N° d'ordre : 2933

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ BORDEAUX I

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

Par **Afif SELLAMI**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

Des Calculs Locaux aux Algorithmes Distribués

Soutenue le : 13 décembre 2004

Sous la direction :

Michel Bauderon . Professeur
Mohamed Mosbah Professeur

Après avis des rapporteurs :

Stefan Gruner .... Maître de Conférences
Vincent Villain ... Professeur

Devant la commission d'examen composée de :

François Dufour ..	Professeur	Examinateur
Stefan Gruner	Maître de Conférences	Rapporteur
Yves Métivier	Professeur	Président
Mohamed Mosbah	Professeur	Codirecteur de thèse
Eric Sopena	Professeur	Examinateur
Vincent Villain ...	Professeur	Rapporteur

– 2004 –

Des Calculs Locaux aux Algorithmes Distribués

Résumé :L'objectif de cette thèse est de montrer que le modèle des systèmes de réécriture de graphe est un modèle pertinent pour l'étude, la compréhension et l'implémentation des algorithmes distribués.

Dans un algorithme distribué, la détection de terminaison est difficile puisqu'un algorithme est exécuté par plusieurs processus en parallèle et l'algorithme ne se termine que lorsque tous les processus ont fini. Pour résoudre ce problème, nous utilisons un produit particulier de deux systèmes de réécriture de graphe. Le premier code l'algorithme dont nous voulons détecter sa terminaison et le second code un des algorithmes qui détecte la terminaison globale.

Les algorithmes asynchrones sont souvent moins performants que les algorithmes correspondants synchrones. Il est très utile de disposer d'une méthode générale qui permet de simuler les algorithmes synchrones sur des systèmes asynchrones. Nous utilisons un autre produit, similaire à celui utilisé pour la résolution du problème de détection de terminaison, pour résoudre ce problème.

Nous avons implémenté une plate-forme appelée Visidia qui utilise les systèmes de réécriture de graphe pour implémenter les algorithmes distribués. Visidia permet de tester, expérimenter, valider et visualiser l'exécution des algorithmes distribués codés sous forme de systèmes de réécriture de graphe ou sous forme de calcul local.

Nous utilisons Visidia pour faire des expérimentations sur des algorithmes qui résolvent le problème de partage des ressources. Plusieurs algorithmes ont été testés dont la généralisation du problème des philosophes, l'exclusion mutuelle dans un arbre et enfin, une généralisation du problème de l'exclusion mutuelle dans un réseau quelconque.

Discipline : Informatique

Mots-Clefs :

Systèmes de réécriture de graphe ;

Algorithmes distribués ;

Détection de la terminaison ;

Synchroniseurs ;

Visidia ;

Partage de ressources.

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

From local computations to distributed algorithms

Abstract : The goal of this thesis is to show that the model of graph relabelling systems is a relevant model for the study, comprehension and the implementation of distributed algorithms.

In a distributed algorithm, the termination detection of an algorithm is difficult since an algorithm is executed simultaneously by many processes and the algorithm terminates only when all the processes terminate. To solve this problem, we use a particular product of two graph relabelling systems. The first encodes the algorithm for which we want to detect the termination and the second encodes an algorithm which detects locally the global termination.

The asynchronous algorithms are often less powerful than the synchronous corresponding ones. It is very useful to have a general method which makes the simulation of synchronous algorithms on asynchronous systems possible. To do so, we use another product, similar to that used for the termination detection problem, to simulate synchronous systems.

We had implemented a platform called Visidia which uses graph relabelling systems to implement distributed algorithms. With Visidia we can test, experiment, verify and visualize the execution of distributed algorithms encoded by graph relabelling systems or local computations.

We use Visidia to make experiments on algorithms which solve the sharing resource problems. Many algorithms have been tested : the drinking philosophers problem, mutual exclusion in trees and finally, mutual exclusion in random networks.

Discipline : Computer-Science

Keywords :

Graph Relabelling Systems ;

Distributed algorithms ;

Termination detection ;

Synchronizers ;

Visidia ;

Sharing resources.

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

Remerciements

Je tiens tout d'abord à remercier chaleureusement :

- Michel Bauderon et Mohamed Mosbah, mes directeurs de thèse, présents de manière continue durant toutes ces années, et qui ont su me guider dans mes recherches ;
- Stefan Gruner et Vincent Villain qui ont accepté d'être mes rapporteurs de thèse ;
- Yves Métivier, le président du Jury de ma soutenance de thèse, qui m'a beaucoup aidé durant mes années de thèse ;
- François Dufour et Éric Sopena, qui ont accepté d'être membres du jury.

Je remercie également tous les membres du LaBRI, de l'IUT et de l'ENSEIRB que j'ai côtoyés et qui m'ont aidé pendant toutes ces années que j'ai passées loin de mon chez-moi.

Je remercie également tous mes collègues de bureau : Aymeric, Bertrand, Bilel, David, Florian, Pascal A., Pascal G., Pierre H., Pierre V., Sylvie, Yon et Yvan avec qui j'ai passé de très bons moments et qui ont été des frères et une soeur pour moi. Je tiens aussi à remercier tous les autres collègues qui viennent souvent me voir et que je vais souvent embêter dans leurs salles : Abdellaziz, Akka, Alexandre, Anthony, Bertrand, Brahim, David, Fabrice, Guillaume, Jeremy, Laurent, Loic, Manue, Marcien, Mickaël, Nicolas, Olivier, Rodrigue... ;
Je tiens aussi à remercier toute l'équipe de football (les loosers... excusez moi... sciences Labri) pour tous ces matchs inoubliables, qu'on a pu gagner (excusez moi... perdre) avec fierté.
Bien sûr, je remercie tous les membres du groupe PLAYZ : Bilel, David, Florian, Laurent, Petra, Pierre H., Sebastien et Yvan pour les moments de musique qu'on partage ensemble. N'oubliez pas, j'ai toutes les K7 de ce qu'on a fait.

Je remercie finalement mes amis hors du laboratoire surtout Chiheb qui m'a permis de venir à Bordeaux et aussi Bilel, Mehdi et Mohamed, mes amis Tunisiens de Bordeaux. Je n'oublie pas mes amis depuis le lycée, la prépa qui sont en France, au Canada, en Tunisie et ailleurs tel que : Youssef, Gaddour et Abdessattar, mes hébergeurs à Paris, Ali, Mehdi, Meher, Ghazi, Karim, Kais, Rafaa... ;

Enfin, Je remercie mon premier professeur d'informatique, sans qui je ne serais pas si "fort" : Noureddine Zouari, tous les membres de ma famille sans exception ; en particulier, ma soeur Monia, son mari Makram, et ses enfants : Youssef et Aicha et enfin les plus

précieux : mes parents sans qui je serais en train de manger du chocolat chez Nestlé, qui ont tout fait pour que je termine mes études de 3ème cycle et sans qui je ne serais même pas là à écrire ces quelques mots.

Merci beaucoup.

Table des matières

Table des figures

Liste des Algorithmes

Introduction

La communication a permis à l'Homme d'évoluer. Chacun partage ses connaissances, son savoir-faire, ses outils avec sa famille, sa tribu jusqu'à ce que toute l'humanité ait accueilli son savoir-faire. Bien sûr, ce chemin peut durer des années et même des siècles selon la période et les moyens de communication utilisés. Nous utilisons ce même principe mais en remplaçant les Hommes par les machines. Cet ensemble de personnes devient un système distribué, composé d'un ensemble d'entités (hommes/machines ou processeurs) pouvant communiquer entre elles. Ces entités sont indépendantes ou semi-indépendantes. Elles peuvent partager des ressources (des imprimantes, des fichiers), collaborer pour résoudre un problème... Comme pour l'être humain, ces entités ne communiquent directement qu'avec un nombre restreint d'autres entités. Tout cet environnement génère différents types de problèmes : tolérances aux pannes, détection de la terminaison, routage, centralisation de certaines informations...
D'autres difficultés apparaissent, liées à la nature du système : synchrone, asynchrone ou semi-synchrone ; le type d'échange des informations : par échange de messages (comme pour le facteur), utilisation de registres, partage de la mémoire. Un système synchrone signifie l'existence d'horloge globale, et, à chaque top d'horloge, tous les processeurs font une action. A l'inverse, dans les systèmes asynchrones, chaque processeur possède sa propre horloge et n'a aucune connaissance de la vitesse des autres processeurs, d'où la difficulté. Un système à mémoire partagée est un système où chaque entité a sa propre partie de mémoire dans laquelle elle peut lire et écrire. Pour les autres parties, elle ne peut que lire. Un système avec partage de registres ressemble au système avec mémoire partagée, cependant, chaque entité a un registre dans lequel elle peut lire et écrire, et n'a accès en lecture qu'à quelques registres que nous appellerons "voisins". Dans un système à échange de message, chaque entité possède un nombre (limité) de "voisins". Cette entité peut leur envoyer des messages et recevoir des messages de leur part. Ce système a deux modes d'échange de messages : synchrone et asynchrone. Comme pour les systèmes distribués, le mode synchrone indique que le temps de passage d'un message dans un canal est connu et que ce temps est le même pour tous les canaux. A l'inverse du mode synchrone, le mode asynchrone n'a aucune connaissance et n'est pas limité par un temps : une entité 1 qui envoie un message à une entité 2 ne sait pas quand est ce que le message va arriver. De la même manière, l'entité 2 ne sait pas quand est ce que le message a été envoyé. Dans les systèmes distribués, il n'existe *à priori* aucun système de centralisation permettant de coordonner globalement les comportements des processeurs.

Dans la suite, lorsque nous parlons des systèmes distribués, nous considérons des systèmes asynchrones avec échange de messages asynchrones. Dans la grande partie de nos travaux, ce système est aussi anonyme : il n'existe pas d'identifiant unique pour chaque entité. Dans cette thèse, nous nous intéressons à l'étude et à l'implémentation d'algorithmes distribués. Soit $\mathcal{T}$ une tâche à réaliser sur un système distribué $\mathcal{S}$, si $\mathcal{A}$ est un algorithme distribué pour la tache $\mathcal{T}$ alors cet algorithme est capable de faire collaborer la totalité, ou une partie, des processeurs du système $\mathcal{S}$ afin de réaliser $\mathcal{T}$. Si le système $\mathcal{S}$ change, l'algorithme peut changer, et comme il n'existe pas de modèle universel pour décrire les algorithmes distribués, leur compréhension et leurs implémentations deviennent plus difficiles à faire. D'où, l'une des motivations de cette thèse consiste à montrer que le modèle que nous allons utiliser est un modèle pertinent pour l'étude, la compréhension et l'implémentation des algorithmes distribués. Le modèle que nous allons présenter ici, qui poursuit de nombreux travaux initiés par Y. Métivier [46], est le suivant :

Le modèle :

Comme nous l'avons déjà mentionné, le système distribué est un système asynchrone avec échange de messages asynchrone. Il peut avoir une topologie arbitraire, et est supposé fiable : aucune défaillance ne peut intervenir sur les entités ou sur les liens de communications. Le réseau est modélisé par un graphe étiqueté dans lequel les entités ou les processeurs correspondent aux sommets du graphe. Les liens de communications correspondent aux arêtes et les états des processeurs correspondent aux étiquettes. Dans ce cas, nous imposant que : le graphe est connexe ; les délais d'acheminement des messages le long des lignes de communication sont finis mais non bornés ; pour tout couple de processus communicants, l'ordre de réception des messages est identique à leur ordre de transmission (les transmissions sont "fifo") ; le mode de transmission est du "full-duplex",i.e. deux messages peuvant se croiser sur une même ligne.

Dans ce modèle, un algorithme distribué est codé sous forme de système de règles de réécriture. Les réécritures n'affectent pas le graphe sous-jacent, seules les étiquettes changent. Un noeud peut changer son état et l'état d'un, de plusieurs ou de tous ses voisins : si les états d'un noeud et de ses voisins sont préalablement dans une certaine configuration, leurs états changent en fonction de cette configuration. Ainsi, chaque noeud possède son propre état initial, qui peut être le même pour tous les noeuds. Le déroulement de l'algorithme est une suite de réétiquetages. Les réétiquetages peuvent se faire en parallèle si les régions où ce réétiquetage se produisent, sont disjointes.

Ce modèle est abstrait et permet de décrire et de résoudre de nombreux problèmes et paradigmes d'algorithmes distribués. Toute solution dans notre modèle peut être utilisée ou peut indiquer une direction de recherche pour un modèle plus faible.

Notre contribution pour ce modèle est l'élaboration d'une méthode générale permettant de générer un algorithme distribué à partir d'un système de réétiquetage. Ceci nous permet d'étudier de manière unifiée quelques paradigmes de l'algorithmique distribuée. De plus, nous avons développé une plate-forme qui utilise cette méthode pour implémenter les

algorithmes distribués. Des applications à la visualisation et à l'expérimentation ont été effectuées.

La détection de la terminaison :

Un des problèmes des algorithmes distribués est la détection de la terminaison de l'algorithme. Dans le cas d'un algorithme séquentiel, il est facile de détecter sa terminaison. Par contre, dans le cas d'un algorithme distribué, c'est plus délicat. En effet, même si localement un processus a fini ou n'a plus d'instruction à effectuer, il ne peut pas savoir si tous les autres processeurs ont fini ou pas. De plus, il ne peut prédire s'il va recevoir, ou non, un message qui lui donne une instruction ou un changement à exécuter. Un algorithme distribué peut finir sans que les processeurs ne puissent détecter cette terminaison. Dans ce cas on parle de terminaison *implicite* : le calcul est effectivement terminé mais aucun noeud ne le sait. Le but de la détection de terminaison est que la terminaison soit détectable par un ou plusieurs processus : on parle dans ce cas de terminaison *explicite*. Pour simplifier le problème, nous exigeons la détection de la terminaison locale : un sommet sait qu'il a atteint un état final et qu'il n'effectuera aucune opération. La détection peut se faire soit par l'état du noeud lui-même, soit par rapport à son état et aux états de tous ses voisins.

Le synchroniseur :

Notre environnement de travail est un environnement asynchrone : il n'y a pas d'horloge globale et les changement d'état des processeurs se font d'une façon arbitraire et aléatoire. En effet, chaque processeur a sa propre horloge et chaque lien de communication possède sa propre vitesse. Et comme les algorithmes asynchrones sont souvent moins performants que les algorithmes correspondants synchrones, et que leur analyse de complexité est beaucoup plus complexe, il est très utile de disposer d'une méthode générale qui permet de simuler les algorithmes synchrones sur des systèmes asynchrones. Le protocole qui implémente cette méthode est appelé *synchroniseur*. Fondamentalement, un synchroniseur est un protocole distribué de synchronisation par communication qui simule les cycles d'horloge de façon à ce que tout message ne puisse être émis et reçu que durant un cycle. Ceci nous assure que le réseau asynchrone se comporte comme un réseau synchrone du point de vue de l'exécution spécifique de l'algorithme synchrone considéré. Nous ne nous intéressons pas à minimiser le sur-coût engendré par le synchroniseur mais nous voulons nous assurer que tous les noeuds du réseau soient dans la même phase avant de passer à la phase suivante.

L'exclusion mutuelle :

Dans notre modèle, les processus exécutent le même algorithme "séquentiel" et possèdent les mêmes définitions de contextes, ce qui assure une réalisation entièrement distribuée des transitions. Les processus sont alors dit *symétriques*. Ils jouent donc le même rôle et n'ont aucune différence entre eux. Ceci étant, les processus n'exécutent pas toujours la même action "simultanément". Un des paradigmes de l'informatique distribuée consiste à briser la symétrie des processus en octroyant à l'un d'entre eux un "*privilège*". C'est le cas des algorithmes d'élection dans un réseau où on veut élire un seul noeud

du réseau. C'est aussi le cas du problème général de l'exclusion mutuelle. Les problèmes d'exclusion mutuelle diffèrent selon les contraintes du problème lui même. L'exclusion mutuelle simple où, dans un ensemble de processus communicants, il faut attribuer à un seul et unique processus, à un instant donné le privilège d'accéder à une ou plusieurs ressources partagées. Il y a aussi un problème connu sous le nom du problème du dîner des philosophes où cinq philosophes forment un cercle et, entre deux philosophes, il y a une fourchette (baguette). Un philosophe a besoin des deux fourchettes à son coté pour pouvoir manger. Dans ce problème, plusieurs processus peuvent être dans l'état privilégié simultanément.

Visidia :

Un algorithme distribué est généralement difficile à comprendre, à mettre au point et à prouver. Les systèmes de réécriture de graphe permettent de coder les algorithmes distribués afin de simplifier leur étude et leur preuve. Nous avons développé une plate-forme appelée Visidia (**Vi**sualisation and **Si**mulation of **Di**stributed **A**lgorithms) qui utilise ce codage pour implémenter les algorithmes distribués. Cette implémentation peut utiliser des procédures probabilistes qui ne rendent pas l'algorithme distribué probabiliste, mais juste l'exécution des règles qui le sera. Si l'algorithme est déterministe, quelque soient les noeuds qui exécutent les règles, l'algorithme aboutit toujours à un résultat. Bien sûr, l'utilisation de ces procédures n'est pas obligatoire sous Visidia. Un programmeur peut implémenter son algorithme distribué uniquement en utilisant le langage Java et les primitives que propose Visidia pour la communication entre les processeurs et autres informations nécessaires à l'exécution de l'algorithme. Il est possible de visualiser l'exécution d'un algorithme à l'aide de Visidia. En effet, on peut visualiser les messages qui passent entre les noeuds, les changements d'états des noeuds et des arêtes, ainsi que les zones de synchronisation locale (dans le cas d'utilisation des procédures probabilistes). Enfin, Visidia nous permet de faire des expérimentations d'algorithmes distribués, pour avoir une borne de sa complexité en nombre de messages, de synchronisations, etc...

Cette thèse est organisée comme suit :

Partie 1 : Algorithmes distribués codés sous forme de calculs locaux

Dans le chapitre 2, nous introduisons les calculs locaux et les systèmes de réécriture de graphe. Dans ce chapitre, nous donnons les définitions, les méthodes de preuve et des exemples de systèmes de réécriture de graphe obtenus par [45, 12, 14, 10, 44, 52]. Ce chapitre est organisé comme suit :

Nous introduisons les définitions et exemples des systèmes de réécriture (ou réétiquetage) de graphe dans la section 2.2. Puis nous introduisons les méta-règles de réécriture de graphe, dans la section 2.3. Une méta-règle est un compactage de plusieurs règles de réécriture en une seule règle générique. Dans la section 2.4, nous donnons une technique pour prouver les calculs locaux codés sous forme de systèmes de réécriture de graphe.

Enfin, nous proposons des systèmes de réécriture de graphe qui calculent un arbre recouvrant dans un réseau avec identité, chaque noeud possède un identifiant unique, dans la section 2.5.

Le chapitre 3 est consacré au problème de détection de la terminaison d'un algorithme distribué. Les travaux de [57, 34] nous ont motivés pour la réalisation d'algorithmes distribués codés sous forme de systèmes de réécriture de graphe qui résolvent ce problème. Nos travaux ont abouti aux résultats publiés dans [33]. Ce chapitre est organisé comme suit :

Dans la section 3.2, nous introduisons une nouvelle opération sur les systèmes de réécriture de graphe, c'est le produit de deux systèmes de réécriture. Dans la section 3.3, nous utilisons le produit légèrement modifié pour détecter la terminaison des algorithmes distribués qui ont la propriété de terminaison implicite. Un nouvel algorithme est obtenu par le produit de l'algorithme qui ne détecte pas sa terminaison avec un algorithme qui rend la terminaison implicite en terminaison explicite. Enfin, dans la section 3.4, nous donnons un résultat général pour détecter localement la terminaison globale.

La première partie se termine par le chapitre 4. Dans ce chapitre nous proposons des algorithmes distribués qui simulent les systèmes synchrones dans un environnement asynchrone. Nos travaux ont abouti aux résultats publiés dans [50, 51]. Ce chapitre est organisé comme suit :
Nous présentons le problème et les propriétés du synchroniseur que nous voulons dans la section 4.2. Nous codons l'algorithme du synchroniseur simple présenté dans [72] et une version allégée dans la section 4.3. Dans la section 4.4, nous proposons une solution de ce problème utilisant l'algorithme de SSP [71]. Nous avons besoin de la connaissance du diamètre ou d'une borne du diamètre du réseau pour que l'algorithme fonctionne. Dans la section 4.5, nous proposons un algorithme "classique" qui a besoin d'un noeud distingué dans le réseau. Dans la section 4.6, la seule connaissance dont nous avons besoin est la topologie du réseau qui doit être un arbre. Dans la section 4.7, nous proposons une équivalence entre le synchroniseur et la détection de la terminaison explicite. Enfin, dans la section 4.8, nous proposons un constructeur des synchroniseurs.

Partie 2 : Visidia

Cette partie est consacrée à Visidia et à une méthode d'implémentation des calculs locaux et des systèmes de réécriture de graphe. Les résultats de cette partie ont été obtenus en collaboration avec M. Mosbah dans [61, 8, 63, 62]. Un seul chapitre constitue cette partie, c'est le chapitre 5. Ce chapitre est composé comme suit :
Dans la section 5.1, nous donnons l'architecture générale de Visidia. Dans les sections 5.2 et 5.3, nous détaillons la conception et la réalisation de la plate-forme Visidia. Nous présentons les outils existants dans la section 5.4. Enfin dans la section 5.5, nous présentons et utilisons les procédures probabilistes [56, 54, 55] qui permettent de faciliter l'implémentation des calculs locaux et des systèmes de réécriture de graphe.

Partie 3 : Exclusion mutuelle

Cette partie est consacrée aux problèmes d'exclusion mutuelle, à leurs implémentations et à leurs expérimentations sur Visidia. Les résultats de cette partie ont été obtenus en collaboration avec M. Mosbah et A. Zemmari dans [64]. Le chapitre 6 est composé comme suit :
La section 6.1 est consacrée au problème de conflits entre processeurs. Ce problème est la généralisation du problème de dîner des philosophes introduit par Dijkstra, où plusieurs philosophes partagent plusieurs ressources. Dans la section 6.4, nous présentons le problème d'exclusion mutuelle dans un arbre. Contrairement à la section précédente, une seule ressource est partagée par les processeurs qui forment un arbre. Enfin, dans la section 6.6, nous utilisons la marche aléatoire et la marche aléatoire biaisée pour résoudre le problème de partage d'une seule ressource dans un réseau d'une topologie arbitraire.

Chapitre 1

Préliminaires

Dans notre étude, les réseaux de communications sont représentés par des graphes. Un petit rappel des notions de la théorie des graphes est nécessaire. Dans ce chapitre, nous commençons par rappeler certaines définitions de graphes.

1.1 Définitions élémentaire pour les graphes

Les définitions utilisées sont celles données dans [11].

Définition 1.1.1. Un graphe *simple non orienté* G est un couple (V, E), noté aussi $(V(G), E(G))$, où V est un ensemble de sommets et E est un ensemble de paires de sommets distincts. Les éléments de E sont appelés *arêtes*.

Définition 1.1.2. Soit $e = \{u, v\} \in E$ une arête telle que u et v sont les deux sommets d'extrémités. Les sommets u et v sont dit *voisins* ou *adjacents*, et e est dite *arête incidente* à u et à v.

Définition 1.1.3. Le graphe G est dit *fini* si les ensembles de ses sommets et de ses arêtes sont finis. Le nombre de sommets du graphe G est appelé *taille* de G.

Définition 1.1.4. Soit $G = (V, E)$ et $H = (V', E')$ deux graphes. On dit que H est un *sous-graphe* de G si l'ensemble des sommets de V' est un sous-ensemble des sommets de V et que l'ensemble des arêtes de E' est un sous ensemble de E. i.e. $V(H) \subseteq V(G)$ et $E(H) \subseteq E(G)$.

Définition 1.1.5. Un *sous-graphe couvrant* du graphe G est un sous-graphe de G qui contient tous les sommets de G. i.e. si $H = (V', E')$ est un sous-graphe couvrant de $G = (V, E)$ alors $V' = V$.

Définition 1.1.6. Un *chemin* dans un graphe G est une suite de sommets voisins, i.e. une suite $P = (u_0, u_1, \ldots, u_k)$ telle que pour tout $0 \leqslant j < k$, l'arête $\{u_j, u_{j+1}\} \in E(G)$.
La *longueur* du chemin est égale au nombre d'arêtes parcourues. La longueur du chemin

P est égale à k.
Un chemin est dit *simple* si les sommets de P sont distincts.
Un *cycle* est un chemin dont les extrémités sont confondues. i.e. $u_0 = u_k$.
Un *anneau* est un graphe constitué d'un cycle simple.

Définition 1.1.7. Deux sommets u et v sont dit *connectés* s'il existe un chemin de u à v.
Un graphe est dit *connexe* si pour tous sommets u et v de G, il existe un chemin entre u et v.

Définition 1.1.8. Un *arbre* est un graphe connexe sans cycle.
Un *arbre recouvrant* d'un graphe G est un sous-graphe couvrant de G qui est un arbre.
Une *forêt* est un graphe dont les composantes connexes sont des arbres.
Une *forêt couvrante* d'un graphe G est une forêt qui contient l'ensemble des sommets de G.

Définition 1.1.9. Un graphe est dit *complet* si pour toute paire de sommets (u, v), l'arête $\{u, v\}$ existe. On note K_n le graphe complet de taille n.
Le nombre d'arêtes dans le graphe complet K_n est $\frac{n(n-1)}{2}$.

Définition 1.1.10. Soit $G = (V, E)$ un graphe connexe. La *distance* entre deux sommets u et v de G, notée $d(u, v)$, est la longueur du plus court chemin de u à v.
Le *diamètre* de G est la distance maximale entre deux de ses sommets :

$$D(G) = \max_{u,v \in V}(d(u, v))$$

Définition 1.1.11. Le voisinage d'un sommet u dans un graphe $G = (V, E)$, noté $N_G(u)$, est l'ensemble des sommets adjacents à u : $N_G(u) = \{v \in V | \{u, v\} \in E\}$.
Le degré de u dans G, noté $deg_G(u)$, est le nombre de voisins de u dans G. i.e. $deg_G(u) = |N_G(u)|$.
Un graphe est dit *régulier* ou *d-régulier*, si tous les sommets du graphe ont le même degré d.

Définition 1.1.12. Soit $G = (V, E)$ un graphe connexe et soit u un sommet de G. On appellera *boule* de centre u, notée $B_G(u)$, le graphe constitué des sommets u et de $N_G(u)$ et des arêtes incidentes à u.
Soit $k \in \mathbb{N}$, la boule de centre u et de rayon k, notée $B_G(u, k)$ ou $B_k(u)$, est le sous-graphe constitué des chemins issus de u de longueur inférieure ou égale à k i.e. :
$V(B_G(u, k)) = \{v \in V(G) \,|\, d(u, v) \leqslant k\}$ et
$E(B_G(u, k)) = \{\{v, w\} \in E(G) \,|\, d(u, v) \leqslant k - 1 \text{ et } d(u, w) \leqslant k\}$.

Définition 1.1.13. Soit $G = (V_G, E_G)$ et $H = (V_H, E_H)$ deux graphes. Un *homomorphisme* entre G et H est une application φ de $V(G)$ dans $V(H)$ telle que si $\{u, v\} \in E(G)$ alors $\{\varphi(u), \varphi(v)\} \in E(H)$.
Si φ est une application bijective et si φ^{-1} est un homomorphisme de H vers G, alors φ définit un *isomorphisme* entre les deux graphes G et H. On dit alors que G et H sont *isomorphes*.

1.2 Revêtements

Définition 1.2.1. Soit γ un homomorphisme surjectif de G sur H. γ est un *revêtement* si pour tout sommet $u \in V(G)$, γ induit un isomorphisme de $B_G(u)$ sur $B_H(\gamma(u))$. On dit dans ce cas que G est un revêtement de H.
Un revêtement de G sur H est dit *propre* si G et H ne sont pas isomorphes.

Remarque 1.2.1. Un revêtement est exactement un revêtement au sens classique de la topologie, voir [48]. A noter que pour parler rigoureusement, le revêtement est le morphisme. Par extension et abus de langage, on parlera d'un graphe comme étant un revêtement d'un graphe donné.

Définition 1.2.2. Un graphe est *minimal* s'il n'est revêtement propre d'aucun autre graphe. La famille des graphes minimaux sera notée $\mathcal{G}_{min}$.

Lemme 1.2.1. *Soit γ un revêtement de G sur H. Soit u un sommet de H. Alors $\gamma^{-1}(B_H(u))$ est une collection de boules disjointes isomorphes à $B_H(u)$.*

Étant donnée une boule B de H, un certain nombre de "copies conformes" de celle-ci sont présentées dans le graphe G, on appellera ces copies des relèvements de B, voir FIG. 1. Un argument de connexité permet de prouver que chaque boule de H est copiée avec la même multiplicité ([32]).

Corollaire 1.2.1. *Soit H un graphe connexe et G un revêtement fini de H via γ. Alors il existe un entier q tel que, pour tout sommet u de $V(H)$, on a $|\gamma^{-1}(u)| = q$.*

Définition 1.2.3. On dit alors que γ est un revêtement à q feuillets.

Proposition 1.2.1. *Soit n et m deux entiers. Alors l'anneau A_n est un revêtement de l'anneau A_m si et seulement si $m|n$. Dans ce cas, $\frac{n}{m}$ est le nombre de feuillets.*

Proposition 1.2.2. *Soit G un graphe connexe, si $|V(G)|$ et $|E(G)|$ sont premiers entre eux, alors G est minimal.*

1.3 Quasi-revêtements

Les quasi-revêtements ont été introduits dans [53] pour étudier la détection de la terminaison des systèmes de réécriture. Ce sont des applications qui se comportent partiellement comme des revêtements.

Définition 1.3.1. Soit K et H deux graphes et soit δ une fonction (partielle) de $V(K)$ dans $V(H)$. δ est un *quasi-revêtement* de rayon r s'il existe un graphe G (éventuellement infini) qui soit revêtement de H via γ (voir FIG. 2) et s'il existe $z_K \in V(K)$, $z_G \in V(G)$ tel que :
- il existe un isomorphisme $\varphi : B_K(z_K, r) \longrightarrow B_G(z_G, r)$,
- le domaine de definition de δ contient $B_K(z_K, r)$,
- $\delta_{|V(B_K(z_K,r))} = \gamma \; o \; \varphi_{|V(B_K(z_K,r))}$.

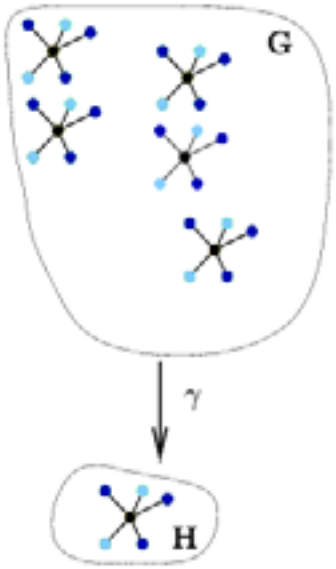


FIG. 1 – Les antécédents d'une boule de H forment une collection de boules disjointes de G

L'entier $s = |B_K(z_K, r)|$ sera appelé l'étendue du quasi-revêtement et le graphe G le revêtement associé.

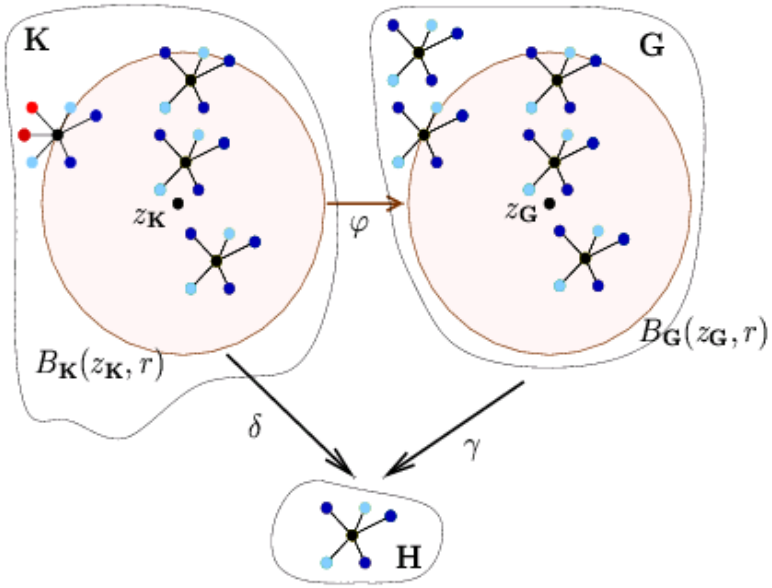


FIG. 2 – δ ($V(K)$ dans $V(H)$) est un quasi-revêtement de rayon r et revêtement associé γ (G vers H)

On effectuera le même abus de langage que pour les revêtements et l'on pourra dire que K est un quasi-revêtement de H de rayon r. On notera qu'en particulier, tout revêtement de H est également un quasi-revêtement pour n'importe quel rayon $r \in \mathbb{N}$. Et que tout quasi-revêtement de rayon r est aussi quasi-revêtement de rayon $r' \leqslant r$.

Définition 1.3.2. On définit le nombre de feuillets q d'un quasi-revêtement comme étant la cardinalité minimale des ensembles d'antécédents des sommets de H :

$$q = \min_{w \in H}\{card(\{v \in B_K(z_K, r) | \delta(v) = w\})\}$$

Définition 1.3.3. Un quasi-revêtement est dit *strict* si $B_K(z_K, r) \subsetneq K$.

Un quasi-revêtement non-strict est un revêtement, et dans ce cas, les deux définitions du nombre de feuillets coïncident.

Chapitre 2

Calculs locaux et systèmes de réécriture de graphe

2.1 Introduction

2.1.1 Le modèle

Les systèmes de réécriture de graphe, et plus généralement, les calculs locaux dans les graphes sont des modèles puissants qui fournissent des outils généraux pour coder les algorithmes distribués, pour prouver leur validité et pour comprendre leur puissance [46]. Pour mieux comprendre les algorithmes distribués, nous nous sommes intéressés à plusieurs ouvrages [3, 6, 41, 47, 69, 72].

Nous considérons un réseau de processeurs avec une topologie arbitraire. Ce réseau est représenté par un graphe connexe et non orienté où les noeuds représentent les processeurs et les arêtes représentent les canaux de communications. Un algorithme distribué est alors codé par des réécritures locales. Des étiquettes attachées aux noeuds et aux arêtes sont *localement* modifiées, c'est à dire, les noeuds et les arêtes appartiennent à un sous-graphe de rayon fixe 1, selon certaines règles dépendant uniquement de ce même sous-graphe (*calcul local*). La réécriture est exécutée jusqu'à ce qu'aucune transformation ne soit possible. La configuration résultante est dite de forme normale. Deux étapes de réécriture séquentielle sont dite *indépendantes* si elles sont exécutées dans des sous-graphes disjoints. Dans ce cas, ces deux étapes peuvent être appliquées dans n'importe quel ordre et même simultanément.

2.1.2 Les modèles liés

D'autres approches similaires à ce modèle ont été étudié par Rosenstiehl et al. [70], Angluin [2], Yamashita et Kameda [75], et Boldi et Vigna [13]. Dans [70], le modèle considéré est un modèle synchrone, où les noeuds représentent un automate déterministe fini. L'étape

de base du calcul consiste à calculer le nouvel état de chaque processeur selon son état et les états de ses voisins. Dans [2], le modèle considéré est le modèle asynchrone. Une étape de base de calcul signifie que deux noeuds adjacents échangent leurs états (étiquettes) et puis chacun calcule son nouvel état. Dans [75] un modèle asynchrone est étudié où une étape de base du calcul signifie qu'un processeur change son état, ou envoie un message, ou reçoit un message. Dans [13], le réseau est un graphe orienté et ses arcs peuvent être colorés ; chaque processeur change son état selon son état précèdent et les états de ses voisins qui lui sont reliés par des arcs entrants. L'activation des processeurs peut être synchrone, asynchrone ou intercalée, c'est à dire, soit tous les processeurs s'activent au même moment (synchrone), chaque processeur s'active indépendemment des autres processeurs (asynchrone) ou quelques processeurs actifs activent les autres.

2.1.3 D'autres modèles

Une autre approche commune et générale de calcul simultané est basée sur les *processus* et les *actions*. De cette façon, un processus P_0 peut exécuter quelques actions $A = \{a_1, \ldots, a_n\}$ ou créer quelques nouveaux processus (fils) $P_1, \ldots, P_m$ dans chaque phase de son cycle de vie. De tels systèmes sont habituellement décrits en termes d'algèbre de processus ou de logique de processus [43], mais des grammaires de graphe peuvent aussi être utilisées à cette fin [38]. Cette notion de calcul distribué est très abstraite par rapport au matériel existant et permet une approche plus dynamique des systèmes distribués. Dans notre approche, le nombre de processus (processeurs) dans chaque modèle est fixé, mais nous avons une notion topologique du réseau fournissant la base matérielle du calcul distribué. En effet, en identifiant les processus par des processeurs, notre notion de calcul distribué apparaît comme un cas particulier utile de la notion d' "acteur" de [38].

L'identification des noeuds d'un graphe avec les processeurs et des arêtes avec les canaux de communications n'est pas la seule possibilité pour décrire le calcul distribué en terme de système de transformation ou de réécriture de graphe. Dans l'approche de [58], les canaux de communications sont représentés par des noeuds et les processeurs sont représentés par des (hyper-) arêtes. Cette proposition consiste à définir une sémantique formelle pour un langage de programmation simultané, tandis que notre but consiste à fournir une plateforme expérimentale pour observer le comportement de l'exécution des systèmes distribués.

2.2 Systèmes de réécriture de graphe

2.2.1 Graphes étiquetés

Dans tout ce qui suit, nous considérons seulement les graphes connexes simples où des sommets et des arêtes sont étiquetés avec des étiquettes appartenant à un alphabet L. Cet alphabet peut être fini ou infini.

Définition 2.2.1. Soit $\mathcal{L}$ un ensemble dont les éléments sont appelés *états*.

Un *graphe $\mathcal{L}$-étiqueté* est un couple (G, λ) où G est un graphe et λ est une fonction de $V(G) \cup E(G)$ vers $\mathcal{L}$.
Le graphe G est appelé le graphe sous-jacent , et λ est une fonction d'étiquetage de G.

La classe des graphes étiquetés sur un alphabet fini L sera notée par $\mathcal{G}_L$.

Définition 2.2.2. Soit (G, λ) et (G', λ') deux graphes étiquetés. (G, λ) est un sous-graphe de (G', λ') , noté par $(G, \lambda) \subseteq (G', \lambda')$, si G est un sous-graphe de G' et λ est la restriction de λ' à $V(G) \cup E(G)$.

Définition 2.2.3. L'application φ: $V(G) \to V(G')$ est un homomorphisme de (G, λ) vers (G', λ'), si φ est un homomorphisme de graphe de G vers G' préservant les étiquettes, i.e., pour tous sommets $v, w \in V(G)$, $\lambda'(\varphi(v)) = \lambda(v)$, et $\lambda'(\{\varphi(v), \varphi(w)\}) = \lambda(\{v, w\})$.
φ est un isomorphisme s'il est bijectif.

Définition 2.2.4. Une *occurrence* de (G, λ) dans (G', λ') est un isomorphisme φ entre (G, λ) et un sous-graphe (H, η) de (G', λ').

2.2.2 Systèmes de réécriture de graphe

Nous décrivons dans cette section les notions formelles des systèmes de réécriture de graphe.

Définition 2.2.5. Une *règle de réécriture* est un triplet $R = (G_R, \lambda_R, \lambda'_R)$ tel que (G_R, λ_R) et (G_R, λ'_R) sont deux graphes étiquetés.

Définition 2.2.6. Un *système de réécriture de graphe* (SRG) est un triplet $\mathcal{R} = (L, I, P)$ où L est un ensemble d'états, I est un sous ensemble de $\mathcal{L}$ appelé l'ensemble des *états initiaux* et P un ensemble fini de règles de réécriture.

Définition 2.2.7. Une *$\mathcal{R}$-étape de réécriture* est un quintuplet $(G, \lambda, R, \phi, \lambda')$ tel que R est une règle de réécriture de P et ϕ est à la fois une occurrence de (G_R, λ_R) dans (G, λ) et une occurrence de (G_R, λ'_R) dans (G, λ').
Une étape de réécriture est notée $(G, \lambda) \to_{R,\phi} (G, \lambda')$.

Définition 2.2.8. Une *séquence de $\mathcal{R}-$réécriture* est un uplet $(G,\lambda_0,R_0,\varphi_0,\lambda_1,R_1,\varphi_1,\lambda_2,\ldots,$ λ_{n-1}, R_{n-1}, φ_{n-1},$\lambda_n)$ tel que pour tout i, $0 \leq i < n$, $(G, \lambda_i, R_i, \varphi_i, \lambda_{i+1})$ est une étape de $\mathcal{R}-$réécriture.
L'existence de telles séquences de réécriture sera notée par $(G, \lambda_0) \xrightarrow[\mathcal{R}]{*} (G, \lambda_n)$.

Le calcul s'arrête quand l'étiquetage du graphe est tel qu'aucune règle de réécriture ne peut être appliquée :

Définition 2.2.9. Un graphe étiqueté (G, λ) est dit *$\mathcal{R}-$irréductible* s'il n'existe aucune occurrence de (G_R, λ_R) dans (G, λ) pour toute règle de réécriture R de P.

Pour tout graphe étiqueté (G, λ) dans $\mathcal{G}_I$ nous notons par $Irred_{\mathcal{R}}((G, \lambda))$ l'ensemble de tous les graphes étiquetés $\mathcal{R}-$irréductible (G, λ) tels que $(G, \lambda) \xrightarrow[\mathcal{R}]{*} (G, \lambda')$. Intuitivement parlant, l'ensemble des $Irred_{\mathcal{R}}((G, \lambda))$ contient tous les étiquetages finaux qui sont obtenus des graphes $I-$étiquetés (G, λ) par application des règles de réécriture de P et peut être vu comme un ensemble de tous les résultats possibles des calculs codés par le système $\mathcal{R}$.

Exemple 2.2.1. Illustrons les systèmes de réécriture de graphe en considérant un algorithme distribué simple qui calcule un arbre recouvrant d'un réseau. Nous supposons l'existence d'un processeur distingué unique dans le réseau qui est dans un état "actif" (codé par l'étiquette A), tous les autres processeurs sont dans un un état "neutre" (étiquette N) et tous les canaux de communications sont dans un état "passif" (étiquette 0). Au début, l'arbre ne contient que le sommet actif. A n'importe quelle étape du calcul, un sommet actif peut activer un de ses voisins neutres et marque le canal de communication correspondant par l'étiquette 1. Le calcul s'arrête dès que tous les processeurs sont activés. L'arbre recouvrant est alors obtenu en considérant tous les canaux de communications avec l'étiquette 1.

Le système est donné par $\mathcal{R}_1 = (L, I, P)$ défini par $L = \{N, A, 0, 1\}$, $I = \{N, A, 0\}$ et $P = \{R\}$ où R est la règle de réécriture suivante :

A chaque fois qu'un noeud étiqueté A est lié par une arête étiquetée 0 à un noeud dont l'état est N, alors la règle R nous permet de réécrire le sous-graphe correspondant.

Un exemple de calcul utilisant cette règle est donné dans FIG 3. Les étapes de réécriture *peuvent* s'exécuter simultanément dans des parties disjointes du graphe. Quand le graphe devient irréductible, i.e. aucune règle ne peut s'appliquer, un arbre recouvrant, constitué des arêtes étiquetées 1, est alors calculé.

