Contribution à l'algorithmique distribuée dans les réseaux mobiles ad hoc - Calculs locaux et réétiquetages de graphes dynamiques

Arnaud Casteigts

▶ **To cite this version:**

Arnaud Casteigts. Contribution à l'algorithmique distribuée dans les réseaux mobiles ad hoc - Calculs locaux et réétiquetages de graphes dynamiques. Réseaux et télécommunications [cs.NI]. Université Sciences et Technologies - Bordeaux I, 2007. Français. NNT : . tel-00193181

HAL Id: tel-00193181

https://theses.hal.science/tel-00193181

Submitted on 1 Dec 2007

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

N° d'ordre : 3430

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ BORDEAUX I

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

Par **Arnaud Casteigts**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

Contribution à l'algorithmique distribuée dans les réseaux mobiles ad hoc
-
Calculs locaux et réétiquetages de graphes dynamiques

Soutenue le : 27 septembre 2007

Après avis des rapporteurs :

Frédéric Guinand .	Professeur	
David Simplot-Ryl	Professeur	

Devant la commission d'examen composée de :

Mohamed Mosbah	Professeur	Président du jury
Isabelle Demeure .	Professeur	Examinateurs
David Simplot-Ryl	Professeur	
Frédéric Guinand .	Professeur	
Afonso Ferreira . . .	Directeur de Recherche	
Serge Chaumette .	Professeur	Directeur de thèse

– 2007 –

à Burgaronne, à ses habitants...

Remerciements

Je tiens à exprimer ma reconnaissance et ma gratitude à un certain nombre de personnes qui ont contribué, chacune à sa manière, à l'aboutissement de ces travaux.

Merci tout particulièrement à Serge Chaumette, mon directeur de thèse, pour m'avoir accueilli au sein de son équipe et encadré, mais surtout pour avoir pressenti, il y a trois ans déjà, le potentiel d'un sujet comme celui-ci.

Un grand merci, ensuite, à Frédéric Guinand et David Simplot-Ryl, rapporteurs de cette thèse, non seulement pour leur disponibilité et leur implication, mais également pour leur grande sympathie.

Je souhaiterai aussi remercier Mohamed Mosbah, pour avoir accepté de présider les délibérations et pour m'avoir manifesté son intérêt tout au long de mes recherches, ainsi qu'à Isabelle Demeure et Afonso Ferreira, qui m'ont fait l'honneur d'évaluer mon travail en participant à ce jury.

L'aboutissement de ces travaux doit également beaucoup, et c'est peu dire, à la qualité de l'environnement dans lequel ils ont été effectués, à commencer par la salle CVT et tous ses "membres". J'aimerais en particulier remercier Eve, Lionel, Rémi, Fabien, Achraf et Jérémie pour leur bonne humeur et pour l'ambiance inimitable qu'ils ont su faire régner au quotidien. Et bien sûr Martin, pour tant de raisons, et pour avoir su mettre entre mes mains le bon livre, au bon moment. Toujours au laboratoire, mais je ne peux ici être exhaustif, j'ai une pensée toute particulière pour Nader, Joan, Omer, Youssou, Florian et Jocelyn, ainsi que pour Yves Métivier et Akka Zemmari.

Enfin, et ce n'est pas la moindre des choses, je ne peux écrire cette page sans citer famille et amis, qui ont tout autant contribué à cette aventure. Merci donc à I. et M., sans limites, à A., A. et A., mes doubles, à GP. et M., à M., et à M., à L. (grand, le merci !), à P., à M., à D., S. et M-L., et aussi à E., bien sûr. Un peu moins directement, merci aussi à H. et A. (et S. et C.), et à A-M. (et M., H., et R.). Merci enfin à Franck Barbier, de l'université de Pau et des Pays de l'Adour, sans qui j'aurais probablement arrêté mon parcours à l'issue de la maîtrise, et à Bruno Jobard, Alain Dagois et Dominique Beyou, qui m'ont éveillé de différentes manières à la recherche ou à l'informatique.

Table des matières

Introduction **1**

1 Préliminaires **7**
1.1 Définitions générales sur les graphes 7
1.2 Graphes étiquetés 9
1.3 Etiquetage des sommets et des brins d'arêtes 10
1.4 Réétiquetages de graphes 11

2 Modèles de calculs et formalismes associés **13**
2.1 Systèmes distribués 14
2.2 Réétiquetages de graphes et calculs locaux 18
2.3 Adaptation à un cadre dynamique 21
2.4 Quelques algorithmes 27

3 Synchronisation entre voisins **33**
3.1 Procédures existantes 34
3.2 Synchronisation à la demande 37
3.3 Synchronisation à la demande avec critères 38
3.4 Critères étendus pour les réseaux sans fil 43
3.5 Exemple d'utilisation 43

4 Analyse d'algorithmes **45**
4.1 Les Graphes Évolutifs 46
4.2 Autres définitions sur les graphes évolutifs 48
4.3 Outils pour l'analyse d'algorithmes distribués 52
4.4 Analyse de quelques algorithmes simples 55
4.5 Vers une classification des réseaux dynamiques 60

5 Développements logiciels **69**
5.1 Simulateur de réétiquetage de graphes dynamiques 70
5.2 Éditeur de graphes évolutifs 76
5.3 Vérificateur de propriétés sur les graphes évolutifs 77
5.4 Convertisseur *DGS* vers et depuis les graphes évolutifs 79
5.5 Récapitulatif de la chaîne logicielle 83

6 Assistance au développement d'applications réelles **85**
6.1 Principe général 86
6.2 Détail de l'architecture et canonicité du développement 86
6.3 Illustration de la généricité du développement à travers deux exemples . . . 89
6.4 Extension du modèle à la non-atomicité des réétiquetages 92

7 Directions de recherches **95**

Conclusion **99**

Table des figures

1 Ordres de lecture du document 6

1.1 Exemple d'étiquetage de sommets et de brins d'arêtes dans un graphe simple 10

2.1 Modèles de calculs locaux de différentes puissances 17
2.2 Exemple de définition d'une règle de réétiquetage avec le langage LIDiA . . 19
2.3 Représentation graphique d'une règle de réétiquetage LC_2 19
2.4 Règle codant un algorithme d'arbre couvrant 20
2.5 Exemple d'exécution de l'algorithme de la figure 2.4 20
2.6 Exemple de règle portant sur des étiquettes de sommets et de brins d'arêtes 22
2.7 Représentation graphique de la règle de la figure 2.6 22
2.8 Apparition d'un lien de communication 23
2.9 Rupture d'un lien de communication 23
2.10 Exemple de règle de réaction à la rupture 23
2.11 Exemple de séquence d'exécution de l'algorithme de la forêt couvrante . . . 26

3.1 Quelques rendez-vous dans un graphe 35
3.2 Calcul de quelques critères physiques 39
3.3 Exemples de critères algorithmiques 40
3.4 Impact des critères algorithmiques 40
3.5 Ordonnancement et sélection des voisins - architecture 41
3.6 Ordonnancement et sélection des voisins - exemple de fonctionnement détaillé 42
3.7 Exemple de fusion manquée 44

4.1 Suite de sous-graphes $\mathcal{S}_G$ de $\mathcal{G}$, indiquant la topologie du réseau à quatre dates différentes 48
4.2 Représentation étiquetée du graphe évolutif de la figure 4.1 49
4.3 Un graphe évolutif $\mathcal{G}$ et trois de ses sous-graphes évolutifs $\mathcal{G}'$, $\mathcal{G}''$ et $\mathcal{G}'''$ . . . 49
4.4 Dates dans les trajets canoniques 50
4.5 Destinations dans un graphe évolutif $\mathcal{G}$ 51
4.6 Un graphe évolutif $\mathcal{G}$ et trois de ses coupes temporelles 52
4.7 Schéma récapitulatif sur l'étiquetage des graphes évolutifs 53
4.8 Classe de graphes dynamiques 1 61
4.9 Classe de graphes dynamiques 2 61
4.10 Classe de graphes dynamiques 3 61
4.11 Classe de graphes dynamiques 4 62
4.12 Classe de graphes dynamiques 5 62

4.13 Classe de graphes dynamiques 6 . . . 63
4.14 Classe de graphes dynamiques 7 . . . 63
4.15 Relations d'inclusion entre quelques classes de graphes évolutifs . . . 66

5.1 Édition des états manipulés par l'algorithme . . . 70
5.2 Édition des règles de réétiquetage de l'algorithme . . . 71
5.3 Représentation d'un algorithme dans le simulateur . . . 72
5.4 Structure d'un fichier d'algorithme pour le simulateur . . . 72
5.5 Édition d'états de sommets durant l'exécution de l'algorithme . . . 73
5.6 Exemple d'arbre binaire pour les préconditions . . . 75
5.7 Vue d'ensemble du simulateur . . . 76
5.8 Format, explications et exemple de fichier stockant un graphe évolutif . . . 77
5.9 Édition d'un graphe évolutif . . . 78
5.10 Exemple de fichier au format DGS . . . 80
5.11 Liens possibles entre les outils logiciels développés . . . 83

6.1 Application reposant sur un algorithme de réétiquetages de graphes . . . 86
6.2 Interface générée depuis un algorithme . . . 88
6.3 Utilisation du moteur de réétiquetage dans une application . . . 88
6.4 Algorithme de propagation et interface associée . . . 89
6.5 Utilisation de l'algorithme de propagation dans une application . . . 89
6.6 Algorithme de la forêt d'arbres couvrants et interface associée . . . 90
6.7 Utilisation de l'algorithme de la forêt couvrante dans une application . . . 91
6.8 Déroulement d'un réétiquetage avec exécution de code . . . 92
6.9 Règle de réétiquetage munie d'états de secours . . . 93
6.10 Interruption d'un réétiquetage avec états de secours . . . 93

7.1 Modèle de calcul de l'étoile ouverte . . . 95
7.2 Impact du délai d'attente dans la procédure de synchronisation à la demande 97

Introduction

L'histoire récente de l'informatique retiendra, parmi les évolutions majeures, la prolifération des réseaux dynamiques et des équipements qui leur sont associés. Ces réseaux peuvent être très variés dans leur nature. Bien qu'il n'existe pas à ce jour de classification complète, il est cependant admis de les séparer en deux grandes familles : ceux qui possèdent une infrastructure, et ceux qui n'en possèdent pas. Dans les réseaux infrastructurés, les différents participants s'interconnectent par le biais de dispositifs dédiés et stables. Ces dispositifs arbitrent, centralisent et parfois acheminent les communications entre les divers équipements du réseau. C'est le cas pas exemple des réseaux de classe IP dont les participants se connectent via des points d'accès Wi-Fi. Les réseaux non-infrastructurés sont, au contraire, caractérisés par l'absence de tels dispositifs. Dans ce type de réseau les différents participants communiquent directement les uns avec les autres, de proche en proche, et de manière décentralisée. La classe des réseaux dynamiques sans infrastructure peut à son tour être découpée selon des critères bien définis. Il est ainsi possible, par exemple, de distinguer d'une part les réseaux dynamiques dont l'évolution topologique est connue à l'avance, dits réseaux à topologie *programmée*, comme les réseaux de satellites en orbite basse [WM97], et d'autre part les réseaux dont l'évolution n'est pas planifiée, comme ceux qui sont constitués de terminaux mobiles communicants pouvant équiper des piétons. On peut également distinguer les réseaux dont l'évolution topologique est *contrôlée*, comme par exemple les réseaux constitués de robots mobiles prenant des décisions sur leurs déplacements [SMR06], de ceux dont l'évolution topologique est *subie* par le système sous-jacent. Lorsqu'aucune hypothèse n'est faite, ni sur la connaissance de la topologie, ni sur son contrôle, ni même sur la liste des participants, on parle alors de réseau dynamique (ou mobile) ad hoc, ou encore de *MANet* (Mobile Ad hoc Networks).

Dans la plupart des travaux sur les MANets, ces derniers désignent des réseaux dont les équipements sont tributaires des déplacements d'une personne ou d'un objet (téléphones ou ordinateurs portables, assistants personnels, voitures communicantes, *etc.*). Ce type de réseaux modifie en profondeur la manière dont l'informatique était jusqu'à présent utilisée, car ce n'est plus l'homme qui s'adapte à l'outil, mais, au contraire, l'outil qui s'adapte à l'homme et le suit dans ses déplacements. Ce paradigme engendre ainsi de nouvelles possibilités, comme la constitution spontanée de réseaux à faible coût de fonctionnement, et de nouveaux problèmes, liés en partie à la nécessité pour ces réseaux de s'auto-organiser. La gestion de ces réseaux représente aujourd'hui un défi important pour la communauté scientifique.

Un certain nombre de travaux relatifs aux MANets ont vu le jour ces dernières années, notamment dans le domaine du routage où plusieurs protocoles dédiés ont été conçus, comme *DSR* [JMB01], *AODV* [Per97] ou *TORA* [PC97]. L'approche du routage dans

les réseaux mobiles ad hoc a, entre autres finalités, l'ambition de masquer le caractère dynamique du réseau aux applications qui s'y exécutent. Une couche de routage ne semble pas systématiquement nécessaire à toute tâche qu'un réseau dynamique doit remplir, et il est parfois bon de ne pas masquer les informations topologiques aux applications, en particulier les informations relatives à l'évolution du voisinage de chaque participant. Par ailleurs, cette approche suppose l'existence d'identifiants uniques sur chaque entité du réseau, hypothèse qui, bien que technologiquement réaliste, est fondamentalement forte dans un cadre théorique.

Dans nos travaux, nous nous sommes intéressés à l'étude des réseaux dynamiques de la manière la plus générale possible (*i.e.,* avec le moins d'hypothèses contraignantes), avec pour principal objectif d'essayer d'en comprendre la substance, les possibilités et les limitations. Le domaine de recherche qui a servi de support à cette étude est celui de l'algorithmique distribuée, qui offre par nature un paradigme décentralisé. L'algorithmique distribuée recèle de nombreux problèmes fondamentaux, pour la plupart devenus classiques, qui permettent de caractériser les capacités et les limitations d'un réseau. On peut citer parmi les plus connus de ces problèmes l'élection (distinction d'un participant parmi les autres), le dénombrement (ou *comptage* du nombre de participants), la propagation d'information ou encore la construction de structures couvrantes telles que les anneaux ou les arbres (par ailleurs utilisés dans le domaine du routage).

Si l'on considère un réseau dont la topologie est susceptible de varier à tout moment, sans avertissement préalable, alors les seules communications que doit mettre en œuvre un algorithme distribué sont celles qui impliquent des voisins directs. Cet état de fait nous a naturellement orienté vers le domaine des calculs locaux, pour lesquels une étape de calcul distribué doit se dérouler sur un ensemble restreint de participants, éloignés les uns des autres d'une distance maximale bornée en nombre de sauts, que nous avons limitée à 1 en contexte dynamique. Les calculs locaux dans le cadre des réseaux statiques ont fait l'objet de plusieurs travaux. On pourra notamment citer [LMS95], où Litovsky, Métivier et Sopena discutent entre autres du problème de l'élection dans les réseaux anonymes, par le biais de calculs locaux. Les traitements y sont présentés sous la forme de règles de réétiquetage de graphes, portant uniquement sur des sommets voisins entre eux. Depuis ce travail précurseur, de nombreux résultats sur les calculs locaux sont disponibles dans la littérature. Une partie importante de ces résultats consiste en la caractérisation, pour un problème donné, des hypothèses sur le réseau qui sont nécessaires et/ou suffisantes pour qu'un algorithme donné fonctionne, qu'il ait une certaine complexité, ou même qu'il puisse exister, *etc.* Ces hypothèses peuvent porter sur différentes propriétés, comme par exemple l'absence de certaines symétries dans la topologie du réseau [Ang80], la présence ou non d'identifiants uniques pour les participants, une borne sur la taille, ou encore la possibilité de distinguer un participant parmi les autres avant le début du calcul. Enfin, les analyses qui accompagnent ces résultats portent généralement sur l'étude de la terminaison des algorithmes, sur la capacité à détecter cette terminaison, ou sur l'étude de la complexité en temps ou en mémoire des algorithmes.

Avec les réseaux dynamiques apparaissent de nouveaux paramètres à prendre en compte. En premier lieu, le succès ou l'échec d'un algorithme pour un réseau donné dépendra très fortement des évolutions topologiques qui s'y sont produites au cours de l'exécution, et non plus seulement d'hypothèses traditionnelles telles que celles portant sur la taille ou la forme du graphe ou les connaissances initiales dont disposent les nœuds. En second lieu, les

algorithmes dans ce contexte ne sont pas tous faits pour terminer. Les algorithmes visant à construire des structures couvrantes sur le réseau devront par exemple tenter de les maintenir au fur et à mesure des changements topologiques, sans qu'un état terminal ne soit jamais atteint. De la même manière, un algorithme de propagation d'information ne pourra jamais terminer sans hypothèse sur le nombre maximal de participants. Cette thèse a tenté de poser quelques bases pour la conception, l'analyse (et accessoirement la visualisation) d'algorithmes à base de calculs locaux dans le cadre des réseaux dynamiques. Les travaux que nous présentons sont essentiellement théoriques et se situent au niveau d'abstraction des graphes, bien qu'ils soient susceptibles de déclinaisons pratiques. Ils s'inscrivent en ce sens dans un cadre très général, qui ne dépend d'aucun contexte technologique donné.

Structure du document

Le présent document est composé de sept chapitres dont chacun traite d'un aspect bien précis de nos recherches. Les paragraphes ci-dessous en résument le contenu.

