the Property 3. □

3.7 Building Synchronizers

The main goal of this section is to give a methodology, that should transform the protocols introduced in previous sections in operative synchronizers. All the developed protocols assume the existence of a *pulse generator* at each vertex of the network. This means, that a vertex v has a pulse variable $p(v)$, and it is supposed to generate a sequence of local *clock pulses* by increasing the value of $p(v)$ from time to time (i.e. $p(v) = 0, 1, 2, ...$). These pulses are supposed to simulate the ticks of the global clock in the synchronous setting. Obviously, the use of these protocols as stand-alone applications will give nothing in an asynchronous environment. For this reason, we introduce some definitions that should help us to ensure some guarantee about the relationship between the pulse values at neighboring vertices at various moments during the execution.

Definition 3.5 *We denote by $t(v, p)$ the physical time in which v has increased its pulse to p. We say that v is at pulse $p(v) = p$(or at pulse p) during the time interval $\tau(v, p) = [t(v, p), t(v, p + 1)]$.*

Since our network is fully asynchronous, we are not able to force all the vertices to maintain the same pulse at all times. However, we know that it is possible to guarantee a weaker form of compatibility between the pulses of neighboring vertices in the network. This form of compatibility is stated in definition 3.6.

Definition 3.6 (Message pulse compatibility) *If a vertex v sends an original message m to a neighbor w during its pulse $p(v) = p$, then m is received at w during its pulse $p(w) = p$ as well.*

3.7.1 Methodology to construct a synchronizer

To build a functioning synchronizer from each of our protocols, we have to change their specifications such that, given an algorithm Π_S written for a synchronous network and a protocol ν, it should be possible to *combine* Π_S on top of ν to yield a protocol $\Pi_A = \nu(\Pi_S)$ that can be executed on an asynchronous network. Π_A has two components: The *original component* and the *synchronization component*. Each of these components has its own local variables and messages at every vertex. The *original component* consists of the local variables and the messages of the original protocol Π_S, whereas the *synchronization component* consists of local synchronization variables and synchronization messages.

As from now, we are going to show that the changes needed to build a synchronizer from any of our protocols can be done effortless. Further, we will show that it is possible to make use of these changes to prove the correctness of the so constructed synchronizers. This proof will be done with respect to the definition given by David Peleg [Pel00].

Conceptually, the modifications of our protocols are done in two steps. The first step affects the label attached to each vertex. In fact, we add three new items to each label. The first item consists of a *boolean* variable S. This variable decides which component(*original component* or *synchronization component*) is *active* on each vertex. Only the rules of the *active* component can be applied on a vertex v. The other items are two buffers B_p and B_{p-1} that represent the messages that a vertex v, at pulse $p(v)$, has to send respectively in pulses $p(v)$ and $p(v) - 1$. We claim that a protocol μ that is generated from the above modifications

satisfies all the specifications of a synchronizer. In a general way, a protocol $\Pi_A = \mu(\Pi_S)$ will work in two phases.

Initially, the item S_v is set to *true* for all vertices v and therefore the *synchronization component* is active (Phase one). As soon as a vertex v increases its pulse $p(v)$, its variable S_v is set to $false$. Hence, only rules from Π_S can be applied on v. The *original component* is active (phase two). Each application of a rule of Π_S sets back the state of S_v to *true*. Thus the synchronization component can begin to synchronize v with all the other vertices. This cycle is done until Π_S stops Π_A with any specific request.

3.7.2 A General Overview

Both phases depicted in the previous section are summarized in Figure 11. The rule $\mathcal{R}_1$ represents phase one and $\mathcal{R}_2$ represents phase two. Δ describes the set of rules used in the synchronization protocol(ν) and Ω is the set of relabeling rules used in the algorithm Π_S. The first rule means that as long as $S_v = True$ holds, it is only allowed to synchronize v with all the vertices in the network. As soon as v changes its pulse, the second phase is started for v. In this phase, it is no more allowed to synchronize but it is permitted to execute rules of the algorithm Π_S. After the execution of Π_S, rule one is active once again, and the cycle goes on. All our synchronizers will have to use a rule that plays the role of $\mathcal{R}_2$. This rule should ensure a good interaction between the two protocols ν and Π_S. Now, it is time to give an example

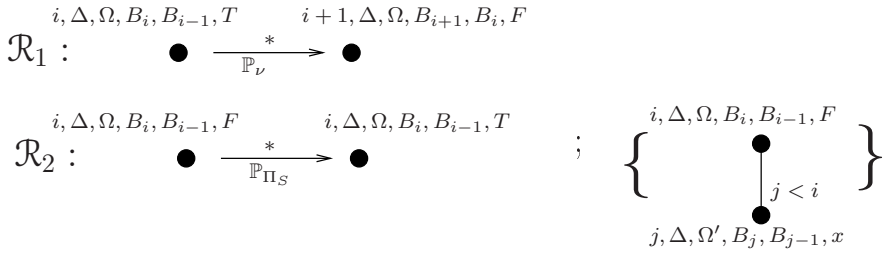


Figure 11: General method to construct a synchronizer.

to illustrate how the described construction can be achieved in a realistic way.

Example: Synchronizer using SSP. Let G be a graph with diameter $d = \Delta(G)$. Consider a labeling function λ, where $\lambda^{(V)} : V \to [1..\infty] \times [-1..d+1] \times \mathcal{M} \times \mathcal{M} \times \{T, F\} \cup \mathbb{L}(\Pi_S)$. Initially all vertices are labeled $(1, -1, B_1, B_0, F, \mathbb{I}_{\Pi_S})$. The graph relabeling system is $\mathcal{R} = (L_1, I_1, P_1)$ defined by $L_1 = \{[1..\infty] \times [-1..d+1] \times \mathcal{M} \times \mathcal{M} \times \{T, F\}\} \cup \mathbb{L}(\Pi_S)$, $I_1 = \{(1, -1, B_1, B_0, F, \mathbb{I}_{\Pi_S})\}$, $P_1 = \{R1, R2, R3, R4\} \cup \mathbb{P}_{\Pi_S}$ where $\mathbb{L}(\Pi_S)$ represents the type of label used in Π_S, $\mathbb{I}_{\Pi_S}$ is the set of initial states in Π_S, $\mathbb{P}_{\Pi_S}$ is the set of rules used in Π_S, $R1$, $R2$, $R3$ and $R4$ are the relabeling rules with forbidden contexts given in Figure 12. The specifications and the properties of the underlying algorithm of this protocol are discussed in Section 3.3.

The rule $R4$ is of extreme importance in this protocol. $R4$ plays the role devoted to $R2$ in Figure 11. It allows the application of exactly one action of the algorithm Π_S on a vertex v. After such an action, the vertex v enters the synchronization phase. $R4$ can be generalized to all the synchronizers we could construct using the protocols introduced in earlier sections.

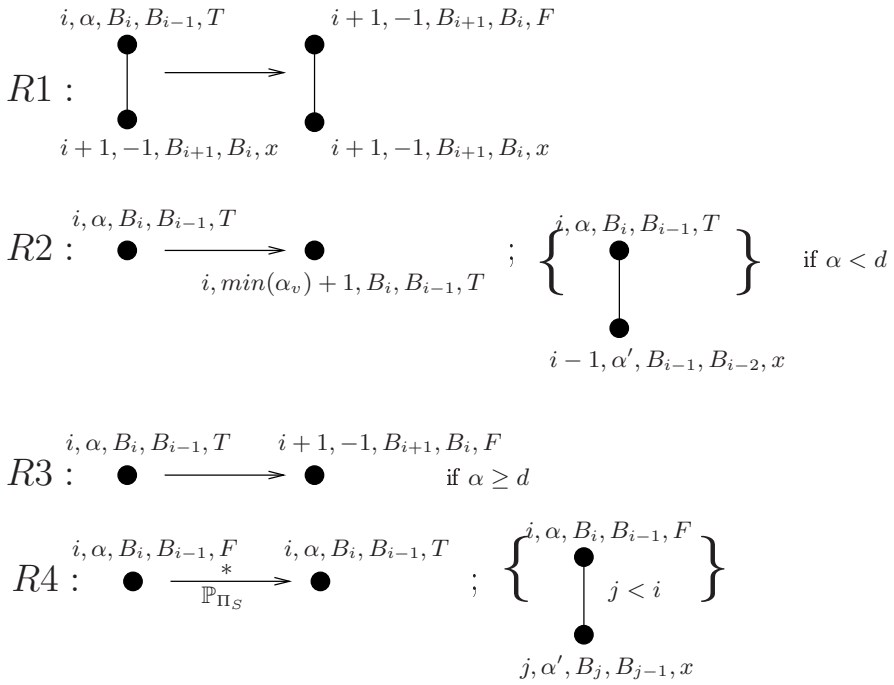


Figure 12: Protocol Π_A using the SSP method.

Correctness of the synchronizers

We now introduce some meaningful properties that can all be guaranteed from the synchronizers we can build.

Definition 3.7 (Readiness)

Pulse readiness: *A vertex v is ready for pulse p, denoted $Ready(v,p)$, once it has already received all messages of the algorithm sent to it by its neighbors during their pulse number $p-1$.*

Readiness rule: *A vertex v is allowed to generate its pulse p once it is finished with its required original actions for pulse $p-1$ and $Ready(v,p)$ holds.*

Definition 3.8 (Delay rule) *If a vertex v receives in pulse p a message sent to it from a neighbor w during some later pulse $p' > p$ of w, then v declines consuming it and temporarily stores it in a buffer. It is allowed to process it only once it has already generated its pulse p'.*

Claim 3.2 *Let μ be a synchronizer built as described in section 3.7.1. μ always satisfies the*

1. *Readiness rule and the*
2. *Delay rule*

Proof.

1. The proof is an immediate consequence of the use of rule $R2$ (see Figure 11). A vertex v is allowed to generate its pulse p, if and only if $S_v = T$ and v satisfies the conditions required from the *synchronization protocol*. The only possibility for S_v to become $True$ is the execution of rule $R2$. Thus, v has executed its required Π_S-actions for pulse $p-1$ and $Ready(v,p)$ holds.

2. Let v be a vertex that has received in pulse p a message m' sent to it from a neighbor w during pulse $p' > p$. All our synchronization protocols guaranteed that after a finite number of steps, v and w will be in the same pulse p'. On the other hand, v can only consume the messages contained in the buffer B_p^w. Such a buffer always exists. Indeed, the pulse difference between two vertices in the whole network is maximal 1. This means that v will be able to consume m' as soon as $p(v) = p'$ holds.

□

Lemma 3.20 *[Pel00] A synchronizer imposing the readiness and delay rules guarantees pulse compatibility.*

The above lemma states easily the reasons why all our synchronizers guaranteed the principle of pulse compatibility introduced in Definition 3.6. Peleg show in the year 2000 an essential relationship between the concept of pulse compatibility and the correctness of a synchronizer. One of the interesting parts of his work was announced as the lemma below .

Lemma 3.21 *[Pel00] If synchronizer μ guarantees pulse compatibility, then it is correct.*

Theorem 3.5 *Let μ be a synchronizer built as described in section 3.7.1. μ is correct.*

Proof. The proof is deduced from Claim 3.2 and according to Lemma 3.21 and Lemma 3.20. □

Second Part

Algorithmic Recognition of Graphs Properties with Local Computations

A reduction algorithm is based on a finite set of *reduction rules* and a finite set of graphs. Each reduction rule describes a way to modify a graph locally. The original idea of a reduction algorithm is to solve a decision problem by repeatedly applying reduction rules on the input graph until no more rule can be applied. If the resulting graph is in the finite set of graphs, then the algorithm returns *true*, otherwise it returns *false*. Up to now, reduction algorithms are studied under two different computation models. While the first model deals with a sequential execution of reduction rules, the second model is concerned with the parallel execution of the corresponding graph transformation operations. In this part, we introduce a third approach that consists in executing the reduction rules in a fully distributed and asynchronous system. We make use of this algorithm to develop an algorithmic approach for solving the labeled graph recognition problem with local computations.

In contrast to the distributed algorithms presented in the previous chapter, the computation of reduction algorithms depends on our capability to solve the *k-local election problem* ($k \geq 3$) in the local computations framework. Thus, the first chapter of this part will be devoted to the presentation of a distributed algorithm that with high probability solves the k-local election problem for arbitrary k. This algorithm is based on distributed computations of rooted trees of minimal paths. Its correctness also depends on the use of local heuristics that try to avoid random numbers collisions, where each random number represents the identifier of a root vertex. The results concerning the performance and the analysis of this algorithm appear in the Proceeding of the Fourth International Workshop on Experimental and Efficient Algorithms [Oss05a].

At the beginning of the second chapter of this part, we will give a formal definition of reduction systems and introduce the notion of *handy reduction systems* that states necessary conditions for a reduction algorithm to be computed in a distributed environment. Thereafter, we will present a distributed protocol that makes use of reduction rules to solve *graph decision problems*. Using these results we will show that all decision problems (or graph properties) for graphs of bounded treewidth can be solved by local computations. From these results, we will state a relationship between the use of handy reduction systems, for solving decision problems, and labeled graphs recognizers introduced by Métivier et al. in [GMM04]. At the end of this part, we will introduce the concept of *unfolding reduction rules* that will help us to solve constructive reduction algorithms by means of local computations.

The last chapter of this part is concerned with the detection of properties during the execution of a distributed algorithm. The main goal is, given a distributed algorithm encoded by local computations $\mathcal{A}_l$ and a set of properties related to the different network components, to be able to check the validity of these properties during the execution of $\mathcal{A}_l$. We will present a virtual time based algorithm which is able to perform an *online* or *offline* check of local and global properties. This algorithm represents a tool that is needed to test the correctness of programs executions in the Lidia environment (see Chapter 7). All the results stated in this chapter have been published in [MO04a].

Chapter 4

A Probabilistic Algorithm for Local Elections

Contents

4.1 The k-local Election Problem

The problem of election is linked to distributed computations in a network. It aims to choose a unique vertex, called leader, which subsequently is used to make decisions or to centralize some information.

For a fixed given positive integer k, a k-local election problem requires that, starting from a configuration where all processes are in the same state, the network reaches a configuration $\mathcal{C}$ such that for this configuration there exists a non empty set of vertices, denoted $\mathcal{E}$, satisfying:

- each vertex $v \in \mathcal{E}$ is in a special state called *leader* and
- $\forall v \in \mathcal{C}$ and for all vertex $w \neq v$ such that $d(v, w) \leq k$ then w is in the state *lost* (i.e. $w \notin \mathcal{E}$).

We assume that each process has the same local algorithm. This problem is then considered under the following assumptions:

- the network is anonymous: unique identities are not available to distinguish the processes,

- the system is asynchronous,
- processes communicate by asynchronous message passing: there is no fixed upper bound on how long it takes for a message to be delivered,
- each process knows from which channel it receives a message.

In the local computations framework, we generally deal with two kinds of local computations:

LC_1: in a computation step, the label attached to the center of a ball of radius 1 is modified according to some rules depending on the labels in the ball. The labels of the other vertices in the ball are not modified.

LC_2: in a computation step, labels attached to the vertices in a ball of radius 1 may be modified according to some rules depending on the labels in the ball.

The implementation of distributed algorithms encoded by means of these local computations [BGM+01] can be performed using $k-$local elections ($k \leq 2$). Once a vertex v is locally elected a local computation can be applied on the ball centered on v. This ensures a faithful relabeling of disjoint subgraphs of radius 1. The main motivation of this work is to extend the implementation of local computations to balls of radius more than 1. Therefore, it is necessary to solve the k-local election problem ($k \geq 3$) such that all the elected vertices should be able to execute graph relabeling steps on disjoint subgraphs of radius $\frac{k}{2}$. From the work of Angluin [Ang80], we deduce the following facts.

Fact 4.1 *[Ang80] There is no deterministic algorithm for solving the k-local election problem for $k \geq 1$.*

Fact 4.2 *[Ang80] There is no Las Vegas algorithm for solving the k-local election problem for $k \geq 3$.*

Based on distributed computations of rooted trees of minimal paths, we present in this chapter a simple randomized algorithm which, with very high probability, solves the k-local election problem ($k \geq 2$) under the above assumptions. For the sake of time complexity, we will assume that each message incurs a delay of at most one unit of time [AP90]. Note that the delay assumption is only used to *estimate* the performance of our algorithms. This does not imply that our model is synchronous, neither does it affect the correctness of our algorithms. That is, our algorithms work correctly even in the absence of this delay assumption.

4.2 Randomized Local Elections

Many problems have no solution in distributed computing [Lyn89]. The introduction of randomization makes possible tasks that admit no deterministic solutions. General considerations about randomized distributed algorithms may be found in [Tel00]. We present in this subsection the two randomized procedures: RL_1 and RL_2 that respectively solve the 1 and 2-local election problem. The introduction and the study of these procedures are due to Métivier et al. [MSZ02]. Let K be a nonempty set equipped with a total order.

Randomized 1-Local Election

RL_1**:** Each vertex v repeats the following actions.

- v selects an integer $rand(v)$ randomly and uniformly from the set K.
- v sends $rand(v)$ to its neighbors.
- v receives the numbers sent by all its neighbors.
- v wins the 1-local election if for each neighbor w of v: $rand(v) > rand(w)$.

Randomized 2-Local Election

RL_2**:** Each vertex v repeats the following actions.

- v selects an integer $rand(v)$ randomly and uniformly from the set K.
- v sends $rand(v)$ to its neighbors.
- v receives messages from all its neighbors. Let Int_w be the maximum of the set of integers that v has received from vertices different from w.
- For all neighbors w, v sends Int_w to w.
- v receives integers from all its neighbors.
- v wins the 2−local election in $B(v, 2)$ if $rand(v)$ is strictly greater than all integers received by v.

Proposition 4.1 *Let $G = (V, E)$ be a connected graph. After the execution of a randomized 2-local election there are at most $\frac{|V|}{2}$ vertices v of G that have won the 2-local election.*

Proof. Let X_2 be the number of vertices that have won the 2-local election. Each time a vertex v wins this local election, at least one vertex w in $V(G)$ (with $|V| > 1$) loses the same election. This means that $X_2 + 1 \times X_2 \leq |V|$. Thus $X_2 \leq \frac{|V|}{2}$. □

4.3 Distributed Computation of a Rooted Tree of Minimal Paths

Let $G = (V, E)$ be an anonymous network with a distinguished vertex u_0. The problem considered here is to find a tree of (G, V), rooted at u_0, which for any $v \in V$ contains a unique minimal path from v to u_0. This kind of tree is generally used to pass a signal along the shortest path from v to u_0 (see Moore [Moo59]).

To solve the above problem, we can simply fan out from u_0, labeling each vertex with a number which counts its distance from u_0, modulo 3. Thus, u_0 is labeled 0, all unlabeled neighbors of u_0 are labeled 1, etc. More generally, at the tth step, where $t = 3m + q, m \in \mathbb{N}, q \in \{0, 1, 2\}$, we label all unlabeled neighbors of labeled vertices with q and we mark the corresponding edges. When no more vertices can be labeled, the algorithm is terminated. It is then quite simple to show that the set of marked edges represents a tree of minimal

paths rooted at u_0. In an asynchronous distributed system, where communication is due to a message passing system, we do not have any kind of centralized coordination. Thus, it is not easy for a labeled vertex to find out that all labeled vertices are in the same step t. To get around this problem, we have slightly modified and adapted the above procedure for distributed systems.

The Algorithm.
Our algorithm works in rounds. All the vertices have knowledge about the computation round in which they are involved, they also have a state token that indicates if they are *locked* or *unlocked*. At the end of round i, all vertices $v \in \{w \in V | d(w, u_0) \leq i\}$ are labeled and *locked*. Initially, all vertices $v \neq u_0$ are *unlocked* and unlabeled, all edges are unmarked and u_0 is labeled 0. At round 1, all unlabeled neighbors of u_0 are labeled 1 and *locked*. A vertex w is said to be *marked* for a vertex v if the edge $e = \{w, v\}$ is marked. For further computations the algorithm has to satisfy the following requirements:

r_1**:** Each time an unlabeled vertex is labeled, it is set in the *locked* state.

r_2**:** A labeled *unlocked* vertex $v \neq u_0$ becomes *locked* if:

- v is in the same round as all its *marked* neighbors,
- v does not have any unlabeled vertex in its neighborhood,
- All the *marked* neighbors w of v that satisfy $d(u_0, w) = d(u_0, v) + 1$ are *locked*.

r_3**:** A *locked* vertex v in round p becomes *unlocked* and increases its round, if it has an *unlocked* *marked* neighbor w in round $p + 1$.

We encode this procedure by means of a graph relabeling system where the *locked* and *unlocked* states are respectively represented by the labels L and U. The root is the only distinguished vertex labeled with R. Marked edges are labeled with 1. The set of labels is $L = \{0, 1, (x, d, r)\}$ with $x \in \{N, L, U, R\}$ and $d, r \in \mathbb{N}$. d and r respectively represent the distance from the root vertex (modulo 3) and the computation round of a given vertex. The initial label on the root vertex u_0 is $(R, 0, 1)$ and all the other vertices have the label $l_0 = (N, 0, 0)$. All the edges have initially the label 0. The graph relabeling system is $\mathcal{R}_1 = (L, I, P)$ with $P = \{R1, R2, R3, R4\}$ where $R1$, $R2$, $R3$ and $R4$ are the relabeling rules given in Procedure 4.1 . The rooted tree computation is described by the next procedure.

Procedure 4.1 *[Computing a rooted tree of minimal paths]*

R1 : **Initializing the first level**

Precondition :

- $\lambda(v_0) = (R, d, r)$,
- $\exists v \in B(v_0, 1)(v \neq v_0 \wedge \lambda(v) = (N, 0, 0))$.

Relabeling :

- $\lambda'([v_0, v]) := 1$,
- $\lambda'(v) := (L, (d+1) \ modulo \ 3, r)$.

R2 : **Unlock the first level (part 1)**

Precondition :

- $\lambda(v_0) = (R, d, r)$,
- $\forall v \in B(v_0, 1)(v \neq v_0 \wedge \lambda(v) = (L, (d+1) \ modulo \ 3, r) \wedge \lambda(\{v_0, v\}) = 1)$.

Relabeling :

- $\lambda'(v_0) := (R, d, r+1)$,

R3 : **Unlock the first level (part 2)**

Precondition :

- $\lambda(v_0) = (R, d, r)$,
- $\exists v \in B(v_0, 1)(v \neq v_0 \wedge \lambda(v) = (L, (d+1) \ modulo \ 3, r-1) \wedge \lambda(\{v_0, v\}) = 1)$.

Relabeling :

- $\lambda'(v) := (U, (d+1) \ modulo \ 3, r)$.

R4 : **Unlock the remaining levels**

Precondition :

- $\lambda(v_0) = (U, d, r)$,
- $\exists v \in B(v_0, 1)(v \neq v_0 \wedge \lambda(v) = (L, (d+1) \ modulo \ 3, r-1) \wedge \lambda(\{v_0, v\}) = 1)$.

Relabeling :

- $\lambda'(v) := (U, (d+1) \ modulo \ 3, r)$.

R5 : **Add new leaves to the tree**

Precondition :

- $\lambda(v_0) = (U, d, r)$,
- $\exists v \in B(v_0, 1)(v \neq v_0 \wedge \lambda(v) = (N, 0, 0))$.

Relabeling :

- $\lambda'(v) := (L, (d+1) \ modulo \ 3, r)$,
- $\lambda'([v_0, v]) := 1$.

R6 : **Lock internal vertices of the tree**

Precondition :

- $\lambda(v_0) = (U, d, r)$,
- $\forall v \in B(v_0, 1)(v \neq v_0 \wedge \lambda(v) \neq (N, 0, 0))$,
- $\forall w \in B(v_0, 1)(w \neq v_0 \wedge \lambda(\{v_0, w\}) = 1 \Rightarrow (\lambda(v) = (L, (d+1)\ \textit{modulo}\ 3, r) \vee \lambda(v) = (U, (d-1)\ \textit{modulo}\ 3, r) \vee \lambda(v) = (R, (d-1)\ \textit{modulo}\ 3, r)))$.

Relabeling :

- $\lambda'(v_0) := (L, d, r)$.

Lemma 4.1 *Let $D = \Delta(G)$ be the diameter of the graph G. At the end of round i $1 \leq i \leq D$, all the $1-$labeled edges build a rooted tree $\mathcal{T}_{u_0}$ that contains the root u_0 and all $L-$labeled vertices. $\mathcal{T}_{u_0}$ has therefore a depth of i.*

Proof. We show this lemma by induction on i. We recall that initially all vertices different from u_0 are labeled with $(N, 0, 0)$. During the execution of round $i = 1$, only rule $R1$ can be executed. This round ends with the execution of rule $R2$. Thus, only the neighbor vertices of u_0 are L-labeled and they build (with u_0) a tree $\mathcal{T}^1_{u_0}$ of minimal paths rooted at u_0. $\mathcal{T}^1_{u_0}$ has therefore depth 1. Let $\mathcal{T}^i_{u_0}$ be the constructed tree after round i. By the induction hypothesis we know that all vertices $v \neq u_0$ of $\mathcal{T}^i_{u_0}$ are L-labeled. During the computation of round $i+1$, all the L-labeled are first unlocked (see rules $R3$ and $R4$). Thereafter, rule $R5$ increases the rooted tree by adding new L-marked vertices (at most one new vertex per leave) to $\mathcal{T}^i_{u_0}$. At the end of round $i+1$, all U-labeled vertices are *locked* through rule $R6$. From the preconditions and the effects of the rules $R5$ and $R6$ we can deduce that $\mathcal{T}^{i+1}_{u_0}$ is a minimal path tree rooted at u_0 and having depth $i+1$. □

Corollary 4.1 *Let $v \in V$ be a $L-$labeled vertex and $p = \{v, v_0, v_1 ..., v_j, u_0\}$ a path from v to u_0 such that all v_i $(i \leq j)$ are $L-$labeled and $v_i \neq u_0$, $v_i \neq v_k$, $\forall i, k \leq j$. Then p is a minimal path from v to u_0.*

Adding two adequate relabeling rules makes it possible to generate rooted trees of minimal paths having a depth k such that $1 \leq k \leq D$, $k \in \mathbb{N}$. Such an improvement is presented in [Oss05a].

Lemma 4.2 *Let $\mathcal{T}^d_{u_0}$ be a rooted tree of depth d. The time complexity of constructing $\mathcal{T}^d_{u_0}$ is $O(d^2)$ and the message complexity is $O(|E| + n * d)$.*

Proof. Let $\mathcal{T}^i_{u_0}$ represents the tree of minimal paths, of depth i and rooted at vertex u_0. We recall that all vertices of $\mathcal{T}^i_{u_0}$ must be *locked* and *unlocked* for the computation of $\mathcal{T}^{i+1}_{u_0}$. Thus, the worst case time and message complexity for computing one path rooted at u_0 of tree $\mathcal{T}^d_{u_0}$ is $\sum_{i=1}^{d} i = O(d^2), 1 \leq d \leq \mathcal{D}$, with $\mathcal{D}$ representing the diameter of G. Thus, the message complexity for computing $\mathcal{T}^d_{u_0}$ starting from G is $O(|E| + n * d)$ [Lyn96]. All the vertices v that satisfy the rules $R2$ and $R3$ can change their labels simultaneously. That is, we assume that a vertex at depth i sends messages to its neighbors at depth $i+1$ simultaneously. The same fact is also true for the rules $R4$ and $R5$. For these reasons, we need $2(d+1)$ time units

to construct the tree $\mathcal{T}_{u_0}^{d+1}$ from $\mathcal{T}_{u_0}^{d}$. Thus, the time complexity of our procedure is given by

$$\sum_{i=1}^{d} 2 * (i+1) = d(d+1) + 2d = O(d^2).$$

□

4.4 Solving the k-Local Election Problem

Starting from the rooted tree procedure described in Section 4.3, we intend to design a simple procedure that should be able to solve the k-local election ($k \geq 2$) in an anonymous network. Let I_u be the identity of a vertex u and $(\mathcal{S}, >)$ be a structure model of tuples of the form (x_1, x_2) where x_1 and x_2 are real numbers and $(x_1, x_2) > (x_3, x_4) \Leftrightarrow (x_1 > x_3) \vee (x_1 = x_3) \wedge (x_2 > x_4)$. Basically, we have developed an algorithm that works in three steps.

Procedure 4.2 (Solving the k-Local Election Problem)

Step 1: *Each vertex u chooses a random number r_u and takes advantage of its tuple $n_u = (t_u, I_u) \in \mathcal{S}$ to perform a $2-$local election (RL_2). The winners and losers of these elections are respectively marked with W and L.*

Step 2: *Each $W-$marked vertex u starts the construction of the tree T_u^d (with depth d) of minimal paths rooted at u.*

Step 3: *Once T_u^d is constructed for a given vertex u, the tuples of all the $W-$marked vertices in T_u^d are compared to n_u. If $n_u > n_v, \forall v \in T_u^d, v \neq u$ (v is W-marked) then u has won the local election in the ball of radius d centered on u.*

The use of minimal path trees ensures that the tree T_u^d contains all vertices of the set $\{v \in V(G) | d(u,v) \leq d\}$. We remind that Procedure 4.1 was designed for a single root. Thus, all labels were related to the computation of the same tree. In Procedure 4.2, several trees have to be computed in a distributed way. For this reason, the label of each vertex v, in the new procedure, includes a set $\mathcal{L}_v$ of tuples representing the different states of v in the computations of all the rooted minimal path trees that contain v. Furthermore, to encode the marking of edges and to distinguish the different elements of $\mathcal{L}_v$, we relax one specification of our model and require that each vertex has a unique identity. The label of each vertex v also indicates if v has won the RL_2 procedure. Moreover, it includes an item that represents the label of v during the computation of the tree of minimal paths rooted at v.

The use of identities is certainly a weak point of our algorithm. Nevertheless, we will see that without identities, our methodology solves the $k-$local election with very high probability. The structure $(\mathcal{S}, >)$ ensures that at least one vertex terminates the election as winner. We are now ready to present the basics of our procedure for solving the $k-$local election problem ($k \geq 2$).