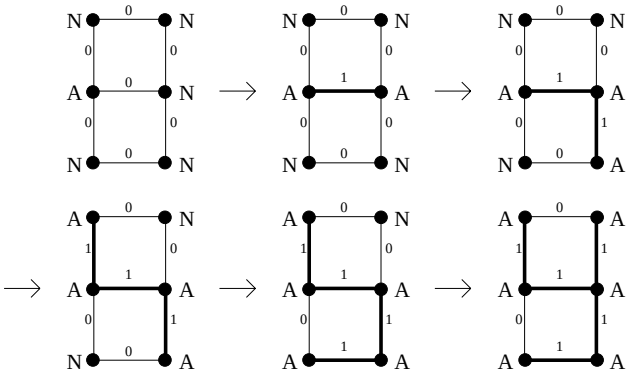


FIG. 3 – Calcul distribué d'un arbre recouvrant

2.2.3 Systèmes de réécriture de graphe avec contextes interdits

L'idée que nous développons ici est d'empêcher l'application d'une règle de réécriture toutes les fois où les occurrences correspondantes sont "inclues" dans certaines configurations spéciales, appelées contextes. Plus formellement, nous avons :

Définition 2.2.10. Soit (G, λ) un graphe étiqueté. Un *contexte* de (G, λ) est un triplet (H, μ, ψ) tel que (H, μ) est un graphe étiqueté et ψ une occurrence de (G, λ) dans (H, μ).

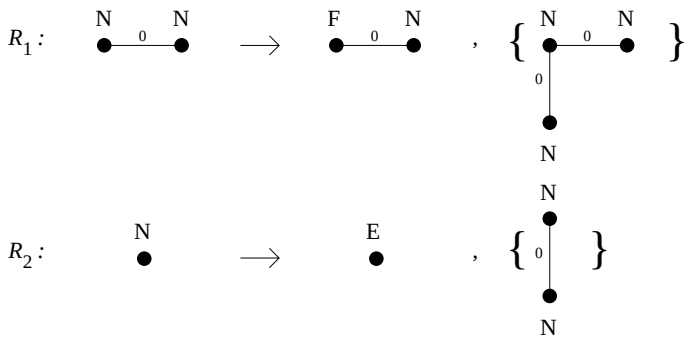
Définition 2.2.11. Une *règle de réécriture avec contextes interdits* est un quadruplet $R = (G_R, \lambda_R, \lambda'_R, F_R)$ tel que $(G_R, \lambda_R, \lambda'_R)$ est une règle de réécriture et F_R est un ensemble fini de contextes de (G_R, λ_R). Une règle de réécriture peut être appliquée dans un sous-graphe si et seulement si ce sous-graphe n'est pas inclus dans une occurrence de l'un de ses contextes interdits [45, 46].

$$\mathcal{R} : Left \longrightarrow Right \ ; \ \{F_C\}$$

Définition 2.2.12. Un *système de réécriture de graphe avec contextes interdits* ($SRGCI$) est un triplet $\mathcal{R} = (\mathcal{L}, \mathcal{I}, \mathcal{P})$ défini comme un SRG sauf que $\mathcal{P}$ est un ensemble de règles de réécriture avec contextes interdits.

Définition 2.2.13. Une étape de $\mathcal{R}-$*réécriture* est un 5-uplet $(G, \lambda, R, \varphi, \lambda')$ tel que R est une règle de réécriture avec contextes interdits de P, φ est une occurrence de (G_R, λ_R) dans (G, λ) et une occurrence de (G_R, λ'_R) dans (G, λ'), et pour tout contexte (H_i, μ_i, ψ_i) de (G_R, λ_R), il n'y a aucune occurrence φ_i de (H_i, μ_i) dans (G, λ) tel que $\varphi_i(\psi_i(G_R, \lambda_R)) = \varphi(G_R, \lambda_R)$.

Exemple 2.2.2. Dans cet exemple, nous donnons un système de réécriture de graphe qui code un algorithme d'élection dans un arbre. Soit $\mathcal{R}_2 = (L, I, P)$ le SRGCI défini par $L = \{N, F, E, 0\}$, $I = \{N, 0\}$ et $P = \{R_1, R_2\}$ où R_1, R_2 sont les règles de réécriture avec contextes interdits suivantes :



Appelons une *feuille* tous les noeuds étiquetés N ayant exactement un voisin avec l'étiquette N. La règle R_1 consiste à "enlever" une feuille de l'arbre (puisque le contexte interdit assure que ce noeud n'a pas d'autre voisin étiqueté N) en lui donnant l'étiquette F. Ainsi, si (G, λ) est un arbre étiqueté où tous les noeuds sont étiquetés N et toutes les arêtes sont étiquetées 0 alors la procédure d'élimination mène à un unique sommet étiqueté N qui deviendra élu

grâce à la règle R_2. Il n'est pas difficile de voir que tous les noeuds de l'arbre peuvent être "élus" par cet algorithme.

Exemple 2.2.3. Nous donnons dans cet exemple une version distribuée du calcul d'un arbre recouvrant avec la détection locale de la terminaison globale i.e. au moins un noeud du graphe sait que le calcul est fini. Cet algorithme est donné dans [72, Chap. 6, p. 215].

Comme pour l'exemple 2.2.1, nous supposons qu'initialement un seul noeud soit étiqueté A, tous les autres sommets sont étiquetés N et toutes les arêtes sont étiquetées 0. L'idée générale est que le sommet initialement étiqueté A gardera son état jusqu'à la fin du calcul, par contre les autres noeuds seront activés et aurons l'étiquette A'. Dès qu'un noeud étiqueté A' n'est plus "utile" pour le calcul, il atteindra l'état final (étiquette F).

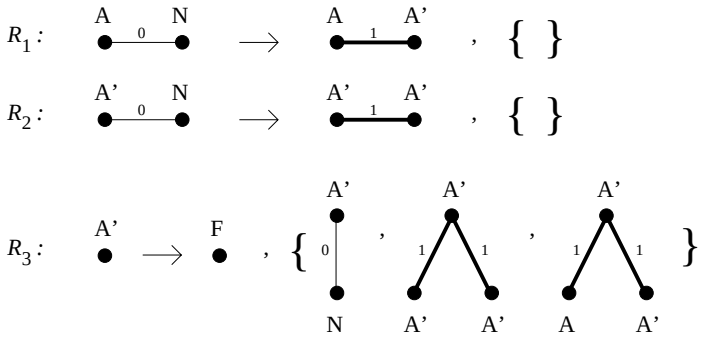
A chaque étape de calcul, un noeud u actif (étiqueté A ou A') agira comme suit :

1. Si u a un noeud voisin v étiqueté N, alors u va activer ce noeud : u gardera son étiquette, v devient actif (étiquette A') et l'arête $\{u, v\}$ prend l'étiquette 1.
2. Si u est étiqueté A', u n'a pas de noeud voisin étiqueté N et il est tel que tous ses noeuds voisins excepté un, dont l'arête qui les lie est étiquetée 1, sont étiquetés F, alors u devient étiqueté F.

A tout moment, le sous-graphe induit par les arêtes étiquetées 1 et les noeuds étiquetés A et A' est un arbre. Intuitivement, la deuxième étape signifie que le noeud u est une feuille de cet arbre.

Ainsi, cet algorithme s'exécute en deux phases (qui peuvent se chevaucher) : dans la première, l'arbre grandit jusqu'à ce que tous les sommets soient atteints ; dans la deuxième phase, l'arbre va diminuer (en perdant ses feuilles) jusqu'à ce qu'il ne reste que le sommet initialement étiqueté A. Ce sommet est capable de détecter que l'algorithme a fini quand tous ses voisins seront étiquetés F.

L'algorithme peut être codé par le système de réécriture de graphe avec contextes interdits $\mathcal{R}_3 = (L, I, P)$ défini par $L = \{N, A, A', F, 0, 1\}$, $I = \{N, A, 0\}$, $P = \{R_1, R_2, R_3\}$ où R_1, R_2 et R_3 sont les règles de réécriture avec contextes interdits suivantes :



Les règles R_1 et R_2 n'ont pas de contextes interdits : il n'y a pas de restriction pour les appliquer. La règle R_3 a 3 contextes interdits.

La Figure FIG. 4 décrit un exemple de calcul utilisant cet algorithme.

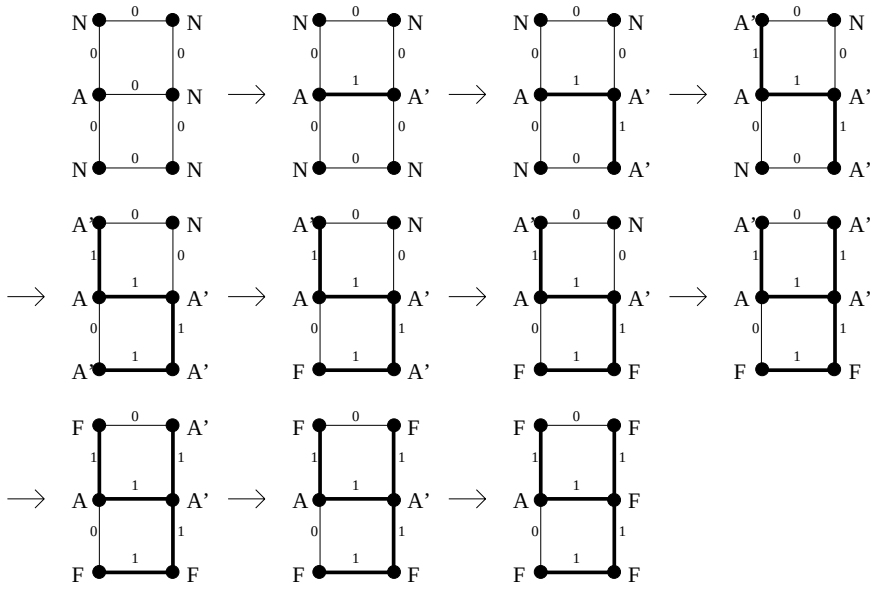


Fig. 4 – Calcul distribué d'un arbre recouvrant avec détection locale de la terminaison globale

2.2.4 Systèmes de réécriture de graphe avec priorités

L'idée de ce système est d'attribuer une priorité à chaque règle de telle sorte que si deux règles de réécriture peuvent s'appliquer en même temps, la règle la plus prioritaire s'applique. Plus formellement, nous avons :

Définition 2.2.14. Un *système de réécriture de graphe avec priorité* ($SRGP$) est un 4-uplet $\mathcal{R} = (\mathcal{L}, \mathcal{I}, \mathcal{P}, >)$ tel que $(\mathcal{L}, \mathcal{I}, \mathcal{P})$ est un SRG et $>$ est un ordre partiel défini sur l'ensemble $\mathcal{P}$ appelé la *relation de priorité*.

Définition 2.2.15. Une étape de $\mathcal{R}-$*réécriture* est un 5-uplet $(G, \lambda, R, \varphi, \lambda')$ tel que R est une règle de réécriture de P et φ est une occurrence de (G_R, λ_R) dans (G, λ) et aussi une occurrence de (G_R, λ'_R) dans (G, λ'), et il n'existe aucune occurrence φ' d'une règle de réécriture R' de $\mathcal{P}$ avec $R' > R$ tel que $\varphi(G_R)$ et $\varphi(G_{R'})$ s'intersectent dans G.

Proposition 2.2.1. *Les systèmes de réécriture de graphe avec priorités et les systèmes de réécriture de graphe avec contextes interdits sont équivalents. Tout algorithme codé avec l'un de ces deux systèmes peut être codé avec l'autre.*

La preuve de cette proposition est donnée dans [45].

Exemple 2.2.4. Nous considérons le problème du calcul d'un arbre recouvrant dans un graphe. Cet algorithme est décrit dans [72, Chap. 6, p. 213] mais il est codé dans cet exemple avec les systèmes de réécriture de graphe avec priorité.

Supposons, comme pour l'exemple 2.2.1, que tous les sommets soient initialisés à l'état "neutre" (étiqueté N) excepté un seul sommet qui est dans un état "actif" (étiqueté A) et

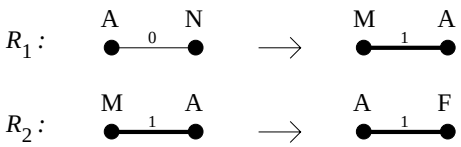
toutes les arêtes sont étiquetées 0.

Dans chaque étape de calcul, le noeud u étiqueté A agira comme suit :

1. Si u a un voisin v étiqueté N, alors u active ce voisin : u devient "marqué" (étiquette M), v devient "actif" (A) et l'arête entre u et v ($\{u, v\}$) change son étiquette à 1.
2. Si u n'a pas de voisin étiqueté N et possède un (unique) voisin w "marqué" par une arête étiqueté 1 alors u va réactiver ce voisin : u entre dans un état final (étiqueté F) et w devient "actif" (A).

Le calcul s'arrête dès qu'aucune règle de réécriture de ce système ne peut être appliquée (dans ce cas, tous les voisins du noeud "actif" sont dans l'état "final"). L'arbre recouvrant est donné par l'ensemble des arêtes étiquetées 1.

L'algorithme est codé par le système de réécriture de graphe avec priorité suivant $\mathcal{R}_4 = (L, I, P, >)$ défini par $L = \{N, A, M, F, 0, 1\}$, $I = \{N, A, 0\}$, $P = \{R_1, R_2\}$ où R_1 et R_2 sont les règles de réécriture de graphe suivantes :



avec la relation de priorité : $R_1 > R_2$.

Nous pouvons remarquer que cet algorithme fonctionne de manière séquentielle puisqu'à tout moment de l'exécution de cet algorithme, un seul noeud est à l'état "actif". Donc à chaque étape, une seule règle peut être appliquée à la fois.

La Figure 5 décrit un exemple de calcul utilisant cet algorithme (les arêtes "marquées", étiquetées 1, sont dessinées en gras).

2.3 les méta-règles de réécriture de graphe

Dans la section précédente, les règles de réécriture de graphe sont considérées comme des règles simples puisque les états des noeuds sont des états simples, ils ont une simple composante appartenant à l'alphabet donné.

Définition 2.3.1. Nous appelons "méta-règles" une règle qui englobe plusieurs règles de réécriture. Ce qui implique l'utilisation d'ensembles finis ou infinis dans les composantes de règles de réécriture.

Exemple 2.3.1. Dans cet exemple, nous considérons le problème des 3-coloration dans un anneau [72, 47]. Le problème consiste à donner pour chaque noeud une couleur d'un ensemble de trois couleurs telle que deux noeuds voisins aient des couleurs différentes. Nous fournissons un système de réécriture de graphe pour colorier un anneau avec 3 couleurs,

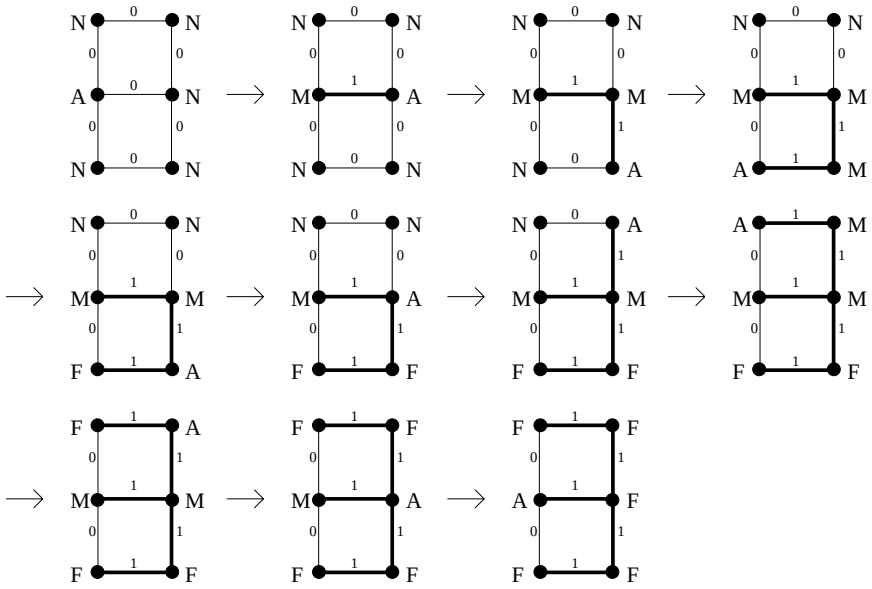


FIG. 5 – Calcul séquentiel d'un arbre recouvrant

initialement, chaque sommet a une couleur arbitraire. Soit $\{c_1, c_2, c_3\}$ l'ensemble des couleurs. Soit $\mathcal{R}_5 = (L, I, P)$ le système de réécriture de graphe défini par $L = \{c_1, c_2, c_3\}$, $I = \{c_1, c_2, c_3\}$ et $P = \{R_1, R_2\}$ où R_1 et R_2 sont les méta-règles de réécriture suivantes :

R_1 :  $; y \neq x$

R_2 :  $; \begin{array}{l} x \neq y \\ z \neq x \\ z \neq y \end{array}$

$x, y, z \in L$

Initialement, chaque sommet possède un état aléatoire.
Ce système de réécriture de graphe attribue une 3-coloration correcte aux noeuds de l'anneau. Notez que cet algorithme est un exemple d'algorithme auto-stabilisant [27].

Maintenant, nous ajoutons à chaque règle de réécriture un ensemble de conditions. Ce qui facilite l'utilisation des méta-règles. Dans ce qui suit, nous donnons une définition formelle.

Définition 2.3.2. Une *méta-règle de réécriture avec conditions* est un quadruplet $R = (G_R, \lambda_R, \lambda'_R, C_R)$ tel que $(G_R, \lambda_R, \lambda'_R)$ est une méta-règle de réécriture et C_R est un ensemble fini de conditions logique de λ_R. Une règle de réécriture peut être appliquée dans un sous-graphe si et seulement si les conditions de la règle sont satisfaites.

Définition 2.3.3. Une *méta-règle de réécriture avec contextes interdits et conditions* est un quintuplet $R = (G_R, \lambda_R, \lambda'_R, C_R, F_R)$ tel que $(G_R, \lambda_R, \lambda'_R, C_R)$ est une méta-règle de réécriture avec conditions et F_R est un ensemble fini de contextes de (G_R, λ_R).

De la même façon, les définitions précédentes des systèmes de réécriture de graphe, des systèmes de réécriture de graphe avec contexte interdits et des systèmes de réécriture de graphe avec priorités restent les mêmes avec utilisation des méta-règles de réécriture.

Remarque 2.3.1. Dans la suite, les méta-règles de réécriture seront appelées des règles de réécriture de graphe.

2.4 Les techniques pour prouver les calculs distribués

Les techniques dans cette section sont similaires à celles de [46].

Définition 2.4.1. Un système de réécriture de graphe $\mathcal{R}$ est dit *noethérien* s'il n'existe pas de séquence infinie de $\mathcal{R}$-réécriture de graphe tel qu'initialement, l'étiquetage du graphe satisfait l'ensemble des états initiaux $\mathcal{I}$.

Ainsi, si un algorithme distribué est codé par un système de réécriture de graphe noethérien alors cet algorithme se termine toujours.
Afin de montrer qu'un système donné est noethérien, nous utilisons généralement la technique suivante. Soit $(S, <)$ un ensemble muni d'un ordre partiel sans chaîne décroissante infinie (c'est à dire, toutes les chaînes décroissantes $x_1 > x_2 > \cdots > x_n > \ldots$ de S sont finies).
$<$ est un *ordre noethérien*. Il est compatible avec $\mathcal{R}$ s'il existe une application f de $\mathcal{G}_L$ vers S tel que pour toute étape de $\mathcal{R}$-réécriture $(G, \lambda) \to (G, \lambda')$ nous avons $f(G, \lambda) > f(G, \lambda')$.
Il n'est pas difficile de voir que si un tel ordre existe alors le système $\mathcal{R}$ est noethérien : puisqu'il n'y a aucune chaîne infinie décroissante dans S, il n'existe pas de séquence infinie de $\mathcal{R}$-réécriture de graphe.
Dans nos exemples, l'ensemble S sera un ensemble de $\mathbb{N}^p$ où p est un entier et l'ordre $>_p$ est un ordre lexicographique ; i.e. on note $(x_1, \ldots, x_p) >_p (y_1, \ldots, y_p)$ s'il existe un entier j tel que $x_1 = y_1, \ldots, x_{j-1} = y_{j-1}$, et $x_j > y_j$.

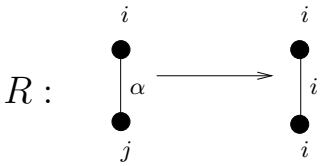
Afin de prouver la *validité* d'un système de réécriture de graphe, et donc, la validité d'un algorithme codé par un tel système, il est utile d'exposer :
(i) quelques propriétés *invariantes* associées au système (par invariant, nous voulons dire quelques propriétés du graphe étiqueté qui sont satisfaites depuis les états initiaux et qui sont préservées après l'application de chaque règle de réécriture de graphe) et
(ii) quelques propriétés des graphes irréductibles.
La validité du système est obtenue grâce à ces propriétés.

2.5 Calcul d'un arbre recouvrant dans un réseau avec identités

Le but de cette section consiste à montrer la puissance de notre modèle, qu'il convient à étudier et à prouver des propriétés des algorithmes distribués sur les réseaux avec identités : i.e. chaque sommet a une identité unique.

2.5.1 Calcul distribué d'un arbre recouvrant

Soit G un graphe de taille n. Chaque sommet a une identité unique : pour simplifier, nous supposons que l'identité est un entier unique de $[1..n]$. Considérons la fonction $\lambda : V \cup E \longrightarrow [0..n]$, qui est initialisée à l'identité des sommets et à 0 pour les arêtes. Afin de calculer un arbre recouvrant, nous considérons l'algorithme codé par le système de réécriture de graphe $\mathcal{R}_6 = (L, I, P)$, défini par $L = [0..n]$, $I = [0..n]$, et $P = \{R\}$ où R est la (méta-)règle de réécriture de graphe suivante :

R : 

; si $j < i$

Nous allons prouver que ce système de réécriture de graphe calcule un arbre recouvrant : ça sera l'arbre induit par les arêtes étiquetées n. Nous démontrons qu'il termine et qu'il est valide.

Terminaison : Soit f une application de $\mathcal{G}_L$ à l'ensemble des entiers naturels $\mathbb{N}$ qui associe à chaque graphe L-étiqueté la somme $\sum_{\substack{v \in V, \\ \lambda(v)=k}} (n-k)$. Il est évident que ce nombre non négatif diminue strictement à chaque application de la règle de réécriture de graphe R, nous obtenons :

Lemme 2.5.1. *Le système de réécriture de graphe $\mathcal{R}_6$ est noethérien.*

Validation : La validation de ce système est obtenue par le lemme suivant :

Lemme 2.5.2. *Soit (G, λ) un graphe simple connexe tel que chaque sommet est étiqueté par son identité (un entier unique de $[1, n]$ où n est la taille du graphe), et chaque arêtes est étiquetée 0. Soit (G, λ') un graphe étiqueté tel que : $(G, \lambda) \xrightarrow[\mathcal{R}_6]{*} (G, \lambda')$. Le graphe (G, λ') satisfait :*

1. *Toutes les arêtes incidentes aux noeuds étiquetés i ont une étiquette α inférieure ou égale à i.*
2. *Tous les sommets étiquetés n sont connectés par des arêtes étiquetées n.*
3. *Le sous-graphe induit par les arêtes étiquetées i n'a pas de cycle ($i > 0$).*

4. *Si (G, λ') est un graphe irréductible obtenu à partir de (G, λ), alors tous les sommets sont étiquetés n.*

Démonstration.
1. Initialement, la propriété est vraie puisque tous les sommets ont leur propre identité comme état et toutes les arêtes sont étiquetées 0. Nous supposons que nous avons appliqué la règle k fois et que la propriété reste vraie. Si nous appliquons la règle pour la $(k+1)^{\text{ième}}$ étape, seul un noeud v étiqueté j et une seule arête e, étiquetée α, vont changer leurs étiquette à i. Puisque par induction, toutes les arêtes incidentes au noeud v ont une étiquette inférieure ou égale à j, et comme $i > j$, alors tous les arêtes, excepté e, ont une étiquette strictement inférieure à i. La nouvelle étiquette de l'arête e est i, qui est égale à l'étiquette des sommets incidents. Ainsi, la propriété est vraie après l'étape de réécriture.
2. La preuve est par induction. Initialement, la propriété est juste car il n'existe qu'un seul sommet étiqueté n. Nous supposons qu'après k applications de la règle la propriété reste vraie. Il existe un sous-graphe H connectant toutes les arêtes et les sommets étiquetés n. Supposons maintenant que nous appliquons la règle pour la $(k+1)^{\text{ième}}$ étape sur un sommet v. Si la nouvelle étiquette de v est n, il est évident que dans ce cas, le noeud v a au moins un voisin étiqueté n qui appartient à H et que l'arête les liant sera étiquetée n. Puisque, par induction, tous les sommets de H qui sont étiquetés n sont connectés par des arêtes étiquetées n, le noeud v, avec le nouvel état n, sera connecté à H par une arête étiquetée n. Ainsi, la propriété est vraie.
3. Au début, toutes les arêtes sont étiquetées 0. Nous supposons que la propriété est vraie après k applications de la règle, nous allons montrer qu'elle est toujours vraie après la $(k+1)^{\text{ième}}$ étape. Quand nous appliquons la règle pour changer l'étiquette d'un noeud v de j à i, nous changeons l'étiquette de v et de l'arête incidente à i. Pour pouvoir appliquer la règle, l'étiquette de v doit être strictement inférieure à i et par la propriété 1 toutes les étiquettes des arêtes adjacentes à v sont inférieures à i donc aucun cycle n'est créé.
4. Nous supposons qu'il existe un noeud avec l'étiquette p, $p < n$ dans (G, λ'). Il existe au moins un sommet avec l'étiquette n. Comme le graphe est connexe, il existe un chemin reliant le sommet étiqueté n au sommet étiqueté p. Ce chemin contient au moins une arête liant un noeud étiqueté n avec un autre noeud étiqueté $k < n$. Dans ce cas, la règle peut être appliquée. Ce qui contredit le fait que (G, λ') est irréductible.

□

Avec ces propriétés, nous avons le résultat suivant :

Théorème 2.5.1. *Le système de réécriture de graphe $\mathcal{R}_6$ calcule un arbre recouvrant.*

Démonstration. La preuve est un résultat de l'application des lemmes précédents, l'arbre formé par les arêtes étiquetées n est un arbre recouvrant du graphe original. □

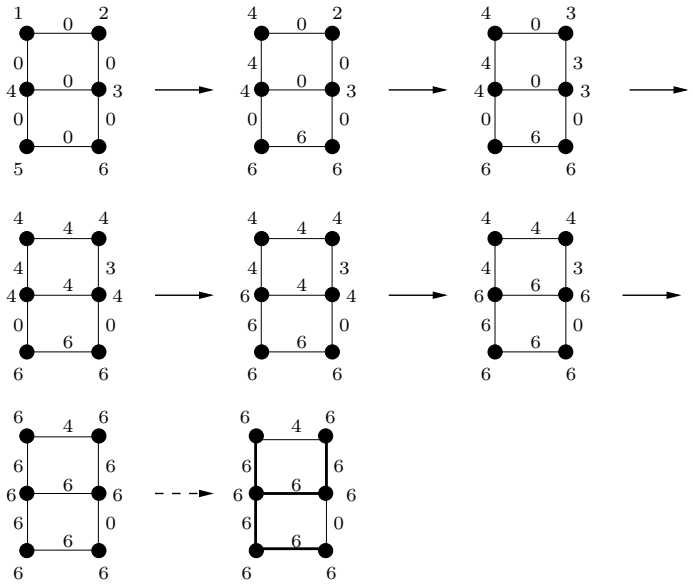


Fig. 6 – Exemple de calcul d'un arbre recouvrant dans un réseau avec identité

On remarque que, même quand l'algorithme a fini, il ne détecte pas localement la terminaison globale. En effet, à la fin de l'exécution, chaque sommet est étiqueté n et sait par conséquent que localement le calcul est terminé, mais aucun des sommets ne peut décider si le graphe est devenu irréductible, i.e. (globalement) que l'algorithme est terminé.

Exemple 2.5.1. Nous donnons un exemple (FIG. 6) d'un procédé de réécriture qui calcule un arbre recouvrant dans un graphe contenant 6 sommets et 7 arêtes, utilisant le système de réécriture de graphe $\mathcal{R}_6$. Les arêtes en gras sont celles de l'arbre recouvrant final.

2.5.2 Calcul séquentiel d'un arbre recouvrant

Dans cet exemple, nous allons présenter un système de réécriture de graphe qui décrit un algorithme distribué calculant un arbre recouvrant dans un réseau avec identité. Nous allons montrer qu'il a la propriété de détection de la terminaison : il existe au moins un noeud qui détecte la terminaison globale.

Soit G un graphe de taille n. Chaque noeud possède une identité unique ; i.e. un nombre unique entre 1 et n. Considérons la fonction λ, où $\lambda^{(V)} : V \to \{A, M, N, F\} \times [1..n]$ et $\lambda^{(E)} : E \to [0..n]$. Initialement, toutes les arêtes sont étiquetées 0 et chaque sommet est étiqueté (A, i) où i est son identité.

A chaque étape de calcul, le noeud u, étiqueté (A, i), va agir comme suit :

1. Si u a un voisin v étiqueté (X, j) où $j < i$ et $X \in \{A, M, N, F\}$, alors u va activer ce voisin : u devient "marqué" (étiqueté (M, i)), v devient "actif" (étiqueté (A, i)) et l'arête $\{u, v\}$ devient étiquetée i.

2. Si u a un voisin étiqueté (X, j) avec $j > i$ alors u devient "neutre" (étiqueté (N, j)).
3. Si u n'a pas de voisin étiqueté (X, j), où $j < i$ et $X \in \{A, M, N, F\}$, et a un (unique) voisin w étiqueté (M, i) relié avec une arête étiquetée i alors u va réactiver son voisin : u entre dans un état final (étiquette (F, i)) et w devient étiqueté (A, i).

Le calcul est fini dès qu'aucune règle de réécriture ne peut être appliquée (dans ce cas, tous les voisins du noeud étiqueté (A, n) sont étiquetés (F, n)). L'arbre recouvrant est donné par toutes les arêtes étiquetées n.

Nous considérons le système de réécriture de graphe avec priorités $\mathcal{R}_7 = (L, I, P, >)$ défini par $L = \{\{A, M, N, F\} \times [1..n] \cup [0..n]\}$, $I = \{\{A\} \times [1..n] \cup \{0\}\}$, et $P = \{R_1, R_2, R_3\}$ où R_1, R_2 et R_3 sont les règles de réécriture de graphe suivantes :

R_1 : $(A,i) \overset{\alpha}{\bullet\!\!-\!\!\bullet} (X,j) \longrightarrow (M,i) \overset{i}{\bullet\!\!-\!\!\bullet} (A,i)$; si $j < i$; $\alpha \leqslant j$; $X \in \{A, M, N, F\}$

R_2 : $(A,i) \overset{\alpha}{\bullet\!\!-\!\!\bullet} (Y,k) \longrightarrow (N,i) \overset{\alpha}{\bullet\!\!-\!\!\bullet} (Y,k)$; si $i < k$; $Y \in \{A, M, N, F\}$

R_3 : $(M,i) \overset{i}{\bullet\!\!-\!\!\bullet} (A,i) \longrightarrow (A,i) \overset{i}{\bullet\!\!-\!\!\bullet} (F,i)$

avec la relation de priorité : $R_1 > R_3$,
$R_2 > R_3$.

La preuve de la terminaison et de la validité sont obtenues par les lemmes suivants, où n est la taille de G.

Lemme 2.5.3. *Le système de réécriture de graphe $\mathcal{R}_7$ est noethérien.*

Démonstration. Ce système est noethérien parce que l'exécution des règles de réécriture implique que le triplet $(\sum_{\substack{v \in V, \\ \lambda(v) = (A,k)}} (n - k), |G|_A, |G|_M)$ est décroissant pour $>_3$, où $|G|_A$ (resp. $|G|_M$) est le nombre de sommets avec l'étiquette A (resp. M) dans G.

□

La validité est prouvée par le lemme suivant :

Lemme 2.5.4. *Soit (G, λ) un graphe étiqueté connexe tel que chaque sommet a une identité unique, appelée ident, appartenant à $[1..n]$ et une étiquette $(A, ident)$ et toutes les arêtes sont étiquetées 0. Soit (G, λ') un graphe étiqueté tel que : $(G, \lambda) \xrightarrow[\mathcal{R}_7]{*} (G, \lambda')$ et soit $X \in \{A, M, N, F\}$. Alors le graphe (G, λ') satisfait :*

- (I_1) *Les étiquettes des arêtes incidentes à un noeud étiqueté (X, i) sont inférieures ou égales à i.*
- (I_2) *Il existe au maximum un sommet avec l'étiquette (A, i), pour un entier fixe $i > 0$.*
- (I_3) *Il existe au moins un sommet étiqueté (X, j) voisin d'un sommet étiqueté (N, i) avec $j > i$.*
- (I_4) *Un noeud étiqueté (F, i) n'a pas de voisin étiqueté (Y, j), où $j \neq i$ et $Y \in \{A, M, N\}$.*

(I_5) *Tous les noeuds étiquetés (X, n) sont connectés par des arêtes étiquetées n.*

(I_6) *Le sous-graphe de G induit par les arêtes étiquetées i ($i > 0$) n'a pas de cycle.*

(I_7) *Les sommets étiquetés (M, n) forment un chemin simple avec les arêtes étiquetées n. D'ailleurs, une des extrémités de ce chemin est connecté à un noeud étiqueté (A, n) par une arête étiquetée n.*

(I_8) *Si (G, λ') est un graphe irréductible obtenu à partir de (G, λ'), il existe un seul sommet étiqueté (A, n) et tous les autres sommets sont étiquetés (F, n).*

Démonstration. (I_1) Prouvons cette propriété par induction sur la taille de la séquence de réécriture. Initialement, la propriété est juste puisque toutes les arêtes ont étiquetées 0. Supposons maintenant que la propriété est juste après k applications des règles R_1, R_2 ou R_3, nous allons montrer que la propriété reste vraie à l'étape $k+1$. Les règles R_2 et R_3 ne changent ni la deuxième étiquette des noeuds ni les étiquettes des arêtes, donc si à l'étape k la propriété est vraie, elle restera vraie à l'étape $k+1$. Si on applique la règle R_1 sur les noeuds u étiquetés (A, i) et v étiqueté (X, j), et sur l'arête incidente étiquetée α, à l'étape k, on sait que $\alpha \leqslant j$. A l'étape $k+1$, l'étiquette de l'arête incidente à u et v va devenir i et le noeud v va avoir l'étiquette (A, i), donc la propriété reste vraie puisque le deuxième état des noeuds u et v est égal à i qui est l'étiquette de l'arête incidente.

(I_2) La propriété est vraie initialement. Nous assurons que la propriété reste vraie après l'étape k. Si nous appliquons la règle R_1 pour la $(k+1)^{\text{ième}}$ étape, nous avons uniquement un seul sommet étiqueté (A, i), puisque le premier change d'étiquette à (M, i) et un noeud voisin aura l'étiquette (A, i) ; pareillement, si nous appliquons la règle R_3 pour la $(k+1)^{\text{ième}}$ étape, le noeud étiqueté (A, i) aura l'étiquette (F, i) et le noeud étiqueté (M, i) sera étiqueté (A, i). Nous avons donc un seul noeud étiqueté (A, i). L'application de la règle R_2 n'ajoute pas de sommet étiqueté (A, i) mais permet d'en enlever un en lui donnant l'étiquette (N, i). Ainsi, la propriété reste vraie.

(I_3) Initialement aucun sommet n'est étiqueté (N, i). Supposons qu'à l'étape k la propriété est vraie, nous allons montrer qu'à l'étape $k+1$ la propriété reste juste. Seule la règle R_2 crée un noeud étiqueté (N, i) et la règle ne peut être appliquée que si un noeud u est voisin d'un noeud v tel que la deuxième étiquette du noeud v est supérieure à celle du noeud u. L'application de la règle R_3 ne change pas la deuxième étiquette des noeuds donc la propriété reste vraie (par hypothèse d'induction). Si la règle R_1 est appliquée, la deuxième étiquette d'un noeud va augmenter d'où la propriété reste vraie.

(I_4) Cette propriété est la conséquence directe de la relation de priorité : si une étape de réécriture change l'étiquette d'un noeud de (A, i) à (F, i), la priorité implique que le noeud étiqueté (A, i) n'a pas de voisin étiqueté (X, j) où $j < i$.

(I_5) initialement la propriété est vraie puisqu'il n'existe qu'un seul noeud étiqueté (A, n). Nous supposons que la propriété est vraie après k étapes de réécriture. Après k étapes, tous les sommets étiquetés (X, n) sont connectés par des arêtes étiquetées n. Seule

l'application de la règle R_1 sur une arête incidente à un sommet u étiqueté (A, n) va ajouter un nouveau sommet v étiqueté (A, n) tel que u aura l'étiquette (M, n), v aura l'étiquette (A, n) et l'arête incidente sera étiquetée n. Ainsi par induction v est connecté à d'autres sommets étiquetés (X, n) par des arêtes étiquetées n, et comme la nouvelle étiquette de l'arête (u, v) est n, alors la propriété reste vraie après l'application de la règle R_1. Notons que dans les autres cas, aucune arête ne change d'étiquette ni de noeud dont la deuxième étiquette est n.

(I_6) Initialement, toutes les arêtes sont étiquetées 0. Nous supposons que la propriété est vraie après k applications des règles, nous allons démontrer qu'elle reste vraie après la $(k+1)^{\text{ième}}$ étape. quand nous appliquons la règle R_2 ou R_3, nous ne changeons pas les étiquettes des arêtes ainsi la propriété reste vraie. L'application de la règle R_1 va changer la deuxième étiquette d'un noeud v à i et l'arête incidente à i. Par la propriété (I_1), les étiquettes des arêtes incidentes à v sont inférieures à i à l'étape k, ainsi, aucun cycle n'est créé. D'où, la propriété est vraie.

(I_7) Au début, la propriété est vraie puisqu'aucun sommet n'est étiqueté (M, n) et uniquement un seul sommet est étiqueté (A, n). Supposons maintenant que la propriété est vraie à la $k^{\text{ième}}$ étape de réécriture et nous allons prouver qu'elle reste vraie à la $(k+1)^{\text{ième}}$ étape. Si la règle R_1 est appliquée sur le sommet étiqueté (A, n) alors ce sommet sera étiqueté (M, n). Le voisin impliqué par l'application de cette règle sera étiqueté (A, n). Par induction, tous les sommets étiquetés (M, n) forment un chemin à la $k^{\text{ième}}$ étape, ainsi la propriété reste vraie à la $(k+1)^{\text{ième}}$ étape. Si la règle R_3 est appliquée sur les noeuds étiquetés (A, n) et (M, n), le second sommet sera étiqueté (A, n) et le premier sera étiqueté (F, n). De même, par induction la propriété reste vraie. L'application de la règle R_2 ne change pas les noeuds étiquetés (X, n). Comme n est la plus grande identité des noeuds, les noeuds ayant le deuxième état égal à i tel que $i < n$, ne génèrent pas des noeuds étiquetés (M, n) ou (A, n). D'où la propriété reste vraie.

(I_8) Par la propriété (I_7), le graphe irréductible (G, λ') n'a pas de noeud étiqueté (M, n). Le graphe (G, λ') a au moins un noeud étiqueté (F, n) adjacent au sommet étiqueté (A, n) qui est unique, d'après la propriété (I_2). Comme (G, λ') est irréductible, le sommet étiqueté (A, n) n'a pas de voisin étiqueté (X, i). Le résultat est alors une conséquence de la règle (I_4).

□

Théorème 2.5.2. *Le système de réécriture de graphe $\mathcal{R}_7$ calcule un arbre recouvrant et détecte la terminaison globale de l'algorithme.*

Démonstration. Par les propriétés $(I_4), (I_5), (I_6)$ et (I_7), nous déduisons que à la fin de l'exécution, nous avons un arbre recouvrant qui a comme racine le noeud étiqueté (A, n) et tous les autres noeuds sont étiquetés (F, n). Les arêtes de cet arbre sont étiquetées n. De plus, le sommet étiqueté (A, n) qui a tous ses voisins étiquetés (F, n) sait que l'algorithme a terminé de s'exécuter. □

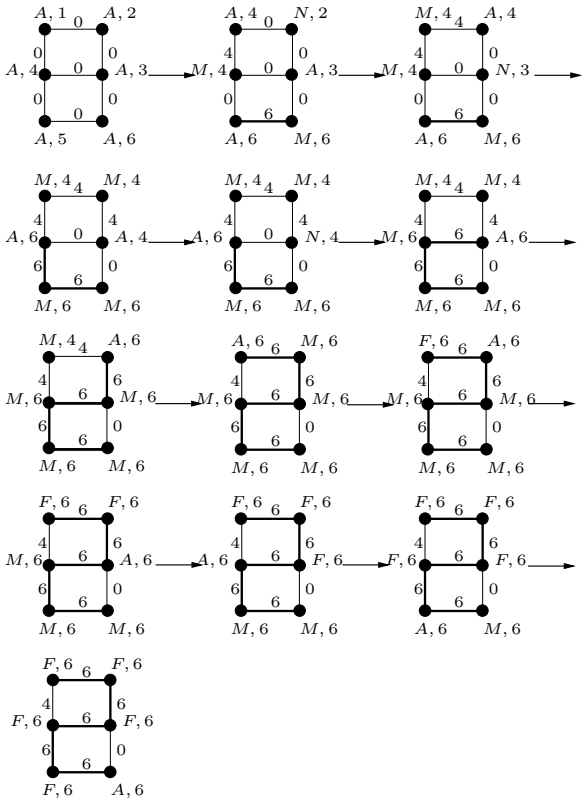


Fig. 7 – Exemple de calcul séquentiel d'un arbre recouvrant dans un réseau avec identité

Remarque 2.5.1. Dans cet algorithme, plusieurs arbres sont calculés en parallèle. Cependant, chaque arbre est calculé d'une manière séquentielle.

Exemple 2.5.2. Nous donnons le même graphe que pour l'exemple précèdent, en exécutant les règles du système $\mathcal{R}_7$. Notez qu'à partir de l'étape 7, le calcul devient séquentiel ; il y a seulement un noeud étiqueté $(A, 6)$.

2.5.3 Calcul d'un arbre recouvrant avec détection globale de la terminaison

Maintenant, nous présentons un système de réécriture de graphe codant un algorithme qui calcule un arbre recouvrant dans un réseau avec identité qui possède ces deux propriétés : l'algorithme est réellement distribué et il détecte localement la terminaison de son exécution.

Soit G un graphe de taille n. Chaque noeud du graphe a une identité unique ; i.e. un nombre unique entre 1 et n. Considérons la fonction de réécriture λ, où $\lambda^{(V)} : V \to \{A, A', F\} \times [1..n]$ and $\lambda^{(E)} : E \to [0..n]$. Initialement, toutes les arêtes sont étiquetées 0 et chaque noeud est étiqueté (A, i) où i est son identité.

L'idée générale de cet algorithme est que le noeud initialement étiqueté (A, n) gardera son étiquette jusqu'à la fin du calcul, tandis que d'autres noeuds activés seront étiquetés (A', n). Dès qu'un sommet étiqueté (A', n) n'est plus "utile" pour la croissance de l'arbre, il atteindra son état final (étiquette (F, n)).

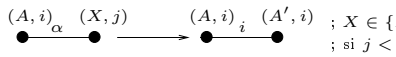
Pour chaque étape de calcul, un noeud "actif" u (avec l'étiquette (A, i) ou (A', i)) agira comme suit :

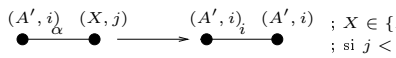
1. Si u a un voisin v étiqueté (X, j), où $j < i$ et $X \in \{A, A', F\}$, alors u va activer ce voisin : u gardera son étiquette, v devient "actif" (étiquette (A', i)) et l'arête $\{u, v\}$ devient étiquetée i.
2. Si u a l'étiquette (A', i), qui n'a pas de voisin étiqueté (X, j), où $j \neq i$ et $X \in \{A, A', F\}$, et tel que tous ses voisins incidents par des arêtes étiquetées i excepté un sont étiquetés (F, i), alors u devient étiqueté (F, i).