Chapitre 1 : Préliminaires

Ce premier chapitre introduit les notions qui seront utiles à la compréhension du document. Ces notions sont pour l'essentiel issues de la théorie des graphes ainsi que du domaine des réétiquetages de graphes. Nous introduisons également le type d'étiquetage que nous manipulerons le plus fréquemment dans ce document, à savoir l'étiquetage des sommets et des *brins d'arêtes* (notion que nous distinguons des *ports de communication*).

Chapitre 2 : Modèles de calculs et formalismes associés

Dans ce chapitre, nous discutons des raisons qui nous ont incités à utiliser des modèles de calculs locaux, et présentons une liste non exhaustive des plus connus d'entre eux, à savoir les calculs locaux sur les étoiles *fermées* et *ouvertes* et sur les arêtes (ou paires de sommets). Le chapitre présente ensuite la première contribution de la thèse, qui réside en l'adaptation de ce type de modèles aux graphes dynamiques. Cette adaptation a fait l'objet d'une publication à *PDCS'05* [CC05]. Le chapitre s'achève par quelques exemples d'algorithmes dont les propriétés sont discutées.

Chapitre 3 : Synchronisation entre voisins

Les modèles de calculs locaux présentés dans le chapitre 2 supposent chacun l'existence d'une procédure de synchronisation sous-jacente à la réalisation des calculs. Ces procédures, également distribuées, déterminent les voisins qui vont travailler ensemble et sont généralement exécutées entre chaque étape de calcul. Après avoir présenté quelques unes des procédures existantes, nous en proposons une nouvelle adaptée aux réseaux dynamiques. Cette procédure, moins équitable que les autres, permet au contraire d'utiliser des critères physiques ou algorithmiques pour favoriser les interactions entre certains liens de communication plutôt que d'autres. Les premiers éléments de ce mode de synchronisation ont été publiés lors de la conférence *WASA'06* [CC06].

Chapitre 4 : Analyse d'algorithmes

Le chapitre 4 s'intéresse à l'analyse d'algorithmes distribués basés sur des calculs locaux dans le cadre des réseaux dynamiques. Comme évoqué plus haut, il existe plusieurs manières d'analyser un algorithme. Dans ce chapitre, nous proposons un nouveau cadre d'analyse dédié à ce que notre contexte a de spécifique : son aspect dynamique. L'objectif est de fournir un ensemble d'outils permettant de caractériser, pour tout algorithme donné, les conditions nécessaires ou suffisantes, sur la dynamique du réseau, pour qu'il atteigne ses objectifs. Nous distinguons à ce propos deux types fondamentaux d'algorithmes, selon que leurs objectifs consistent à atteindre une configuration spécifique (algorithmes dits *à finalité*), ou à maintenir continuellement une propriété (algorithmes dits *de maintien*). Toutes les caractérisations sont exprimées par le biais de propriétés sur les graphes évolutifs, omniprésents dans ce chapitre et dont nous avons étendu certaines définitions. Les propriétés que l'analyse fait émerger définissent à leur tour, de fait, des classes de graphes dynamiques bien particulières, pour lesquelles on peut dire que tel algorithme fonctionne ou ne fonctionne pas. La dernière partie du chapitre montre que certaines de ces classes en englobent d'autres, et que le cadre d'analyse proposé permet ainsi, indirectement, d'alimenter l'élaboration d'une classification des réseaux dynamiques. Le chapitre se termine par une discussion sur l'intérêt d'une classification de ces réseaux, en particulier dans le domaine de l'algorithmique distribuée.

Chapitre 5 : Développements logiciels

Dans ce chapitre nous présentons les réalisations logicielles qui ont été effectuées autour de nos travaux de recherche. Ces réalisations comprennent avant tout un simulateur de réétiquetage de graphes dynamiques s'inspirant de ce que propose la plateforme ViSiDiA pour les graphes statiques [MMZG]. Cet outil permet entre autres d'éditer un algorithme à base de réétiquetages, en utilisant le modèle de calcul qui a été proposé au chapitre 2. Les algorithmes édités peuvent ensuite être testés sur une topologie dynamique, dont les évolutions sont pilotées directement par l'utilisateur pendant l'exécution. Sur la base de ce simulateur, nous avons également réalisé un éditeur de graphes évolutifs, qui permet d'exporter les graphes créés vers un format de fichier dédié. Ce format de fichier est une représentation à plat des évolutions topologiques qui se produisent dans un réseau : la structure du graphe est fixe et les informations concernant les changements topologiques sont mémorisées sous forme d'attributs des éléments du graphe. Un autre format de description de graphes dynamiques, nommé *Dynamic GraphStream* (ou *DGS*) [PD], représente au contraire les évolutions du réseau par un flux d'évènements séquentiels qui s'appliquent à la structure même du graphe (ajouts ou retraits d'arêtes et de sommets). Un double convertisseur a été developpé pour permettre de transformer ces fichiers *DGS* en graphes évolutifs, et *vice versa*. Ce convertisseur permet notamment de mettre en commun les graphes dynamiques issus de nos travaux et ceux qui sont issus de simulations faites par l'équipe qui est à l'origine de *DGS*, ces dernières utilisant des modèles de mobilité réalistes *via* le simulateur *Madhoc* [HC]. Le dernier outil que nous avons développé est un vérificateur de propriétés pour graphes évolutifs. A ce jour, le vérificateur implémente le test des propriétés nécessaires et suffisantes qui ont été mises en évidence au chapitre 4. Il permet ainsi de déterminer la classe à laquelle appartient un graphe dynamique donné et par conséquent de savoir quels algorithmes y fonctionneront, ou n'y fonctionneront pas.

Combiné au convertisseur de graphe dynamique, ce vérificateur de propriétés permettra de tester une grande variété de graphes issus des simulations de *Madhoc*. L'intégralité des réalisations logicielles présentées dans ce chapitre est le fruit de nos travaux. La plupart d'entre elles, dont le simulateur de réétiquetage de graphes dynamiques, a été publiée sous licence GPL [FSF] et est accessible sur [Cas07].

Chapitre 6 : Assistance au développement d'applications réelles

Le sixième chapitre s'intéresse à ce que peut apporter notre modèle de calcul au domaine du génie logiciel. Nous proposons dans cette partie une méthode de développement d'applications distribuées pour réseaux dynamiques qui permet d'utiliser comme base un algorithme de réétiquetage de graphe. L'architecture correspondante est composée de trois niveaux hiérarchiques. Elle n'a pas été implémentée et nous n'en proposons ici que le principe : chaque équipement du réseau est doté d'une couche de synchronisation, au-dessus de laquelle s'exécute un moteur de réétiquetage, au-dessus duquel s'exécute à son tour l'application. La couche de synchronisation régit le choix des voisins qui vont coopérer. A chaque synchronisation réussie, la couche de synchronisation passe la main au moteur de réétiquetage, qui applique de manière distribuée les règles de l'algorithme choisi. Par le biais d'un mécanisme d'*interfaces* objets, l'application qui s'exécute en haut de cette pile est alors prévenue, et exécute un traitement de haut niveau correspondant. Plus précisément, l'application associe à chaque règle de l'algorithme un traitement particulier qui sera exécuté à chaque fois que la règle est jouée par le moteur de réétiquetage. La généricité naturelle qu'offrent les réétiquetages de graphes permet ainsi aux applications de s'affranchir d'un certain nombre de traitements ayant trait à l'organisation du réseau, et donc de se focaliser sur des aspects haut niveau. Dans les exemples d'utilisation que nous donnons, nous faisons l'hypothèse que chaque terminal possède une plateforme d'exécution orientée objet, de type *J2ME* [RTH+03] ou équivalent. En fin de chapitre, nous proposons une extension du modèle de calcul prenant en compte le cas où les liens de communication peuvent être rompus en cours de réétiquetage. Ce problème se pose dès lors qu'on ne considère plus les réétiquetages comme étant atomiques, hypothèse que nous avons faite dans le cadre de nos travaux théoriques. L'architecture présentée dans ce chapitre a été publiée dans le cadre de la conférence *ICAS'06* [Cas06].

Chapitre 7 : Perspectives

Ce chapitre regroupe des axes de recherche et des extensions qui découlent des travaux effectués. Les pistes de recherche étant nombreuses, nous avons décidé d'en faire un chapitre à part entière.

Dépendance entre chapitres

Les chapitres de cette thèse entretiennent des liens de dépendance les uns avec les autres. Même si elle semble préférable, une lecture linéaire du document n'est pas obligatoire. Le chapitre 4, par exemple, peut être lu directement après le chapitre 2. Le lecteur pourra se référer à la figure 1 page suivante pour connaître les liens de dépendance entre chapitres.

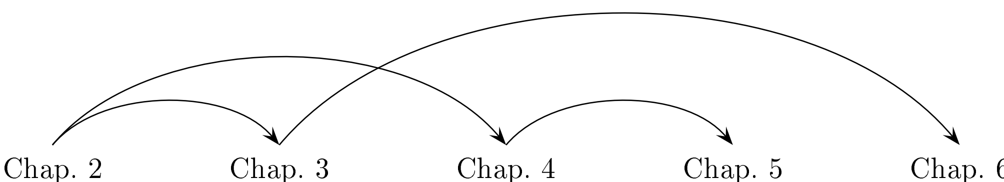


FIG. 1 – Liens de dépendance de lecture entre les chapitres de cette thèse

Chapitre 1

Préliminaires

Sommaire

1.1 Définitions générales sur les graphes 7
1.2 Graphes étiquetés 9
1.3 Etiquetage des sommets et des brins d'arêtes 10
1.4 Réétiquetages de graphes 11

Dans ce chapitre nous donnons des définitions générales, conventions et notations qui seront utilisées dans l'ensemble du document.

1.1 Définitions générales sur les graphes

Définition 1.1 *Un **graphe non-orienté** G est un triplet constitué d'un **ensemble de sommets** $V(G)$, d'un **ensemble d'arêtes** $E(G)$ et d'une relation endpoints qui associe à chaque arête de $E(G)$ deux sommets (non nécessairement distincts) de $V(G)$.*

*Si e est une arête de $E(G)$, et si u et v sont deux sommets de $V(G)$ tels que $endpoints(e) = \{u, v\}$, alors u et v sont appelés **extrémités** de e, et e est dite **incidente** à u et v.*

Définition 1.2 *Une **boucle** est une arête dont les deux extrémités sont identiques. Des **arêtes multiples** sont des arêtes ayant la même paire de sommets pour extrémités.*

*Un **graphe simple non-orienté** est un graphe non-orienté n'ayant ni boucles ni arêtes multiples. De tels graphes peuvent être spécifiés par leur ensemble de sommets et leur ensemble d'arêtes, en considérant chaque arête comme une paire de sommets distincts, et en notant $e = uv$ (ou $e = vu$) si u et v sont les extrémités de e. Les sommets u et v sont alors dits **adjacents**, ou **voisins** (ces termes ne sont pas exclusifs aux graphes simples).*

Dans ce document, sauf mention contraire, le terme *graphe* désignera un graphe simple non orienté.

Définition 1.3 *Un graphe est dit **fini** si son ensemble de sommets et son ensemble d'arêtes sont tous deux finis. Le nombre de sommets d'un graphe G est appelé **ordre** de G, le nombre d'arêtes d'un graphe G est appelé **taille** de G.*

Définition 1.4 *Soient H et G deux graphes.*

*H est un **sous-graphe** de G si et seulement si $V(H) \subseteq V(G)$ et $E(H) \subseteq E(G)$.*

*H est un **graphe partiel** de G si et seulement si $V(H) = V(G)$ et $E(H) \subseteq E(G)$.*

*H est un **sous-graphe induit** de G si et seulement si $V(H) \subseteq V(G)$ et $E(H)$ est l'ensemble des arêtes de G dont les deux extrémités sont dans $V(H)$. Pour tout ensemble $S \subseteq V(G)$, on note $G[S]$ le sous-graphe induit de G dont les sommets sont les éléments de S.*

Définition 1.5 *Le **voisinage** d'un sommet u dans un graphe G, noté $N_G(u)$, est l'ensemble des sommets adjacents à (ou voisins de) u, i.e., $\forall v \in V(G), v \in N_G(u) \Leftrightarrow \exists e \in E(G) \mid endpoints(e) = \{u, v\}$. On note également $I_G(u)$ l'ensemble des arêtes incidentes à u, i.e., $\forall u \in V(G), \forall e \in E(G), e \in I_G(u) \Leftrightarrow u \in endpoints(e)$.*

*Le **degré** d'un sommet u, noté $deg_G(u)$ est le nombre d'arêtes incidentes à u : $deg_G(u) = |I_G(u)|$ (aussi égal à $|N_G(u)|$ pour les graphes simples).*

Définition 1.6 *Si les sommets du graphe sont tous de degré 0 ($E(G) = \emptyset$), on dit que G est un **stable**. Un graphe G est **biparti** si on peut partitionner les sommets de $V(G)$ en deux ensembles V_1 et V_2 tels que $G[V_1]$ et $G[V_2]$ sont des stables.*

Définition 1.7 *Dans un graphe simple G, un **chemin** Γ est une suite de sommets $P = (u_0, u_1, ..., u_n)$ telle que pour tout $0 \leq i < n$, $u_i \in V(G)$ et $\{u_i, u_{i+1}\} \in E(G)$. Les sommets u_0 et u_n sont les **extrémités** du chemin, les autres sommets de P sont les **sommets internes** du chemin. La **longueur** d'un chemin est le nombre d'arêtes traversées, soit n. Un chemin dont tous les sommets sont distincts est dit **élémentaire**.*

*Un chemin dont les extrémités sont confondues est un **cycle**. Un cycle élémentaire est un cycle dont tous les sommets internes sont distincts. Un **anneau** est un graphe constitué uniquement d'un cycle élémentaire.*

Remarque : Les chemins peuvent également être représentés par une suite alternée $u_1, e_1, u_2, e_2, ..., u_n$ de sommets et d'arêtes, dont chaque arête e_i relie u_i à u_{i+1} dans le graphe.

Définition 1.8 *La **distance** entre deux sommets u et v de G, notée $dist_G(u, v)$ est la longueur du plus court chemin de u à v. Le **diamètre** d'un graphe connexe G, noté $D(G)$, est la plus grande distance entre deux de ses sommets, i.e., $D(G) = max\{dist_G(u, v) \mid u, v \in V(G)\}$.*

Définition 1.9 *Deux sommets u et v sont dits **connectés** si et seulement si il existe un chemin de u à v. Un graphe **connexe** est un graphe dont tous les sommets sont connectés deux à deux, i.e., pour tous les sommet $u, v \in V(G)$, il existe un chemin entre u et v.*

*Dans un graphe G, un **ensemble connexe** est un sous-ensemble de $V(G)$ dont tous les éléments sont connectés deux à deux. La **composante connexe** contenant un sommet donné est le plus grand ensemble connexe contenant ce sommet.*

Définition 1.10 *Un **arbre** est un graphe connexe sans cycle.*

*Un **arbre couvrant** d'un graphe G est un arbre qui contient tous les sommets de G.*

*Une **forêt** est un graphe dont toutes les composantes connexes sont des arbres.*

*Une **forêt couvrante** d'un graphe G est une forêt qui contient tous les sommets de G.*

Définition 1.11 *Un graphe G est dit **complet** si et seulement si $\forall x, y \in V(G), \{x, y\} \in E(G)$. On note K_n le graphe complet de taille n.*

Définition 1.12 *Dans un graphe G, on appele* **étoile** *de centre u, notée $B_G(u)$, le sous-graphe de G constitué de u, de ses voisins $N_G(u)$ et de ses arêtes incidentes $I_G(u)$.*

Plus généralement, on appele **boule** *de centre u et de rayon k, notée $B_G(u, k)$ ou simplement $B(u, k)$ le sous-graphe de G constitué des chemins (sommets et arêtes) issus de u de longueur inférieure ou égale à k, i.e.,*

$V(B_G(u, k)) = \{v \in V(G) \mid dist_G(u, v) \leq k\}$.

$E(B_G(u, k)) = \{\{v, w\} \in E(G) \mid dist(u, v) \leq k - 1 \text{ et } dist(u, w) \leq k\}$.

Définition 1.13 *Dans un graphe G, un ensemble de sommets $S \subseteq V(G)$ est dit* **dominant** *si tout sommet de $V(G) \setminus S$ a un voisin dans S.*

Définition 1.14 *Un* **isomorphisme** *d'un graphe simple G vers un graphe simple H est une bijection $f : V(G) \to V(H)$ telle que $uv \in E(G)$ si et seulement si $f(u)f(v) \in E(H)$. On dit que G est* **isomorphe** *à H, noté $G \cong H$, si et seulement si il existe un isomorphisme de G vers H.*

1.2 Graphes étiquetés

Les graphes sur lesquels nous travaillons sont étiquetés par un ensemble d'étiquettes L, qui peut être fini ou infini. On supposera l'ensemble L muni d'un ordre total, noté $<_L$.

Un graphe étiqueté, noté (G, λ), est un graphe G muni d'une fonction d'étiquetage $\lambda : V(G) \cup E(G) \to L$ qui associe une étiquette à chaque sommet $v \in V(G)$ et à chaque arête $e \in E(G)$. Lorsque la fonction d'étiquetage n'aura pas à être explicitée, les graphes $(G, \lambda_G), (H, \lambda_H)$,... seront dénotés $\mathbf{G}$, $\mathbf{H}$,... ; on dit que le graphe G est le graphe **sous-jacent** de $\mathbf{G}$.