Definition 4.1 *A tuple structure model $(\mathcal{T}, <)$ is an irreflexive total ordering of tuples $t_w = (I_v, s_w, m_w, r_v^w, M_w, \mathcal{F}_w^v), \forall w \in V$ where:*

- *I_v is the identity of a root node v,*
- *s_w is an element of the set $\{R, U, L\}$,*
- *$m_w = d(v, w)$ modulo 3,*
- *$0 \leq r_v^w = d(v, w) \leq k$ is the round of vertex w in the computation of the minimal path tree rooted at v,*
- *M_w is the maximal tuple (on the minimal path between v and w) known by w so far,*
- *$\mathcal{F}_w^v$ is the identity of the father of vertex w in the minimal paths tree rooted at v.*

Definition 4.2 *For all $u, v, i, j \in V$, let t_i and t_j be two elements of $(\mathcal{T}, <)$. Then*

- *$t_i = t_j^+$ if and only if $I_v = I_u$, $(m_j + 1)\%3 = m_i$, $r_u^j = r_v^i$, $\mathcal{F}_i^v = I_j$, $s_i = L$ if $s_j = U$ and $s_i = L$ if $s_j = R$. We say $t_j < t_i$.*
- *$t_i = t_j^-$ if and only if $I_v = I_u$, $(m_j - 1)$ modulo $3 = m_i$ and $r_v^i = r_u^j$, $\mathcal{F}_j^u = I_i$, $s_i = U$ if $s_j = L$ and $s_i = R$ if $s_j = L$. We say $t_i < t_j$.*

Let G be a graph, for each vertex v we assume that $\lambda(v) = (\mathcal{I}_v, \mathcal{S}_v, \mathcal{L}_v, t_v, \mathcal{E}_v)$ where:

- $\mathcal{I}_v$ is the identity of the vertex v,
- $\mathcal{S}_v$ is an item that indicates the state of vertex v (L or W). Initially all vertices are in the L state.
- $\mathcal{L}_v$ is a set of elements of $(\mathcal{T}, <)$,
- t_v is the label of vertex v in the tree of minimal paths rooted at v,
- $\mathcal{E}_v$ is an item that indicates if v has won the $k-$local election (*elected*, *non_elected*).

Initially $\lambda(v) = (\mathcal{I}_v^0, \mathcal{S}_v^0, \mathcal{L}_v^0, t_v^0)$ for all vertices v with $\mathcal{I}_v^0 = \mathcal{I}_v$, $\mathcal{S}_v^0 \in \{L, W\}$, $\mathcal{L}_v^0 = \emptyset$ and $t_v^0 = (I_v, R, 0, 1, n_v, I_v)$. All vertices are in the state *non_elected*. Procedure 4.2 (Steps 2 and 3) is then computed by the relabeling rules given in Procedure 4.3.

Procedure 4.3 (Solving the k-local election)

R1 : **Initializing the first level**

Precondition :
- $\lambda(v_0) = (\mathcal{I}_{v_0}, \mathcal{S}_{v_0}, \mathcal{L}_{v_0}, t_{v_0}) \wedge \mathcal{S}_{v_0} = W$,
- $\exists v \in B(v_0, 1)(v \neq v_0 \wedge \forall t \in \mathcal{L}_v(t = (I_w, x, m_v, r_w^v, M_{v_0}, \mathcal{F}_v^w) \wedge I_w \neq I_{v_0} \wedge x \in \{R, L, U\}))$.

Relabeling :
- $\lambda'(v) := (\mathcal{I}_v, \mathcal{S}_v, \mathcal{L}_v + t_{v_0}^+, t_v, \mathcal{E}_v)$.

R2 : **Unlock the first level (part 1)**

Precondition :

- $\mathcal{S}_{v_0} = W \wedge t_{v_0} = (I_{v_0}, R, m_{v_0}, r_{v_0}^{v_0}, M_{v_0}, \mathcal{F}_{v_0}^{v_0})$,
- $\forall v \in B(v_0, 1)(v \neq v_0 \wedge t_{v_0}^{+} \in \mathcal{L}_v)$.

Relabeling :

- $t'_{v_0} := (I_{v_0}, R, m_{v_0}, r_{v_0}^{v_0} + 1, M_{v_0}, \mathcal{F}_{v_0}^{v_0})$,
- $\lambda'(v_0) := (\mathcal{I}_{v_0}, \mathcal{S}_{v_0}, \mathcal{L}_{v_0}, t'_{v_0}, \mathcal{E}_{v_0})$.

R3 : **Unlock the first level (part 2)**

Precondition :

- $\mathcal{S}_{v_0} = W \wedge t_{v_0} = (I_{v_0}, R, m_{v_0}, r_{v_0}^{v_0}, M_{v_0}, \mathcal{F}_{v_0}^{v_0})$,
- $\exists v \in B(v_0, 1)(v \neq v_0 \wedge \exists t_i \in \mathcal{L}_v(t_i = (I_{v_0}, L, (m_{v_0} + 1)\ \mathit{modulo}\ 3, r_{v_0}^{v_0} - 1, M_v, \mathcal{F}_v^{v_0})))$.

Relabeling :

- $\mathcal{L}_v := \mathcal{L}_v - t_i$,
- $t := (I_{v_0}, L, (m_{v_0} + 1)\%3, r_{v_0}^{v_0} + 1, M_v, \mathcal{F}_v^{v_0})$,
- $\lambda'(v) := (\mathcal{I}_v, \mathcal{S}_v, \mathcal{L}_v + t, t_v, \mathcal{E}_v)$,

R4 : **Unlock the remaining levels**

Precondition :

- $\exists v \in B(v_0, 1)(v \neq v_0 \wedge \exists t_i \in \mathcal{L}_v(\exists t_j \in \mathcal{L}_{v_0}(t_i = (I_p, L, m_v, r_p^v, M_v, \mathcal{F}_v^p) \wedge t_j = (I_p, U, (m_v - 1)\%3, r_p^v + 1, M_{v_0}, \mathcal{F}_{v_0}^p))))$.

Relabeling :

- $\lambda'(v) := (\mathcal{I}_v, \mathcal{S}_v, \mathcal{L}_v - t_i, t_v, \mathcal{E}_v)$,
- $t_i := (I_p, U, m_v, r_p^v + 1, M_v, \mathcal{F}_v^p)$,
- $\lambda'(v) := (\mathcal{I}_v, \mathcal{S}_v, \mathcal{L}_v + t_i, t_v, \mathcal{E}_v)$.

R5 : **Add new leaves to the tree**

Precondition :

- $\exists v \in B(v_0, 1)(v \neq v_0 \wedge \exists t_i \in \mathcal{L}_{v_0}(t_i = (I_p, U, m_{v_0}, r_p^{v_0}, M_{v_0}, \mathcal{F}_{v_0}^p)))$,
- $\nexists t \in \mathcal{L}_v(t = (I_q, x, m_v, r_q^v, M_v, \mathcal{F}_v^q) \wedge I_p = I_q) \wedge x \in \{R, L, U\}$.

Relabeling :

- $M_v^* := max(M_{v_0}, n_v)$,
- $t_i := (I_p, U, m_{v_0}, r_p^{v_0}, M_v^*, \mathcal{F}_{v_0}^p)$,
- $\lambda'(v) := (\mathcal{I}_v, \mathcal{S}_v, \mathcal{L}_v + t_i^{+}, t_v, \mathcal{E}_v)$.

R6 : **Lock internal vertices of the tree**

Precondition :

- $\exists t_i \in \mathcal{L}_{v_0}(t_i = (I_p, U, m_{v_0}, r_p^{v_0}, M_{v_0}, \mathcal{F}_{v_0}^p))$,
- $\forall v \in B(v_0, 1)(v \neq v_0 \wedge \exists t_j \in \mathcal{L}_v(t_j = (I_q, x, m_v, r_q^v, M_v, \mathcal{F}_v^q) \wedge \mathcal{F}_v^p = I_{v_0} \wedge I_q = I_p \wedge x \in \{R, L, U\}) \Rightarrow t_j = (I_p, L, (m_{v_0} + 1)\%3, r_p^{v_0}, M_v, \mathcal{F}_v^p) \vee t_j = (I_p, U, (m_{v_0} - 1)\%3, r_p^{v_0}, M_v, \mathcal{F}_v^p) \vee t_v = (I_p, R, (m_{v_0} - 1)$ *modulo* $3, r_p^p, M_p, \mathcal{F}_p^p))$.

Relabeling :

- $\lambda'(v_0) := (\mathcal{I}_{v_0}, \mathcal{S}_{v_0}, \mathcal{L}_{v_0} - t_i, t_{v_0}, \mathcal{E}_{v_0})$,
- $\forall v \in B(v_0, 1)(v \neq v_0 \wedge \exists t_j \in \mathcal{L}_v(t_j = (I_p, L, (m_{v_0} + 1)$ *modulo* $3, r_p^{v_0}, M_v, \mathcal{F}_v^p) \wedge \mathcal{F}_v^p = I_{v_0} \wedge I_q = I_p \Rightarrow M_{v_0} := max(M_{v_0}, M_v)))$,
- $t_i := (I_p, L, m_{v_0}, r_p^{v_0}, M_{v_0}, \mathcal{F}_{v_0}^p)$,
- $\lambda'(v_0) := (\mathcal{I}_{v_0}, \mathcal{S}_{v_0}, \mathcal{L}_{v_0} + t_i, t_{v_0}, \mathcal{E}_{v_0})$.

To know that a vertex v is already involved in the computation of the tree rooted at a vertex w, one has in Procedure 4.3 to look for an element $t \in \mathcal{L}_v$ whose root vertex has the same identity as w. This fact gives the above rules a more complicated aspect as the rules described in Procedure 4.1 . Nevertheless, these rules exactly perform the required task. The rules $R5$ and $R6$ ensure that each time an internal vertex w is *locked*, the maximal known tuple M_w is actualized bottom-up. This actualization is done up to and including the vertices of the first level.

Corollary 4.2 *During the computation of the rooted minimal paths tree T_v^D, at the end of round $i \leq D$, v is aware of the maximum tuple $n_m \in \mathcal{S}$ amongst the tuples generated by all the W-marked vertices of T_v^i.*

4.4.1 An Experimental Algorithm for Anonymous Networks

Due to the use of identities, Procedure 4.3 is not adapted for anonymous networks. We now present a variant of Procedure 4.2 that is able to solve the k-local election problem in an anonymous network.

Procedure 4.4 (Solving the k-local election problem in an anonymous network)

Step 1: *Each vertex v chooses a random number n_v and performs a $2-$local election (RL_2). The winners and losers of these elections are respectively marked with W and L.*

Step 2: *Each $W-$marked vertex u chooses a random number n_u^* and sets its identity to be the number n_u^*.*

Step 3: *For each $W-$marked vertex u, we construct the tree T_u^D (with depth D) of minimal paths rooted at u.*

Step 4: *Once T_u^D is constructed for a given vertex u, the chosen numbers of all the $W-$marked vertices in T_u^D are compared to n_u^*. If $n_u^* > n_v^*, \forall v \in T_u^D, v \neq u$ (v is W-marked) then u has won the local election in its ball of radius $k = D$.*

As soon as the first two steps have been performed, the last parts of this procedure can be computed effortlessly by Procedure 4.3. Obviously, the correctness of Procedure 4.4 heavily depends on the absence of random numbers coincidences in the whole network. That is, if two W-marked vertices generate the same random number, the algorithm can not guarantee a faithful execution.

Assumption 1 *We assume that each vertex v selects at random uniformly and independently an integer $rand(v)$ from $\{1, .., N\}$. Let X be a set of vertices containing a given vertex v such that $|X| = h$. Under the above assumptions on rand we obtain the probability,*

$$Pr(rand(v) \neq rand(w), \forall w \in X - \{v\}) = \frac{1}{N} \sum_{i=1}^{N} \left(\frac{N-1}{N}\right)^{h-1} = \left(1 - \frac{1}{N}\right)^{h-1}.$$

We need to reduce the value of $Pr(rand(v) = rand(w), \forall w \in X - \{v\})$. To achieve this task, we assume in our framework that the set X represents the set of all W-marked vertices in the network and that $|X| < N$. Due to Fact 4.1 we know that $|X| \leq \frac{|V|}{2}$. Furthermore, the integer N, which is the range of selection for vertices, is supposed to be large enough, so that the probability of coincidence of rand in the whole network becomes small. Thus,

$$Pr(rand(v) = rand(w), \forall w \in X - \{v\}) \quad = \quad 1 - \left(1 - \frac{1}{N}\right)^{\frac{|V|}{2}-1}. \tag{1}$$

This means that if N is large enough, the probability that two vertices generate the same random number converges to 0. This assumption is equivalent to the one supposing that all vertices choose at random uniformly and independently a *real* from the interval $[0, 1]$.

Theorem 4.1 (Correctness) *Procedure 4.4 solves with high probability the k-local election problem in an anonymous network.*

Proof. The proof is derived from the requirements presented in Assumption 1 □

Lemma 4.3 (Complexity) *The time complexity of solving the k-local election for a vertex v_0, in an anonymous network, is the same as the complexity required to compute a tree of minimal paths with depth k and rooted at v_0, that is, the complexity of Procedure 4.4 is $O(k^2)$.*

Proof. Let $n = |V(G)|$. The RL_2 procedure has a message complexity of $O(n^2 + \sum_{v \in G} |N_G(v)|) = O(n^2 + n(n-1))$. Under the time assumptions we have made, it is easy to see that a given vertex v_0 knows, after a constant time, that it has won or lost the 2-local election. Furthermore, Procedure 4.4 has also a worst case message complexity of $O(\gamma(|E| + n * k))$. With $\gamma \leq \frac{n}{2}$ representing the number of vertices that have won the RL_2 procedure. The time complexity of solving the k-local election problem for a given vertex v_0 is the same as the time required to compute a tree of minimal paths with depth k and rooted at v_0. Thus, the time complexity of our procedure is $O(k^2)$. This means that the time complexity of solving the k-local election problem is not constrained by the topology of the network nor by the size of the underlying graph G. □

4.4.2 Collisions Detection

From the expression described in equation (1) of Assumption 1, we know that the probability of collision in the network is indeed very small but it is still higher than 0. For this reason we have to find out some heuristics that should help us to detect and correct random numbers collisions. Due to the use of rooted trees of minimal paths, we are not interested in avoiding collisions in the whole network. In fact, as long as the computations of the trees can be performed faultless, Procedure 4.4 correctly solves the k-local election problem. Thus, we are only interested in avoiding collisions in balls of radius $2k$. Inspired by proofs given in [Ang80], Métivier et al. [MSZ02] have proved the following proposition.

Proposition 4.2 ([MSZ02]) *There is no deterministic or Las Vegas algorithm to detect if there is a coincidence in a ball of radius k for $k > 2$.*

The main consequence of this proposition is that the 2-local election of Procedure 4.4 can be performed such that there are no collisions in balls of radius 2. The main heuristic for detecting random numbers coincidences in a ball of radius $2k$ can then be deduced from the next lemma.

Lemma 4.4 *Let w and v_0 be two vertices such that $w \in \mathcal{T}_{v_0}^k$. If $d(w, v_0) = d$ and $\exists w_1 \in B(w, 1)$ such that $w_1 \in \mathcal{T}_{v_0}^k$ and $|d(w, v_0) - d(w_1, v_0)| > 1$ then there exists a vertex v_0' such that $n_{v_0}^* = n_{v_0'}^*$.*

Proof. During the computation of the tree $\mathcal{T}_{v_0}^k$ all unlabeled vertices that are at distance d (with $d \leq k$) from v_0 are labeled in the same round and all of them are aware of the value of d. This means that for a given vertex w that is at distance d from v_0 and that belongs to $\mathcal{T}_{v_0}^k$, all the neighbors of w that belong to $\mathcal{T}_{v_0}^k$ are at distance $d_i \in \{d-1, d, d+1\}$. Thus, $\forall w_1 \in B(w, 1)$, $w_1 \in \mathcal{T}_{v_0}^k \Rightarrow |d(w, v_0) - d(w_1, v_0)| \leq 1$. □

Once a collision is detected by a vertex w, an error message is sent to the vertices involved in the collision. As soon as a vertex knows that it is involved in a collision, it set its state to L and continues with the computation. A second alternative could also consist in generating a new random number and starting the construction of a new rooted tree of minimal paths.

4.5 Concluding Remarks

It is clear that, using Lemma 4.4, allows to detect most of numbers coincidences in balls of radius $2k$. Nevertheless, there are a small set of collisions that can not be detected by a vertex $w \in \mathcal{T}_{v_0}^k$ where $d(w, v_0) = d$. As a matter of course, the vertex w could never be able to detect the collision between v_0 and v_0' if $d(v_0, v_0') = D \leq 2k$ and $D \in \{2d, 2d+1, 2d+2\}$. This relies on the fact that all the neighbors of w that belong to the tree $\mathcal{T}_{v_0'}^k$ are supposed to be at distance $d_i \in \{d-1, d, d+1\}$ from a vertex, in this case v_0', having the same identity as v_0. That is, the conditions of Lemma 4.4 could not be satisfied and w could not be able to detect this collision. In spite of our impossibility to detect this kind of random numbers coincidences, our algorithm has performed very significant results in practical uses. This is due to the fact that the probability of collision can be considerably reduced by using random real numbers

from the set $[0, 1]$. The use of several heuristics also plays a decisive role for the robustness of our methodology.

Chapter 5

Recognition of Graph Properties with Local Computations

Contents

5.1 Reduction Systems

This chapter focus on a classical problem for distributed algorithms, the recognition problem. Formally, we intend to compute some topological information on a network of processes, possibly using additional knowledge about the structure of the underlying graph.

Litovsky et al. [LMZ95] have used local computations to define a general notion of graph *recognizers*. To this end they specified an initial labeling and a terminal family $\mathfrak{K}$ of irreducible labeled graphs. A graph G is recognized if the labeled graph (G, λ), where λ is an initial labeling, can be reduced to a graph in $\mathfrak{K}$. They introduced the notion of k-covering and proved, for instance, that the family of series-parallel graphs and the family of planar graphs can not be recognized by means of local computations. Starting from these results, Godard et al. [GMM04] have proposed the notion of recognition with structural knowledge by means of local computations. Using coverings and a distributed enumeration algorithm proposed by

A. Mazurkiewicz [Maz97], they characterized the graph classes that are recognizable with or without structural knowledge. Their notion of recognition with structural knowledge consists in defining a labeled graph recognizer with structural knowledge ι as a triple $(\mathcal{R}, \mathcal{K}, \iota)$ where $\mathcal{R}$ is a locally generated relabeling relation that is noetherian on the set $\{(G, \Lambda_{\iota(\boldsymbol{G})}) | \boldsymbol{G} \in \mathcal{G}_L\}$, $\mathcal{K}$ is the final condition (i.e., there is a recursive set K of finite subsets of L such that $\mathcal{K} = \{\boldsymbol{G} \in \mathcal{G}_L | lab(\boldsymbol{G}) \in K\}$), and ι is a computable function which associates with each labeled graph $\boldsymbol{G}$ a label $\iota(\boldsymbol{G}) \in L$. In this framework $\mathcal{G}_L$ is the class of labeled graphs over a fixed set L and a labeled graph $\boldsymbol{G}$ is *recognized* by $(\mathcal{R}, \mathcal{K}, \iota)$ if $\mathrm{Irred}_{\mathcal{R}}((G, \Lambda_{\iota(\boldsymbol{G})})) \cap \mathcal{K} \neq \emptyset$.

The main goal of the present work is to present an algorithmic solution for solving the recognition problem in the local computations environment. To this end the procedure based on labeled graph recognizers is not very adequate. As a matter of fact, deciding that a labeled graph $\boldsymbol{G}$ satisfies a given property, consists in checking if $\mathrm{Irred}_{\mathcal{R}}((G, \Lambda_{\iota(\boldsymbol{G})})) \cap \mathcal{K} \neq \emptyset$. That is, we have to find an element of $\mathcal{K}$ isomorphic to an element of $\mathrm{Irred}_{\mathcal{R}}((G, \Lambda_{\iota(\boldsymbol{G})}))$. Unfortunately, the graph isomorphism problem is known to be NP-hard. Even if we assume the existence of unique processes identifiers, solving this problem by means of local computations could be very tedious. A further objection to the use of labeled graph recognizers concerns the set $\mathcal{K}$ whose size can be unbounded. A better solution is proposed in [LMZ95] where the set of terminal graphs $\mathcal{K}$ is specified by special conditions defined by means of propositional formulas. Nevertheless, this approach makes it possible neither to count vertices or edges with given labels, nor to verify their relative positions. For example, it is impossible to specify that the final labeled graph contains exactly one vertex labeled by a given element $\boldsymbol{T}$ or that there exists two adjacent vertices labeled by $\boldsymbol{T}$. We could certainly used a better logical language to express the propositional formulas. But we would still have the problem of evaluating them efficiently in the local computations framework.

We propose an algorithmic approach of the recognition problem based on the concept of *graphs reduction* which is a further means to define sets of finite graphs.

In fact, there are several ways to define sets of finite graphs by finite devices. The main ones are graph grammars (see Handbook on graph grammars [EKMR99], Courcelle [Cou90b, Cou90a]), systems of equations (see. Bauderon et al. [BC87]), logical formulas (in particular monadic second-order logic [Cou90b, ALS91]), by forbidden configurations such as forbidden minors (Robertson et al. [RS87], Arnborg et al. [APC90]) and finally by reduction.

A terminating reduction system $(\mathcal{R}, K)$ consists of a rewriting relation $\rightarrow_{\mathcal{R}}$ and a finite set K of accepting graphs. Given any graph G, every sequence of $\rightarrow_{\mathcal{R}}$-rewritings terminates. Such a terminating sequence yields a graph called a *normal form* of G. A rewriting system defines a class L of graphs if every normal form of a graph $G \in L$ is in K and if no normal form of $H \notin L$ is in K.

Classical examples of graph reduction concern trees or series-parallel graphs. For instance, consider a graph G. Remove a pendant edge $\{u, v\}$ with its end vertex u. Repeat this process until a graph G_0 with no pendant edge is obtained. Obviously, G_0 is an isolated vertex if and only if the original graph was a tree.

The idea of reduction algorithms originates from Duffin's characterization of series-parallel graphs using *series* and *parallel* reductions rules [Duf65]. By taking advantage of these reduction rules, Valdes et al. [VTL79] have presented a reduction algorithm that recognizes series-parallel graphs in linear time. Arnborg and Proskurowski [AP86] extended these ideas and obtained reduction rules that characterize the graphs of treewidth at most 3 and, amongst

other, they showed that these reduction rules can be used to recognize graphs of treewidth at most 3 in $O(n^3)$ time. In [MT91] an improved version of this algorithm that runs in linear time is given.

Arnborg et al. [ACPS93] have presented a more general approach based on algebraic setting. They introduced a set of conditions that ensure faithful graph reduction algorithms. Moreover, they showed that for a large class of decision problems on graphs of bounded treewidth, there is a set of rules for which these conditions hold. These include all MS-definable decision problems. A linear reduction algorithm based on such a set of reduction rules was given.

Bodlaender and Hagerup [BH98] have shown that the sequential reduction algorithm presented by Arnborg, Courcelle and Proskurowski [ACPS93] can be efficiently parallelized, if the set of reduction rules satisfies some additional conditions. They also showed that these conditions are satisfied by all finite state decision problems, assuming yes-instances have bounded treewidth.

In this chapter, we will first extend the framework of reduction algorithms by presenting the first distributed algorithm that computes reduction algorithms with success. The local computations systems are essentially "static" in the sense that they never change the underlying structure of the graph on which they work, but only the labeling of its components (edges or vertices). This principle represents the main challenge in encoding reduction algorithms by local computations. As a matter of fact, to achieve this task, we have to find a way to express the removal or the addition of graph components in the underlying network. Therefore, we introduce an encoding system for reduction algorithms based on four basic reduction rules that can easily be encoded by local computations. These are *edge addition*, *edge deletion*, *vertex deletion* and *edge contraction*. Starting from these rules, we state necessary conditions that must hold for a set of reduction rules to ensure a faithful distributed execution. These conditions have induced the introduction of the concept of *handy reduction system*. Moreover, we prove that these conditions are satisfied by all sets of reduction rules which characterize finite state decision problems on graphs of bounded treewidth. For instance, we prove that all graphs of bounded treewidth that satisfy a given graph property definable in the monadic second order logic can be *recognized* by local computations.

Using the notion of *recognition with structural knowledge by means of local computations* [GMM04], we state an immediate relationship between handy reduction systems and labeled graph recognizers with structural knowledge. We also prove that if $\mathfrak{C}$ is a class of covering minimal graphs of bounded treewidth that satisfy any MS-definable property P, then $\mathfrak{C}$ is size recognizable. That is, there is an algorithm encoded by local computations that, given a covering minimal graph G and its size $|V(G)|$, decides whether G belongs to $\mathfrak{C}$ or not.

Ordinary reduction algorithms do not provide a possibility of constructing solutions, but only decide upon membership in a class of graphs. Therefore, we make use of *constructive reduction systems*, introduced in [BvAdF01], to present a distributed algorithm that, given a graph G and a constructive reduction system $(\mathcal{R}, \mathcal{I})$ for a property P, performs the reduction rules on G until a normal form is reached. If G belongs to P, a solution that attests that G satisfies P is constructed. Obviously, constructive reduction systems are only useful for graph properties of the form: $P(G) =$ "there is an $S \in D(G)$ for which $Q(G, S)$ holds" where $D(G)$ is a solution domain, which depends on G and Q is an extended graph property of G and S; i.e., $Q(G, S) \in \{\text{true}, \text{false}\}$ for all G and all $S \in D(G)$.

We now present some basic definitions that are useful for the understanding of the graph reduction paradigm [BvAdF01].

Definition 5.1 (Sourced graphs) *A sourced graph G is a triple (V, E, X) with (V, E) an undirected graph, and $X \subseteq V$ is an ordered subset of the vertices, denoted by $\langle x_1, ..., x_l \rangle, l \geq 0$, called the set of sources. Vertices in $V - X$ are called inner vertices. A sourced graph (V, E, X) is called an l-sourced graph if $|X| = l$. A sourced graph (V, E, X) is said to be open, if there are no edges between sources $(\forall v, w \in X, \{v, w\} \notin E)$. The usual undirected graph (i.e., without sources) will be simply called graph.*

Definition 5.2 *The operation $\oplus$ maps two sourced graphs G and H with the same number l of sources to a graph $G \oplus H$, by taking the disjoint union of G and H, then identifying the corresponding sources, i.e., for $i = 1, ..., l$ the ith source of G is identified with the ith source of H, and then removing multiple edges (i.e. see Figure 13).*

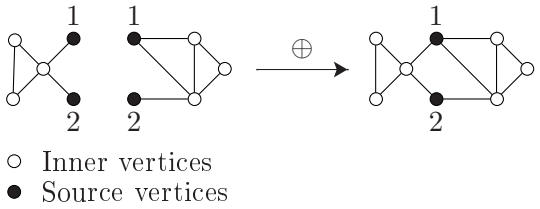


Figure 13: Example of $\oplus$-operation for two graphs

Two sourced graphs $(V_1, E_1, \langle x_1, ..., x_k \rangle)$ and $(V_2, E_2, \langle y_1, ..., y_l \rangle)$ are said to be *isomorphic*, if $k = l$ and there exists a bijective function $f : V_1 \to V_2$ with $\forall v, w \in V_1, \{v, w\} \in E_1 \leftrightarrow \{f(v), f(w)\} \in E_2$ and for all $i, 1 \leq i \leq k, f(x_i) = y_i$. The main difference with the usual definition of graph isomorphism is that we require the corresponding sources to be mapped to each other.

Definition 5.3 (Reduction rule) *A reduction rule r is an ordered pair (H_1, H_2), where H_1 and H_2 are l-sourced graphs for some $l \geq 0$. An application of the reduction rule (H_1, H_2) is the operation, that takes a graph G of the form $G_1 \oplus G_3$, with G_1 isomorphic to H_1, and replaces it by the graph $G_2 \oplus G_3$, with G_2 isomorphic to H_2. We write $G \xrightarrow{r} G_2 \oplus G_3$.*

For two graphs G and G', and a set of rules $\mathcal{R}$, we write $G \xrightarrow{\mathcal{R}} G'$, if there exists an $r \in \mathcal{R}$ with $G \xrightarrow{r} G'$. An example of the application of a reduction rule $r = (H_1, H_2)$ is given in Figure 14. Given a reduction rule $r = (H_1, H_2)$, we call H_1 the left-hand side of r, and H_2 the right-hand side of r. G contains a match G_1 if there is an $r \in \mathcal{R}$ such that G_1 is a match to r in G. If G contains no match, then we say that G is *irreducible* (for $\mathcal{R}$).

The following conditions are useful for a set of reduction rules in order to get a characterization of a graph property P.

Definition 5.4 *Let P be a graph property and $\mathcal{R}$ a set of reduction rules.*

- *$\mathcal{R}$ is safe for P if, whenever $G \xrightarrow{\mathcal{R}} G'$, then $P(G) \Leftrightarrow P(G')$.*

- *$\mathcal{R}$ is complete for P if the set $\mathcal{I}$ of irreducible graphs for which P holds is finite.*
- *$\mathcal{R}$ is decreasing if, whenever $G \xrightarrow{\mathcal{R}} G'$, then G' contains fewer vertices than G or more generally $|V(G')| + |E(G')| > |V(G)| + |E(G)|$.*

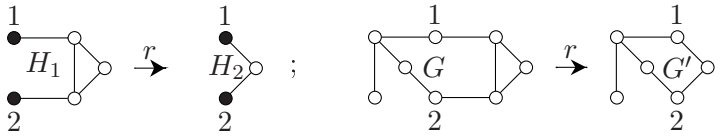


Figure 14: Applying rule r to G yields G'.

Definition 5.5 *(Reduction System) A reduction system for a graph property P is a pair $(\mathcal{R},\mathcal{I})$, with $\mathcal{R}$ a finite set of reduction rules which is safe, complete, and decreasing for P, and $\mathcal{I}$ the set of irreducible graphs for which P holds.*

A reduction system $(\mathcal{R},\mathcal{I})$ for a property P gives a complete characterization of P: $P(G)$ holds for a graph G if and only if any sequence of reductions from $\mathcal{R}$ on G leads to a graph G' which belongs to $\mathcal{I}$ (i.e. is isomorphic to a graph in $\mathcal{I}$).

The following lemma both serves to show a limit on what problems can be solved with graph reduction, and can be used to turn sequential algorithms with running time linear in the number of edges into algorithms with time linear in the number of vertices.

Lemma 5.1 *([BvAdF01]) Suppose that property P is characterized by the reduction system $(\mathcal{R},\mathcal{I})$. Then, there is a number c, such that for all graphs $G = (V, E)$ with $P(G) = true$, $|E| \leq c \cdot |V|$.*

5.2 Encoding Reduction Rules in a Distributed System

The main task of this section is to present a general distributed algorithm that, given a graph G and a set of reduction rules $\mathcal{R}$, performs all the reduction steps on G until the resulting graph is irreducible for $\mathcal{R}$. Before we plunge in the details of this procedure, we first state the boundary conditions that ensure the correctness of our algorithm.