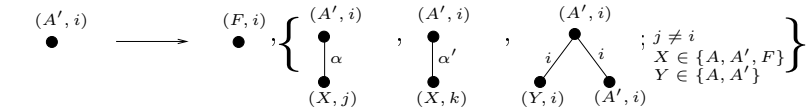
A tout moment, le sous-graphe induit par les arêtes étiquetées n et les sommets étiquetés (A, n) et (A', n) est un arbre.

Ainsi, cet algorithme fonctionne en deux phases (qui peuvent se chevaucher) : dans la première phase, l'arbre se crée jusqu'à ce que tous les sommets soient atteints ; dans la deuxième phase, il décroît (par élimination de ses feuilles) jusqu'à ce qu'il soit réduit au sommet initialement étiqueté (A, n). Ce sommet peut alors détecter que l'algorithme a terminé puisque tous ses voisins sont étiquetés (F, n).

L'algorithme peut être codé par le système de réécriture de graphe avec contextes interdits $\mathcal{R}_8 = (L, I, P)$ défini par $L = \{\{A, A', F\} \times [1..n] \cup [0..n]\}$, $I = \{\{A\} \times [1..n] \cup \{0\}\}$, et $P = \{R_1, R_2, R_3\}$ où R_1, R_2 et R_3 sont les règles de réécriture avec contextes interdits suivantes :

R_1 : 

R_2 : 

R_3 : 

Les règles R_1 et R_2 n'ont pas de contextes interdits : il n'y a pas de restriction pour les appliquer. La règle R_3 a 3 contextes interdits. Nous démontrons que ce systèmes de réécriture de graphe termine et qu'il est valide. La terminaison est donnée par le lemme suivant :

Lemme 2.5.5. *Le système de réécriture de graphe $\mathcal{R}_8$ est noethérien.*

Démonstration. considérons le couple $(\sum_{\substack{v \in V, \\ \lambda(v) = (X,k)}} (n - k), |G|_{A'})$, où $X \in \{A, A', F\}$. Il est décroissant pour $>_2$, ainsi le système est noethérien. □

La validité est donné par :

Lemme 2.5.6. *Soit (G, λ) un graphe étiqueté connexe tel que chaque sommet v est étiqueté (A, i) et chaque arêtes est étiquetées 0, où $i \in [1..n]$ est l'identité du noeud v et n est la taille du graphe G. Soit (G, λ') un graphe étiqueté tel que : $(G, \lambda) \xrightarrow[\mathcal{R}_8]{*} (G, \lambda')$ et $X \in \{A, A', F\}$. Alors le graphe (G, λ') satisfait :*

- (I_1) *Les étiquettes des arêtes incidentes au noeud étiquetés (X, i) sont inférieures ou égales à i.*
- (I_2) *Il existe au plus un sommet avec l'étiquette (A, i), $i > 0$.*
- (I_3) *Il existe exactement un seul noeud étiqueté (A, n).*
- (I_4) *Le noeud étiqueté (F, i) a uniquement des voisins étiquetés (X, i).*
- (I_5) *Tous les noeuds étiquetés (X, n) sont connectés par des arêtes étiquetées n.*
- (I_6) *Les noeuds étiquetés (A, n) et (A', n) sont connectés par des arêtes étiquetées n.*
- (I_7) *Le sous graphe de G induit par les arêtes étiquetées i n'a pas de cycle, $i > 0$.*
- (I_8) *Si (G, λ') est un graphe irréductible obtenue à partir de (G, λ'), il existe uniquement un noeud étiqueté (A, n) et tous les autres noeuds sont étiquetés (F, n).*

Démonstration. (I_1) La propriété est vraie initialement. Nous supposons qu'elle est vraie après k étapes. Dans la $(k+1)^{\text{ième}}$ étape, si nous appliquons la règle R_3, aucune étiquette des arêtes ne sera changée, d'où la propriété reste juste ; si nous appliquons la règle R_1 ou R_2, l'étiquette d'une arête va changer en i, un noeud incident sera étiqueté (X, i) et tous les autres sommets ne vont pas changer ainsi la propriété est toujours vraie.

(I_2) Cette propriété est simple à démontrer puisqu'aucune règle ne permet d'étiqueter un noeud en (A, i) et comme initialement chaque noeud possède une identité unique donc il n'existe pas deux noeuds avec la même étiquette.

(I_3) Cette propriété est une conséquence de la propriété I_2. Comme n est le plus grand identifiant des noeuds, aucune règle ne permet de modifier l'étiquette du noeud étiqueté (A, n).

(I_4) Cette propriété est la conséquence du contextes interdits ; nous ne pouvons pas appliquer la règle R_3 s'il existe un voisin étiqueté (X, j) où $j < i$.

(I_5) La preuve se fait par induction. Initialement, la propriété est vraie puisqu'un seul sommet est étiqueté (A, n). Supposons que la propriété est vraie après k étapes de réécriture et supposons que G_k est le sous-graphe de (G, λ') induit par les sommets étiquetés (X, n). Nous allons démontrer que la propriété reste vraie à la $(k+1)^{\text{ième}}$ étape. Nous n'allons considérer ici que les cas où un noeud change son étiquette à (X, n), parce que dans les autres cas la propriété demeure vraie. Ainsi, si la règle R_1 ou la règle R_2 est appliquée sur un noeud u de G_k et sur son voisin v qui n'appartient pas à G_k, la nouvelle étiquette de v sera (A', n). L'arête liant u et v sera étiquetée n. Donc, la propriété reste juste.

(I_6) Par la propriété (I_5), les sommets étiquetés $(A, n), (A', n)$ et (F, n) sont connectés par des arêtes étiquetées n. Nous allons montrer maintenant que les noeuds étiquetés (A, n) et (A', n) sont connectés par des arêtes étiquetées n. Initialement, la propriété est vraie. Nous supposons que la propriété est vraie après k étapes de réécriture et nous supposons que G_k est le sous-graphe de (G, λ') induit par les sommets étiquetés (A, n) et (A', n). A la $(k+1)^{\text{ième}}$ étape, si la règle R_1 ou la règle R_2 est appliquée à un noeud u de G_k et à un noeud voisin v n'appartenant pas à G_k, la nouvelle étiquette de v sera (A', n). L'arête incidente à u et à v sera étiquetée n. D'où le noeud v appartient à G_{k+1}. Si nous appliquons la règle R_3, nous allons enlever un sommet u de G_k. le nouveau sous-graphe G_{k+1} est connexe puisque u est une feuille d'après les contextes interdits. Par conséquent, la propriété reste vraie.

(I_7) Au début de l'exécution, la propriété est vraie puisque toutes les arêtes sont étiquetées 0. Nous supposons que la propriété est vraie à l'étape k et soit G_i le sous-graphe de (G, λ') induit par les arêtes étiquetées i. Nous allons démontrer que la propriété reste vraie à la $(k+1)^{\text{ième}}$ étape. L'application de la règle R_3 ne va pas changer les étiquettes des arêtes, donc la propriété reste vrai. Si la règle R_1 ou la règle R_2 est appliquée sur un noeud u étiqueté (A, i) ou (A', i) de G_i et sur un noeud v voisin de u étiqueté (X, j), où $X \in \{A, A', F\}$ et $j < i$, l'étiquette du noeud v sera (A', i) et l'arête reliant u et v sera étiquetée i. Par la propriété (I_1), toutes les étiquettes des arêtes adjacentes à v sont inférieures à i à l'étape k donc aucune de ces arêtes n'est dans G_i, D'où, aucun cycle n'est créé. Par conséquent, la propriété reste vraie.

(I_8) Par la propriété (I_3), il existe exactement un sommet étiqueté (A, n). Les voisins de ce sommet sont étiquetés (A', n) ou (F, n) puisque s'il existe un noeud étiqueté (X, i) où $i < n$, nous pouvons appliquer la règle R_1 ou R_2. Nous supposons qu'il existe au moins un noeud étiqueté (A', n), par les propriétés (I_5) et (I_6), le graphe G_n induit par les noeuds étiquetés (A, n) et (A', n) forme un arbre. Ce qui contredit le fait que (G, λ') est irréductible puisque la règle R_3 peut être appliquée. D'où il n'existe pas de noeud étiqueté (A', n). Ainsi, le noeud étiqueté (A, n) n'a que des voisins étiquetés (F, n).

□

Théorème 2.5.3. *Le système de réécriture de graphe $\mathcal{R}_8$ calcule un arbre recouvrant. En plus, il possède la propriété de la détection de la terminaison de l'algorithme.*

Démonstration. Par les propriétés $(I_5), (I_7)$ et (I_8), nous déduisons qu'à la fin du calcul, nous obtenons un arbre recouvrant avec le sommet étiqueté (A, n) comme racine et tous les autres sommets sont étiquetés (F, n). Les arêtes de cet arbre sont les arêtes étiquetées n. Comme nous avons montré qu'il existe uniquement un seul somme étiqueté (A, n), ce sommet détecte la terminaison globale du calcul dès que tous ses voisins sont étiquetés (F, n). C'est une caractérisation que le graphe est irréductible. □

Corollaire 2.5.1. *Cet algorithme nous permet d'élire dans un réseau avec identité. Le sommet qui a la plus grande identité sera élu.*

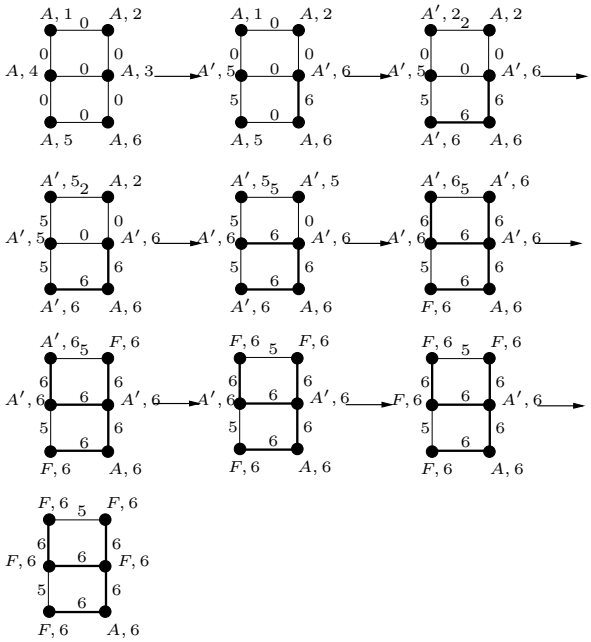


Fig. 8 – Exemple de calcul distribué d'un arbre recouvrant dans un réseau avec identité avec détection de la terminaison globale

Exemple 2.5.3. Nous donnons un exemple d'exécution de cet algorithme sur le graphe utilisé dans les exemples précédents à la figure 8.

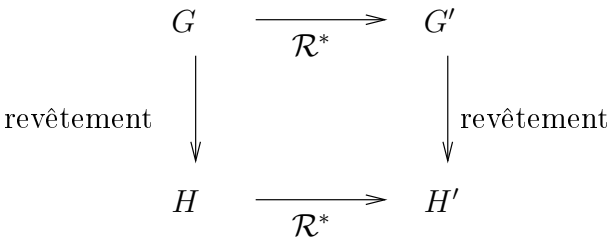
2.6 Revêtements et calculs locaux

On étend la notion de revêtement aux graphes étiquetés de la manière suivante, [32]. γ est un revêtement de (G, λ) sur (H, λ') dont la restriction à $B_G(v)$ est un isomorphisme de $(B_G(v), \lambda_{B_G(v)})$ sur $(B_H(\gamma(v)), \lambda'_{B_H(\gamma(v))})$.

Lemme 2.6.1 (Relèvement [2]). *Soit $\mathcal{R}$ un système de réécriture de graphe et soit $\gamma : G \longrightarrow H$ un revêtement. Supposons qu'il existe un graphe H' tel que $H\mathcal{R}^*H'$. Alors il existe un graphe étiqueté G' (de même graphe sous-jacent que G) tel que :*

- *$G\mathcal{R}^*G'$,*
- *G' soit un revêtement de H'.*

La figure suivante résume ce lemme qui signifie intuitivement que l'on peut simuler sur G toute exécution de $\mathcal{R}$ sur H.

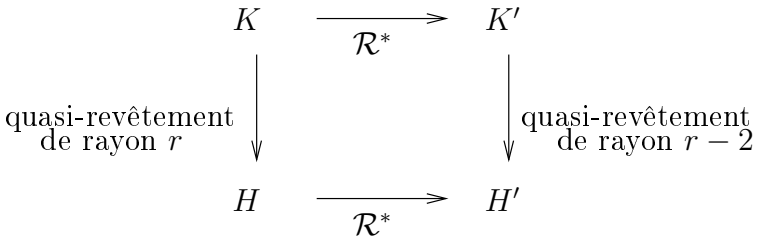


2.7 Quasi-revêtements et calculs locaux

Lemme 2.7.1. *Soit $\mathcal{R}$ un système de réécriture de graphe et soit K un quasi-revêtement de H via γ de rayon $r \geqslant 2$. Supposons qu'il existe un graphe H' tel que $H\mathcal{R}^*H'$.*
Alors il existe un graphe K' tel que :

- *$K\mathcal{R}^*K'$,*
- *K' est un quasi-revêtement de H' de rayon $r-2$.*

Ainsi, dans le cas d'une relation de quasi-revêtement, on peut, comme dans le cas des revêtements, effectuer des simulations mais seulement un nombre fini de fois.



Chapitre 3

Détection de la Terminaison

Dans ce chapitre, nous allons parler de la terminaison dans les algorithmes distribués. Nous proposons, aussi, une méthode unifiée et générale pour détecter la terminaison des calculs distribués. Cette méthode utilise le codage des algorithmes distribués sous forme de systèmes de réécriture de graphe pour transformer le problème à un simple ajout d'un détecteur de terminaison sur un algorithme distribué. Cet ajout se fait par une simple opération sur les systèmes de réécriture de graphe. On obtient ainsi une méthode générale. De nombreux exemples sont codés pour illustrer cette approche.

3.1 Introduction

La détection de la terminaison dans les algorithmes distribués est l'un des problèmes les plus importants dans les calculs locaux dûs à ses nombreuses applications pratiques et théoriques. Il est étroitement lié à d'autres problèmes tels que la détermination d'un état global cohérent [19], détection des "deadlocks" [18], ramasse-miettes distribuées [73], et la reconstruction universelle des graphes [57]. Beaucoup d'algorithmes ont déjà été conçus pour des modèles particuliers et sous diverses suppositions. Cependant, une approche générale et des méthodes unifiées pour traiter le problème de détection de terminaison sont rarement suggérées. C'est pour cela que nous proposons un cadre général basé sur les systèmes de réécriture de graphe pour ajouter un détecteur de terminaison sur les algorithmes distribués. Nous nous sommes basé aussi sur les résultats de Métivier et Tel [72], qui caractérisent les familles de graphes dans lesquelles la détection locale de la terminaison globale est possible. Nous donnons ici des solutions efficaces et pratiques pour ajouter la détection de terminaison à un algorithme distribué.

3.2 Le produit de deux systèmes de réécriture de graphe

Dans cette section, nous allons introduire le produit de deux systèmes de réécriture de graphe. Cette opération sera utilisée pour ajouter la détection de terminaison à un algorithme distribué. L'idée principale est d'appliquer ce produit pour combiner deux systèmes de réécriture : le premier est un algorithme distribué et le second est un algorithme de détection de terminaison.

Définition 3.2.1. Soit $G = (V, E)$ un graphe. Soit L_1 et L_2 deux alphabets finis et soit λ_1 et λ_2 deux fonctions de réécriture. Le résultat du *produit* des deux fonctions de réécriture λ_1 et λ_2, noté $\lambda_1 \times \lambda_2$, est défini comme suit :
$\lambda_1 \times \lambda_2 : V \cup E \longrightarrow L_1 \times L_2$.
$\lambda_1 \times \lambda_2(v) = (\lambda_1(v), \lambda_2(v))$.

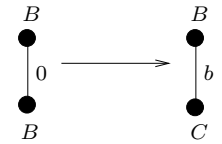
Soit $\lambda = \lambda_1 \times \lambda_2$. Sans perdre la généralité, nous allons étendre la fonction de réécriture λ_1 (resp. λ_2) à λ comme suit. nous allons écrire $\lambda(1)$ (resp. $\lambda(2)$) pour la projection du $i$ème élément de l'ensemble obtenu λ, i.e. $\lambda(i)\colon V \cup E \to L_i$ avec $\lambda(i)(x) = y_i$ pour $\lambda(x) = (y_1, y_2)$. En effet, parfois nous devons manipuler un seul système λ_1 ou λ_2 extrait du système produit λ.

Définition 3.2.2. Soit $\mathcal{R}_f = (L_f, I_f, P_f)$ et $\mathcal{R}_s = (L_s, I_s, P_s)$ deux systèmes de réécriture de graphe. Le produit de $\mathcal{R}_f$ et $\mathcal{R}_s$ est le système de réécriture de graphe $\mathcal{R} = \mathcal{R}_f \times \mathcal{R}_s$ défini par (L, I, P) où $L = L_f \times L_s$, $I = I_f \times I_s$ et $P = P_f \cup P_s$.

L'ensemble des règles de réécriture $\mathcal{R}$ est l'union des règles de réécriture de $\mathcal{R}_f$ et $\mathcal{R}_s$ en accroissant les étiquettes. Pour chaque règle $R_f = (G_{\mathcal{R}_f}, \lambda_{\mathcal{R}_f}, \lambda'_{\mathcal{R}_f})$ de $\mathcal{R}_f$, le système de réécriture de graphe $\mathcal{R}$ contient toutes les règles $R = (G_{\mathcal{R}}, \lambda_{\mathcal{R}}, \lambda'_{\mathcal{R}})$ avec $G_{\mathcal{R}} = G_{\mathcal{R}_f}$, $\lambda_{\mathcal{R}}(1) = \lambda_{\mathcal{R}_f}$ et $\lambda'_{\mathcal{R}}(1) = \lambda'_{\mathcal{R}_f}$. De la même façon, les règles de $\mathcal{R}_2$ sont étendues pour être ajoutées à $\mathcal{R}$.

Exemple 3.2.1. Nous allons illustrer la définition précédente par un exemple. Considérons le système de réécriture de graphe suivant : $\mathcal{S} = (L'_s, I'_s, P'_s)$ où $L'_s = \{B, C, 0, b\}$, $I'_s = \{B, 0\}$ et P'_s contient les règles de réécriture suivantes :

R :



Le produit $\mathcal{R}_1 \times \mathcal{S}$, où $\mathcal{R}_1$ est le système de réécriture de graphe donné dans l'exemple 2.2.1, est le système de réécriture de graphe (L, I, P) où $L = L_{\mathcal{R}_1} \times L'_s$, $I = I_{\mathcal{R}_1} \times I'_s$ et P contient les règles de réécriture suivantes :

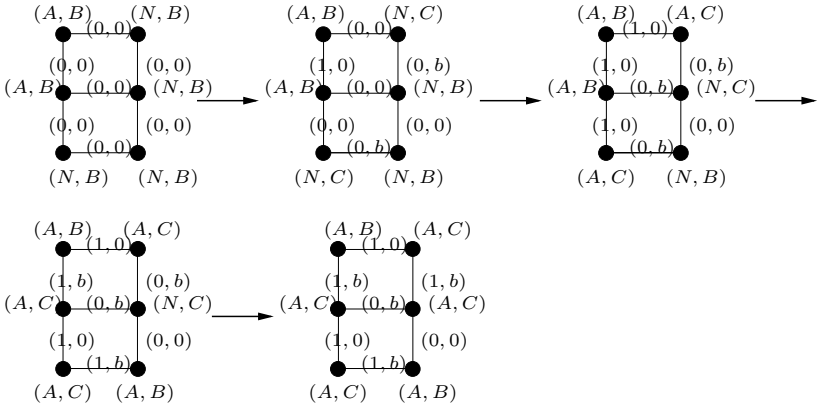
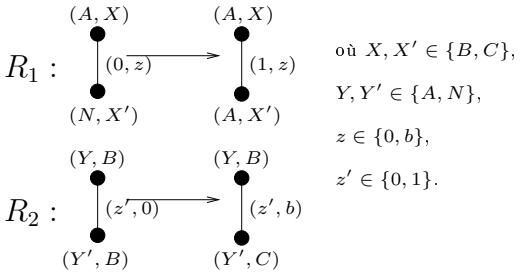


FIG. 9 – Un exemple d'exécution du produit de deux systèmes de réécriture



Un exemple d'exécution de ce système est donné par la figure FIG 9.

Lemme 3.2.1. *Soit $\mathcal{R}_x = (L_x, I_x, P_x)$ et $\mathcal{R}_y = (L_y, I_y, P_y)$ deux systèmes de réécriture de graphe et soit $\mathcal{R} = \mathcal{R}_x \times \mathcal{R}_y$ le produit de $\mathcal{R}_x$ et de $\mathcal{R}_y$. Si $\mathcal{R}_x$ et $\mathcal{R}_y$ terminent, alors $\mathcal{R}$ termine.*

Démonstration. Comme $\mathcal{R}$ est le produit de deux systèmes de réécriture de graphe $\mathcal{R}_1$ et $\mathcal{R}_2$, les règles de réécriture de $\mathcal{R}$ sont l'union des règles de $\mathcal{R}_1$ et $\mathcal{R}_2$ en additionnant les étiquettes comme c'est déjà vu dans la définition 3.2.2. Ainsi, à chaque application d'une règle de réécriture, nous allons appeler réellement une des règles de réécriture de $\mathcal{R}_1$ ou $\mathcal{R}_2$ puisque aucune des règles d'un système n'influence les règles de l'autre système. Par conséquent, si $\mathcal{R}_1$ et $\mathcal{R}_2$ terminent, alors $\mathcal{R}$ termine. Autrement, s'il existe un ensemble S_1 (resp. S_2) qui diminue pour chaque application des règle de réécriture de $\mathcal{R}_1$ (resp. $\mathcal{R}_2$), l'union de S_1 et S_2, appelé l'ensemble S qui résulte des nouveaux états obtenus par l'opération de produit, décroît pour chaque application d'une des règles de réécriture du système $\mathcal{R}$. □

Nous allons, maintenant, prouver la terminaison de l'exemple précèdent $\mathcal{R}_1 \times \mathcal{S}$: Pour chaque application de la règle de $\mathcal{R}_1$, le nombre de noeuds étiquetés N ($|G|_N$) diminue. Pour le système de réécriture de graphe $\mathcal{S}$, le nombre d'arête incidente aux noeuds étiquetés B ($|G|_{(B-B)}$) décroît. Nous notons S_1 le premier nombre et S_2 le second. Alors l'union de ces deux ensemble nouvellement ré-étiquetés sera

$(|G|_{(N,X)}, |G|_{((Y,B)-(Y',B))})$ est noethérien pour chaque application des règles de $\mathcal{R}_1 \times \mathcal{S}$, où $X \in \{B, C\}$ et $Y, Y' \in \{A, N\}$.

Lemme 3.2.2. *Soit $\mathcal{R}_x = (L_x, I_x, P_x)$ et $\mathcal{R}_y = (L_y, I_y, P_y)$ deux systèmes de réécriture de graphe et soit $\mathcal{R} = \mathcal{R}_x \times \mathcal{R}_y$ le produit de $\mathcal{R}_x$ et de $\mathcal{R}_y$. L'ensemble des invariants de $\mathcal{R}$ est l'union de l'ensemble des invariants des deux systèmes de réécriture de graphe $\mathcal{R}_x$ et $\mathcal{R}_y$.*

Démonstration. La preuve de ce lemme est identique à la preuve du lemme précèdent puisque les règles de réécriture de graphe obtenues par le produit des deux systèmes de réécriture est un nouveau système qui permet l'exécution indépendante de ces deux systèmes de réécriture simultanément. Ainsi, comme nous l'avons déjà dit auparavant, chaque règle de $\mathcal{R}$ appartient à $\mathcal{R}_x$ ou $\mathcal{R}_y$. Pour simplifier la preuve, nous notons par $L_{\mathcal{R}_x}$ (resp. $L_{\mathcal{R}_y}$) l'ensemble des états qui appartiennent à $\mathcal{R}_x$ (resp. $\mathcal{R}_y$). Ce qui implique que nous pouvons écrire les étiquettes des règles de $\mathcal{R}$ en deux ensembles indépendants qui appartiennent à $L_{\mathcal{R}_x}$ et $L_{\mathcal{R}_y}$ et aussi, nous pouvons mettre en avant l'indépendance des deux systèmes de réécriture.
L'application des règles appartenant au premier système ne change pas les invariants du second puisque leurs applications ne change que l'ensemble $L_{\mathcal{R}_x}$ et l'ensemble $L_{\mathcal{R}_y}$ ne change pas. De même pour l'application des règles du système $\mathcal{R}_y$, l'ensemble $L_{\mathcal{R}_x}$ ne va pas changer, ainsi les invariants du premier système resterons juste. Par conséquent, les invariants du système $\mathcal{R}$ sont l'union des invariant des systèmes $\mathcal{R}_x$ et $\mathcal{R}_y$. □

Les invariants du système de réécriture de graphe $\mathcal{R}_1 \times \mathcal{S}$ sont :

1. Un noeud qui a l'étiquette (N, X) n'a pas d'arête incidente qui a l'étiquette $(1, Z)$, où $X \in \{B, C\}$ et $Z \in \{0, b\}$,
2. Un sommet dont l'étiquette est (Y, C) a une seule arête incidente avec l'étiquette $(Z', 1)$, où $Y \in \{A, N\}$ et $Z' \in \{0, 1\}$,
3. Tous les noeuds qui ont (A, X) comme étiquette sont connectés par des arêtes étiquetées $(1, Z)$.
4. Le sous-graphe induit par les arêtes étiquetées $(1, Z)$ n'a pas de cycle.
5. Si le graphe G' obtenu à partir de G est irréductible, alors il n'y a pas deux noeuds voisins étiquetés (Y, B) dans le graphe.
6. Si le graphe G' obtenu à partir de G est irréductible, alors tous les sommets ont l'étiquette (A, X).

Nous pouvons remarquer que le deuxième et le cinquième invariants appartiennent aux invariants du système $\mathcal{S}$ et les autres au système $\mathcal{R}_1$.

3.3 La détection de la terminaison dans les systèmes de réécriture de graphe

3.3.1 Le problème de la détection de terminaison

Nous considérons un système distribué où les processeurs communiquent uniquement par passage de message. Dans un tel système, un processeur échange ses messages exclusivement avec ses voisins. Par conséquent, il n'est pas facile pour un processeur d'avoir une mise à jour des informations des états de tout le système, et ensuite de décider ou de savoir si le calcul distribué est fini. Il existe deux types de terminaisons : terminaison implicite et explicite. Dans une terminaison implicite, chaque exécution est finie et dans la dernière exécution de l'algorithme, chaque noeud est dans un état final et exact. Cependant, les noeuds ne se rendent pas compte que leur état est le dernier dans l'exécution, donc aucun noeud ne sait que l'exécution est finie. La terminaison est dite explicite s'il existe au moins un processus qui détecte que c'est fini. Il y a beaucoup de propositions d'algorithme pour la *détection de la terminaison* : tel que les algorithmes qui transforment la terminaison implicite en terminaison explicite. Plusieurs conditions et hypothèses apparaissent permettant de proposer des algorithmes qui résolvent ce problème (voir [72, 6]). Dans ce chapitre, le mot terminaison désigne la terminaison explicite.

Dans la section suivante, nous donnons une solution qui permet de détecter la terminaison globale dans un système de réécriture de graphe. Nous utilisons les techniques de détection de la terminaison comme celle de l'algorithme de Dijkstra-Scholten [25], ou l'algorithme de détection des propriétés stables [71]. En effet, nous réduirons le problème de détection de terminaison en un produit de deux systèmes de réécriture de graphe. Nous commençons par décrire quelques algorithmes de détection de la terminaison codés sous forme de système de réécriture de graphe. Ensuite, nous montrons comment les utiliser pour détecter la terminaison d'autres algorithmes distribués.

3.3.2 Caractérisation de la détection de terminaison

Dans cette sous-section, nous allons rappeler les résultats de Godard, Métivier et Tel [57, 34].

Théorème 3.3.1 (Impossibilité). *Soit I une classe des graphe étiquetés connexes. S'il existe un graphe $G \in I$ tel que G a un quasi-revêtement d'une taille arbitraire grande dans I alors il n'y a pas de terminaison explicite pour la classe I.*

Théorème 3.3.2 (Possibilité). *Soit I une classe de graphes étiquetés connexes. Si pour tout graphe $G \in I$ il existe un entier $h_G \geqslant 0$ tel que G n'a pas de quasi-revêtement d'une taille plus grande que h_G dans I alors il existe une relation de ré-étiquetage locale avec la détection de la terminaison explicite qui calcule pour tout $G \in I$ un graphe G' dans I tel que G est un revêtement de G'.*

Théorème 3.3.3. *Soit I une classe de graphes étiquetés connexes. Si pour tout graphe $G \in I$ il existe un entier $h_G \geqslant 0$ tel que G n'a pas de quasi-revêtement d'une taille plus grande que h_G dans I alors toute relation de ré-étiquetage locale ayant la propriété de la terminaison implicite peut se transformer en une relation de réétiquetage locale ayant la propriété de la terminaison explicite.*

Corollaire 3.3.1. *Les classes de graphes pour lesquelles nous pouvons détecter la terminaison explicite sont :*

- *graphes ayant un leader,*
- *graphes avec des identités,*
- *graphes dont on connaît la taille ou une borne du diamètre,*
- *arbres,*
- *graphes triangulés,*
- *graphes ayant exactement k leaders,*
- *graphes ayant au moins 1 et au plus k leaders,*
- *le diamètre du graphe est au plus égal à d,*
- *la taille du graphe est au plus égale à n.*

3.3.3 Exemples d'algorithmes de détection de la terminaison

Dans cette sous-section, nous présentons quelques algorithmes distribués qui détectent la terminaison globale dans les systèmes de réécriture de graphe.

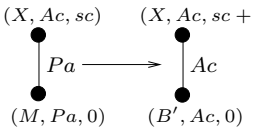
L'algorithme de Dijkstra-Scholten

Considérons un réseau de processeurs représenté par un graphe. Chaque processeur, représenté par un noeud, est toujours dans un de ces deux états *actif* ou *passif*. Les processeurs communiquent seulement par des messages. Certains de ces messages sont des messages d'activation ; ils permettent de rendre l'état du processeur récepteur à l'état actif. Un processeur dans l'état actif maintient un compteur qui compte le nombre de noeuds qu'il a activés et qui n'ont pas encore envoyé un message d'acquittement. Initialement, un seul processeur est dans l'état actif ; ensuite il active ses voisins. L'ensemble des noeuds actifs forment un arbre dynamique où la racine est le noeud actif initial, et un noeud est le père des noeuds qu'il a activés. Un noeud feuille est supprimé de l'arbre s'il n'a aucun noeud descendant actif ; i.e., il a reçu un message d'acquittement pour tous les messages d'activation qu'il a envoyés. Dans ce cas, il envoie un signal d'acquittement à son père qui va décrémenter son propre compteur de ses fils actifs. Nous appelons cet arbre l'*arbre de contrôle*. Le calcul termine quand la racine n'a plus de descendants actifs.

Nous proposons un système de réécriture de graphe qui code cet algorithme. Pour cela, nous associons pour chaque noeud du graphe un compteur (appelé *sc*) et un état décrivant

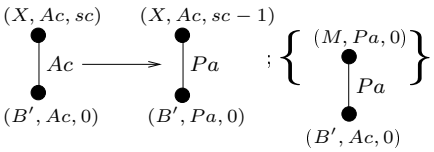
sont état d'activation (les états sont $\{Ac, Pa\}$). Par conséquent, un sommet sera étiqueté par un triplet (X, Y, sc), où X est un état M, B, B' ; Y est son état d'activation, et sc est le compteur décrit ci-dessus. Nous utilisons l'état M pour les noeuds qui n'ont jamais été activés, B pour le noeud initial et B' pour les noeuds qui ont été activés au moins une fois. Une arête sera étiquetée par un état dans $\{Ac, Pa\}$ indiquant si l'arête appartient ou non à l'arbre de contrôle. Initialement, toutes les arêtes sont dans l'état Pa et chaque sommet est étiqueté $(M, Pa, 0)$; sauf le noeud distingué qui a l'étiquette $(B, Ac, 0)$ et qui sera la racine de l'arbre de contrôle. Ce noeud initialisera l'algorithme. Nous supposons que le graphe initial est fini. Dans ce cas, la valeur du compteur sc pour chaque noeud est finie et elle est limité par le degré maximal du graphe.
Soit $\mathcal{R}_9 = (L, I, P)$ un système de réécriture de graphe défini par $L = \{\{B, B', M\} \times \{Ac, Pa\} \times [0..n]\} \cup \{Pa, Ac\}$, $I = \{(B, Ac, 0), (M, Pa, 0), Pa\}$, et P comprenant les règles de réécriture suivantes :

R_1 :



$; X \in \{B, B'\}$

R_2 :



Pour prouver la correction de cet algorithme, nous donnons les invariants suivants. Soit $G(V, E, \lambda)$ un graphe étiqueté connexe tel que le noeud distingué est étiqueté $(B, Ac, 0)$, tous les autres noeuds sont étiquetés $(M, Pa, 0)$ et toutes les arêtes sont étiquetées Pa. Soit $G'(V, E, \lambda')$ un graphe étiqueté tel que : $G(V, E, \lambda) \xrightarrow[\mathcal{R}_9]{*} G'(V, E, \lambda')$. Le graphe $G'(V, E, \lambda')$ satisfait :

1. Toutes les arêtes incidentes à un noeud étiqueté $(M, Pa, 0)$ sont étiquetées Pa.
2. Il y a exactement un seul sommet étiqueté (B, Ac, sc).
3. Un noeud étiqueté (B, Ac, sc) a au moins une arête incidente étiquetée Ac, $(sc \neq 0)$.
4. Un noeud étiqueté (B', Ac, sc) a exactement $sc + 1$ arête incidente étiquetée Ac.
5. Le sous-graphe induit par les arêtes étiquetées Ac n'a pas de cycle.
6. Un noeud étiqueté $(B', Pa, 0)$ n'a pas de noeud voisin étiqueté $(M, Pa, 0)$.
7. Un noeud étiqueté $(B', Pa, 0)$ n'a au plus qu'une seule arête incidente étiquetée Ac.
8. Les noeuds étiquetés (B, Ac, sc) et (B', Ac, sc') sont connectés par des arêtes étiquetées Ac.
9. Si G' est un graphe irréductible, alors tous les noeuds sont étiquetés $(B', Pa, 0)$ excepté un étiqueté $(B, Pa, 0)$.

La détection de la terminaison de l'algorithme se produit dès que le compteur (sc) de la racine redevient zéro (0).

Démonstration. 1. Initialement, la propriété est vraie puisque tous les noeuds sont étiquetés $(M, Pa, 0)$ seulement un seul étiqueté $(B, Ac, 0)$ et toutes les arêtes ont Pa comme étiquette. Supposons que nous ayons appliqué les règles de réécriture k fois et que la propriété reste vrai. Si nous appliquons la règle R_1 pour la $(k+1)^{\text{ième}}$ étape, uniquement un seul noeud, noté v, étiqueté $(M, Pa, 0)$ va changer son étiquette à $(B', Ac, 0)$. Puisque par induction, toutes les arrêts incidentes à v sont étiquetées Pa, et puisqu'aucune autre arête ne change d'étiquette. L'application de la règle R_2 ne va pas changer aucun noeud étiqueté $(M, Pa, 0)$ et aucune arête étiquetée Pa. Par conséquent, la propriété est vraie après l'étape de réécriture.

2. La propriété est vraie initialement. Nous supposons qu'elle est vraie après l'étape k. Si nous appliquons la règle R_1 pour la $(k+1)^{\text{ième}}$ étape, nous avons qu'un seul noeud étiqueté (B, Ac, sc) qui ne va pas changer son état uniquement son compteur sc ; de même, si nous appliquons la règle R_2 pour la $(k+1)^{\text{ième}}$ étape, le noeud étiqueté (B, Ac, sc) va changer son étiquette à $(B, Ac, sc-1)$. Ainsi les deux premières étiquettes ne change pas. Par conséquent, la propriété demeure vraie.

3. Pour prouver cette propriété, nous allons montrer que le compteur sc est le nombre des fils du noeud. Initialement, c'est vraie puisque toutes les arêtes sont étiquetées Pa et tous les compteurs sc des noeuds sont zéro. Supposons que la propriété est vraie après k applications de la règle R_1 ou R_2, nous allons démontrer que cette propriété demeure vraie après l'application de la règle R_1 ou R_2, i.e. à l'étape $k+1$. Par le dernier invariant, il n'y a qu'un seul noeud avec l'étiquette (B, Ac, sc) et par induction chaque compteur sc représente le nombre de sommets qui ont été activés par ce noeud. Alors l'application de la règle R_1 active un noeud passif, incrémente son compteur sc et le nombre d'arête incidente étiqueté Ac. Donc, le compteur sc est le nombre de fils du noeud. La règle R_2 désactive un noeud en changeant sont étiquette en $(B', Pa, 0)$ et informe son père qui décrémente son compteur sc et change l'état de l'arête incidente en Pa. Par conséquent, le compteur sc est le nombre des noeuds et d'arête actifs.
   Ainsi, si la valeur du compteur sc d'un noeud étiqueté (B, Ac, sc) est supérieur à zéro, il existe exactement $sc+1$ arêtes incidentes étiquetées Ac et un nombre supérieur ou égal de sommets étiquetés (B', Ac, sc').

4. Initialement, la propriété est vraie puisque tous les noeuds sont étiquetés $(M, Pa, 0)$ seulement un seul étiqueté $(B, Ac, 0)$ et toutes les arêtes ont Pa comme étiquette. Supposons que nous ayons appliqué les règles de réécriture k fois et que la propriété reste vrai. Si nous appliquons la règle R_1 pour la $(k+1)^{\text{ième}}$ étape sur le noeud u étiqueté (B', Ac, sc) et sur le noeud v étiqueté $(M, Pa, 0)$, d'après la règle de récurrence, u possède $sc+1$ arêtes incidentes étiquetées Ac. Donc à l'étape $k+1$, le compteur de u devient $sc+1$, et l'arête incidente devient étiquetée Ac. Pour le noeud v, D'après le premier invariant, à l'étape k, toutes les arêtes sont étiquetées Pa, donc à l'étape $k+1$, v aura une seule arête étiquetée Ac. D'où la propriété est vraie. L'application de la règle R_2 va diminuer le compteur d'un noeud étiqueté (B', Ac, sc) et va changer l'étiquette d'une arête étiquetée Ac incidente à ce noeud à Pa. Par conséquent, la

propriété est vraie après l'étape de réécriture.

5. Au début, toutes les arêtes sont étiquetées Pa. Nous supposons que la propriété est vraie après k applications des règles de réécriture, démontrons qu'après $(k+1)^{\text{ième}}$ étape la propriété reste vraie. Quand nous appliquons la règle R_1, nous allons ajouter une nouvelle arête à l'ensemble des arêtes étiquetées Ac. Mais par le premier invariant, toutes les arêtes incidentes aux sommets étiquetés $(M, Pa, 0)$ sont étiquetées Pa, La nouvelle arête ne va pas créer de cycle. Si nous appliquons la règle R_2, nous allons enlever une arête de l'ensemble des arêtes étiquetées Ac, ainsi nous ne créons pas de cycle.

6. Cette propriété est la conséquence du contexte interdit ; nous ne pouvons pas appliquer la règle R_2 s'il existe un voisin étiqueté $(M, Pa, 0)$.

7. Seule la règle R_2 permet de changer l'étiquette d'un noeud u étiqueté $(B', Ac, 0)$ en $(B', Pa, 0)$. D'après le quatrième invariant, le noeud u a une seule arête étiquetée Ac. La règle R_2 change l'étiquette de cet arête en Pa, d'où il n'y a aucune arête incidente qui est active (avec l'état Ac).

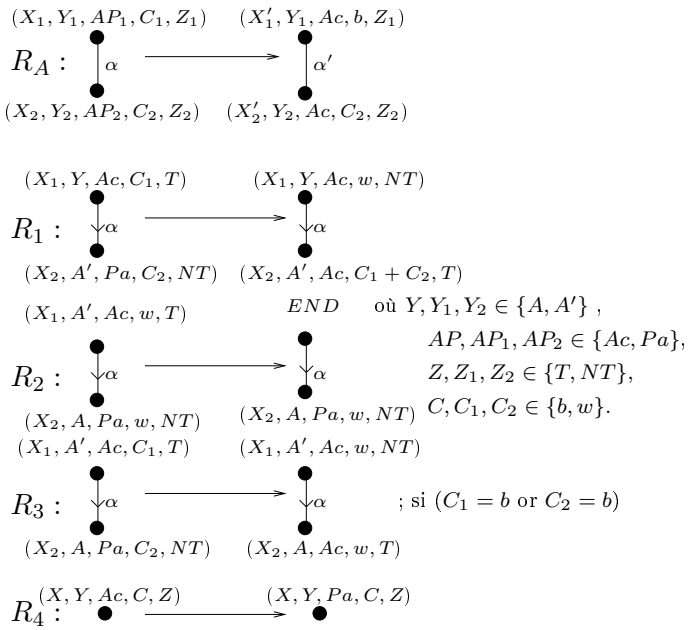
8. La preuve est par induction. Initialement, c'est vraie puisqu'il n'y a qu'un seul sommet étiqueté $(B, Ac, 0)$ et tous les autres noeuds sont étiquetés $(M, Pa, 0)$. Supposons qu'après k applications de la règle, la propriété est vraie. Soit H un sous-graphe connectant tous les noeuds étiquetés (B, Ac, sc) et (B', Ac, sc') et les arêtes étiquetées Ac. Supposons que nous allons appliquer la règle R_1 pour la $(k+1)^{\text{ième}}$ étape. Un nouveau sommet, nouvellement étiqueté $(B', Ac, 0)$, sera connecté à H par son voisin qui l'a activé et qui appartient à H, et l'arête qui lie ces deux noeuds sera étiqueté Ac. Si nous allons appliquer la règle R_2 pour la $(k+1)^{\text{ième}}$ étape, par le dernier invariant, un noeud u avec l'étiquette $(B', Ac, 0)$ ne possède qu'une seule arête incidente étiquetée Ac. Ainsi u est une simple arête incidente dans H. Supprimons u de H ne va pas changer la propriété que le graphe H' est connecté. Ainsi, la propriété reste vraie.

9. Par le deuxième invariant, il n'existe qu'un seul noeud étiqueté (B, Ac, sc). Ses voisins sont étiquetés (B', Ac, sc') ou $(B', Pa, 0)$ puisque s'il existe un noeud étiqueté $(M, Pa, 0)$ nous pouvons appliquer la règle R_1. Nous supposons qu'il existe au moins un noeud étiqueté (B', Ac, sc') où $sc' \neq 0$, par le cinquième et huitième invariants, le graphe G_n induit par les sommets étiquetés (B', Ac, sc') et le sommet étiqueté (B, Ac, sc) forment un arbre. Ceci contredit le fait que G' est un graphe irréductible puisque il existe au moins un noeud avec l'étiquette $(B', Ac, 0)$, ce qui permet l'application de la règle R_2. Ainsi il n'y a aucun noeud étiqueté (B', Ac, sc'). Par conséquent, le noeud étiqueté $(B, Ac, 0)$ n'a que des noeuds voisins étiquetés $(B', Pa, 0)$.