On dit que l'étiquetage d'un graphe (G, λ) est **uniforme** si et seulement si il existe deux étiquettes $\alpha, \beta \in L$ telles que pour tout sommet $v \in V(G), \lambda(v) = \alpha$ et pour toute arête $e \in E(G)$, $\lambda(e) = \beta$.

Les notions de graphes partiels, sous-graphes, sous-graphes induits s'étendent aux graphes étiquetés, avec préservation de l'étiquetage :

Définition 1.15 *Un graphe $\mathbf{H} = (H, \eta)$ est un graphe partiel (resp. sous-graphe, sous-graphe induit) d'un graphe $\mathbf{G} = (G, \lambda)$ si H est un graphe partiel (resp. sous-graphe, sous-graphe induit) de G et pour tout $x \in V(H) \cup E(H), \lambda(x) = \eta(x)$.*

Définition 1.16 *Deux graphes étiquetés sont dits* **isomorphes** *si et seulement si leurs graphes sous-jacents sont isomorphes et qu'on peut passer de l'étiquetage de l'un à l'étiquetage de l'autre par une bijection ϕ.*

Une **occurrence** *de $\mathbf{G} = (G, \lambda)$ dans* **G'** *$= (G', \lambda')$ est un isomorphisme entre $\mathbf{G} = (G, \lambda)$ et un sous-graphe $\mathbf{H} = (H, \eta)$ de* **G'** *tel que* **G** *et* **H** *ont le même étiquetage : $\phi(\lambda) = \eta$ avec $\phi = fonction\ identite$ (autrement dit $\eta = \lambda$).*

Définition 1.17 *Soit $\mathbf{G} = (G, \mu)$ un graphe étiqueté. On parle de* **sur-étiquetage**, *ou d'***étiquetage produit** *lorsque la fonction d'étiquetage $\mu : V(G) \to L^{n \in \mathbb{N}}$ associe à chaque élément étiqueté du graphe plusieurs valeurs de L. On désigne alors par* **registre** *les différentes composantes des étiquettes ainsi formées.*

1.3 Etiquetage des sommets et des brins d'arêtes

Dans ce document, il nous arrivera fréquemment de considérer des graphes dont les brins d'arêtes, et non les arêtes, sont étiquetés. Lorsque les graphes manipulés sont des graphes simples, cela revient à associer à chaque sommet une étiquette pour chacun de ses voisins. Nous distinguons ici la notion de brins d'arêtes de la notion traditionnelle de « ports » pour souligner, dans un contexte de graphes dynamiques, le fait que ces brins peuvent exister même après que l'arête correspondante a été supprimée.

Définition 1.18 *Un **brin d'arête**, en contexte statique, peut être défini comme une paire constituée d'un sommet v et d'une arête e telle que e est adjacente à v. Dans un contexte dynamique, un **brin d'arête à la date** t peut être défini comme une paire constituée d'un sommet v et d'une arête e telle que e est adjacente à v à la date t ou a été adjacente au sommet v à une date antérieure à t. Cela représente le fait qu'un brin d'arête peut continuer à exister après suppression de l'arête correspondante. La suppression du brin dans ce contexte doit alors être effectuée de manière explicite.*

Définition 1.19 *Un graphe dont les sommets et les brins d'arêtes sont étiquetés, noté (G_λ) est un graphe G muni d'une fonction d'étiquetage $\lambda : V(G) \cup (V(G) \times E(G) \mid e \in I_G(v)) \rightarrow L$ qui associe une étiquette à chaque sommet $v \in V(G)$ et à chaque paire $\{v, e\} \in V(G) \times E(G) \mid e \in I_G(v)$. En cas de suppression d'arête (en contexte dynamique), le brin correspondant conserve son étiquetage.*

La figure 1.1 donne l'exemple d'un graphe simple dont les sommets et les brins d'arêtes sont étiquetés.

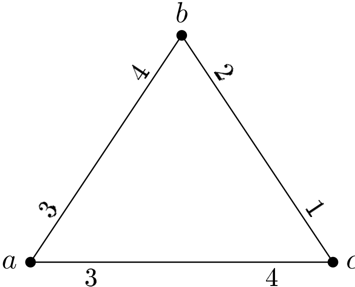


FIG. 1.1 – Exemple d'étiquetage de sommets et de brins d'arêtes dans un graphe simple

Observation 1 Il ne faut pas confondre l'étiquetage des brins d'arêtes avec une numérotation de ports, appelée couramment *étiquetage de ports*, où les sommets attribuent à chacun de leurs voisins un numéro local unique permettant de les distinguer les uns des autres. Nous pouvons néanmoins considérer l'étiquetage de ports comme un cas particulier d'étiquetage de brins d'arêtes.

Lorsque la fonction d'étiquetage n'aura pas à être explicitée, les graphes $(G, \lambda_G), (H, \lambda_H)$,... seront également dénotés $\mathbf{G}$, $\mathbf{H}$,... ; leur étiquetage sera dit uniforme s'il existe deux étiquettes $\alpha, \beta \in L$ telles que pour tout sommet $v \in V(G), \lambda(v) = \alpha$ et pour toute paire

$\{v, e\} \in V(G) \times E(G) \mid e \in I_G(v),\ \lambda(\{v, e\}) = \beta$.

Les notions de graphes partiels, sous-graphes, sous-graphes induits sont redéfinies de la manière suivante :

Définition 1.20 *Un graphe* $\mathbf{H} = (H, \eta)$ *est un graphe partiel (resp. sous-graphe, sous-graphe induit) d'un graphe* $\mathbf{G} = (G, \lambda)$ *si et seulement si* H *est un graphe partiel (resp. sous-graphe, sous-graphe induit) de* G *et pour tout* $x \in V(H) \cup (V(H) \times E(H) \mid e \in I_H(v)), \lambda(x) = \eta(x)$.

Les notions d'isomorphisme, d'occurence et de sur-étiquetage restent les mêmes que pour les graphes dont les sommets et les arêtes sont étiquetés.

1.4 Réétiquetages de graphes

Définition 1.21 *Une* ***relation de réécriture*** *entre deux graphes étiquetés* $\mathbf{G}$ *et* $\mathbf{H}$*, notée* $\mathbf{G}\ \mathcal{R}\ \mathbf{H}$ *est une relation binaire qui transforme* $\mathbf{G}$ *en* $\mathbf{H}$.

Définition 1.22 *Nous dirons qu'une relation de réécriture* $\mathcal{R}$ *entre* $\mathbf{G} = (G, \lambda)$ *et* $\mathbf{H} = (H, \eta)$ *est une* ***relation de réétiquetage*** *si et seulement si* $G = H$*, autrement dit si la structure du graphe est préservée par la relation et que seules les étiquettes changent.*

Chapitre 2

Modèles de calculs et formalismes associés

Sommaire

2.1 Systèmes distribués **14**
2.1.1 Définitions et contexte 14
2.1.2 Modèles de communication 16
2.1.3 Principaux modèles de calculs locaux 17
2.2 Réétiquetages de graphes et calculs locaux **18**
2.2.1 Représentation du réseau 19
2.2.2 Représentation des calculs 19
2.2.3 Exemple : construction d'un arbre couvrant 20
2.3 Adaptation à un cadre dynamique **21**
2.3.1 Calculs locaux et contexte dynamique 21
2.3.2 Représentation du réseau 21
2.3.3 Représentation des calculs 21
2.3.4 Quelques précisions à travers un exemple 23
2.4 Quelques algorithmes **27**
2.4.1 Algorithme de propagation - version basique 27
2.4.2 Algorithme de propagation avec reprise 27
2.4.3 Comptage - version 1 28
2.4.4 Comptage - version 2 30
2.4.5 Comptage - version 3 (et élection probabiliste) 31

Les problèmes rencontrés dans les systèmes distribués, bien que très variés d'un point de vue applicatif, peuvent généralement se réduire à un petit nombre de problèmes plus fondamentaux, tels qu'acheminer une information entre deux nœuds distants, construire une structure recouvrante, déterminer le nombre de participants, en choisir certains parmi d'autres, *etc.* Plus précisément, il est possible de réduire à un ou plusieurs problèmes fondamentaux la majorité des problèmes applicatifs de manière à ce que l'existence d'une solution pour les uns implique l'existence d'une solution pour les autres. Nous pouvons citer parmi ces problèmes fondamentaux les plus étudiés : l'élection (distinction d'un sommet), le *nommage* (attribution d'une identité unique à chaque sommet), le *comptage* (dénombrement

des sommets), la construction d'arbres ou d'anneaux couvrants, ou encore la détection de la terminaison d'un de ces algorithmes.

L'étude de ces problèmes théoriques, qui permettent d'abstraire des problèmes concrets, nécessite à son tour l'abstraction des objets étudiés, à savoir, en ce qui nous concerne, les systèmes distribués dans les réseaux dynamiques. Dans ce chapitre, nous présentons le cadre dans lequel s'inscrivent nos travaux, des notions les plus générales (systèmes distribués) aux notions les plus précises (calculs locaux et réétiquetages de graphes). Nous présentons ensuite les adaptations effectuées sur le modèle de calcul et sur le formalisme des réétiquetages de manière à les rendre applicables aux graphes (et donc aux réseaux) dynamiques. Le chapitre se termine avec la présentation de quelques algorithmes.

2.1 Systèmes distribués

2.1.1 Définitions et contexte

Lorsque l'on tente de définir ce qu'est un système distribué, il est d'usage de citer les définitions qu'en ont donné Andrew Tanenbaum et Leslie Lamport. Nous ne dérogeons pas à la règle dans cette section, mais observons quelques différences entre les systèmes distribués ainsi décrits et ceux que nous considérons.

Andrew Tanenbaum

"A collection of independent computers that appears to its users as a single coherent system." [TS01] que l'on pourrait traduire par *"Un système distribué est un ensemble d'ordinateurs indépendants qui apparaît à un utilisateur comme un système unique et cohérent."*

Un système distribué est donc un ensemble d'ordinateurs indépendants. Plus généralement, nous parlerons d'entités de calcul autonomes, ou simplement de *nœuds*. Dans cette définition d'Andrew Tanenbaum, le système se comporte globalement comme une boîte noire remplissant des fonctions précises pour son environnement extérieur (l'utilisateur). Ce point de vue ne convient pas au cadre des réseaux mobiles ad hoc (et plus généralement des réseaux dynamiques), où le système distribué et son environnement sont très souvent fondus l'un dans l'autre, non seulement parce qu'ils sont susceptibles d'interagir au niveau de chaque nœud, mais aussi parce que ces systèmes ont fréquemment pour finalité leur propre organisation.

Leslie Lamport

"You know you have a distributed system when the crash of a computer you've never heard of stops you from getting any work done." [LLW] que l'on pourrait traduire par *"Un système distribué est un système qui vous empêche de travailler quand une machine dont vous n'avez jamais entendu parler tombe en panne"*.

Là encore, la définition n'est pas tout à fait adaptée au cadre de notre étude, car nous considérons l'instabilité des entités de calcul et des liens de communication comme faisant partie intégrante de la vie du réseau (pannes byzantines mises à part). Le système ne doit donc pas changer son mode de fonctionnement, ni s'effondrer complètement, lorsque de tels événements se produisent.

Dans le cadre de ce document

Dans le cadre de ce document, nous considérons un système distribué comme étant un environnement au sein duquel plusieurs entités de calcul autonomes (ordinateurs, terminaux mobiles communicants, processeurs, processus, etc.), aussi appelées « nœuds », collaborent pour atteindre un objectif commun. Nous appelons « réseau » l'ensemble constitué des nœuds et des liens de communication qu'ils partagent. Par « lien de communication » entre deux nœuds nous entendons la possibilité d'une communication entre ces nœuds, que cette possibilité soit exploitée ou non. Nous considérons des liens bidirectionnels et symétriques (le terme de *voisins réciproques* est utilisé dans [TWB03]). Deux nœuds qui partagent un lien de communication sont dits « voisins ». Dans un cadre dynamique, nous considérons que l'ensemble des nœuds et des liens de communication est variable dans le temps : des nœuds peuvent apparaître ou disparaître spontanément à tout moment, ce qui correspond par exemple à l'allumage ou à l'extinction d'un périphérique par un utilisateur. De même, des liens de communication entre nœuds peuvent apparaître ou disparaître au cours du temps, ce qui correspond par exemple au déplacement d'un terminal en environnement sans fil, ou à la survenue d'un obstacle qui se déplace (véhicule, individu, *etc.*) et "bloque" la communication.

Dans ce document, nous entendons par « algorithmique distribuée » le domaine qui s'attache à étudier les problèmes fondamentaux des systèmes distribués, c'est à dire à trouver des solutions à ces problèmes, lorsqu'elles existent, et à décrire les conditions requises par les solutions qui ne sont pas universelles. Ces conditions, ou hypothèses, peuvent parfois porter sur le déroulement même d'un algorithme, mais dans la plupart des cas, ce sont des hypothèses sur le réseau sous-jacent qui sont considérées. Par exemple, certains problèmes sont plus faciles à résoudre si tous les « nœuds » possèdent un identifiant unique (*p. ex.* une adresse mac ou IP), ou si l'on sait qu'aucun nœud n'est ou ne sera isolé du reste du réseau. Il peut être montré de la même manière que certains problèmes n'ont pas de solution du tout lorsqu'une condition particulière est vérifiée (*p. ex.* le nommage dans les réseaux anonymes avec certaines symétries [Ang80]). Si l'on considère uniquement des réseaux décentralisés, où chaque nœud exécute le même algorithme et ne communique qu'avec ses voisins directs, alors l'algorithmique distribuée revient aussi à étudier les comportements globaux qui peuvent émerger, de proche en proche, de ces comportements locaux.

Lorsque les problèmes « classiques » de l'algorithmique distribuée sont plongés dans un cadre dynamique, ils doivent souvent être redéfinis ; par exemple l'énumération peut être envisagée comme le maintien permanent d'une estimation, sur chaque nœud, du nombre de nœuds qui sont présents dans la même composante connexe. Les outils utilisés pour représenter les algorithmes doivent aussi être redéfinis afin de pouvoir prendre en compte les « événements topologiques » qui se produisent sur le réseau, c'est-à-dire, pour chaque nœud, l'apparition ou la disparition possible d'un lien de communication vers un voisin. Dans ce document, lorsque nous parlons de réseau dynamique sans précision supplémentaire, cela désigne un réseau anonyme (pas d'hypothèse sur l'existence d'un identifiant unique pour les nœuds), dont les liens de communication sont bidirectionnels, et dont tout nœud et tout lien de communication est susceptible de disparaître à tout moment. Sauf mention contraire, nous considérons des réseaux dont la dynamique est imprévisible, *c.-à-d.*, qu'aucune information sur l'évolution topologique du réseau n'est connue. De même, nous considérons par défaut que la mobilité des nœuds n'est pas contrôlée par l'application

(ce qui pourrait être le cas de robots mobiles [SMR06]). Enfin, aucune hypothèse n'est faite quant à l'existence d'une horloge globale dans le réseau, *c.-à-d.* que les systèmes étudiés sont, en ce sens, asynchrones.

2.1.2 Modèles de communication

Différents modèles de communication peuvent être utilisés au sein d'un système distribué. Parmi les modèles les plus répandus se trouvent la boîte aux lettres, la mémoire partagée et l'envoi de messages.

Communication par boîte aux lettres

Dans ce modèle, chaque nœud possède une zone mémoire dans laquelle ses voisins peuvent écrire des données. Un nœud n_1 désirant communiquer avec un voisin n_2 doit ainsi écrire dans la zone mémoire de n_2 et, s'il en attend une réponse, la lire dans sa zone mémoire lorsque n_2 y aura écrit. Dans la plupart des cas, la lecture de ces données par le nœud sous-jacent a pour effet de les supprimer, on parle alors de consommation de messages.

Communication par registres

Dans ce modèle, chaque lien de communication se voit associer deux registres, un de chaque côté. Pour communiquer avec leurs voisins, les nœuds écrivent dans le registre local correspondant au lien de communication vers ce voisin. Chaque registre peut être accédé en lecture/écriture par le nœud local, et en lecture uniquement par le nœud distant.

Communication par mémoire partagée

Dans ce modèle, les différents nœuds qui désirent communiquer entre eux partagent une ressource mémoire commune. L'accès à cette ressource est généralement exclusif pour les écritures, mais peut être concurrent en lecture.

Communication par envoi de messages

Ce modèle de communication est celui qui est le plus fréquemment rencontré dans la littérature. Dans ce modèle, les nœuds sont reliés entre eux par des canaux de communication, et communiquent en s'échangeant des messages. Selon les variantes, les canaux peuvent être unidirectionnels ou bidirectionnels. Le nœud émetteur envoie un message en le « déposant » dans le canal et le nœud récepteur scrute le canal pour recevoir un éventuel message.