As a matter of course, depending on the computational model, the distributed execution of reduction rules comes up against several problems. The major difficulty we have to face while performing reduction rules in the local computations environment relies on the fact that local computations do not allow structural changes of the basic underlying graph. To tackle this inconvenient, we have to design an adequate encoding of reduction rules. This encoding should satisfy all the requirements of the local computations framework.

5.2.1 Encoding the Reduction Rules

To encode reduction rules, we define four basic rules: *vertex deletion*, *edge deletion*, *edge addition* and *edge contraction*. Starting from these rules, we will define a class Υ of reduction

rules that can easily be encoded in our framework. Further, we will show that there is a large class of decisions problems that can be encoded using the rules of the class Υ. To encode the basic reduction rules on a graph G, some requirements have to be satisfied:

1. each vertex $v \in G$ has a unique identifier I_v.
2. G is a labeled graph described as (G, λ, δ).
3. $\delta_v : I_G(v) \rightarrow \sum$ is an injective function which associates to each edge incident to v a distinct port number from the set $\sum$.
4. λ is a vertex labeling function such that for a vertex v, $\lambda(v) = (S, \mathcal{P}_v)$, with
   - $S \in \{A, T\}$ is a token that should express the state of vertex v during the computation.
   - $\mathcal{P}_v$ is a set of paths that contains for each neighbor w of v exactly one path p_v^w from v to w.

Initially, all the vertices of a labeled graph (G, λ, δ) are A-labeled. That is, for each vertex v it holds $\lambda(v) = (A, \mathcal{P}_v)$.

Definition 5.6 *Let (G, λ, δ) be a labeled graph with port labeling. The underlying graph of (G, λ, δ) denoted G_λ^A is the subgraph of G that contains all the A-labeled vertices of (G, λ, δ).*

In this representation, a vertex can be in the states *active* (A) or *tunneling* (T). Moreover, each path $p = \{v, v_1, v_2, ..., v_k\} \in \mathcal{P}_v$ is encoded using the port labeling of the vertices traversed by p. To avoid confusion and to distinguish the elements of $\mathcal{P}_v$ from each other, each path $p = \{v_0, v_1, v_2, ..., v_n\}$ is identified by I_{v_n}. The use of the port numbering function δ does not represent a restriction to regular graphs having known degrees. In fact, the graphs we generally deal with could have unbounded degrees.

Encoding the basic reduction rules in the local computations framework can be done in the following way.

Vertex Deletion (*delete_node*(u))**:** The deletion of an active vertex u is performed in two steps. In the first step, each neighbor w of u removes the path p_w^u from its set $\mathcal{P}_w$. In the second step, the state of u is set to D and $\mathcal{P}_u$ remains unchanged.

Edge Deletion (*delete_edge*$((u, v))$)**:** During the deletion of an edge $e = (u, v)$, u removes the path p_u^v from its set $\mathcal{P}_u$ and v removes the path p_v^u from its set $\mathcal{P}_v$.

Edge Addition (*Add_edge*$((u, v))$)**:** During the addition of an edge $e = \{u, v\}$, u adds the path p_u^v to its set $\mathcal{P}_u$ and v do the same with the path p_v^u.

Edge Contraction (*contract_edge*$((u, v))$)**:** To contract an edge $e = (u, v)$, we set u in the state T and $\mathcal{P}_u$ remains unchanged. Afterward, the values of $\mathcal{P}_v$ and $\mathcal{P}_w$ for w such that $p_u^w \in \mathcal{P}_u$ are actualized as follows: $\mathcal{P}_v = \mathcal{P}_v - p_v^u \cup \{p_v^w | p_u^w \in \mathcal{P}_u \wedge p_v^w \notin \mathcal{P}_v\}$ and $\mathcal{P}_w = \mathcal{P}_w - p_w^u + p_w^v$. That is, each active neighbor of u that was not a neighbor of v becomes a neighbor of v after the contraction of the edge e. The above representation also ensures the suppression of multiple edges that can appear during an edge contraction.

All the described actions are depicted in Figure 15. Obviously, an edge contraction rule can be simulated by successive executions of the other three basic rules. Due to the requirements involved by our computation model, any kind of reduction rules can not be expressed by local computations. As a matter of course, if the left-hand side of a reduction rule is not connected, our encoding system will never be able to simulate its execution. This is why we restrict our attention to *handy reduction rules.*

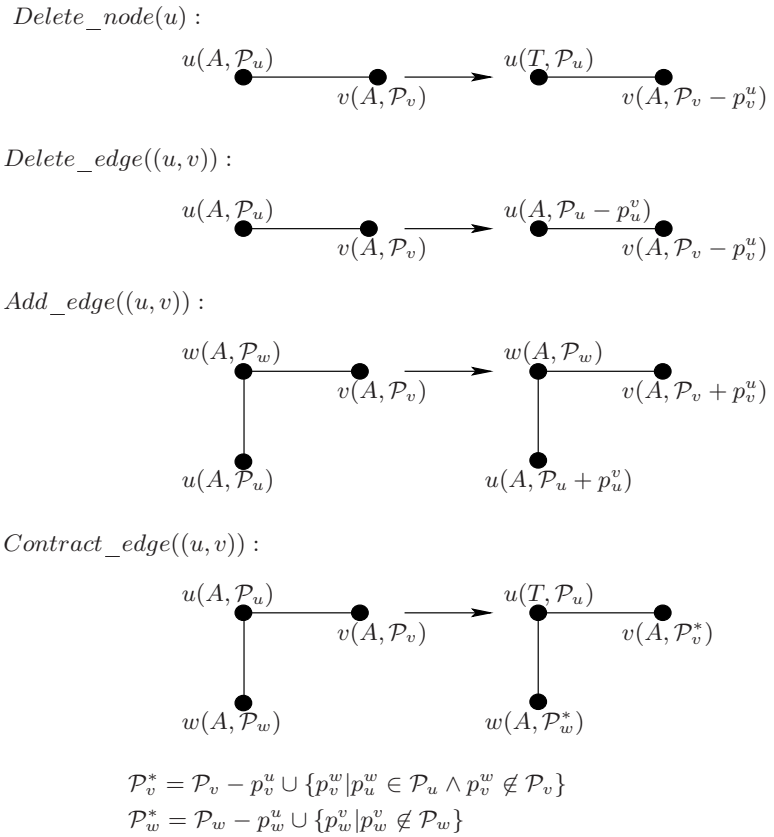


Figure 15: Encoding basic reduction rules by local computations

Definition 5.7 (Handy reduction rule) *Let $r = (H_1, H_2)$ be a reduction rule. r is said to be a handy reduction rule if the following conditions hold.*

1. *H_1 and H_2 are connected,*
2. *$|V(H_2)| \leq |V(H_1)|$.*

The main aspect of handy reduction rules resides in the fact that no arbitrary vertices are created during the reduction step. That is, new edges and vertices can only be created while executing an edge addition or an edge contraction rule.

Definition 5.8 (Handy reduction system) *A handy reduction system is a reduction system $(\mathcal{R}, \mathcal{I})$ where $\mathcal{R}$ only contains handy reduction rules and all graphs $G \in \mathcal{I}$ are connected.*

Lemma 5.2 *Let $(\mathcal{R}, \mathcal{I})$ be a handy reduction system. Then any $r \in \mathcal{R}$ can be computed by a fixed sequence of rules $r_l = \{b_1, b_2, ..., b_k\}$ where b_i are basic reduction rules.*

Proof. Let $r = (H_1, H_2) \in \mathcal{R}$ be a handy reduction rule. To show the validity of this lemma we are going to give a procedure $\mathcal{C}_r$ that, starting from the graph H_1 constructs the graph H_2. Let $\mu : V(H_2) \to V(H_1)$ be an injective function that maps each vertex of H_2 to a vertex of H_1. μ maps source vertices of H_2 to source vertices of H_1. Assume $\mathfrak{D} = \{u \in V(H_1) | \forall w \in V(H_2), \mu(w) \neq u\}$. $\mathcal{C}_r$ constructs the graph H_2' by performing the following steps on H_1:

Step 1: Remove all vertices of $\mathfrak{D}$ from H_1. After this, μ becomes a bijection.

Step 2: For each edge $\{u, v\} \in E(H_2)$ if $\{\mu(u), \mu(v)\} \notin E(H_1)$, then add the edge $\{\mu(u), \mu(v)\}$ to $E(H_1)$.

Step 3: For each edge $\{u, v\} \in E(H_1)$ if $\{\mu^{-1}(u), \mu^{-1}(v)\} \notin E(H_2)$, then the edge $\{u, v\}$ is deleted from $E(H_1)$.

As a matter of fact, the graphs H_2' and H_2 have the same number of vertices. Furthermore, $\{u, v\}$ is an element of $E(H_2')$ if and only if $\{\mu^{-1}(u), \mu^{-1}(v)\}$ is an element of $E(H_2)$. Thus, the graphs H_2' and H_2 are isomorphic. This means that H_2 can be constructed from H_1 by successive executions of the basic reduction rules: vertex deletion (Step 1), edge addition (Step 2) and edge deletion (Step 3). □

The particularity of basic reduction rules is that, within our encoding system, they can easily be computed by local computations on balls of a fixed radius k (see Figure 15). By the hand of Lemma 5.2 we can also apply this property to handy reduction rules.

Fact 5.1 *Each handy reduction rule r can be represented by a sequence of basic reduction rules r_l that allows the execution of r in the local computations framework.*

Remark 5.1 *Handy reduction rules can be seen as a strong restriction of the concept of reduction algorithm. Nevertheless, we will show that all decision problems, on graphs of bounded treewidth, definable in the monadic second order logic [Cou90b] can be decided using adequate sets of reduction rules satisfying the conditions stated in Definition 5.7.*

In the local computation framework, we generally deal with asynchronous messages passing systems. This means that we have to act with caution while sending messages. For this reason, each message m sent by any process u to a process v contains the path p_u^v. Each time a vertex w receives the message m, it checks if the identity of m is the same as its own identity. If this is not the case, w sends m to the next vertex that appears in the path p_u^v.

The way reduction rules are encoded plays a significant role for the correct execution of the computed algorithms. For instance, two active vertices, that become neighbors after the execution of an edge contraction rule, must have a global knowledge about their actual neighborhoods. Hopefully, the properties of the basic reduction rules ensure that, at any given time, each vertex of G_λ^A has global knowledge about the set of its neighbors. Hence, the set of all A-labeled vertices gives a complete representation of the graph obtained after a reduction operation on graph G. In fact, if there is a graph G' such that there is a rule r with $G \xrightarrow{r} G'$,

then there exists a labeling function λ' such that $(G, \lambda, \delta) \xrightarrow{r} (G, \lambda', \delta)$. In the same way it is easy to show that the graphs G' and $G^A_{\lambda'}$ are isomorphic. The labeling function λ' is therefor represented by the encoding of λ after the execution of the sequence r_l on (G, λ, δ).

Proposition 5.1 *Let P be a property characterized by a handy reduction system $(\mathcal{R}, \mathcal{I})$ and G a graph. If $(G, \lambda, \delta) \xrightarrow{\mathcal{R}} (G, \lambda_i, \delta)$, then $P(G) \Leftrightarrow\Leftrightarrow P(G^A_{\lambda_i})$.*

Proof. Due to the fact that the initial basic underlying graph of (G, λ, δ) has the same properties as G, it is evident that $P(G) \Leftrightarrow P(G^A_\lambda)$. The set $\mathcal{R}$ is safe and we know from Lemma 5.2 that the execution of any rule $r \in \mathcal{R}$ can be effectively simulated in (G, λ, δ). Thus, $(G, \lambda, \delta) \xrightarrow{\mathcal{R}} (G, \lambda_i, \delta)$ implies $P(G^A_\lambda) \Leftrightarrow P(G^A_{\lambda_i})$. □

5.2.2 Distributed Computations of Reduction Rules

Our main purpose here is to present a distributed algorithm that, given a graph (G, λ, δ) and a set of handy reduction rules $\mathcal{R}$, performs all the reduction steps on G^A_λ until the resulting graph is irreducible for $\mathcal{R}$. Before we go into the details of this procedure, we first state the boundary conditions that ensure the correctness of the designed algorithm.

First of all we require that the functions λ and δ satisfy the conditions of our encoding system as described in Section 5.2.1. Moreover, we set $k = min\{d \in \mathbb{N} | d \geq \Delta(H_1), \forall r = (H_1, H_2) \in \mathcal{R}\}$. Starting from the graph G, we construct the labeled graph (G, λ, δ) and simulate the execution of the reduction rules of $\mathcal{R}$. This computation works in rounds and in each round all vertices execute the three steps of Algorithm 5.1. The correctness of this

Algorithm 5.1 Computing reduction rules

Input: A graph (G, λ, δ) and a set of handy reduction rules $\mathcal{R}$.
Each vertex v executes the following steps forever.
Step 1: v performs a $(2k+2)$-local election. All the winners become W-marked and the other L-marked.
Step 2: each W-marked vertex v looks for a rule $r \in \mathcal{R}$ such that r_l can be applied on $(B_{G^A_\lambda}(v, k+1), \lambda, \delta)$.
if such a rule could be found, **then**
  go to Step 3;
else
  the procedure continues in Step 1.
end if
Step 3: r_l is performed on $(B_{G^A_\lambda}(v, k+1), \lambda, \delta)$. The procedure continues in Step 1.

algorithm strongly depends on the correctness of the procedure used to perform the local election and the matching search. We are going to briefly refer to the details of Step 1 and Step 2 of Algorithm 5.1.

The k-local Election. To solve the k-local election problem, we take advantage of the methodology introduced in [Oss05b] and presented in Chapter 4. This methodology is suitable for graphs with identities. Moreover, it can can also be used to generate a Las Vegas algorithm

that solves the local election problem in anonymous graphs with very high probability. For a given graph $G = (V, E)$, the procedure given in [Oss05b] ensures that any winner vertex $v \in V$ is aware of the subgraph representing the ball $B_G(v, k)$. After each k-election round, each winner vertex v has computed a tree of minimal paths rooted at v having a maximal number of levels. This tree, denoted T_v^k, has at most k levels. That is, all vertices $w \neq v$ are at distance at most k from v. This principle induces the following fact.

Fact 5.2 ([Oss05a]) *Let v be a vertex that has won the k-local election in a graph G. If T_v^k has at most $k-1$ levels, then $B_G(v,k) = G$ and all vertices $u \neq v$ are L-marked.*

Finding Reduction Matches. Once a vertex v has won the $(k+1)$-local election of Algorithm 5.1, it has to find a match G_1 for any reduction rule $r = (H_1, H_2) \in \mathcal{R}$ in $B_{G_\lambda^A}(v, k+1)$. In contrast to the sequential and parallel reduction algorithms ([BvAdF01], [BH98]) where one looks for a match in the whole graph, our algorithm tries, for a W-marked vertex v, to find a match in the subgraph containing v. It is clear that if $k \geq \frac{\Delta(G_\lambda^A)}{2} - 1$, then one has to search in the whole graph G_λ^A. This allows several reduction steps to be performed simultaneously on non overlapping balls of radius $k+1$.

Let now $r = (H_1, H_2) \in \mathcal{R}$ be a reduction rule. The problem of finding a match G_1 in $B_{G_\lambda^A}(v, j)$ is equivalent to looking for an induced subgraph of $B_{G_\lambda^A}(v, j)$ ($j \in \mathbb{N}$) isomorphic to H_1. For this reason, we use the algorithms of Bunke et al. [BM00] and the procedure introduced by Cortadella et al. [CV00] to solve the subgraph isomorphism problem in Step 2 of Algorithm 5.1. With this strategy we are able to correctly solve the subgraph isomorphism problem in our framework. If j has a reasonable size and G has a small degree, then it is also possible to use the tree pruning algorithm introduced by Ullman [Ull76].

In order to speed up the performance of the used subgraph isomorphism algorithm, we require that the W-marked vertex v of $B_{G_\lambda^A}(v, k+1)$ has to be mapped to one source of the graph H_1. If H_1 only contains inner vertices, then v is mapped to a vertex that has to be removed from the underlying graph, or which adjacent edge has to be deleted during the reduction Step. Thus, the size of the pruning tree can be considerably reduced.

Lemma 5.3 *Let $\mathcal{M} \subseteq \mathcal{R}$ be a set of rules that have a match in G_λ^A. If $\mathcal{M} \neq \emptyset$ then during some computation round t, Algorithm 5.1 performs at least one reduction rule on G_λ^A.*

Proof. If a given rule $r = (H_1, H_2) \in \mathcal{R}$ has a match in G_λ^A, then this match is in any case a subgraph of a ball of radius $k+1$ centered on some vertex w. Thus, as soon as w wins a $(2k+2)$-local election, rule r can be performed on $B_{G_\lambda^A}(w, k+1)$. If more than one rule can be executed on $B_{G_\lambda^A}(w, k+1)$, only one rule is performed. The choice of the performed rule is done in an equiprobable way. □

We are now interested in the way parallel reduction steps on overlapping subgraphs are avoided.

Lemma 5.4 *All the parallel reduction steps executed in Algorithm 5.1 occur on distinct subgraphs of radius $k+1$.*

Proof. Without loss of generality, let r_i, $r_j \in \mathcal{R}$ be two rules executed respectively by the vertices w_1 and w_2. The execution of r_i and r_j are performed on balls of radius $k+1$. If w_1 and w_2 have won a $(2k+2)$-local election, then all vertices of the sets $\{v \in V | d(v, w_1) \leq 2k+2\}$

and $\{v \in V | d(v, w_2) \leq 2k+2\}$ are L-marked. This means that $d(w_1, w_2) > 2k+2$. If $d(w_1, w_2) = 2k+3$ then r_i and r_j are performed on different balls of radius $k+1$. More generally, if $v_0 \in B_{G_\lambda^A}(w_1, k+1)$ and $u_0 \in B_{G_\lambda^A}(w_2, k+1)$, then $d(v_0, u_0) \geq 2$. □

The correctness of the next lemma follows immediately from Lemma 5.3 and Lemma 5.4.

Lemma 5.5 *Algorithm 5.1 performs reduction rules on simple graphs.*

5.3 Distributed Computations of Decision Problems

Now we exploit the concepts described in the previous section to present the first distributed algorithm that, given a handy reduction system $(\mathcal{R}, \mathcal{I})$ for a property P, decides whether property P holds for a given graph G. Starting from the input graph, our algorithm repeats applying rules from $\mathcal{R}$ until no rule can be applied anymore. If the resulting graph belongs to the set $\mathcal{I}$, then P holds for the input graph. The main challenge in this procedure relies in the local detection of the global termination of the reduction algorithm. In fact, the local detection of the explicit termination of a distributed algorithm can be pretty difficult and is sometimes not practicable ([MMW97]). To overcome this difficulty, we slightly modify Algorithm 5.1 and added new computation steps that permit the local detection of the global termination.

Let (G, λ, δ) be a labeled graph and $(\mathcal{R}, \mathcal{I})$ a handy reduction system for a property P. We define k and k_1 as $k = min\{d \in \mathbb{N} | d \geq \Delta(H_1), \forall r = (H_1, H_2) \in \mathcal{R} \text{ and } d \geq \Delta(G_i), \forall G_i \in \mathcal{I}\}$ and $k_1 = min\{d \in \mathbb{N} | d = \Delta(G_i), \forall G_i \in \mathcal{I}\}$. Starting from the graph (G, λ, δ), we simulate the execution of the reduction system $(\mathcal{R}, \mathcal{I})$ using Algorithm 5.2.

Termination Detection. Detecting the termination in Algorithm 5.2 includes at most three cases. There is the case where $P(G)$ holds and two other cases that can occur when P does not hold for the input graph G. The second case is subsumed in the correctness of the next lemma.

Lemma 5.6 *During the computation of Algorithm 5.2 by vertex v, if the tree $T_v^{(2k+2)}$ has at most $2k+1$ levels, no reduction rule could be applied and no graph $G_i \in \mathcal{I}$ isomorphic to $B_{G_\lambda^A}(v, k+1)$ was found, then $P(G)$ does not hold.*

Proof. We deduce from Fact 5.2 that $B_{G_\lambda^A}(v, k+1) = G_\lambda^A$. This induces that if no reduction rule can be applied on $B_{G_\lambda^A}(v, k+1)$ and there is no graph $G_i \in \mathcal{I}$ isomorphic to $B_{G_\lambda^A}(v, k+1)$, then the graph G_λ^A is irreducible and $P(G_\lambda^A)$ does not hold. Proposition 5.1 leads to the correctness of the above lemma. □

The remaining situation concerns the case where $P(G)$ does not hold, G_λ^A is irreducible and the size of G_λ^A does not allow any vertex v to have a view over the whole network. That is specially the case when $T_v^{(2k+2)}$ has at least $2k+2$ levels. In this situation, no vertex will be able to detect locally the global termination of the reduction procedure. This means that Algorithm 5.2 will never stop. Nevertheless, it is possible to overcome this difficulty by using snapshot based property detection techniques as described in Chapter 6 and presented in [CL85, MO04a]. For instance, if any vertex v knows that each vertex has won at least

Algorithm 5.2 Distributed test of a graph property

Input: A graph (G, λ, δ) and a handy reduction system $(\mathcal{R}, \mathcal{I})$.
Each vertex v executes the following steps:
Step 1: v performs a $(2k+2)$-local election. All the winners become W-marked and the other L-marked.
Step 2: each W-marked vertex v looks for a rule $r \in \mathcal{R}$ such that r_l can be applied on $(B_{G_\lambda^A}(v, k+1), \lambda, \delta)$.
if such a rule could be found, **then**
  the procedure continues in Step 3.
else
  the procedure continues in Step 4.
end if
Step 3: r_l is performed on $(B_{G_\lambda^A}(v, k+1), \lambda, \delta)$. The procedure continues in Step 1.
Step 4: v checks if any graph $G_i \in \mathcal{I}$ is isomorphic to the graph $B_{G_\lambda^A}(v, k+1)$.
if a graph $G_i \in \mathcal{I}$ is found, **then**
  v sends a stop signal to all $u \in G_\lambda^A$. Thus, $P(G)$ holds and the algorithm stops.
else
  the procedure continues in Step 5.
end if
Step 5:
if $T_v^{(2k+2)}$ has at most $2k+1$ levels, **then**
  $P(G)$ does not hold and the execution can be stopped.
else
  the procedure continues in Step 1.
end if

one local election and, regardless of this fact, no reduction rule was performed, then v can broadcast the stop signal.

Lemma 5.7 (Decidability of P) *Given a decreasing reduction system $(\mathcal{R},\mathcal{I})$ for a property P, Algorithm 5.2 correctly recognizes simple graphs for which the property P holds.*

Proof. This is an immediate consequence of the fourth step of algorithm 5.2 and of the correctness of Lemma 5.5 and Lemma 5.6. □

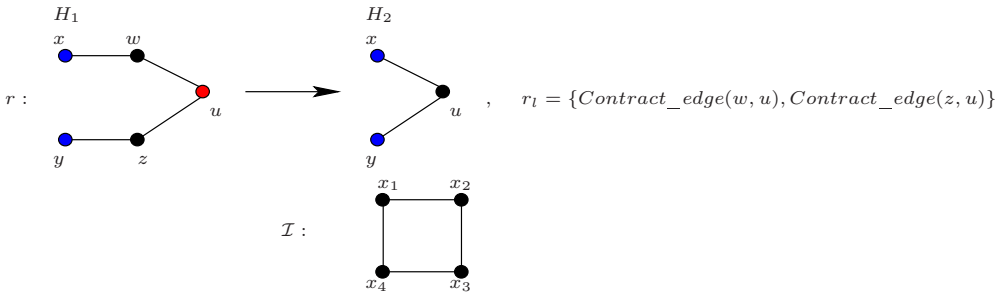
The two-colorable cycle problem. We illustrate the execution of our algorithm by the hand of a simple example. Consider the reduction system of a graph property P, where $P(G)$ holds if and only if G is a two-colorable cycle. Let $(\mathcal{R},\mathcal{I})$ be the reduction system where $\mathcal{I}$ is the set containing only the cycle on four vertices. We assume that vertices that are ready to perform a reduction rule are red colored and the others are respectively black, green or blue colored depending on their relative position in the ball centered on a red colored vertex. Figure 16 presents the set $\mathcal{I}$, the rule $r \in \mathcal{R}$ and the corresponding sequence of rules r_l. It also depicts the simulation of the execution of $(\mathcal{R},\mathcal{I})$ on a graph (G,λ,δ). The property P is decided using three steps that successively transform (G,λ,δ) in (G,λ_1,δ) and (G,λ_2,δ). In the depicted example $k = 4$ and $k_1 = 4$. This means that the vertices have to perform a 10-local election before executing the reduction rule r. Due to the size of G, only one vertex is able to win the local election at a given time. This leads to a sequential execution of the reduction algorithm. In fact, parallel reduction steps can only occur if $\Delta(G) \geq 2k+3$. Unfortunately, the graph G presented in Figure 16 has a diameter less than 11. To increase the number of parallel reduction steps, we have to adjust the choice of the parameters k and k_1 to reach a better degree of parallelism.

About the time complexity. Due to the use of unique identifiers in our encoding system, it is possible to elect a vertex v_0 that locally reconstructs the whole graph and execute the sequential reduction algorithms of Arnborg et al. [ACPS93] or Bodlaender et al. [BvAdF01] which yield a linear time complexity. Nevertheless, the election algorithm requires $O(n^2)$ time. Hence, this adds up to a general time complexity of $O(n^2)$. A second possibility consists in executing the reduction algorithm in rounds such that in each round, any vertex v tries, exactly once, to find an executable reduction rule $r \in \mathcal{R}$. This can be reached by using the synchronizers protocols described in Chapter 3. In this case, the worst case time complexity will be at *least* $O(n^2)$. This results to the fact that, in many cases, the number of reduction rules that must be checked seems to grow rapidly. Even if we can check the existence of a match for a given rule r in constant time, the general time complexity is always expressed as $O(Kn^2)$, where $K = |\mathcal{R}|$.

5.3.1 Increasing the degree of parallelism

Each time a vertex v wins the k-local election, it looks for a match in the ball $B_{G^A_\lambda}(v,k+1)$. If we are able to reduce the diameter of $B_{G^A_\lambda}(v,k+1)$, we will also be able to reach a better degree of parallelism during the execution of Algorithm 5.2. Reducing the value of k will scale down the size of the subgraphs of G^A_λ in which matches of reduction rules are sought. Now we

Part A:



Part B:

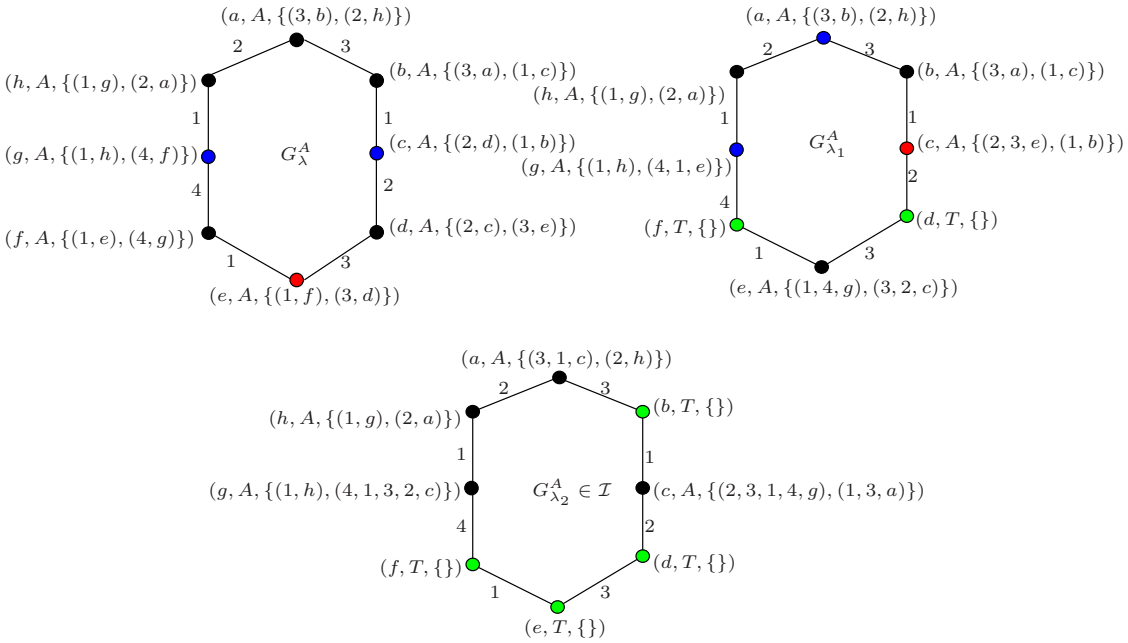


Figure 16: Example: recognizing the property that a graph is a two colorable cycle.

present a way the value of k can be reduced without loosing the properties of our reduction algorithm.

Definition 5.9 (Feasible radius) *Let $(\mathcal{R},\mathcal{I})$ be a reduction system for a property P. For each $r = (H_1, H_2)$ and each $w \in V(H_1)$ we define the feasible radius of w in r, denoted k_w^r, as*

$$k_w^r = min\{k_i \in \mathbb{N} | \forall w_0 \in V(H_1), w_0 \in B_{H_1}(w, k_i)\}.$$

For a given connected graph H, k_w^H is defined in the same way. That is

$$k_w^H = min\{k_i \in \mathbb{N} | \forall w_0 \in V(H), w_0 \in B_H(w, k_i)\}.$$

The feasible radius of a vertex w in a reduction rule $r = (H_1, H_2)$ represents the minimal depth of the tree of minimal paths, rooted at w, that contains all the vertices of H_1. Different vertices of H_1 could have different *feasible radiuses* in r. This is why we are interested in finding the vertex among all vertices of H_1 that has the minimal feasible radius.

Definition 5.10 (Radius) *Let $(\mathcal{R},\mathcal{I})$ be a reduction system for a property P. For each $r = (H_1, H_2)$ we define the radius of r, denoted k^r, as*

$$k^r = min\{k_i \in \mathbb{N} | \exists w_0 \in V(H_1), k_{w_0}^r = k_i\}.$$

The radius of a connected graph $I_1 \in \mathcal{I}$, denoted k^{I_1}, is also defined as

$$k^{I_1} = min\{k_i \in \mathbb{N} | \exists w_0 \in V(I_1), k_{w_0}^{I_1} = k_i\}.$$

Definition 5.11 (feasible radius of a system) *Let $S = (\mathcal{R},\mathcal{I})$ be a reduction system for a property P. The feasible radius of the system S is the radius K_S such that*

$$k_{\mathcal{R}} = max\{k_i \in \mathbb{N} | \exists r \in \mathcal{R} \text{ with } k^r = k_i\}.$$
$$k_{\mathcal{I}} = max\{k_i \in \mathbb{N} | \exists H \in \mathcal{I} \text{ with } k^H = k_i\}.$$
$$K_S = max\{k_{\mathcal{R}}, k_{\mathcal{I}}\}.$$

The value of each k^r with $r = (H_1, H_2) \in \mathcal{R}$ is at most equal to the maximal diameter of H_1. This implies that $k_{\mathcal{R}} \leq i$ with $i = min\{d \in \mathbb{N} | d \geq \Delta(H_1),\ \forall r = (H_1, H_2) \in \mathcal{R}\}$. In the same way, $k_{\mathcal{I}} \leq j$ with $j = min\{d \in \mathbb{N} | d = \Delta(G_i), \forall G_i \in \mathcal{I}\}$. The consequences of all the above definitions induce the following corollary.