□

L'algorithme de Dijkstra-Feijen-Van Gasteren

Pour cet algorithme, nous supposons que le graphe contient un anneau qui passe par tous les noeuds du graphe $p_0, p_1, ..., p_{n-1}$ (chemin hamiltonien). Cet anneau est utilisé uniquement pour les messages de terminaison, d'autres messages peuvent circuler normalement dans le réseau. L'algorithme de Dijkstra-Feijen-Van Gasteren [26, 72] utilise un jeton qui tourne autour de l'anneau à partir du noeud p_0. Le jeton et les processeurs sont colorés par noir (black) ou blanc (white), noté b ou w. Initialement, les processeurs sont coloriés en blanc (w) ainsi que le jeton. Si un processeur envoie un message, il devient colorié en noir b (règle R_A). Dans ce cas, le processeur de destination devient actif. Un jeton blanc n'est transmit que dans le cas où un processeur blanc reçoit un jeton blanc ; dans les autres cas, un jeton noir est transmit (les règles R_1 à R_3). Le calcul est fini si un jeton blanc arrive au processeur p_0, et que p_0 est blanc. Nous codons cet algorithme par le système de réécriture de graphe suivant.
Soit $\mathcal{R}_{10} = (L, I, P)$ le système de réécriture de graphe défini par $L = \{A, A'\} \times \{Ac, Pa\} \times \{b, w\} \times \{T, NT\}$, $I = \{(A, Ac, w, T) \cup (A', Pa, b, NT)\}$ et P est constitué par les règles de réécriture de graphe suivantes, où X_1, X_2, X_3, X_4, α et α' sont les étiquettes d'un système de réécriture de graphe dont nous voulons tester la terminaison. T (Token) et NT signifie que le jeton est dans ce noeud ou non.



Lemme 3.3.1 (Terminaison). *Ce système est basé sur la vérification des canaux de communication. S'il n'y a aucun message dans un canal alors l'algorithme a fini. Ainsi, la règle R_A sert à vérifier si les canaux sont vides ou non. Si nous appliquons ce système au système de réécriture de graphe, si les états ou étiquettes, autres que ceux utilisés pour la détection de la terminaison, changent (un message a été envoyé), la couleur du noeud*

expéditeur devient noire (étiquette b). Les règles de R_1 à R_3 testent chaque noeud s'il a fini ou non et passe le jeton au noeud suivant. Comme le jeton change la couleur du noeud à blanc (étiquette w) avant de passer à l'autre noeud et que la couleur ne change en noir que si un message est envoyé, alors l'algorithme ne finit que lorsque la couleur de tous les sommets est blanche (w) ainsi l'état de fin apparaît (étiquette End).

Lemme 3.3.2 (Correction). *La preuve de la validité de l'algorithme est donnée par les invariants suivants.*

(I_1) *Il n'existe qu'un seul noeud ayant T en étiquette.*

(I_2) *Si le jeton devient noir (b), il y reste jusqu'au noeud étiqueté $(, A, , ,)$.*

(I_3) *Un noeud possédant Ac dans les étiquettes ne peut pas recevoir le jeton.*

(I_4) *Chaque noeud actif (Ac) ne change en passif (Pa) que s'il détecte localement la terminaison.*

(I_5) *Chaque changement de l'étiquette des noeuds (un simple envoi de message), active le noeud récepteur et change la couleur de l'expéditeur à la couleur noire (b).*

(I_6) *Chaque noeud change sa couleur à blanc (w) après la réception du jeton.*

(I_7) *Si un noeud étiqueté $(, A, , ,)$ reçoit un jeton noir ou sa couleur est noire, il initialise à nouveau le détecteur de terminaison.*

(I_8) *Si le noeud étiqueté $(, A, , ,)$ reçoit un jeton blanc et sa couleur est blanche, il détecte la terminaison globale de l'algorithme.*

Démonstration. (I_1) Nous allons prouver cette propriété par induction. Initialement, elle est vraie puisque tous les noeuds sont étiquetés (X, A', Pa, b, NT) seul un noeud est étiqueté (X, A, Ac, w, T). Nous supposons maintenant que la propriété est vraie après k applications des règles, nous allons démontrer que la propriété reste vraie après l'application des règles de ce système, i.e. à l'étape $k+1$. Les règles R_A, R_2 et R_4 ne changent pas la position du jeton. L'application de la règle R_1 ou R_3 sur un noeud u ayant le jeton (étiqueté $(, , , , T)$) sur un noeud v étiqueté $(, , , , NT)$ va changer l'étiquette du noeud u à $(, , , , NT)$ et l'étiquette du noeud v à $(, , , , T)$. Puisqu'il n'y a qu'un seul noeud qui a le jeton à la $k^{\text{ième}}$ étape, il n'y a qu'un seul noeud possédant le jeton à l'étape $k+1$. Donc la propriété reste vraie.

(I_2) La couleur du jeton est la couleur de l'ancien jeton plus la couleur du noeud qui possède le jeton. Donc si la couleur du noeud est noir ou la couleur du jeton était noir (b), il demeure noir par application de la règle R_1. Quand le noeud distingué (étiqueté $(, A, , ,)$) reçoit le jeton, la règle R_2 ou R_3 est appliquée, soit il va arrêter l'algorithme ou bien le réinitialise et change la couleur du jeton à blanc (w).

(I_3) C'est une des conditions des règles R_1 et R_3 : Les règles ne peuvent s'appliquer si un noeud est dans la phase active, il a Ac dans son étiquette. Ce sont seulement ces règles qui transmettent le jeton d'un noeud à un autre.

(I_4) Comme l'algorithme possède la propriété de terminaison locale, la règle R_4 va être appliquée à tous les sommets et un noeud actif (Ac) deviendra passif (Pa). Par conséquent, la propriété est vraie.

(I_5) C'est la conséquence de l'application de la règle R_A. Si on applique une de ces règles sur un noeud u et sur un noeud v, donc on va changer au moins un état de ces sommets, nous pouvons dire qu'un message est envoyé du noeud u vers le noeud v. L'application de cette règle change la couleur du noeud u au noir (b) et active le noeud v (Ac).

(I_6) Les règles qui nous intéressent et qui utilisent le jeton sont les règles R_1, R_2 et R_3. Or la règle R_2 arrête le cycle du jeton, donc elle n'influe pas sur la propriété. La règle R_1 change le jeton du noeud u vers le noeud suivant, le noeud v. Pour cela, u doit posséder le jeton et v doit être dans un état passif (Pa) pour pouvoir appliquer la règle. A la phase de réécriture, le noeud u qui possédait le jeton (T) change sa couleur à la couleur blanche (w) et donne le jeton à v. Donc, la propriété reste vraie. Le même raisonnement si nous appliquons la règle R_3 sauf que le noeud v doit être étiqueté $(, A, , , NT)$ et au moins un des noeuds u ou v doit avoir la couleur noir (b). Après l'application de cette règle, le noeud u qui possédait le jeton change sa couleur à la couleur blanche (w). Par conséquent, la propriété reste toujours vraie.

(I_7) Cette propriété est le résultat de l'application de la règle R_3. Par la propriété I_5, si un noeud envoie un simple message, il change sa couleur en noir, et par la propriété I_2, si le jeton est noir (b), il reste noir jusqu'à ce qu'il arrive au noeud distingué (étiqueté $(, A, , ,)$. Donc, nous allons appliquer la règle R_3 puisque la couleur du jeton est noir ou la couleur du noeud distingué est noir (b), ainsi, le noeud étiqueté $(, A, ,)$ aura le jeton (T) et va changer sa couleur à la couleur blanche (w). Par conséquent, il initialise la détection de la terminaison globale.

(I_8) Par les propriétés I_2, I_5 et I_4, chaque noeud a détecté sa terminaison locale et n'a pas envoyé de message simple depuis le dernier passage du jeton (I_6 et I_7). Donc, quand le jeton arrive au noeud u qui est le noeud qui précède le noeud distingué v, et il est capable d'appliquer la règle R_2, le noeud v va détecter la terminaison globale puisque la couleur du jeton est blanche (w) ainsi tous les noeuds du graphe ont fini localement. Par conséquent, la propriété est vraie.

□

SSP

Nous décrivons l'algorithme de Szymanski, Shi et Prywes (l'algorithme SSP) [71].

Nous considérons un algorithme distribué qui termine quand tous ces processus atteignent leurs conditions locale de terminaison. Chaque processus est capable de déterminer uniquement sa propre condition de terminaison. L'algorithme SSP détecte l'instant où le calcul total est achevé.

Soit G un graphe simple et connexe. Pour chaque noeud v, nous lui associons un prédicat $P(v)$ et un entier $a(v)$. Initialement, $P(v)$ est à "false" (faux) et $a(v)$ est égale à -1. Les modifications de la valeur de $a(v)$ sont définies par les règles suivantes.

Chaque calcul local agit sur l'entier $a(v_0)$ associé au noeud v_0; la nouvelle valeur de $a(v_0)$ dépend des valeurs associés au voisins de v_0. Plus précisément, soit v_0 un sommet et soit $\{v_1, ..., v_d\}$ l'ensemble des noeuds voisins de v_0.
Nous considérons dans cette section l'hypothèse suivante : pour chaque noeud v, si la valeur de $P(v)$ devient "true" (vrai) elle y reste jusqu'à la fin de l'algorithme.

- Si $P(v_0) = false$ alors $a(v_0) = -1$;
- si $P(v_0) = true$ alors $a(v_0) = 1 + Min\{a(v_k) \mid 0 \leqslant k \leqslant d\}$.

Nous considérons l'hypothèse suivante :

Pour tout noeud v du graphe, si la valeur de $P(v)$ devient "true", elle y reste jusqu'à la fin de l'exécution.

Nous allons utiliser les notations suivantes : soit $(\mathbf{G}_i)_{0 \leqslant i}$ une chaîne de réécriture associée à l'algorithme SSP. Nous dénotons par $P_i(v)$ (resp. $a_i(v)$) la valeur du prédicat (resp. le compteur) associés au noeud v de $\mathbf{G}_i$.

Nous obtenons [71] :

Proposition 3.3.1. *Soit $(\mathbf{G}_i)_{0 \leqslant i \leqslant n}$ une chaîne de réécriture associée à l'algorithme SSP ; soit v un noeud de G, on suppose que $h = a_i(v) \geqslant 0$. alors :*

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

A partir de cette proposition, nous déduisons que si un noeud connaît la taille du graphe (ou une borne de la taille, ou le diamètre du graphe, ou une borne du diamètre) alors il peut détecter quand $P(v)$ est vrai (true) pour tous les sommets du graphe.

3.3.4 Composition des systèmes de réécriture de graphe pour la détection de la terminaison globale

Soit $\mathcal{R}_\mathcal{A} = (L_\mathcal{A}, I_\mathcal{A}, P_\mathcal{A})$ un système de réécriture de graphe codant un algorithme distribué dans lequel le détecteur de terminaison sera ajouté. Nous supposons qu'après un nombre fini d'étapes, chaque noeud termine localement ; C'est-à-dire, aucune règle de $P_\mathcal{A}$ ne peux être appliquée. Notez qu'une telle configuration peut être facilement testée en utilisant les contextes interdits comprenant l'ensemble des règles de $P_\mathcal{A}$. Pour le système de réécriture de graphe de l'exemple 2.2.1, la terminaison locale d'un noeud est atteinte quand son état est A et l'état de tous ses voisins est A. Notre but est d'appliquer l'algorithme de détection de terminaison sur les sommets qui sont dans un état final pour établir une détection globale de terminaison. Plus précisément, soit $\mathcal{R}_\mathcal{T} = (L_\mathcal{T}, I_\mathcal{T}, P_\mathcal{T})$ un système de réécriture de graphe codant un algorithme de détection de la terminaison. Nous supposons

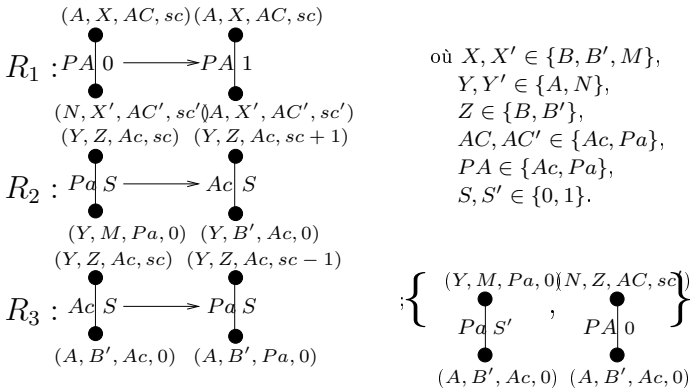
que la règle R_{Pa} de $P_{\mathcal{T}}$ change l'état du processeur de l'état actif à l'état passif (ou de l'état *False* à l'état *True* comme pour l'algorithme SSP). Afin de détecter la terminaison globale de $\mathcal{R}_{\mathcal{A}}$ en utilisant $\mathcal{R}_{\mathcal{T}}$, nous construisons le produit $\mathcal{R}_{\mathcal{A}} \times \mathcal{R}_{\mathcal{T}}$ et nous ajoutons à la règle R_{Pa} un ensemble de contextes interdits indiquant la terminaison locale de l'algorithme pour le sommet Courant (aucune règle de l'algorithme ne sera appliquée sur le sommet courant). Nous notons le système de réécriture résultant par $\mathcal{R}_{\mathcal{A}} \otimes \mathcal{R}_{\mathcal{T}}$.

Théorème 3.3.4. *Soit $\mathcal{R}_{\mathcal{A}} = (L_{\mathcal{A}}, I_{\mathcal{A}}, P_{\mathcal{A}})$ un système de réécriture de graphe codant un algorithme distribué $\mathcal{A}$. Soit $\mathcal{R}_{\mathcal{T}} = (L_{\mathcal{T}}, I_{\mathcal{T}}, P_{\mathcal{T}})$ un système de réécriture de graphe codant un algorithme de détection de terminaison T. Le système de réécriture de graphe $\mathcal{R}_{\mathcal{TA}} = \mathcal{R}_{\mathcal{A}} \otimes \mathcal{R}_{\mathcal{T}}$ code l'algorithme $\mathcal{A}$ avec la propriété de détection de terminaison.*

Démonstration. La preuve de ce théorème est assez simple. Nous utilisons les propriétés décritent auparavant concernant le produit de deux systèmes de réécriture de graphe et les propriétés de nos deux systèmes. L'opération $\mathcal{R}_{\mathcal{TA}} = \mathcal{R}_{\mathcal{A}} \otimes \mathcal{R}_{\mathcal{T}}$ n'est autre que l'opération $\mathcal{R}_{\mathcal{TA}} = \mathcal{R}_{\mathcal{A}} \times \mathcal{R}_{\mathcal{T}} \cup \mathcal{R}_{Pa}$ où le système de réécriture $\mathcal{R}_{Pa}$ n'est composé que de la règle R_{Pa} avec les étiquettes de $L_{\mathcal{TA}}$. Or, le produit des systèmes de réécriture $\mathcal{R}_{\mathcal{A}}$ et de $\mathcal{R}_{\mathcal{T}}$ donne un nouveau système qui permet l'exécution des deux systèmes simultanément. Seulement, le système $\mathcal{R}_{\mathcal{T}}$ ne va pas s'exécuter puisque par définition, ce système doit avoir un état actif et passif qui permet son exécution ou non ; i.e. si l'état d'un noeud est passif alors les règles du système $\mathcal{R}_{\mathcal{T}}$ peuvent s'appliquer sinon aucune règle appartenant à ce système n'est applicable. D'où le système de réécriture $\mathcal{R}'_{\mathcal{TA}} = \mathcal{R}_{\mathcal{A}} \times \mathcal{R}_{\mathcal{T}}$ est équivalent au système $\mathcal{R}_{\mathcal{A}}$ puisque les règles du système de réécriture $\mathcal{R}_{\mathcal{T}}$ ne sont pas appliquées. L'ajout du système de réécriture $\mathcal{R}_{Pa}$ permet de débloquer le système $\mathcal{R}_{\mathcal{T}}$ et c'est alors qu'il sera fonctionnel. L'ajout de ce système ne va pas modifier les propriétés des lemmes 3.2.1 et 3.2.2, puisque cette règle permet pour chaque noeud u dans l'état final pour le système $\mathcal{R}_{\mathcal{A}}$ (c'est-à-dire que par rapport à son état et aux états de ses voisins, u sait qu'il est dans un état final et qu'il n'appliquerait aucune règle du système $\mathcal{R}_{\mathcal{A}}$), cette règle ne modifiera que l'état "actif" du système $\mathcal{R}_{\mathcal{T}}$ pour le rendre "passif". Donc, le système de réécriture $\mathcal{R}_{\mathcal{A}}$ est exécuté par les noeuds jusqu'à ce qu'un noeud sache qu'il a fini (il est dans un état final), il applique alors la règle R_{Pa} qui permet de débloquer le système $\mathcal{R}_{\mathcal{T}}$, qui s'exécutera à son tour. Comme ce système détecte la terminaison globale du système, si un noeud arrive à l'état final du système $\mathcal{R}_{\mathcal{T}}$ alors ce noeud sait que le système $\mathcal{R}_{\mathcal{A}}$ est fini.
Supposons maintenant qu'un noeud u détecte la terminaison globale du système $\mathcal{R}_{\mathcal{TA}}$ alors que le système $\mathcal{R}_{\mathcal{A}}$ n'est pas encore fini. Il existe au moins un noeud v qui n'a pas terminé l'exécution de ce système. D'après les propriétés du système $\mathcal{R}_{\mathcal{T}}$, le système ne termine que si tous les noeuds du graphe sont dans l'état "passif". Or l'état d'un noeud w ne se transforme en passif que si la règle R_{Pa} est appliquée, c'est-à-dire, w est localement fini. Si u détecte la terminaison alors v a forcement appliqué la règle R_{Pa} ce qui contredit le fait que v n'a pas terminé localement. □

Calcul d'un arbre recouvrant avec l'algorithme de Dijkstra-Scholten pour détecter la terminaison

Nous donnons un système de réécriture de graphe codant un algorithme distribué qui est initialisé par exactement un noeud, pour détecter la terminaison avec la technique précédente, nous devons inclure le calcul de réécriture dans l'arbre de contrôle. Un noeud deviendra passif s'il a terminé localement (aucune règle de réécriture ne peut être appliquer sur son contexte). les règles du système de réécriture de graphe sont modifiées en utilisant le nouveau triplet des étiquettes. Afin de suivre l'arbre de contrôle de l'algorithme de terminaison, une règle de réécriture codant la terminaison locale et l'acquittement doit être ajoutée. Le système de réécriture de graphe de l'exemple 2.2.1 combiné avec l'algorithme de Dijkstra-Scholten devient le système de réécriture de graphe avec contextes interdits suivant $\mathcal{R}_{11} = (L, I, P)$, défini par $L = \{\{A, A', N\} \times \{B, B', M\} \times \{Ac, Pa\} \times [0..n]\} \cup \{\{0, 1\} \times \{Ac, Pa\}\}$, $I = \{(A, M, Ac, 0), (N, B, Ac, 0), (N, M, Pa, 0), (0, Pa)\}$, et P_6 est composé par les règles de réécriture de graphe suivantes :



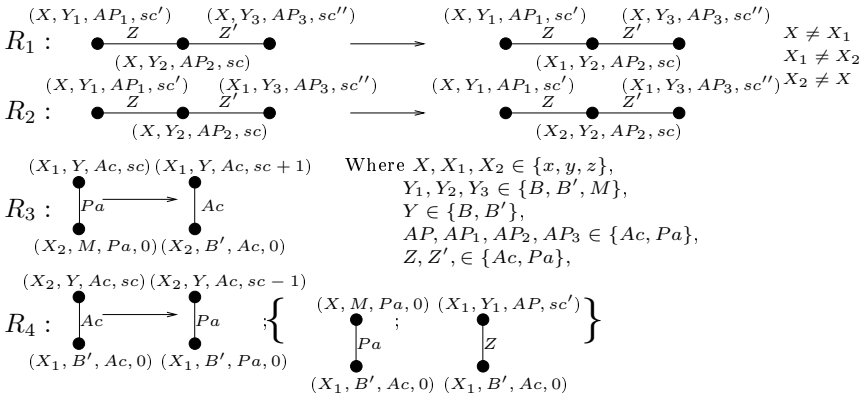
Notez que nous donnons ici une simplification du système obtenu par la combinaison du système de l'exemple 2.2.1 et du système de réécriture de Dijkstra-Scholten. La règle R_1 et la règle R_2 sont similaire aux règles originales, par contre, pour la règle R_3, nous lui avons ajouté un autre contexte interdit comme expliqué avant. un noeud ne termine localement que si son état est A et il n'a pas de voisin qui est dans l'état N, qui n'était pas encore activé. Un tel noeud sera placé dans l'état passif, et envoie un acquittement à son père si son compteur est nul (égale à zéro). Dans ce cas, nous décrémentons le compteur sc de son père. Le calcul est globalement terminé si la racine, le noeud initialement distingué, devient passif et n'a pas de fils dans l'état actif.

Théorème 3.3.5. *Le système de réécriture de graphe $\mathcal{R}_{11}$ calcule un arbre recouvrant. Cet algorithme possède la propriété de détection globale de la terminaison.*

La preuve de ce théorème est basé sur les invariants donnés pour l'algorithme de Dijkstra-Scholten et celui de l'algorithme $\mathcal{R}_1$.

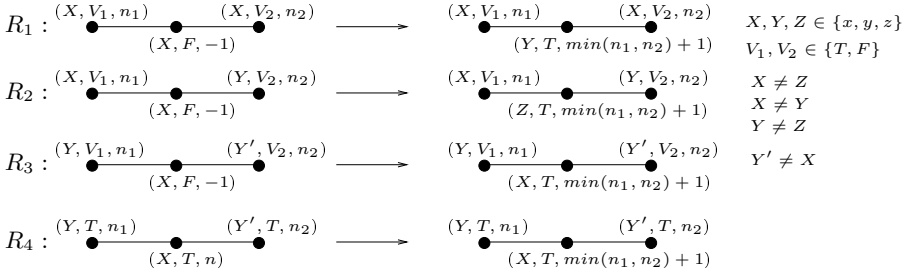
La 3-Coloration d'un anneau avec l'algorithme de Dijkstra-Scholten

Nous considérons le système de réécriture de graphe $\mathcal{R}_5$. Il colorie un anneau avec 3 couleurs, mais il ne détecte pas la terminaison. Nous démontrons par la méthode décrite en dessous comment ajouter un détecteur de terminaison. La propriété de la terminaison locale pour un noeud avec l'état X est que ce noeud n'a pas de voisin avec l'étiquette X, où $X \in \{c_1, c_2, c_3\}$. Nous allons ajouter cette propriété à la deuxième règle de l'algorithme de Dijkstra-Scholten. L'algorithme résultant est le suivant. Soit $\mathcal{R}_{12} = (L, I, P)$, défini par $L = \{\{c_1, c_2, c_3\} \times \{B, B', M\} \times \{Ac, Pa\} \times [0..n]\} \cup \{Ac, Pa\}$, $I = \{(X, B, Ac, 0), (X', M, Pa, 0), Pa\}$, où X et $X' \in \{c_1, c_2, c_3\}$, et P constitué par les règles de réécriture de graphe suivantes :



La 3-Coloration d'un anneau avec l'algorithme SSP

Nous considérons le même exemple que précédemment seule l'algorithme de détection de terminaison qui change, nous utilisons l'algorithme SSP [71] et en plus chaque sommet connaît la taille de graphe ou le diamètre du graphe. Soit $\mathcal{R}_{13} = (L, I, P)$, défini par $L = \{\{c_1, c_2, c_3\} \times \{T, F\} \times [-1..n]\}$, $I = \{(X, F, -1)\}$, où $X \in \{c_1, c_2, c_3\}$, et P est composé par les règles de réécriture de graphe suivantes :



L'algorithme termine si tous les compteurs des noeuds ont une valeur supérieur ou égale à n (D), où n est la taille du graphe (D diamètre du graphe).

3.4 Généralisation du détecteur de terminaison

Dans cette section, nous donnons un résultat général obtenu en utilisant l'algorithme SSP qui détecte localement la terminaison globale. D'abord, nous considérons le cas où chaque noeud est capable de détecter localement sa propre terminaison locale, et où le diamètre ou la taille du graphe est connu. Ensuite, nous considérons une variation de l'algorithme de SSP et nous donnons un résultat plus général.

3.4.1 De la détection de la terminaison locale à la détection de la terminaison globale

Pour le dernier exemple (la 3-Colorations), il est clair que quand un noeud est correctement coloré, il met son prédicat P à la valeur *true*. Dans ce cas, il détecte localement le terminaison locale. Le prédicat P reste à "true" (vrai) jusqu'à la détection locale de la terminaison globale. Plus généralement, l'algorithme SSP peut transformer un système de réécriture de graphe dans lequel chaque noeud est capable de détecter sa propre terminaison locale en un autre système de réécriture de graphe qui détecte localement la terminaison globale.

Théorème 3.4.1. *Soit $\mathcal{R}$ un système de réécriture de graphe. Supposons que pour un graphe dont nous avons la connaissance de la taille du graphe, du diamètre ou d'une borne du diamètre et que chaque noeud est capable de détecter sa propre terminaison locale. Le système $\mathcal{R}$ peut être transformé en un nouveau système de réécriture de graphe qui détecte localement la terminaison globale.*

La preuve est basé sur l'algorithme SSP, et elle est similaire à l'exemple sur les 3-Coloration. En mettant à "true" le prédicat des noeuds qui ont détecter leurs terminaison locale, et en propageant la valeur du compteur, $a(v)$ pour le noeud v, il est possible de détecter la terminaison globale.

3.4.2 La détection de la terminaison avec une variante de l'algorithme SSP

Pour le théorème précédent, nous supposons que chaque noeud est capable de détecter sa propre terminaison locale. Nous pouvons affaiblir cette supposition en s'occupant seulement de l'application des règles. Ainsi, durant le processus de réécriture, si pour un noeud v, aucune règle de réécriture n'est applicable sur u (et sur son entourage), alors la valeur du prédicat $P(v)$ est mise temporairement à *true*. Le processus de calcul continue comme pour le cas précèdent. Maintenant, si plus tard, le noeud v change d'étiquette (par exemple parce qu'un de ses voisins a changé), alors la valeur de son prédicat $P(v)$ est réinitialisé à *false* et $a(v)$ est mis à -1. Si le prédicat P redevient "true", alors le compteur $a(v)$ devient 0 et ainsi de suite. Donc, nous utilisons une généralisation de l'algorithme de SSP en enlevant

la dernière supposition. Nous pouvons prouver qu'il est possible de détecter localement la terminaison globale.

Théorème 3.4.2. *Soit $\mathcal{R}$ un système de réécriture de graphe. Supposons que le diamètre du graphe est connu. le système $\mathcal{R}$ peut être transformé en un système de réécriture de graphe qui détecte localement la terminaison globale.*

Démonstration. Commençons par expliquer le fonctionnement de ce système. Nous pouvons dire qu'il existe une priorité entre l'exécution du système, dont nous voulons détecter la terminaison, et l'algorithme SSP puisque si aucune règle du premier système n'est applicable, alors l'algorithme SSP peut commencer à s'exécuter.
Par la propriété de l'algorithme SSP, le compteur $a(v)$ du noeud v est le rayon de la boule de centre v tel que tous les sommets dans cette boule ont fini. C'est que tous les noeuds dans une boule de centre v et de rayon $a(v)$ ont leurs prédicats $P(v)$ à *true*.
Nous supposons qu'au moins un noeud u a détecté la terminaison globale de l'algorithme. Par la propriété de l'algorithme SSP, le compteur $a(v)$ doit être supérieur ou égale au diamètre du graphe. Ainsi, chaque sommet v a exécute les règles au moins $a(v) - d(u, v) + 1$ fois, où $d(u, v)$ est la distance entre le noeud u et v. Ce qui implique que pour chaque application des règles sur v, il n'a appliqué aucune règle du système $\mathcal{R}$. Donc s'il existe un noeud w qui n'a pas fini, son prédicat $P(w)$ doit être égal à *false*, ce qui contredit le fait que u a fini, ou il a appliqué $a(v) - d(u, w) + 1$ fois l'algorithme SSP, ce qui contredit le fait que les règles de $\mathcal{R}$ sont plus prioritaires que les règles de l'algorithme SSP. Par conséquent, la propriété est vraie. □

Chapitre 4

Les Synchroniseurs

4.1 Introduction

Les études effectuées sur les algorithmes distribués sont faites selon plusieurs modèles. Un des critères fondamentaux est de savoir si le système est synchrone ou asynchrone [21, 72, 47, 67, 3]. Dans le modèle synchrone, nous supposons qu'il y a une horloge globale et que les opérations de calculs se font simultanément : à chaque top d'horloge, les processus effectuent une action. C'est un modèle de temps idéal même si ce n'est pas très réaliste.

En effet, la synchronisation dans un réseau peut être vue comme une structure de contrôle qui permet de contrôler les étapes relatives des différents processus. Elle peut être illustrée par ces exemples :

1. Les processus exécutent des actions dans des étapes de temps appelés pulsations ou tours. Dans un round, un processus q exécute les étapes générales suivantes :
   (a) q envoie un message,
   (b) q reçoit quelques messages,
   (c) q exécute des calculs locaux ;
2. une autre supposition est que les évènements de calculs d'un round p s'effectuent après le calcul du round $p-1$ et que tous les messages envoyés au round p arrivent avant d'effectuer le calcul du round $p+1$;
3. si chaque processus est équipé d'un compteur qui calcule sa synchronisation locale, nous pouvons assumer que la différence entre deux compteurs est au maximum de 1; et plus généralement, pour un entier non négatif donné k nous pouvons assumer que la valeur absolue de la différence entre deux compteurs est au maximum de k.
4. certains algorithmes ont besoin de quelques barrières de synchronisation appliquées à un groupe de processus, ce qui signifie que tous les membres du groupe sont bloqués jusqu'à ce que tous les processus du groupe aient atteint cette barrière.

Tandis que dans le modèle asynchrone il n'y a pas d'horloge globale, chaque composante effectue ses étapes à des vitesses arbitraires. On suppose que les messages sont reçus, les

processus exécutent les calculs locaux et envoient des messages, mais aucune supposition n'est faite au niveau du temps d'exécution de cette procédure.

Il existe aussi des modèles intermédiaires comme le réseau ABD [72, 39]. Dans ce modèle, on connaît une borne maximale des vitesses des processeurs ou des canaux de communications.

4.1.1 Les synchroniseurs

Un synchroniseur est un mécanisme qui permet d'exécuter un algorithme conçu pour les systèmes synchrones dans les systèmes asynchrones.

Comme le non-déterminisme dans les systèmes synchrones est, en général, assez faible, les algorithmes des systèmes synchrones sont plus faciles à concevoir et à analyser que ceux des systèmes asynchrones. Dans les systèmes asynchrones, la conception et l'analyse sont difficiles en absence d'une synchronisation globale des processus. En conséquence, il est utile d'avoir une méthode générale pour transformer un algorithme conçu pour les réseaux synchrones en algorithme qui fonctionne dans les réseaux asynchrones. Donc, il devient possible de concevoir un algorithme synchrone, de le tester et de l'analyser, puis d'utiliser la méthode standard pour l'implémenter dans un réseau asynchrone. Un synchroniseur fonctionne en produisant une séquence d'horloge locale *impulsions* à chaque processeur. Une introduction et les résultats principaux concernant les synchroniseurs se trouvent dans [21, 72, 47, 67, 3].

4.2 Les propriétés du synchroniseur

Tout au long de ce chapitre, plusieurs protocoles distribués de synchronisation seront décrits.

Les opérations des processus prennent place dans une séquence d'étapes appelées des pulsations, tours, rounds ou phases : nous identifions une pulsation à un compteur. Nous associons à chaque processeur un compteur de phase qui est initialisé à 0 ou 1 et, à chaque étape, un processeur va de la pulsation i à la pulsation $i + 1$.

Tous ces protocoles permettent de synchroniser le système à *chaque étape synchrone*. C'est nécessaire parce que les protocoles sont conçus pour fonctionner pour des algorithmes arbitraires synchrones. Tous les synchroniseurs que nous construirons sont "globaux", dans le sens qu'ils permettent la synchronisation entre des noeuds quelconques du réseau entier. Pour préserver cette synchronisation "globale", chaque synchroniseur doit satisfaire quelques propriétés. La propriété essentielle que nous cherchons à préserver en transformant un algorithme générique synchrone $\mathcal{A}_s$ vers un algorithme asynchrone $\mathcal{A}_{as}$, est que la différence de phase entre deux noeuds arbitraires est au plus de 1 (*théorème principal*). Afin de s'assurer que cette propriété est valable pour tous les noeuds et à tout moment ou pulsation, nous commençons par exiger une *compatibilité de phase* dans le réseau, i.e. un

noeud peut seulement augmenter sa pulsation, quand il est sûr qu'il n'y a aucun noeud dans le réseau qui a une pulsation inférieure à la sienne. Cette propriété est garantie par la validité du Théorème 4.2.1. En outre, nous renforcerons notre supposition de synchronisation en forçant une *convergence de phase* à tout moment. Par *convergence de phase*, nous voulons dire que tous les sommets d'un réseau sont simultanément dans la pulsation π avant qu'un noeud quelconque ne commence la pulsation $\pi+1$. La convergence de phase est énoncée par le Théorème 4.2.2. Ainsi, la preuve du Théorème 4.2.3 (*théorème principal*) peut être déduit de la compatibilité de phase et de la convergence de pulsation.
La correction de nos synchroniseurs vont dépendre de la validité des trois théorèmes suivants :

Théorème 4.2.1 (Theoreme de Compatibilité de phase). *Un noeud u de G change sa pulsation $p(u)$ uniquement quand il n'y a aucun noeud v de G tel que $v \neq u$ et $p(v) < p(u)$.*

Théorème 4.2.2 (Theoreme de Convergence). *Soit $\pi \quad (\pi > 0)$ la pulsation maximale qui a été atteinte. Après un nombre fini d'étape $T_\pi \geqslant 0$, tous les sommets de G sont dans la phase π.*

Théorème 4.2.3 (Theoreme principal). *A tout instant, la différence de pulsation entre deux noeuds v et u de G est au maximum de* 1.

4.3 Un synchroniseur simple

Nous rappelons ici la synchronisation comme présentée et utilisée dans [70]. Il y a sur chaque sommet v du graphe un compteur $p(v)$, la valeur initiale de $p(v)$ est 0. À chaque étape la valeur du compteur $p(v_0)$ dépend de la valeur des compteurs des voisins de v_0, plus précisément, si $p(v_0) = i$ et si pour chaque voisin v de v_0, $p(v) = i$ ou $p(v) = i+1$ alors $p(v_0)$ devient $i+1$.

R1 : **la règle de synchronisation**

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

Réétiquetage :
- $p(v_0) := i+1$.

Une simple induction sur la distance entre noeuds prouve que :

Proposition 4.3.1. *Pour tous les sommets v_1 et v_2*

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

Remarque 4.3.1. Cette synchronisation n'a besoin d'aucune connaissance sur le graphe, comme sa taille.

Cette synchronisation est équivalente à la synchronisation décrite dans l'introduction où chaque processeurs exécute ses actions dans une étape bloquante appelée pulsation (ou round) : dans un round, un processeur envoie un message, reçoit des messages et exécute un calcul local.

En effet, pour implémenter cette synchronisation, un compteur modulo 3 est suffisant : chaque processeurs a besoin de comparer la valeur de son compteurs à ceux de ses voisins. Plus précisément, pour chaque processeur v_0 et pour chaque voisin v de v_0, nous déterminons si : $p(v_0) = p(v)-1$, ou $p(v_0) = p(v)$, ou $p(v_0) = p(v)+1$. Finalement, le synchroniseur peut être codé comme suit :

R1 : **La règle de synchronisation**

Précondition :

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

Réétiquetage :

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

Le synchroniseur α [72, 4] est similaire au synchroniseur présent dans cette section. En effet, la pré-condition de la règle R_1 formule que le noeud v_0 est sûr. Par contre, dans notre synchroniseur, un sommet v_0 passe à la pulsation suivante dès qu'il est sûr. Il n'attend pas que tous ses voisins deviennent sûrs. Ainsi, ce synchroniseur ne satisfait pas le Théorème 4.2.3. Il n'est donc pas approprié pour exécuter une synchronisation "globale" dans un réseau. La prochaine section sera consacrée à la présentation des synchroniseurs qui peuvent préserver cette synchronisation "globale" à chaque phase.

4.4 Synchroniseur utilisant l'algorithme SSP

Dans cette section, nous allons décrire un synchroniseur utilisant l'algorithme de détection de propriétés stables de Szymanski, Shi and Prywes (SSP) que nous avons détaillé dans la section 3.3.3.

Soit u un noeud d'un graphe G, l'entier $p(u)$ décrit la valeur de la pulsation associée au sommet u. Dans notre supposition, un noeud v satisfait la propriété de stabilité, s'il n'y a aucun noeud $u \in B_G(v, 1)$ tel que $p(u) \neq p(v)$. Avant de donner une description formelle de l'algorithme, nous commençons par expliquer comment il marche.

Soit G un graphe avec un diamètre D. Pour chaque noeud v de G, on lui associe deux entier $p(v)$ et $a(v)$, où $p(v)$ décrit la pulsation et $a(v)$ décrit le compteur de l'algorithme de SSP pour le noeud v. Initialement, $p(v)$ et $a(v)$ sont respectivement de 1 et 0.

Un sommet v peut passer au nouveau round ($p(v) = p(v) + 1$) et son compteur $a(v)$ est remis à 0 quand il détecte que pour tous les noeuds du graphe, les valeurs de leurs

pulsation est égale à la sienne $p(v)$. Nous donnons, à la suite, une description formelle des arguments ci-dessus.

Nous considérons la fonction d'étiquetage λ, où $\lambda : V \to \mathbb{N}^* \times [0..D]$. Initialement, tous les noeuds sont étiquetés $(1,0)$. Le système de réécriture du graphe est $\mathcal{R}_{14} = (L, I, P)$ définie par $L = \{[1..\infty] \times [0..D]\}$, $I = \{(1,0)\}$, $P = \{R_1, R_2\}$ où R_1 et R_2 sont les règles de réécriture décrit ci-dessous. Soit v_0 un sommet, le synchroniseur utilisant l'algorithme de SSP est définie par :

R1 : **La règle d'observation**

Précondition :

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

Réétiquetage :

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

R2 : **La règle de changement de phase**

Précondition :

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

Réétiquetage :

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

Nous allons utiliser les notations suivantes : soit $(\mathbf{G}_i)_{0 \leqslant i}$ une chaîne de réécriture associée à notre algorithme. Nous dénotons par $p_i(v)$ (resp. $a_i(v)$) la pulsation (resp. le compteur) associés au noeud v de $\mathbf{G}_i$.

Pour prouver la correction de l'algorithme, nous énonçons quelques invariants.

Fait 4.4.1.

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

Fait 4.4.2. *Si $p_{i+1}(v) = p_i(v) + 1$ alors $a_i(v) = D$.*

Lemme 4.4.1. *Si $a_i(v) = h$ alors $i \geqslant h$.*

Démonstration. Nous démontrons ce lemme par induction sur i. Si $i = 0$ la propriété est vrai puisque $h = 0$.

Supposons que la propriété est vrai jusqu'à i, i.e. $\forall k \leqslant i, a_k(v) = h_k$ alors $k \geqslant h_k$. Montrons que la propriété est vrai pour $i + 1$. $a_{i+1}(v)$ peut avoir trois valeurs : soit égale à $a_i(v)$, soit égale à $a_i(v) + 1$ ou égale à 0.

Par l'hypothèse d'induction nous avons : $a_i(v) = h$. Ainsi $i \geqslant h$.

Nous supposons, maintenant, que $a_{i+1}(v) = a_i(v) = h$. Comme $i+1 > i$ et par hypothèse d'induction, nous obtenons $i + 1 > i \geqslant h$. D'où la propriété reste vrai.

Supposons que $a_{i+1}(v) = a_i(v) + 1 = h + 1$. Par hypothèse d'induction, nous avons $i \geqslant h$, donc $i + 1 \geqslant h + 1 = a_{i+1}(v)$. Ainsi la propriété reste vrai.

Dans le dernier cas, nous avons $a_{i+1}(v) = 0$, donc $i + 1 > 0$. Ainsi la propriété est juste. □

Lemme 4.4.2. *Si $p_i(v) = \pi$ et $a_i(v) = h$ alors :*

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

Démonstration. Nous démontrons ce lemme par induction sur i. Si $i = 0$ la propriété est évidente.

Supposons que $p_{i+1}(v) = p_i(v) = \pi$ et $a_{i+1}(v) = a_i(v) = h$. Par l'hypothèse d'induction nous avons : $d(v, w) \leqslant h \Rightarrow p_{i-h}(w) \geqslant \pi$. Par le Fait 4.4.1, $p_{i-h+1}(w) \geqslant p_{i-h}(w) \geqslant \pi$. Ainsi $p_{(i+1)-h}(w) \geqslant \pi$.

Nous supposons, maintenant, que $p_{i+1}(v) = p_i(v) = \pi$ et $a_{i+1}(v) = a_i(v) + 1 = h$. Si $d(v, w) \leqslant h = a_i(v)+1$ alors soit u un noeud tel que $d(v, u) = 1$ et $d(u, w) = a_i(v) = h-1$.

Nous avons $a_{i+1}(v) = a_i(v) + 1 \Rightarrow p_i(u) \geqslant p_i(v) \geqslant \pi$ et $a_i(u) \geqslant h - 1$ (par les pré-conditions de la règle d'observation). En appliquant les hypothèses d'induction sue le sommet u, $p_{i-(h-1)}(w) \geqslant \pi$ et enfin $p_{(i+1)-h}(w) \geqslant \pi$.

Le dernier cas est $p_{i+1}(v) = p_i(v) + 1$, alors nécessairement $a_{i+1}(v) = 0$ ce qui termine la preuve. □

De ce lemme et du Fait 4.4.1, nous avons :

Corollaire 4.4.1. *Si $p_i(v) = \pi$ et $a_i(v) = h$ alors*

$$\forall w \in V(G) \quad d(v, w) \leqslant h \Rightarrow p_i(w) \geqslant \pi.$$

Lemme 4.4.3. *Si $p_i(v) = \pi$ et $p_i(w) = \pi+1$ alors $\forall u \in V(G)$ ($p_i(u) = \pi$ ou $p_i(u) = \pi+1$).*

Démonstration. Soit j un entier tel que $p_j(w) = \pi$ et $p_{j+1}(w) = \pi + 1$, par la pré-condition de la règle de changement de phase et le corollaire précédent, nous avons :

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

Pour les mêmes raisons, comme $p_i(v) = \pi$, il n'existe pas de noeud u tel que $p_i(u) > \pi + 1$. □

Remarque 4.4.1. Le même résultat peut être obtenu si nous n'avons qu'une borne du diamètre du graphe, la taille du graphe ou une borne de la taille du graphe.

4.5 Graphe avec un noeud distingué

Dans cette section, nous proposons une description sous forme de système de réécriture de graphe et une amélioration de l'algorithme *Synchroniseur* β [4]. L'approche de notre algorithme se base sur deux fameux paradigmes des calculs distribués. Le premier est le calcul d'un arbre recouvrant, tandis que le second est l'élection dans un arbre anonyme. Ces deux algorithmes sont écrits sous forme de calculs locaux et plus précisément sous forme de système de réécriture de graphe pour résoudre de manière optimale ces deux problèmes.
Notre protocole est composé de deux étapes. Dans la première, un arbre recouvrant T est calculé pour un graphe donné G. Le but de la seconde étape est d'élire un noeud de l'arbre T qui lancera une nouvelle phase du protocole de synchronisation. Ensuite, l'algorithme entre dans un nouveau cycle en calculant un nouvel arbre recouvrant de G. L'exécution des deux étapes n'est pas forcement séquentielle. Nous pouvons avoir le cas où dans une région du graphe l'arbre est en train d'être construit et dans une autre région, l'élection est exécuté simultanément. Avant de présenter notre protocole, nous proposons quelques définitions et formalismes utilisés par la suite.

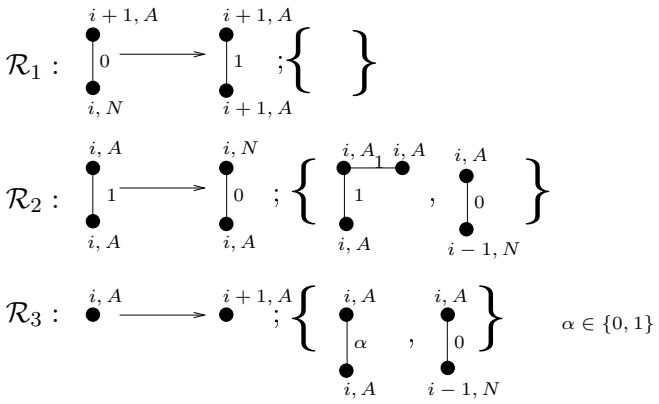
Définition 4.5.1. L'étiquette de chaque noeud est un couple composé d'un entier appelé phase, pulsation ou tour et d'un état.