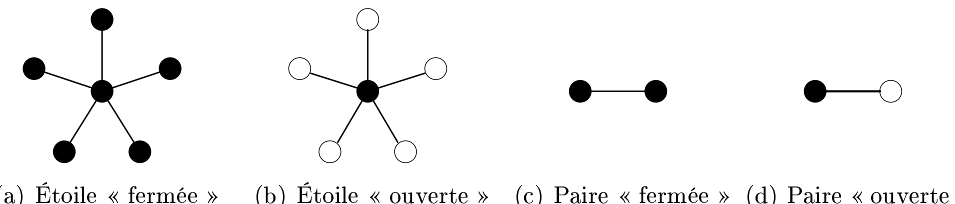
Abstraction du modèle de communication

Dans ce document, nous ne considérons que les systèmes asynchrones (sans horloge globale) dont les nœuds communiquent par envoi de messages. Les algorithmes présentés, qu'ils relèvent de l'existant ou de nos propres travaux, font cependant abstraction du modèle de communication en utilisant des calculs locaux. Cette abstraction sur le modèle de

communication permet, et c'est une de ses qualités, de pouvoir ensuite reporter les résultats négatifs (démonstrations d'impossibilités) vers tous les modèles de communications existants[1]. Les résultats positifs ne sont en revanche pas tous transposables dans tous les modèles de communication donnés, mais ils fournissent tout de même de bonnes indications sur la solution à adopter. Un pont entre les calculs locaux et le modèle de communication par envois de messages asynchrones, dans un contexte statique, a été proposé par Chalopin et Métivier dans [CM05] et des travaux de ce type portant sur les mémoires partagées sont en cours dans la même équipe.

2.1.3 Principaux modèles de calculs locaux

Les calculs locaux sont des modèles de calculs permettant de décrire les algorithmes indépendamment du modèle de communication sous-jacent. Chaque étape de calcul, *c.-à-d.* chaque opération de l'algorithme distribué, est spécifiée par un changement d'état portant sur un ensemble connexe de nœuds, séparés les uns des autres par une distance bornée. De nombreux modèles de calculs locaux peuvent être imaginés, mais les travaux dans ce domaine se bornent généralement à considérer ceux dans lesquels le champ d'action d'une étape de calcul est limité à une étoile (un nœud ainsi que tous ses voisins et les liens de communication qui le relient à eux) ou à une paire (deux nœuds directement voisins). Nous ne parlerons ici que de quatre de ces modèles de calculs locaux, qui sont ceux présentés figure 2.1. Différentes variantes de ces modèles, ainsi qu'une hiérarchisation de leurs puissances de calculs respectives sont étudiées dans [Cha06] et [CMZ04].



(a) Étoile « fermée » (b) Étoile « ouverte » (c) Paire « fermée » (d) Paire « ouverte »

Fig. 2.1 – Modèles de calculs locaux de différentes puissances

Calculs locaux sur des étoiles « ouvertes » - LC_1

Ce modèle, aussi appelé *Calculs Locaux 1*, ou LC_1, est présenté figure 2.1(b). Dans ce modèle, une étape de calcul consiste à modifier l'état du centre d'une boule de rayon 1 en fonction de l'état de tous les éléments de la boule. L'état du reste de la boule n'est, en revanche, pas modifié. Deux étapes de calcul peuvent être effectuées en parallèle si et seulement si elles se produisent sur des boules dont les centres sont éloignés d'une distance supérieure ou égale à 2.

[1] On pourrait cependant imaginer qu'un modèle basé sur certaines propriétés de la physique quantique permette de « dépasser » les notions de localité et de voisinage par la notion d'intrication [BBC$^+$93].

Calculs locaux sur des étoiles « fermées » - LC_2

Ce modèle, aussi appelé *Calculs Locaux 2*, ou LC_2, est présenté figure 2.1(a). Dans ce modèle, une étape de calcul consiste à modifier l'état de tous les éléments d'une boule de rayon 1 en fonction de leur état actuel. Deux étapes de calcul peuvent être effectuées en parallèle si et seulement si elles se produisent sur des boules disjointes.

Calculs locaux sur des paires « ouvertes »

Ce modèle, aussi appelé *Calculs Locaux sur les arêtes*, est présenté figure 2.1(d). Dans ce modèle, une étape de calcul revient à modifier l'état d'un nœud en fonction de son état et de celui d'un de ses voisins. Deux étapes de calcul peuvent être effectuées en parallèle si et seulement si elles se produisent sur des paires dont le nœud modifié est différent.

Calculs locaux sur des paires « fermées »

Ce modèle, aussi appelé *Calculs Locaux sur les sommets*, est présenté figure 2.1(c). Dans ce modèle, une étape de calcul revient à modifier l'état de deux nœuds voisins en fonction de leurs propres états. Deux étapes de calcul peuvent être effectuées en parallèle si et seulement si les paires correspondantes sont disjointes.

Réétiquetages des arêtes

Pour chacun de ces modèles, on peut distinguer plusieurs sous-modèles selon que les arêtes appartenant aux paires ou aux boules peuvent être réétiquetées ou non. Nous considérons dans ce document des modèles dont les sommets et les arêtes peuvent être réétiquetés.

Synchronisation entre voisins

Lorsque l'on considère le déroulement d'un algorithme de ce type il faut faire en sorte que les différents nœuds se mettent d'accord, avant chaque étape de calcul, sur un certain nombre de choix concernant par exemple la sélection du voisin avec lequel l'étape va se dérouler, le choix de l'opération qui va être effectuée, ou les rôles respectifs que les nœuds vont jouer dans l'opération. Cette phase qui précède le calcul est appelée synchronisation et fait l'objet du chapitre 3 de cette thèse. Ces synchronisations peuvent varier selon le modèle de calcul considéré.

2.2 Réétiquetages de graphes et calculs locaux

Cette section présente l'utilisation des graphes et des réétiquetages de graphes que l'on utilise pour représenter respectivement les réseaux, et les calculs locaux qui y sont effectués. Nous nous bornons ici à présenter les outils théoriques existants qui s'appliquent au contexte des réseaux statiques.

2.2.1 Représentation du réseau

Le réseau est représenté par un graphe simple non orienté $G = (V(G), E(G))$ dont les sommets $V(G)$ représentent les nœuds, et les arêtes $E(G)$ représentent les possibilités de communications (bidirectionnelles) entre nœuds. Les états des nœuds sont codés par des étiquettes sur les sommets correspondants, et les états des liens de communication par des étiquettes sur les arêtes. L'état global du réseau peut donc être donné à tout moment par un graphe étiqueté $\mathbf{G} = (G, \lambda)$ où λ est la fonction qui associe à chaque sommet de $V(G)$ et à chaque arête de E(G) une étiquette codant l'état du nœud ou du lien de communication correspondant. Chacune de ces étiquettes peut être composée de plusieurs registres (*sur-étiquetage*, définition 1.17 page 9). Dans ce document nous parlerons indistinctement d'étiquette à plusieurs registres, ou d'étiquettes multiples pour désigner le sur-étiquetage.

2.2.2 Représentation des calculs

Chaque étape de calcul consiste en un réétiquetage portant sur un sous-graphe de $\mathbf{G}$. Selon le modèle de calcul local choisi (et en se bornant à ceux de la figure 2.1 page 17), les sous-graphes impliqués peuvent être des étoiles ou des paires de sommets.

Formellement, une règle de réétiquetage est définie par un couple *(précondition, réétiquetage)* tel que si la précondition de la règle est vérifiée par un sous-graphe de $\mathbf{G}$, alors la partie correspondante est réétiquetée comme spécifié par la règle. La figure 2.2 donne l'exemple d'une règle abstraite utilisant le modèle de calcul en étoile fermée (a) exprimée dans le langage logique LIDiA [MO04].

<u>*préconditions :*</u> – $\lambda(v_0) = label_1$ – $\exists v \in V(B(v_0)) \setminus \{v_0\} \mid \lambda(v) = label_2$ <u>*réétiquetage :*</u> – $\lambda'(v_0) := label_3$	Si un sommet v_0 est étiqueté $label_1$ et qu'un de ses voisins est étiqueté $label_2$, alors v_0 est réétiqueté $label_3$.

Fig. 2.2 – Exemple de définition d'une règle de réétiquetage avec le langage LIDiA

Lorsque les préconditions et les réétiquetages le permettent, la règle peut également être décrite de manière graphique, comme montré figure 2.3.

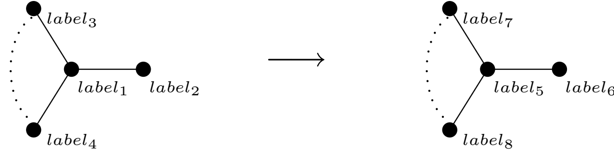


Fig. 2.3 – Représentation graphique d'une règle de réétiquetage LC_2

Un algorithme peut être constitué d'une ou plusieurs règles de ce type. Dans le cas d'un algorithme à plusieurs règles, des mécanismes de contrôle comme les « priorités entre règles » ou les « contextes interdits » [LMS95] peuvent être utilisés pour déterminer dans quel ordre et sous quelles conditions les utiliser. Un système distribué complet consiste

alors en une affectation d'états initiaux et un ensemble de règles de réétiquetage. Nous ne considérons dans ce document que des systèmes distribués dont chaque nœud possède les mêmes règles de réétiquetage, les états initiaux pouvant en revanche varier d'un nœud à l'autre.

2.2.3 Exemple : construction d'un arbre couvrant

Ce paragraphe présente l'exemple d'un algorithme comportant une seule règle de réétiquetage qui permet de construire de proche en proche un arbre couvrant sur le réseau. Les sommets ont une étiquette qui peut prendre pour valeur A ou N selon qu'il fait déjà partie de l'arbre ou non. De même, les arêtes faisant partie de l'arbre sont étiquetées 1, les autres 0. Cet algorithme suppose qu'un sommet, la racine de l'arbre, est distingué des autres au début du calcul, et étiqueté A. Tous les autres sommets sont initialement étiquetés N et toutes les arêtes 0. La règle de réétiquetage codant l'algorithme est présentée figure 2.4. On peut la lire de la manière suivante : lorsqu'un sommet de l'arbre voit un sommet ne faisant pas partie de l'arbre, le nouveau sommet et l'arête correspondante sont tout deux intégrés à l'arbre, et le sommet intégré devient capable, à son tour, d'intégrer d'autres sommets voisins. Un exemple de déroulement de cet algorithme est donné figure 2.5.

FIG. 2.4 – Règle codant un algorithme d'arbre couvrant

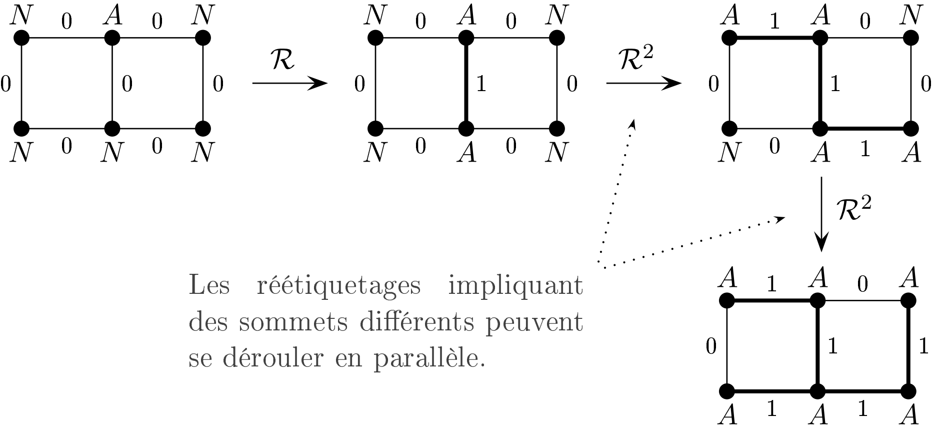


FIG. 2.5 – Exemple d'exécution de l'algorithme de la figure 2.4

Il est évident que le déroulement d'un tel algorithme, ainsi que son résultat final, dépendent de la manière dont les sommets se synchronisent (*i.e.*, se choisissent les uns les autres) pour appliquer les règles. Nous rappelons que la problématique de la synchronisation, ainsi que son adaptation à un cadre dynamique, font l'objet du chapitre 3.

2.3 Adaptation à un cadre dynamique

Dans cette section, nous présentons les différentes propositions que nous avons fait autour des réétiquetages de graphes pour les utiliser dans un contexte dynamique.

2.3.1 Calculs locaux et contexte dynamique

Dans le cadre d'un MANet, et plus généralement de tout réseau dynamique imprévisible, les liens de communication entre nœuds peuvent rompre à tout moment. Dans ce contexte, toute hypothèse d'une communication réussie entre nœuds non directement voisins devient une hypothèse optimiste. Il apparaît donc nécessaire, plus encore que dans un cadre statique, de ne faire intervenir, pour chaque étape de calcul, que des nœuds directement voisins.

2.3.2 Représentation du réseau

Le réseau est représenté par un graphe simple non orienté $G = (V, E)$ dont les sommets $V(G)$ représentent les nœuds, et les arêtes $E(G)$ représentent les possibilités de communications (bidirectionnelles) entre nœuds. Les apparitions ou disparitions de possibilités de communication (que nous appelons *liens* dans ce document) entre nœuds sont représentées par des ajouts ou des suppressions d'arêtes dans $E(G)$. Les mises en route ou extinctions (ou pannes) de nœuds sont représentés par des ajouts ou des suppressions de sommets dans $V(G)$. On distingue ainsi les périphériques qui s'éteignent des périphériques qui s'isolent. Dans la suite, nous désignerons par « graphe dynamique » ce type de graphe.

Les états des nœuds sont codés par des étiquettes sur les sommets correspondants. L'état des liens de communication est codé par des étiquettes sur les brins d'arêtes. Cela permet de représenter de façon explicite le fait que l'état algorithmique d'un lien de communication est mémorisé sur chacun des nœuds extrémités, et non sur le lien lui-même. Les nœuds possèdent donc toujours cette information même lorsque le lien a été rompu. Par effet de bord, cet étiquetage permet également de représenter des états asymétriques sur les arêtes, comme par exemple une relation *père/fils* dans un arbre, en affectant des valeurs différentes de part et d'autre de l'arête. L'état global du réseau peut être donné à tout moment par un graphe étiqueté $\mathbf{G} = (G, \lambda)$ où λ est la fonction qui associe à chaque sommet de $V(G)$ une étiquette codant l'état du nœud correspondant, et à chaque couple $(v \in V(G), e \in E(G)) \mid e \in I_G(v)$ une étiquette codant l'état *algorithmique* du lien de communication correspondant, vu depuis le sommet v.

2.3.3 Représentation des calculs

Nous distinguons ici deux types de réétiquetages : ceux qui sont causés par les règles normales de l'algorithme et ceux qui résultent d'une réaction de l'algorithme à un événement topologique.

Règles de réétiquetages normales

Comme dans un cadre statique, les opérations de l'algorithme distribué sont représentées par des règles de réétiquetage portant sur un ensemble restreint d'arêtes et de

sommets voisins. Pour des raisons de synchronisation, nous avons écarté de nos travaux les modèles (a) et (b) de la figure 2.1 page 17, pour lesquels une étape de calcul implique tous les sommets d'une boule de rayon 1, afin de nous concentrer sur les modèles (c) et (d), plus réalistes dans un contexte dynamique car chaque étape n'implique que deux sommets directement voisins. La figure 2.6 donne l'exemple d'une règle de réétiquetage abstraite, utilisant le modèle de calcul de la paire fermée (c), exprimée dans un langage logique proche de LIDiA. Une représentation graphique de la même règle est donnée figure 2.7. A ce stade, le seul élément qui diffère du cadre statique est l'étiquetage des brins d'arêtes (remplaçant l'étiquetage des arêtes).

préconditions : – $\lambda(v_0) = label_1$ – $\lambda(v_0, (v_0, v_1)) = label_2$ *réétiquetage :* – $\lambda'(v_0) := label_4$ – $\lambda'(v_1) := label_7$ – $\lambda'(v_0, (v_0, v_1)) := label_5$ – $\lambda'(v_1, (v_0, v_1)) := label_6$	Si un sommet v_0 est étiqueté $label_1$ et que le brin d'arête sortant vers le voisin considéré est étiqueté $label_2$, alors v_0 se réétiquette en $label_4$, v_1 en $label_7$, et leurs brins respectifs de l'arête commune sont réétiquetés $label_5$ et $label_6$.

FIG. 2.6 – Exemple de règle abstraite portant sur des étiquettes de sommets et de brins d'arêtes

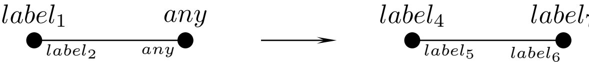


FIG. 2.7 – Représentation graphique de la règle de la figure 2.6

Gestion des événements topologiques

Vus depuis chaque nœud du réseau, les événements topologiques se réduisent à deux types : les apparitions de liens de communication vers un nouveau voisin, et les ruptures de liens de communication vers un voisin actuel. Sur le graphe dynamique représentant le réseau, cela se traduit par l'ajout ou la suppression d'une arête incidente. Le modèle de calcul doit donc permettre à l'algorithme de réagir à ces deux types d'événements.

Apparition d'un nouveau lien de communication. Gérer cet événement consiste à faire en sorte que toute nouvelle arête soit exploitable par l'algorithme. Cela peut se faire en attribuant aux nouvelles arêtes un étiquetage par défaut (figure 2.8), propre à l'algorithme considéré. Cet étiquetage est également appliqué à toutes les arêtes présentes au début de l'algorithme.

Rupture d'un lien de communication. Lorsqu'un lien de communication est rompu entre deux nœuds, la possibilité doit être donnée à chacun des nœuds concernés de réagir par un traitement adapté. Pour ce faire, une étiquette spéciale est ajoutée à chaque brin d'arête, indiquant l'état d'activité du lien, notée λ_{act} valant *on* à la création de l'arête.