Corollary 5.1 *Let k and k_1 be defined as in Algorithm 5.2. Then $K_S \leq k$ and $k_{\mathcal{I}} \leq k_1$.*

Proposition 5.2 *For each rule $r = (H_1, H_2) \in \mathcal{R}$, there is at least one vertex $w \in V(H_1)$ such that any vertex of the graph H_1 is in the ball $B_{H_1}(w, K_S)$.*

Proof. The correctness of this statement is a direct implication of the definitions of K_S and $k_{\mathcal{R}}$. □

We call $\overline{W}_r$ the set of vertices that satisfies the property of Proposition 5.2. In the same way, this property can be extended to any graph $H \in \mathcal{I}$ instead of $r \in \mathcal{R}$. In this case, we will refer the set of vertices w, which contains all vertices of H in their balls $B_H(w, K_S)$, as $\overline{W}_H$. On the basis of these ideas, it is then possible to execute the reduction algorithm using K_S instead

of k. Under these conditions, we could face the situation where, for a given rule $r = (H_1, H_2)$, balls of radius K_S of some vertices can not contain the whole set $V(H_1)$. For instance, in the reduction rule depicted in Part A of Figure 16, $K_S = 2$ and the ball of radius 2 centered on y does not contains the vertices x and w. This means that if there is a subgraph G_1 of G_λ^A isomorphic to H_1 and that contains a W-marked vertex v, it is useless to associate v with the vertex y of H_1. Hence, each time a vertex v tries to find a match of a rule r (or a graph $H \in \mathcal{I}$) in $B_{G_\lambda^A}(v, K_S)$, it has to be associated with the vertices of $\overline{W}_r$ (or $\overline{W}_H$). The degree of parallelism is thereby increased, but the number of vertices of H_1 that could be associated to a W-marked vertex v of G_λ^A are reduced considerably. Such a reduction has an interesting side effect. In fact, it speeds up the solution of the subgraph isomorphism problem as stated in [BM00, Ull76].

Remark 5.2 *Obviously, K_S is the smallest value that ensures that whenever a vertex v wins a K_S-local election, it is possible for v to find a match for any rule $r \in \mathcal{R}$ or for any graph $H \in \mathcal{I}$. As a matter of fact, if we perform our reduction procedure using a k_0-local election with $k_0 < K_S$, then there will always exists a rule $r_0 = (H_1', H_2') \in \mathcal{R}$ or a graph $H_0 \in \mathcal{I}$ such that H_1' or H_0 are not contained in any ball of radius k_0 centered on any vertex v of G_λ^A. This induces the following lemma.*

Lemma 5.8 *Let P be a property and $S = (\mathcal{R}, \mathcal{I})$ a reduction system that characterizes P. Then the use of K_S induces the maximal degree of parallelism in Algorithm 5.2.*

Proof. k-local election with higher values of k yields potential parallel execution of reduction rules on non overlapping balls of radius k. The fact that K_S is the smallest value that allows to execute the reduction algorithm leads in the correctness of the above statement. □

5.3.2 Applications: Decision Problems for Graphs of Bounded Treewidth

A reduction system $(\mathcal{R}, \mathcal{I})$ for a property P corresponds to a polynomial-time algorithm that decides whether property P holds for a given graph G: repeat applying rules from $\mathcal{R}$ starting with the input graph, until no rule from $\mathcal{R}$ can be applied anymore. If the resulting graph belongs to the set $\mathcal{I}$, then P holds for the input graph, otherwise, it does not. During this procedure, the number of reductions that has to be performed is at most n, since each reduction reduces the number of vertices by at least one. In order to obtain a linear-time reduction algorithm for deciding membership in a recognizable set of graphs with treewidth bounded by some known number k, Courcelle et al. [ACPS93] have introduced a new type of reduction systems called *special reduction systems*. This type of reduction systems has the property that for any graph G for which $P(G)$ holds, either G belongs to $\mathcal{I}$, or G contains a match which can be found in an efficient way.

Based on the principle of *d-discoverability*, Bodlaender et al. [BvAdF01] have given an equivalent definition of special reduction system. Their definition uses the *bounded adjacency list search method* introduced in [BH98].

Definition 5.12 (bodlaender et al. [BvAdF01]) *Let d be a positive integer. Let G be a graph given by some adjacency list representation and let G_1 be an i sourced graph. G_1 is said to be d-discoverable in G if*

1. *G_1 is open and connected, and the maximum degree of any vertex in G_1 is at most d,*
2. *there is an i-sourced graph G_2 such that $G = G_1 \oplus G_2$, and*
3. *G_1 contains an inner vertex v such that for all vertices $w \in V(G_1)$ there is a walk W in G_1 with $W = (u_1, u_2, ..., u_s)$, $v = u_1$, $w = u_s$, and for each i, $2 \leq i \leq s-1$, in the adjacency list of u_i in G, the edges $\{u_{i-1}, u_i\}$ and $\{u_i, u_{i+1}\}$ have distance at most d.*

From the above definition, Bodlaender and De Fluiter have proved the next lemma and stated that if we have a special reduction system for a property P, then we have a sequential algorithm which decides P, on connected graphs, in $O(n)$ time and space.

Lemma 5.9 ([BvAdF01]) *Let v be a vertex in G. If v is the inner vertex of some d-discoverable match G_1 to a rule $r = (H_1, H_2)$, then such a match can be found from v in an amount of time that only depends on the integer d and the size of G_1, but not on the size of the graph G.*

Definition 5.13 (Special Reduction System [BvAdF01]) *Let P be a graph property and $(\mathcal{R}, \mathcal{I})$ a reduction system for P. Let n_{max} be the maximum number of vertices in any left-hand side of a rule $r \in \mathcal{R}$. $(\mathcal{R}, \mathcal{I})$ is a special reduction system for P is we know positive integers n_{min} and d, $n_{min} \leq n_{max} \leq d$, such that the following hold.*

- *For each reduction rule $(H_1, H_2) \in \mathcal{R}$, H_1 and H_2 are open and connected.*
- *For each connected graph G and each adjacency list representation of G, if $P(G)$ holds and G has at least n_{min} vertices, then G contains a d-discoverable match.*

Example 5.1 (Special reduction system) *consider the graph property described in Figure 16. It can easily be seen that $(\mathcal{R}, \mathcal{I})$ is a special reduction system for P with $d = n_{min} = n_{max} =$ 5.*

Definition 5.14 *For each integer $k \geq 1$, let TW_k be the graph property defined as follows: for each graph G, $TW_k(G)$ holds if and only if $tw(G) \leq k$. Furthermore, for a property P and an integer k, we define the property P_k as $P_k(G) = P(G) \wedge TW_k(G)$. The property $TW_k(G)$ is satisfied if and only if the treewidth of G is at most k.*

Definition 5.15 (Finite index) *Let P be a graph property, and l a non-negative integer. For l-sourced graphs G_1 and G_2, we define the equivalence relation $\sim_{P,l}$ as follows:*

$$G_1 \sim_{P,l} G_2 \Leftrightarrow \text{for all } l\text{-sourced graphs } H : P(G_1 \oplus H) \Leftrightarrow P(G_2 \oplus H).$$

property P is of finite index if for all $l \geq 0$, $\sim_{P,l}$ has finitely many equivalence classes.

Lemma 5.10 ([Cou90b]) *All MS-definable graph properties are of finite index.*

There are many equivalent terms for a graph class of which the corresponding graph property is of finite index: such a graph class is *recognizable*, [Cou90b], *finite state* or *fully cutset regular*, [AF91], or *regular* [BLW87]. The equivalence has been shown by Courcelle and Lagergren [CL96]. Throughout the rest of this chapter we will use the term *recognizable* for this kind of graph classes.

Lemma 5.11 (Arnborg et al. [ACPS93]) *Every recognizable set of graphs L of bounded treewidth can be defined by a special reduction system.*

From the above lemma we deduce the following theorem.

Theorem 5.1 *Let P be an MS-definable graph property. Every special reduction system that characterizes P_k, for $(k \geq 1)$, is a handy reduction system.*

Proof. It is deduced from Lemma 5.10 and Lemma 5.11 that there is a special reduction system S that characterizes P_k. We now assume that $S = (\mathcal{R}, \mathcal{I})$. Without loss of generality, let $r = (H_1, H_2) \in \mathcal{R}$ be a reduction rule. From the above definition we know that H_1 and H_2 are open and connected. Furthermore, $\mathcal{R}$ is decreasing. Thus $|V(H_2)| < |V(H_1)|$. We know that $\mathcal{I} \subseteq L$ (see Lemma 5.11). That is, all the elements of $\mathcal{I}$ have bounded treewidth. This means that they are connected. Thus, r is a handy reduction rule. □

Courcelle [Cou90b] has given a large class of graph properties which are recognizable, namely the class of properties that are MS-definable. There are many (even NP-complete) decision problems which are MS-definable (i.e., the corresponding properties are MS-definable). These include *Hamiltonian Circuit* and (for fixed k) *k-Colorability*. A detailed list of such properties was presented by Arnborg et al. [ALS91]. Lemma 5.11 and Proposition 5.1 now immediately imply the following result.

Corollary 5.2 *P is a MS-definable graph property. Then there is an algorithm, encoded by means of local computations, which decides P on every bounded treewidth graph.*

5.3.3 Labeled graphs recognizable by local computations

Godard, Métivier and Muscholl [GMM04] have stated necessary conditions for recognizability with structural knowledge in the local computations environment. In their framework, they say that a class $\mathfrak{F}$ is recognizable if there exists some locally generated relabeling relation such that starting from any labeled graph $\boldsymbol{G}$ some final labeling can be reached, under the condition that it is possible to decide whether $\boldsymbol{G}$ belongs to $\mathfrak{F}$ or not. They were specially interested in recognizing labeled graphs which have a certain structural knowledge encoded in the initial labeling.

Let $\boldsymbol{G} = (G, \lambda)$ be a labeled graph and let α be a label. Then Λ_α is the uniform labeling on $\boldsymbol{G}$ with label α, that is, every vertex v is labeled by the pair $(\alpha, \lambda(v))$, the labels of the edges remain unchanged. The labeled graph $\boldsymbol{G}$ has some structural information encoded by its labels (a distinguished vertex, identities) and α can encode some structural properties of the graph, like the size or a bound of the diameter of the graph. They define recognition with structural knowledge in the following way.

Definition 5.16 (Graph recognizer) *A nondeterministic labeled graph recognizer with structural knowledge ι is a triple $(\mathcal{R}, \mathcal{K}, \iota)$ where $\mathcal{R}$ is a locally generated relabeling relation that is noetherian on the set $\{(G, \Lambda_{\iota(\boldsymbol{G})}) | \boldsymbol{G} \in \mathcal{G}_L\}$, $\mathcal{K}$ is the final condition (i.e., there is a recursive set K of finite subsets of L such that $\mathcal{K} = \{\boldsymbol{G} \in \mathcal{G}_L | lab(\boldsymbol{G}) \in K\}$), and ι is a computable function which associates with each labeled graph $\boldsymbol{G}$ a label $\iota(\boldsymbol{G}) \in L$.*

Fact 5.3 states the necessary condition for recognizability.

Fact 5.3 *A labeled graph $\boldsymbol{G}$ is recognized by $(\mathcal{R}, \mathcal{K}, \iota)$ if $Irred_{\mathcal{R}}((G, \Lambda_{\iota(\boldsymbol{G})})) \cap \mathcal{K} \neq \emptyset$.*

Definition 5.17 (Graph recognizer with knowledge) *A deterministic labeled graph recognizer with structural knowledge ι satisfies, in addition to Definition 5.16 for each $\boldsymbol{G} \in \mathcal{G}_L$, either $Irred_{\mathcal{R}}((G, \Lambda_{\iota(\boldsymbol{G})})) \cap \mathcal{K} = \emptyset$ or $Irred_{\mathcal{R}}((G, \Lambda_{\iota(\boldsymbol{G})})) \subseteq \mathcal{K}$.*

For this kind of recognizer, a labeled graph $\boldsymbol{G}$ is recognized if and only if $\text{Irred}_{\mathcal{R}}(\boldsymbol{G}) \subseteq \mathcal{K}$. A class $\mathcal{F}$ of labeled graphs is *recognizable with structural knowledge* ι (or, informally, recognizable knowing ι) if there exists a labeled graph recognizer with structural knowledge $(\mathcal{R}, \mathcal{K}, \iota)$ such that the set of labeled graphs that are recognized by $(\mathcal{R}, \mathcal{K}, \iota)$ is $\mathcal{F}$. Some examples of structural knowledge are given below:

- The diameter $\Delta(G)$ of the graph.
- The topology of the graph (e.g., the adjacency matrix is given).
- The size of the graph, i.e., the number of vertices $|V(G)|$. In this case a class of graphs is said to be *size recognizable*.

Proposition 5.3 (Godard et al. [GMM04]) *The class of covering-minimal graphs is size recognizable.*

All the reduction algorithms we stated in the previous sections of this chapter were dedicated to the local computations framework. Thus, any reduction system $(\mathcal{R}, \mathcal{I})$ for a property P can also be seen as a labeled graph recognizer with structural knowledge. In this case, each vertex is aware of the existence of unique identifiers. This induces the next lemma.

Lemma 5.12 *Let $(\mathcal{R}, \mathcal{I})$ be a handy reduction system that characterizes property P. Then there exists a labeled graph recognizer with structural knowledge $(\mathcal{R}^*, \mathcal{K}, \iota)$ that recognizes the class of labeled graphs satisfying P.*

Proof. Let λ be a relabeling relation that satisfies the conditions of our encoding system as introduced in Section 5.2.1. Furthermore, assume $\mathcal{R}^* = (L, I, P^*)$ where the sets L and I are defined as in Section 5.2.1. The set P^* and $\mathcal{K}$ can be constructed in the following way.

- For each $r = (H_1, H_2) \in \mathcal{R}$ we add the relabeling rule (H_1, λ, λ') to P^*, where λ' is obtained from λ after executing the sequence r_l on H_1.
- $\mathcal{K}$ contains the set $\mathfrak{M}$ of graphs g such that there are graphs g_0 and (g_1, λ, δ) satisfying:
  - $g_0 \in \mathcal{I}$,
  - the underlying graph of g_1 is a subgraph of g,
  - g_0 is isomorphic to the underlying graph of g_1 (with respect to the labeling function λ) and
  - for all vertices $w \in V(g)$, if $w \notin V(g_1)$ then w is in the *tunneling* state. That is, w is no more active in the reduction procedure.

In this construction ι represents the existence of unique identities in the graph. The so constructed recognizer can be seen as a simulation of the reduction algorithm computed by

the reduction system $(\mathcal{R}, \mathcal{I})$. Thus, it is obvious to show that the obtained labeled graph recognizer correctly recognizes the class of graphs satisfying property P. □
From the above statements we derive the following lemma.

Lemma 5.13 *Let P be a graph property characterized by a handy reduction system $(\mathcal{R}, \mathcal{I})$. Assume $\mathcal{F}$ is a class of covering minimal graphs which satisfy P_k, for $k \geq 1$. Then $\mathcal{F}$ is size recognizable.*

Proof. Given an input graph G and the size $|V(G)|$ of G. From Proposition 5.3 we can check if G is covering-minimal or not. Assume G is covering-minimal. Using Lemma 2.1 we can elect a vertex v_0 in G that assigns to each vertex of G a unique identifier. Henceforth, Lemma 5.12 is used to construct the labeled graph recognizer $(\mathcal{R}^*, \mathcal{K}, \iota)$ for the class $\mathcal{F}$. In this case, ι computes the size of G. □
An immediate consequence of Lemma 5.13 concerns the recognition of MS-definable properties of graphs of bounded treewidth.

Corollary 5.3 *Let P be a MS-definable property. Assume $\mathcal{F}$ is a class of covering minimal graphs which satisfy P_k, for $k \geq 1$. Then $\mathcal{F}$ is size recognizable.*

Proof. The proof is obvious using Proposition 5.1 and Lemma 5.13. □

Remark 5.3 *It would also be interesting to ask the question about the possibility to construct a handy reduction system $(\mathcal{R}, \mathcal{I})$ starting from a labeled graph recognizer $(\mathcal{R}^*, \mathcal{K}, \iota)$. Obviously, such a construction is addicted to the characteristics of the graph recognizer $(\mathcal{R}^*, \mathcal{K}, \iota)$. Let $(\mathcal{R}^*, \mathcal{K}, \iota)$ be a labeled graph recognizer with structural knowledge satisfying*

- *the property recognized by $(\mathcal{R}^*, \mathcal{K}, \iota)$ is size recognizable,*
- *$\mathcal{R}^*$ is a locally generated relabeling relation that is noetherian on the set $\{(G, \Lambda_{\iota(\boldsymbol{G})}) | \boldsymbol{G} \in \mathcal{G}_L\}$, $\mathcal{K}$ with L being a two elements set: $L = \{l_1, l_2\}$.*

If we assume, without loss of generality, that

- *initially, all l_1-labeled components or all l_2-labeled components are connected,*
- *each relabeling never transforms an l_1-labeled graph component to an l_2-labeled one,*
- *after a relabeling step, all l_2-labeled components form a connected subgraph of the input graph,*
- *each irreducible graph G^* contains at least one l_2-labeled vertex.*

Then such a recognizer can be computed by a handy reduction system $(\mathcal{R}, \mathcal{I})$ where the fact of changing an l_2-labeled component to an l_1-labeled one is seen as a removal of this component. Hence, the rules contained in $\mathcal{R}^*$ can be easily transformed in handy reduction rules and the set $\mathcal{I}$ is built from $\mathcal{K}$ by removing all l_1-labeled components from the graphs contained in $\mathcal{K}$. Note that if $|L| > 2$, then it is not easy to construct a handy reduction system without changing the basics of the algorithm computed by $(\mathcal{R}^*, \mathcal{K}, \iota)$.

5.4 Unfolding Reduction Rules

In the previous sections we have seen how the underlying graph G_λ^A of a labeled graph (G, λ, δ) can be reduced to a smaller underlying graph $G_{\lambda'}^A$ of (G, λ', δ) using a set $\mathcal{R}$ of handy reduction rules. The labeling relation λ is stated as in Section 5.2.1. We are now interested in presenting a set $\mathfrak{R}_u$ of corresponding *unfolding rules* that satisfies $(G, \lambda', \delta) \xrightarrow{\mathfrak{R}_u} (G, \lambda, \delta)$.

Definition 5.18 (Unfolding rule) *The unfolding rule r_u of a handy reduction rule $r = (H_1, H_2) \in \mathcal{R}$ is the relabeling rule defined by (H_1, λ', λ).*

From now on we present the way the graph (G, λ, δ) can be reached from the graph (G, λ', δ) by executing rules of $\mathfrak{R}_u$ in a distributed way.

To achieve the operation $(G, \lambda', \delta) \xrightarrow{\mathfrak{R}_u} (G, \lambda, \delta)$ one has to take some preventive measures during the reduction of $G, \lambda, \delta)$. In fact, it is necessary for each vertex to have a global knowledge about the reduction rules it has performed during the reduction procedure. For this reason, we enhance the label of any vertex v with an ordered list $\mathfrak{L}_v$ containing the unfolding rules of all rules $r \in \mathcal{R}$ that were executed by v. That is, after executing a rule $r = (H_1, H_2) \in \mathcal{R}$ vertex v stores $(H_1', \lambda', \lambda)$ in $\mathfrak{L}_v$ where the underlying graph of H_1' is the match of H_1 in $B_{G_\lambda^A}(v, k)$ and each vertex of $V(H_1')$ corresponds to exactly one identifier that appears in $B_{G_\lambda^A}(v, k)$. Thus,

$$\mathfrak{R}_u = \bigcup_{v \in V(G)} \mathfrak{L}_v.$$

On the other hand, only A-labeled vertices are considered to be active in the execution of Algorithm 5.1. This means that it is imperative for the unfolding procedure that the graph (G, λ', δ) contains at least one A-labeled vertex. Subsequently, the distributed execution of unfolding rules becomes similar to the procedure described in Algorithm 5.1. This procedure is depicted in Algorithm 5.3.

Fact 5.4 *The execution of an unfolding rule r_u exactly corresponds to the reverse execution of the corresponding reduction rule r.*

Due to the definition of unfolding rules and to the evident similarity between Algorithm 5.3 and Algorithm 5.1, the properties of the last named algorithm are summarized in the next proposition.

Proposition 5.4

A: *If a reduction rule r was executed by a vertex v, then the unfolding rule r_u will be executed by v during the execution of Algorithm 5.3.*

B: *Let r^i, r^j be two reduction rules performed by vertex v such that r^j was executed after r^i. Then Algorithm 5.3 will execute r_u^j before executing r_u^i.*

C: *If two reduction rules r^i, r^j were executed on a subgraph (G_0, λ, δ) of (G, λ, δ), then the corresponding unfolding rules r_u^i and r_u^j will be performed respecting the execution order of r^i and r^j.*

Algorithm 5.3 Undoing reduction rules

Input: A graph (G, λ', δ) and a set $\mathfrak{R}_u$ of unfolding rules .
For all vertices v of (G, λ', δ), $\lambda(v) = (S, \mathcal{P}_v, \mathfrak{L}_v)$ with $S \in \{A, T\}$.
Each A-labeled vertex v executes the following actions.
Step 0:
if $\mathfrak{L}_v = \emptyset$, **then**
  v stops the computation.
else
  the procedure continues in Step 1.
end if
Step 1: v performs a $(2k+2)$-local election. All the winners become W-marked and the other L-marked.
Step 2: each W-marked vertex v takes the next rule $r_u \in \mathfrak{L}_v$ and checks if r_u can be applied on $B_{G^A_{\lambda'}}(v, k+1)$.
if r_u can be applied, **then**
  the procedure continues in Step 3.
else
  go to Step 0.
end if
Step 3: r_u is removed from $\mathfrak{L}_v$ and performed on $(B_{G^A_{\lambda'}}(v, k+1), \lambda', \delta)$ and the algorithm continues in Step 1.

Proof. The first two statements are immediate consequences of the definition of $\mathfrak{L}$ and of the fact that the probability for each vertex v to win the $(2k+2)$-local election is different from zero. We now turn our attention to the last statement.

Assume that r^i and r^j were performed on two non overlapping balls of radius $k+1$ centered respectively on vertices w_i and w_j. The statements A and B imply that r^i_u and r^j_u will be performed in any order without troubles. Suppose now that r^i and r^j were performed by two different vertices w_i and w_j on two overlapping balls of radius $k+1$. Without loss of generality, let u_0 be a vertex that is contained in both balls. The state changes of u_0 are done in a sequential way. Thus, if r^i was performed on u_0 before r^j, then r^i_u will not be performed until r^j_u is done. □

From the above proposition it is granted that Algorithm 5.3 never falls in a deadlock and that unfolding rules that can not be executed in parallel will be executed in the reverse order of the execution of the corresponding reduction rules. This implies the correctness of the following fact.

Fact 5.5 *Algorithm 5.3 correctly performs the unfolding rules.*

Now we turn our attention to the presentation of a procedure that, using unfolding rules, executes *constructive reduction algorithms* in a distributed way.

5.4.1 Constructive Reduction Algorithms

Bodlaender et al. [BvAdF01] have extended the notion of reduction algorithms to *constructive* reduction algorithms, which can be used to construct solutions for decision problems.

The basic idea of constructive reduction algorithm is the following. The algorithm consists of two parts. In the first part, an ordinary reduction algorithm is applied. The reduced graph is then passed to the second part. In this part, a solution is constructed for the reduced graph, if it exists. After that, the reductions that are applied in Part 1 are undone one by one in reverse order, and each time a reduction is undone, the solution of the graph is adapted to a solution of the new graph. This results in a solution of the input graph. Constructive reduction rules are dedicated to graph properties of the form

$$P(G) = \text{“there is an } S \in D(G) \text{ for which } Q(G,S) \text{ holds”}.$$

where $D(G)$ is a *solution domain* (or shortly domain), which is some set depending on G, and Q is an *extended* graph property of G; i.e., $Q(G,S) \in \{\text{true}, \text{false}\}$ for all graphs G and all $S \in D(G)$. An $S \in D(G)$ for which $D(G)$ holds is called a *solution* for G. If a graph property P is of the form “$P(G) =$ there is an $S \in D(G)$ for which $Q(G,S)$ holds”, then P is called a *construction property* defined by the pair (D,Q).

Now we introduce constructive reduction algorithms which for a construction property P defined by (D,Q); not only decide P, but if P holds for an input graph G, also construct an $S \in D(G)$ for which $Q(G,S)$ holds.

Definition 5.19 (Constructive Reduction System) . *Let P be a construction property defined by (D,Q). A constructive reduction system for P is a quadruple $(\mathcal{R},\mathcal{I},\mathcal{A}_\mathcal{R},\mathcal{A}_\mathcal{I})$, where*

- *$(\mathcal{R},\mathcal{I})$ is a reduction system for P,*
- *$A_\mathcal{R}$ is an algorithm which, given*
  - *a reduction rule $r = (H_1, H_2) \in \mathcal{R}$,*
  - *two terminal graphs G_1 and G_2, such that G_1 is isomorphic to H_1 and G_2 is isomorphic to H_2,*
  - *a graph G with $G = G_2 \oplus H$ for some H, and*
  - *an $S \in D(G)$ for which $Q(G,S)$ holds,*

  computes an $S' \in D(G_1 \oplus H)$ such that $Q(G_1 \oplus H, S')$ holds,
- *$A_\mathcal{I}$ is an algorithm which , given a graph G which is isomorphic to some $H \in \mathcal{I}$, computes an $S \in D(G)$ for which $Q(G,S)$ holds.*

Algorithm $A_\mathcal{I}$ in a constructive reduction system $(\mathcal{R},\mathcal{I},\mathcal{A}_\mathcal{R},\mathcal{A}_\mathcal{I})$ is used to construct an initial solution of the reduced graph G, if $G \in \mathcal{I}$. Algorithm $A_\mathcal{R}$ is used to reconstruct a solution, each time a reduction is undone on the graph.

Definition 5.20 *Let P be a construction property defined by (D,Q). A constructive reduction system $(\mathcal{R},\mathcal{I},\mathcal{A}_\mathcal{R},\mathcal{A}_\mathcal{I})$ for P is a special constructive reduction system for P if*

1. *$(\mathcal{R},\mathcal{I})$ is a special reduction system for P (see Definition 5.13), and*
2. *algorithms $A_\mathcal{R}$ and $A_\mathcal{I}$ run in $O(1)$ time.*

It is clear that one way to obtain an Algorithm $A_{\mathcal{R}}$ in a constructive reduction system which runs in $O(1)$ time is to ensure that $A_{\mathcal{R}}$ only has to change a solution locally. That is, the solution to be constructed only differs from the input solution in the part of the graph that was involved in the reduction. Special constructive reduction systems that satisfy this property will be called *locally constructive systems*.

Let P be a constructive property defined by (D, Q) and let $(\mathcal{R}, \mathcal{I}, \mathcal{A}_{\mathcal{R}}, \mathcal{A}_{\mathcal{I}})$ be a locally constructive system for P. Algorithm 5.4 computes for a given graph G a solution for G if one exists.

Algorithm 5.4 Distributed computation of constructive reduction systems

Input: A graph (G, λ, δ), a locally constructive system $(\mathcal{R}, \mathcal{I}, \mathcal{A}_{\mathcal{R}}, \mathcal{A}_{\mathcal{I}})$ and an empty set $\mathfrak{R}_u$ of unfolding rules.
For all vertices v of (G, λ, δ), $\lambda(v) = (S, \mathcal{P}_v, \mathfrak{L}_v)$ with $S \in \{A, T\}$.

(∗Part 1∗)

if a vertex v is A-labeled, **then**
  v execute Algorithm 5.1 and constructs $\mathfrak{L}_v$.
  if $P(G)$ holds, **then**
    a *True* signal is sent to all A-labeled vertices.
  else
    a *false* signal is sent to all vertices
    the procedure stops.
  end if
end if

(∗Part 2∗)

Any A-labeled vertex v that has received the *True* signal performs the remaining steps:
Step 1: v performs a $(2k+2)$-local election. All the winners become W-marked and the other L-marked.
Step 2: each W-marked vertex v takes the next rule $r_u \in \mathfrak{L}_v$ and checks if r_u can be applied on $B_{G^A_\lambda}(v, k+1)$.
if r_u can be applied, **then**
  go to Step 3.
else
  go to Step 1.
end if
Step 3: $(B_{G^A_\lambda}(v, k+1), \lambda, \delta)$ is transformed to $(B_{G^A_\lambda}(v, k+1), \lambda', \delta)$ by the execution of r_u. r_u is removed from $\mathfrak{L}_v$. Thereafter, $A_{\mathcal{R}}$ is used to construct a new solution S for which $Q(B_{G^A_{\lambda'}}(v, k+1), S)$ holds and the algorithm continues in Step 1.

Fact 5.6 *The correctness of Algorithm 5.4 can be deduced from Lemma 5.5 and from the definition of constructive reduction systems.*

If we restrict our attention to graphs of bounded treewidth, we can see that constructive

reduction systems can be used for a large class of construction properties. For this purpose, we require solution domains to be of specific form: for a graph G, there is a t with $D(G) = D_1(G) \times D_2(G) \times \cdots \times D_t(G)$, where each $D_i(G)$ $(1 \leq i \leq t)$ is either $V, E, \mathcal{P}(V(G))$, or $\mathcal{P}(E(G))$. If D is of this form, we say that D is a *t-vertex-edge-tuple* or, if t is not important, a *vertex-edge-tuple*. $\mathcal{P}(V(G))$ and $\mathcal{P}(E(G))$ are respectively the power set of V and E.

As an important case, we consider the MS-definable construction properties. The construction properties defined by (D, Q), where D is a *vertex-edge-tuple* and Q is an MS-definable extended graph property, correspond exactly to the MS-definable construction problems (Arnborg et al. [ALS91]).These problems have the particularity that they can be solved in $O(n)$ time and space for graphs of bounded treewidth if a tree decomposition of bounded width is given for the input graph.