- Un noeud v est étiqueté N si le deuxième terme de son étiquette est N.
- De même, un noeud v est dit étiqueté A si son état est A.
- Une arête e est étiqueté i si son état est i.
- Un noeud v est appelé *feuille* si ce noeud a exactement un seul voisin étiqueté A via une arête étiqueté 1.
- Un sous graphe G' est appelé étiqueté N (resp. étiqueté A) si tous ses sommets sont étiquetés N (resp. étiqueté A).

Soit G un graphe avec un noeud distingué u. Initialement, toutes les arêtes sont étiquetées 0, le noeud u a l'étiquette $(1, A)$ et tous les autres sommets ont l'étiquette $(0, N)$. Le système de réécriture de graphe correspondant à notre protocole est donné par : $\mathcal{R}_{15} = (L, I, P)$ où L, I et P sont définies comme suit :
$L = \{[0..\infty] \times \{A, N\} \cup \{0, 1\}\}$, $I = \{\{(1, A), (0, N)\} \cup \{0\}\}$, et $P = \{R_1, R_2, R_3\}$. R_1, R_2 et R_3 sont les règles de réécriture de graphe suivantes.

La règle R_1 calcule un arbre recouvrant T qui ne contient que les noeuds étiquetés A, tandis que les règles R_2 et R_3 exécutent l'algorithme d'élection dans T.

Soit $G(V, E, \lambda)$ un graphe connexe et étiqueté tel qu'il existe un noeud distingué v qui est étiqueté $(1, A)$, tous les autres noeuds sont étiquetés $(0, N)$ et toutes les arêtes sont



dans l'état 0. Soit $G'(V, E, \lambda')$ un graphe tel que : $G(V, E, \lambda) \xrightarrow[\mathcal{R}_{15}]{*} G'(V, E, \lambda')$.

Le graphe $G'(V, E, \lambda')$ satisfait les invariants suivants :

Propriété 1 : Il existe au moins un noeud étiqueté A dans G'.

Propriété 2 : Toutes les arêtes incidentes à un noeud étiqueté N sont étiquetées 0.

Propriété 3 : Un noeud v étiqueté N n'a pas de voisin u tel que $p(u) < p(v)$.

Propriété 4 : Le sous-graphe G'_T induit par les sommets étiquetés A est connexe par toutes les arêtes étiquetées 1 et tous les noeuds de G_T sont dans la même phase P_A.

Propriété 5 : Le sous-graphe induit par les arêtes étiquetées 1 est un arbre.

Le reste de cette section sera consacré à la preuve de la correction de notre nouvelle méthode de synchronisation. Par conséquent, nous devons montrer la validité des propriétés ci-dessus dans l'espoir de simplifier la compréhension des preuves que nous énoncerons plus tard.

Démonstration. (**Propriété 1**)
La validité de cette propriété dépend entièrement de la règle R_2. La règle R_1 crée des noeuds étiquetés A et la règle R_3 conserve l'état des noeuds et ne modifie que la pulsation d'un noeud. Par contre la règle R_2 change l'état d'un noeud étiqueté A par l'état N. Or la règle R_2 ne peut être appliquée que sur deux sommets u et v, tous deux étiquetés A, et ne change qu'un seul noeud. Donc à chaque application de cette règle un noeud étiqueté A disparaît. Quand il n'existe qu'un seul noeud étiqueté A, la règle R_2 ne peut plus être appliqué pour les raisons que nous avons déjà mentionnées, et comme les deux autre règles ne diminuent pas le nombre de noeuds étiquetés A, nous pouvons conclure que la propriété est vraie. □

Démonstration. (**Propriété 2**)
Initialement, toutes les arêtes sont étiquetées 0 donc la propriété est vraie. Supposons que c'est vrai à l'étape i, nous allons montrer que ça reste toujours vrai à l'étape $i+1$. A l'étape i, toutes les arêtes incidentes aux sommets étiquetés N sont étiquetées 0. La règle R_3 ne modifie pas l'état des arêtes, donc la propriété reste vraie si nous appliquons cette règle. Si la règle R_1 est appliquée sur le noeud u étiqueté N et le noeud v étiqueté A, l'arête incidente aux deux noeuds sera étiquetée en 1 mais le noeud u sera étiqueté A. Par conséquent, une arête étiquetée 1 reliera deux noeuds étiquetés A. De la même façon, si la règle R_2 est appliquée sur u, les contextes interdits impliquent que u n'a pas un voisin étiqueté A relié par une arête étiquetée 1. Donc, tous les voisins de u sont étiquetés N ou A. Or, pour le premier cas, ces noeuds n'ont pas d'arêtes incidentes étiquetées 1, d'après l'hypothèse de récurrence. Dans le cas où les voisins sont étiquetés A, les arêtes incidentes à ces noeuds et à u sont étiquetées 0, suivant les contextes interdits, d'où les arêtes incidentes à u sont étiquetées 0 à l'étape $i+1$. Ce qui valide notre propriété. □

Démonstration. (**Propriété 3**)
Nous allons démontrer cette propriété par induction. Initialement la propriété est juste car tous les noeuds sont etiquetés $(N, 0)$ sauf un seul etiqueté $(A, 1)$. Nous supposons qu'à l'étape i la propriété est toujours vraie. L'exécution de la règle R_1 à l'étape $i+1$, va changer l'état d'un noeud u étiqueté N vers un noeud étiqueté A et sa pulsation va augmenter. En utilisant l'hypothèse d'induction sur u, tous les voisins étiquetés N de u ont la pulsation égale à $p_i(u)$ or $p_{i+1}(u) > p_i(u)$ donc la nouvelle pulsation est supérieure ou égale à celles des voisins de u. La règle R_2 ne change pas la phase des noeuds, donc la propriété reste toujours juste. Enfin la règle R_3 augmente la phase d'un noeud étiqueté A donc la propriété reste toujours vraie. □

Démonstration. (**Propriété 4**)
Initialement, toutes les arêtes sont étiqueté 0 et un seul noeud est étiqueté A donc la propriété est vraie. Supposons que c'est vrai à l'étape i, nous allons montrer que ça reste toujours vrai à l'étape $i+1$. Si nous appliquons la règle R_1 à un noeud u étiqueté N, ce noeud devient étiqueté A et relié avec un autre noeud étiqueté A par une arête étiquetée 1. Or, par hypothèse de récurrence, à l'étape i, tous les noeuds étiquetés A sont connectés par des arêtes étiquetées 1, ce qui restera vrai car u sera relié à un noeud étiqueté A, qui appartient à G'_T, par une arête étiqueté 1. Donc u appartient à G'_T. De plus l'application de la règle R_1 change la pulsation de u à celle de son voisin étiqueté A, donc $p(u) = P_A$.
Si nous appliquons la règle R_2 sur un noeud u, G'_T reste toujours connexe car les contextes interdits de R_2 impliquent que u n'a qu'un seul voisin étiqueté A donc la suppression de u de G'_T ne va pas changer sa connexité. Si la règle R_3 est appliquée sur le noeud u, il n'y a aucun ajout ou suppression de noeud étiqueté A, donc G'_T reste connexe. Or les contextes interdits impliquent qu'il n'y a pas de noeud étiqueté A voisin de u donc G'_T n'est composé

que d'un seul noeud u et quand la pulsation de u change, P_A change.
D'après la propriété 2, toutes les arêtes incidentes aux noeuds étiquetées N sont étiquetés 0, donc nous pouvons conclure que toutes les arêtes étiquetées 1 sont incidentes aux noeuds étiquetés A, d'où, toutes ces arêtes appartiennent à G'_T. □

Démonstration. (**Propriété 5**)
Soit T le graphe formé par toutes les arêtes étiquetées 1. Initialement, la propriété est vraie puisque toutes les arêtes sont étiquetées 0. Supposons que la propriété est vraie à l'étape i. Si la règle R_3 est appliquée à l'étape $i+1$ la propriété reste vraie car aucun changement ne sera fait. Si la règle R_1 est appliquée au noeud u étiqueté N, ce noeud change d'état et devient étiqueté A. Or d'après la propriété 2, u n'a pas d'arête incidente étiquetée 1, donc u n'appartenait pas à T et le graphe $T \cup \{u\}$ ne contient pas de cycle. Si nous appliquons la règle R_2 à v, les contextes interdits et la propriété 2 impliquent que v n'a qu'une seule arête étiquetée 1, donc u est une feuille de l'arbre T. Conclusion, la propriété est prouvée. □

Théorème 4.5.1. *Il existe un temps $t_0 > 0$, pendant lequel tous les noeuds étiquetés N sont dans la même phase $P_N > 0$.*

Démonstration. Soit v_i le noeud élu au début de la phase i ($p(v_i) = i$). Soit t_{i-1} le temps avant l'exécution de la règle R_3 sur v_i, donc tous les noeuds sont dans la phase $p(v_i)$. A ce moment, il n'y a qu'un seul sommet étiqueté A dans tout le graphe (d'après les contextes interdits et la propriété 4), et tous les noeuds étiquetés N sont dans la même phase $p(v_i)-1$ puisque d'après la propriété 3, tous les voisins d'un noeud étiqueté N sont dans la même phase ou sont dans une phase supérieure. Donc tous les noeuds voisins du noeud v sont dans la phase $p(v_i) - 1$, et comme leurs voisins exceptés v sont étiquetés N, alors ils sont dans la même phase (propriété 3). Ainsi, on pourra prendre $t_0 = t_{i-1}$. □

Nous allons maintenant prouver la validité du Théorème 4.2.2 (*théorème de convergence*).

Démonstration. (**Théorème de Convergence**)
Soit t_0 et v_i comme défini dans le théorème 4.5.1. Soit $u \neq v_i$ le voisin de v_i tel que u est le dernier noeud étiqueté A avant l'application de la règle R_2 sur u et v_i. Il est évident que u est un voisin de v_i. Soit t_u le temps quand cette règle est exécutée sur u. Nous savons que $p(u) = p(v_i)$ au temps t_u. Ainsi, tous les sommets étiqueté N de G sont dans la même phase que $p(u) = p(v_i)$ dans le temps t_u. Ce qui implique que tous les noeuds sont dans la même phase P_u à ce temps. □

Nous allons maintenant démontrer et prouver la validité du Théorème 4.2.1 (*compatibilité de phase*).

Démonstration. (**Compatibilité de phase**)
Il n'y a que deux possibilités pour qu'un noeud v change de phase. La première est par l'exécution de la règle R_3. Cette règle assure que v est étiqueté A. En raison des propriétés 3 et 4, nous avons la relation $p(v) \geqslant P_N$. Où P_N est la pulsation de tous les noeuds étiquetés N du graphe G.
La deuxième possibilité pour le noeud v pour changer de pulsation, est l'exécution de la règle R_1. Cette règle est toujours exécutée après que le noeud u ait démarré la phase $i+1$. Au début de cette phase, tous les noeuds étiquetés N sont dans la phase i (*voir Théorème 4.5.1*). Avec la *propriété 4*, nous savons que tous les noeuds étiquetés A sont dans la phase $i+1$ et que tous les noeuds étiquetés N sont au moins à la phase i. Ainsi, le noeud v sait qu'il n'existe aucun noeud w dans G tel que $p(w) < p(v)$. □

Remarque 4.5.1. Le synchroniseur β [72] est un cas particulier de notre synchroniseur. Tandis que dans notre synchroniseur la racine n'est pas connue à l'avance (sauf au début) et elle est déterminée par un algorithme d'élection, le synchroniseur β nécessite la création d'un arbre recouvrant ayant un noeud distingué (la racine de l'arbre) avant son exécution. Avant de passer à la phase suivante, la racine attend les messages de ses fils que le "sous-arbre est sûr", qui eux aussi doivent attendent à leur round le même message venant de leur fils et ainsi de suite. En d'autres termes, la racine s'assure que tous les sommets de l'arbre recouvrant sont sûrs avant d'aller à la phase suivante ce qui augmente le temps d'attente et de passage à la pulsation suivante.

4.6 Arbres

Les protocoles de synchronisation développés jusqu'ici sont consacrés à tout type de graphe. Ces protocoles ont, néanmoins, besoin d'avoir de connaissance proportionnée de certaines caractéristiques de graphe pour être exacte et sans faute. Nous introduisons une nouvelle méthode dédiée au arbres. Bien que cette méthode soit limitée aux arbres, elle a l'avantage de ne pas avoir besoin de plus de connaissance (taille, diamètre ou existence d'un noeud distingué).
L'idée principale de ce protocole réside sur l'utilisation d'un algorithme d'élection pour décider quel noeud va commencer le nouveau round. L'algorithme d'élection [10] que nous utilisons est le même que dans la section 4.5. Ce problème résout le problème d'élection dans les arbres anonymes utilisant les systèmes de réécriture de graphe. Au début du protocole de synchronisation, tous les noeuds sont dans la même pulsation. Leurs étiquettes possèdent deux états. Le premier est la valeur de la pulsation et le second est utilisé pour l'algorithme d'élection. Nous utilisons l'avantage de l'élection pour choisir un sommet u qui commence une nouvelle pulsation. En outre, tous les noeuds v, qui ont un voisin w tel que $p(w) > p(v)$, augmentent leur pulsation. Quand un noeud incrémente

son round, son étiquette d'*élection* est remise à sa valeur initiale. Une description formelle de ce protocole de synchronisation est donnée ci-dessous.

Soit T un arbre. Initialement, tous les sommets sont étiquetés $(1, N)$. Le système de réécriture du graphe est $\mathcal{R}_{16} = (L_{\mathcal{R}}, I_{\mathcal{R}}, P_{\mathcal{R}})$ définie par $L_{\mathcal{R}} = \{[1..\infty] \times \{L, N\}\}$, $I_{\mathcal{R}} = \{(1, N)\}$, $P_{\mathcal{R}} = \{R_1, R_2, R_3\}$ où R_1, R_2 et R_3 sont les règles de réécriture suivantes :

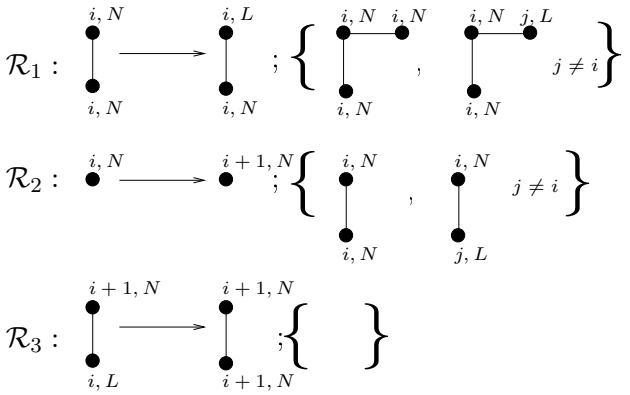


FIG. 10 – Protocole de Synchronisation pour les arbres anonymes.

Définition 4.6.1.

- Un noeud v est étiqueté N si la deuxième valeur de son étiquette est égale à N.
- Un noeud v est étiqueté L si la deuxième valeur de son étiquette est L.
- Un noeud *feuille* v est un noeud qui a exactement un voisin avec l'étiquette N.
- Un sous-graphe T_i est appelé étiqueté N (resp. étiqueté L) si tous ses noeuds sont étiqueté N (resp. étiqueté L).

Nous commençons par énoncer des invariants qui vont nous servir pour démontrer le lemme et le théorème qui suivront.
Soit $T(V, E, \lambda)$ un arbre tel que tous ses noeuds sont étiquetés $(N, 1)$, et soit $T'(V, E, \lambda')$ un arbre tel que : $T(V, E, \lambda) \xrightarrow[\mathcal{R}_{16}]{*} T'(V, E, \lambda')$.
L'arbre $T'(V, E, \lambda')$ satisfait les propriétés suivantes :

Propriété 1 : le sous-graphe induit par tous les sommets étiqueté N est connexe.

Propriété 2 : un noeud v étiqueté L a au plus un voisin w tel que w est étiqueté N et $p(w) \geqslant p(v)$.

Propriété 3 : un noeud v étiqueté L n'a aucun voisin w tel que w est étiqueté N et $p(w) < p(v)$.

Propriété 4 : tous les noeuds étiqueté N ont la même pulsation P_N.

Propriété 5 : tous les sommets étiqueté L formant un sous-arbre T_i ont la même pulsation P_{T_i}.

Démonstration. (**Propriété 1**)
Initialement, il n'y a que des noeuds étiquetés N et comme le graphe est connexe la propriété est vraie. Supposons qu'après la $i$ème étape, le graphe G_N^i formé par les sommets étiqueté N soit connexe, montrons qu'ils le restent à l'étape $i+1$. Si nous appliquons la règle R_1 sur le noeud u, les contextes interdits nous assurent que u ne possède qu'un seul voisin v étiqueté N. Donc u est un noeud feuille du graphe G_N^i, le graphe G_N^{i+1} reste connexe.
Si la règle R_2 est appliquée, aucun noeud étiqueté N ne change d'état, donc le graphe G_N^{i+1} est connexe selon l'hypothèse de récurrence. Enfin si nous appliquons la règle R_3 sur u, nous ajoutons ce noeud au graphe G_N^i ($G_N^{i+1} = G_N^{i+1} \cup u$). G_N^{i+1} reste connexe puisque u est voisin d'un noeud étiqueté N. □

Démonstration. (**Propriété 2**)
Nous commençons par montrer que les noeuds étiquetés L ont au plus un voisin étiqueté N.
Initialement, tous les noeuds sont étiquetés N. Supposons que la propriété est juste jusqu'à l'étape i, montrons qu'elle est toujours vraie à l'étape $i+1$. Si la règle R_2 est appliquée, la propriété reste vraie car aucun noeud étiqueté L n'est créé. Si à l'étape $i+1$, la règle R_1 est appliquée, soit v le noeud étiqueté N à l'étape i tel que à l'étape $i+1$, v devient étiqueté L. Pour pouvoir appliquer la règle R_1 sur v, les contextes interdits assurent que v a au plus un noeud w étiqueté N. Donc u n'a au maximum qu'un seul voisin étiqueté N. Si la règle R_3 est appliquée sur le noeud w, d'après la propriété 1, w ne possède qu'un seul voisin étiqueté N. D'où, les noeuds étiquetés L ont au plus un voisin étiqueté N.
Maintenant, nous allons, de la même manière, prouver $p(w) \geqslant p(v)$ avec v est un noeud étiqueté L possédant un voisin w étiqueté N à l'étape $i+1$. Si la règle R_1 est appliquée au noeud u, d'après la possibilité d'application de la règle et ses contextes interdits, $p(v)$ doit être égale à $p(w)$. Donc à l'étape $i+1$, $p(w) = p(v)$. Pour que la règle R_2 soit appliquée sur w, il faut que tous ses voisins soient dans la même pulsation ($p(w)$). Donc à l'étape $i+1$, pour tout noeud v étiqueté L voisin de w, $p(w) > p(v)$. Enfin, si la règle R_3 est appliquée sur le noeud w, d'après l'hypothèse de récurrence, quelques soit le voisin w étiqueté N d'un noeud v étiqueté L a $p(w) \geqslant p(v)$. Donc à l'étape $i+1$, chaque noeud v étiqueté L voisin du noeud w ont $p(w) \geqslant p(v)$. D'où la propriété reste toujours vraie. □

Démonstration. (**Propriété 3**)
Cette propriété est une conséquence de la propriété précédente. □

Démonstration. (**Propriété 4**)
Soit T_N le sous arbre induit par tous les sommets étiqueté N. Initialement, tous les sommets sont dans la pulsation 1 et sont étiquetés N. Si nous appliquons la règle R_1, un noeud s'élimine de T_N et aucune pulsation de noeud ne change. Si la règle R_2 est appliquée, un noeud v étiqueté N va changer de pulsation, or d'après la propriété 1 et les contextes interdits de cette règle, T_N ne se compose que du noeud v. Enfin si la règle R_3 est appliquée, un nouveau noeud v étiqueté N sera ajouté au graphe T_N avec comme pulsation la pulsation de son noeud voisin w étiqueté N. D'après l'hypothèse de récurrence, la pulsation de w est égale à P_N d'où $p(v) = p(w) = P_N$. Donc, la propriété est toujours vraie. □

Démonstration. (**Propriété 5**)
Supposons que dans un sous arbre T_i composé de sommets étiquetés L, tous les sommets ont la pulsation égale à i sauf un sommet u qui est dans la pulsation j. Seule la règle R_1 crée des noeuds étiquetés L. Supposons que la règle R_1 est appliquée à u à l'étape k, pour pouvoir appliquer cette règle à u, u doit avoir un seul voisin étiqueté N et ne possède pas de voisin étiqueté L qui ont une pulsation inférieur à $p(u)$. D'où j ne peut pas être diffèrent à i. Ce qui contredit notre hypothèse. Donc, dans le sous arbre T_i composé de sommets étiquetés L, tous les sommets sont dans la même pulsation P_{T_i}. □

Le théorème suivant est aussi assez important pour la compréhension et la validité de notre protocole de synchronisation.

Théorème 4.6.1. *Il existe une étape t_0, dans laquelle tous les sommets sont étiquetés L sauf un noeud étiqueté N et tous les noeuds sont dans la même pulsation P_N.*

Démonstration. Initialement, il n'y a que la règle R_1 qui peut être appliquée car tous les sommets sont étiquetés N et sont dans la même phase 1. Comme le graphe est un arbre, la règle appliquée enlève à chaque fois un noeud étiqueté N et d'après la propriété 1, à une étape t il ne restera qu'un seul sommet étiqueté N. Nous pouvons définir t_0 égale à t.
Nous allons montrer maintenant que l'étape t_0 n'est pas unique. Après l'étape t, la seule règle applicable est R_2 et ensuite la règle R_3 est appliquée jusqu'à ce que tous les noeuds étiquetés L disparaissent et d'après la propriété 4 nous recommencerons le même cycle avec une pulsation égale à P_N et la prochaine étape t_1 sera supérieur à t_0 □

Remarque 4.6.1. *Si pour chaque étape, un seul changement est effectué, t_0 sera égale à $n-1$, avec n est le nombre de sommet de l'arbre. L'étape t_1 sera égale à $3n-2$.*

Nous allons, maintenant, démontrer la preuve des deux algorithmes (4.2.2, 4.2.1) que nous avions annoncé précédemment.

Démonstration. (**Théorème de convergence**)
Ce théorème est un corollaire du théorème précèdent 4.6.1. □

Démonstration. (**compatibilité de phase**)
C'est aussi un corollaire du théorème 4.6.1. Puisqu'à l'étape t_0 tous les noeuds sont étiquetés L et un seul est étiqueté N et ils sont dans la même phase. À l'étape $t_0 + 1$ seule la règle R_2 est applicable et elle permet de passer à la phase suivante. □

4.7 Généralités

Dans les sections précédentes, nous avons proposés quelques algorithmes qui permettent une *synchronisation globale* des systèmes asynchrones. Une similitude avec notre problème de synchroniseur et le problème de terminaison nous permet de donner ce lemme suivant :

Lemme 4.7.1. *Tous les algorithmes qui résolvent le problème de terminaison peuvent être utilisés pour résoudre le problème de la synchronisation globale.*

Démonstration. Le problème de synchronisation globale est de savoir si tous les noeuds sont dans la même phase pour pouvoir passer à la phase suivante. Le problème revient à ce qu'un noeud u détecte si tous les noeuds sont dans la même phase ou dans une phase supérieure à la sienne. u peut savoir si localement il est dans le même round d'où l'utilisation d'algorithme pour transformer la terminaison implicite en terminaison explicite. Un noeud qui passe d'une phase à une autre met sa variable de terminaison à faux. Une fois que tous ses voisins sont dans la même phase que lui, il remet cette variable à vrai. C'est la que le détecteur de terminaison intervient pour vérifier si toutes les variables des noeuds du graphe sont à vrai. Si c'est le cas, au moins un noeud du graphe augmente sa phase et informe les autres noeuds. □

D'après le théorème 3.3.3, nous avons une caractérisation des graphes sur lesquels on peut exécuter un *synchroniseur parfait*.

Corollaire 4.7.1. *Nous présentons les caractéristiques des graphes pour lesquels nous pouvons trouver un algorithme de synchroniseur parfait :*

- *graphes ayant un leader,*
- *graphes avec des identités,*
- *graphes dont on connaît la taille ou une borne du diamètre,*
- *arbres,*
- *graphes triangulés,*
- *graphes ayant une borne du temps d'envoi et de réception d'un message (réseaux ABD).*

Un algorithme pour la synchronisation des réseaux ABD est donnée par Tel et al dans [39]. Pour les autres types de graphe, nous pouvons utiliser le même principe que pour les algorithmes présentés auparavant. Pour le graphe avec identité, nous pouvons appliquer l'algorithme de calcul d'arbre recouvrant avec identité $\mathcal{R}_8$ puis exécuter l'algorithme d'élection pour les arbres.

4.8 Constructions des synchroniseurs

Le but de cette section est de donner une méthodologie, celle-ci devrait transformer les protocoles présentés dans les sections précédentes dans des synchroniseurs opérationnels.

Tous les protocoles développés dans ce chapitre supposent l'existence d'un *générateur de pulsation* pour chaque noeud du réseau. Ce qui signifie qu'un noeud v possède une variable qui calcule la pulsation $p(v)$, et nous supposons que le noeud peut générer une séquence locale de l'*horloge des pulsations* en incrémentant les valeurs de la variable $p(v)$ de temps en temps (i.e. $p(v) = 0, 1, 2, ...$). Ces pulsations sont censées simuler les tours d'une horloge globale dans un système synchrone. Évidemment, l'utilisation de ces protocoles en tant qu'applications autonomes ne donnera rien dans un environnement asynchrone. Pour cette raison, nous introduisons quelques définitions qui devraient nous aider à assurer des garanties sur la relation entre les valeurs des pulsations des noeuds voisins à divers moments de l'exécution.

Définition 4.8.1. Nous notons par $t(v, p)$ le temps réel (global) pour lequel le noeud v a augmenté sa pulsation à p. Nous disons que le noeud v est *au round* $p(v) = p$ (ou à la pulsation p) pendant l'intervalle de temps $\tau(v, p) =]t(v, p), t(v, p + 1)]$.

Le fait que nous travaillons sur un réseau totalement asynchrone, nous ne pouvons pas forcer les sommets à avoir la même pulsation à tout moment. Cependant, nous savons qu'il est possible de garantir une forme plus faible de compatibilité entre les pulsations des noeuds voisins dans le réseau. Cette forme de compatibilité est énoncée dans la définition 4.8.2.

Définition 4.8.2 (compatibilité de pulsation). Si un noeud v envoie un message original m à un noeud voisin w durant sa pulsation $p(v) = p$, alors le message m est reçu par le noeud w durant au mieux la pulsation $p(w) = p$.

4.8.1 Une méthodologie pour construire un synchroniseur

Pour construire un synchroniseur pour chacun de nos protocoles, nous devons changer leurs caractéristiques tels que, soit Π_S un algorithme écrit pour un réseau synchrone et soit ν un protocole, il devrait être possible de *combiner* Π_S sur ν pour obtenir un nouveau protocole $\Pi_A = \nu(\Pi_S)$ qui permet de l'exécuter dans un réseau asynchrone. Π_A a deux composantes : la composante *originale* et la composante *synchronisation*. Chacune de ces composantes possèdent leurs propres variables locales et leurs types de messages et cela pour tous les noeuds. La composante *originale* comprend les variables locales et les messages du protocole original Π_S, tandis que la composante *synchronisation* comprend les variables locales de la synchronisation et ses messages.
Les modifications de nos protocoles seront faites en deux étapes. La première étape affecte l'étiquette attachée à chaque sommet. Effectivement, nous ajoutons trois nouveaux états à chaque étiquette. Le premier état est une variable *booléenne* S. Cette variable décide quelle composante (*composante originale* ou *composante synchronisation*) est *active* sur chaque noeud. Seules les règles de la composante *active* peuvent être appliquées sur un noeud v. Les autres états sont deux tampons B_p et B_{p-1} qui représentent les messages qu'un noeud v, dans la pulsation $p(v)$, doit envoyer respectivement dans les phases $p(v)$ et $p(v)-1$. Nous supposons qu'un protocole μ, qui est le produit des modifications ci-dessus, satisfait tous les caractéristiques d'un synchroniseur. D'une manière générale, un protocole $\Pi_A = \mu(\Pi_S)$ fonctionnera en deux phases.
Initialement, l'état S_v est mis à *true* (vrai) pour tous les noeuds v et donc la *composante synchronisation* est active (première phase). Dès qu'un noeud v incrémente sa pulsation $p(v)$, sa variable S_v est mise à $false$ (faux). Par conséquent, seule les règles de Π_S peuvent être appliquées sur v. La *composante originale* est active (deuxième phase). Chaque application des règles de Π_S remet la valeur de S_v à *true*. Ainsi, la composante synchronisation peut commencer à synchroniser v avec tous les autres noeuds. Ce cycle est exécuté jusqu'à ce que l'algorithme Π_S arrête Π_A avec n'importe quelle demande spécifique.

4.8.2 Une vue générale

Les deux phases représentées dans la section précédente sont récapitulées dans la figure 11. La règle R_1 représente la première phase et la règle R_2 représente la deuxième phase. Δ décrit l'ensemble de règles utilisées dans le protocole de synchronisation (ν), et Ω est l'ensemble des règles de réécriture de graphe utilisées dans l'algorithme Π_S. La première règle signifie qu'aussi longtemps que $S_v = True$, il n'est permis que de synchroniser v avec tous les noeuds du réseau. Dès que le noeud v change sa pulsation, la deuxième

phase commence pour lui. Dans cette phase, il n'est plus permis de synchroniser mais il est permit d'exécuter les règles de l'algorithme Π_S. Après l'exécution de Π_S, la première règle est activée de nouveau. Ainsi le cycle continue. Tous nos synchroniseurs vont utiliser la règle qui joue le rôle de R_2. Cette règle devrait assurer une bonne interaction entre les deux protocoles ν et Π_S.

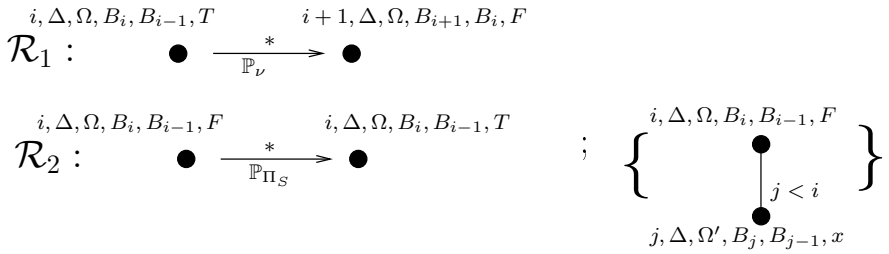


FIG. 11 – Une méthode générale pour construire un synchroniseur.

Maintenant, nous donnons un exemple pour illustrer la réalisation de la construction décrite d'une manière réaliste.

Exemple 4.8.1 (Synchroniseur avec SSP). Soit G un graphe avec diamètre D. Considérons la fonction d'étiquetage λ, où $\lambda^{(V)} : V \to [1..\infty] \times [-1..d+1] \times \mathcal{M} \times \mathcal{M} \times \{T, F\} \cup \mathbb{L}(\Pi_S)$. Initialement, tous les sommets sont étiquetés $(1, -1, B_1, B_0, F, \mathbb{I}_{\Pi_S})$. Le système de réécriture de graphe est $\mathcal{R}_{17} = (L, I, P)$ défini par $L = \{[1..\infty] \times [-1..d+1] \times \mathcal{M} \times \mathcal{M} \times \{T, F\}\} \cup \mathbb{L}(\Pi_S)$, $I = \{(1, -1, B_1, B_0, F, \mathbb{I}_{\Pi_S})\}$, $P = \{R_1, R_2, R_3, R_4\} \cup \mathbb{P}_{\Pi_S}$ où $\mathbb{L}(\Pi_S)$ représente le type des états utilisé dans Π_S, $\mathbb{I}_{\Pi_S}$ est l'ensemble des états initiaux dans Π_S, $\mathbb{P}_{\Pi_S}$ est l'ensemble des règles utilisées dans Π_S. R_1, R_2, R_3 et R_4 sont les règles de réécriture de graphe avec contextes interdits données dans la figure 12. Les caractéristiques et les propriétés de l'algorithme fondamental de ce protocole sont discutées dans la section 4.4. La règle R_4 est d'une importance extrême dans ce protocole. Elle joue le rôle consacré à R_2 dans la figure 11. Il permet l'application d'exactement une action de l'algorithme Π_S sur un noeud v. Après une telle action, le noeud v entre dans la phase de synchronisation. La règle R_4 peut être généralisée à tous les synchroniseurs que nous pourrions construire en utilisant les protocoles présentés dans les sections précédentes.

La validité de nos synchroniseurs

Nous présentons maintenant quelques propriétés significatives qui peuvent toutes être garanties par les synchroniseurs que nous construisons.

Définition 4.8.3 (Propriété des sommets prêts). un noeud v est *prêt* pour la pulsation p, noté $Ready(v, p)$, une fois qu'il a reçu tous les messages de l'algorithme envoyé à lui par ses voisins pendant leur pulsation $p - 1$.

Définition 4.8.4 (règle des sommets prêts (readiness)). Nous permettons à un noeud v de générer sa pulsation p une fois qu'il en a fini avec ses actions originales exigées pour

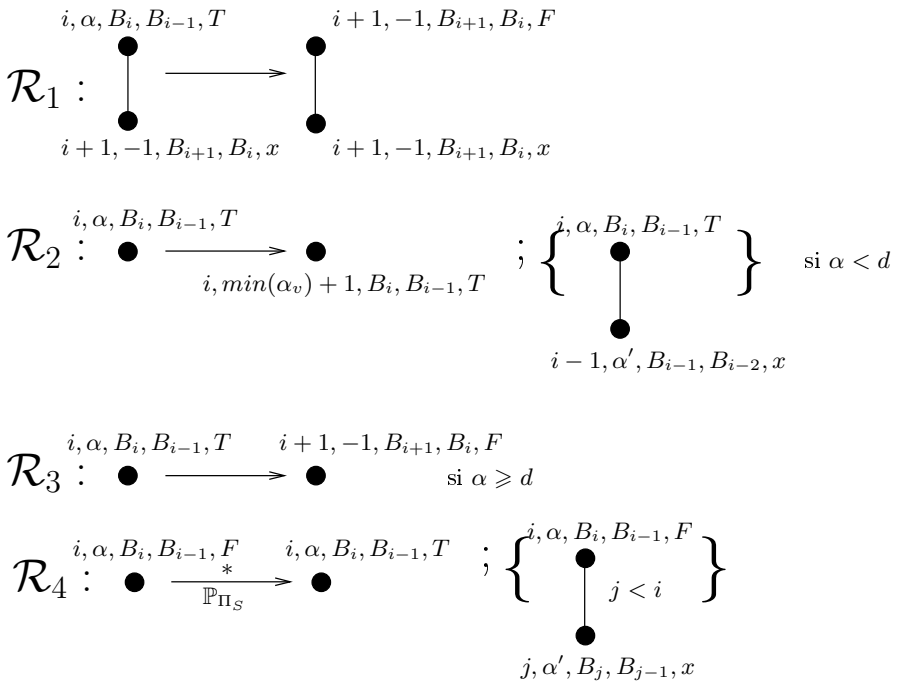


Fig. 12 – Le Protocole Π_A utilisant SSP.

la pulsation $p-1$ et qu'il est $Ready(v,p)$.

Affirmation 4.8.1. *Soit μ un synchroniseur construit comme nous l'avons décrit dans la section 4.8.1. μ satisfait toujours la règle des sommets prêts.*

Démonstration. La preuve est immédiatement déduite par l'utilisation de la règle R_2(*voir Figure 11*). Nous permettons à un noeud v de générer sa pulsation p, si et seulement si $S_v = T$ et v satisfait les conditions requises du *protocole de synchronisation*. La seule possibilité pour S_v de devenir $True$ (vrai) est l'exécution de la règle R_2. Ainsi, v a exécuté son action Π_S à la pulsation $p-1$ et il est $Ready(v,p)$. □

Définition 4.8.5 (règle de délai). Si un noeud v reçoit, dans la pulsation p, un message envoyé par son voisin w pendant la pulsation $p' > p$ de w, alors le noeud v range ce message dans un buffer. Ce message ne sera traité qu'une fois que v aura généré la pulsation p'.

Affirmation 4.8.2. *Soit μ un synchroniseur construit comme nous l'avons décrit dans la section 4.8.1. μ satisfait toujours la règle de délai.*

Démonstration. Soit v un noeud qui a reçu, à la pulsation p, un message m' envoyé par son voisin w pendant la pulsation $p' > p$. Tous nos protocoles de synchronisation garantissent qu'après un nombre fini d'étapes, v et w seront dans la même pulsation p'. D'autre part, le noeud v peut seulement utiliser les messages contenus dans le buffer B_p^w. Un tel buffer existe toujours. En effet, la différence de pulsation entre deux noeuds dans tout le réseau

est au maximum de 1. Ce qui signifie que le noeud v sera capable d'utiliser le message m' dès que $p(v) = p'$. □

Lemme 4.8.1. *[67] Un synchroniseur, imposant la règle des sommets prêts (readiness) et la règle de délai, garantit la compatibilité de pulsation.*

Le lemme ci-dessus énonce facilement les raisons pour lesquelles tous nos synchroniseurs garantissent le principe de la compatibilité de pulsation présenté dans la définition 4.8.2. Peleg a démontré en 2000 un rapport essentiel entre le concept de la compatibilité de pulsation et la validité du synchroniseur. Une des parties intéressantes de son travail a été énoncée en lemme ci-dessous.

Lemme 4.8.2. *[67] Si un synchroniseur μ garanti la compatibilité de pulsation, alors il est correct.*

Lemme 4.8.3. *Soit μ un synchroniseur construit comme nous l'avons décrit dans la section 4.8.1. μ est correct.*

Démonstration. La preuve est déduite des affirmations 4.8.1, 4.8.2 et des lemmes 4.8.2 et 4.8.1. □

Chapitre 5

Visidia

L'implémentation, le débogage, le test et l'expérimentation des algorithmes distribués sont des tâches assez complexes et délicates, associées à de nombreux pièges et difficultés. Dans ce contexte, il est essentiel de comprendre les idées algorithmiques de haut niveau, indépendamment du langage et de la plate-forme de l'implémentation. La visualisation de l'exécution d'un algorithme distribué fournit une abstraction permettant de mieux comprendre son comportement et donc de simplifier sa mise au point et sa preuve. La perception des réseaux et des événements obtenus à partir de la visualisation graphique est plus simple pour les humains et donne plus d'informations qu'une description textuelle. ViSiDiA [9, 8, 10, 61] (Visualization and Simulation of Distributed Algorithms) est développé dans le but de tester, vérifier et visualiser l'exécution des algorithmes distribués écrit sous forme de calculs locaux.

5.1 Architecture de l'outil logiciel Visidia

Le but de Visidia est de simplifier l'implémentation, et de tester les algorithmes distribués codés sous forme de calcul local et spécialement ceux décrits sous forme de règles de réécriture. L'environnement de Visidia est décrit par un système asynchrone (pas d'horloge globale, chaque processeur a sa propre horloge) et anonyme (les processeurs n'ont pas d'identité). En plus, il doit être ouvert, c'est à dire, si une information est requise, comme la taille du réseau ou l'identité, les processus peuvent accéder à l'information demandée. Vu que les algorithmes distribués sont difficiles à déboguer, une animation graphique de l'algorithme est essentielle. L'animation devrait s'exécuter en "temps réel", i.e. ce qu'on regarde ce n'est pas la trace de l'exécution mais ce sont les changements qui s'effectuent avec un petit décalage temporel par rapport à l'exécution réelle, mais qui respecte l'ordre des événements.

5.2 Conception

La conception de Visidia nous a permis de distinguer trois parties importantes : l'*Interface Graphique*, le *Simulateur* et les *Algorithmes distribués*. La figure 13 représente ces parties et les liens qui les unissent. Ces modules sont bien séparés, ce qui implique la possibilité de les modifier et de les développer de manière indépendante.

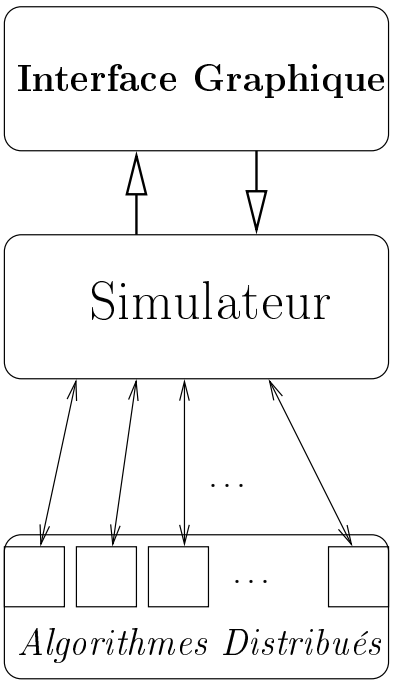


FIG. 13 – *Architectures de Visidia*

5.2.1 L'interface graphique (GUI)

L'interface graphique de Visidia est un environnement graphique qui permet à l'utilisateur de dessiner facilement un réseau et de visualiser l'exécution d'un algorithme distribué. C'est la partie la plus importante pour l'utilisateur car c'est via cette interface qu'il pourra interagir avec Visidia.
Pour faciliter la maintenance de Visidia, nous avons opté pour deux interfaces graphiques semi-indépendantes qui se partagent des informations. La première interface est celle qui permet la création, la modification, l'import du réseau. La deuxième concerne l'animation de l'exécution sur le réseau déjà choisi dans l'interface précédente. Dans cette interface, nous pouvons choisir ou créer un algorithme distribué. Elle permet aussi de visualiser son exécution, de faire des expérimentations...

La création d'un réseau

L'interface graphique permet à l'utilisateur de construire un réseau. Par une simple manipulation de la souris, l'utilisateur peut ajouter, supprimer, ou choisir des processeurs, des liens de communications ou des sous-réseaux. Le réseau ainsi créé doit pouvoir être sauvegardé pour une utilisation ultérieure. Il est également possible d'importer un autre format de fichier ou d'exporter d'autres formats, en particulier en GML[36]. Ceci permet d'utiliser d'autres éditeurs de graphes pour la génération de graphes qui peuvent être chargés dans Visidia.

La visualisation

Une fois que le réseau est dessiné, l'utilisateur doit pouvoir choisir un algorithme distribué ou bien en créer un sous forme de règles de réécriture. La simulation est alors exécutée. Durant l'exécution, les messages échangés entre les processeurs et leurs états sont affichés et les états des canaux de communications et des processeurs sont mis à jour en temps réel. Les états des processeurs peuvent être modifiés par l'utilisateur. L'utilisateur peut influer sur l'animation de l'exécution de l'algorithme. Enfin, il est possible d'expérimenter l'algorithme et cela sans même l'animer. L'expérimentation peut s'exécuter plusieurs fois sans tout relancer à la main.

5.2.2 Le simulateur

Le simulateur est le lien entre l'interface graphique et les algorithmes. Il modélise un réseau de processeurs asynchrones. Il crée des entités autonomes selon le réseau, les dote de plusieurs informations les concernant et leur attribue l'algorithme distribué à exécuter. Chaque entité communique seulement avec ses voisins par échange de message. Le simulateur doit contrôler les échanges de messages entre les processeurs, ainsi que la visualisation des événements. C'est lui qui communique à l'interface graphique les événements à visualiser, qui reçoit les changements des états des processeurs et qui les communique aux entités correspondantes. Le simulateur est donc le centre de Visidia. Il doit aussi assurer l'ordonnancement des événements exécutés par les processeurs et aussi la visualisation en temps réel de ces événements.