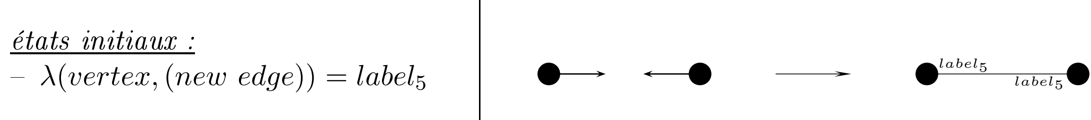


Fig. 2.8 – Apparition d'un lien de communication

Lorsque le lien de communication correspondant est rompu, l'arête est supprimée mais ses deux brins persistent et leur étiquette d'activité prend la valeur off (figure 2.9).

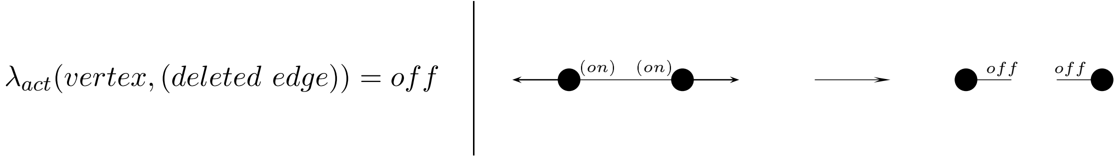


Fig. 2.9 – Rupture d'un lien de communication

Réaction de l'algorithme. Avec un tel mécanisme, il devient possible de spécifier, via de simples règles de réétiquetage, la manière dont un algorithme doit réagir à la rupture d'un lien de communication. Ces règles, que nous appelons « règles de réaction à la rupture », n'impliquent qu'un seul sommet, et lui affectent l'état qui a été prévu en pareil cas. Ces règles sont toujours prioritaires sur les « règles normales », *i.e.,* les règles impliquant plusieurs sommets. La figure 2.10 donne l'exemple d'une telle règle, décrite avec les notations logiques et graphiques.

préconditions : – $\lambda(v_0) = N$ – $\lambda(v_0, (v_0, v_{lost})) = 1$ – $\lambda_{act}(v_0, (v_0, v_{lost})) = off$ *réétiquetage :* – $\lambda'(v_0) := J$	N off 1 ⟶ J	Si un sommet étiqueté N a perdu une arête vers un de ses voisins (v_{lost}), alors que le brin de cette arête est étiqueté 1, alors il se réétiquette en J, et le brin est définitivement supprimé.

Fig. 2.10 – Exemple de règle de réaction à la rupture

Le système complet. Le système distribué complet peut être donné par 1) une affectation d'états initiaux, 2) un ensemble de règles de réétiquetage normales et, optionnellement, 3) un ensemble de règles de réaction aux ruptures. Ces dernières ayant pour vocation première de rétablir la cohérence locale de l'état d'un sommet suite à un événement topologique, nous avons choisi de les rendre prioritaires sur les règles normales. Cette priorité permet de garantir qu'aucune opération *normale* ne pourra être effectuée par un sommet dont l'état est incohérent.

2.3.4 Quelques précisions à travers un exemple

Nous présentons ici un algorithme complet reposant sur des réétiquetages de graphes dynamiques. Cet algorithme, publié dans [Cas06], est complètement décentralisé et ne fait

aucune hypothèse sur le réseau sous-jacent en terme de dynamique ou de propriétés (comme l'existence d'identifiants uniques).

Présentation de l'algorithme

La construction de structures couvrantes tels que les arbres, dans les réseaux instables, ad hoc ou plus généralement dynamiques, ont fait l'objet de nombreux travaux. On peut citer de manière non exhaustive la compilation que Gärtner a réalisé [GÖ3] autour des algorithmes auto-stabilisants et les travaux qui ont été effectués plus récemment autour des réseaux de capteurs ad hoc (*p.ex.* [FISR07]).

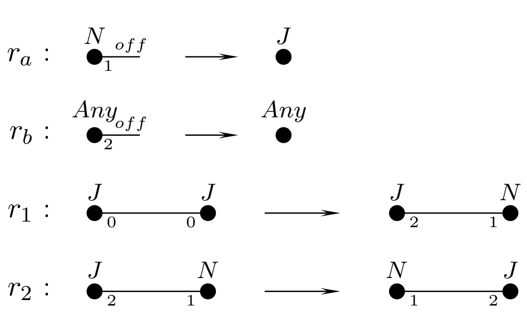
L'algorithme de la forêt couvrante que nous proposons, et dont les règles sont données par l'algorithme 1 page suivante, a pour objectif de maintenir une forêt d'arbres couvrants sur un graphe dynamique, sans effectuer la moindre hypothèse de centralisation, de stabilité, ou d'existence d'identités. Comme pour la majorité des exemples présentés dans ce document, les nœuds exécutent tous le même algorithme. Les principales propriétés du système sont les suivantes :

1. En se plaçant au niveau d'abstraction des graphes, l'algorithme ne requiert pas l'existence d'identifiants uniques pour les sommets (avec les technologies actuelles, son implémentation effective nécessiterait cependant de tels identifiants).
2. Les sommets sont tous initialisés de la même manière, ce qui en fait un algorithme complètement décentralisé.
3. Lorsqu'un lien de communication interne à un arbre est rompu, la mise à jour globale des deux composantes ainsi partitionnées se fait en une et une seule opération localement à chaque extrémité de l'arête rompue.
4. La convergence en un temps fini vers un seul et unique arbre par composante connexe n'est pas garantie pour un temps fini (ce point précis sera discuté dans le chapitre 3).

Le fonctionnement de l'algorithme est le suivant : initialement, chaque sommet forme un arbre isolé dont il est la racine (étiquette J). Lorsque deux sommets racines se retrouvent synchronisés pour l'application d'une règle de réétiquetage, ils appliquent la règle r_1, qui fusionne les deux arbres. Lors de cette fusion, l'un des sommets devient père de l'autre (et racine de l'arbre résultant). Le fils, de son côté, se réétiquette en N (le choix de l'attribution de chacun de ces deux rôles ne relève pas du niveau d'abstraction auquel se situent les règles). L'arête qu'ils partagent prend alors de part et d'autre les valeurs d'étiquette *1* et *2*, marquant ainsi l'orientation *père/fils*. Plus précisément, l'étiquette *1* dénote un lien en direction de la racine, l'étiquette *2* un lien s'en éloignant. A chaque fois que deux sommets se synchronisent pour appliquer une règle, ils tentent en priorité d'appliquer r_1 (ordre naturel des règles). Plusieurs arbres peuvent ainsi fusionner, de proche en proche, ne conservant à chaque fois qu'une seule racine. Si r_1 n'est pas applicable, ils tentent d'appliquer r_2. Si r_2 n'est pas applicable, ils n'appliquent aucune règle ensemble. La règle r_2 implique une racine et un de ses sommets *fils*, et a pour effet d'inverser le rôle de ces deux sommets, le fils devenant racine, et la racine devenant fils. Pour les autres sommets de l'arbre, la règle r_2 ne modifie pas la route locale vers la racine, ce changement n'a donc pas besoin d'être propagé. Quand un nœud détecte la rupture d'un de ses liens, il exécute la règle de réaction à la rupture correspondante. Cette réaction est alors prioritaire sur les règles normales (comme l'indique l'ordre des règles donné par l'algorithme 1). Si le lien perdu

était un lien vers la racine (étiquette *1*), alors le sommet correspondant s'autoproclame racine (règle r_a), en se réétiquetant J. Si le lien rompu est un lien vers un fils (étiquette *2*), le résidu (brin) de l'arête correspondante est supprimé sans traitement supplémentaire (règle r_b). Si l'étiquette a pour valeur 0 (zéro), alors les deux sommets correspondant n'entretiennent aucun lien direct de parenté dans l'arbre, le brin est donc supprimé sans traitement particulier. Cette dernière règle est considérée comme implicite et n'est donc pas donnée par l'algorithme.

Algorithme 1 Maintien d'une forêt d'arbres couvrants.



Quand un sommet perd une arête qui mène vers la racine, il devient racine.

Quand un sommet perd une arête qui ne mène pas vers la racine, il n'effectue aucun traitement particulier.

Si les deux sommets sont racine, seul l'un d'eux le reste, et les arbres fusionnent sur l'arête impliquée. Les étiquettes de l'arête sont mises à jour pour indiquer la nouvelle relation *père/fils*.

La racine se déplace au sein de l'arbre.

Un exemple de séquence d'exécution de cet algorithme est proposé figure 2.11 page suivante. En (a), la topologie est couverte par deux arbres différents, bien que le graphe soit connexe. La seule règle applicable, r_2, est appliquée à divers endroits (potentiellement plusieurs fois). En (b), deux sommets étiquetés J (sommets racines) se retrouvent en vis-à-vis. S'ils choisissent de travailler ensemble, la règle r_1 est appliquée, ce qui a pour effet de fusionner les deux arbres. En (c), une rupture topologique a lieu entre deux sommets parents dans l'arbre. Celui des deux qui était le fils s'autoproclame alors racine de son arbre (règle r_a) en se réétiquetant J, ce qui lui permet d'appliquer (à nouveau) les règles r_1 ou r_2 (en l'occurrence, r_1 a été appliquée entre (d) et (e)).

Preuves de quelques propriétés

Dans cette partie, nous supposons que l'étiquette J signifie que le sommet sous-jacent possède un jeton. Tout sommet racine de son arbre est donc un sommet qui a le "jeton" de son arbre. Cela ne change rien à l'algorithme, mais simplifie les explications des preuves ci-dessous.

Proposition 2.1 *Il y a à tout moment au moins un jeton par arbre (après réaction à la rupture en une étape).*

Preuve 2.1 *Initialement, tous les sommets sont étiquetés J et forment un arbre à un élément. La propriété est donc vraie au début du calcul.*
Chaque rupture dans un arbre engendre deux nouveaux arbres, l'un d'entre eux ayant un jeton, l'autre l'ayant perdu. Dans l'arbre sans jeton, le sommet impliqué dans la rupture a perdu son père. Ce sommet applique donc r_a et régénère un nouveau jeton. □

Proposition 2.2 *Il ne peut y avoir deux jetons dans le même arbre.*

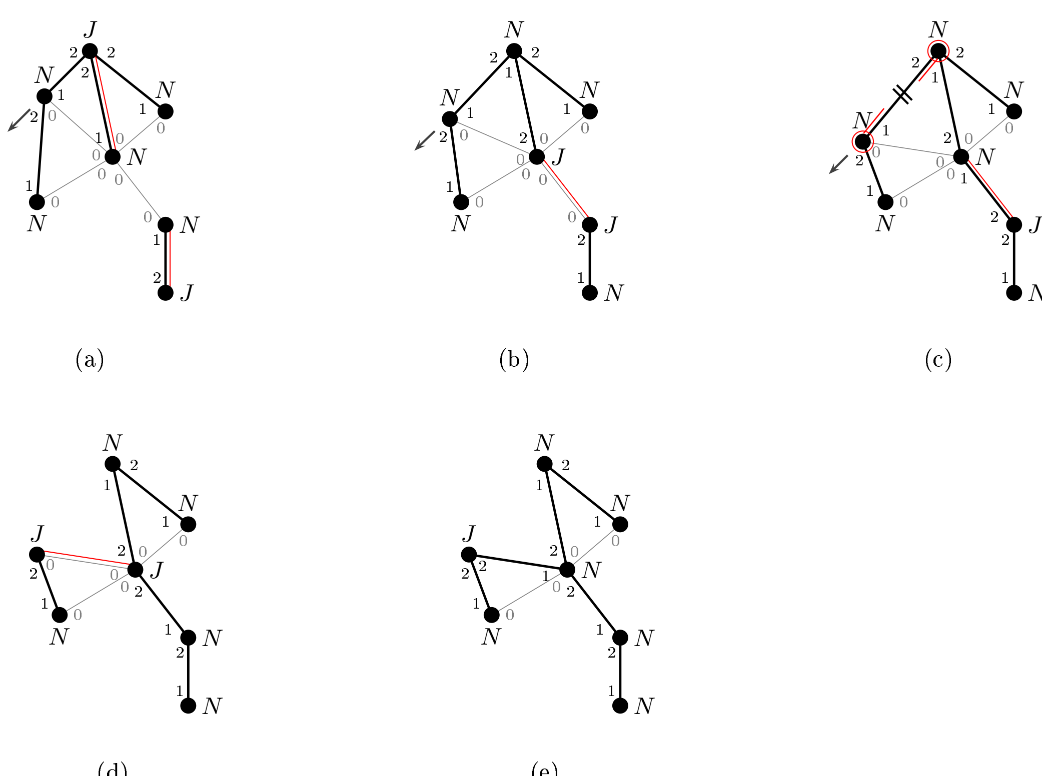


FIG. 2.11 – Exemple de séquence d'exécution de l'algorithme 1 (les éléments doublés en rouge sont ceux relativement auxquels les traitements vont s'effectuer)

Preuve 2.2 *La seule règle provoquant la création d'un jeton est r_a. S'il y a deux jetons dans le même arbre, alors deux cas sont possibles :*

1. *r_a a été appliquée alors qu'il y avait déjà un jeton dans l'arbre.*
2. *r_a a été appliquée deux fois dans l'arbre simultanément.*

Cas 1 : Si r_a a été appliquée, alors le sommet qui l'a appliquée avait un de ses brins d'arêtes étiqueté 1 à off*. Or, de manière triviale, un brin d'arête étiqueté 1 implique que l'arête correspondante conduisait vers le jeton. Le jeton se trouvait donc dans la composante de l'arbre dont il a été déconnecté.*
$\Longrightarrow$ *cas 1 impossible.*
Cas 2 : Soit v et v' les deux sommets du même arbre ayant appliqué r_a. Il y a trois possibilités :

1. *v est ancêtre de v' dans l'arbre.*
   $\longrightarrow$ *impossible, puisque v' n'applique la règle que s'il a perdu son père dans l'arbre.*
2. *v' est ancêtre de v dans l'arbre.*
   $\longrightarrow$ *impossible, puisque v n'applique la règle que s'il a perdu son père dans l'arbre.*
3. *v et v' ont un ancêtre commun.*
   $\longrightarrow$ *impossible, puisque v et v' n'appliquent la règle que s'ils ont perdu leur père et ne sont donc pas dans le même arbre.*

$\Longrightarrow$ *cas 2 impossible.* □

2.4 Quelques algorithmes

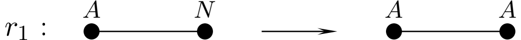
Cette section donne quelques exemples d'algorithmes à base de réétiquetages de graphes dynamiques. Ces exemples sont constitués de deux algorithmes de propagation d'information, deux algorithmes de comptage, et un algorithme d'élection probabiliste.

2.4.1 Algorithme de propagation - version basique

L'algorithme 2 a pour but de propager une information dans le réseau, de proche en proche, depuis un émetteur donné. La règle r_1 codant cet algorithme peut être lue intuitivement de la manière suivante : soit A (resp. N) la valeur d'étiquette codant le fait que le nœud sous-jacent possède (resp. ne possède pas) l'information propagée. Si un nœud étiqueté A rencontre un nœud étiqueté N, alors il lui transmet l'information. Le second nœud, qui possède désormais l'information (étiquette A), peut alors à son tour la transmettre à ses voisins, et ainsi de suite.

Algorithme 2 Propagation d'une information (version basique)

Etats initiaux : un sommet distingué étiqueté A, les autres sommets étiquetés N

r_1 :


Cet algorithme n'est pas spécifique aux réseaux dynamiques, et les traitements locaux qu'il effectue seront les mêmes dans tous les types de réseaux. Son comportement global, en revanche, ne dépendra pas des mêmes paramètres dans un contexte statique et dans un contexte dynamique. Si le réseau est statique et si le graphe correspondant est connexe, alors tous les nœuds seront informés en un nombre d'opérations inférieur ou égal au diamètre du graphe. La connexité du graphe est donc une condition à la fois nécessaire et suffisante au succès de l'algorithme. Dans un cadre dynamique, les choses sont différentes. En supposant que le nombre total de nœuds ne varie pas dans le temps, ce qui est déjà une hypothèse, la connexité du graphe à un instant donné ne sera ni une condition nécessaire, ni une condition suffisante pour que tous les nœuds soient informés. Il se peut par exemple que tous les nœuds soient informés sans qu'à aucun moment le graphe n'ait été connexe. Pour cet algorithme comme pour d'autres qui suivent, la caractérisation des conditions nécessaires et suffisantes sur la dynamique du réseau sera effectuée au chapitre 4.

2.4.2 Algorithme de propagation avec reprise

L'algorithme de propagation avec reprise (ou propagation de document), présenté algorithme 3 page 29, est un peu plus complexe que l'algorithme 2. Il permet de coder la propagation d'une information de taille conséquente, telle un fichier, et dont le transfert peut être interrompu à tout moment. Lorsqu'un transfert est interrompu entre un nœud émetteur et un nœud récepteur, le nœud récepteur peut reprendre le transfert là où il s'était arrêté avec n'importe quel nœud qui possède la suite du document. La réception du

document par un nœud donné est forcément séquentielle, quel que soit le nombre d'émetteurs impliqués au cours du temps. Enfin, on considère que le document est constitué d'une suite de blocs numérotés.