Theorem 5.2 ([BvAdF01]) *Let P be a construction property defined by (D, Q), where D is a vertex-edge-tuple and Q is MS-definable. For each $k \geq 1$ there is a special constructive reduction system for P_k, which can be effectively constructed if a definition of Q in MSOL is known.*

An immediate consequence of Theorem 5.2 is that if the constructed special constructive reduction system is also a locally special constructive reduction system, then P_k can also be decided and constructed using Algorithm 5.4. This implies the correctness of the next theorem.

Theorem 5.3 *Let P be a construction property defined by (D, Q), where D is a vertex-edge-tuple and Q is MS-definable. Then there exists a distributed algorithm encoded by means of local computations that decides P_k. If there is a locally special constructive system for P_k, then an $S \in D$ for which $Q(G, S)$ holds can be constructed.*

5.5 Concluding Remarks

In this chapter, we have presented a general framework that enables the execution of reduction algorithms in a distributed environment. From this framework we have presented an algorithm, encoded by local computations, that decides whether a property holds for a given graph or not. This has helped us to define a new sort of reduction systems, called *handy reduction systems*, and to establish a direct relationship between *labeled graph recognizers with structural knowledge* [GMM04] and handy reduction systems. In the same way, we were able to show that any MS-definable property on graphs of bounded treewidth can effectively be recognized using specific handy reduction systems.

The approach described here has also been adapted to the distributed computation of constructive reduction systems. That is, for a certain class of decision problems on graphs, our framework has the power of simultaneously solving a decision problem and computing a corresponding solution that reinforces the validity of the result of the decision problem. All the computations are done in a fully distributed way.

In further researches, we expect to extend the methodology presented in this work to the distributed computation of *reduction-counter rule* as described in [BvAdF01]. This should help us to develop distributed procedures for solving classes of optimization problems such as *Max Independent Set*, *Hamiltonian Circuit* and *k-Colorability* for fixed k.

Another interesting problem is to determine the class of graph properties characterized by handy reduction systems. The property to have maximum degree at most some fixed constant k is an example of a property that is characterized by a handy reduction system and that has yes-instances of unbounded treewidth.

Chapter 6

Checking Properties in Distributed Systems

Contents

6.1 Introduction

The detection of global conditions is a fundamental problem in an asynchronous distributed system. In such a system, a process can not know the state of other processes at any given time due to communication delays. This makes it difficult to detect properties, or predicates, spread across the system. Thus, computing a global predicate or function, a need that occurs frequently in many distributed systems, typically requires significant programming.

Being able to observe a distributed computation is useful for many fundamental problems in distributed software, such as debugging, testing, and fault-tolerance. After a program is debugged and tested, it must be monitored for fault-tolerance, this also requires something that will observe the global state. Finally, the ability to observe global predicates generalizes algorithms for many other problems such as detecting program termination, token loss, and deadlock.

Research on how to detect global predicates has yielded three sets of algorithms. In the *global snapshot* algorithms [Mat93], global snap-shots of the computation are repeatedly computed until the desired predicate becomes true. However, this approach works only for

stable predicates like deadlock and termination, which do not turn false once they become true.

In the second set of algorithms, a lattice of global states is constructed. Unlike the global snapshot approach, this approach lets users detect unstable predicates [CM91]. Nevertheless, this can mean exploring a prohibitive number of global states. Garg and Waldecker, Chandra and Kshemkalyani [CK03, GM97, GW92] introduced a third approach which exploits the structure of the predicate, but does not build a lattice. Instead, the computation itself is examined to deduce if a predicate became true or not. These algorithms are computationally efficient and can be used to detect even unstable predicates.

In this chapter, we present a procedure that is able to check stable and unstable properties in a fully distributed environment. To this end, using the global snapshot algorithm is not very helpful for our purpose. Although the third approach has some complexity advantages, its application in the case of local computations is problematic. First of all, the predicates we expect to check do not always contain variables indexed by process identifiers. These predicates could also contain existential quantifiers as well as universal quantifiers. Furthermore, the networks we deal with are, in general, anonymous. For these reasons, we have improved the approach of Cooper and Marzullo [CM91] and stated an algorithm that does not necessary need to build a lattice of all global states and is able to check the validity of any type of given properties in an acceptable time and message complexity. The approach presented here has the particularity that it also works in systems whose models are equivalent to the standard distributed system model.

6.2 Properties Description and Computation Sequences

The *local state* of a process p_i is defined as the union of the state of p_i with the states of all its incident edges. A *global state* is a collection of local states, one from each process. A history of a distributed system can be modeled as a sequence of events in their order of occurrence. Since execution of a particular sequence of events leaves the system in a well-defined *global state*, a history uniquely determines a sequence of *global states* through which the system has passed. Unfortunately, in an asynchronous distributed system, no process can determine the sequence of *global states* through which the system has passed. In absence of a global time, it is also tedious to classify all computation events in their order of occurrence. This problem is due to the existence of causally independent events, that can be performed simultaneously or in any order.

Definition 6.1 (Coherent sequence) *A coherent computation sequence of a distributed execution is a sorted sequence $\mathcal{S}$ of events such that if an event e occurred (during the computation) before an event e', then e' appears after e in the sequence $\mathcal{S}$.*

Our goal is to sort all computation events to obtain a *coherent computation sequence* of the system. Causally independent events have the particularity that they can be computed in any order or at the same time. Thus, we restrict consistent computation sequences, to coherent sequences that describe the computation of such independent events in a strict order.

Description of Properties.

In opposition to the langauge Lidia (*Language for implementing distributed algorithms* see Chapter 7), other programming languages like IOA [GLV97] take advantage of invariants discovery tools like Daikon [NE01, BE04] to find all invariants and check their validity. All these invariants are related, in the case of IOA, to a specific automata and are therefore *locally checkable*. For our purpose, we hope to check local properties as well as global properties. Moreover, we do not intend to pay attention to all types of properties. Rather, we are only interested in two classes of properties defined as followed.

Definition 6.2 (Class of Quasi-invariants)
A property ψ is a quasi-invariant *if for every consistent observation of the execution, there exists a global state of it in which ψ holds.*

Definition 6.3 (Class of Invariants)
A property ψ is an invariant *if for every consistent observation of the execution, ψ holds in all global states of the execution.*

Our class of quasi-invariant properties corresponds to the class of predicates that satisfy the *Definitely* condition introduced by Cooper and Marzullo in [CM91]. One of the goals we hope to reach in Lidia is to prove the correctness of program executions by requiring the validity of some given predicates. Thus, other classes of properties are not of interest for our purpose.

Unfortunately, most of the invariants we intend to check are not *locally checkable*. We aim also to test the kind of invariants that are described in the following examples.

Example 6.1 *During the computation of a spanning tree (see Lidia-Program 7.1), the set of all $A-$labeled processes and all $1-$labeled edges represents a spanning tree.*

Example 6.2 *During the execution of an election algorithm (Lidia-Program 7.2), there exists at most one process that is in the elected state.*

A process that has only a *local view* of the network, can not test if these predicates are satisfied or not. In fact, testing this kind of invariants requires a global knowledge over the whole network.

Theorem 6.1 (Local checkability) *Let Φ_g be a global property. Φ_g is locally checkable if and only if there exists a* local *formula ψ_i for each process p_i, such that one of the following statements holds:*

(1) $\Phi_g \Longleftrightarrow \forall i \psi_i$.

(2) $\Phi_g \Longleftrightarrow \exists i \neg \psi_i$.

Proof.

Case 1 $\Phi_g \Longleftrightarrow \forall i \psi_i$: If ψ_i is not satisfied for a given process i, then i can detected that Φ_g does not hold and broadcast a corresponding error message.

Case 2 $\Phi_g \Longleftrightarrow \exists i \neg \psi_i$: This is the same as in Case 1 with the small difference that the error message is sent whenever ψ_i holds for a given process i.

□

Definition 6.4 *A strong conjunctive predicates ψ is of the form $\bigwedge_i \phi_i$, where ϕ_i is a predicate defined on variables local to process p_i.*

The first statement of Theorem 6.1 can be seen as a strong conjunctive predicate as described in [CK03]. Checking the correctness of this kind of invariants can be done locally in the following way. We use a local computation rule that is executed after each state transition and issues a warning if the invariant ϕ_i of process p_i fails. This rule should be executed by each process p_i. In the same way, the second condition can also be checked locally.

6.3 A Virtual Time Based Algorithm

We assume an asynchronous distributed system in which n processes communicate by reliable message passing. Messages are delivered by *FIFO* channels. An event structure model $(\mathcal{E}, \prec)$, where $\prec$ is an irreflexive partial ordering representing the causality relation [Lam78] on the event set $\mathcal{E}$, is used as model for the distributed system execution. Three kinds of events are considered: *send*, *receive* and *internal* events. $\mathcal{E}$ is partitioned into local executions at each process. Let V be the set of all processes. Each $\mathcal{E}_i$ is a totally ordered set of *internal* events, executed by process p_i, that have modified the *local state* of p_i. The *local state* of a process p_i is represented by the state of p_i and the states of the edges that are in the ball of radius 1 centered on p_i. We also assume that vector clocks C are available [Fid88, Mat89]. Each process maintains a vector clock C of size $n = |V|$ integers, by using the following rules [Mat89].

(1) Before an internal event at process p_i, p_i executes $C_i[i] = C_i[i] + 1$.

(2) Before a send event at process p_i, the process p_i executes $C_i[i] = C_i[i] + 1$. It then sends the message timestamped by C_i.

(3) When process p_j receives a message with timestamp T from process p_i, it executes the following operations before delivering the message.

Op_1**:** $(\forall k \in [1, ..., n]\ C_j[k] := max(C_j[k], T[k]))$.
Op_2**:** $C_j[j] := C_j[j] + 1$.

The timestamp of an event is defined as the value of the vector clock when the event occurs.

In the above construction each process p_i is equipped with a simple clock $C_i[i]$ which is incremented by 1 each time an event happens. An idealized external observer having immediate access to all local clocks knows at any moment the local times of all processes. An appropriate structure to store this global time knowledge is a vector with one element for each process. Mattern [Mat89] has constructed the above mechanism by which each process gets an *optimal approximation* of this global time. Before going further in the description of our algorithm, we first introduce some theorems and definitions stated in [Mat89].

Definition 6.5 *For two vector clocks vc_1, vc_2*

- $vc_1 \leq vc_2$ *iff* $\forall i : vc_1[i] \leq vc_2[i]$.
- $vc_1 < vc_2$ *iff* $vc_1 \leq vc_2$ *and* $vc_1 \neq vc_2$.
- $vc_1 || vc_2$ *iff* $\neg(vc_1 < vc_2)$ *and* $\neg(vc_2 < vc_1)$.

Theorem 6.2 *(Mattern [Mat89]) At any instant of the physical time we have the relation:*

$$C_i[i] \geq C_j[i] \ \forall i, j$$

Theorem 6.3 *(Mattern [Mat89])* $\forall e_1, e_2 \in \mathcal{E} : e_1 \prec e_2$ *iff* $C(e_2) < C(e_2)$ *and* $e_1 || e_2$ *iff* $C(e_1) || C(e_2)$. *With* $C(e)$ *representing the vector timestamp of the event* e.

Theorem 6.4 *(Mattern [Mat89]) If* $e \in \mathcal{E}$ *occurs at process* p_i *then for any event* $e' \neq e : e \prec e'$ *iff* $C(e)[i] \leq C(e')[i]$.

Looking at their timestamps, we can conclude, from the above definition and theorems, that two events are either independent or that the first occurred once the second was finished. Thus, we are able to build a partial order on $\mathcal{E}$ to compute global states on distributed systems.

In [Mat89] a variant of Chandy-Lamport algorithm for computing snapshots of a distributed system using time vectors is given. Snapshots are very useful for the detection of global properties in distributed systems. The problems with this technique stem from the fact that it is not possible to take snapshots for all global states through which the system passes during the computation. Thus, in the case of testing the validity of invariants during a distributed computation, the use of snapshots can be error-prone.

We restrict our attention to *strong consistent* global states, *i.e.*, global states through which the system has passed during the computation.

Definition 6.6 (Local snapshot) *The local snapshot of a process p_i is the state of p_i and the states of its incident edges.*

We represent the internal event of a process p_i as the *local snapshot* that is taken after the internal event is executed. We now introduce a mechanism that explicitly enumerate all the global states of a computation in a distributed system.

6.3.1 Computing Global States

The algorithm we present here makes use of time vectors to collect the *local snapshots* of all processes. Starting from these *local snapshots*, we will show how the set of all *strong consistent* global snapshots can be efficiently computed. Basically, our procedure works as follows. Each time a process p performs an internal action that changes its *local state*, it takes a timestamped *local snapshot* and stores it in an appropriated data structure. Once process p has terminated the computation, it is ready to be involved in the computation of a spanning tree containing all processes. Thereafter, p exploits the constructed spanning tree to send all the stored events to an elected process P^0. As soon as P^0 has received all the *local snapshots*, it computes all the *strong consistent* snapshots and starts to check the validity of the given properties. Vector

clocks of processes are maintained using the three rules described at the beginning of this section.

For simplicity, we consider here only *off-line* detection of global states, in which the detection algorithm is run after the distributed computation has terminated. We use identities to characterize processes. This identities are helpful to manage the vector clocks. Furthermore, we store the events in a dynamic array, whose element are lists of events. The approach can also be applied to online detection. In this case, the spanning tree is computed at the beginning of the distributed execution and the snapshots can be immediately sent to the elected process. All the basics of our procedure are described in Algorithm 6.1.

Algorithm 6.1 Computing a coherent sequence

Rule 0: At the initialization, a sorted array $\mathcal{A}_i$ of process p_i contains the *local snapshot* s_0^i. s_0^i is therefore timestamped with the vector $[0, ..., 0]^T$ of size N.

{**Snapshots construction**}

Rule 1: When a process p_i performs an internal action that changes its *local state*, it takes a *local snapshot* s_i and stores it in the array $\mathcal{A}_i$. The stored s_i is timestamped with the value of the vector clock of p_i when the internal action occurred.

{**Spanning tree computation**}

Rule 2: Once process p_i has terminated the distributed computation, p_i becomes *free* and can be involved in the construction of a spanning tree.

{**Collecting the snapshots**}

Rule 3: If a *free* process p_i is a leaf (in the constructed spanning tree) and its *father process* is *free*, it sends its array $\mathcal{A}_i$ to its *father process* and terminates the computation. The father process merges its array with the arrays of its children.

Rule 4: If a *free* process p_i has received and merged the arrays $\mathcal{A}_j$ of all its children processes p_j, then p_i sends, if possible, the resulting array to its father process.

Rule 5: When a process has received all the arrays of its children, and it does not have a father process, it merges all the received arrays and starts to check the validity of the given invariants.

Proposition 6.1 *Algorithm 6.1 correctly computes a coherent sequence containing all snapshots generated by all the processes.*

Proof. In this proof we assume that our merging procedure is faithful. This assumption will be proved in the next section.

Let s_i be the local snapshots of any process p_j that executes Algorithm 6.1. After executing *Rule 2*, the elements of the array A_j are in a total sorted order $s^0 \prec s_1 \prec s_2 \prec s_3 \prec ... \prec s_l$ with l being the maximum number of events that have been stored by p_j. The principle used in Rule 3, Rule 4 and Rule 5 to collect the snapshots consists in sending each snapshot bottom up until it arrives by P^0. Snapshots of processes that have a common father process will be merged before arriving at P^0. Thus, after the execution of all rules, process P^0 is

expected to have compute a coherent array $\mathcal{A}^0$ that contains all the local snapshots of all processes. In the next sections we will see how coherent arrays can be merged and how strong consistent global states can be enumerate. □

Due to Theorem 6.3 and Theorem 6.4, the elements of $\mathcal{A}^0$ are in a partial sorted order

$$s_0^0, ..., s_0^{n-1} \prec s_0 \prec ... \prec s_i, ..., s_k \prec s_j \prec ... \prec s_L$$
$$\text{with } i < k < h < L.$$

The events s_0^i are snapshots stored by *Rule 0*. All of them were stamped with the same vector time $[0, 0, ..., 0]^T$ of size N. In this partial order, all events that could be performed simultaneously are in the same list in $\mathcal{A}^0$. That is, they are separated from each other by commas. Such events are said to be *pseudo-parallel*. Moreover, in the off-line version of our algorithm, the spanning tree is computed once. Therefore, we decide to let the vertex indexed by 0 starting the computation. That is, as soon as this vertex becomes *free* for the spanning tree computation, it changes its spanning tree label to A. This avoids loss of time in performing more complicated election to decide which process should starts the construction of the spanning tree.

On the other hand, it is not always easy to merge two arrays that contain *pseudo-parallel* events. Many problems originate from the fact that the relation $||$ is not transitive. Thus, if $s_i||s_j$ and $s_j||s_k$, then the relation $s_i||s_k$ is not always satisfied.

Example 6.3 *Let e_1, e_2 and e_3 be three events such that $C(e_1) = [30, 15]$, $C(e_2) = [1, 16]$ and $C(e_3) = [3, 20]$. From Definition 6.5 and Theorem 6.3 we deduce that $e_2||e_1$ and $e_1||e_3$. Because $e_2 \prec e_3$, we can not deduce that $e_2||e_3$.*

This means that, while merging two arrays of events, we have to take some precautions to avoid incoherences in the end array $\mathcal{A}^0$. To this end, we meet the following requirements in any array $\mathcal{A}$:

R_1**:** If $s_i, s_j \in \mathcal{A}[k] \Rightarrow (s_i||s_j),\ \forall i, j, k \in \mathbb{N}$.

R_2**:** If $s \in \mathcal{A}[i]$, then $\forall s_j \in \mathcal{A}[k] : s \prec s_j \vee (s||s_j), \forall i,\ k \in \mathbb{N}$ with $k > i$.

We then state the following lemma.

Lemma 6.1 *Let $e_i, e_j \in \mathcal{E}$ be two events with $e_i||e_j$. Then*

1. $\forall e \in \mathcal{E} : e_i \prec e \Rightarrow (e_j||e) \vee (e_j \prec e)$
2. $\forall e \in \mathcal{E} : e \prec e_i \Rightarrow (e_j||e) \vee (e \prec e_j)$

Proof.

1. Due to the fact that $C(e_i)[k] < C(e)[k]\ \forall k$ and $\exists k' \in \mathbb{N}$ such that $C(e_i)[k'] \geq C(e_j)[k']$, then $\exists k_0 \in \mathbb{N}$ such that $C(e_j)[k_0] < C(e)[k_0]$. This means that $e \prec e_j$ does not hold.
2. Due to the fact that $C(e_i)[k] > C(e)[k]\ \forall k$ and $\exists k' \in \mathbb{N}$ such that $C(e_i)[k'] \leq C(e_j)[k']$, then $\exists k_0 \in \mathbb{N}$ such that $C(e_j)[k_0] > C(e)[k_0]$. This means that $e_j \prec e$ does not hold.

□

Definition 6.7 (Coherent Array) *An array $\mathcal{A}$ is said to be* coherent *if the requirements R_1 and R_2 hold in $\mathcal{A}$.*

Proposition 6.2 *Let $k_0, k_1 \in \mathbb{N}$, $\mathcal{A}$ be a coherent array and $e_0 \in \mathcal{A}[k_0]$. Then it holds*

1. $k_1 < k_0 \ \wedge \ e_1 \in \mathcal{A}[k_1] \Rightarrow ((e_1 \prec e_0) \vee (e_1 || e_0))$
2. $k_1 > k_0 \ \wedge \ e_1 \in \mathcal{A}[k_1] \Rightarrow ((e_0 \prec e_1) \vee (e_1 || e_0))$

Proof. The proof of this proposition is a straightforwards consequence of the requirement R_2 combined with Lemma 6.1. □

6.3.2 The Merging Procedure

In this section we introduce an algorithm that generates a *coherent* array $\mathcal{A}$ which satisfies the conditions of Definition 6.7. This is reached by the execution of Algorithm 6.2 that takes two dynamic arrays and merges the elements of the first in the second array. In this representation $\mathcal{A}.size()$ denotes the size of the array $\mathcal{A}$.

Fact 6.1 *During the execution of Algorithm 6.2, Step 0 is executed once. For each event s_i of $\mathcal{A}_p$, only one of the next three steps (Step 1, Step 2 or Step 3) is executed.*

Lemma 6.2 *The result of merging two coherent arrays with Algorithm 6.2 is a coherent array.*

Proof. Events are added in $\mathcal{A}_q$ using Step 0, Step 1, Step 2 and Step 3. Obviously, $\mathcal{A}_q^*[0]$ contains all the events timestamped with the vector $[0, 0, ..., 0]^T$. This means that after the execution of Step 0 the resulting array is coherent. During the insertion of s_i, Step 1 is executed until a value k and an event $s_j \in \mathcal{A}_q[k]$ are founded such that $s_i \prec s_j$. Hence, if $k' < k \wedge \exists s_0 \in \mathcal{A}_q[k'] \Rightarrow (s_0 || s_i)$. In the same way, if $k' > k \wedge \exists s_0 \in \mathcal{A}_q[k'] \Rightarrow (s_0 || s_i \vee s_i \prec s_0)$. Due to Proposition 6.2 we deduce that the array resulting from an execution of Step 1 on a coherent array is also coherent. Using the same argument we can proved that the execution of the last two steps also ensures to obtain a coherent array as output. Thus, merging two coherent arrays with Algorithm 6.2 produces a coherent array. □

The array containing the local snapshots of any process P is known to be coherent. In Algorithm 6.1 all these arrays are merged pairwise starting from the leaves of a given spanning tree. From all of this we deduce the following corollary.

Corollary 6.1 (Correctness) *Algorithm 6.1 generates a* coherent sequence *of the global states of a given distributed algorithm.*

6.3.3 Enumerating Strong Consistent Global States

On the basis of array $\mathcal{A}^0$ generated by Algorithm 6.1, it is possible to compute all the global states through which the system has passed during the distributed computation. Due to the

Algorithm 6.2 Merging two coherent arrays

Input: dynamic coherent arrays $\mathcal{A}_p$ and $\mathcal{A}_q$.
Output: array $\mathcal{A}_q^*$ that contains the elements of $\mathcal{A}_p \cup \mathcal{A}_q$.

Step 0:
Insert all elements from $\mathcal{A}_p[0]$ in the list $\mathcal{A}_q[0]$.
$k = 0$.
repeat
  if $\mathcal{A}_p[k] = \emptyset$, **then**
    $k = k + 1$
  end if
  s_i is the smallest element of $\mathcal{A}_p[k]$
  remove s_i from $\mathcal{A}_p[k]$.

  Step 1:
  $d = 0$
  while $(s_i || s_j)\ \forall s_j \in \mathcal{A}_q[d]$ and $d < \mathcal{A}_q.size()$, **do**
    add s_i to $\mathcal{A}_q[d]$,
    $d = d + 1$.
  end while

  Step 2:
  Let d be the smallest index such that $\exists s_j \in \mathcal{A}_q[d]$ and $s_i \prec s_j$.
  if $s_i \in \mathcal{A}_q[d-1]$, **then**
    there is nothing to do.
  else
    increment the size of $\mathcal{A}_q$ by 1,
    for $m = \mathcal{A}_q.size()$ to $m = d + 1$ **do**
      move all elements from $\mathcal{A}_q[m-1]$ to $\mathcal{A}_q[m]$.
    end for
    Add s_i to $\mathcal{A}_q[d]$.
  end if

  Step 3:
  if $s_j \prec s_i$ for $s_j \in \mathcal{A}_q[L]$ with $L = \mathcal{A}_q.size() - 1$, **then**
    the size of $\mathcal{A}_q$ is incremented by 1,
    s_i is added to $\mathcal{A}_q[L+1]$.
  end if
until $\mathcal{A}_p[k] = \emptyset$ and $k = \mathcal{A}_p.size() - 1$.

impossibility to sort *pseudo-parallel* events in a strict order of appearance, we have to make a decision about the global states that are relevant for our purpose. We define $\mathcal{P}^i$ as the set of processes whose snapshots are in $\mathcal{A}^0[i]$.

Definition 6.8 (Idealized global state) *Let $\mathcal{G}^i$ be the global state represented by the list $\mathcal{A}^0[i]$, $i \in \mathbb{N}, 0 \leq i < \mathcal{A}^0.size()$. An idealized global state $\mathcal{G}^{i+1}$ is obtained from $\mathcal{G}^i$ by replacing*

in the global state $\mathcal{G}^i$, the snapshots of the elements of $\mathcal{P}^{i+1}$ through the snapshots stored in $\mathcal{A}^0[i+1]$.

The intuitive idea of *idealized global state* relies on the observation that during the simulation of a distributed computation, all causality independent events can take place simultaneously without generating computation errors or disturbing the execution of the algorithm. Let $\mathcal{G}^i$ be the global state that is represented by the list $\mathcal{A}^0[i],\ i \in \mathbb{N}, 0 \leq i < \mathcal{A}^0.size()$. The entire set of *idealized global states* is generated by the execution of the next procedure.

Procedure 6.1 (Computing idealized global states)

Rule 0: *At real time t_0, the global state of the network is encoded in the values of the local snapshots contained in $\mathcal{A}^0[0]$.*

Rule 1: *Each list $\mathcal{A}^0[i], \forall i \in \mathbb{N}, i > 0$ represents the local snapshots of processes whose local states have changed since the last global state $\mathcal{G}^{i-1}$. We compute the global state $\mathcal{G}^i$ by replacing, in the global state $\mathcal{G}^{i-1}$, the snapshots of the elements of $\mathcal{P}^i$ through the snapshots stored in $\mathcal{A}^0[i]$.*

Remark 6.1 *Obviously, if the replacements depicted in Rule 1 are performed one by one, then Procedure 6.1 will compute exactly $|\mathcal{A}^0[i]|$ different idealized global states for each i.*

Naturally, the ideas introduced in Procedure 6.1 represent a relaxation of the problem consisting in finding all global states trough which a distributed system has really passed. Nevertheless, the informations stored in the coherent array $\mathcal{A}^0$ are enough to perform an exhaustive enumeration of all the global states trough which the system could have passed. This enumeration is obtained using Procedure 6.2.

Procedure 6.2 (Computing all global states)

Rule 0: *At real time t_0, the global state of the network is encoded in the values of the local snapshots contained in $\mathcal{A}^0[0]$.*

Rule 1: *Each list $\mathcal{A}^0[i], \forall i \in \mathbb{N}, i > 0$ represents the local snapshots of processes whose local states have changed since the last global state $\mathcal{G}^{i-1}$. We construct each global state $\mathcal{G}^k,\ i \leq k \leq |\mathcal{A}^0[i]| \times |\mathcal{A}^0[i]|!$, by replacing, in the global state $\mathcal{G}^{i-1}$, the snapshots of the elements of $\mathcal{P}^i$ through the snapshots stored in $\mathcal{A}^0[i]$.*

Remark 6.2 *The replacement sequence used to generate the state $\mathcal{G}^k$ in Procedure 6.2 corresponds to a permutation of the elements of $\mathcal{A}^0[i]$. This means that for each list $\mathcal{A}^0[i]$ we generate exactly $|\mathcal{A}^0[i]| \times |\mathcal{A}^0[i]|!$ different global states.*

For the sake of completeness, once a global state is computed, the validity of any given predicate p can be evaluated in a polynomial time with respect to the length of p. The value of the previous generated global state can be always stored in order to simplify the evaluation of the next global state. Moreover, if we use clever data structures, we can evaluate any predicate p in a time linear to the length of p.

6.3.4 Complexity Analysis

Let n be the number of involved processes in the network, $D = \Delta(G)$ be the diameter of the corresponding graph and m be the maximal number of events stored at each process. To Analyze the message complexity, we consider separately the cost of two procedures. The first one consists of all the send events executed by all processes while sending their local arrays to their father process (see rule 3 of Algorithm 6.1). The second procedure is represented by the message complexity of building a spanning tree in an anonymous network with a distinguished process. We take advantage of the algorithm described in Chapter 1 to construct a spanning tree. Thus, the message complexity of this procedure is $O(n)$. If $D_0 \leq D$ is the diameter of the constructed spanning tree, then the message complexity of the first procedure is $O(mD_0n)$. In the worst-case, this complexity is $O(mDn)$. Thus, the worst-case message complexity of the given algorithm is $O(mDn)$.

The two main components that contribute to the time complexity is the merging procedure and the enumeration of all *idealized global states* through which the system has passed. In the worst case, the merging algorithm merges two arrays of size $\frac{mn}{2}$. Thus, the worst-case time complexity of Procedure 6.1 is given by $O(mn)$. Each list $\mathcal{A}^0[i], 0 \leq i \leq \mathcal{A}^0, i \in \mathbb{N}$ contains at most n events. Thus the time complexity of enumerating all the idealized global states is $O(mn^2)$. The global worst-case time complexity is therefore $O(mn^2)$.

6.4 Concluding Remarks

The correctness of the approach introduced in this chapter depends on the existence of an initially elected (or distinguished) process P^0 and on the computation of a spanning tree that is need to send all the local snapshots to P^0. In fact, the problem of computing a spanning tree is closely related to the election problem. That is, if it is no possible to elect a process P^0 in a given graph G, then the same impossibility result yields for the spanning tree computation. If the processes dispose of some specific knowledge on the network, it is possible to perform a faithful leader election and therefore a spanning tree computation. Godard et al. [GMM00] stated the conditions under which such an election is possible. In [Maz97] Mazurkiewicz gives an election algorithm for the families of graphs which are minimal for the covering relation when the network size is known. For this family of graphs, the approach presented here can obviously be used without assigning to each process a unique identifier.

Due to the fact that each process maintains a vector clock indexed by processes identifiers, all processes are aware of the network size and all the identifiers that exist in the network. Thus, it is always possible to successfully perform a leader election [GM02].

The methodology presented here has been successfully implemented and tested in the ViSiDiA platform. Basically, the implementation works on two levels and use two different kinds of messages. The first type of messages is dedicated to the computation of the election and spanning tree algorithms, whereas The second type of messages only contains the informations that are necessary to execute the distributed algorithm whose properties we intend to check.

If we consider the case where there exists a possibility to synchronize the underlying network [Tel00], then it is useless for a process P to manage a vector clock and to have knowledge of the network size. In fact, in each synchronization round i, a process P performs exactly one computation step s_i. Afterward, s_i could be forwarded to an elected process P^0

using the edges of a preprocessed spanning tree rooted at P^0. The elected process P^0 could be represented by the process that starts the first synchronization round and the computation of a spanning tree could be performed in the same way as done before. Nevertheless, being able to synchronize a network is a task that depends on the network topology and on the initial knowledge of the processes.