Les événements avec l'interface graphique

L'interface graphique gère les actions de l'utilisateur et l'animation de l'exécution des algorithmes distribués. Et comme les modules doivent être séparés et distincts, une communication entre eux est alors nécessaire. Il est évident que les évènements échangés ne doivent pas être nombreux et doivent être précis, cela permet de garantir une exécution à "temps réel" de l'algorithme.
Les événements principaux reçus de l'interface graphique sont :

- l'arrêt provisoire de la simulation ;
- l'arrêt définitif de la simulation ;
- le redémarrage de l'exécution ;
- le changement de l'état d'un processeur.

Les événements principaux envoyés à l'interface graphique sont :

- les échanges de message ;
- la modification de l'état d'un processeur ;
- le changement de l'état d'un canal de communication ;
- la fin de l'exécution de l'algorithme.

Les événements avec les algorithmes distribués

Les événements échangés entre le simulateur et les algorithmes distribués sont les mêmes que ceux utilisés entre le simulateur et l'interface graphique.
Les événements principaux reçus des algorithmes distribués sont :

- l'envoi de message ;
- la modification de l'état d'un noeud ;
- le changement de l'état d'un canal de communication ;
- l'achèvement de l'exécution de l'algorithme.

Les événements principaux envoyés aux algorithmes distribués sont :

- l'arrêt provisoire de la simulation ;
- l'arrêt définitif de la simulation ;
- le redémarrage de l'exécution ;
- le changement de l'état d'un processeur.

5.2.3 Types d'algorithmes distribués

Un algorithme distribué est un algorithme simple exécuté par plusieurs processus qui communiquent entre eux. La communication sert à échanger des informations et des résultats pour résoudre le même problème. Cette comunication se fait par échange de messages. Dans le cas des calculs locaux, un algorithme distribué a besoin de connaître le nombre de ses voisins et les canaux de communications. Une bibliothèque contenant les primitives suivantes est indispensable :

- `rendezVous` : une fonction qui permet de faire la synchronisation par rendez-vous.

- `LC1` : une fonction qui permet de faire la synchronisation en utilisant une élection locale de rayon 1.
- `LC2` : une fonction qui permet de faire une élection locale de rayon 2.
- `breakSynchronisation` : Arrête la synchronisation.
- `putState` :permet de changer l'état d'un processeur.
- `getArity` : retourne le degré d'un processeur, i.e. le nombre de ses voisins.
- `sizeOfTheGraph` : permet de renvoyer la taille totale du réseau pour les algorithmes qui ont besoin de cette connaissance.

Puisque les communications entre les processeurs sont basées sur des messages, les fonctions pour manipuler les messages sont fournies. Un message est programmé par une classe qui contient toutes les informations indispensables. Le programmeur peut manipuler un message par les méthodes suivantes :

- `sendTo` : envoie un message à un voisin.
- `receiveFrom` : (resp. `receive`) reçoit un message d'un voisin particulier (resp. le premier message dans la file d'attente du processeur).

Plusieurs algorithmes distribués classiques décrits par les calculs locaux sont implémentés. Citons par exemple les algorithmes suivants :

- élection dans les arbres, les graphes triangulés et graphes complets,
- Rendez-vous probabiliste et élections locale probabilistes,
- arbre recouvrant dans des réseaux anonymes,
- arbre recouvrant dans des réseaux avec identités,
- l'algorithme de Mazurkiewicz de numérotation et de reconstruction de graphe (universal graph reconstruction),
- détections de propriétés stables,
- l'algorithme de Chang-Robert,
- la 3-coloration d'un anneau,
- l'algorithme de Dijkstra-Scholten pour la détection de terminaison,

5.3 Réalisation-Implémentation

Visidia se compose de trois grandes parties : l'interface graphique, la simulation et les algorithmes distribués. Dans cette section, nous allons approfondir leurs contenus et décrire leurs implémentation. Nous avons choisi une approche orientée objet pour implémenter Visidia.

5.3.1 L'Interface Graphique

L'interface graphique est composée de deux parties, la partie conception du réseau, éditeur de graphe et la partie simulation et visualisation de l'exécution de l'algorithme. Dans cette sous-section, nous n'allons parler que de la première partie, la seconde sera décrite dans la sous-section simulateur.

Comme pour les systèmes de réécriture de graphe, dans Visidia, les réseaux sont représentés sous forme de graphes. Nous allons commencer par détailler un des objets les plus importants qui est l'objet Graphe.

Graphe

Un graphe est composé de noeuds et d'arêtes.

- **vertex :** représente un noeud du graphe. Un noeud a besoin d'une identité unique, de l'ensemble de ses voisins et de l'ensemble des arêtes incidentes à ce noeud. De même pour les graphes orientés, nous avons aussi l'ensemble des arêtes entrantes et sortantes.
- **edge :** une arête relie deux noeuds. Les deux sommets doivent être représentés dans l'objet arête.

L'objet graphe n'est qu'un ensemble d'objet noeud (vertex). Les principales méthodes de l'objet graphe sont :

- **put() :** cette méthode permet d'ajouter un sommet au graphe. Une condition nécessaire dans cette méthode est que ce noeud n'existe pas déjà dans le graphe.
- **size() :** retourne la taille du graphe. La taille est le nombre de noeuds dans le graphe.
- **link() :** ajoute une arête entre deux noeuds passés en paramètre. Une autre méthode pour ajouter une arête orientée est **orientedLink()**.
- **unlink() :** à l'inverse de la méthode précédente, celle-ci permet d'enlever une arête du graphe.
- **remove() :** permet d'enlever un noeud du graphe et toutes ces arêtes incidentes.

Éditeur

L'objet Editeur n'est autre que l'interface graphique utilisée pour la création du réseau ou son chargement à partir d'un fichier.
Pour créer un graphe, la méthode de "drag and drop" est utilisée suivant la position de la souris :

- s'il n'y a *rien* dessous en cliquant et relâchant, un noeud sera créé avec une nouvelle identité. L'identité est nécessaire pour la création du réseau mais pour les algorithmes distribués, cette propriété peut ne pas être utilisée.

- s'il n'y a *rien* dessous mais cette fois-ci on clique, on glisse et enfin on relâche, deux noeuds seront créés avec des identités différentes et en plus, ces noeuds seront reliés par une arête. Si on glisse et on relâche sur un noeud existant, le premier noeud sera créé et une arête reliant ces deux noeuds le sera aussi.
- si maintenant on clique sur un noeud déjà existant et on glisse sur un espace vide (rien), un noeud sera créé et un lien entre le noeud sur lequel on a cliqué dessus et le nouveau sera créé. Si on glisse sur un autre noeud existant, une arête sera créé entre ces deux noeuds.

Un graphe créé peut être sauvegardé pour une utilisation ultérieure. Pour la création de graphe de grande taille, l'interface graphique ne le permet pas, par contre elle permet de les importées. D'où l'utilisation d'autre outil plus spécialisé pour la création de ce genre de graphe est recommandé. C'est pour cela que nous avons opté pour l'importation et l'exportation de graphe de type **GML**.
Un objet de type GML est alors introduit dans Visidia qui crée un objet de type Graphe à partir d'un fichier GML et de faire l'opération inverse.

Une fois le graphe créé, chargé ou importé, le bouton *Simulation* permet de passer à l'étape de simulation.

D'autres fonctionnalités utiles :

Comme dans tout programme de dessin de graphe et autre, les fonctions suivantes existent dans Visidia :

- *undo* pour annuler la dernière tâche,
- *redo* pour ré-exécuter la tâche annulée,
- *selection* pour sélectionner une région du graphe,
- *copie* pour copier la partie sélectionnée,
- *coller* pour coller la partie sauvegardée en mémoire avec la fonction copie ...

Nous avons aussi implémenté un petit programme qui permet de créer des graphe de type GML. Diffèrents types de graphe nous sont proposés, nous citons les plus importants :

- *Grid n m* : le graphe est une grille de $n \times m$ sommets,
- *Cyclic n* : un anneau de n sommets,
- *Tree n [arity]* : un arbre de n sommets et/ou de degré limité à *arity*
- *Rand n [e]* : un graphe quelconque de taille n et/ou de nombre d'arêtes fixé à e,
- *Comp n* : le graphe est un graphe complet de taille n.

Remarque 5.3.1. *Deux remarques importantes sont à signaler. La première concerne la création de graphe, tous les graphes créés satisfont notre modèle de base i.e. tout le graphe est simple, sans cycle, non orienté et connexe.*
Pour créer un graphe complet, nous pouvons utiliser aussi une méthode se trouvant dans l'éditeur de graphe qui permet de rendre le graphe existant en graphe complet.

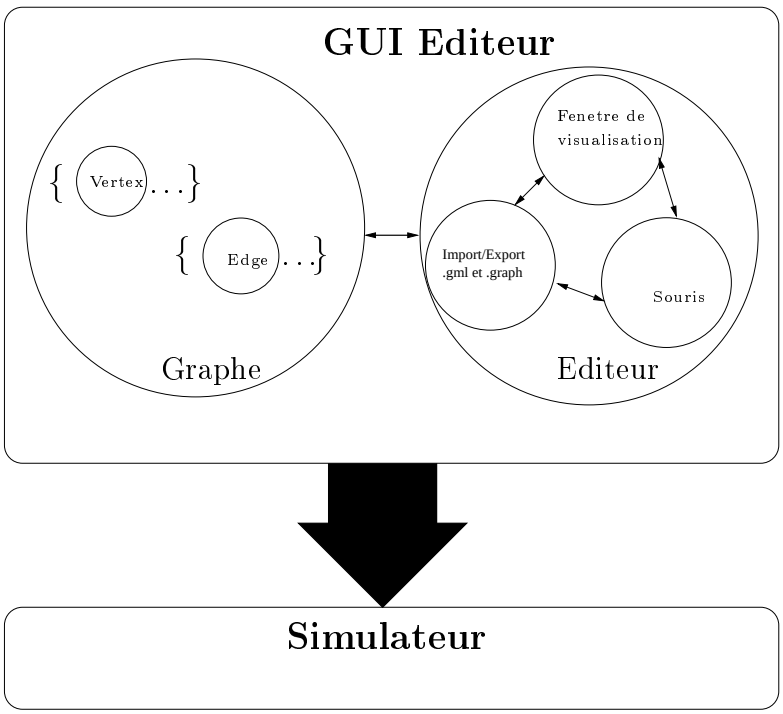


FIG. 14 – *Implémentation du GUI Editeur*

5.3.2 Le Simulateur

Le simulateur est le centre de Visidia. C'est la partie qui communique avec les deux autres parties. Nous allons d'abord décrire cet objet et nous détaillerons par la suite les événements qui circulent entre les principaux objets.

simulation

Ayant construit un objet de type Graphe, le simulateur associe à chaque noeud un objet de type **ProcessData** où pour chaque noeud nous lui ajoutons :

- *processThread* : nous représentons chaque noeud par une entité autonome de calcul. Dans l'implémentation actuelle de Visidia, cette entité est un *Thread*.
- *props* : c'est un ensemble de propriété qu'il peut avoir. Initialement, un noeud possède un champ nommé *label* qui a la valeur N.
- *msgQueue* : est une file d'attente contenant tous les messages destinés à ce sommet.
- *algo* : est l'objet algorithme que nous voulons simuler. Nous parlerons de cet objet ultérieurement.

Pour assurer la communication entre le simulateur et ces entités, un objet de type file d'attente est obligatoire. Cet objet est nommé *evtObjectTmp*. Ce sont, en général, les messages

envoyés qui ne sont pas encore mis dans les files d'attentes des noeuds destinataires i.e. ce sont les messages qui ne sont pas encore visualisés.
Deux autres files sont aussi indispensables, ce sont les files qui relient le simulateur au GUI simulation. La première file qui est une file d'événements qui relie le simulateur au GUI simulation est nommée *evtQ* et la deuxième qui est une file d'acquittements est appelée *ackQ*

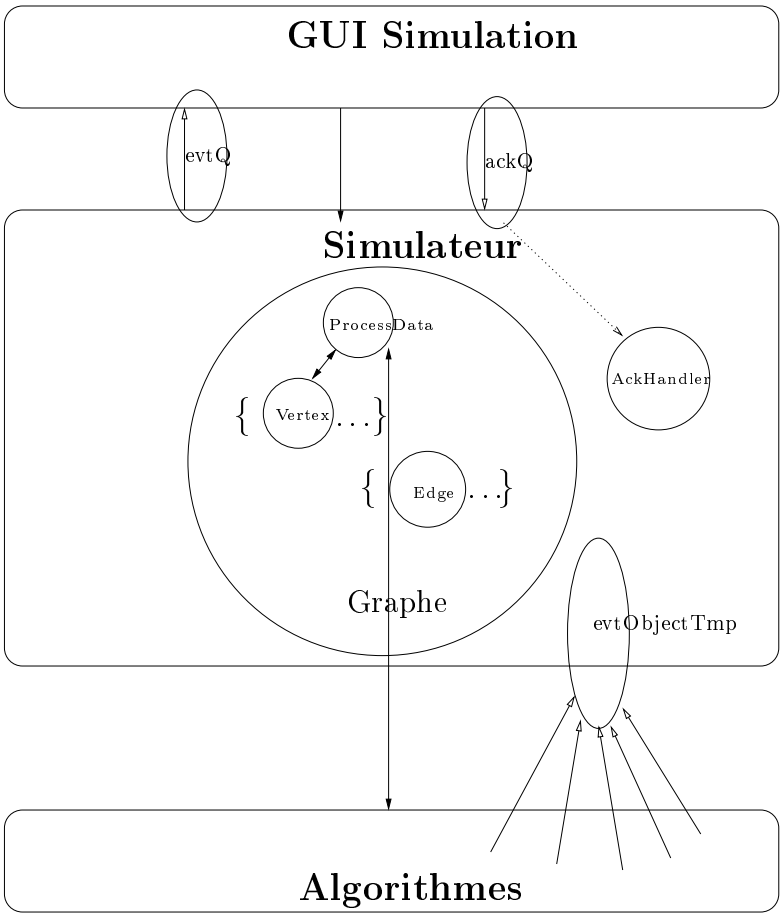


Fig. 15 – *Implémentation du Simulateur*

Les principales méthodes utilisées dans l'objet simulateur et leur fonctionnement sont :

- **sendTo() :** (resp. **sendToNext()**) envoie un message d'un noeud *id* vers un port *door* (le noeud suivant si le graphe est orienté). Cette méthode ne met pas le message directement vers la file d'attente du noeud destinataire mais elle met un événement dans *evtQ* pour permettre l'animation du message avant son envoi réel vers le noeud destination,
- **getNextMessage() :** (resp. **getNextMessageFromPrevious()** vérifie la file d'attente du noeud, s'il existe un message ou bien s'il existe un message d'un certain port

et/ou avec un certain critère et retourne le message (un des critères dans ce cas est le noeud précédent pour un graphe orienté). S'il n'y a pas de message, le noeud est bloqué jusqu'à la réception d'un message qui satisfait tous les critères voulus,

- **getArity() :** pour un noeud donné, cette méthode retourne le degré de ce noeud dans le graphe,
- **sizeOfTheGraph() :** retourne la taille du graphe (le nombre de noeuds du graphe),
- **putNodeProperty() :** permet de modifier ou d'ajouter une propriété à un noeud donné. Une propriété est définie par un identifiant qui est une chaîne de caractères et par un objet qui est la valeur de cette propriété,
- **getNodeProperty() :** retourne la valeur d'une propriétée donné d'un noeud donné en paramètre,
- **startSimulation() :** c'est la méthode la plus importante car c'est ici que nous allons créer les Threads, mettre l'algorithme que nous voulons exécuter dans chacun et vérifier tous les acquittements reçus pour les traiter. La dernière vérification est le travail de l'objet *AckHandler*.

L'objet *AckHandler* permet de traiter tous les acquittements que le simulateur reçoit et les actions qu'il fait. Les acquittements sont les suivants : **NODE_PROPERTY_CHANGE**, **EDGE_STATE_CHANGE**, **MESSAGE_SENT**. Pour les deux premiers, le simulateur ne fait rien puisque les actions de ces événements sont déjà traitées, leur utilité est de savoir que la GUI simulation les a réellement exécutés. Par contre, pour le dernier acquittement, MESSAGE_SENT, le simulateur sait que l'animation du message est faite et que maintenant le message doit être mis dans la file d'attente du noeud destinataire et ainsi, on sait que ce qu'on voit à l'écran, GUI simulation, est bien ce qui se passe réellement dans la simulation. i.e. un message qui est en transit dans l'animation ne doit pas exister réellement dans la file d'attente du destinataire tant que l'animation n'a pas fini.

GUI simulation

C'est dans cette fenêtre que nous allons visualiser l'animation de l'exécution de l'algorithme distribué. Même si c'est une partie indépendante du simulateur, les deux parties sont très liées car beaucoup d'actions que l'utilisateur appelle sont exécutées par l'objet simulateur comme par exemple le début de la simulation, son arrêt provisoire, son arrêt définitif...Dans cette partie, nous expliquons les actions et les événements reçu par le simulateur.
Comme pour les autres modules, l'objet graphe figure aussi enrichi, pour chaque objet vertex (noeud) de son emplacement dans le plan. i.e. chaque sommet possède deux variables *posx* et *posy*. Les coordonnés du sommet se trouve dans l'objet *SommetDessin*.
L'objet *SimulationPanel* est l'équivalent de l'objet *AckHandler* du simulateur. Cet objet va traiter tous les événements qu'il reçoit du simulateur. Nous avons déjà parlé de ces événements dans la section simulateur.

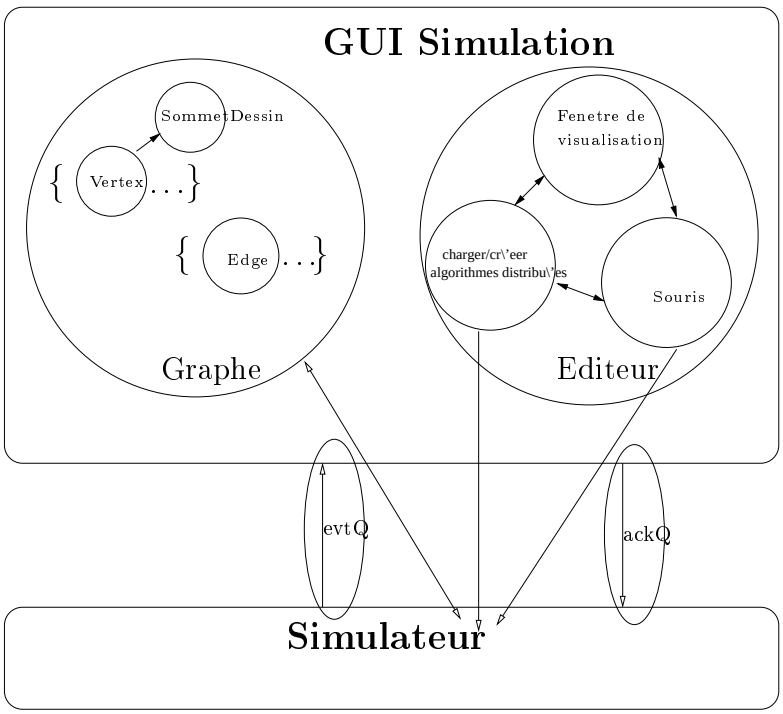


Fig. 16 – *Implémentation du GUI Simulation*

- **NODE_PROPERTY_CHANGE :** la fenêtre de simulation sait que le noeud *id* a modifié sa table de propriétés. Elle récupère sa nouvelle table et met à jour les propriétés relatives à ce noeud. Un des exemples les plus visible est le changement de la propriété *label* qui génère une modification de la propriété existante sous le noeud et sa couleur.
- **EDGE_STATE_CHANGE :** en général, nous utilisons les propriétés des arêtes pour visualiser les synchronisations et l'évolution des arbres recouvrants dans le graphe. La première est appelé *SyncState*, qui permet, pour deux noeuds *id1* et *id2*, d'enluminer (de distinguer) ces noeuds et l'arête incidente ou bien de les rendre normaux (enlever cet effet). La deuxième, *MarkedState*, permet seulement de marquer l'arête incidente entre les deux noeuds *id1* et *id2*. Si l'arête est déjà marquée, c'est l'effet inverse qui se fait.
- **ALGORITHM_END :** si tous les algorithmes ont fini le simulateur envoie cet événement au GUI simulation. Ce dernier affiche un message disant que l'algorithme a fini.
- **MESSAGE_SENT :** une fois qu'un événement d'envoi de message arrive, le GUI simulation anime le message entre le noeud *id1* et le noeud *id2* par pas. Le pas est calculé à partir de la distance des deux noeuds et de la vitesse de l'animation. Dans la version actuelle de Visidia, les arêtes n'ont pas de vitesse propre et la durée que le message met pour être animé est proportionnelle à la distance réelle entre les deux

noeuds.

Pour chaque action traitée dans l'interface, le GUI simulation envoie un acquittement vers le simulateur.

5.3.3 Bibliothèques de primitives fournies

Ce sont les entités autonomes qui exécutent l'algorithme distribué que nous voulons visualiser, tester ou expérimenter. Pour faire le lien entre le simulateur et les algorithmes distribués, un grand objet appelé *Algorithm* est créé. Dans cet objet nous trouvons l'algorithme à exécuter, et des méthodes utilisant les méthodes du simulateur mais qui ne font pas apparaître l'identité des noeuds pour le programmeur. Les principales méthodes et leurs actions sont :

- **sendTo(int, Message) :** (resp. **sendToNext(Message)**) pour envoyer un message à un noeud voisin à partir d'un port (au noeud suivant si le graphe est orienté). Le numéro de port de chaque noeud est compris entre 0 et son degré moins un. L'équivalence du noeud destination est faite dans le simulateur. Donc le message est envoyé au simulateur qui lui va le mettre directement dans la file d'attente du destinataire. Cette opération ne se fait pas instantanément, elle suit l'ordre d'opérations dont nous avons déjà parlé.
- **sendAll(Message) :** envoie le même message à tous les voisins du noeud. Cette méthode utilise la méthode précédente.
- **receive(Door) :** cette méthode permet de prendre le premier message dans la file d'attente du noeud. S'il n'y a aucun message, le Thread du noeud est bloqué jusqu'à ce qu'un message arrive.
- **receiveFrom(int) :** (resp. **receiveFromPrevious(int)** est équivalente à la méthode précédente sauf que nous voulons le premier message reçu d'un noeud donné identifié par son port (par le noeud pointant vers ce noeud). Cette méthode est aussi bloquante.
- **getArity() :** retourne le degré du sommet.
- **getNetSize() :** retourne la taille du graphe. Cette méthode est optionnelle, elle n'est utilisée que si l'algorithme a besoin de cette connaissance pour fonctionner.
- **getId() :** retourne l'identité du noeud dans le graphe. Comme pour la méthode précédente, elle est optionnelle et dépend de l'algorithme lui même.
- **putProperty(String, Object) :** permet de créer une propriété si le nom donné n'existe pas et met la valeur. Si la propriété existe déjà, elle sera modifiée avec la nouvelle valeur.
- **getProperty(String) :** retourne la valeur de la propriété donnée en paramètre.
- **setDoorState(EdgeState,int) :** modifie l'état de l'arête se trouvant au port donné en paramètre.

Nous remarquons que tous ces méthodes figurent aussi dans l'objet simulateur mais à la différence que celles-ci n'ont pas besoin de connaître le type du graphe, ni les identifiants des voisins. Certes, ça sera chose faite pour pouvoir voir l'animation de l'algorithme mais pour l'utilisateur et le concepteur, les équivalences sont cachés.

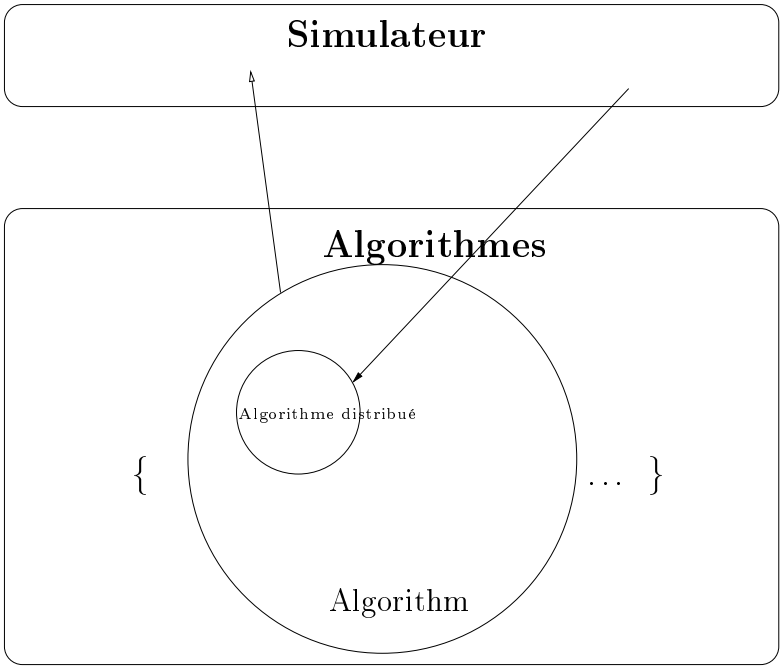


Fig. 17 – *Implémentation des Algorithmes*

5.4 Les outils existants

Visidia n'est pas le seul outil développé pour la visualisation des algorithmes distribués, nous pouvons citer par exemple VADE [65], PARADE... Dans la suite, nous allons détailler quelques caractéristiques de ces outils. La particularité de Visidia est de reposer sur une base théorique solide.

5.4.1 VADE

VADE (Visualisation of Algorithms in Distributed Environnements) est un outil développé au Weizmann Institute of Science. VADE permet de visualiser les algorithmes distribués dans un système asynchrone. Il permet, entre autre, à l'utilisateur de visualiser l'exécution d'un algorithme dans une page Web tandis que l'exécution réelle se déroule sur un serveur distant. Ceci permet son utilisation à distance sans avoir besoin de l'installer. Le système distribué utilisé par VADE est le suivant :

- le réseau est fiable ;
- les messages envoyés par un processeur arrivent dans le même ordre de leur envoi ;
- le réseau est asynchrone, chaque processeur possède sa propre horloge et il n'y a pas d'horloge globale.

Pour synchroniser l'animation et l'exécution de l'algorithme, des événements sont envoyés pour les modifications des états et l'utilisation du *send synchronization* et du *receive synchronization*.

5.4.2 LYDIAN

LYDIAN (Library of Distributed Algorithms and Animations) est un environnement pour la simulation et la visualisation des algorithmes distribués qui permet aux étudiants d'avoir un environnement expérimental pour tester et visualiser le comportement des algorithmes distribués.
Lydian possède un créateur de réseau et une autre fenêtre pour l'animation. Celle-ci est basée sur la trace de l'exécution des algorithmes distribués. L'environnement distribué utilise une horloge globale.

5.4.3 PARADE

PARADE (PARallel program Animation Development Environnement) est un environnement pour la visualisation et pour déboguer les algorithmes parallèles et distribués. Il se compose de trois parties :

- programme de surveillance et de trace ;
- système de visualisation ;
- utilisation de la trace pour visualiser les actions du programme.

5.5 Méthode d'implémentation des systèmes de réécriture de graphe et des calculs locaux

Pour implémenter les systèmes de réécriture de graphe, plusieurs méthodes existent. Une des méthodes est l'échange directe des étiquettes entre les noeuds du réseau. Cette implémentation est rapide mais pas automatique. C'est à dire, il n'est pas facile d'implémenter un système de réécriture de graphe avec cette méthode. La méthode la plus simple est de synchroniser les noeuds avec leurs voisins, échanger les étiquettes puis exécuter une règle de calcul local.

5.5.1 Les différents types des calculs locaux

Bien qu'ils aient en commun le fait que l'état d'un noeud dépende seulement de l'état de ses voisins ou de certains de ses voisins, les calculs locaux sont de trois types :

RV (Rendez-Vous) : dans une étape de calcul, les étiquettes d'un couple de sommets reliés par une arête sont modifiées selon la règle où les étiquettes apparaissant sur l'arête et sur ses sommets.

LC_1 (Calcul local de type 1) : dans une étape de calcul, l'étiquette attachée au centre de l'étoile est modifiée selon une règle qui dépend des étiquettes de l'étoile (les étiquettes des feuilles ne sont pas modifiées). On appelle étoile un sous-graphe composé d'un sommet et de tous ses voisins.

LC_2 (Calcul local de type 2) : dans une étape de calcul, les étiquettes attachées au centre et aux feuilles de l'étoile peuvent être modifiées selon une règle qui dépend des étiquettes de l'étoile.

5.5.2 Implémentation avec des procédures probabilistes

Puisque Angluin [2] a montré qu'il n'y a aucun algorithme déterministe pour implémenter des synchronisations locales dans un réseau anonyme par échange de messages asynchrones (voir [72]), nous allons utiliser des procédures probabilistes pour les implémenter (voir [54, 55, 8]). D'ailleurs, l'aspect aléatoire fournit des réalisations efficaces et faciles, en particulier, dans le contexte de la visualisation parce qu'il permet à l'utilisateur d'observer l'exécution entière de l'algorithme comme nous allons le montrer dans la suite.

Implémentation de *RV*. Nous considérons la procédure probabiliste suivante donnée dans l'algorithme 1 pour implémenter le Rendez-Vous. L'exécution est divisée en rounds ; dans chaque round, chaque sommet v choisit au hasard un de ses voisins $c(v)$, et lui envoie 1 et envoie 0 à tous les autres. Il y a un rendez-vous entre v et $c(v)$ si $c(v) = c(c(v))$, on dit alors que v et $c(v)$ sont synchronisés. Quand v et $c(v)$ sont synchronisés il y a un échange des messages entre v et $c(v)$ qui permet aux deux noeuds de modifier leurs étiquettes.

Chaque noeud v répète infiniment les actions suivantes :
Un noeud v choisit au hasard un de ses voisins $c(v)$;
le noeud v envoie 1 à $c(v)$;
le noeud v envoie 0 aux autres voisins à part $c(v)$;
le noeud v reçoit les messages de ses voisins.
(Il y a un rendez-vous entre v et $c(v)$ si v reçoit 1 de $c(v)$; dans ce cas, une étape de calcul peut être effectuée. *)*

Algorithme 1: Rendez-vous probabiliste

Implémentation de LC_1. Soit LE_1 l'élection locale donnée par l'algorithme 2 pour implémenter LC_1 ; elle est divisée en rounds, dans chaque round, chaque processeur v choisit

un nombre entier $rand(v)$ de manière aléatoire et l'envoie à ses voisins. v est alors élu dans $B(v,1)$ si pour chaque sommet w de $B(v,1)$ différent de v : $rand(v) > rand(w)$. Dans ce cas, une étape de calcul dans $B(v,1)$ sera exécutée : le centre reçoit les étiquettes de ses feuilles (voisins) puis il change son étiquette.

Chaque noeud v répète indéfiniment les actions suivantes :
le noeud v choisit au hasard un entier $rand(v)$;
le noeud v envoie $rand(v)$ à tous ses voisins ;
le noeud v reçoit les entiers de tous ses voisins.
(le noeud v est élu si $rand(v)$ est strictement supérieur à tous les entiers reçus par v; dans ce cas, une étape de calculs peut être exécutée dans $B(v,1)$. *)*

Algorithme 2: Élection LE_1 probabiliste

Implémentation de LC_2. Soit LE_2 l'élection locale donnée par l'algorithme 3 pour implémenter LC_2 ; elle est divisé en rounds, dans chaque round, chaque processeur v choisit aléatoirement un nombre $rand(v)$.

Le processeur v envoie à ses voisins $rand(v)$. Quand il reçoit de tous ses voisins leurs entiers, il envoie à chaque voisin w l'entier maximum de l'ensemble des entiers reçus de ses voisins sauf $rand(w)$. v est élu dans $B(v,2)$ si $rand(v)$ est strictement supérieur à tous les entiers $rand(w)$ pour tout sommet w dans la boule de centre v et de rayon 2; dans ce cas, une étape de calcul peut être faire dans $B(v,1)$. Durant cette étape de calcul, il y a un échange d'étiquette des noeuds de $B(v,1)$ qui permet aux noeuds de $B(v,1)$ de mettre à jour leurs étiquettes.

Chaque noeud v répète indéfiniment les actions suivantes :
le noeud v choisit au hasard un entier $rand(v)$;
le noeud v envoie $rand(v)$ à ses voisins ;
le noeud v reçoit un message de tous ses voisins ;
soit Int_w l'entier maximum de l'ensemble des entiers que v a reçu de ses voisins différents de w ;
le noeud v envoie à chaque sommet w Int_w;
le noeud v reçoit les entiers de tous ses voisins ;
(Il y a une élection LE_2- dans $B(v,2)$ si $rand(v)$ est strictement supérieur à tous les entiers reçus de v; dans ce cas, une étape de calcul peut être faite dans $B(v,1)$. *)*

Algorithme 3: Élection LE_2 probabiliste

L'analyse de ces algorithmes a été faite dans [54, 55, 76]. Elle est basée sur la considération de *tours* : afin de mesurer l'exécution de l'algorithme en termes de nombre de rendez-vous ou d'élection locales qui ont lieux, nous supposons que à un certain instant, chaque noeud envoie et reçoit des messages. Ainsi le paramètre d'intérêt, qui est le nombre (aléatoire) de rendez-vous ou d'élections locales, est le nombre maximal (c.-à-d. dans le cas où les noeuds sont activés) autorisé par l'algorithme.

5.5.3 Implémentation des calculs locaux

Les procédures probabilistes nous facilitent l'implémentation et l'animation des algorithmes distribués codés sous forme de calculs locaux et plus précisément sous forme de systèmes de réécriture de graphe.
Le modèle général de l'algorithme devient comme suit :

```
tant que(run){
        synchroniser avec un/tous les voisins(RDV,LC1 ou LC2)
        Échange des étiquettes, des attributs et des informations
        faire un pas de calcul
        Mise à jours des étiquettes, attributs et états
        Arrêt de la synchronisation
}
```

Algorithme 4: Algorithme général pour implémenter les calculs locaux

Cette implémentation diffère du modèle de synchronisation utilisé. Même l'implémentation de ces procédures est un peu plus différente que celle décrite en dessus. Dans ce qui suit, nous allons détailler l'implémentation des procédures probabilistes et de l'algorithme distribué général pour leurs utilisation.

RDV

L'implémentation du RDV reste inchangé. L'algorithme écrit avec les primitives de Visidia est décrit par l'algorithme 5.

L'implémentation des algorithmes distribués codés sous forme de calculs locaux utilisant le RDV est décrit par l'algorithme 6.

LC1

L'implémentation du LC1 est différente que celle du RDV. Dans le LC1, on distingue deux parties : Le centre de la synchronisation et ses voisins. Ces deux parties sont appelées à Visidia *starCenter* et *starBorder*. L'algorithme écrit avec les primitives de Visidia est décrit par l'algorithme 7.

L'implémentation des algorithmes distribués codés sous forme de calculs locaux utilisant le LC1 est décrit par l'algorithme 8.

LC2

L'implémentation du LC2 est équivalente à une LC1, sauf que dans ce cas les feuilles de l'étoile peuvent changer leurs états. L'algorithme écrit avec les primitives de Visidia est

```
int RDV(){
        int i=-1;
        while (i <0) {
           i = trySynchronize();
        }
        return i;
}

int trySynchronize(){
        int[] answer = new int[arity];
        int nb;
        int choosenNeighbour=chooseRandom(arity);

        sendTo(choosenNeighbour, 1);
        for(int door= 0; door < arity; door++){
           if ( door != choosenNeighbour)
              sendTo(door, 0);
        }

        for(int door= 0; door < arity; door++){
           Message msg = receiveFrom(door);
           answer[i]= msg.value();
        }

        if (answer[choosenNeighbour] == 1){
           return choosenNeighbour;
        }
        else {
           return -1;
        }
}
```

Algorithme 5: Implémentation du RDV simple avec Visidia

décrit par l'algorithme 9.

L'implémentation des algorithmes distribués codés sous forme de calculs locaux utilisant le LC2 est décrit par l'algorithme 10.

Remarque 5.5.1. Les algorithmes de synchronisations sont des algorithmes simple puisqu'ils ne traitent pas le problème de la terminaison. Il existe deux niveaux de terminaison. La terminaison globale de l'algorithme, dans ce cas le noeud qui détecte la terminaison globale fait un broadcast de la terminaison et c'est au niveau de la synchronisation que cette information sera traitée. La terminaison locale, un noeud a fini son calcul et il sort

```
while(run){
        synchro=RDV() ;
        sendTo(synchro,états) ;
        Message msg=receiveFrom(synchro) ;
        exécute un pas de calcul
        mise à jour de mes états
        breakSynchro() ;
}
```

Algorithme 6: Algorithme général avec le RDV

de la boule de synchronisation, il informe ses voisins pour qu'ils ne se bloquent pas en attendant un message de ce noeud.

5.5.4 LE1_2

Un autre algorithme a été implémenté qui fusionne le LC1 et le LC2. Dans ce cas un noeud doit connaître quel type de synchronisation (LC1 ou LC2) il a besoin et il informe la fonction de synchronisation. Cette synchronisation est utile dans le cas où quelques règles de réécriture de graphe nécessitent la synchronisation LC1 et les autres la synchronisation LC2.

Cet algorithme ressemble à l'algorithme de LC1 puisqu'un noeud u choisit un nombre aléatoire et l'envoie à tous ses voisins en indiquant le type de synchronisation qu'il veut (LC1 ou LC2). Si le nombre du noeud est le maximum de ces voisins, s'il a choisi le synchronisation LC1 et si aucun noeud voisin n'est sollicité par un noeud qui demande une synchronisation LC2 et qui a un nombre supérieur à u alors u sort de la synchronisation en étant le centre. Dans le cas où il veut une synchronisation de type LC2, il demande à ses voisins si son nombre est le plus grand que ceux de leurs voisins et il attend une confirmation. Si un de ces voisins lui envoie un "non", il informe ces voisins qu'il n'est pas l'élu et il essaie une autre fois. L'algorithme LE1_2 est implémenté dans l'algorithme 11.

Analyse du LE1_2

Soit p la probabilité qu'un noeud v veut se synchroniser avec ses voisins en LC1 et soit $q = 1 - p$. En d'autres termes, $p = \frac{card\{r \in R | r \text{ ne change que son état}\}}{card\{R\}}$, c'est le nombre de règles qui ne modifient pas l'état de ses voisins sur le nombre de règles totales. Nous allons étudier la probabilité pour un noeud v pour qu'il soit centre d'une synchronisation.

Lemme 5.5.1. *La probabilité pour qu'un noeud u soit centre d'une synchronisation est égale à* $\frac{q}{N_2(v)} + \frac{p}{d(v)+1} * (1 - \frac{q}{2})^{(N_2(v)-(d(v)+1))}$

Démonstration. Soit $P_{LC1}(v)$ la probabilité que le noeud v gagne l'élection de LC1 et aucun noeud $w \in B(v,2)$ ne gagne l'élection de LC2, $P(LC2(v))$ la probabilité que $rand(v)$ est le plus grand des noeuds de $B(v,2)$ et $P(LC1(v))$ la probabilité que $rand(v)$ est le plus grand des noeuds de $B(v,1)$. D'après [54, 55], nous avons : $P(LC1(v)) = \frac{1}{d(v)+1}$ et $P(LC2(v)) = \frac{1}{N_2(v)}$, où $N_2(v) = |B(v,2)|$.

$$P(LE1\_2(v)) = q * P(LC2(v)) + p * P_{LC1}(v)$$

$$\begin{aligned}
P_{LC1}(v) &= P(LC1(v)) * (\prod_{w \in B(v,2)/B(v,1)} (p + q(1 - P(rand(w) > rand(v))))) \\
&= P(LC1(v)) * (\prod_{w \in B(v,2)/B(v,1)} (p + q - q * P(rand(w) > rand(v)))) \\
&= P(LC1(v)) * (\prod_{w \in B(v,2)/B(v,1)} (1 - q * P(rand(w) > rand(v)))) \\
&= \frac{1}{d(v)+1} * (\prod_{w \in B(v,2)/B(v,1)} (1 - \frac{q}{2})) \\
&= \frac{1}{d(v)+1} * (1 - \frac{q}{2})^{(N_2(v) - (d(v)+1))}
\end{aligned}$$

d'où,

$$P(LE1\_2(v)) = \frac{q}{N_2(v)} + \frac{p}{d(v)+1} * (1 - \frac{q}{2})^{(N_2(v) - (d(v)+1))}$$

Nous allons démontrer que $P(LE1\_2(v))$ est une fonction de probabilité. Pour cela, soit Δ le degré maximum de G, pour tout noeud u de G, nous avons :

$$1 \leqslant N_2(u) \leqslant \Delta^2 + 1$$

La fonction inverse nous donne :

$$\frac{1}{\Delta^2 + 1} \leqslant \frac{1}{N_2(u)} \leqslant 1$$

Ce qui nous donne une simplification du produit :

$$P(LE1\_2(v)) \leqslant q + \frac{p}{d(v)+1} * (1 - \frac{q}{2})^{N_2(v) - (d(v)+1))}$$

Pour maximiser cette fonction, $(1-\frac{q}{2})^{(N_2(v)-(d(v)+1))}$ doit être maximal donc doit être égal à 1. De plus, pour $\frac{1}{d(v)+1}$ qui peut être égal à 1 (on suppose que $d(v)=0$). Donc, on aura :

$$\begin{aligned} P(LE1\_2(v)) &\leqslant q + \frac{p}{d(v)+1} * (1-\frac{q}{2})^{(N_2(v)-(d(v)+1))} \\ &< q+p \\ &< 1 \end{aligned}$$

□

Nous allons simplifier le calcul en supposant que le graphe est Δ-régulier. Donc, la probabilité devient :

$$\begin{aligned} P(LE1\_2(v)) &= \frac{q}{\Delta^2+1} + \frac{p}{\Delta+1} * (1-\frac{q}{2})^{(\Delta^2+1-(\Delta+1))} \\ &= \frac{q}{\Delta^2+1} + \frac{p}{\Delta+1} * (1-\frac{q}{2})^{(\Delta^2-\Delta)} \end{aligned}$$

Dans le cas d'un anneau, Δ est égal à 2. L'équation devient :

$$\begin{aligned} P(LE1\_2(v)) &= \frac{q}{5} + \frac{p}{3} * (1-\frac{q}{2})^2 \\ &= \frac{p^3}{12} + \frac{p^2}{6} - \frac{7*p}{60} + \frac{1}{5} \end{aligned}$$

Cette fonction nous donne la courbe de la figure 18.

Ce qui signifie que pour que la synchronisation LE1_2 soit efficace, il faut que p soit égal à 0 ou p soit supérieur à $\frac{2}{5} * \sqrt{15} - 1 \simeq 0.549$.