L'algorithme fonctionne de la manière suivante : chaque sommet possède deux étiquettes (ou deux registres d'étiquette). La première, qui représente l'état du sommet, vaut R ou N selon que le sommet est en train de recevoir le document ou non. La seconde étiquette, numérique, indique le dernier bloc que le sommet possède (numéro du dernier bloc reçu). Initialement, tous les sommets ont leur première étiquette à N. Celui qui possède le document complet a le nombre de blocs du document comme seconde étiquette, tandis que les autres ont cette seconde étiquette à 0. La règle r_1 établit une connexion entre deux nœuds s'ils ne possèdent pas le même nombre de blocs et si celui qui en a le moins n'est pas déjà en train de recevoir (étiquette N et non R). Ce sommet est alors réétiqueté R pour indiquer qu'il est désormais en position de récepteur, et l'arête correspondante est étiquetée de manière à indiquer l'orientation des transferts (2 côté émetteur, 1 côté récepteur). Le récepteur, étiqueté R, ne peut plus appliquer la règle r_1 en tant que récepteur (il ne peut ainsi recevoir que d'un émetteur à la fois). Il peut en revanche l'appliquer en tant qu'émetteur, devenant ainsi simultanément récepteur pour un nœud et émetteur pour un ou plusieurs autres nœuds. La règle r_2 code le cas où un récepteur a obtenu tout ce que son émetteur pouvait lui donner, la connexion est alors fermée. Le transfert du document n'est pas, pour autant, forcément terminé et le nœud peut redevenir récepteur pour un autre émetteur (on peut supposer que le document est complet lorsque le dernier bloc reçu contient le caractère EOF). Enfin, la règle r_3 code le transfert effectif d'un bloc de document entre un émetteur et un récepteur qui sont connectés.

En ce qui concerne les réactions aux ruptures, il y a deux cas possibles : 1) L'arête rompue n'a aucun rôle particulier (étiquette 0 de part et d'autre). Les brins sont alors supprimés de part et d'autre sans traitement supplémentaire. Cette règle est, comme pour l'algorithme de la forêt couvrante, considérée comme implicite ; 2) L'arête rompue était le support d'une connexion entre deux nœuds. Dans ce cas, l'émetteur supprime simplement le brin local correspondant (règle r_b). Le récepteur, quant à lui, applique r_a, et reprend l'étiquette N, ce qui lui permettra de se reconnecter ultérieurement à un autre émetteur.

2.4.3 Comptage - version 1

L'algorithme 4 page suivante a pour fonction de compter le nombre maximal de participants dans un réseau. Le fonctionnement de l'algorithme est le suivant : un sommet compteur est distingué au début de l'exécution. Ce sommet possède deux étiquettes, l'une indiquant qu'il est compteur (étiquette C), l'autre indiquant le nombre de participants qu'il a déjà compté. L'état initial du compteur est donc $(C, 1)$, car il s'inclut dès le départ dans le décompte des sommets. Les autres sommets sont initialement étiquetés N, valeur signifiant qu'ils n'ont pas encore été comptés. A chaque fois que le compteur se synchronise avec un sommet non compté, la règle r_1 est appliquée entre eux. Cette règle a pour effet d'incrémenter le compteur et de marquer le sommet compté comme désormais *compté* (étiquette F). Chaque sommet ne peut ainsi être compté qu'une fois.

Soit $\mathcal{F}$ l'ensemble des sommets étiquetés F (sommets comptés), soit $\mathcal{N}$ l'ensemble des sommets étiquetés N (sommets non comptés), et $\mathcal{C}$ l'ensemble des sommets étiquetés C (ensemble à un élément). Si l'on suppose que le nombre de sommets ne varie pas dans le

Algorithme 3 Propagation d'un document avec reprise

Etats initiaux : un sommet distingué étiqueté $(N, nblocs)$, $nblocs$ étant le nombre de blocs constituant le document. Tous les autres sommets étiquetés $(N, 0)$, tous les brins d'arêtes étiquetés 0.

Règle	Avant		Après	Commentaire
r_a :	R, i ; brin 1, arête off	$\longrightarrow$	N, i	Quand un sommet perd l'arête par laquelle il recevait le document, il se repositionne comme n'étant plus en train de recevoir, ce qui lui permet de se reconnecter à un autre émetteur *via* r_1.
r_b :	Any, i ; brin 2, arête off	$\longrightarrow$	Any, i	Quand un sommet perd une arête par laquelle il envoyait le document, il n'effectue aucun traitement particulier.
r_1 :	Any, i —(0, 0)— $N, j < i$	$\longrightarrow$	Any, i —(2, 1)— R, j	Quand un sommet possède une plus grande part du document qu'un autre sommet, et que ce dernier n'a pas de réception en cours, alors une connexion s'opère entre eux, permettant ensuite l'envoi du document *via* r_3.
r_2 :	Any, i —(2, 1)— $R, j = i$	$\longrightarrow$	Any, i —(0, 0)— N, j	Si parmi deux sommets connectés le récepteur a obtenu tout ce que l'émetteur pouvait lui donner, alors la connexion est fermée.
r_3 :	Any, i —(2, 1)— $R, j < i$	$\longrightarrow$	Any, i —(2, 1)— $R, j + 1$	Si parmi deux sommets connectés le récepteur n'a pas encore toutes les parties que l'émetteur possède, alors la partie suivante du document est transmise.

Algorithme 4 Comptage avec un sommet compteur distingué.

Etats initiaux : un sommet distingué étiqueté $(C, 1)$, les autres sommets étiquetés N

r_1 : C, i —— N $\longrightarrow$ $C, i+1$ —— F

temps, alors cet algorithme possède quelques invariants.

A tout instant du calcul, chaque sommet a l'une des trois étiquettes F, N ou C. Le nombre total de sommets est donc égal à la somme des sommets ayant ces étiquettes :

$$\begin{aligned} nbtotal &= |\mathcal{F}| + |\mathcal{N}| + |\mathcal{C}| \\ &= |\mathcal{F}| + |\mathcal{N}| + 1 \text{ car il n'y a qu'un compteur.} \end{aligned}$$

Soit c le sommet compteur, on désigne par $c.counter$ son deuxième registre d'étiquette, totalisant à tout moment les sommets qu'il a déjà comptés.

Proposition 2.3 *A tout instant du calcul,* $c.counter = 1 + |\mathcal{F}|$.

Preuve 2.3 *Au début du calcul,* $c.counter = 1$ *et* $|\mathcal{F}| = 0$*, donc* $c.counter = 1 + |\mathcal{F}|$*. La seule opération qui peut modifier ces variables est l'application de la règle* r_1*. Or, à chaque application de* r_1*,* $c.counter$ *est incrémenté de* 1*, et un sommet étiqueté* N *devient étiqueté* F*, donc* $|\mathcal{F}|$ *est également incrémentée de* 1. □

Proposition 2.4 $|\mathcal{N}| = 0 \implies c.counter = nbtotal$

Preuve 2.4
$nbtotal = |\mathcal{F}| + |\mathcal{N}| + 1$
donc $|\mathcal{N}| = 0 \implies nbtotal = |\mathcal{F}| + 1$
or, à tout moment, $c.counter = |\mathcal{F}| + 1$
donc $|\mathcal{N}| = 0 \implies c.counter = nbtotal$ □

Le compteur possède donc le décompte total du nombre de sommets si $|\mathcal{N}| = 0$, autrement dit si tous les sommets, excepté le compteur, sont étiquetés F.

Lorsqu'aucune hypothèse n'est faite sur l'invariance du nombre total de sommets du réseau, l'algorithme 4 n'a pas de configuration finale. Il fournit alors dans le meilleur des cas une borne inférieure sur cette valeur. Les conditions nécessaires et suffisantes sur le réseau, pour que tous les participants soient comptés, sont étudiées au chapitre 4.

2.4.4 Comptage - version 2

L'algorithme 5 a pour fonction de compter le nombre maximal de participants (*i.e.* obtenir une borne inférieure sur le nombre de sommets du graphe). Contrairement à l'algorithme 4, il n'y a pas de compteur distingué au début de l'algorithme et tous les sommets démarrent avec le même état, ce qui en fait un algorithme totalement décentralisé. Le fonctionnement de l'algorithme est le suivant : chaque sommet possède deux registres d'étiquettes. Le premier registre représente son état : *compteur*(C) ou *compté(F)*. Le deuxième, si le sommet est compteur, représente le nombre de sommets qui ont été comptés par lui ou par ceux qu'il a lui-même compté. Ce registre est appelé *valeur de comptage*. Initialement tous les sommets sont étiquetés C et 1. A chaque fois que deux sommets compteurs appliquent la règle r_1, l'un des deux reste compteur, et l'autre devient compté. Le sommet compteur ajoute alors à sa valeur de comptage le nombre correspondant à la valeur de comptage de l'autre sommet.

Algorithme 5 Comptage avec multiples sommets compteurs, qui fusionnent.

Etats initiaux : tous les sommets étiquetés $(C, 1)$

r_1 : $\overset{C,i}{\bullet}$———$\overset{C,j}{\bullet}$ ⟶ $\overset{C,i+j}{\bullet}$———$\overset{F}{\bullet}$

Soit $\mathcal{F}$ l'ensemble des sommets étiquetés F (sommets comptés) et $\mathcal{C}$ l'ensemble des sommets étiquetés C (sommets compteurs). Si l'on suppose que le nombre de sommets ne varie pas dans le temps, alors cet algorithme possède quelques invariants.

A tout instant du calcul, chaque sommet a l'une des deux étiquettes F ou C. Le nombre total des sommets est donc égal à la somme des sommets ayant ces étiquettes :

$$nbtotal = |\mathcal{F}| + |\mathcal{C}|$$

Pour tout sommet c compteur, on désigne par *c.counter* son deuxième registre d'étiquette, correspondant à sa valeur de comptage.

Proposition 2.5 *A tout instant du calcul,* $\sum_{c \in \mathcal{C}} c.counter = nbtotal$

Preuve 2.5
Initialement, chaque sommet est étiqueté $(C, 1)$*. La proposition est donc vérifiée au début du calcul. La seule opération pouvant modifier ces variables est l'application de la règle* r_1*. Or, lorsque* r_1 *est appliquée entre deux sommets, un seul des deux reste étiqueté* C*, et sa valeur de comptage est augmentée de la valeur de comptage de l'autre. La somme des valeurs de comptage des sommets étiquetés* C *est donc conservée par l'application de* r_1*.*$\square$

Proposition 2.6 $|\mathcal{C}| = 1$ *(avec* $\mathcal{C} = \{x\}$*)* $\Longleftrightarrow x.counter = nbtotal$

Preuve 2.6
Évident, en considérant l'invariant $\sum_{c \in C} c.counter = nbtotal$

Le décompte du nombre total de sommets est donc atteint sur un sommet si et seulement si ce sommet est le dernier étiqueté C, tous les autres étant alors étiquetés F.

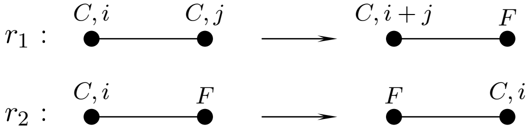
2.4.5 Comptage - version 3 (et élection probabiliste)

Objectif de l'algorithme

Comme les deux algorithmes précédents, l'algorithme 6 a pour objectif de compter le nombre maximal de participants du réseau. Le fonctionnement de cet algorithme est quasiment le même que celui de l'algorithme 5, à ceci près qu'une règle permettant aux compteurs de changer de nœud est ajoutée. Ainsi, quand les compteurs ne fusionnent pas, ils se déplacent dans le graphe.

Algorithme 6 Comptage avec multiples compteurs qui fusionnent et circulent.

Etats initiaux : tous les sommets étiquetés $(C, 1)$



La règle r_2 qui a été ajoutée par rapport à l'algorithme précédent, ne fait qu'échanger les étiquettes des deux sommets qui l'appliquent. L'invariant et la configuration finale de l'algorithme précédent ne sont donc pas modifiés pour cet algorithme. La circulation des compteurs ne garantit pas qu'ils auront tous fusionné en un temps fini. Cependant, lorsque la dynamicité du réseau est faible en regard des opérations distribuées, la circulation permet de lever des configurations temporairement bloquantes, comme par exemple si les compteurs sont éloignés les uns des autres. L'étude de probabilité de fusion dans ce type d'algorithme (et notamment de l'algorithme de la forêt couvrante), en regard des opérations de circulation de compteur (assimilable à un jeton) et de la dynamique du réseau, fait actuellement l'objet de travaux menés à l'université du Havre par l'équipe RI2C.

L'algorithme 6 peut être, sur plusieurs points, comparé à l'algorithme 7 page suivante. Ce dernier a pour but d'élire un unique sommet dans le réseau (objectif nécessitant un nombre de sommets invariant dans le temps). Cet algorithme d'élection a été utilisé par Angluin *et al.*, notamment dans [AAD$^+$06]. Le principe de fonctionnement est le même que pour l'algorithme 6 : initialement, tous les sommets sont candidats. Au gré des opérations, les candidats s'éliminent les uns les autres (règle r_1) ou se déplacent (règle r_2). S'il ne reste qu'un candidat à l'arrivée, alors il peut être considéré comme élu. Le problème qui se pose est alors, comme pour le comptage, de trouver le moyen de garantir qu'il n'y aura plus qu'un candidat au bout d'un temps fini. Dans [AAD$^+$06], les auteurs font l'hypothèse que, si une configuration topologique donnée peut laisser place à une autre, et que la première se répète indéfiniment, alors la seconde se répétera également indéfiniment. Partant de cette hypothèse, les auteurs montrent qu'il existera de manière répétée des configurations permettant aux candidats de fusionner. Il n'en reste pas moins que dans un contexte réel,

pour une durée d'exécution finie et une topologie dont la dynamique est aléatoire, aucune garantie de succès d'un tel algorithme n'a jamais, à notre connaissance, été donnée.

Algorithme 7 Élection probabiliste, basée sur la circulation et la fusion de "candidats".

Etats initiaux : tous les sommets étiquetés C

r_1 : C — C $\longrightarrow$ C — F

r_2 : C — F $\longrightarrow$ F — C

Chapitre 3

Synchronisation entre voisins

Sommaire

3.1 Procédures existantes 34
3.1.1 Élections locales pour calculs locaux LC_1 34
3.1.2 Élections locales pour calculs locaux LC_2 34
3.1.3 Algorithme du *Rendezvous* pour calcul entre paires 35
3.1.4 Bilan et commentaires 36
3.2 Synchronisation à la demande 37
3.2.1 Motivations 37
3.2.2 Principe de fonctionnement 37
3.3 Synchronisation à la demande avec critères 38
3.3.1 Application des critères 39
3.3.2 Ordonnancement des voisins 40
3.3.3 Adaptation au contexte 41
3.4 Critères étendus pour les réseaux sans fil 43
3.5 Exemple d'utilisation 43

Lorsqu'une étape de calcul distribué est effectuée par plusieurs nœuds, cela suppose que ces nœuds ont au préalable réussi une synchronisation, c'est à dire qu'ils ont réussi à décider de leur participation commune dans l'opération. Différentes techniques de synchronisation existent et le choix de celle employée dépend de nombreux critères, parmi lesquels l'adéquation avec le modèle de calcul utilisé : les synchronisations amenant à des opérations sur des étoiles ne sont pas les mêmes que celles amenant à des opérations sur des paires de nœuds. Un second critère peut être par exemple la part laissée au hasard dans la procédure, ainsi que l'équité souhaitée entre les nœuds, ces deux considérations allant souvent de pair. Enfin, et c'est ce qui nous intéresse tout particulièrement, la prise en compte du caractère statique, ou dynamique, de l'environnement dans lequel la synchronisation s'opère est aussi un facteur fondamental. Ces différents aspects peuvent avoir un impact important sur l'exécution des algorithmes, allant dans certains cas jusqu'à en régir le déroulement complet.

Dans ce chapitre nous passons en revue les principales méthodes de synchronisation existant dans un cadre statique, à savoir les *élections locales*, utilisées conjointement aux modèles de calculs en étoiles LC_1 et LC_2, et l'algorithme du *Rendezvous* utilisé entre autres

lorsque le calcul porte sur des paires de nœuds. Nous discutons ensuite des contraintes liées aux réseaux dynamiques et présentons différentes variantes d'un mode de synchronisation que nous avons développé spécifiquement pour ces réseaux.

3.1 Procédures existantes

Cette section présente une compilation d'informations concernant les principales procédures de synchronisation utilisées pour mettre en œuvre des calculs locaux, à savoir les élections locales et la procédure du *Rendezvous*. Ces informations ont été compilées à partir des articles [MSZ00], [MSZ02] et [MSZ03], de Métivier, Saheb-Djahromi et Zemmari.

3.1.1 Élections locales pour calculs locaux LC_1 - ⁂ - [MSZ02]

Chaque sommet v tire au hasard, selon une loi de probabilité uniforme, un entier dans un ensemble $\{1, .., N\}$ convenu à l'avance. Le sommet v envoie ensuite cet entier à tous ses voisins et reçoit les entiers de chaque voisin. Il est élu dans la boule $B(v, 1)$ (étoile dont il est le centre) si son entier est strictement plus grand que l'entier de chacun de ses voisins ; cette élection locale est appelée élection L_1 et sa procédure est donnée par l'algorithme 8. A chaque élection réussie, une étape de calcul de type LC_1 (section 2.1.3 page 17 - calculs sur des étoiles « ouvertes ») peut être effectuée : le centre collecte les valeurs des étiquettes de chaque sommet de l'étoile et, selon applicabilité du traitement, change la valeur de son étiquette.