Third Part

A programming language for local computations in graphs

Within the scope of our researches, we have developed a programming language for implementing distributed algorithms encoded by local computations. This language, called Lidia, is based on a *two-level* transition system model: the first level is used to specify the behavior of each single component, whereas the second level captures their interactions. Transitions are basically expressed in a *precondition-effect* style where the precondition part is expressed in the logic language $\mathcal{L}^*_\infty$. The language $\mathcal{L}^*_\infty$ is an extension of first-order logic by means of new counting quantifiers and additional computation symbols.

In the first chapter of this part, we will give a formal presentation of Lidia by detailing all the computational aspects inherent to our programming language. An important section of this chapter is devoted to the presentation of the logic $\mathcal{L}^*_\infty$. The use of this logic in the Lidia framework will be illustrated by the hand of two Lidia programs. These programs show how a spanning tree and an election algorithm can be implemented in Lidia.

In the second chapter, we will turn our attention to the computational completeness of Lidia. We will be specially interested in defining the class of distributed problems that can be easily implemented in Lidia. The proofs we will state take advantage of *finite model theory* and of the descriptive complexity of $\mathcal{L}^*_\infty$. All the results presented in this part have been published in [MO04c, MO04d].

Chapter 7

Implementing Local Computations: Lidia

Contents

7.1 Introduction

The growing interest in distributed computing systems has resulted in a large number of languages for programming such systems [BST89]. Many of these languages are oriented towards systems programming and are typically used for distributed operating systems, file servers, and other systems programs. All these distributed systems are required to provide increasingly powerful services, with increasingly strong guarantees of performance, fault-tolerance,

and security. At the same time, the networks in which these systems run are growing larger and becoming less predictable. Hence, distributed systems have become very complex.

The best approach to managing the increased complexity involves organizing systems in structured ways, viewing them at different levels of abstraction and as parallel compositions of interacting components. Such structure makes systems easier to understand, build, maintain, and extend, and can serve as the basis for documentation and analysis. However, in order to be most useful, this structure must rely on a solid mathematical foundation. This is obviously necessary if the structure has to support methods of constructing or analyzing systems.

One reasonable mathematical basis is the concept of local computations. In fact, local computations have been shown to be a suitable tool for encoding distributed algorithms, for proving their correctness and for understanding their power. In this model, a network is represented by a graph whose vertices denote processors, and edges denote communication links. The local state of a processor (resp. link) is encoded by the label attached to the corresponding vertex (resp. edge).

From this solid theoretical basis, we have developed the Lidia[1] programming language that is entirely devoted to expressing and programming distributed algorithms encoded by means of local computations.

7.2 The Computation Model

Our approach consists of defining an operational model for Lidia that is based on a two-level transition system. The first level consists of a number of transition systems, each of which defines the behavior of a single process. The second level consists of a single transition system that defines the interactions among the first-level transition systems. Transition systems, which are structures commonly used in operational semantics, have been used in an uniform and universal way.

A transition system consists of the set of all possible states of the system, the transitions the system can make in this set, and a subset of states in which the system is allowed to start. To avoid the confusion between the states of a single process and the states of the entire algorithm (*the global states*), the latter will from now on be called *configurations*.

Definition 7.1 *A transition system is a triple $S = (\mathcal{C}, \rightarrow, \mathcal{I})$, where $\mathcal{C}$ is a set of configurations, $\rightarrow$ is a binary transition relation on $\mathcal{C}$, and $\mathcal{I}$ is a subset of $\mathcal{C}$ of initial configurations.*

Hence, a transition relation is a subset of $\mathcal{C} \times \mathcal{C}$. Instead of $(\gamma, \delta) \in \rightarrow$ the more convenient notation $\gamma \rightarrow \delta$ is used.

Definition 7.2 *Let $S = (\mathcal{C}, \rightarrow, \mathcal{I})$ be a transition system. An execution of S is a maximal sequence $E = (\gamma_0, \gamma_1, \gamma_2, ...)$, where $\gamma_0 \in \mathcal{I}$, and for all $i \geq 0$, $\gamma_i \rightarrow \gamma_{i+1}$.*

A *terminal* configuration is a configuration γ for which there is no δ such that $\gamma \rightarrow \delta$. Note that a sequence $E = (\gamma_0, \gamma_1, \gamma_2, ...)$ with $\gamma_i \rightarrow \gamma_{i+1}$ for all i, is maximal if it is either infinite or ends in a terminal configuration.

We use a set of first-level transition systems to specify processes as autonomous entities that

[1] Language for Implementing DIstributed Algorithms.

can compute and / or interact with their environment. Thus every step of the computation in such a process may depend not only on the internal state of the process, but also on some input it may obtain from its environment. These processes are open systems in a sense analogous to Wegner's notion of Interaction Machines [Weg98]. Typically, each transition system is unbounded and nondeterministic, reflecting the fact that the process it represents is an interactive system; i.e., its unpredictable behavior depends on the input it obtains from an external environment that it does not control. The environment of each process is represented by the set of processes that belong to its neighborhood and by system external actions that could force the execution of a given action in the network.

Every process that exists in Lidia, is modeled as a transition system (in the first-level). Each of these systems describes the potential steps that its corresponding process can perform, assuming that it is embedded in an environment that is optimally cooperative. The details of the internal activity of a process (e.g., its computations) are described by its respective first-level transition system. Most such detail is irrelevant for, and hence unobservable by, the second-level transition system.

Although, the multi-level transition system is powerful enough to model distributed algorithms, the power of Lidia is addicted to the computational characteristics of the descriptive logic $\mathcal{L}^*_\infty$. Similar logics have been studied by other authors, and shown to be particularly robust by [Ott96, GT95, HLN99]. The most important aspect of the language $\mathcal{L}^*_\infty$ is, among other things, its ability to express counting. In fact, counting is a fundamental operation of numerous algorithms. Counters constitute also an essential primitive of query languages. In relational databases, practical query languages, such as SQL, provide counters as built-in functions of the languages. Counters map relations to integers. They are of great importance from a practical point of view. Moreover, counters raise challenging theoretical problems. Logical languages generally lack the ability to express counting, though it is very easy to count on any computational device [AV91].

The Communication Model

Our communication model is a point-to-point communication network which is represented as a simple connected undirected graph where vertices represent processes and two vertices are linked by an edge if the corresponding processes have a direct communication link. Processes communicate by message passing, and each process knows from which channel it receives or sends a message. That is, each process assigns numbers to its ports. An edge between two vertices v_k and $v_{k'}$ represents a channel connecting a port i of v_k to a port j of $v_{k'}$. The communication in Lidia is based on a symmetric port numbering function. That is, if there exists a channel connecting a port i of v_k to a port j of $v_{k'}$, then $i = j$. Moreover, Lidia uses an asynchronous message passing model: processes can not access a global clock and a message sent from a process p to its neighbor q arrives within some finite but unpredictable time. The operating mode of communication channels is encoded in the second-level transition system.

The second-level transition system, abstract away the semantics of the first-level processes, and is only concerned with their (mutually engaging) externally observable behavior. The external activities of an entire Lidia application are modeled by the second-level transition system. Here, a configuration corresponds to a set of processes each of which is associated

with a list of pending messages that have already been broadcast but not yet received. Each second-level transition is defined in terms of transitions reflecting the message passing actions of interacting processes. In a computational point of view, the second-level transitions are the same for all processes involved in the computation. They represent a formal way how communication links between two processors are implemented.

7.3 An informal Overview

The Lidia language is designed to allow precise and direct description of distributed algorithms encoded by means of local computations. Since the model we used is a reactive system model rather than a sequential program model, the language reflects this fundamental distinction. That is, it is not a standard sequential programming language with some constructs for concurrency and interaction added on; rather, concurrency and interaction are at its core.

Two major concepts in Lidia are separation of concerns and anonymous communication. Separation of concerns means that computation concerns are isolated from the communication and cooperative concerns. Anonymous communication means that the processes engaged in communication do not need to know each other.

The starting points for Lidia were the pseudocodes used in earlier works on *Graph Relabeling Rules* and on *I/O-automata*. These pseudocodes contain, in the case of *I/O automata*, explicit representations of state transition definition in form of *(actions, states, transitions,...)*. In this framework, a transition is described using a *transition definition* (*TD*) containing the *preconditions* and the *effects*. This pseudocode has evolved in two different forms: *a declarative style* (see, e.g., [FLMW94]), in which effects are described by predicates relating pre- and post-states, and an *imperative style* (e.g., [Lyn96]), in which effects are described by simple imperative programs.

Because of our intention to build a formally defined programming language, we made some design decisions in order to reach a suitable relationship between the local computations model and the Lidia programming language.

- We use graph data type to symbolize a distributed systems. Each vertex represents a process and each edge is seen as a communication link between two processes.
- Each vertex or edge has a label that describes its state at any time.
- All computing entities in Lidia, vertices or edges, are represented by processes.
- We only allow the imperative program style in each *TD*. Thus, the *effects* part of a rule is entirely described by a simple imperative program consisting of (possibly nondeterministic) assignments, conditionals, and simple bounded loops. This simplicity makes sense, because transitions are supposed to be executed atomically.
- Variables can be initialized using ordinary assignments and nondeterministic choice statements. The entire initial state may be constrained by predicates.
- The Lidia language can make use of some local computations protocols previously introduced and used by Bauderon et al. [BMMS02]. These are randomized algorithms used to implement local computations in asynchronous systems.

- Each *TD* corresponds to a relabeling rule that is represented in a *precondition-effect* style. A rule can have additional *choose* parameters, which are not formally part of the action name, but which allow values to be chosen to satisfy the precondition and then used in describing the effect.

- An important aspect of nondeterministic programming is allowing maximum freedom in the order of action execution. Control over action order is sometimes needed, particularly at lower levels of abstraction where performance requirements may force particular scheduling decisions. For this reason, we have integrated an explicit support for specifying action order in Lidia. Thus, each rule is enhanced with a list of all rules that have a higher order of priority. If there exists no priority decision between two actions, a random choice is made to designate which rule should be executed.

Related Works and Models

Languages such as IOA [GLV97], UNITY [CM88], SPECTRUM [BFG+93] or TLA [Lam89] are similar to Lidia in that their basic program units are transition definitions with preconditions and effects parts. However, effects in TLA are described declaratively, effects in UNITY and SPECTRUM are described imperatively and IOA allows both declaration styles. In contrast to Lidia, SPECTRUM is an algebraic specification language that is based on axiomatic specification techniques and is oriented towards functional programs. It includes features such as λ-abstraction which would be inconsistent with our aim to construct a simple programming language.

In the same way, instead of representing processes as *I/O-automata* (i.e., IOA, SPECTRUM, TLA), any process in Lidia is consider as a processing entity that belongs to a compact system and can perform several computation rules (first-level system). This departure from the automata model is motivated by the fact that we do not want to deal with complex automata operations that are useless in the local computation environment. As a matter of course, the underlying networks of the algorithms we intend to encode in Lidia are always static and we will never take advantage of operations that compose smaller components in bigger ones.

Moreover, TLA and UNITY are based on state automata that communicate via shared variables, whereas Lidia uses an asynchronous message passing system. In most of these languages, the preconditions of any *TD* are expressed using descriptive languages whose complexities are unknown.

The main particularity of Lidia resides in the fact that, within its framework, the preconditions are exclusively described in the logic $\mathcal{L}^*_\infty$ [MO04d] and the effects are expressed using an imperative language constructed on $\mathcal{L}^*_\infty$ (*LASL*). We have shown that the logic $\mathcal{L}^*_\infty$ has enough descriptive power to fully described all *PTIME* queries in the structures used in Lidia. Finally, this particularity of $\mathcal{L}^*_\infty$ has helped us to characterize the class of distributed problems that are suitable to the Lidia programming environment.

7.4 The Logic $\mathcal{L}^*_\infty$

Here, we introduce some extensions of first-order logic that are necessary to understand how the language $\mathcal{L}^*_\infty$ is built. These are fixed-point logics, infinitary logic and infinitary logic with counting. Throughout the rest of this chapter, we will assume that the reader is already familiar with the basic concepts of first-order logic and fixed-point logics as the definition of formulae and how the notion of truth is defined.

First of all, recall that infinitary logic $\mathcal{L}_{\infty\omega}$ is the extension of first-order logic where infinite disjunctions and conjunctions of formulas are also allowed. It is well known that any (isomorphism closed) class $\mathcal{C} \subseteq STRUCT[\sigma]$ can be defined in $\mathcal{L}_{\infty\omega}$ (where $STRUCT[\sigma]$ denotes the class of finite σ-structures). Interest of this logic comes from its fragments which have weaker expressive power. One such fragment is $\mathcal{L}^k_{\infty\omega}$ where only k distinct variables, free or bound, are allowed. The finite variable logic $\mathcal{L}^\omega_{\infty\omega}$ is then the union of $\mathcal{L}^k_{\infty\omega}$ over all natural numbers k. Fixed-point logics (least inflationary logic, partial fixed-point logics and transitive closure logic) can all be embedded into $\mathcal{L}^\omega_{\infty\omega}$. It is also easy to see that $\mathcal{L}^\omega_{\infty\omega}$ can not express certain counting properties, such as parity of cardinality. For an extensive study of this logic, see e.g. [EF95].

$\mathcal{L}^*_\infty$ is obtained by first adding counting terms, counting quantifiers to the logic $\mathcal{L}^\omega_{\infty\omega}$ over two-sorted structures (the second sort being interpreted as $\mathbb{N}$), and then restricting it to formulae of finite rank. The idea of using the set of natural numbers in the two-sorted structure is influenced by meta-finite model theory of [GG98]. Similar extensions exist in the literature [Ott97], but they restrict the logic by means of the numbers of variables, which still permits fixed-point computation. In contrast, following [HLN99], we restrict the logic by requiring the rank of a formula be finite (where the rank is defined as quantifier rank, except that it does not take into account quantifiers over $\mathbb{N}$), thus putting no limits at all on the available arithmetic.

7.4.1 The Alphabet of $\mathcal{L}^*_\infty$

The alphabet of $\mathcal{L}^*_\infty$ is obtained by adding the following counting terms and quantifiers to the symbols of first-order logic.

- Counting Quantifiers: $\exists_{\boldsymbol{i}}$, $\exists^{\perp}_{\boldsymbol{i}}$
- Counting Terms: $\exists_{\mathbf{1}}$, $\exists_{\mathbf{2}}$, $\exists_{\mathbf{3}}$,... $\exists^{\perp}_{\mathbf{1}}$, $\exists^{\perp}_{\mathbf{2}}$, $\exists^{\perp}_{\mathbf{3}}$,...
- Equality symbol: $=, \neq$
- Usual arithmetic functions: $+, -, \times, \div, \boldsymbol{modulo}, \boldsymbol{div}, \boldsymbol{exp}, \boldsymbol{factorial}$,...

All the arithmetic help functions have the intuitive meaning. The counting quantifier $\exists_i$ is satisfied if the set of elements, that satisfy a given formula ϕ, has a cardinality greater or equal to i. On the other hand, the formula $\exists^{\perp}_i x\phi(x)$ is *true*, if there are exactly i elements x that satisfy $\phi(x)$. We have to notice that the variable i is an element of the set of natural numbers and we can also use it to construct further predicates.

We now define formally the syntax of the infinitary logic $\mathcal{L}^*_\infty$.

Definition 7.3 (Syntax of $\mathcal{L}^*_\infty$) *Let σ be a vocabulary consisting of finitely many relational and constant symbols and let $\{v_1, ..., v_n, ...\}$ be an infinite set of distinct variables. The class of $\mathcal{L}^*_\infty$ formulas over σ is the smallest collection of expressions such that:*

- *it contains all first-order formulas over σ;*
- *if ϕ is a formula in $\mathcal{L}^*_\infty$, then so is $\neg\phi$;*
- *if ϕ is a formula in $\mathcal{L}^*_\infty$, $i \in \mathbb{N}$ and v_j is a variable, then $(\exists_i)v_j\phi$ and $(\exists_i^\perp)v_j\phi$ are also in $\mathcal{L}^*_\infty$;*
- *if ϕ is a formula in $\mathcal{L}^*_\infty$, v_j is a variable, then the formulas $(\exists_0)v_j\phi$, $(\exists_0^\perp)v_j\phi$, $(\exists_1)v_j\phi$, $(\exists_1^\perp)v_j\phi$, $(\exists_2)v_j\phi$, $(\exists_2^\perp)v_j\phi$, $(\exists_3)v_j\phi$, $(\exists_3^\perp)v_j\phi$... are also in $\mathcal{L}^*_\infty$;*
- *if Φ is a set of $\mathcal{L}^*_\infty$ formulas, then $\bigvee\Phi$ and $\bigwedge\Phi$ are also $\mathcal{L}^*_\infty$ formulas.*

Example 7.1 *With the logic $\mathcal{L}^*_\infty$ we can express the fact that a given set ψ has an even cardinality. This is done by the following expression:*

$$\exists_i^\perp x(x \in \psi \ \wedge \ i \bmod 2 = 0);$$

The next example reinforces the fact that the power of first-order logic is not enough to describe all the preconditions that can be executed in Lidia.

Example 7.2 *Consider the case where a given vertex v_0 has to test the following precondition:* The number of its $N-$labeled neighbors is equal to the number of its $A-$labeled neighbors. *Without any counting mechanism, it is not possible to express this kind of precondition in first-order logic. Whereas, this query can be checked in $\mathcal{L}^*_\infty$ with the following sentence:*

$$\exists_i u \exists_j w(u \neq v_0 \wedge w \neq v_0 \wedge \lambda(u) = N \wedge \lambda(w) = A \wedge i = j).$$

7.4.2 The Semantic of $\mathcal{L}^*_\infty$

The semantics of $\mathcal{L}^*_\infty$ formulas is a direct extension of the semantics of first-order logic, with $\bigvee\Phi$ interpreted as a disjunction over all formulas in Φ and $\bigwedge\Phi$ interpreted as a conjunction.

Although most of the basic theory on infinitary logics was developed for arbitrary structures, the interesting results only speak about finite ones. This is why we have restricted our attention to finite structures with finite vocabularies, unless it is explicitly stated otherwise. Furthermore, we always assume classes of structures to be closed under isomorphism. In the same sense that sentences of a logic define classes of structures, formulas with free variables define queries.

Definition 7.4 (Structures of $\mathcal{L}^*_\infty$ in Lidia) *A structure $\mathbb{A}$ is of the form:*

$$\begin{aligned}\mathbb{A} = \quad & \langle\{v_1, \cdots, v_m\}, \{1, \cdots, n\}, \{L\}, <, =, \neq, 0, 1, \mathit{true}, \mathit{false}, MIN, MAX, \\ & R_1^{\mathbb{A}}, \cdots, R_l^{\mathbb{A}}, f_1^{\mathbb{A}}, \cdots, f_k^{\mathbb{A}}\rangle.\end{aligned}$$

Here the relations $R^{\mathbb{A}}$ are defined on the vertices domain $\{v_1, \cdots, v_m\}$, while on the numerical domain $\{1, \cdots, n\}$ one has constants 0 and 1. L is the finite set of labels and $\{v_1, \cdots, v_m\}$ even represents the neighborhood of a given vertex v_0.

In essence, any vertex u can only see the part of the general domain of $\mathbb{A}$ that is represented by its neighborhood. Further on, the added universe of numbers gives us the ability to do some arithmetic on the side as we express a property of the input structures.

7.4.3 The Satisfaction Relation

The satisfaction relation makes precise the notion of a formula being true under an interpretation. Let D be the domain of our logic, I be an interpretation relation, $\mathbb{A} = \langle D, I \rangle$ be an interpretation structure, g be an assignment in $\mathbb{A}$ and ϕ be a formula. If ϕ can be represented in the *FOL* then the satisfaction relation of *FOL* can be used on ϕ. If ϕ contains new introduced quantifiers, then the satisfaction relation of ϕ by g in $\mathbb{A}$, is given by the following rules:
Counting Quantifiers

$$\begin{aligned} \mathbb{A} \models \exists_i x\phi[g] \Leftrightarrow & \quad \mathbb{A} \models \phi[g[x/d, i/j]] \Leftrightarrow \mathbb{A} \models \phi[g[x/d]] \\ & \quad \text{for at least } j \text{ elements } d \in D \text{ with } j \leq |D| \\ \mathbb{A} \models \exists_i^{\perp} x\phi[g] \Leftrightarrow & \quad \mathbb{A} \models \phi[g[x/d, i/j]] \Leftrightarrow \mathbb{A} \models \phi[g[x/d]] \\ & \quad \text{for exactly } j \text{ elements } d \in D \text{ with } j \leq |D| \end{aligned}$$

The satisfaction of counting terms is a special case of the above satisfaction relations.

Note 7.1 *The fact that we allow formulas with infinite disjunctions and conjunctions in $\mathcal{L}_{\infty}^{*}$ can be a bit weird. Nevertheless, all the formulas in $\mathcal{L}_{\infty}^{*}$ have finite models. Furthermore, it can be proved that in finite structures, any formula in $\mathcal{L}_{\infty}^{*}$ is exactly equivalent to a formula of a finite logic language* (IFP+C). *Moreover, the evaluation of all formulas of $\mathcal{L}_{\infty}^{*}$ can be performed in polynomial time. All these results are stated in [MO04d], however they will be presented in the next chapter.*

7.5 Transition in Lidia

Any distributed algorithm encoded in Lidia is represented as a reactive network system consisting in processors and communication links. One such a link enables two processes to communicate with each other. Because of the fact that the system is reactive, the state transitions of each process are represented in a *Precondition-Effect* style. Basically, all processes execute the same algorithm. This means that the set of transitions defined in the first level transition system is the same for all processes.

7.5.1 Preconditions

Let now G be a connected graph (network) and $v \in V$ be a vertex(processor) of G. The main task of the *preconditions* side is to describe the local properties of the ball of radius k centered on v. Although the logic $\mathcal{L}_{\infty}^{*}$ achieves this goal, we made some decisions to adjust it to Lidia.

First of all, we set the universe of the used structure, $\mathbb{A}$, be specific to the neighborhood of the center of the star we considered. This means that each vertex of G can only *"see"* the part of the vertices of V that belongs to its neighborhood of radius k. In the Lidia language, we restricted ourself to balls of radius 1. Thus, the neighborhood of a vertex v is represented by

the entire set of vertices u that are neighbors of v and to the set of edges e that are adjacent to v. It must be clear that all relations $R_i^{\mathbb{A}}$ that try to catch any property between two neighbors of v are strictly forbidden. That is, they do not belong to the structure $\mathbb{A}$ related to the neighborhood of v.

7.5.2 Effects

The *Effects* sides are described by *LASL* programs [Oss04]. Basically, the language LASL requires only one data type: nonnegative integers. The identifiers are defined to be alphanumeric strings starting with a letter. Therefore, we do not need any type declaration (since there is only one data type and since constants can easily be distinguished from identifiers).

Due to arithmetic functions introduced in the language $\mathcal{L}_\infty^*$, it is clear that LASL contains the *increment* and *decrement* statement types that respectively increment or decrement the value named by an identifier. More generally, LASL contains the following statement types:

1. Basic statements
   - $\boldsymbol{x := varname}$: is a simple assignation statement.
   - $\boldsymbol{\forall x(\Phi(x)), \rho}$: the statement ρ is performed exactly once if and only if all elements x satisfy the predicate Φ_i.
   - $\boldsymbol{\exists_i x(\Phi(x)), \rho}$: the statement ρ is performed once if and only if there are at least i elements x that satisfy the predicate $\Phi(x)$.
   - $\boldsymbol{\exists_i^{\perp} x(\Phi(x)), \rho}$: the statement ρ is performed exactly once if and only if there are exactly i elements x that satisfy the predicate $\Phi(x)$.
   - $\boldsymbol{\forall x(\Phi(x), \rho)}$: the statement ρ is performed each time an element x satisfying the predicate Φ is found.
2. Loop statement
   - $\boldsymbol{\forall(\Phi)\{\rho\}}$: a given statement ρ is performed as long as a given predicate Φ is satisfied.

Starting from the statements of LASL, we have proved [Oss04] that this language is capable of expressing the solution of any algorithmically solvable problem. Thus, LASL is able to express the effects of any transition in Lidia. An important side effect of the power of LASL is that all arithmetic functions can be computed in Lidia. However, instead of using a LASL bock procedure to describe a given arithmetic function, for instance *modulo*, we simply use the corresponding common notation, % or *mod*, to refer to this procedure.

7.6 Data Types in Lidia

A general description of all Lidia data types with their respective operations is presented in [MO04b]. Lidia enables users to define new data types in order to characterize the actions and states of each process. The data types *graph*, *node*, *edge*, *Bool*, *Int*, *Nat*, *Real*, *Char*, and *String* can appear in Lidia descriptions without explicit declarations. It goes without saying that the list of data types considered in this section is not exhaustive.

The used *graph* data type represents parameterized graphs. Thus, any instance of *graph* contains labels attached to its vertices and edges. Compound data types can be constructed using the following type constructors without explicit declarations:

- *Array[I, E]* is an element of elements of type E indexed by elements of type I.
- *Seq[E]* is a finite sequence of elements of type E.
- *Set[E]* is a finite set of elements of type E.
- *Mset[E]* is a finite multiset of elements of type E.
- *node_set* is a finite set of nodes.
- *edge_set* is a finite set of edges.
- *int_set* is a finite set of integer numbers.
- ...

Users can define additional data types, as well as redefine built-in types. For instance, they can explicitly declare enumeration, tuple, and union types analogous to those found in many common programming languages. For example,

- **type** *Color* = enumeration of *red white blue*
- **type** *Msg* = tuple of *source*, *dest* : *Process*, *contents* : *String*

If S is a set of elements of type E, then the operation $x :=$ choose s in S consists in assigning to x the value of an element of S that has been randomly chosen. Generally, the *choose* operation is used to randomly select an element from a set of items that satisfy some characteristics. It is a simple way to perform random initializations in Lidia.

7.7 Structure of a Lidia Program

We illustrate our model, as well as the use of the language Lidia to describe distributed algorithms, by two simple examples.

Example 7.3 (Spanning tree computation) *The Lidia-Program 7.1 represents a Lidia implementation that computes a spanning tree in a graph. The corresponding procedure may be encoded by the graph relabeling system* $\mathcal{R}_1 = (L_1, I_1, P_1)$ *defined by* $L_1 = \{N, A, 0, 1\}$, $I_1 = \{N, A, 0\}$, *and* $P_1 = \{R1\}$ *where* $R1$ *is the following relabeling rule:*

R1 : **Expanding rule**

Precondition :

- $\lambda(v_0) = A$,
- $\exists\, v \in B(v_0, 1), v \neq v_0, \lambda(v) = N, \lambda([v_0, v]) = 0$.

Relabeling :

- $\lambda'(v) := A$
- $\lambda'([v_0, v]) := 1$

We assume that a unique vertex has initially label A, all other vertices having label N and all edges having label 0. At each step of the computation, an A-labeled vertex u may activate any of its neutral neighbors, say v. In that case, u keeps its label, v becomes A-labeled and the edge $\{u, v\}$ becomes 1-labeled. Thus, several vertices may be active at the same time. Concurrent steps will be allowed provided that two such steps involve distinct vertices. The computation stops as soon as all the vertices have been activated. The spanning tree is then given by the 1-labeled edges.

As depicted in Lidia-Program 7.1, every Lidia program consists of four parts. General variables are declared and initialized in the first and second parts. The types *nLabel* and *eLabel* are introduced to define the kind of labels that will be respectively used on vertices and edges. The third part defines the kind of local synchronization protocol used during the computation (see Chapter 4). This part is only used when overlapping parallel rewriting steps can occur. That is, as long as the rewriting rules are applied on balls of radius $k > 0$, k-local elections are needed to ensure the correctness of the computation. In Lidia-Program 7.1, any vertex v_0 has to win a 1-local election before computing rule $R1$. The fourth part contains the list of all rules that can be applied during the computation.

There are two kinds of rules in Lidia: active and passive rules. While active rules are executed by vertices that have won the required k-local election, passive rules are executed by the losers of the same election. In the above example R_0 and P_0 are respectively active and passive rules. A vertex v_0 applies R_0 if it is N-labeled and it has a A-labeled neighbor v that has won the k-local election. The consequences of the application of rule R_0 are represented by the relabeling of edge $\{v_0, w\}$ that becomes 1-labeled and by the fact that v_0 becomes A-labeled. Generally, a passive rule consists in sending or receiving messages. Thus, instead of using the term *Relabeling*, the second part of a passive rule is called *Action* in Lidia programs. In any Lidia program the names of all active rules are first listed before each of them is specified in details. Each specification begins with the rule name followed by the statements concerning *Precondition*, *Relabeling* and *Priorities*. Passive rules are introduced in the same lines.

Lidia also provides a structure that allows to set an execution priority between two rules. For each rule r, this structure is given as a list of rules that have higher execution priority than r. This is a natural way to encode graph relabeling systems with priorities in Lidia. One has also to notice that, in Lidia, the forbidden contexts of a relabeling rule r are encoded as preconditions in r. If the list of rules that have higher execution priorities than a given rule r_0 is empty, this does not mean that r_0 has the highest priority. This simply denotes the fact that, if the preconditions of of r_0 are satisfied, then r_0 can be executed regardless of the other rules. Moreover, it may sometimes appear that more than one rule have empty lists of priorities. In this case, if at least two of them can be applied, an equiprobable choice is made to decide which rule should be effectively executed.

The general idea of the Lidia-Program 7.1 consists in three phases. In the first phase, all vertices participate in a 1-local election (lines $14 - 15$). In the second phase, each loser w of the election sends its label to all the winner vertices that belong to its ball of radius 1 (lines $35 - 43$). The third phase is executed by all the winner vertices. Each of them receives the

Lidia-Program 7.1 Spanning tree implementation

```
    Declaration
 2: type nLabel = tuple of var: string;
    type eLabel = tuple of var: int;
 4: G : graph < nLabel, eLabel >;
    edgeLab : eLabel;
 6: nodeLab : nLabel;
    Initialization
 8: edgeLab.var := 0;
    nodeLab.var := 'N';
10: G.init(nodeLab, edgeLab);
    v : node;
12: v := G.choose_node();
    v.label.var := 'A';
14: Synchronization
    Synchro_LC1();
16: SpanningTree(v_0 : node) :
    Universe
18: B : node_set;
    B := G.neighborhood(v_0);
20: Active rules: R_0;
    L_v: node_array < nLabel >;
22: L_v.init(B, nodeLab);
    w : node;
24: ∀w ∈ B(w ≠ v_0, L_v[w] := ReceiveFrom(w));
    R_0:={
26: Precondition
    v_0.sync = v_0;
28: v_0.label.var = 'N' ∧ ∃w ∈ B(w ≠ v_0 ∧ L_v[w].var = 'A' ∧ [v_0, w].var = 0);
    Relabeling
30: v_0.label.var := 'A';
    [v_0, w].var := 1;
32: Priorities
    {};
34: };
    Passive rules:P_0;
36: P_0:={
    Precondition
38: v_0.sync ≠ v_0;
    Action
40: v : node;
    ∀ v ∈ B(v ≠ v_0 ∧ v.sync = v, SendTo(v, v_0.label));
42: Priorities
    {};
44: };
    SpanningTree.run(G);
```

labels sent by all its neighbors (lines $22-24$). Thereafter, it checks if the relabeling rule R_0 can be performed in its ball of radius 1 (lines $25-34$). For a given vertex v, the ball of radius 1 is represented by a *node_set* variable B that contains the star graph centered on v (lines $17-19$). B is called the *universe* of v. Once a loser vertex has sent its label it performs the election procedure again. This also yields for a winner vertex that has performed the third phase.