Et plus généralement, pour un graphe Δ-régulier, $P(LE1\_2(v)) \geqslant \frac{1}{\Delta^2+1}$ implique que p doit être supérieur ou égal à $2 * e^{\frac{\ln\left(\frac{\Delta+1}{\Delta^2+1}\right)}{(\Delta*(\Delta-1))}} - 1$

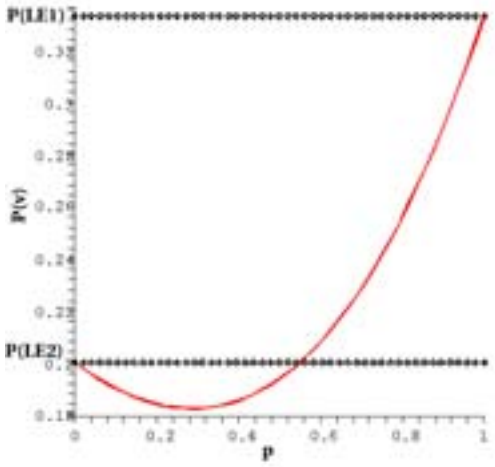


FIG. 18 – P(v centre dans LE1_2) en fonction de p dans un anneau

Pour un graphe quelconque, nous avons :

$$\begin{aligned}
\frac{q}{N_2(v)} + \frac{p}{d(v)+1} * (1-\frac{q}{2})^{(N_2(v)-(d(v)+1))} \geqslant \frac{1}{N_2(v)} &\implies \\
\frac{q}{N_2(v)} + \frac{p}{d(v)+1} * (1-\frac{q}{2})^{(N_2(v)-(d(v)+1))} - \frac{1}{N_2(v)} \geqslant 0 &\implies \\
q + \frac{p}{d(v)+1} * N_2(v) * (1-\frac{q}{2})^{(N_2(v)-(d(v)+1))} - 1 \geqslant 0 &\implies \\
1-p + \frac{p}{d(v)+1} * N_2(v) * (1-\frac{1-p}{2})^{(N_2(v)-(d(v)+1))} - 1 \geqslant 0 &\implies \\
p * (-1 + \frac{N_2(v)}{d(v)+1} * (1-\frac{1-p}{2})^{(N_2(v)-(d(v)+1))}) \geqslant 0 &
\end{aligned}$$

Comme p est positive, soit $F(p)$ est égal à $-1+\frac{N_2(v)}{d(v)+1}*(1-\frac{1-p}{2})^{(N_2(v)-(d(v)+1))}$, cherchons p pour que $F(p)$ soit toujours positif. Donc, nous avons :

$$\begin{aligned}
p * \left(-1 + \frac{N_2(v)}{d(v)+1} * (1-\frac{1-p}{2})^{(N_2(v)-(d(v)+1))}\right) \geqslant 0 &\implies \\
F(p) = -1 + \frac{N_2(v)}{d(v)+1} * (1-\frac{1-p}{2})^{(N_2(v)-(d(v)+1))} \geqslant 0 &\implies \\
\frac{N_2(v)}{d(v)+1} * (1-\frac{1-p}{2})^{(N_2(v)-(d(v)+1))} \geqslant 1 &\implies \\
\frac{N_2(v)}{d(v)+1} * (\frac{1+p}{2})^{(N_2(v)-(d(v)+1))} \geqslant 1 &
\end{aligned}$$

De la même façon, soit $G(p) = \frac{N_2(v)}{d(v)+1} * (\frac{1+p}{2})^{(N_2(v)-(d(v)+1))}$. Comme $N_2(v) \geqslant d(v)+1$ alors $G(p)$ est une fonction croissante, $G(0) = \frac{N_2(v)}{d(v)+1} * (\frac{1}{2})^{(N_2(v)-(d(v)+1))}$ et $G(1) = \frac{N_2(v)}{d(v)+1}$, il existe p_0 tel que $G(p_0) = 1$. i.e. :

$$\begin{aligned}
\frac{N_2(v)}{d(v)+1} * (\frac{1+p_0}{2})^{(N_2(v)-(d(v)+1))} = 1 &\implies \\
(\frac{1+p_0}{2})^{(N_2(v)-(d(v)+1))} = \frac{d(v)+1}{N_2(v)} &\implies \\
\ln(\frac{1+p_0}{2})^{(N_2(v)-(d(v)+1))} = \ln\frac{d(v)+1}{N_2(v)} &\implies \\
(N_2(v)-(d(v)+1)) * \ln(\frac{1+p_0}{2}) = \ln\frac{d(v)+1}{N_2(v)} &\implies \\
\ln(\frac{1+p_0}{2}) = \frac{\ln\frac{d(v)+1}{N_2(v)}}{(N_2(v)-(d(v)+1))} &\implies \\
(\frac{1+p_0}{2}) = e^{\frac{\ln\frac{d(v)+1}{N_2(v)}}{(N_2(v)-(d(v)+1))}} &\implies \\
(1+p_0) = 2 * e^{\frac{\ln\frac{d(v)+1}{N_2(v)}}{(N_2(v)-(d(v)+1))}} &\implies \\
p_0 = 2 * e^{\frac{\ln\frac{d(v)+1}{N_2(v)}}{(N_2(v)-(d(v)+1))}} - 1 &\implies \\
p_0 = 2 * (\frac{d(v)+1}{N_2(v)})^{(\frac{1}{N_2(v)-(d(v)+1)})} - 1 &
\end{aligned}$$

Ce qui nous donne le théorème suivant :

Théorème 5.5.1. *La synchronisation LE1_2 est plus performante que la synchronisation LC2 si* $p = 0$ *ou* $p \geqslant 2 * (\frac{d(v)+1}{N_2(v)})^{(\frac{1}{N_2(v)-(d(v)+1)})} - 1$.

Nous remarquons que si p tend vers 0, nous avons la même probabilité que celle de la synchronisation LC2 et de la même manière, si p tend vers 1, nous obtenons la même probabilité pour la synchronisation LC1. Cette nouvelle synchronisation, nous permet d'avoir plus de synchronisation selon la probabilité p. Ce qui nous permet d'augmenter la probabilité d'être centre d'une synchronisation et aussi de diminuer le temps d'attente d'une synchronisation.

```
Vector LC1(){
    int choosenNumber = chooseRandomNumber();

    sendAll(choosenNumber);

    for (int i = 0; i < arity; i++){
        Message msg = receiveFrom(i);
        answer[i]= msg.value();
    }

    int max = choosenNumber;
    for (int i=0;i < arity; i++){
        if ( answer[i] >= max )
            max = answer[i];
    }

    if (choosenNumber >= max) {
        sendAll(1);

        for (int i=0;i<arity;i++)
            Message msg=receiveFrom(i);

        neighbourCenter=new Vector();
        neighbourCenter.add(starCenter);

        return neighbourCenter;
    }
    else {          neighbourCenter=new Vector();

        sendAll(0);

        for (int i=0; i<arity;i++) {
            Message msg=receiveFrom(i);
            if (msg.value() == 1) {
                neighbourCenter.add(i);
            }
        }
        if (neighbourCenter.size()==0)
            neighbourCenter=null;

        return neighbourCenter;
    }
}
```

Algorithme 7: Implémentation du LC1 simple avec Visidia

```
while(run){
    synchro=LC1() ;
    if (synchro != null) { // le noeud n'appartient pas à une étoile
      if(synchro.elementAt(0)== starCenter) { // le noeud est le centre de l'étoile
        for (int door=0 ;door<arity ;door++)
          Message msg[door]=receiveFrom(door) ;
        exécute un pas de calcul
        mise à jour de mes états
        breakSynchro() ;
      }
      else{ // le noeud est une feuille de l'étoile
        for (int i=0 ;i<synchro.size() ;i++)
          sendTo(synchro.elementAt(i),labels) ;
      }
    }
}
```

Algorithme 8: Algorithme général avec le LC1

```
int LC2(){
        int choosenNumber = chooseRandomNumber() ;
        sendAll(choosenNumber) ;
        for (int i = 0 ; i < arity ; i++){
            Message msg = receiveFrom(i) ;
            answer[i]= msg.value() ;
        }

        int max = choosenNumber ;
        for (int i=0 ;i < arity ; i++){
            if ( answer[i] >= max )
                max = answer[i] ;
        }
        sendAll(max) ;

        for (int i = 0 ; i < arity ; i++){
            Message msg = receiveFrom(i) ;
            answer[i]= msg.value() ;
        }
        int max = choosenNumber ;
        for (int i=0 ;i < arity ; i++){
            if ( answer[i] >= max )
                max = answer[i] ;
        }

        if (choosenNumber >= max) {
            sendAll(1) ;
            for (int i=0 ;i<arity ;i++)
                Message msg=receiveFrom(i) ;
            return neighbourCenter ;
        }
        else {              star=notIntheStar ;
            sendAll(0) ;

            for (int i=0 ; i<arity ;i++) {
                Message msg=receiveFrom(i) ;
                if (msg.value() == 1) {
                    star=i ;
                }
            }
            return star ;
        }
}
```

Algorithme 9: Implémentation du LC2 simple avec Visidia

```
while(run){
        synchro=LC2() ;
        if(synchro== starCenter) { // le noeud est le centre de l'étoile
           for (int door=0 ;door<arity ;door++)
              Message msg[door]=receiveFrom(door) ;
           exécute un pas de calcul
           mise à jour de mes états
           for (int door=0 ;door<arity ;door++)
              sendTo(door,mise à jour des états des voisins) ;
           breakSynchro() ;
        }
        else{           if (synchro !=NotInTheStar) {// le noeud est une feuille de l'étoile
              sendTo(synchro,labels) ;
              Message msg=ReceiveFrom(synchro) ; // reçoit ses nouveaux états
              mise à jour de mes états
           }
        }
}
```

Algorithme 10: Algorithme général avec le LC2

Chaque noeud v répète indéfiniment les actions suivantes :
le noeud v choisit au hasard un entier $rand(v)$;
le noeud v envoie $rand(v)$ à ses voisins et le type de synchronisation qu'il souhaite ;
le noeud v reçoit un message de tous ses voisins ;
soit Int_w l'entier maximum de l'ensemble des entiers que v a reçu de ses voisins différents de w ;
si $rand(v) > Int_w$ alors
le noeud v envoie $OK?$ à ses voisins ; le noeud v reçoit un message $Conf_w$ de tous ses voisins ;

(le noeud v est élu si tous les messages $Conf_w$ sont OK ; dans ce cas, une étape de calculs peut être exécutée dans $B(v, 1)$. *)*
sinon
le noeud v reçoit $OK?$ de quelques uns de ses voisins W ;
si aucun u de W n'a demandé une synchronisation $LC2$ alors
le noeud v envoie OK à ses voisins W ;
sinon si il existe un noeud u tel que $rand(u) > Int_u$ et u a choisi la synchronisation $LC2$ alors
le noeud v envoie OK à son voisin u et $!OK$ à tous les voisins dans W/u ;
sinon pour tout noeud u de W tel que u a choisit la synchronisation $LC1$; le noeud v envoie OK à son voisins u ;

Algorithme 11: Élection LE1_2 probabiliste

Chapitre 6

Partages des Ressources

Dans ce chapitre, nous présentons quelques algorithmes qui résolvent les problèmes du partages des ressources. Nous commençons par le problème du dîner des philosophes, qui est un problème de conflits entre les processus. Dans ce problème il y a un ensemble de ressources à partager entre plusieurs processus. Nous nous sommes intéressé au problème de l'exclusion mutuelle. Dans ce cas, il n'y a qu'une seule ressource à partager entre tous les processus. Nous avons traité le cas où les liens entre processus forment un arbre ensuite nous avons généralisé cette approche pour un graphe quelconque. Dans ce chapitre, nous faisons le lien entre l'analyse des algorithmes distribués et Visidia. En effet, nous les avons implémentés testés et expérimentés sur Visidia.

6.1 Résolutions des conflits

6.1.1 Introduction

Résoudre les conflits dans un système distribué est l'un des problèmes majeurs de l'algorithmique distribuée. Nous considérons un réseau de processeurs partageant un ensemble de ressources. Ce réseau est représenté par un graphe connexe non orienté. Les sommets du graphe sont les processeurs et les arêtes représentent les conflits entre les processeurs.

Nous considérons le cas où un processeur a besoin de toutes les ressources pour pouvoir effectuer ses calculs. Ce problème est connu sous le nom de *drinking philosophers problem* qui a été introduit par Chandy et Misra dans [20] comme une généralisation du *dining philosophers problem* [24], un des fameux paradigmes des calculs distribués.

Le problème du dîner des philosophes

Le problème du dîner des philosophes, aussi connu sous le nom du *problème des philosophes*, est soulevé par Dijkstra dans [23]. Cinq philosophes sont assis autour d'une table

chacun possède un plat de spaghettis mais il n'y a que cinq baguettes mises entre les assiettes. Un philosophe a besoin des deux baguettes pour pouvoir manger, deux philosophes voisins ne peuvent donc pas manger en même temps. A un instant t, un philosophe peut être dans l'un des deux états suivant : mange, pense. Un philosophe ne peut rester indéfiniment dans le même état. Comme deux philosophes voisins ne peuvent être à un même instant dans l'état mange, les voisins d'un philosophe dans l'état mange sont donc dans l'état pense.
Une solution pour ce problème est l'utilisation d'un algorithme qui assure l'exclusion mutuelle pour les baguettes (une baguette ne peut être utilisée que par une seule personne), prévient du "deadlock" (au moins un philosophe affamé peut manger en un temps fini) et assure la survie (tous les philosophes qui veulent manger finissent par le faire après un temps fini).

Le problème des philosophes généralisés

Une généralisation du problème du dîner des philosophes est introduit par Chandy et Misra dans [20]. Cette modification consiste à changer les plats par des verres et les baguettes par des bouteilles. Pour qu'un philosophe puisse boire son verre, il a besoin de faire un cocktail avec toutes les bouteilles qui sont à coté. A la différence du problème précèdent, un philosophe peut avoir une ou plusieurs bouteilles voisines et non plus exactement deux, et une bouteille peut être partagée par plusieurs philosophes à la fois. Si un philosophe a besoin de boire et si un autre philosophe utilise une bouteille dont le premier a besoin, alors le premier doit attendre que le dernier finisse de boire pour pouvoir boire aussi. Donc le problème revient à utiliser une exclusion mutuelle sur toutes les bouteilles. Dans le problème des philosophes généralisé, un philosophe peut choisir une ou plusieurs bouteilles et pas toutes à la fois [74, 7]. Notre approche est probabiliste puisque le système utilisé est asynchrone et anonyme. Cette approche a été utilisée par Lehmann et Rabin [42] pour résoudre le problème des philosophes. Cet algorithme est la base de nombreuses études dont celle de Duflot et al[28].

Généralité

Tous les algorithmes qui résolvent ce problème doivent satisfaire les propriétés suivantes :

- Une ressource ne peut être partagée par deux processus en même temps, on dit que l'algorithme assure l'*exclusion mutuelle* des ressources partagées,
- si deux processus p_1 et p_2 n'ont pas de ressources communes, et donc n'ont pas une arête incidente, alors ils peuvent accéder à leurs ressources indépendamment et même simultanément. L'algorithme assure la propriété de *concurrence* (*concurrency*),
- si une ressource est demandée par deux processus p_1 et p_2 et si p_1 a formulé sa requête avant p_2 alors il doit avoir la ressource avant p_2. C'est la propriété de l'*ordonnancement* (*ordering*),

– si un processus a besoin de plusieurs ressources, il doit finir par les obtenir, ceci est appelé la propriété de *survie* (*liveness*).

Remarque 6.1.1. Il y a une similitude entre notre algorithme et l'algorithme de Lamport [40], sauf que notre algorithme traite le problème des conflits dans un système asynchrone et anonyme tandis que l'algorithme de Lamport traite le problème de l'exclusion mutuelle dans un graphe complet. En plus, l'approche de l'auteur est une approche de validation et de vérification alors que notre approche est une approche d'analyse d'algorithme.

6.1.2 Description de l'algorithme

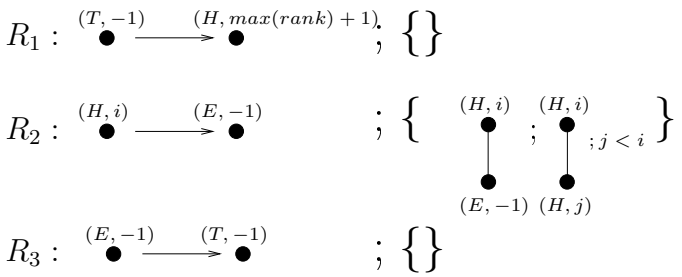
Dans cette sous-section, nous décrivons l'algorithme par un système de réécriture de graphe. Chaque processus peut avoir trois états : *tranquille (tranquil)*, *assoifé (thirsty)* ou *buvant (drinking)*. Ces états seront codés avec les étiquettes T, Th et D. Une autre étiquette est utilisée, cette étiquette est un entier représentant le rang de la requête. C'est ainsi que chaque noeud sera étiqueté par (X, i) avec $X \in \{T, Th, D\}$ et i un entier correspondant au rang.

Initialement, tous les noeuds du graphe sont dans l'état *tranquille*, cela se traduit par l'étiquette $(T, -1)$. A chaque étape de calcul, chaque noeud u à l'état *tranquille* peut demander l'accès à ses sections critiques, il devient alors *assoifé*. Dans ce cas, u change son état à $(Th, rank + 1)$ où $rank = max\{k_v | v \in B(u, 1)$ et $\lambda(v) = (X, k_v), X \in \{T, Th, D\}\}$. Cela signifie que l'ordre de u est le plus grand parmi l'ordre de tous ses voisins. Dans un graphe complet, cet ordre peut être vu comme un temps universel puisque seulement un seul noeud peut changer son état en Th. Dans ce cas, notre algorithme calcule l'exclusion mutuelle.

Si un noeud u, avec une étiquette (Th, i), n'a pas de voisin dans la section critique (avec l'étiquette $(D, -1)$) et n'a pas de voisin avec une étiquette (Th, j) avec $j < i$ (le rang de u est le plus petit comparé avec ses voisins), le noeud u peut entrer dans la section critique. Dans ce cas, u aura l'étiquette $(D, -1)$.

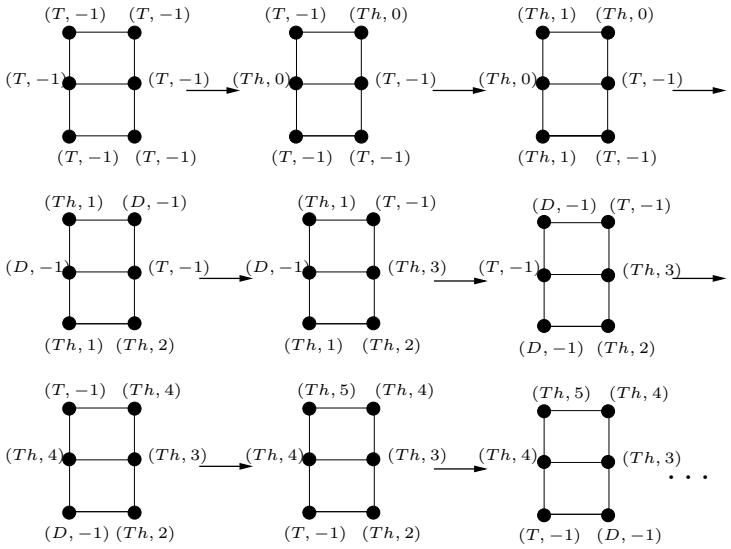
Une fois que le noeud dans la section critique a terminé, il retourne à un état *tranquille*. Son étiquette redevient $(T, -1)$.

L'algorithme peut être codé par le système de réécriture de graphe avec contextes interdits suivant $\mathcal{R}_{18} = (L, I, P)$ défini par $L = \{\{T, Th, D\} \times [-1..\infty]\}$, $I = \{(T, -1)\}$ et $P = \{R_1, R_2, R_3\}$ avec R_1, R_2 et R_3 sont les règles de réécriture avec contextes interdits suivantes :



L'exemple 6.1.1 illustre une exécution de ce système.

Exemple 6.1.1. Voici une séquence de réécriture du système $\mathcal{R}_{18}$.



Pour pouvoir implémenter le système de réécriture de graphe (SRG), nous avons besoin d'assurer qu'à chaque étape, deux noeuds voisins ne peuvent pas appliquer une des règles du système en même temps. Pour cela, nous avons utilisé une fonction probabiliste (LE1) donnée dans [55] et detaillée dans la section 5.5. Dans [55], les auteurs démontrent que si d est le degré d'un noeud v alors le temps d'attente moyen pour que v soit localement élu est de $d+1$.

6.2 Preuve de la correction

Avant de prouver la correction de notre algorithme, nous allons rappeler quelques propriétés du système de réécriture de graphe que nous avons décrit dans la section précédente.

Soit (G,λ) un graphe connexe et étiqueté tel que chaque noeud est étiqueté $(T,-1)$. Soit (G,λ') un graphe étiqueté tel que : $(G,\lambda) \xrightarrow[\mathcal{R}_{18}]{*} (G,\lambda')$. Alors le graphe (G,λ') satisfait :

- Chaque noeud u étiqueté $(T,-1)$ qui veut entrer dans la section critique (CS), change son étiquette à $(Th, rank+1)$, où $rank = max\{k_v | v \in B(u,1)$ et $\lambda(v) = (X,k_v), X \in$

$\{T, Th, D\}\}$,

- pour qu'un noeud *assoifé* (étiquette (Th, i)) change son état à *buvant* ($(D, -1)$), ne doit pas avoir de voisin avec l'étiquette $(D, -1)$ et pas de voisin étiqueté (Th, j) avec $j < i$.

Avec ces deux propriétés, nous pouvons assurer que

Lemme 6.2.1. *L'exclusion mutuelle et la propriété de l'ordonnancement sont vérifiées par l'algorithme.*

Pour la propriété de concurrence, la preuve donnée dans [55] stipule que si un noeud v est localement élu dans une boule $B(v, 1)$, aucun autre noeud dans $B(v, 1)$ ne sera localement élu, c'est-à-dire, v peut appliquer une des règles du système sans influencer les noeuds qui se trouvent à une distance plus grande que 1 de lui. Ce qui nous donne :

Lemme 6.2.2. *La propriété de concurrence est satisfaite par l'algorithme.*

Pour prouver la propriété de *survie*, nous avons besoin de prouver quelques lemmes vérifiés par le SRG qui code l'algorithme.

Définition 6.2.1. Soit $s > 0$ un entier positif et soit $\mathcal{H}_s$ un ensemble de noeuds étiquetés (Th, i) dans G à l'étape s de l'algorithme. Soit $P_k = (v_0, v_1, \cdots, v_{k-1})$ un chemin dans $\mathcal{H}_s$. P_k est un *chemin consécutif* si

$$\forall j \in [0..k-2] \quad \text{si } \lambda(v_j) = (Th, i) \text{ et } \lambda(v_{j+1}) = (Th, l) \text{ alors } l = i + 1.$$

Ce qui nous intéresse dans notre étude c'est le chemin consécutif car c'est le pire cas dans notre algorithme. En effet, si $P_k = (v_0, v_1, \cdots, v_{k-1})$ est un chemin consécutif, il est clair que v_{k-1} ne peut accéder aux ressources (donc changer son étiquette à $(D, -1)$) que si tous les noeuds v_0, v_1,$\cdots$, et v_{k-2} accèdent aux ressources dans cet ordre.

Un noeud dont l'état est *Tranquille* $(T, -1)$ doit être localement élu pour pouvoir changer son étiquette à (Th, i) et une deuxième fois pour vérifier qu'il a bien le rang le plus petit de ses voisins et qu'aucun de ses voisins n'est dans la section critique, c'est alors qu'il peut devenir $(D, -1)$. Si dans l'étape s un chemin consécutif $P_k = (v_0, v_1, \cdots, v_{k-1})$ est formé alors v_0, v_1, $\cdots$, et v_{k-1} ont été localement élus dans cet ordre. Suivant le même raisonnement, il est facile de voir que v_{k-1} ne peut changer son étiquette à $(D, -1)$ que si les noeuds v_0, v_1, $\cdots$, et enfin v_{k-1} sont localement élus dans cet ordre là.

Dans cette section, nous ne nous concentrons que sur le temps d'attente moyen nécessaire pour que le noeud $v = v_{k-1}$ change son étiquette à $(D, -1)$ car c'est le même temps nécessaire pour que P_k soit construit.

Un noeud dont l'étiquette est (Th, i) devra être localement élu pour pouvoir la changer en $(D, -1)$. Comme cela a été prouvé dans [55], le temps moyen d'attente pour un noeud v pour être localement élu dans la boule $B(v, 1)$ est

$$\mu(v) = d(v) + 1. \tag{1}$$

$d(v)$ étant le degré de v. Le temps moyen pour que le noeud v_{k-1} remplace son étiquette par $(D, -1)$ va donc être le temps moyen pour que les noeuds $v_0, v_1, \cdots, v_{k-2}$ soient localement élus.

Lemme 6.2.3. *Soit $P_k = (v_0, v_1, \cdots, v_{k-1})$ un chemin consécutif de longueur k. Si τ_k est le temps moyen pour que les noeuds v_0, v_1,... et v_{k-1} soient localement élus dans cet ordre alors*

$$E(\tau_k) = \sum_{i=0}^{k-1} (d(v_i) + 1). \tag{2}$$

Démonstration. Soit t_i une variable aléatoire définie par : $t_i = 1$ si v_i est localement élu, et $t_i = 0$ sinon. Il est évident que $\tau_k = \sum_{i=0}^{k-1} t_i$. Comme l'espérance mathématique est linéaire, nous avons

$$E(\tau_k) = \sum_{i=0}^{k-1} E(t_i) = \sum_{i=0}^{k-1} \mu(v_i) \tag{3}$$

alors avec (1), la preuve est faite. □

Comme $\sum_{i=0}^{k-1} d(v_i) \leqslant 2m$ (où m est le nombre d'arêtes), nous avons :

Corollaire 6.2.1.

$$2k - 2 \leqslant E(\tau_k) \leqslant 2m + k. \tag{4}$$

Remarque 6.2.1. Pour avoir une borne supérieure de $E(\tau_k)$ en fonction de la taille n du graphe, le graphe complet maximise m. Or k ne peut pas dépasser $n - 1$. Par conséquence nous avons

$$E(\tau_k) \leqslant \frac{n(n+1)}{2} + n - 1. \tag{5}$$

Le but du prochain lemme est de démontrer que l'utilisation de la fonction probabiliste assure la faible probabilité d'avoir un *long* chemin consécutif.

Lemme 6.2.4. *Soit v un noeud de G et soit $P = (v_0 = v, v_1, \cdots, v_{k-1})$ un chemin consécutif de longueur k. Avec une probabilité $1 - \frac{d(v)+1}{\alpha}$ nous avons $k \leqslant \alpha$, $\forall \alpha > d_{max} + 1$, où d_{max} est le degré maximal du graphe G*

Démonstration. Nous supposons que $\lambda(v) = (Th, i), \lambda(v_1) = (Th, i+1), \cdots, \lambda(v_{k-1}) = (Th, i+k-1)$. Avoir une telle chaîne signifie que les noeuds v_0, v_1, ..., et v_{k-1} sont localement élus dans cet ordre et que v n'a pas été localement élu durant au moins k étapes.

Soit t une variable aléatoire qui compte le nombre d'étapes pour que v soit localement élu.

En utilisant l'inégalité de Markov [31] et (1), nous avons

$$\mathbb{P}r(t > \alpha) \leqslant \frac{\mu(v)}{\alpha} = \frac{d(v)+1}{\alpha}, \ \forall \alpha > d_{max} + 1.$$

□

La borne sur la probabilité pour qu'un chemin consécutif soit long est faible si α est assez grand. En effet, pour les graphes qui ont un degré borné, principalement s'il est dans $O(\log n)$, il est suffisant de prendre $\alpha = \log n$ pour obtenir une probabilité de l'ordre de $1 - O(\frac{1}{\log n})$, donc qu'il n'y ait pas de chemin consécutif de longueur supérieure à $\log n$.

D'où le théorème principal suivant :

Théorème 6.2.1. *L'algorithme assure la propriété de survie.*

6.3 Implémentation

L'algorithme décrit précédemment a été implémenté sur la plate-forme Visidia. Le système $\mathcal{R}_{18}$ décrit précédemment est implémenté dans l'algorithme 12.

6.3.1 Résultats expérimentaux

Dans cette section nous présentons quelques résultats expérimentaux concernant l'algorithme distribué présenté auparavant. Le but de notre expérimentation est de valider l'analyse probabiliste que nous avons évoquée dans la section précédente. Nous nous concentrons sur la mesure expérimentale de la propriété de *survie*. Plus précisément, nous calculons le temps moyen d'attente pour qu'un processeur puisse obtenir les ressources, i.e. le temps moyen qu'un noeud avec l'état Th va attendre pour pouvoir passer à l'état D. L'unité de temps que nous utilisons dans l'expérimentation est l'étape invoquée dans la section 6.1.2. Dans le tableau suivant, nous présentons le temps moyen d'attente expérimental obtenu par Visidia pour quelques classes de graphes.

Graphe	**Taille**	**Temps moyen d'attente**
Chaîne	30	4.54
Arbre	100	8.68
Anneaux	30	4.83
Anneaux	100	8.51
Grille	30	9.76
Grille	100	19.70
Graphe complet	30	430.86
Graphe aléatoire	30	12.01

6.4 Exclusion Mutuelle dans un arbre

Nous nous intéressons à l'exclusion mutuelle, où il n'y a qu'une seule ressource à partager entre tous les processus. Ce problème a été traité par Naïmi et Tréhel dans [66] et aussi par K. Raymond [68]. Cet algorithme construit une arborescence sur l'ensemble des noeuds

```
while(run){
      if ((mEtat=="D") && (finiBoire())){ // Appliquer la règle R3.
         mEtat ="T";
       }
       synchro=LC1(); // Fonction aléatoire pour élire un centre
       if (synchro == starCenter ){ // synchro est l'élu, il peut appliquer une règle
          for(door=0;door<arity;door++) {
             etatVoisin[door] = receiveFrom(door);
             rangVoisin[door] = receiveFrom(door);
          }
          if ((mEtat=="T") && (besoinBoire())) { // Appliquer la règle R1.
             mEtat = "Th";
             mRang=rangMax+1;
          }
                if ((mEtat=="Th") && (!existD(etatVoisin)) &&
(mRang<=rangMin(rangVoisin))) { // Appliquer la règle R2.
             mEtat="D";
             mRang=-1;
             commenceBoire();
          }
          breakSynchro();
       }
       else
          if (synchro==starBorder) {
             // les voisins d'un centre lui envoient leur état et leur rang.
             sendTo(synchro,mEtat);
             sendTo(synchro,mRang);
          }
}
```

Algorithme 12: Résolution des conflits

en privilégiant un sommet-racine et en orientant les arêtes vers cette racine. Les caractéristiques de l'arborescence sont les suivantes : chaque noeud connaît son successeur dans l'arborescence et toute demande faite par un noeud transite le long d'un chemin menant à la racine. La file d'attente est organisée en sorte que le premier élément est le noeud qui possède le privilège, les éléments suivants étant les demandeur en attente. Chaque noeud connaît son successeur dans la file, s'il en existe un. Quand un noeud sort de la section critique, il transmet le privilège au noeud qui le suit dans la file.

Remarque 6.4.1. Une analyse de l'algorithme de Naïmi et Tréhel [66] a été faite par Lavault [41].

6.4.1 Description de l'algorithme de l'exclusion mutuelle dans un arbre

Dans ce qui suit nous allons décrire le fonctionnement de l'algorithme à l'aide des systèmes de réécriture de graphes. Nous commençons par décrire les états qu'un noeud peut avoir durant l'exécution de l'algorithme. Il y a trois états principaux qui transcrivent le besoin ou non de la ressource. Ces états sont *tranquille*, *besoin* et *accède*. Le premier état est celui de l'état *tranquille* (noté T), un noeud qui n'a pas besoin d'accéder à la ressource a cet état. Le second, noté H, est celui du *besoin*, un noeud qui a besoin de la ressource change d'état et devient H. Le dernier est celui d'un noeud qui est dans la ressource. Cet état est symbolisé par l'étiquette E. Ces états suivent un cycle bien déterminé qui est *tranquille* puis *besoin* puis *accède* et enfin *tranquille*. Un noeud ne peut sauter aucune étape. Mais pour cela un noeud a besoin d'un *jeton*, noté To. Dans tout le graphe il n'y a qu'un seul jeton. Tous les autres sommets, qui n'ont pas de jeton, ont l'étiquette NT. Une dernière étiquette apparaît, cette étiquette n'est qu'un simple compteur. Initialement elle vaut 0. Si un noeud a besoin de la ressource, il augmente cette étiquette de 1 ainsi, son père peut demander la ressource à celui qui la détient via son propre père. Comme nous travaillons sur des arbres, nous attribuons à chaque arête une orientation et un entier. La direction indique le père de chaque noeud et se termine par la racine. La racine est le noeud qui possède le jeton. Donc la racine n'est pas fixe, elle change selon celui qui possède le jeton et à tout instant, les arêtes de l'arbre sont orientées vers cette racine. L'entier est utilisé pour donner des priorités au noeud qui ont demandé la ressource.

Initialement, tous les noeuds du graphe sont à l'état *tranquille* et sans jeton (leur étiquette est $(T, NT, 0)$) sauf la racine qui possède le jeton qui est étiquetée $(T, To, 0)$. Toutes les arêtes sont étiquetées par 0 et elles sont orientées vers leur père. A chaque étape du calcul un noeud étiqueté (T, H, c) peut demander l'accès à la ressource, son étiquette devient alors $(H, Y, c+1)$, avec $Y \in \{To, NT\}$. Si un noeud possède le jeton et s'il a besoin de la ressource (son étiquette est (H, To, c')) il peut l'utiliser et son étiquette change en (E, To, c'). Une fois qu'il a fini, il revient à l'état *tranquille* et diminue son compteur de 1 $((T, To, c' - 1))$.
Un noeud u qui s'aperçoit que son fils a besoin de la ressource, que le compteur de celui-ci est diffèrent de 0 et que l'étiquette de l'arête adjacente est égale à 0, incrémente son compteur de 1 et change l'étiquette de l'arête en $i = max\{k_v | v \in B(u, 1) \text{ et } \lambda(u, v) = k_v\} + 1$. Ainsi, l'étiquette de cette arête prend la valeur du maximum des étiquettes des autres arêtes adjacentes à u incrémenté de 1. Cette étiquette peut être vue comme le rang de chaque noeud, c'est celui qui a la valeur minimale hors 0 qui aura le jeton le premier.
Si le noeud u qui détient le jeton est dans l'état tranquille et que son compteur c n'est pas égale à 0 $((T, To, c))$, il donne le jeton au noeud v dont l'arête adjacente a l'étiquette minimale hors 0. Dans ce cas, u aura l'étiquette $(T, NT, c-1)$, v aura l'étiquette (X, To, c') et l'orientation de l'arête adjacente changera de sens et aura la valeur 0. L'arbre dans ce cas changera de racine.

L'algorithme qu'on vient de décrire peut être codé par les systèmes de réécriture de

graphes avec contextes interdits comme suit $\mathcal{R}_{19} = (L, I, P)$ défini par $L = \{\{T, H, E\} \times \{To, NT\} \times [0..\infty] \cup \{[0..\infty] \times \{\vee, \wedge\}\}\}$, $I = \{\{(T, NT, 0), (T, To, 0)\} \cup \{(0, \wedge)\}\}$ et $P = \{R_1, R_2, R_3, R_4, R_5\}$ avec R_1, R_2, R_3, R_4 et R_5 sont les règles de réécriture avec contextes interdits suivantes :

R_1 : $(T, Y, c) \longrightarrow (H, Y, c+1)$; $\{\}$

R_2 : $(X, Y, c) \longrightarrow (X, Y, c+1)$; $c' > 0$; $\{\}$
0 : (X', NT, c') — $max(\lambda(E)) + 1$: (X', NT, c')

R_3 : $(T, To, c) \longrightarrow (T, NT, c-1)$; $c', i > 0$; $\{ (T, To, c) \xrightarrow{j} (X', NT, c'') \ ; j < i \ et j \neq 0 \}$
i : (X, NT, c') — 0 : (X, To, c')

R_4 : $(H, To, c) \longrightarrow (E, To, c)$; $\{\}$

R_5 : $(E, To, c) \longrightarrow (T, To, c-1)$; $\{\}$

; Avec $X, X' \in \{T, H, E\}$, $Y, Y' \in \{To, NT\}$.

6.4.2 Preuve de la correction

Nous rappelons les propriétés principales pour les problèmes de partages des ressources. Ces propriétés sont différentes de celles de la section précédentes puisque dans cette section, on ne partage qu'une seule ressource. Pour démontrer la correction de notre système, il faut prouver les propriétés suivantes :

- *exclusion mutuelle* : Dans notre cas, un seul processus peut être dans la section critique,
- *ordonnancement* : cette propriété est limitée dans notre cas aux branches des arbres. Nous ne pouvons pas assurer l'ordonnancement de tous les processus, seulement au niveau des processus pères. Si un processus père p possède deux processus fils f_1 et f_2 et si le processus f_1 a demandé la ressource avant f_2 alors f_1 accédera à la ressource avant f_2,
- *survie* : un processus peut avoir la ressource autant de fois qu'il veut.

Soit (G, λ) un arbre connexe et étiqueté tel que chaque noeud est étiqueté $(T, NT, 0)$ sauf la racine qui est étiquetée par $(T, To, 0)$ et les arêtes sont toutes orientées des fils vers les pères et sont étiquetées par 0. Soit (G, λ') un graphe étiqueté tel que : $(G, \lambda) \xrightarrow[\mathcal{R}_{19}]{*} (G, \lambda')$. Alors le graphe (G, λ') satisfait :

1. Chaque noeud u étiqueté (T, Y, c) qui veut entrer dans la section critique, change son étiquette en $(H, Y, c+1)$, où $Y \in \{To, NT\}$,
2. Il n'y a qu'un seul noeud dans le graphe qui possède le jeton. Ce noeud est étiqueté par (X, To, c) où $X \in \{T, H, E\}$,

3. Un jeton ne passe d'un noeud u à un noeud v que si u ne possède pas d'arêtes dont l'étiquette est différente de 0 et inférieure à l'étiquette de l'arête entre u et v,
4. Un noeud qui est dans l'état d'attente de la ressource et qui possède le jeton $((H, To, c))$ peut entrer dans la section critique et devient (E, To, c).

Démonstration. La preuve des propriétés 1 et 4 sont triviales puisque c'est l'application simple des règles R_1 et R_4 qui ne possèdent pas de contextes interdits d'où leur application immédiate.
Pour les propriété 2 et 3, elles sont issues de la règle R_3 puisque pour la propriété 2, le sommet u qui possède le jeton et qu'il est dans l'état tranquille (T, To, c) le donne au noeud v étiqueté (X, NT, c'). Ce dernier, le noeud v, changera son étiquette à (X, To, c'), c'est alors lui qui a le jeton, et le noeud u n'ayant plus le jeton devient $(T, NT, c-1)$. Pour les autres règles, aucun changement n'intervient au niveau du jeton. La propriété 3 est assurée par les contextes interdits de la même règle. On ne peut appliquer cette règle que si les contextes interdits ne sont pas présents dans la localité du sous-graphe. □

Avec ces propriétés, nous pouvons assurer que

Lemme 6.4.1. *L'exclusion mutuelle et la propriété de l'ordonnancement sont vérifiées par l'algorithme.*

Lemme 6.4.2. *L'orientation du graphe pointe vers le noeud qui possède le jeton.*

Démonstration. Initialement, toutes les arêtes sont orientés du noeud fils vers le noeud père et la racine possède le jeton, donc, la propriété est initialement vraie. Supposons qu'après k étapes la propriété est toujours juste, montrons qu'elle l'est à l'étape $k+1$.
A l'étape k, l'orientation du graphe pointe vers le noeud avec le jeton. Toutes les arêtes de ce noeud sont orientées vers lui. Si une des règles de réécriture R_1 ou R_2 ou R_4 ou R_5 est appliquée, la propriété reste exacte puisque aucune application de ces règles ne change ni l'orientation des arêtes ni la possession du jeton. C'est seulement la règle $R3$ qui modifie l'orientation et le détenteur du jeton. Soit u le noeud qui a le jeton donc avec l'étiquette (T, To, c) et soit v un noeud voisin de u qui satisfasse la contrainte de l'application de la règle R_3. Avant l'application de la règle, l'arête entre u et v, noté e est orienté de v vers u et toutes les autres arêtes incidentes à u pointent vers lui. En appliquant la règle R_3, le jeton va passer du noeud u au noeud v et l'arête e va pointer vers v. D'après l'hypothèse d'induction, à l'étape k, les arêtes du noeud v sont pointées vers lui sauf une seule arête (e) qui est pointée vers u. A l'étape $k+1$, seule l'arête e a changé d'orientation d'où toutes les arêtes du noeud v sont pointées vers lui. D'autre part, les noeuds qui pointent vers u à l'étape k pointent vers v à l'étape $k+1$ puisque u pointe elle même vers v. La propriété est exacte à l'étape $k+1$ d'où le résultat. □

Pour démontrer la propriété de *survie*, nous allons considérer le pire cas qu'on peut avoir. Ce cas arrive dans une chaîne où une extrémité possède le jeton et l'autre extrémité a besoin de la ressource. Dans ce qui suit, nous allons utiliser les propriétés de l'élection

locale dans une boule de rayon 2 ($LE2$) démontrés dans [55]. Dans cet article, les auteurs ont démontré que le temps d'attente pour un noeud u, noté $\mu(u)$, est égale à $N_2(u)$ avec $N_2(u)$ le nombre de noeud dans la boule de centre u et de rayon 2.

Lemme 6.4.3. *Soit τ_k le temps moyen pour qu'une information parte d'un noeud u à un noeud v distant de k, avec $k \leqslant D$ le diamètre du graphe, alors : $E(\tau_k) = \sum_{i=0}^{k-1} N_2(v_i)$.*

Démonstration. Soit $C_{u,v} = (u_0 = u, u_1, \ldots, u_{k-2}, u_{k-1} = v)$ le chemin dans un arbre entre les noeuds u et v. Pour qu'une information parte d'un noeud u à un noeud v, il faut que les noeuds dans $C_{u,v}$ soient localement élus dans cet ordre là.

D'après la section précédente, nous avons :

$$E(\tau_k) = \sum_{i=0}^{k-1} \mu(v_i) \tag{6}$$

comme nous utilisons une élection locale de rayon 2, nous avons :

$$E(\tau_k) = \sum_{i=0}^{k-1} N_2(v_i) \tag{7}$$

□

Lemme 6.4.4. *Le temps moyen pour qu'une information arrive d'un noeud u à un noeud v distant de k est majorée selon les cas suivants :*

- *Dans une étoile, $E(\tau_k) \leqslant 3 * n$.*
- *Dans une chaîne, $E(\tau_k) \leqslant 5 * k$.*
- *Dans un arbre Δ-régulier, $E(\tau_k) < k * (\Delta^2 + 1)$.*
- *Dans un arbre à degré borné par Δ, $E(\tau_k) < k * (\Delta^2 + 1)$.*
- *Dans une chaîne d'étoiles de degré Δ, $E(\tau_k) \leqslant k * (3 * \Delta - 1)$.*

Démonstration.

- Dans une étoile, le diamètre étant limité à 2, la somme (7) n'est que celle relative à trois sommets : le sommet centre et les deux extrémités u et v. Donc,

$$E(\tau_k) = \sum_{i=0}^{2} N_2(v_i) \leqslant 3 * n$$

– Dans une chaîne, le nombre de noeuds maximum dans une boule de rayon 2 est de 5. Donc nous avons :

$$E(\tau_k) = \sum_{i=0}^{k-1} N_2(v_i) \leqslant \sum_{i=0}^{k-1} 5$$
$$\text{i.}e. E(\tau_k) \leqslant 5 * k$$

– Dans un arbre Δ-régulier, le temps d'attente d'un noeud u, $N_2(u)$, est inférieur ou égal à $\Delta^2 + 1$. D'où,

$$E(\tau_k) = \sum_{i=0}^{k-1} N_2(v_i) < \sum_{i=0}^{k-1} (\Delta^2 + 1) < k * (\Delta^2 + 1)$$

– La même démonstration pour le cas d'un arbre à degré borné par Δ puisque $N_2(u)$ ne peut pas dépasser $\Delta^2 + 1$ pour chaque noeud u. d'où la borne supérieure.

– Dans une chaîne d'étoiles de degré Δ, soit u un noeud au centre du graphe. u possède Δ voisins dont deux ont aussi Δ voisins et les autres n'ont qu'un seul voisin qui est u.

$$N_2(u) = (\Delta + 1) + (\Delta - 1) + (\Delta - 1) = 3 * \Delta - 1$$

Donc nous avons :

$$E(\tau_k) = \sum_{i=0}^{k-1} N_2(v_i) \leqslant \sum_{i=0}^{k-1} (3 * \Delta - 1) \leqslant k * (3 * \Delta - 1)$$

□

Pour que le jeton arrive à un noeud, il suit un chemin bien défini et il peut repasser par un noeud plusieurs fois.