Algorithme 8 Élection locale L_1

Entre chaque étape de calcul distribué, chaque sommet v répète en boucle les actions suivantes :

1. *v tire un entier $rand(v)$ au hasard ;*
2. *v envoie $rand(v)$ à ses voisins ;*
3. *v reçoit les entiers de tous ses voisins.*

Le sommet v est localement élu dans $B(v, 1)$ si $rand(v)$ est strictement supérieur à tous les entiers qu'il a reçus de ses voisins.

3.1.2 Élections locales pour calculs locaux LC_2 - ⁂ - [MSZ02]

Chaque sommet v tire au hasard, selon une loi de probabilité uniforme, un entier dans l'ensemble $\{1, .., N\}$. Le sommet v envoie cet entier à tous ses voisins et reçoit les entiers de chaque voisin. Une fois ces entiers reçus, il envoie à chaque voisin w le plus grand des entiers provenant des autres voisins. Le sommet v est élu dans la boule $B(v, 2)$ si son entier $rand(v)$ est strictement plus grand que chacun des entiers qu'il a reçu au cours de la procédure ; cette élection locale est appelée élection L_2 et sa procédure est donnée par l'algorithme 9 page suivante. A chaque élection réussie, une étape de calcul de type LC_2 (section 2.1.3 page 18 - calculs sur des étoiles « fermées ») peut être effectuée sur $B(v, 1)$: le centre collecte les valeurs des étiquettes de chaque sommet de l'étoile et, selon applicabilité du traitement, change potentiellement la valeur de toutes ces étiquettes.

Algorithme 9 Élection locale L_2

Entre chaque étape de calcul distribué, chaque sommet v répète en boucle les actions suivantes :

1. *v tire un entier $rand(v)$ au hasard ;*
2. *v envoie $rand(v)$ à ses voisins ;*
3. *v reçoit les entiers de tous ses voisins.*
4. *pour chaque voisin w de v, v envoie à w le nombre $m_{v/w}$, qui est le plus grand entier que v a reçu de ses autres voisins ;*
5. *v reçoit les entiers de tous ses voisins ;*

Le sommet v est localement élu dans $B(v, 2)$ si $rand(v)$ est strictement supérieur à tous les entiers qu'il a reçus de ses voisins.

3.1.3 Algorithme du *Rendezvous* - ↔ - [MSZ00] et [MSZ03]

Chaque sommet v choisit un de ses voisins $c(v)$ au hasard. Il y a rendez-vous entre v et $c(v)$ si $c(v)$ a également choisi v. On dit alors que v et $c(v)$ sont synchronisés. A ce stade ils peuvent effectuer une étape de calcul portant, en lecture comme en écriture, sur leurs deux sommets (section 2.1.3 page 18 - calculs sur des paires). La procédure est décrite par l'algorithme 10.

Algorithme 10 Algorithme du rendez-vous

Chaque sommet v répète à l'infini les actions suivantes :

1. *v choisit au hasard un de ses voisins $c(v)$;*
2. *v envoie 1 à $c(v)$;*
3. *v envoie 0 à tout voisin différent de $c(v)$;*
4. *parallèlement, v reçoit les messages de tous ses voisins.*

Il y a un rendez-vous entre v et $c(v)$ si v reçoit 1 de $c(v)$.

La figure 3.1 montre quelques exemples de rendez-vous dans un graphe, suite aux tirages de 1 et de 0 effectués par chaque sommet.

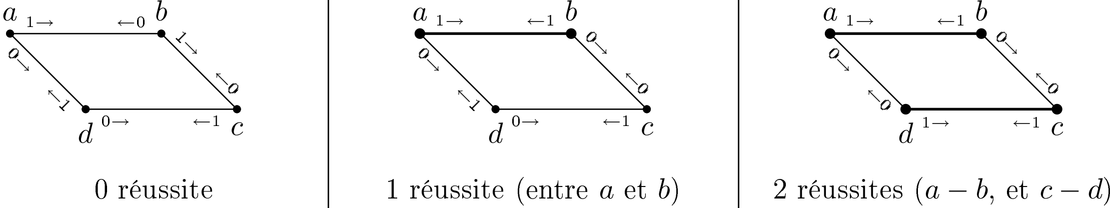


Fig. 3.1 – Quelques rendez-vous dans un graphe

Le choix de la règle de réétiquetage à appliquer peut ensuite être régi par différents mécanismes, comme les *priorités* ou les *contextes interdits* étudiés par Litovsky, Métivier

et Sopena dans [LMS95][1].

Algorithme de poignée de mains avec agenda dynamique

Un dérivé de l'algorithme du rendez-vous, l'algorithme de poignée de mains avec agenda dynamique, a récemment été proposé dans [HMR+06]. Ce dernier se base sur des nombres tirés aléatoirement (selon une loi de probabilité uniforme) dans l'intervalle $[0, 1]$ sur chaque sommet, et pour chaque voisin. Le nombre moyen de synchronisations par round est supérieur à celui de l'algorithme du *Rendezvous*, mais l'algorithme suppose une horloge globale, hypothèse que nous pouvons difficilement faire dans un cadre ad hoc et dynamique. A titre d'information, la procédure est tout de même présentée par l'algorithme 11.

Algorithme 11 Algorithme de poignée de main avec agenda dynamique

---

Entre chaque étape de calcul distribué, chaque sommet v répète en boucle les actions suivantes :

1. *v génère un réel $t_v(u)$, correspondant à une date aléatoire comprise entre 0 et 1, pour chacun de ses voisins u ;*
2. *v attend jusqu'à ce qu'un des événements suivants survienne :*
   - *v reçoit 1 d'un de ses voisins w ; il envoie alors 0 à tous les autres.*
     (Il y a poignée de main entre v et w)
   - *la date courante arrive en un des $t_v(u)$ et v n'a toujours rien reçu de ses voisins ; v envoie alors 1 à u, et 0 à tous les autres voisins.*
     (Il y a poignée de main entre v et u)
   - *la date courante arrive en 1.*
     (Le round termine sans que v n'ait pris part à une poignée de main)

Une étape de calcul peut alors être effectuée entre chaque paire de sommets ayant échangé une poignée de main.

3.1.4 Bilan et commentaires

Dans les trois principales procédures décrites précédemment (LC_1, LC_2 et *Rendezvous*), le hasard et la recherche d'équité jouent un rôle important. Dans le cas du *Rendezvous* par exemple, le choix du voisin est aléatoire (*c.f.* action 1). En supposant que les tirages suivent une loi de probabilité uniforme, cela implique que tout sommet peut sélectionner chacun de ses voisins avec la même probabilité (*i.e.* équitablement). En revanche, la probabilité pour qu'un sommet soit sélectionné par chacun de ses voisins varie selon le degré du voisin (*p. ex.* si v est le seul voisin de u, alors u le choisira systématiquement). La répartition des rendez-vous dans un graphe dépend donc essentiellement de la structure du graphe [MSZ03]. La structure du graphe influe également sur les procédures L_1 et L_2. Par exemple, on peut trouver dans [MSZ02] une étude sur le nombre moyen de synchronisations auquel on peut s'attendre à chaque round dans le cas particulier des arbres. Des probabilités de réussite peuvent également être calculées localement à chaque nœud. Pour L_1 et L_2, la probabilité

[1] En réalité, la mise en œuvre de ces mécanismes de contrôle (qui ont été prouvés équivalents l'un à l'autre) nécessite une synchronisation plus forte (et correspond au modèle de calcul LC_2).

qu'a chaque sommet v d'être élu vaut respectivement $p_1 = \frac{1}{d(v)+1}$, et $p_2 = \frac{1}{N_2(v)}$, avec N_2 le nombre de sommets à distance inférieure ou égale à 2 de v, plus 1 (v lui-même). Pour le *Rendezvous*, la probabilité que deux sommets voisins v et u se choisissent respectivement est, pour chaque cycle, de $\frac{1}{d(v)\times d(u)}$. Cette dernière baisse donc fortement avec l'augmentation du degré moyen du graphe (de l'ordre du carré). De plus amples informations sur ces procédures peuvent être trouvées dans les articles précédemment cités.

Dans chacune de ces trois premières procédures, les sommets attendent de manière bloquante de recevoir des messages de tous leurs voisins. Ce jeu de réceptions bloquantes a un effet secondaire important : chaque cycle d'exécution de la procédure, pour le *Rendezvous* comme pour L_1 et L_2, est dépendant sur chaque sommet des cycles d'exécution des voisins. Ce phénomène synchronise l'exécution de la procédure dans ce qu'on appelle des *rounds*. Métivier, Mosbah, Ossamy et Sellami ont montré dans [MMOS04] qu'en utilisant l'algorithme du *Rendezvous* pour synchroniser un réseau, l'écart de round entre deux sommets éloignés d'une distance *dist* devenait inférieur ou égal à *dist*. Cet effet synchronisant, souhaitable dans certains cas, a des implications qui peuvent être gênantes. L'une d'elles, problématique dans notre contexte, est la non-tolérance du système aux ruptures des liens de communication. En effet, une telle rupture entraînerait le blocage des deux sommets concernés (les réceptions sont bloquantes) et, de proche en proche, celui du réseau tout entier. De plus, les procédures de synchronisation étant en phase d'un nœud à l'autre, cela implique que les nœuds doivent attendre à chaque round que tous leurs voisins aient terminé les calculs dans lesquels ils sont impliqués. De proche en proche, ce phénomène propage les attentes sur tout le réseau, et peut ralentir à terme un nœud situé à grande distance de l'endroit où a été générée une attente.

3.2 Synchronisation à la demande

3.2.1 Motivations

Dans un cadre statique, une synchronisation aléatoire équitable n'aura pas d'impact, performances mises à part, sur le bon déroulement d'un algorithme. Dans un cadre dynamique, où les liaisons sont volatiles, peut en revanche apparaître la nécessité de réussir rapidement une synchronisation, quitte à rendre la procédure moins équitable. Dans cette section, nous proposons un nouveau mode de synchronisation où les sommets s'adressent directement des demandes de synchronisation. Ce type de synchronisation, qui garde une part d'aléatoire, élimine les inconvénients liés à l'existence des *rounds*, et ouvre la voie à certaines extensions présentées dans la section suivante.

3.2.2 Principe de fonctionnement

A chaque itération, v tire un nombre aléatoire *rand* et attend *rand* unités de temps (étape 1), durée pendant laquelle il réceptionne d'éventuelles demandes de synchronisation (réception d'un bit à 1) provenant de ses voisins, et les accepte en mettant un terme à l'itération courante. S'il n'a rien reçu à l'issue de cette étape, v choisit un de ses voisins au hasard (étape 2) et lui envoie une demande de synchronisation (étape 3). Il attend ensuite une réponse pendant un temps fini (*time out*), temps au-delà duquel il considère avoir reçu 0. Parallèlement aux étapes 2, 3 et 4, v peut également recevoir des demandes venant de

ses voisins, mais les refuse systématiquement en leur renvoyant 0. Si aucun échange de 1 n'a eu lieu à l'issue de l'étape 4, v recommence la procédure. L'algorithme 12 rappelle les principales étapes de cette procédure.

Algorithme 12 Synchronisation à la demande

1. *v attend une durée aléatoire bornée ;*
2. *v choisit un de ses voisins $c(v)$;*
3. *v envoie une demande (1) à $c(v)$;*
4. *v reçoit une réponse (0 ou 1) de $c(v)$;*

parallèlement à ces étapes, v peut recevoir des demandes de synchronisation. Si une demande arrive pendant l'étape 1, il y répond positivement, en renvoyant 1 à l'expéditeur. Si une demande arrive pendant les autres étapes, il y répond négativement, en renvoyant 0.

Il y a synchronisation entre v et un de ses voisins s'ils se sont échangé un 1.

Lorsqu'une synchronisation est réussie entre deux sommets, ces derniers tentent d'effectuer une étape de calcul de type paire, ouverte ou fermée. Les éventuelles demandes qui arriveraient pendant une étape de calcul sont systématiquement refusées.

Observation 2 *Cet algorithme suppose qu'un nœud peut émettre et recevoir simultanément sur le réseau. Cette hypothèse n'est pas gênante lorsqu'on se place à un niveau applicatif, où les primitives de communication transitent généralement par des pilotes de périphérique effectuant les multiplexages nécessaires. Si, en revanche, on désire implémenter cet algorithme à très bas niveau et sans faire de multiplexage, alors on devra envisager un fonctionnement légèrement dégradé où les demandes de synchronisation ne seront prises en compte que pendant l'étape 1. Le seul inconvénient qui en résulte est une levée plus fréquente de time out lors des demandes de synchronisation.*

A propos de l'étape 1 : Le choix d'une durée aléatoire a été décidé afin d'éviter aux nœuds d'entrer en phase les uns avec les autres, ce qui aurait pu entraîner la répétition de circuits de dépendance entre nœuds *via* les demandes et les réponses qu'ils s'adressent et par conséquent des répétitions stériles dans l'exécution de la procédure. La valeur du délai d'attente, ou plus précisément de l'intervalle à l'intérieur duquel ce délai doit être choisi, n'est pas spécifié par la procédure. Cette question fait actuellement dans notre équipe l'objet de simulations sur des réseaux de capteurs sans fil, simulations dont les premiers résultats obtenus seront évoqués au chapitre 7.

3.3 Synchronisation à la demande avec critères

Nous présentons ici une adaptation du mode de synchronisation à la demande. Cette adaptation permet de privilégier certains liens de communication plutôt que d'autres. Les modifications apportées à la procédure ne concernent que l'étape 2, à savoir le choix du voisin. L'idée directrice est, pour chaque nœud, d'attribuer une note à chacun de ses voisins selon des critères décidés lors de la conception ou lors du déploiement de l'algorithme. Il peut être intéressant par exemple, selon les besoins de l'application, de privilégier l'utilisation des liens selon qu'ils maximisent une des caractéristiques suivantes : stabilité,

instabilité, fort débit, faible latence, chute subite de qualité, *etc.* Il peut également être intéressant de privilégier certains liens en tenant compte de l'historique des communications et des traitements, par exemple en favorisant ceux qui n'ont jamais été utilisés (*rareté*), ont souvent été utilisés (*fréquence*), ont été marqués par une opération précise (*marquage*), *etc.*

3.3.1 Application des critères

Les critères portant sur la nature même du lien de communication sont appelés *Critères physiques*. Ils correspondent généralement à des informations que l'on peut acquérir de manière passive, par des écoutes régulières sur le voisinage[2]. Un score est ainsi calculé pour chaque voisin, sur la base d'un ou plusieurs critères qui peuvent être combinés par des opérations arithmétiques. Quelques exemples de mesure de tels critères sont donnés figure 3.2.

critère	score	caractéristique privilégiée
stabilité	$score(v) = \frac{1}{\lvert\Delta\omega(signal(v))\rvert}$	variation minimale du signal.
instabilité	$score(v) = \lvert\Delta\omega(signal(v))\rvert$	variation maximale du signal.
fort débit	$score(v) = \omega(signal(v))$	puissance maximale du signal.
chute du signal	$score(v) = -(\Delta\omega(signal(v)))$	chute maximale du signal.
⋮	⋮	⋮
sécurité	$score(v) = isEncrypted(v)?1 : 0$	chiffrement du canal.

Où $\omega(signal(v))$ désigne la puissance perçue du signal correspondant au lien de communication vers v, et où $\Delta\omega(signal)$ est calculé en tenant compte des dernières mesures pour ce même lien, lorsqu'elles existent (et en utilisant une valeur nulle sinon).

FIG. 3.2 – Calcul de quelques critères physiques

Les critères portant sur l'historique des communications et/ou des traitements sont appelés *Critères algorithmiques*. Le calcul de ces critères s'appuie généralement sur le déroulement des synchronisations et des réétiquetages avec chaque voisin. La figure 3.3 donne quelques exemples de critères algorithmiques. Les critères algorithmiques peuvent également être combinés entre eux par des fonctions arithmétiques. Ces formules sont alors appliquées sur le score résultant des critères physiques, comme indiqué figure 3.4.

Le score complet de chaque voisin peut finalement être exprimé de la manière suivante : $score(v) = algo \circ phys(v)$, où la fonction $phys()$ correspond à l'application des critères physiques, la fonction $algo()$ correspond à l'application des critères algorithmiques, et v

[2]Correspondant par exemple à l'invocation des commandes *hcitool scan* et *hcitool info bdaddr* pour Bluetooth, et *iwlist scan iface* pour Wi-Fi (sur une plateforme Linux).

critère	action
rareté	$score(v)$ est diminué à chaque synchronisation réussie avec v.
fréquence	$score(v)$ est augmenté à chaque synchronisation réussie avec v.
marquage	$score(v)$ est augmenté si une condition c est vérifiée.
marquage	$score(v)$ est augmenté si une règle r_i a été appliquée.
⋮	⋮

FIG. 3.3 – Exemples de critères algorithmiques

Voisin	*Exemple de modification*
v1	$score \times 2$
v2	$score \div 2$
v3	$(score \times 3) + 20$
...	...

FIG. 3.4 – Impact des critères algorithmiques

correspond à chaque voisin détecté. L'ordre d'application des critères (critères physiques avant critères algorithmiques) reflète le sens naturel des informations manipulées, qui remontent ici du périphérique vers l'application. Enfin, selon les désirs du concepteur, on peut imaginer qu'en dessous d'un certain score les voisins soient tout simplement retirés du tableau. Nous appellerons ce score particulier *seuil*.