Example 7.4 (Election in anonymous trees) *We now present a second programming example that aims to solve the election problem in Lidia. The main idea is to perform the election algorithm in the family of tree-shaped networks. The corresponding election algorithm can be encoded using the following relabeling system.*

$L = \{N, elected, non\text{-}elected\}$ is the set of labels. The initial label on all vertices is $l_0 = N$ and there are two meta-rules described as follows.

R1 : **Pruning rule**

Precondition :

- $\lambda(v_0) = N$,
- $\exists!\, v \in B(v_0, 1), v \neq v_0, \lambda(v) = N$.

Relabeling :

- $\lambda'(v_0) :=$ *non-elected*.

R2 : **Election rule**

Precondition :

- $\lambda(v_0) = N$,
- $\forall v \in B(v_0, 1), v \neq v_0, \lambda(v) \neq N$.

Relabeling :

- $\lambda'(v_0) :=$ *elected*.

Let us call a pendant vertex any vertex labeled N having exactly one neighbor with the label N. The meta-rule $R1$ consists in cutting a pendant vertex by giving it the label *non-elected*. The label N of a vertex v becomes *elected* by the meta-rule $R2$ if the vertex v has no neighbor labeled N. A complete proof of this system may be found in [LMS99].

This example is implemented in Algorithm 7.2 and Figure 17 simulates one possible execution of the algorithm. In this simulation, all black colored vertices are N-labeled and non-elected. The white colored vertices are F-labeled and non-elected. The elected vertex is the red colored vertex that is N-labeled and does not have any N-labeled neighbor. Many other programming examples in Lidia can be found in [MO04b].

The operating mode of the Lidia-Program 7.2 is similar to the one introduced in Lidia-Program 7.1. The difference resides in the fact that here, a winner vertex has to check if two rules(R_0 and R_1) can be executed before performing the 1-local election again. None of the both presented programming examples stops. In fact, even though a vertex is elected, the

Lidia-Program 7.2 Election in an anonymous tree

Declaration
type $nLabel$ = **tuple of** var: **string**;
type $eLabel$ = **tuple of** var: **int**;
G : ***graph < nLabel, eLabel >;***
$edgeLab : eLabel$;
$nodeLab : nLabel$;
Initialization
$edgeLab.var := 0$;
$nodeLab.var :=$ 'N';
$G.init(nodeLab, edgeLab)$;
Synchronization
$Synchro\_LC1()$;
$ElectionInTree(v_0 :$ ***node***$)$
Universe
B: ***node_set***;
$B := G.voisinage(v_0)$;
L_v: ***node_array < nLabel >***;
$L_v.init(B, nodeLab)$;
Active rules: R_0, R_1;
$\forall w \in B(w \neq v_0,\ L_v[w] := ReceiveFrom(w))$;
$R_0 :=\{$
Precondition
$v :$ ***node***
$v_0.label.var =$ 'N' $\wedge\ \exists_1^{\perp} v \in B(v \neq v_0 \wedge L_v[v].var =$ 'N');
Relabeling
$v_0.label.var :=$ 'F';
Priorities
$\{\}$;
$\}$;
$R_1 :=\{$
Precondition
$v :$ ***node***
$v_0.label.var =$ 'N' $\wedge\ \forall v \in B \neg(v \neq v_0 \wedge L_v[v].var \neq$ 'N');
Relabeling
$v_0.label.var :=$ 'E';
Priorities
$\{\}$;
$\}$;
Passive rules: R_0;
$R_0 :=\{$
Precondition
$v_0.sync \neq v_0$;
Action
$v : node$;
$\forall\, v \in B(v \neq v_0 \wedge v.sync = v,\ SendTo(v, v_0.label))$;
Priorities
$\{\}$;
$\}$;
$ElectionInTree.run(G)$;

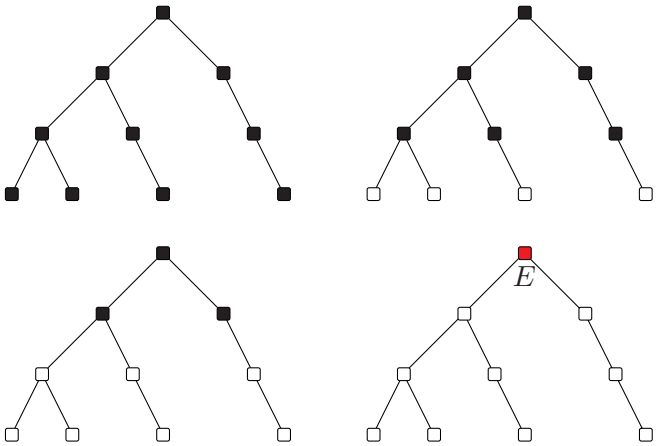


Figure 17: Distributed computation of the election algorithm in trees

loser vertices have no means to realize this situation. It is possible to deal successfully with this situation by adding appropriate rules that should broadcast, to all F-labeled vertices, the information concerning the existence of an elected vertex.

7.8 Communication Level

Processes are the basic computational entities in the execution of Lidia programs. A process operates on its internal data which are not accessible to other processes. The interaction between processes is only performed by means of sending and receiving data through communication channels that are created dynamically. In fact, the creation of an edge (or link) e connecting two processes p_i and p_j consists in the creation of a communication channel which connects p_i with p_j. This channel has an unique identity which is only known to the processes p_i and p_j. Figure 18 represents such a communication channel with identity 1. This channel is composed of

- two input buffers: I_{ij}, I_{ji},
- two output buffers: O_{ij}, O_{ji},
- two **transmission** processes C_{ij}, C_{ji} and
- an edge relabeling function λ.

The behaviors of these processes are implemented in the second level transition system. If process p_i has to send a message M to p_j, p_i appends M to the buffer O_{ij}. In the same way, if p_i has to receive a message M' from p_j, it looks for M' in the corresponding input buffer I_{ji}. The transmission processes C_{ij} and C_{ji} are responsible for the deliveries of the sent messages in both directions. For instance, C_{ij} has periodically to move messages from O_{ij} to I_{ij}. The default transmission channels guarantee all the properties of a reliable communication channel, which neither loses nor reorders messages in transit.

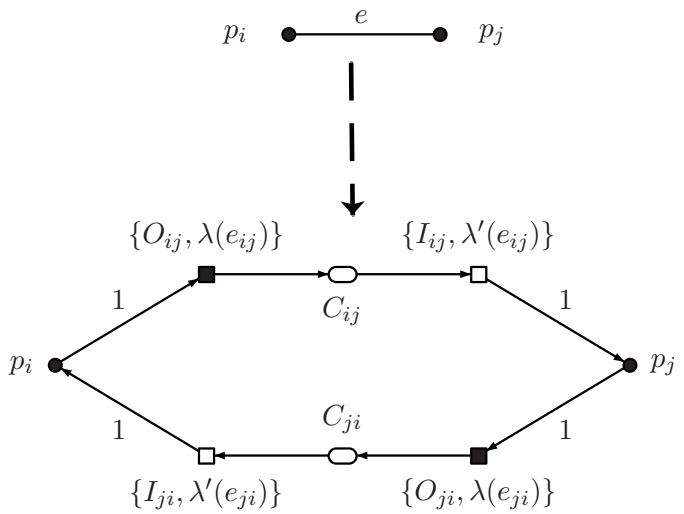


Figure 18: A communication channel between p_i and p_j

7.8.1 Basic Communication Instructions

In order to communicate with other processes, a process p_i can make use of four basic instructions that are also implemented in the Lidia language. Informal descriptions of these instructions are given as:

ReceiveFrom(p_j): This instruction forces p_i to look for a new message in I_{ji}. If the buffer is not empty, the message on the top of I_{ji} is returned and deleted from I_{ji} (i.e. with the $Pop()$ operation). Otherwise the value *nil* is returned.

$R1$: **ReceiveFrom(p_j)**

Precondition :
- $p_j \in B(v_0, 1)$,
- $I_{ji} \neq empty$.

Relabeling :
- $M := I_{ji}.Pop()$

SentBy(p_j): This instruction forces p_i to look for a new message in I_{ji}. If the buffer is not empty, the message on the top of I_{ji} is returned (i.e. with the $pop()$ operation), otherwise the value *nil* is returned.

$R1$: **SentBy(p_j)**

Precondition :
- $p_j \in B(v_0, 1)$,

Relabeling :
- $M := I_{ji}.pop()$

SendTo(p_j, M): With this instruction p_i sends a message M to p_j. That is, p_i appends M to the buffer O_{ij}.

$R1$: **SendTo**(p_j,M)

Precondition :
- $p_j \in B(v_0, 1)$,

Relabeling :
- $O_{ji}.append(M)$

SendAll(M)**:** With this instruction p_i executes the instruction $SendTo(p_j, M)$ for all its neighbors p_j. That is, p_i appends M to all the buffers O_{ij}.

7.8.2 Edge Labeling

In the model of Mazurkiewicz, that has been proved to be more powerful than all the others [CM04], it is possible for a vertex v to change the label of one of its adjacent edges. Due to the characteristics of the communication model presented above, changing the label of an edge $e = \{p_i, p_j\}$ can be performed by process p_i in the following steps.

Step 1: Process p_i checks if $\lambda(e_{ij}) =$ nil and it has soon read $\lambda'(e_{ij})$. If this is the case, it set $\lambda(e_{ij})$ to the new value of the label of e. Otherwise, it repeats Step 1.

Step 2: Process C_{ij} repeats the following actions forever:

- If $\lambda(e_{ij}) \neq nil$ and $\lambda'(e_{ij})$ has soon been read by p_j, then
  - C_{ij} set $\lambda'(e_{ij})$ to $\lambda(e_{ij})$,
  - it actualizes the timestamp of $\lambda'(e_{ij})$ and set $\lambda(e_{ij})$ to nil.
- If $\lambda(e_{ij}) =$ nil or the old value of $\lambda'(e_{ij})$ has not been read yet by p_j, then C_{ij} actualizes the timestamp of $\lambda'(e_{ij})$.

Step 3: p_j waits until a newer value of $\lambda'(e_{ij})$ is set by C_{ij}. It stores the new label l_e and marks $\lambda'(e_{ij})$ as seen.

Each time process p_i has to perform a computation using the label of one of its adjacent edges e, it must be sure that its stored value l_e is the actual value of e. To fulfill this requirement, p_i respect the following conditions.

- Each time p_i changes the label of one of its incident edges, its stores the tuple $\{l_o, l_n\}$ in its local memory, where l_o and l_n are respectively the old and new label values of edge e. The old value l_o is stored with the corresponding timestamp.
- Actualization: if $\lambda'(e_{ij})$ has not been seen, then $\lambda'(e_{ij})$ is the new value of the edge e. Otherwise, l_n is the actual label of e.

As depicted in Figure 18, it is obvious that if p_i and p_j try simultaneously to change the label of e, they will have two different values of $\lambda(e)$. In fact, it is not possible to break the communication symmetry between two processes connected by an edge e. Thus, there is no deterministic algorithm that solves the local election problem in a graph composed by two connected processes. More generally, it can be proved that all the impossibility results stated in [CM04] and [CMZ04] are also valid in the Lidia framework.

7.8.3 Implementation of the Communication Level

In this section we present a possible implementation of the transmission processes described in the previous sections. As presented in Figure 18, the goal of this kind of processes is to transmit messages in transit and to ensure a robust edge relabeling system. For a process C_{ij}, the basic implementation contains the following four relabeling rules which are encoded in Lidia-Program 7.3. In the first rule C_{ij} notices that no new message has been sent by p_i and that either p_i has not set a new edge label or p_j has not taken the actual edge label $\lambda'(e'_{ij})$ into account. In this case, it only actualizes the timestamp of $\lambda'(e'_{ij})$. In the second rule, no new messages were sent, but p_i has changed the edge label. Thus, C_{ij} actualizes the label of the corresponding edge by executing the operations described in Section 7.8.2. Rule R_3 is devoted to the transmission of a new message and rule R_4 simultaneously actualizes the edge label and transmits the new message sent by p_i to p_j.

$R1$: **NOP**

Precondition :

- $O_{ij}.empty() = \text{true}$,
- $\lambda(e_{ij}) = \text{nil} \ \vee \ Read(\lambda'(e'_{ij})) = \text{false}$.

Relabeling :

- $Timestamp(\lambda'(e_{ij}))$.

$R2$: **EdgeLabel**

Precondition :

- $O_{ij}.empty() = \text{true}$,
- $\lambda(e_{ij}) \neq \text{nil}$,
- $Read(\lambda'(e'_{ij})) = \text{true}$.

Relabeling :

- $\lambda'(e_{ij}) := \lambda(e_{ij})$,
- $\lambda(e_{ij}) := \text{nil}$,
- $Timestamp(\lambda'(e_{ij}))$.

$R3$: **TransmitMessage**

Precondition :

- $O_{ij}.empty() \neq \text{true}$,
- $\lambda(e_{ij}) = \text{nil} \ \vee \ Read(\lambda'(e'_{ij})) = \text{false}$.

Relabeling :

- $I_{ij}.append(O_{ij}.Pop())$,
- $Timestamp(\lambda'(e_{ij}))$.

$R4$: **MessageLabel**

Precondition :

- $O_{ij}.empty() \neq \text{true}$,
- $\lambda(e_{ij}) \neq \text{nil}$,
- $Read(\lambda'(e'_{ij})) = \text{true}$.

Relabeling :

- $I_{ij}.append(O_{ij}.pop())$,
- $\lambda'(e_{ij}) := \lambda(e_{ij})$,
- $\lambda(e_{ij}) := \text{nil}$,
- $Timestamp(\lambda'(e_{ij}))$.

By the hand of these actions, Lidia-Program 7.1 and Lidia-Program 7.2 can be made up by inserting the transitions of Lidia-Program 7.3 after the last transition rule of the corresponding programs. Due to the length of the so constructed programs, it is common to describe Lidia programs by restricting the list of rules to the first level transition system. This should then have the intuitive meaning that the default communication system is used.

7.9 Concluding Remarks

Lidia is a new programming language that manages the interactions among concurrent components and uses a two-levels transition system to model reactive systems. It is devoted to the implementation of algorithms encoded by local computations.In a programming point of view, we have developed a code generator that performs a source-to-source translation from Lidia to the programming language Java. Our main motivation was to enhance the platform of ViSiDiA [MS] with the so constructed generator. Thus, it is now possible to reach a faithful simulation for the distributed algorithms implemented in Lidia. Furthermore, we have provided Lidia with a tool that is able to test the validity of given programs properties. The inputs of this tool are properties expressed in the language $\mathcal{L}^*_\infty$ or in the monadic second order logic (MSOL) that is known to have the capability to express some NP-hard graph properties [Cou97]. We expect this tool to represent a first step in the design and development of tools which should contribute to prove the correctness of the execution of Lidia programs.

The main remaining research problem concerns the construction of a virtual machine for Lidia. In fact, our efforts aspire now towards the development of such a machine which should give us the possibility to design a platform independent compiler for Lidia programming.

Lidia-Program 7.3 Default transition system for the second level

```
    Declaration
 2: type Lab = tuple of time: int, label: string, status: boolean;
    type nLabel = tuple of O_ij, I_ij: stack,e_ij, e'_ij: Lab;
 4: type eLabel = var: int;
    G : graph < nLabel, eLabel >;
 6: transmission(v_i : node, v_j : node) :
    Universe
 8: Active rules: R_1, R_2, R_3, R_4;
    R_1:={
10: Precondition
    O_ij.empty() = True ∧ (e_ij ≠ nil ∨ e'_ij.status = false),
12: Relabeling
    e'_ij.time := e'_ij.time + 1.
14: Priorities
    {};
16: };
    R_2:={
18: Precondition
    O_ij.empty() = true ∧ (e_ij ≠ nil ∧ e'_ij.status = true),
20: Relabeling
    e'_ij := e_ij,
22: e'_ij.time := e'_ij.time + 1,
    e_ij := nil,
24: Priorities
    {};
26: };
    R_3:={
28: Precondition
    O_ij.empty() ≠ true ∧ (e_ij = nil ∨ e'_ij.status = false),
30: Relabeling
    I_ij.append(O_ij.Pop()),
32: e'_ij.time := e'_ij.time + 1.
    Priorities
34: {};
    };
36: R_4:={
    Precondition
38: O_ij.empty() ≠ true ∧ (e_ij ≠ nil ∧ e'_ij.status = true),
    Relabeling
40: I_ij.append(O_ij.pop()),
    e'_ij := e_ij,
42: e'_ij.time := e'_ij.time + 1,
    e_ij := nil,
44: Priorities
    {};
46: };
    transmission.run(G);
```

Chapter 8

Computational Characteristics of LIDiA

Contents

8.1 Introduction

In the last chapter we have presented a new programming language for implementing distributed algorithms encoded by means of local computations. This language, called Lidia, is based on a *two-level* transition system model: the first level is used to specify the behavior of each single component, whereas the second level captures their interactions. Transitions are basically expressed in a *precondition-effect* style. Moreover, Lidia depends on a logic $\mathcal{L}^*_\infty$ that is used to express the preconditions of each transition. The logic $\mathcal{L}^*_\infty$ is an extension of first-order logic by means of new counting quantifiers and additional computation symbols.

The main topic of this chapter is to state a relationship between the logical definability in $\mathcal{L}^*_\infty$ and the computational power of Lidia.

We will make use of results of the *finite model theory* to show that if the rules' preconditions of a given problem p can be expressed in $\mathcal{L}^*_\infty$, then p can be computed in Lidia. More generally, we will state that, in the presence of users defined function, Lidia is able to compute any distributed problem encoded by means of local computations. Hence, the class of distributed algorithms expressible in Lidia is exactly the class of problems encoded by local computations.

8.2 Descriptive Complexity of $\mathcal{L}^*_\infty$

The main task of this section is to determine the descriptive complexity of the logic $\mathcal{L}^*_\infty$. More precisely, we want to find out the main complexity class that is captured by this logic. We look for results saying that, on a certain domain $\mathcal{D}$ of structures, the logic $\mathcal{L}^*_\infty$ *captures* the complexity class PTIME. We expect to satisfy the following conditions:

(1) For every fixed sentence $\phi \in \mathcal{L}^*_\infty$, the data complexity of evaluating ϕ on structures from $\mathcal{D}$ is a problem in the complexity class PTIME.

(2) Every property of structures in $\mathcal{D}$ that can be decided with complexity PTIME is definable in the logic $\mathcal{L}^*_\infty$.

As a matter of course, the domain $\mathcal{D}$ and the corresponding structures will be defined as presented in Section 7.4. Our goal is to describe complexity classes by logics. Thus, we agree to consider complexity classes as classes of languages over $\{0,1\}$. However, our results are not only valid for computations on classes of words (that is, languages), but also on classes of graphs or arbitrary relational structures.

Example 8.1 *A well-known result of Hopcroft and Tarjan [HT74] says that PLANARITY of graphs is in PTIME. What this means is that there is a PTIME-algorithm that, given any adjacent matrix of a graph, decides whether the graph is planar or not.*

In general, with each class $\mathfrak{C}$ of $\tau-$structures we associate the language

$$\mathcal{L}(\mathfrak{C}) = \{\mathbf{e}((\mathbf{S}, \leq^{\mathbf{S}}))|\mathbf{S} \in \mathfrak{C}, \leq^{\mathbf{S}} \text{, linear order of } \mathbf{S}\} = \bigcup_{\mathbf{S}\in\mathfrak{C}} \mathcal{L}(\mathbf{S})$$

An encoding for a set $\mathbf{S}$ is an injective function $\mathbf{e} : \mathbf{S} \to \{0,1\}^*$. Formulas can be considered as words over some richer alphabet and thus easily be translated to words over $\{0,1\}$. As from now, we assume that the reader is familiar with the basics of complexity theory and has heard of some of the common complexity classes, such as LOGSPACE, PTIME, NPTIME and PSPACE. We assume further that the reader has knowledge about the principles of logics such as *fixed-point logic*, *least fixed-point logic (LFP)* and *inflationary fixed-point logic (IFP)*. Fixed-point logic is an extension of first-order logic designed to reflect the power of induction. There are several formalizations which are not in general equivalent, but the differences are of no concern to us. This is also justified by the results of Gurevich and Shelah [GS86] stating that many different definitions of fixed-point logic coincide for finite structures. We refer the reader to [EF95] for a detailed presentation of the basic material of this section.

8.2.1 Logic with Counting

from the point of view of expressiveness, *FOL* has two main deficiencies: It lacks the power to express anything that requires recursion (the simplest example is transitive closure) and it can not count, as witnessed by the impossibility to express that a structure has even cardinality. A number of logics add recursion in one way or another to *FOL*, notably the various forms of fixed-point logics. On ordered finite structures, some of these logics can express precisely the queries that are computable in PTIME or PSPACE. However, on arbitrary finite structures

they do not, and almost all known examples showing this involve counting. While in presence of an ordering, the ability to count is inherent in fixed-point logic, hardly any of it is retained in its absence.

Therefore Immerman proposed [Imm86] to add counting quantifiers to logics and asked whether a suitable variant of fixed-point logic with counting would suffice to capture PTIME. Meanwhile, fixed-point logic with counting has turned out to be an important and robust logic, that defines natural level of expressiveness and allows to capture PTIME on interesting classes of structures.

There are different ways of adding counting mechanisms to a logic, which are not necessarily equivalent. The most straightforward possibility is the addition of quantifiers of the form $\exists^{\geq 2}$, $\exists^{\geq 3}$, with the obvious meanings. While this is perfectly reasonable for bounded-variable fragments of *FOL* or infinitary logic (see e.g. [Ott97]) it is not general enough for fixed-point logic, because it does not allow for recursion over the counting parameters i in quantifiers $\exists^{\geq i} x$. These counting parameters should therefore be considered as variables that range over natural numbers. This implies indirectly the use of two-sorted structures in most counting logics with the second sort being the set of natural numbers $\mathbb{N}$. We denote by (FOL + C) the *FOL* with counting.
We now define *inflationary fixed-point logic with counting* (IFP + C) by adding to (FOL + C) the usual rules for building inflationary, ranging over both sorts.

Definition 8.1 *Inflationary fixed-point logic with counting (IFP + C), is the closure of two-sorted FOL under the following rules:*

(i) *The rule for building counting terms;*

(ii) *The usual rules of FOL for building terms and formulae;*

(iii) *The fixed-point transformation rule: Suppose that $\psi(R, \bar{x}, \bar{\mu})$ is a formula of vocabulary $\tau \cup \{R\}$ where $\bar{x}$, $\bar{\mu}$ and R has mixed arity (k, l), and that $(\bar{u}, \bar{\nu})$ is a $k+l$-tuple of first- and second-sort terms, respectively. Then $[ifp\ R\bar{x}\bar{\mu}.\psi](\bar{u}, \bar{\nu})$ is a formula of vocabulary τ.*

It is clear that counting terms can be computed in polynomial-time. Hence the data complexity remains in PTIME for (IFP + C).

Infinitary logic with counting. Let $C^k_{\infty\omega}$ be the infinitary logic with k variables, $\mathcal{L}^k_{\infty\omega}$, extended by the quantifiers $\exists^{\geq m}$ ("There exists at least m") for all $m \in \mathbb{N}$. Further, let $C^\omega_{\infty\omega} := \bigcup_k C^k_{\infty\omega}$.

Proposition 8.1 (Otto [Ott97]) $(IFP + C) \subseteq C^\omega_{\infty\omega}$.

From the definitions of $C^\omega_{\infty\omega}$ and $\mathcal{L}^\omega_{\infty\omega}$ we can deduce that $\mathcal{L}^\omega_{\infty\omega} \subseteq C^\omega_{\infty\omega}$. The next proposition states a relation between the logics $C^\omega_{\infty\omega}$ and $\mathcal{L}^*_\infty$.

Proposition 8.2 *In finite variable logics, it is effective that $C^\omega_{\infty\omega} \subseteq \mathcal{L}^*_\infty$.*

Proof. The correctness of this proposition can be shown by a structural induction. One has also to notice that the logic $\mathcal{L}^*_\infty$ can be constructed by augmenting $\mathcal{L}^\omega_{\infty\omega}$ with counting

quantifiers, counting terms, equality symbol and useful arithmetic functions. This means that in finite structures the above proposition is satisfied. □

Note 8.1 *Due to a remark of Otto [Ott97], any sentence of* IFP *is equivalent (over finite structures) with any sentence in* $\mathcal{L}^{\omega}_{\infty\omega}$. *As a consequence, any sentence of* (IFP+C) *is equivalent with a sentence of* $\mathcal{C}^{\omega}_{\infty\omega}$.

Proposition 8.3 *For every fixed sentence* $\phi \in \mathcal{L}^{*}_{\infty}$, *the time complexity of evaluating* ϕ *on structures presented in Section 7.4 is a problem in the complexity class PTIME.*

Proof. Vardi [Var95] has shown that the time data complexity of fixpoint logics is in PTIME. Thus, it is obvious to show that all *(IFP+C)*-sentences can be evaluated in PTIME. Due to Note 8.1, all the sentences of $\mathcal{C}^{\omega}_{\infty\omega}$ can also be evaluated in PTIME. With the help of Proposition 8.2, the proof of Proposition 8.3 is a straightforward induction on ϕ. □

8.2.2 $\mathcal{L}^{*}_{\infty}$ Captures PTIME

We present, in this section, the proof of the fact that the logic $\mathcal{L}^{*}_{\infty}$ captures PTIME on the class of structures used in Lidia and presented in Section 7.4.

First of all, we can derive Corollary 8.1 from Proposition 8.1 and Proposition 8.2. Hence, we have a direct relationship between the language $\mathcal{L}^{*}_{\infty}$ and the well-known logic $(IFP + C)$.

Corollary 8.1 $(IFP + C) \subseteq \mathcal{L}^{*}_{\infty}$.

Definition 8.2 *We define* $\mathfrak{F}$ *as the class of all (finite) structures. If* L *is a logic, then* $L[\tau]$ *denotes the set of all* $L-$*formulas of vocabulary* τ.

Classical definitions of a logic, such as the notion of *regular logic* (see [Ebb85]) do not suffice for our purpose; in addition our logics are supposed to satisfy certain effectivity conditions. Gurevich [Gur88] suggested the following definitions.

Definition 8.3 *A logic is a pair* $(L, \models_L)$ *where* L *is a function that assigns a recursive set* $L[\tau]$ *of sentences to each vocabulary* τ, *and* $\models_L$ *is a binary relation between sentences and structures such that for all* $\phi \in L[\tau]$ *the class* $Mod(\phi) = \{\mathcal{A} \in \mathfrak{F} | \mathcal{A} \models_L \phi\}$ *is an isomorphism closed class of* τ*-structures.*

Definition 8.4 *A class* $\mathfrak{C}$ *of* τ*-structures is definable in* $(L, \models_L)$ *if there is an* $L[\tau]$*-sentence* ϕ *such that* $\mathfrak{C} = Mod(\phi)$.

Now we can give a better definition that should make clear what we really mean when we say that a logic (effectively) captures a complexity class.

Definition 8.5 *A logic* $\mathfrak{L} = (L, \models_L)$ *effectively captures a complexity class* K *on a domain* D *if the following two conditions hold:*

(i) *Each* K*-computable class in* D *is* $\mathfrak{L}$*-definable.*

(ii) *For all vocabularies τ there is a recursive mapping M that associates a clocked K-Turing machine $M(\phi)$ with each sentence $\phi \in L[\tau]$ so that if $Mod(\phi) \in D$ then $M(\phi)$ accepts the language $\mathcal{L}(Mod(\phi))$.*

Clearly, a logic that effectively captures a complexity class captures the class as defined in Definition 8.5. The archetypal example of the above definition is given by Fagin's Theorem:

Theorem 8.1 (Fagin [Fag74]) *Existential second-order logic $\sum_1^1$ captures NPTIME.*

The proof of Fagin's Theorem also shows that second-order logic captures the polynomial hierarchy PH. Moreover, it is known that the extension of second-order logic by the partial fixed-point operator captures PSPACE (see [EF95]). Now we are going to state some capturing results concerning the logic (IFP + C).

(i) (Immerman, Lander [IL90]) (IFP + C) effectively captures PTIME on the class of trees.

(ii) (Grohe, Marino [GM99]) For each $k \geq 1$, (IFP + C) effectively captures PTIME on the class of graphs of tree-width at most k.

Theorem 8.2 *The language $\mathcal{L}^*_\infty$ effectively captures PTIME on the class of structures introduced in Section 7.4.*

Proof. The proof of this theorem is due to the characteristics of the logic (IFP + C) associated with the consequences of corollary 8.1. □
Theorem 8.2 leads to the main result of this section. In fact, we can use the properties of the logic $\mathcal{L}^*_\infty$ to give a class of distributed problems that can be encoded and solved by Lidia.

Note 8.2 *Even though the languages* (IFP+C) *and $\mathcal{L}^*_\infty$ have the same expressive power in finite structures, we further prefer to use $\mathcal{L}^*_\infty$ to express the rules' preconditions in Lidia. This preference can be legitimated by two reasons. First, we want to keep our logic as simple as possible. This means that its syntax should not be far away from* FOL. *Thus, we avoid using any logical operator like the one used in fixpoint logic. Moreover, the interesting results of* (IFP+C) *are limited to trees (graphs) of bounded tree-width. Unfortunately, future works on Lidia could involved the expression of rules' preconditions in graphs with unbounded tree-width. In this case, the use of a "simpler" formalism like* (IFP+C) *could be inadequate.*

Let $\mathcal{F}$ denotes the class of distributed problems that can be expressed in a precondition/effect style where all the preconditions can be evaluated in PTIME.