Lemme 6.4.5. *Le temps moyen maximum, $\overline{M}(G)$, qu'un noeud attend pour recevoir le jeton est égale à $\sum_{i=0}^{n-1}(N_2(v_i) * d(v_i))$.*

Démonstration. Le pire des cas est lorsque le chemin du jeton passe par tous les noeuds et finit par arriver au noeud qui nous intéresse. Donc le jeton passe par chaque noeud u $d(u)$ fois. Or le temps moyen pour qu'un noeud soit localement élu pour $LE2$ est de $N_2(u)$. D'où le temps moyen maximale est $\sum_{i=0}^{n-1}(N_2(v_i) * d(v_i))$. □

Dans ce qui suit nous allons étudier quelques cas particulier d'arbres pour maximiser cette somme. Le pire cas est une chaîne d'étoiles.

Lemme 6.4.6. *Dans une étoile, $\overline{M}(G) \leqslant 2 * n * (n - 1)$.*
Dans une chaîne, $\overline{M}(G) \leqslant (10 * n - 18)$.
Dans un arbre Δ-régulier, $\overline{M}(G) < n * \Delta * (\Delta^2 + 1)$.
Dans un arbre à degré borné par Δ, $\overline{M}(G) < n * \Delta * (\Delta^2 + 1)$.
Dans une chaîne d'étoile de degré Δ, $\overline{M}(G) < n * \Delta * (3 * \Delta - 1)$.

Lemme 6.4.7. *Le temps d'attente globale d'un noeud, appartenant à un graphe quelconque, de la demande jusqu'à la réception de la ressource est de l'ordre de $O(n^3)$.*

Démonstration. Dans notre preuve, nous considérons le graphe complet comme le pire des cas car il maximise l'espérance $\overline{M}(G)$.

$$\overline{M}(G) = \sum_{i=0}^{n-1} d(v_i) * N_2(v_i) = \sum_{i=0}^{n-1} n * n = n^3$$

Dans le même cas, l'espérance $E(\tau_k)$ étant majorée par $\overline{M}(G)$, la somme des deux espérances est inférieure à $2 * n^3$. D'où le fait que le temps d'attente globale est de l'ordre de $O(n^3)$. □

Dans le cas des arbres le pire des cas constaté est celui de l'arbre Δ-régulier. Dans ce cas, le temps d'attente globale est inférieur à $(n * \Delta + D) * (\Delta^2 + 1)$.

D'où le théorème suivant :

Théorème 6.4.1. *L'algorithme assure la propriété de survie.*

6.5 Implémentation

L'algorithme décrit précédemment a été implémenté sur la plate-forme Visidia. Le système $\mathcal{R}_{19}$ décrit précédemment est implémenté dans l'algorithme 13. Mais comme Visidia ne traite pas les graphes orientés, nous exécutons un petit algorithme sur l'arbre pour pouvoir orienter les arêtes vers la racine. Pour cela, la racine envoie un message 1 à tous ses voisins. Un noeud qui reçoit 1 d'un lien de communication marque ce lien comme celui de son père et envoie le même message à ses voisins. Si le noeud, qui a déjà marqué son père, est une feuille de l'arbre ou bien s'il a reçu le message 0 de tous ses voisins sauf de son père alors il envoie le message 0 à son père. l'algorithme finit lorsque le noeud racine reçoit le message 0 de tous ses voisins. Dès lors, l'algorithme encodé par le système $\mathcal{R}_{19}$ peut commencer.

6.5.1 Résultats expérimentaux

Dans cette section nous présentons quelques résultats expérimentaux concernant l'algorithme distribué présenté auparavant. Le but de notre expérimentation est de valider l'analyse probabiliste que nous avons évoquée dans la section précédente. Nous nous concentrons sur la mesure expérimentale de la propriété de *survie*. Plus précisément, nous calculons le temps moyen d'attente pour qu'un processeur puisse obtenir les ressources, i.e. le temps moyen qu'un noeud avec l'état H va attendre pour pouvoir passer à l'état E. L'unité de temps que nous utilisons dans l'expérimentation est l'étape invoquée dans la section 6.4.1.

```
while(run){
    if ((mEtat=="E") && (finiManger())){ // Appliquer la règle R5.
        mEtat ="T";
        mCount =mCount-1;
     }
    if ((mEtat=="T") && (besoinManger())){ // Appliquer la règle R1.
        mEtat ="H";
        mCount =mCount+1;
     }
    if ((mEtat=="H") && (jeton)){ // Appliquer la règle R4.
        mEtat ="E";
     }
     synchro=LC2(); // Fonction aléatoire pour élire un centre
     if (synchro == starCenter ){ // synchro est l'élu, il peut appliquer une règle
        for(door=0;door<arity;door++) {
            etatVoisin[door] = receiveFrom(door);
            jetonVoisin[door] = receiveFrom(door);
            countVoisin[door] = receiveFrom(door);

            if ((etatArete[door] == 0) && (countVoisin[door]>0)) { // Appliquer la règle R2.
                etatArete[door]=max(etatArete)+1;
                mCount=mCount+1;
            }
        }
        if ((mEtat=="T") && (jeton)) { // Appliquer la règle R3.
            pos=min(etatArete);                    if (pos != -1) {
                jeton=false;
                mCount=mCount-1;
                sendTo(pos,token);
                pere=pos;                  }
        }
        breakSynchro();
    }
    else
        if (synchro==starBorder) {
            // les voisins d'un centre lui envoient leur état et leur rang.
            sendTo(synchro,mEtat);
            sendTo(synchro,jeton);
            sendTo(synchro,mCount);
            if (receiveFrom(synchro) == token) { // règle R3
                pere=-1;
                jeton=true;
            }
        }
}
```

Algorithme 13: Exclusion mutuelle dans un arbre

Dans le tableau suivant, nous présentons le temps moyen d'attente expérimental obtenu par Visidia pour quelques classes d'arbre.

Graphe	Taille	Temps moyen d'attente
Chaîne	30	4.54
Arbre aléatoire	20	132.77
Arbre 4-régulier de diamètre 5	484	5695.05
Arbre 5-régulier de diamètre 5	1706	23468.73
Arbre aléatoire	100	1225.18

Nous remarquons que le temps de calcul pour les résultats expérimentaux est très inférieur à celui des résultats théoriques. Pour comparer, le temps moyen théorique pour une chaîne de 30 noeuds est inférieur ou égal à 1740, ce qui est largement au dessus du temps moyen expérimental de 4.54. Cela est dû au fait que la théorie est basée sur le pire des cas alors que les expérimentations sont basées sur le cas moyen.

6.6 Marche aléatoire pour résoudre l'exclusion Mutuelle

6.6.1 Introduction

Nous nous intéressons à la marche aléatoire pour résoudre le problème de l'exclusion mutuelle. Nous utilisons deux modèles de la marche aléatoire : le modèle classique et le modèle biaisé. Il y a beaucoup de chercheurs qui ont travaillé sur l'exclusion mutuelle avec jeton mais ils utilisent en général des hypothèses sur le graphe (graphe cyclique) ou bien ils supposent l'existence d'un chemin connu par le jeton .... Dans notre cas, nous voulons utiliser notre environnement sans aucune hypothèse.

6.6.2 Marche aléatoire

Description de l'algorithme de l'exclusion mutuelle

Le terme de "marche aléatoire", [1], désigne le plus souvent un processus stochastique en temps discret. Une marche aléatoire est la trajectoire d'une particule qui se déplace d'un sommet vers un voisin de manière aléatoire. Un jeton a la même probabilité de passer d'un noeud à un de ses voisins ou bien de rester dans le même sommet. Si un noeud u a $d(u)$ voisin, chaque noeud dans la boule $B(u,1)$, donc y compris u, a la même probabilité de recevoir le jeton qui est égale à $\frac{1}{(d(u)+1)}$. Nous utilisons cette méthode pour résoudre le problème de l'exclusion mutuelle tel que le sommet qui possède le jeton peut entrer dans la section critique, s'il en a besoin. Sinon il donne le jeton à un de ses voisins ou bien il le garde [37].

Le système de réécriture de graphe de cet algorithme est le système $\mathcal{R}_{20} = (L, I, P)$

défini par $L = \{\{T, H, E\} \times \{To, NT\}\}$, $I = \{(T, To), (T, NT)\}$, et $P = \{R_1, R_2, R_3, R_4\}$ où R_1, R_2, R_3 et R_4 sont les règles de réécriture suivantes :

R_1 : $(T, XT) \longrightarrow (H, XT)$ Avec $XT \in \{To, NT\}$, $X, X', Y \in \{T, H\}$

R_2 : $(H, To) \longrightarrow (E, To)$

R_3 : $(E, To) \longrightarrow (T, To)$

R_4 : $(T, To) \longrightarrow (T, NT)$, (X', To)

Les règles R_1, R_2 et R_3 sont les règles classiques pour résoudre le problème de l'exclusion mutuelle. Un noeud Tranquille étiqueté par T s'il a besoin de la ressource devient étiqueté H (la règle R_1). Si un noeud étiqueté H possède le jeton, il peut utiliser la ressource et devient étiqueté E (la règle R_2). Lorsque ce noeud fini d'utiliser la ressource il redevient *Tranquille* (étiqueté T) pour cela il applique la règle R_3. La règle R_4 correspond à l'algorithme de marche aléatoire. Puisque le jeton va passer d'un noeud u à un noeud voisin v choisit aléatoirement.

Plusieurs recherches ont été faites sur les marches aléatoires mais pour ce qui nous concerne les résultats des deux articles, [29, 30] nous intéressent le plus.

Théorème 6.6.1. *[29][Borne inférieure] Pour tout graphe G de taille n, et quelque soit le noeud u possédant le jeton initialement nous avons :*

$$E_u[G] \geqslant n \ln n + o(n \ln n).$$

Théorème 6.6.2. *[30][Borne supérieure] Pour tout graphe G de taille n, le temps de couvrement ; c'est à dire que met le jeton pour passer par tous les sommets (ECC) satisfait :*

$$ECC \leqslant \frac{4}{27} n^3 + O(n^{5/2}).$$

6.6.3 Marche aléatoire biaisée

Description de l'algorithme de l'exclusion mutuelle

Dans la section 6.4 nous avons traité le cas de l'exclusion dans un arbre. Nous pouvons utiliser cet algorithme dans un graphe quelconque en utilisant un algorithme de calcul d'arbre recouvrant ensuite nous exécutons l'algorithme de l'exclusion mutuelle sur l'arbre déjà construit.

Dans ce qui suit, nous décrivons un algorithme qui utilise un jeton qui passe par tous les sommets du graphe sans avoir un chemin définie ni un graphe particulier. C'est le même principe que pour la marche aléatoire mais en ajoutant une stratégie pour minimiser le temps de passage du jeton à tous les sommets du graphe, ce problème est un cas particulier de problème de la marche aléatoire biaisée [5]. Tout ce dont nous avons besoin est que chaque noeud connaît son degré. Pour résoudre le problème de l'exclusion mutuelle, nous avons besoin des trois états qui ont été définis dans la section précédente. Nous introduisons aussi la notion de jeton avec les étiquettes To et NT pour dire possède le jeton ou non. Enfin un réelle, représenté par une étiquette, qui calcul le nombre de fois que le jeton est passé par ce noeud par rapport à son degré. Un noeud qui a le jeton peut entrer dans la section critique. Ensuite le jeton passe vers le noeud dont le réel est le plus petit par rapport aux noeuds voisins du possesseur du jeton.

Le système de réécriture de graphe de cet algorithme est le système $\mathcal{R}_{21} = (L, I, P)$ défini par $L = \{\{T, H, E\} \times [0..\infty] \times \{To, NT\}\}$, $I = \{(T, 0, To), (T, 0, NT)\}$, et $P = \{R_1, R_2, R_3, R_4\}$ avec R_1, R_2, R_3 and R_4 sont les règles de réécriture suivantes :

R_1 : $(T, nt, XT) \longrightarrow (H, nt, XT)$; $\{\}$

R_2 : $(H, nt, To) \longrightarrow (E, nt, To)$; $\{\}$

Avec $XT \in \{To, NT\}$
$X, X', Y \in \{T, H\}$
$nt, nt', m \in [0..\infty]$

R_3 : $(E, nt, To) \longrightarrow (T, nt, To)$; $\{\}$

R_4 : $\begin{matrix}(X, nt, To) \\ \bullet \\ | \\ \bullet \\ (X', nt', NT)\end{matrix} \longrightarrow \begin{matrix}(X, nt + 1/d, NT) \\ \bullet \\ | \\ \bullet \\ (X', nt', To)\end{matrix}$; $\left\{ \begin{matrix}(X, nt, To) \\ \bullet \\ | \\ \bullet \\ (Y, m, NT)\end{matrix} ; m < nt' \right\}$

Remarque 6.6.1. On note que d est le degré du noeud.

6.6.4 preuve de la correction

Lemme 6.6.1. *Soit (G, λ) un graphe connexe étiqueté tel que tous les noeud ont l'étiquette $(T, 0, NT)$ sauf le noeud distingué, qui possède le jeton, a l'étiquette $(T, 0, To)$. Soit (G, λ') un graphe étiqueté tel que : $G(V, E, \lambda) \xrightarrow[\mathcal{R}_{21}]{*} G(V, E, \lambda')$. Alors le graphe $G(V, E, \lambda')$ satisfait :*

1. *Tout noeud peut demander la section critique et il change son état de T à H.*
2. *Le noeud (E, nt, To) qui sort de la section critique change son étiquette à (T, nt, To).*
3. *Seules les noeuds qui possèdent le jeton peuvent entrer dans la section critique.*

Démonstration. La preuve de ces propriétés est l'exécution des règles R_1 et R_3 et R_2 pour la dernière propriété. □

Avec ces propriétés le lemme suivant est démontré :

Lemme 6.6.2. *L'exclusion mutuelle est vérifiée par l'algorithme.*

Deux propriétés essentielles restent à vérifier, la propriété de concurrence et de survie. Pour la première, nous avons besoin de deux lemmes pour la démontrer. Le premier consiste à démontrer qu'il n'y a qu'un seul jeton dans le graphe et le second à dire que seul un noeud peut entrer dans la section critique.

Lemme 6.6.3. *Il n'y a qu'un seul noeud qui possède le jeton et ce noeud est étiqueté* (X, nt, To) *et les autres noeuds sont étiquetés* (X', nt', NT)*, avec* $X \in \{T, H, E\}$ *et* $X' \in \{T, H\}$.

Démonstration. Pour démontrer ce lemme, nous n'avons besoin que d'utiliser une seule règle de réécriture qui est la règle R_4. Puisque les autres règles (R_1, R_2 et R_3) ne modifient pas l'emplacement du jeton.
Initialement, il n'y a qu'un seul noeud qui possède le jeton. Supposant que c'est vrai à l'étape k et montrons que ça reste toujours vrai à l'étape $k+1$. A l'étape k, il n'y a qu'un seul noeud dans le graphe qui a le jeton. Soit u ce noeud. Comme nous l'avons déjà dit, seule l'application de la règle R_4 nous intéresse. L'application des autres règles conserve l'hypothèse. Si on applique la règle R_4 entre le noeud u et le noeud v, le jeton passe du noeud u au noeud v. Donc il n'y a qu'un seul noeud avec le jeton qui est le noeud v. D'où l'hypothèse est toujours vrai. □

Lemme 6.6.4. *Il existe au plus un sommet dans la section critique.*

Démonstration. D'après le lemme précédent, il n'existe qu'un seul jeton dans le graphe et d'après la troisième propriété du système, seuls les noeuds qui possèdent le jeton peuvent entrer en section critique. D'où le résultat. □

Dans [5], une étude a été faite sur une marche aléatoire biaisée telle que avec une probabilité de $(1-\epsilon)$, le jeton suit le même principe que pour la marche aléatoire et avec une probabilité ϵ, le jeton passe à un noeud voisin bien défini.

Théorème 6.6.3. *[5][Borne inferieure] Soit* $G = (V, E)$ *un graphe connexe,* $S \subset V$*,* $v \in S$ *et* $x \in V$*. Soit* $\Delta(x, v)$ *la longueur du plus petit chemin entre le noeud* x *et* v *dans* G *et* $\Delta(x, S) = min_{v \in S}\Delta(x, v)$*. Soit* $\beta = 1 - \epsilon$*. Il existe une stratégie pour laquelle la probabilité stationnaire à* S *est au moins*

$$\frac{\sum_{v \in S} d(v)}{\sum_{v \in S} d(v) + \sum_{x \notin S} \beta^{\Delta(x,S)-1} d(x)}.$$

Théorème 6.6.4. *[5][Borne superieure] Si* $G = (V, E)$ *un graphe connexe de degré maximum* Δ*. Alors pour toute stratégie, la probabilité stationnaire pour tout sommet est au plus* $n^{c\epsilon-1}$ *pour une constante* c *dépendant uniquement de* Δ.

6.7 Implémentation

L'implémentation des deux algorithmes décrits précédemment est la même. La seule différence réside sur l'application de la règle R_4 où dans le système $\mathcal{R}_{20}$ la fonction *randomChoice* choisit aléatoirement un noeud voisin du centre par contre la fonction *randomChoiceMin* du système $\mathcal{R}_{21}$ choisit aléatoirement un voisin de l'ensemble des voisins tel que leur compteur est le minimum des autres.

```
while(run){
    if ((mEtat=="T") && (besoinManger())){ // Appliquer la règle R1.
        mEtat ="H";
    }
    if ((mEtat=="H") && (jeton)){ // Appliquer la règle R2.
        mEtat ="E";
    }
    if ((mEtat=="E") && (finiManger())) { // Appliquer la règle R3.
        mEtat ="T";
    }
    synchro=LC2(); // Fonction aléatoire pour élire un centre
    if (synchro == starCenter ){ // synchro est l'élu, il peut appliquer une règle
        for(door=0;door<arity;door++) {
            etatVoisin[door] = receiveFrom(door);
            countVoisin[door] = receiveFrom(door);
            jetonVoisin[door] = receiveFrom(door);

        }
        if ((mEtat=="T") && (jeton)) { // Appliquer la règle R4.
            pos=randomChoice();
            // pos=randomChoiceMin(countVoisin);
            if (pos!= -1) {
                jeton=false;
                sendTo(pos,token);
            }
        }
        breakSynchro();
    }
    else
        if (synchro==starBorder) {
            // les voisins d'un centre lui envoient leur état et leur rang.
            sendTo(synchro,mEtat);
            sendTo(synchro,mCount);
            sendTo(synchro,jeton);
            if (receiveFrom(synchro) == token) { //Appliquer la règle R4
                jeton=true;
                //mcount=mcount+1/arity
            }
        }
}
```

Algorithme 14: Exclusion mutuelle en utilisant la marche aléatoire

6.7.1 Résultats expérimentaux

Dans cette section nous présentons quelques résultats expérimentaux concernant l'algorithme distribué présenté auparavant. Le but de notre expérimentation est de valider l'analyse probabiliste que nous avons évoquée dans la section précédente. Nous nous concentrons sur la mesure expérimentale de la propriété de *survie*. Plus précisément, nous calculons le temps moyen d'attente pour qu'un processeur puisse obtenir les ressources, i.e. le temps moyen qu'un noeud avec l'état H va attendre pour pouvoir passer à l'état E. Dans le tableau suivant, nous présentons le temps moyen d'attente expérimental obtenu par Visidia pour quelques classes de graphes.

Graphe	**Taille**	**Temps moyen d'attente** ($\mathcal{R}_{20}$)	**Temps moyen d'attente** ($\mathcal{R}_{21}$)
Chaîne	30	2771.29	165.31
Arbre	100	10550.96	1593.24
Anneaux	30	1164.27	152.31
Anneaux	100	19036.93	524.16
Grille	30	1214.82	367.44
Grille	100	5996.26	1463.86
Graphe complet	30	1740.15	937.25
Graphe complet	100	19212.52	9848.44
Graphe aléatoire	30	1708.64	966.44

Les expérimentations nous ont démontré que notre stratégie est intéressante puisque nous obtenons des gains de temps qui atteignent les 36 fois le temps moyen pour une marche aléatoire non biaisé. Notre stratégie minimise le nombre de passages du jeton dans un noeud qui a été visité plusieurs fois, ce qui nous donne un gain de temps important surtout pour les Anneaux ou le graphe complet. Dans le premier cas, le jeton suit une orientation et il tourne dans ce sens tout le temps. Pour les graphes complets, le jeton va vers les noeuds qui n'ont pas encore reçu le jeton.

Conclusion et perspectives

Les travaux de recherche que nous venons de présenter dans cette thèse se scindent en trois parties. La première partie est constituée de l'analyse et des preuves d'algorithmes distribués, tel que le calcul d'arbre recouvrant, codés sous forme de systèmes de réécriture de graphe. La deuxième partie est consacrée à la conception de l'outil Visidia. Enfin l'expérimentation et l'analyse de plusieurs algorithmes de partage de ressources sont traitées dans la dernière partie.
Cette thèse nous a permis d'aborder trois aspects de la recherche : la théorie, la pratique et l'expérimentation. L'outil Visidia en a constitué l'outil central, il nous a en effet permis l'implémentation, le test, et la validation de nombreux algorithmes.

Le système de réécriture de graphe :

Les systèmes de réécriture de graphe, et plus généralement les calculs locaux, nous ont facilité la compréhension des algorithmes distribués ainsi que les preuves de leur validité et/ou terminaison. Les systèmes de réécriture de graphe facilitent ces preuves puisque pour montrer la terminaison, il faut chercher un ensemble d'état qui soit noethérien. Or, certains algorithmes distribués, comme les algorithmes de partage de ressources, ne terminent pas, mais sont tout de même valides. La preuve de validité est basée sur les invariants. Les invariants sont un ensemble de propriétés qui restent toujours vrai à chaque instant de l'exécution du système. Notre contribution a consisté à enrichir les systèmes de réécriture de graphe de manière à les rapprocher des calculs locaux. Pour cela, nous avons ajouté des méta-règles de réécriture aux règles existantes. Ces méta-règles ne changent pas les méthodes de preuve de validité et de terminaison du système de réécriture de graphe, par contre, elles permettent de coder un plus grand nombre d' algorithmes distribués, tels que les algorithmes de calcul d'arbre recouvrant avec identité.

Nous avons prouvé que le produit de deux systèmes de réécriture de graphe, est un système qui conserve les propriétés des deux systèmes. Or, pour résoudre deux des paradigmes des algorithmes distribués : le problème de détection de terminaison, et le problème du synchroniseur, nous avons également besoin de permettre l'activation et la désactivation d'un des systèmes par l'autre. C'est pourquoi nous avons modifié le produit, afin de permettre cette activation/désactivation.
Concernant la détection de terminaison globale d'un algorithme distribué, deux cas de figures sont à considérer : l'algorithme détecte la terminaison locale ou ne la détecte pas.

Dans le premier cas, il suffit d'effectuer notre nouveau produit entre cet algorithme et n'importe quel algorithme détectant la terminaison globale. En effet, dès qu'un noeud détecte sa terminaison locale, il active le deuxième algorithme qui en détectant la terminaison globale permet de conclure que le premier algorithme a fini.
Dans le second cas, notre nouveau produit doit être effectué avec l'algorithme de SSP [71]. L'activation de cet algorithme se fait si aucune règle du système ne peut être appliquée. Par contre, si une règle est applicable, l'algorithme de SSP se désactive et ainsi de suite jusqu'à ce qu'il détecte la terminaison globale, i.e. tous les noeuds exécutent l'algorithme de SSP.
Concernant le problème du synchroniseur, nous avons simulé un système synchrone par un système asynchrone. En effectuant notre nouveau produit entre un algorithme synchrone et un synchroniseur, nous obtenons une simulation d'un système synchrone. En effet, initialement, le synchroniseur est actif et l'algorithme synchrone est désactivé. Ainsi, dès que tous les noeuds sont dans la même phase, le synchroniseur se désactive ce qui active l'algorithme synchrone qui exécute une étape de calcul. Une fois l'étape exécutée, le synchroniseur redevient actif et l'algorithme synchrone se désactive à son tour. Un grand nombre de synchroniseurs existent dans la littérature [72, 6, 4]. Cependant, nous avons proposé deux nouveaux synchroniseurs : l'un est une amélioration du synchroniseur β [4], l'autre est nouveau et se base sur l'algorithme de SSP [71].

Tous les résultats que nous venons de présenter se basent sur les réseaux statiques. Il serait intéressant de modifier le modèle afin qu'il puisse supporter les graphes dynamiques et plus particulièrement les réseaux ad-hoc. De plus, les canaux de communications et les processus sont supposés infaillibles, une étude pour connaître les limites du modèle au niveau de la tolérance aux pannes serait donc nécessaire.
Pour le problème de la détection de terminaison, [32] a donné les classes de graphe dans lesquelles la détection de terminaison est possible. Sur des graphes dynamiques, cette classification ne marche plus, la question que nous pouvons poser est : faut-il redéfinir la terminaison pour les graphes dynamiques ? ou bien faudra-t-il restreindre cette classe de graphes ?

Visidia :

Visidia est un outil qui permet d'animer, de simuler, de tester et d'expérimenter les algorithmes distribués basés sur les calculs locaux et les systèmes de réécriture de graphe. Ainsi, nous avons pu implémenter plusieurs types d'algorithmes distribués, le plus intéressant et le plus difficile étant l'algorithme d'énumération de Mazurkiewicz [49, 16, 17, 15]. Celui-ci se révèle adapté au traitement de plusieurs problèmes tels que l'énumération, l'élection et la reconstruction de graphes. Mais en contre partie, la complexité mémoire de cet algorithme est cubique en le nombre de noeuds et il n'est valide que sur les graphes minimaux. Cependant, l'utilisant des procédures de synchronisation probabilistes, l'expérimentation nous a permis d'exécuter cet algorithme sur plusieurs graphes non minimaux. De plus, nous avons implémenté une simplification de cet algorithme qui permet de diminuer la mémoire utilisée par contre, cet algorithme ne résout pas le problème de reconstruction.

Deux autres implémentations de Visidia ont vu le jour, la distribution totale des entités dans plusieurs machines [22], et l'ajout d'un module de tolérance aux pannes [35]. La première implémentation permet d'augmenter considérablement la taille des graphes que nous voulons tester, mais plusieurs machines sont nécessaires. La limite passe alors de 5000 à 12000 noeuds. La deuxième implémentation diminue la performance de Visidia puisque deux entités autonomes seront créées : l'une implémente l'algorithme distribué et la deuxième implémente le détecteur de panne.

Néanmoins, ces améliorations ne sont pas complètement satisfaisantes, il faudra réécrire Visidia en utilisant un autre langage de programmation plus rapide tel que le C++ en enlevant toutes les redondances. Il faudra inclure le langage formel LiDIA étudié dans [60, 59], ce qui facilitera la création d'algorithmes distribués. Il faudra aussi permettre au graphe d'évoluer : gérer l'ajout et la suppression de sommets et d'arêtes. Pour les réseaux ad-hoc, il faut permettre le déplacement des noeuds. Il faudra aussi ajouter des étiquettes aux arêtes, afin qu'un noeud, ou que l'utilisateur puisse changer l'état d'une ou plusieurs arêtes ainsi, l'information peut être récupérée par les noeuds incidents.

Implémentations et expérimentations :

Les procédures probabilistes de synchronisation peuvent être utilisées pour implémenter les calculs locaux et les systèmes de réécriture de graphe [56, 54, 55]. Ceci, nous a permis d'avoir le même corps de programme pour nos algorithmes, et ainsi facilité l'implémentation sous Visidia. Ça nous a permis aussi d'avoir un générateur automatique d'algorithmes distribués à partir des règles de réécriture de graphe. L'utilisation de ces algorithmes de synchronisation nous a permis l'analyser plusieurs algorithmes de partage de ressources. Le premier est le partage de plusieurs ressources par plusieurs utilisateurs. Ce problème est connu sous le nom du problème des philosophes généralisé, le second est l'algorithme de Naïmi et Tréhel [66] que nous avons codé sous forme de système de réécriture de graphe. Les analyses théoriques ont démontré leur validité et leur complexité. Enfin, nous avons implémenté et expérimenté deux autres algorithmes d'exclusion mutuelle en se basant sur la marche aléatoire et la marche aléatoire biaisé. Les résultats expérimentaux montrent que la marche aléatoire biaisée, est plus rapide que la marche aléatoire.

Nous avons également proposé un autre algorithme probabiliste de synchronisation qui permet d'augmenter la probabilité qu'un noeud se synchronise avec ses voisins. Cette probabilité est bornée par la probabilité du LE1 [55, 76] mais elle permet d'avoir une probabilité meilleur que celle du LE2 [55, 76].

Nos travaux ont montré que l'utilisation de procédures probabilistes facilitent l'implémentation mais rendent l'algorithme plus lent. Il faudra chercher d'autre méthode d'implémentation qui rendent l'algorithme plus rapide. Une autre solution réside à influer sur la synchronisation pour que les noeuds qui n'appliquent aucune règle ne soient pas au centre d'une synchronisation.

Bibliographie

[1] R. Aleliunas, R. M. Karp, R. J. Lipton, L. Lovász, and C. Rackoff, *Random walks, universal traversal sequences, and the complexity of maze problems*, in In 20th Annual Symposium on Foundations of Computer Science, 1979, pp. 218–223.

[2] D. Angluin, *Local and global properties in networks of processors*, in Proceedings of the 12th Symposium on theory of computing, 1980, pp. 82–93.

[3] H. Attiya and J. Welch, *Distributed computing*, McGraw-Hill, 1998.

[4] B. Awerbuch, *Complexity of network synchronization*, in J. ACM, vol. 32, 1985, pp. 804–823.

[5] Y. Azar, A. Z. Broder, A. R. Karlin, N. Linial, and S. Phillips, *Biased random walks*, 1992, pp. 1–9.

[6] V. C. Barbosa, *An introduction to distributed algorithms*, MIT Press, 1996.

[7] V. C. Barbosa, M. R. F. Benevides, and A. L. O. Filho, *A priority dynamics for generalized drinking philosophers*, in Information Processing Letters, vol. 79, 2001, pp. 189–195.

[8] 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, vol. 50,3 of ENTCS, 2001, pp. 229–239.

[9] M. Bauderon, S. Gruner, and M. Mosbah, *A new tool for the simulation and visualization of distributed algorithms*, Tech. Rep. 1245-00, LaBRI, 2000. Accepted in MFI'01, Toulouse, 21-23 May 2001.

[10] M. Bauderon, Y. Métivier, M. Mosbah, and A. Sellami, *From local computation to asynchronous message passing systems*, Tech. Rep. RR-1271-02, University of Bordeaux 1, 2002.

[11] C. Berge, *Graphs and Hypergraphs*, North-Holland Publishing, 1983.

[12] M. Billaud, P. Lafon, Y. Métivier, and E. Sopena, *Graph rewriting systems with priorities*, Lecture notes in computer science, 411 (1989), pp. 94–106.

[13] P. Boldi and S. Vigna, *Computing anonymously with arbitrary knowledge*, in Proceedings of the 18th ACM Symposium on principles of distributed computing, ACM Press, 1999, pp. 181–188.

[14] A. Bottreau, *Réécritures de graphe et calculs distribués*, PhD thesis, Université Bordeaux I, 1997.

[15] J. Chalopin, *Local computations on closed unlabelled edges : the election problem and the naming problem (extended abstract)*, (to appear in Proc. of SOFSEM 2005).

[16] 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, no. 2987 in LNCS, 2004, pp. 90–104.

[17] 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, no. 3256 in LNCS, 2004, pp. 242–256.

[18] K. M. Chandy, L. M. Haas, and J. Misra, *Distributed deadlock detection*, ACM Transactions on Computer Systems, 1 (1983), pp. 144–156.

[19] K. M. Chandy and L. Lamport, *Distributed snapshots : Determining global states of distributed systems*, ACM Transations on Computer Systems, 3 (1985), pp. 63–75.

[20] K. M. Chandy and J. Misra, *The drinking philosophers problem*, ACM Transactions on Programming Languages and Systems, 6 (1984), pp. 632–646.

[21] ——, *Parallel program design - A foundation*, Addison-Wesley, 1988.

[22] B. Derbel and M. Mosbah, *Distributing the execution of a distributed algorithm over a network*, in 7th International Conference on Information Visualization (IV03), London, England, 16-18 July 2003, IEEE Computer Society, pp. 485–490.

[23] E. Dijkstra, *Cooperating sequential processes*, in Programming Languages, Academic Press, 1968, pp. 43–112.

[24] ——, *Hierarchical ordering of sequential processes*, in Operating Systems Techniques, C. Hoare and R. Perrott, eds., Academic Press, 1972.

[25] E. Dijkstra and C. Sholten, *Termination detection for diffusing computations*, Information Processing Letters, 11 (1980), pp. 1–4.

[26] E. W. Dijkstra, W. H. J. Feijen, and A. J. M. van Gasteren, *Derivation of a termination detection algorithm for distributed computations*, Information Processing Letters, 16 (1983), pp. 217–219.

[27] S. Dolev, *Self-stabilization*, MIT Press, 2000.

[28] M. Duflot, L. Fribourg, and C. Picaronny, *Randomized dining philosophers without fairness assumption*, in International Conference on Theoretical Computer Science TCS, 2002, pp. 169–180.

[29] Feige, *A tight lower bound on the cover time for random walks on graphs*, RSA : Random Structures and Algorithms, 6 (1995).

[30] ——, *A tight upper bound on the cover time for random walks on graphs*, RSA : Random Structures and Algorithms, 6 (1995).

[31] W. Feller, *An Introduction to Probability Theory and Its Application, Volume I*, John Wiley and Sons, 1960.

[32] E. Godard, *Réécritures de Graphes et Algorithmique Distribuée*, PhD thesis, Université Bordeaux I, 351 cours de la Libération, F-33405 Talence Cedex, France, 2002.

[33] E. Godard, Y. Métivier, M. Mosbah, and A. Sellami, *Termination Detection of Distributed Algorithms by Graph Relabelling Systems*, in Proc. of 1st International Conference on Graph Transformation, LNCS 2505, Springer-Verlag, 2002, pp. 106–119.

[34] E. Godard, Y. Métivier, and G. Tel, *Election, termination and graph cartography*, (in preparation).

[35] B. Hamid and M. Mosbah, *Détection de pannes dans un système distribué par échange local de messages*, in Journées Scientifiques Francophones, JSF 2003, Tozeur, 2003.

[36] M. Himsolt, *Gml — graph modelling language, university of passau, germany*, 1997. *http ://infosun.fmi.uni-passau.de/Graphlet/GML/*.

[37] A. Israeli and M. Jalfon, *Token management schemes and random walks yield self-stabilizing mutual exclusion*, in Proc. of the Ninth Anual Symposium on Principles of Distributed Computing, 1990, pp. 119–131.

[38] D. Janssens, *Actor grammars and local actions*, in Handbook of Graph Grammars and Computing by Graph Transformation : Vol 3, Concurrency, Parallelism, and Distribution, Hardcover (World Scientific Pub Co Inc), 1999, ch. 2, pp. 57–106.

[39] E. Korach, G. Tel, and S. Zaks, *Optimal synchronization of ABD networks*, pp. 353–367.

[40] L. Lamport, *A new solution of dijkstra's concurrent programming problem*, in Commun. ACM, vol. 17(8), 1974, pp. 453–455.

[41] C. Lavault, *Évaluation des algorithmes distribués*, Hermes, 1995.

[42] D. J. Lehmann and M. O. Rabin, *On the advantages of free choice : A symmetric and fully distributed solution to the dining philosophers problem*, in ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1981, pp. 133–138.

[43] M. Leuschel, *Design and implementation of the high-level specification language csp(lp)*, in PADL'01, 3rd International Symposium on Practical Aspects of Declarative Languages, LNCS, ed., vol. 1990, Springer-Verlag, 2001, pp. 14–28.

[44] I. Litovsky and Y. Métivier, *Computing with graph rewriting systems with priorities*, Theoret. Comput. Sci., 115 (1993), pp. 191–224.

[45] I. Litovsky, Y. Métivier, and E. Sopena, *Different local controls for graph relabelling systems*, Math. Syst. Theory, 28 (1995), pp. 41–65.

[46] ——, *Graph relabelling systems and distributed algorithms*, in Handbook of graph grammars and computing by graph transformation, H. Ehrig, H. Kreowski, U. Montanari, and G. Rozenberg, eds., vol. 3, World Scientific, 1999, pp. 1–56.

[47] N. A. Lynch, *Distributed algorithms*, Morgan Kaufman, 1996.

[48] W. S. Massey, *A basic course in algebraic topology*, Springer-Verlag, 1991. Graduate texts in mathematics.

[49] A. Mazurkiewicz, *Distributed enumeration*, Inf. Processing Letters, 61 (1997), pp. 233–239.

[50] Y. Métivier, M. Mosbah, R. Ossamy, and A. Sellami, *Synchronizers for local computations*, Technical Report, LaBRI, RR-1322-04 (2004).

[51] ——, *Synchronizers for local computations*, vol. 3256 of Lecture Notes in Comput. Sci., 2004, pp. 271–286.

[52] Y. Métivier, M. Mosbah, and A. Sellami, *Proving distributed algorithms by graph relabeling systems : Examples of trees in networks with processor identities*, in Applied Graph Transformations, Grenoble, April, 2002, pp. 45–58.

[53] Y. Métivier, A. Muscholl, and P.-A. Wacrenier, *About the local detection of termination of local computations in graphs*, in International Colloquium on structural information and communication complexity, 1997, pp. 188–200.

[54] Y. Métivier, N. Saheb, and A. Zemmari, *Randomized rendezvous*, in Mathematics and computer science : Algorithms, trees, combinatorics and probabilities, Trends in mathematics, 2000, pp. 183–194.

[55] ——, *Randomized local elections*, Inform. Proc. Letters, 82 (2002), pp. 313–120.

[56] ——, *Analysis of a randomized rendezvous algorithm*, Information and Computation, 184 (2003), pp. 109–128.

[57] Y. Métivier and G. Tel, *Termination detection and universal graph reconstruction*, in International Colloquium on structural information and communication complexity, 2000, pp. 237–251.

[58] U. Montanari, M. Pistore, and F. Rossi, *Modeling concurrent, mobile and coordinated systems via graph transformation*, World Scientific, 1999, ch. 4, in Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 3 : Concurrency, Parallelism, and Distribution, pp. 189–268.

[59] M. Mosbah and R. Ossamy, *A programming language for local computations in graphs.*, Tech. Rep. 1276-04, LaBRI, Université Bordeaux I, 2004.

[60] ——, *A programming language for local computations in graphs : Computational completeness*, in Proceedings of the 5th. Mexican International Conference in Computer Science Colima Mexico 20-24 September, IEEE, 2004.

[61] M. Mosbah and A. Sellami, *Visidia : A tool for the VIsualization and SImulation of DIstributed Algorithms. http ://www.labri.fr/visidia/.*

[62] ——, *Implementation of an enumeration protocol algorithm using local graph computations*, in Second International Workshop on Graph Transformation and Visual Modeling Techniques , Crete, Greece, 2001.

[63] ——, *Un environnement général pour étudier et implémenter les algorithmes distribués*, in Eighth Maghrebian Conference on Software Engineering and Artificial Intelligence (MCSEAI'04), 2004.

[64] M. Mosbah, A. Sellami, and A. Zemmari, *Résolution distribuée de conflits dans un réseau par les systèmes de réécritures*, in Journées Scientifiques Francophones, JSF 2003, Tozeur, 2003.

[65] Y. Moses, Z. Polunsky, and A. Tal, *Algorithm visualization for distributed environments*, in Proceedings IEEE Symposium on Information Visualization, 1998, pp. 71–78.

[66] M. Naïmi and M. Tréhel, *An improvement of the log n distributed algorithm for mutual exclusion*, in Proceedings of the 7th Int. Conference on Distributed Computing Systems, 1987, pp. 371–375.

[67] D. Peleg, *Distributed computing - A locality-sensitive approach*, SIAM Monographs on discrete mathematics and applications, 2000.

[68] K. Raymond, *A tree-based algorithm for distributed mutual exclusion*, in ACM Trans. Comput. Systems, vol. 7(1), 1989, pp. 61–77.

[69] M. Raynal, *Networks and distributed computation : concepts, tools, and algorithms*, MIT Press, Cambridge, 1988.

[70] P. Rosenstiehl, J.-R. Fiksel, and A. Holliger, *Intelligent graphs*, in Graph theory and computing, R. Read, ed., Academic Press (New York), 1972, pp. 219–265.

[71] 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, 1985, pp. 287–292.

[72] G. Tel, *Introduction to distributed algorithms*, Cambridge University Press, 2000.

[73] G. Tel and F. Mattern, *The derivation of distributed termination detection algorithms from garbage collection schemes*, ACM Transactions on Programming Languages and Systems, 15 (1993), pp. 1–35.

[74] J. L. Welch and N. A. Lynch, *A modular drinking philosophers algorithm*, in Distributed Computing, vol. 6(4), 1993, pp. 233–244.

[75] M. Yamashita and T. Kameda, *Computing on anonymous networks : Part i - characterizing the solvable cases*, IEEE Transactions on parallel and distributed systems, 7 (1996), pp. 69–89.

[76] A. Zemmari, *Contribution à l'Analyse d'Algorithmes Distribués*, PhD thesis, Université Bordeaux I, 2000.

Index

Des Calculs Locaux aux Algorithmes Distribués

Résumé :L'objectif de cette thèse est de montrer que le modèle des systèmes de réécriture de graphe est un modèle pertinent pour l'étude, la compréhension et l'implémentation des algorithmes distribués.

Dans un algorithme distribué, la détection de terminaison est difficile puisqu'un algorithme est exécuté par plusieurs processus en parallèle et l'algorithme ne se termine que lorsque tous les processus ont fini. Pour résoudre ce problème, nous utilisons un produit particulier de deux systèmes de réécriture de graphe. Le premier code l'algorithme dont nous voulons détecter sa terminaison et le second code un des algorithmes qui détecte la terminaison globale.

Les algorithmes asynchrones sont souvent moins performants que les algorithmes correspondants synchrones. Il est très utile de disposer d'une méthode générale qui permet de simuler les algorithmes synchrones sur des systèmes asynchrones. Nous utilisons un autre produit, similaire à celui utilisé pour la résolution du problème de détection de terminaison, pour résoudre ce problème.

Nous avons implémenté une plate-forme appelée Visidia qui utilise les systèmes de réécriture de graphe pour implémenter les algorithmes distribués. Visidia permet de tester, expérimenter, valider et visualiser l'exécution des algorithmes distribués codés sous forme de systèmes de réécriture de graphe ou sous forme de calcul local.

Nous utilisons Visidia pour faire des expérimentations sur des algorithmes qui résolvent le problème de partage des ressources. Plusieurs algorithmes ont été testés dont la généralisation du problème des philosophes, l'exclusion mutuelle dans un arbre et enfin, une généralisation du problème de l'exclusion mutuelle dans un réseau quelconque.

Discipline : Informatique

Mots-Clefs : Systèmes de réécriture de graphe, algorithmes distribués, détection de la terminaison, synchroniseurs, Visidia, partage de ressources.

From local computations to distributed algorithms

Abstract : The goal of this thesis is to show that the model of graph relabelling systems is a relevant model for the study, comprehension and the implementation of distributed algorithms.

In a distributed algorithm, the termination detection of an algorithm is difficult since an algorithm is executed simultaneously by many processes and the algorithm terminates only when all the processes terminate. To solve this problem, we use a particular product of two graph relabelling systems. The first encodes the algorithm for which we want to detect the termination and the second encodes an algorithm which detects locally the global termination.

The asynchronous algorithms are often less powerful than the synchronous corresponding ones. It is very useful to have a general method which makes the simulation of synchronous algorithms on asynchronous systems possible. To do so, we use another product, similar to that used for the termination detection problem, to simulate synchronous systems.

We had implemented a platform called Visidia which uses graph relabelling systems to implement distributed algorithms. With Visidia we can test, experiment, verify and visualize the execution of distributed algorithms encoded by graph relabelling systems or local computations.

We use Visidia to make experiments on algorithms which solve the sharing resource problems. Many algorithms have been tested : the drinking philosophers problem, mutual exclusion in trees and finally, mutual exclusion in random networks.

Discipline : Computer-Science

Keywords : Graph Relabelling Systems, distributed algorithms, termination detection, synchronizers, Visidia, sharing resources.

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