Theorem 8.3 *For every computable problem f that belongs to the class $\mathcal{F}$ there exists a Lidia program that computes f.*

Proof. The preconditions of any problem $f \in \mathcal{F}$ can be computed in Lidia (see Theorem 8.2). Moreover, the language LASL used to characterize the relabeling steps in Lidia has enough computation power to compute every computable function. Hence, there exists a Lidia program that computes f. □

8.3 Computational Completeness of LIDiA

We are now ready to state our main result, namely that every computable distributed problem is computed by a Lidia program. We introduce, therefore, some definitions and notations that will help us to bring out the core of the proofs we will state later.

Definition 8.6 *We define $\mathcal{W}$ as the class of all Lidia programs.*

Definition 8.7 *We define $\mathcal{C}$ as the class of distributed algorithms that can be encoded by means of local computations.*

It is clear that each instance of $\mathcal{C}$ can be expressed in a *precondition/effect* style and that all nodes and edges of the network have labels that describe their state variables at each computation step.

Theorem 8.4 *(Completeness of Lidia) For every computable $f \in \mathcal{C}$ there exists a Lidia program that computes f.*

The proof of Theorem 8.4 follows from the following lemmas.

Let $f \in \mathcal{C}$ be a distributed problem and P_f be the set of preconditions that appear in the rules of f. We assume that the network has n vertices.

Lemma 8.1 $\forall p \in P_f, p \in PTIME \Rightarrow \exists f_p \in \mathcal{W}$ *that computes* f.

Proof. This lemma is a direct consequence of Theorem 8.3. Note that we use the notation "$p \in$ PTIME" to express the fact that the complexity needed for the execution of the query p is polynomial. □

Remark 8.1 *Lemma 8.1 is of prime importance for the design of Lidia. In fact, it is clear that as long as all preconditions of a given problem are in PTIME, we do not need to introduce user defined functions in the logic $\mathcal{L}^*_\infty$. The next lemma states the case where we face a precondition, that is not in PTIME. Such preconditions are not common, but they exist.*

Example 8.2 *Consider the case where all the neighbors P_i of a process P are labeled with numbers x_i, and process P has to answer the query $D \leq k$? with $D := max(\sum_{P_i \in S} x_i)$, $S \subseteq V$ and $|S| \leq T$ (For given k and T). Solving this query is the same as solving the maximum set packing problem that is known to be NP-complete.*

Lemma 8.2 $\forall p \in P_f, p \notin PTIME \Rightarrow \exists f_p \in \mathcal{W}$ *that computes* f.

Proof. Without loss of generality, let $p \notin$ PTIME, $\mathcal{A}_p$ be the class of elements (sets of star graphs of diameter 2) that satisfy p and $\mathcal{O}_r(u, \mathcal{A}_p)$ an oracle that is *true* if the ball of radius 1 centered on u belongs to the class $\mathcal{A}_p$. Thus, for any node v we can express the precondition p of any rule using the oracle $\mathcal{O}_r(v, \mathcal{A}_p)$. The oracle will return *true* if the precondition p is satisfied and false otherwise. Hence, the above Lemma is satisfied if and only if the oracle $\mathcal{O}_r$ can be implemented in Lidia as an user defined function. In this case, the function $\mathcal{O}_r(u, \mathcal{A}_p)$ is represented outside the precondition of p and thus outside the language $\mathcal{L}^*_\infty$. □

Remark 8.2 *In the proof of Lemma 8.2, we do not mean that Lidia is able to tackle computing problems that are* NP-Hard. *Our aim was rather to show that, regardless of the complexity of a given precondition, it is possible to express it in Lidia. This is only possible if users defined functions are allowed in the design of Lidia.*

Theorem 8.5 *(General Completeness of Lidia) The class of problems that are computed by Lidia programs is exactly the class $\mathcal{C}$.*

Proof. Because of Theorem 8.4, Theorem 8.5 is obvious. In fact, Theorem 8.4 states the completeness of the language Lidia. This means that any distributed algorithm encoded by local computations can be implemented in Lidia. Furthermore, all computational actions in Lidia are local in the sense that only network computations in a ball of radius 1 are allowed. Thus, any distributed algorithms designed in Lidia can be encoded as a list of local computations rules. □

8.4 Concluding Remarks

As we have seen, Lidia is exactly devoted to the design and implementation of distributed algorithms encoded by means of local computations. Although Lidia and other languages like IOA use *transition definition* (guarded commands) consisting of *preconditions* and *effects*, the preconditions in Lidia are exclusively described in the logic $\mathcal{L}^*_\infty$. We have proved, in this chapter, that this logic has enough descriptive power to fully describe all *PTIME* queries in the structures used in Lidia. This particularity of $\mathcal{L}^*_\infty$ helped us to state the completeness of Lidia in presence of user defined functions. Hopefully, distributed algorithms encoded by local computations where preconditions do not belong to the PTIME complexity class are not usual. For this reason, we have restricted the user defined functions to procedures that can be easily implemented in the LASL language.

Conclusion and Perspectives

In this thesis we have studied three major problems related to local computations in particular and to distributed computations in general.

The first problem concerns the execution of synchronous algorithms in fully asynchronous networks systems. To do this, we have develop network protocols (synchronizers) that are able to synchronize network processes having some network knowledge. The main purpose of synchronizers is to execute the computation steps of each process in round, such that the round difference of any two processes is at most 1 and all processes have to be in the same round before any of them starts the next round. We have presented different types of synchronizers that differ in the required network informations. The first type deals with synchronizers having no informations about the network. We have shown that this kind of synchronizers are not able to synchronize the executions of processes that are at distance more than 1. In the second type, we have presented a synchronizer that need to have knowledge of the network diameter. This protocol takes advantage of the SSP algorithm [SSP85] to check if all processes have reached the actual round before starting the next one. The idea of the third type is based on the use of random works in a network with known size. We have modeled the execution of such a synchronizer as a Markov chain and we were able to state its correctness. Further on, we have considered synchronizers dedicated to tree-shaped networks and to networks with a distinguished process. We have proved that all these synchronizers can be implemented by means of local computations.

The second problem we have considered is about the algorithmic recognition of graph properties with local computations. We proposed a solution based on the execution of reduction algorithms in the local computations framework. We were able to present the first distributed algorithm to encode reduction algorithms. This has induced the introduction of the concept of *handy reduction systems* that can be seen as reduction systems for distributed environments. Starting from this algorithm, we have presented a procedure that solves graph decision problems in distributed network systems. Inspired by the work of Bodlaender et al. [BvAdF01] we have proved the correctness of an algorithm that solves a graph decision problem and computes, if possible, a witness solution for the input problem. At the end, we have stated a direct relationship between *handy reduction systems* and *labeled graph recognizers with structural knowledge*. We stated that any *handy reduction systems* can be seen as a *labeled graph recognizer with structural knowledge* and that, as a corollary, all MS-definable properties on graph of bounded treewidth are size recognizable. The implementation of reduction systems by local computations has necessitated the solution of k-local elections with $k \geq 3$. To this end, we have introduced a probabilistic methodology based on distributed computations of rooted trees of minimal paths. At last we have presented a protocol that, given a distributed

algorithm, checks the validity of given properties at run time. This protocol extends the framework of local computations with a tool that helps to state the correctness of program executions.

The third major problem addressed in this thesis concerns the development of a programming language for implementing distributed algorithms encoded by local computations. Our general approach has consisted in defining a two-level transition systems based language called *Lidia*. Each transition is represented in the precondition-effect style. In Lidia, we use the logic $\mathcal{L}^*_\infty$ to express the precondition part of each transition and we stated that $\mathcal{L}^*_\infty$ has enough descriptive power to fully express all preconditions that can be evaluated in PTIME. This has permitted us to state the computational power of Lidia and to show that in presence of user-defined functions, Lidia is able to describe any distributed algorithm encoded by local computations.

Perspectives

A first topic for future research that we consider particularly important treats of network synchronization by mobile agents. In the first part of this thesis we have presented a synchronizer based on random walks of a set of tokens without computation power. We have seen that the correctness of this protocol is strongly addicted to the fact that the network size is known by all processes. In this representation, the number of tokens is the same as the network size. The idea here is to replace the tokens by a set of mobile agents (mobile token with computation power) and to see if the synchronization can be performed under the same requirements. It also arouses our interest to check if weaker network informations are enough for the fulfillment of the network synchronization. More precisely, it could be of importance to define the necessary conditions for a faithful mobile agents synchronization. These could include the mobile agents computation model, the minimal number of agents, the presence of sense of direction and the use of anonymous or identified mobile agents. All these conditions could be associated with anonymous networks or networks with unique processes identifiers. On the other hand, dealing with mobile agents could lead us to have a look on the minimal number of computation steps required for synchronizing a network.

The second direction of research consists of enlarging the application area of reduction algorithms. We have seen that for each MS-definable property P of graphs of bounded treewidth there exists a handy reduction system that decides P in a distributed network system. We think that it could also be interesting to give an equivalent characterization for MS-definable properties of graphs of bounded clique-width. It is understood that the corresponding theoretical aspects have first to be established for sequential executions.

In the same way, reduction algorithms have been, up to now, exclusively used to solve graph decision problems. These algorithms generally match a graph G to a boolean value. A potential way to increase the power of graph reduction algorithms is to compute specific functions while reducing the network. The inputs of these functions could then be represented by the labels attached to the graph components (edges or vertices) that have to be reduced. For instance, let r be the reduction rule consisting in removing one of the vertices of a pendant edge having degree 1. Obviously, if we execute rule r iteratively on a graph until

we obtain an irreducible graph containing exactly one vertex, then the input graph is a tree. Suppose now that all the vertices are i-labeled (initially $i = 1$ for all vertices), and each time a pendant edge is removed, the label of its remaining endpoint is increased by 1. After an iterative execution of r on a graph G, if G is a tree, then the number of its vertices is given by the label attached to the remaining vertex. A resembling technique has soon been used by Okada et al. [OH91] for solving problems related to network reliability. Our intuition and the first attempts to design a formal model for this kind of reduction rules permit us to think that more complicated problems on graphs can be solved using this technique.

A further direction of future work is certainly concerned with the enrichment of the Lidia programming platform. Foremost, we intend to specify and implement a virtual machine for the language Lidia. Our aim is to reach an implementation similar to the one introduced in [MÖ3] and presenting a virtual machine for the programming language introduced by Habel et al. [HP02]: *A core language for graph transformation*. The immediate consequence of such a specification would be to have a platform independent implementation of Lidia and to have the possibility to take advantage of tools like MPI for implementing parallel executions in Lidia.

Bibliography

[ACPS93] S. Arnborg, B. Courcelle, A. Proskurowski, and D. Seese. An algebraic theory of graph reduction. *Journal of the ACM*, 40(5):1134–1164, November 1993.

[AF91] K. R. Abrahamson and M. R. Fellows. Finite automata, bounded treewidth, and well-quasiordering. In *Graph Structure Theory*, pages 539–564, 1991.

[ALS91] S. Arnborg, J. Lagergren, and D. Seese. Easy problems for tree-decomposable graphs. *J. Algorithms*, 12(2):308–340, 1991.

[Ang80] D. Angluin. Local and global properties in networks of processors. In *Proceedings of the 12th Symposium on Theory of Computing*, pages 82–93, 1980.

[AP86] S. Arnborg and A. Proskurowski. Characterization and recognition of partial 3-trees. *SIAM J. Algebraic Discrete Methods*, 7(2):305–314, 1986.

[AP90] B. Awerbuch and D. Peleg. Network synchronization with polylogarithmic overhead. In *IEEE Symp. on Foundations of Computer Science*, pages 514–522, 1990.

[APC90] S. Arnborg, A. Proskurowski, and D. G. Corneil. Forbidden minors characterization of partial 3-trees. *Discrete Math.*, 80(1):1–19, 1990.

[AV91] S. Abiteboul and V. Vianu. Generic computation and its complexity. *In proc. ACM Symp. on Theory of Computing*, New Orleans May 1991.

[AW98] H. Attiya and J. Welch. *Distributed computing.* McGraw-Hill, 1998.

[Bar96] V. C. Barbosa. *An introduction to distributed algorithms.* MIT Press, 1996.

[BC87] M. Bauderon and B. Courcelle. Graph expressions and graph rewritings. *Mathematical Systems Theory*, 20(2-3):83–127, 1987.

[BCG+96a] P. Boldi, B. Codenotti, P. Gemmell, S. Shammah, J. Simon, and S. Vigna. Symmetry breaking in anonymous network: Characterizations. In *Proc. 4th Israeli Symposium on Theory of Computing and Systems.* IEEE Press, 1996.

[BCG+96b] P. Boldi, B. Codenotti, P. Gemmell, S. Shammah, J. Simon, and S. Vigna. Symmetry breaking in anonymous networks: Characterizations. In *Proc. 4th Israeli Symposium on Theory of Computing and Systems*, pages 16–26. IEEE Press, 1996.

[BdF96] Hans L. Bodlaender and Babette de Fluiter. Parallel algorithms for series parallel graphs. In LNCS Sprinber verlag Berlin, editor, *4th annual European Symposium on Algorithms (ESA 96)*, volume 1136, pages 277–289, 1996.

[BE04] A. Birka and M. D. Ernst. A practical type system and language for reference immutability. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2004)*, pages 35–49, Vancouver, BC, Canada, October 26–28, 2004.

[BFG+93] Manfred Broy, Christian Facchi, Radu Grosu, Rudi Hettler, Heinrich Hussmann, Dieter Nazareth, Oscar Slotosch, Franz Regensburger, and Ketil Stølen. The requirement and design specification language Spectrum, an informal introduction (V 1.0), part 1 & 2. Technical Report TUM-I9312, 1993.

[BGM+01] M. Bauderon, S. Gruner, Y. Métivier, M. Mosbah, and A. Sellami. Visualization of distributed algorithms based on labeled rewriting systems. In *Second International Workshop on Graph Transformation and Visual Modeling Techniques, Crete, Greece*, July 12-13, 2001.

[BH98] Hans L. Bodlaender and Torben Hagerup. Parallel algorithms with optimal speedup for bounded treewidth. *SIAM Journal on Computing, 27,*, pages 1725–1746., 1998.

[BL86] H.-L. Bodlaender and J. Van Leeuwen. Simulation of large networks on smaller networks. *Information and Control*, 71:143–180, 1986.

[BLW87] M. W. Bern, E. L. Lawler, and A. L. Wong. Linear-time computation of optimal subgraphs of decomposable graphs. *J. Algorithms*, 8(2):216–235, 1987.

[BM00] H. Bunke and B. Messmer. Efficient subgraph isomorphism detection: a decomposition approach. In *IEEE Trans. on Knowledge and Data Engineering 12, No 2*, pages 307 – 323, 2000.

[BMMS02] M. Bauderon, Y. Métivier, M. Mosbah, and A. Sellami. From local computations to asynchronous message passing systems. Technical Report 1271-02, LaBRI-University of Bordeaux I, 2002.

[Bre99] P. Bremaud. *Markov Chains: Gibbs Fields, Monte Carlo Simulation, and Queues.* Springer New York, 1999.

[BST89] Henri E. Bal, Jennifer G. Steiner, and Andrew S. Tanenbaum. Programming languages for distributed computing systems. *ACM Comput. Surv.*, 21(3):261–322, 1989.

[BV99] P. Boldi and S. Vigna. Computing anonymously with arbitrary knowledge. In *Proceedings of the 18th ACM Symposium on principles of distributed computing*, pages 181–188, 1999.

[BvAdF01] Hans L. Bodlaender and Babette van Antwerpen-de Fluiter. Reduction algorithms for graphs of small treewidth. *Inf. Comput.*, 167(2):86–119, 2001.

[Cha99] J. Chang. *Stochastic processes.* 1999.

[CK03] Punit Chandra and Ajay D. Kshemkalyani. Distributed algorithm to detect strong conjunctive predicates. *Information Processing Letters 87*, 2003.

[CL85] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems, 3(1)*, pages 63–75, 1985.

[CL96] B. Courcelle and J. Lagergren. Equivalent definitions of recognizability for sets of graphs of bounded treewidth. *Mathematical Structures in Computer Science*, 6(2):141–165, 1996.

[CLR69] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, Massachusetts, 1969.

[CM88] K. M. Chandy and J. Misra. *Parallel Programs Design: A Foundation.* Addison-Wesley Publishing Co., Reading, MA, April 1988.

[CM91] R. Cooper and K. Marzullo. Consistent detection of global predicates. *In Proc. ACM/ONR workshop on parallel and distributed Debugging*, May 1991.

[CM04] J. Chalopin and Y. Métivier. Election and local computations on edges (*extended abstract*). In *Proc. of Foundations of Software Science and Computation Structures, FOSSACS'04*, number 2987 in LNCS, pages 90–104, 2004.

[CMZ04] J. Chalopin, Y. Métivier, and W. Zielonka. Election, naming and cellular edge local computations (*extended abstract*). In *Proc. of International conference on graph transformation, ICGT'04*, number 3256 in LNCS, pages 242–256, 2004.

[Cou90a] B. Courcelle. Graph rewriting: An algebraic and logic approach. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics (B)*, pages 193–242. 1990.

[Cou90b] B. Courcelle. The monadic second-order logic of graphs I. Recognizable sets of finite graphs. *Inf. Comput.*, 85(1):12–75, 1990.

[Cou97] B. Courcelle. The expression of graph properties and graph transformations in monadic second-order logic. *Handbook of graph grammars and computing by graph transformation*, 1:313–397, 1997.

[CTW93] Don Coppersmith, Prasad Tetali, and Peter Winkler. Collisions among random walks on a graph. *SIAM J. Discret. Math.*, 6(3):363–374, 1993.

[CV00] Jordi Cortadella and Gabriel Valiente. A relational view of subgraph isomorphism. In *RelMiCS*, pages 45–54, 2000.

[Duf65] R. J. Duffin. Topology of series-parallel networks. *Journal of Math. Analysis and Applications*, 10:303–318, 1965.

[Ebb85] H.-D. Ebbinghaus. *Extended logics: The general framework.* In J. Barwise and S. Feferman, editors, 1985.

[EF95] H.-D. Ebbinghaus and J. Flum. *Finite model theory.* Springer, 1995.

[EKMR99] Hartmut Ehrig, Hans-Jörg Kreowski, Ugo Montanari, and Grzegorz Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 3: Concurrency, Parallelism, and Distribution.* World Scientific, 1999.

[Fag74] R. Fagin. Generalized first-order spectra and polynomial-time recognizable sets. pages 43–73, 1974.

[Fid88] C.J. Fidge. Timestamps in message-passing systems that preserve partial ordering. *Australian Computer Science Comm.10 (1)*, 1988.

[FLM86] M. J. Fisher, N. A. Lynch, and M. Merritt. Easy impossibility proofs for distributed consensus problems. *Distrib. Comput.*, 1:26–29, 1986.

[FLMW94] A. Fekete, N. Lynch, M. Merritt, and W. Weihl. *Atomic Transactions.* Morgan Kaufmann Publishers, San Mateo, CA, 1994.

[GG98] E. Grädel and Y. Gurevich. Metafinite model theory. *Information and computation 140*, pages 26–81, 1998.

[GLV97] S. Garland, N. Lynch, and M. Vaziri. Ioa: A language for specifying, programming and validating distributed systems. *Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA*, 1997.

[GM97] Vijay K. Garg and J. Roger Mitchell. Detecting conjunctions of global predicates. *Information Processing Letters*, 63(6):295–302, 1997.

[GM99] M. Grohe and J. Marino. Definability and descriptive complexity on databases of bounded treewidth. In *In C. Beeri. proceedings of the 7th international Conference on Database Theory, lecture notes in computer science. Springer Verlag*, 1999.

[GM02] E. Godard and Y. Métivier. A characterization of families of graphs in which election is possible. In *Foundations of Software Science and Computation Structures*, Lecture notes in computer science, pages 159–172. Spinger-Verlag, 2002.

[GMM00] E. Godard, Y. Métivier, and A. Muscholl. The power of local computations in graphs with initial knowledge. In *Theory and applications of graph transformations*, volume 1764 of *Lecture notes in computer science*, pages 71–84. Spinger-Verlag, 2000.

[GMM04] Emmanuel Godard, Yves Métivier, and Anca Muscholl. Characterizations of classes of graphs recognizable by local computations. *Theory Comput. Syst.*, 37(2):249–293, 2004.

[GS86] Y. Gurevich and S. Shelah. Fixed-point extensions of firdt-order logic. *Annals of Pure and Applied Logic 32*, pages 265–280, 1986.

[GT95] S. Grumbach and C. Tollu. On the expressive power of counting. *Theor. Comput. Sci.*, 149(1):67–99, 1995.

[Gur88] Y. Gurevich. Logic and the challenge of computer science. *In E. Börger, editor, Current trends in theoretical computer science.*, pages 1–57, 1988.

[GW92] V. K. Garg and B. Waldecker. Detection of unstable predicates in distributed programs. *Lecture Notes in Computer Science*, 652:253–264, December 1992.

[Har69] F. Harary. *Graph Theory.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1969.

[HLN99] L. Hella, L. Libkin, and J. Nurmonen. Notions of locality and their logical characterizations over finite models. *Journal of symbolic logic 64*, pages 1751–1773, 1999.

[HP02] A. Habel and D. Plump. A core language for graph transformation (extended abstract). In *Applied Graph Transformation (AGT 2002)*, pages 187–199, 2002.

[HT74] John Hopcroft and Robert Tarjan. Efficient planarity testing. *J. ACM vol. 21, no. 4*, pages 549–568, 1974.

[HT94] Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425, Ithaca, NY, USA, 1994.

[Hui03] W. Huisinga. Markov chains: A complete description, 2003.

[IL90] N. Immerman and E. Lander. Describing graphs: A first-order approach to graph canonization. *In A. Selman, editor, Complexity theory restrospective*, pages 59–81, 1990.

[Imm86] N. Immerman. Relational queries computable in polynomial time. *Information and Control*, pages 25:76–98, 1986.

[Klo94] T. Kloks. Treewidth: computations and approximations. Lecture note in computer science vol. 842, 1994.

[KY96] T. Kameda and M. Yamashita. Computing on anonymous networks: Part i - characterizing the solvable cases. *IEEE Transactions on parallel and distributed systems*, 7(1):69–89, 1996.

[Lam78] L. Lamport. Time, clocks and the ordering of events in a distributed system. *System Comm. ACM 21(7)*, pages 558–565, 1978.

[Lam89] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems, 16(3)*, pages 872–923, May 1989.

[Lav95] C. Lavault. *Évaluation des algorithmes distribués*. Hermes, 1995.

[LL90] L. Lamport and N. Lynch. Distributed computing: Models and methods. *In J. van Leewen, editor, Handbook of Theoretical Computer Science, volume B: Formal Models and Semantics, chapter 19*, pages 1157–1199, The MIT Press, New York, NY, 1990.

[LMS95] I. Litovsky, Y. Métivier, and E. Sopena. Different local controls for graph relabelling systems. *Math. Syst. Theory*, 28:41–65, 1995.

[LMS99] I. Litovsky, Y. Métivier, and E. Sopena. Graph relabelling systems and distributed algorithms. In H. Ehrig, H.J. Kreowski, U. Montanari, and G. Rozenberg, editors, *Handbook of graph grammars and computing by graph transformation*, volume 3, pages 1–56. World Scientific, 1999.

[LMZ95] Igor Litovsky, Yves Métivier, and Wiesław Zielonka. On the recognition of families of graphs with local computations. *Information and Computation*, 118(1):110–119, April 1995.

[Lyn89] N. Lynch. A hundred impossibility proofs for distributed computing. In *8th International Conference on Distributed Computing Systems*, pages 1–28, 1989.

[Lyn96] N. A. Lynch. *Distributed algorithms.* Morgan Kaufman, 1996.

[MÖ3] Andreas Möller. Eine virtuelle Maschine für Graphprogramme, Diplomarbeit, Carl von Ossietzky Universität Oldenburg, 2003.

[Mas91] W. S. Massey. *A basic course in algebraic topology.* Springer-Verlag, 1991. Graduate texts in mathematics.

[Mat89] F. Mattern. Virtual time and global states of distributed systems. *In Parallel and Distributed Algorithms, North-Holland*, 1989.

[Mat93] F. Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing*, 18(4):423–434, 1993.

[Maz87] A. Mazurkiewicz. Trace theory. In W. Brauer et al., editor, *Petri nets, applications and relationship to other models of concurrency*, volume 255 of *Lecture notes in computer science*, pages 279–324. Spinger-Verlag, 1987.

[Maz97] A. Mazurkiewicz. Distributed enumeration. *Inf. Processing Letters*, 61:233–239, 1997.

[MMOS04] Yves Métivier, Mohamed Mosbah, Rodrigue Ossamy, and Afif Sellami. Synchronizers for local computations. In *Second International Conference on Graph Transformations (ICGT)*, volume 3256 of *Lecture Notes in Computer Science*, pages 271–286. Springer Verlag, Rome, Italy, September 28 - October 2, 2004.

[MMW97] Yves Métivier, Anca Muscholl, and Pierre-André Wacrenier. About the local detection of termination of local computations in graphs. In D. Krizanc and P. Widmayer, editors, *SIROCCO 97 - 4th International Colloquium on Structural Information & Communication Complexity*, Proceedings in Informatics, pages 188–200. Carleton Scientific, 1997.

[MO04a] M. Mosbah and R. Ossamy. Checking global properties for local computations in graphs with applications to invariants testing. In *IEEE Proc. of the Fifth Mexican International Conference on Computer science*, pages 35–42, 20-24 September 2004.

[MO04b] M. Mosbah and R. Ossamy. LIDiA: A programming language for local computations in graphs. Technical Report 1314-04, LaBRI University of Bordeaux I, 2004.

[MO04c] M. Mosbah and R. Ossamy. A programming language for local computations in graphs. In *8th. MCSEAI*, 9-12 May, Sousse Tunesia, 2004.

[MO04d] M. Mosbah and R. Ossamy. A programming language for local computations in graphs: Computational completeness. In IEEE, editor, *Proceedings of the 5th. Mexican International Conference in Computer Science*, Colima 20-24 September 2004.

[Moo59] E. F. Moore. The shortest path through a maze. In *Proceedings of an International Symposium on the theory of Switching*, pages 285–292. Cambridge, Massachusetts, Harvard University Press, 2-5 April 1959.

[MS] M. Mosbah and A. Sellami. Visidia: A tool for the visualization and simulation of distributed algorithms. *http://www.labri.fr/visidia/*.

[MSZ02] Y. Métivier, N. Saheb, and A. Zemmari. Randomized local elections. *Inform. Proc. Letters*, pages 313–320, 2002.

[MT91] Jirí Matousek and Robin Thomas. Algorithms finding tree-decompositions of graphs. *J. Algorithms*, 12(1):1–22, 1991.

[NE01] J. W. Nimmer and M. D. Ernst. Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java. In *Proceedings of RV'01, First Workshop on Runtime Verification*, Paris, France, July 23, 2001.

[OH91] Yasuyoshi Okada and Masahiro Hayashi. Graph rewriting systems and their application to network reliability analysis. In *17th International Workshop, WG '91, Fischbachau*, volume 570 of *Lecture Notes in Computer Science*, pages 36–47, Germany, June 17-19, 1991.

[Oss04] R. Ossamy. LASL: A language for assignation statements in Lidia. Technical Report 1339-04, LaBRI, University of Bordeaux I, 2004.

[Oss05a] Rodrigue Ossamy. A simple randomized k-local election algorithm for local computations. In *4th International Workshop on Experimental and Efficient Algorithms, WEA 2005, Santorini Island, Greece, May 10-13, 2005, Proceedings*, volume 3503 of *Lecture Notes in Computer Science*, pages 290–301. Springer Verlag, 2005.

[Oss05b] Rodrigue Ossamy. A simple randomized k-local election algorithm for local computations. Technical Report 1344-05, LaBRI-University of Bordeaux I, 2005.

[Ott96] M. Otto. The expressive power of fixed-point logic with counting. *Journal of symbolic Logic*, 61:147–176, 1996.

[Ott97] M. Otto. *Bounded Variable logics and Counting*. Springer-Verlag, 1997.

[Pel00] D. Peleg. *Distributed computing - A locality-sensitive approach*. SIAM Monographs on discrete mathematics and applications, 2000.

[Rei32] K. Reidemeister. Einführung in die kombinatorische Topologie. 1932.

[RFH72] P. Rosenstiehl, J.-R. Fiksel, and A. Holliger. Intelligent graphs. In R. Read, editor, *Graph theory and computing*, pages 219–265. Academic Press (New York), 1972.

[RS83] N. Robertson and P.D. Seymour. Graph minors I. Excluding a forest. *J. Combin. Theory*, Series B 35:39–61, 1983.

[RS86] N. Robertson and P.D. Seymour. Graph minors. II. Algorithmic aspects of treewidth. *J. Algorithms*, 7:309–322, 1986.

[RS87] N. Robertson and P.D. Seymour. Some new results on the well-quasi ordering of graphs. *Ann. Discr. Math.*, 23:343–354, 1987.

[Sch89] P. Scheffler. *Die Baumweite von Graphen als ein Maßfür die Kompliziertheit algorithmischer Probleme.* PhD thesis, Akademie der Wissenschaften der DDR, Berlin, 1989.

[Sen80] E. Seneta. *Non-negative matrices and Markov Chains 2nd. ed.* Springer-Verlag New York, 1980.

[SSP85] Y. Shi, B. Szymanski, and N. Prywes. Terminating iterative solutions of simultaneous equations in distributed message passing systems. In *4th International Conference on Distributed Computing Systems*, pages 287–292, 1985.

[Tel91] G. Tel. *Topics in distributed algorithms.* Cambridge University Press, New York, NY, USA, 1991.

[Tel00] G. Tel. *Introduction to distributed algorithms.* Cambridge University Press, 2000.

[Ull76] J. R. Ullman. An algorithm for subgraph isomorphism. In *Journal of the ACM, 23(1)*, pages 31–42, January 1976.

[vAdF97] Babette van Antwerpen-de Fluiter. *Algorithms for graphs of small treewidth.* PhD thesis, University of Utrecht 1997.

[Var95] M. Y. Vardi. On the complexity of bounded-variables queries. In *In Proceedings of the 14-th ACM Symposium on Principles of Database System*, pages 266–276, 1995.

[VTL79] Jacobo Valdes, Robert E. Tarjan, and Eugene L. Lawler. The recognition of series parallel digraphs. In *STOC '79: Proceedings of the eleventh annual ACM symposium on Theory of computing*, pages 1–12. ACM Press, 1979.

[Weg98] P. Wegner. Interactive foundations of computing. *Theoretical Computer Science*, 192:315–351, 1998.

[YK96] 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.

Index

Q

R

S

T

U