3.3.2 Ordonnancement des voisins

Le fonctionnement global de la procédure est donné par les figures 3.5 et 3.6. A intervalles réguliers, le périphérique scanne son entourage et renseigne, pour chaque voisin trouvé, des informations sur le lien de communication correspondant (3.6(a)). En fonction de ces informations, et éventuellement des précédentes si un historique est disponible, un score est calculé pour chaque voisin (3.6(c)). Si aucun critère physique n'est spécifié, un score par défaut doit être fourni. Le score de chaque voisin est ensuite (potentiellement) modifié par l'application des critères algorithmiques (3.6(d) et 3.6(e)). Enfin, les voisins sont triés par ordre de score décroissant, et ceux dont le score est inférieur au seuil sont éliminés (3.6(f)).

A l'issue de ces traitements, le nœud sous-jacent possède une liste ordonnée de voisins avec lesquels il souhaite travailler en priorité. Cette liste peut alors être utilisée pour guider le choix d'un voisin lors de l'étape 2 de la procédure de synchronisation à la demande (algorithme 12 page 38).

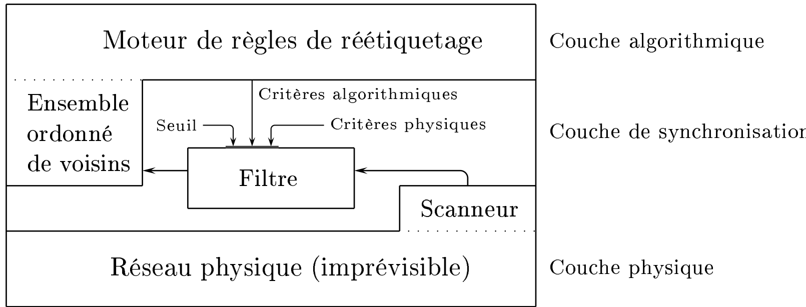


FIG. 3.5 – Ordonnancement et sélection des voisins - architecture

3.3.3 Adaptation au contexte

La sélection de liens et l'utilisation de critères permet également d'adapter un même algorithme à différents contextes de mobilité sans le modifier. En effet, sans toucher aux règles de réétiquetage, mais simplement en jouant sur les critères appliqués, un même algorithme peut être déployé de différentes manières. Nous illustrons ceci en reprenant l'exemple de l'algorithme de propagation de document (algorithme 3 page 29). Son comportement peut être adapté aux contextes suivants :

Faible mobilité et messages conséquents. Si les nœuds du réseau se déplacent lentement et que les données à échanger sont assez importantes (*p.ex.* une personne souhaite propager un fichier multimédia dans une salle), alors la priorité peut être donnée aux liens de communication les plus stables, en favorisant $min(\Delta\omega(signal))$.

Forte mobilité et petits messages. Si les nœuds du réseau sont très dynamiques et que les messages à transmettre sont courts, *p.ex.* des automobiles sur une route qui propagent une information d'avertissement relatif à un accident, il peut être intéressant d'avertir en priorité les véhicules dont la direction est différente, les autres véhicules pouvant être avertis peu après. Cela peut être fait en donnant la priorité aux liens de communication dont la puissance du signal perçue varie le plus vite ($max(\Delta\omega(signal))$).

Couverture spatiale maximale. Si l'objectif de la propagation est de couvrir rapidement une vaste étendue, alors il sera bon d'informer en priorité les nœuds les plus éloignés, ce qui pourrait revenir à privilégier $min(\omega(signal))$.

Nécessité d'un bon débit. Si l'application nécessite une bonne bande passante (*p.ex.* pour du *streaming*), le critère choisi pourra être le signal le plus fort ($max(\omega(signal))$) ou, selon la technologie utilisée, le signal étant dans la plage de fréquence la moins encombrée ($min(bruit(signal))$), *etc.*

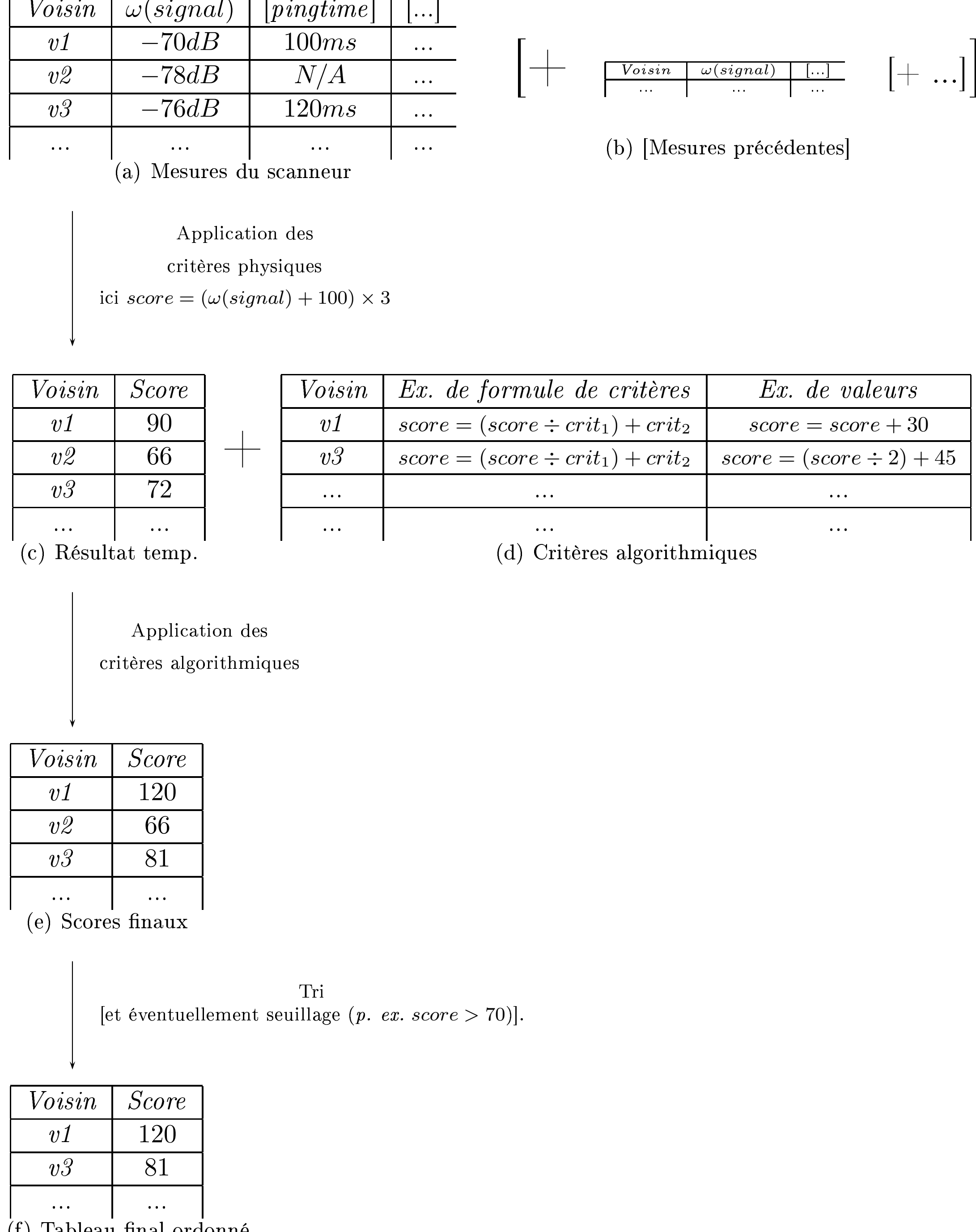


FIG. 3.6 – Ordonnancement et sélection des voisins - exemple de fonctionnement détaillé

3.4 Critères étendus pour les réseaux sans fil

La synchronisation à la demande avec critères présentée précédemment propose d'utiliser l'état physique ou algorithmique des liens de communication incidents à un nœud pour l'aider à choisir les voisins avec lesquels il souhaite travailler en priorité. Nous proposons ici d'utiliser les capacités naturelles de *broadcast* des technologies sans fil pour aller plus loin, en permettant aux nœuds de considérer l'état même de leurs voisins comme un critère possible de synchronisation.

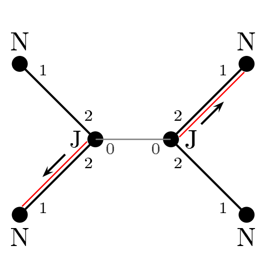
Principe de fonctionnement : A intervalles réguliers, les nœuds diffusent des informations sur leur état. Comme pour les critères sur les liens de communication, ces informations peuvent être de nature physique (niveau de charge de batterie, capacité cpu, *etc.*) ou algorithmique (étiquette du sommet, étiquettes des brins d'arêtes, *etc.*). Chaque nœud peut mettre à jour ces informations sur ses voisins, en parallèle de l'étape 1 de la procédure (Algorithme 12 page 38), étape qui précède le choix d'un voisin. Ces données n'influant que sur le choix du voisin, une valeur périmée n'aura pas d'impact sur la cohérence des calculs. Enfin, ces informations peuvent être traitées comme n'importe quel critère algorithmique, *via* le tableau de la figure 3.6(a).

3.5 Exemple d'utilisation

L'algorithme 1 page 25 permet de maintenir une forêt d'arbres couvrants sur un réseau dynamique. Pour rappel, cet algorithme repose sur la circulation de jetons qui représentent chacun, à tout moment, la racine d'un arbre différent. Lorsque deux sommets racines (sommets ayant un jeton) appliquent la règle r_1, cela a pour effet de fusionner les deux arbres de manière instantanée. Le reste du temps, les jetons circulent au sein de leurs arbres par application de la règle r_2.

La procédure de synchronisation utilisée dans le cadre de cet algorithme peut avoir un impact déterminant sur les opérations de circulation et de fusion des jetons. Par exemple, si le mode de synchronisation ne privilégie aucun lien de communication particulier, alors il se peut que deux voisins dans le réseau aient chacun un jeton, et qu'ils ne fusionnent pas, comme dans l'exemple de la figure 3.7 page suivante. Il est également possible que certaines synchronisations n'aboutissent à aucun réétiquetage, si les deux voisins synchronisés ne vérifient les préconditions d'aucune règle. C'est le cas par exemple lorsque l'arête sous-jacente à la synchronisation est étiquetée 0 de part et d'autre et que les sommets correspondants ne sont pas tous deux racines.

Le mode de synchronisation à la demande avec critères, si l'on suppose comme en 3.4 que les critères peuvent s'étendre aux étiquettes des sommets, permet de résoudre ces problèmes assez simplement. Ce mode permet par exemple de favoriser : 1) les fusions par rapport aux circulations, 2) les circulations entre elles, en utilisant à chaque fois l'arête la plus anciennement parcourue par le jeton (idée inspirée de [GL93]). La mise en œuvre de ces priorités peut se faire de la manière suivante : à chaque application de la règle r_2 (règle de circulation), le sommet u qui reçoit le jeton marque localement le voisin v duquel il l'a reçu. Ce marquage a pour valeur la date locale à laquelle la règle a été appliquée. Les voisins desquels u n'a jamais reçu le jeton peuvent être marqués avec la date de début



Dans l'exemple ci-contre, deux sommets racines (étiquette J) se trouvaient à portée l'un de l'autre. Ils étaient donc dans la capacité de fusionner leur deux arbres via l'application de la règle r_1. Cependant, la procédure de synchronisation mise en œuvre ici les a chacun, par hasard, synchronisés avec un autre de leurs voisins (en rouge sur le dessin). C'est donc la règle de circulation du jeton, r_2, qui sera finalement appliquée de part et d'autre, et la possibilité de fusion sera temporairement supprimée.

FIG. 3.7 – Exemple de fusion manquée

d'exécution de l'algorithme. La formule suivante peut alors être utilisée comme formule de critère :

$$\begin{aligned}&\text{Pour tout voisin } v,\\ score(v) = &(isLabelled_J(me) \text{ and } isLabelled_J(v))?2:0\\ &+(isLabelled_2(edgeTo(v)) \text{ and } (isLabelled_J(me)))?1:0\\ &+\frac{1}{date_{marquage}(v)}\end{aligned}$$

Une telle combinaison aura pour conséquence de favoriser les demandes de synchronisations entre racines (première opérande). Si une racine n'a aucune autre racine à portée, alors elle privilégiera les synchronisations avec ses enfants dans l'arbre (deuxième opérande), et privilégiera pour ces dernières les voisins ayant la date de marquage la plus ancienne (troisième opérande). Enfin, si l'on instaure un seuil minimum de 1 pour les scores ainsi obtenus, cela supprime les demandes de synchronisation entre sommets dont les états n'auraient mené à aucun réétiquetage. Il est important de rappeler ici qu'à aucun moment l'algorithme n'a lui-même été modifié.

L'optimisation de la circulation du jeton nous a été proposée par Guinand et Pigné de l'université du Havre, qui mènent actuellement des travaux sur son impact en terme de performances. Il semblerait, selon leurs premiers résultats de simulation, que le mode de circulation du jeton divise à lui seul par un facteur deux le temps moyen nécessaire à une fusion dans la forêt.

Chapitre 4

Analyse d'algorithmes

Sommaire

4.1 Les Graphes Évolutifs . . . 46
4.1.1 Graphe sous-jacent . . . 46
4.1.2 Gestion du temps . . . 46
4.1.3 Délais . . . 47
4.1.4 Définition complète d'un graphe évolutif . . . 47
4.2 Autres définitions sur les graphes évolutifs . . . 48
4.2.1 Représentation étiquetée . . . 48
4.2.2 Sous-graphe évolutif . . . 49
4.2.3 Trajets . . . 49
4.2.4 Destinations communes . . . 51
4.2.5 Coupe temporelle . . . 51
4.3 Outils pour l'analyse d'algorithmes distribués . . . 52
4.3.1 Graphes évolutifs et graphes étiquetés . . . 52
4.3.2 Réétiquetages . . . 52
4.3.3 Nature des objectifs d'un algorithme . . . 53
4.3.4 Conditions sur le réseau . . . 54
4.3.5 Conditions nécessaires sur la dynamique du réseau . . . 55
4.3.6 Conditions suffisantes sur la dynamique du réseau . . . 55
4.4 Analyse de quelques algorithmes simples . . . 55
4.4.1 Propagation d'information . . . 56
4.4.2 Comptage - version 1 . . . 58
4.4.3 Comptage - version 2 . . . 59
4.5 Vers une classification des réseaux dynamiques . . . 60
4.5.1 Classes résultant de l'analyse . . . 60
4.5.2 Autres classes . . . 63
4.5.3 Relations entre classes . . . 64
4.5.4 Intérêt d'une classification . . . 66

Dans ce chapitre, nous proposons un nouveau cadre d'analyse pour les algorithmes distribués en environnement dynamique. Ce cadre a pour but d'aider à dégager, pour un algorithme donné, les propriétés sur la dynamique du réseau qui sont nécessaires ou suffisantes à la réalisation de ses objectifs. Les outils théoriques que nous utilisons ici sont

les réétiquetages de graphes (pour les algorithmes), les graphes évolutifs (pour les réseaux dynamiques) et la logique du premier ordre [CPW] (pour exprimer les propriétés nécessaires et suffisantes). Contrairement au chapitre 2, qui ne faisait aucune hypothèse sur la représentation du temps, nous passons ici à une représentation discrète des dates et événements liés au réseau. La première section de ce chapitre présente les principales définitions relatives au modèle des graphes évolutifs. La seconde le complète avec plusieurs définitions reposant sur ces graphes, la plupart d'entre elles faisant partie de notre contribution. Ces définitions sont nécessaires à la compréhension des sections suivantes. La section 3 propose un ensemble de concepts pour l'analyse d'algorithmes, concepts que nous utilisons pour l'étude de quelques algorithmes dans la section 4. Enfin, la dernière section propose une ébauche de classification de graphes dynamiques, portant en grande partie sur les classes que l'analyse de la section 4 a fait émerger. Cette dernière section, ainsi que le chapitre, se termine par une discussion sur les usages liés à une telle classification dans le contexte de l'algorithmique distribuée et des réseaux dynamiques.

4.1 Les Graphes Évolutifs

Les graphes évolutifs sont des objets combinatoires permettant de représenter des interactions qui évoluent de manière discrète dans le temps. Ces objets ont été utilisés comme modèle de réseau dynamique dans plusieurs travaux récents, comme [Fer04] et [Jar05] dont nos premières définitions sont issues. La présente section s'attache ainsi à définir de manière formelle ce qu'est un graphe évolutif, à travers les différents éléments qui le composent. Ces éléments sont 1) un graphe sous-jacent G qui représente l'union de tous les sommets et de toutes les arêtes existant au cours de la vie du réseau 2) une suite de dates $\mathcal{S}_{\mathbb{T}}$, dont chaque élément désigne le moment où un (ou plusieurs) changements topologiques se produisent dans le réseau 3) une suite de graphes $\mathcal{S}_G$ dont chaque élément, sous-graphe du graphe sous-jacent, représente l'état du réseau à une date donnée de $\mathcal{S}_{\mathbb{T}}$. Une fonction de délai de traversée d'arête ζ peut également être ajoutée à la définition d'un graphe évolutif mais celle-ci n'étant pas nécessaire à nos travaux, nous l'abstrairons après l'avoir malgré tout expliquée.

4.1.1 Graphe sous-jacent

Définition 4.1 ***(Graphe sous-jacent** $G = (V, E)$**)*** *On appelle graphe sous-jacent d'un graphe évolutif le graphe* $G = (V, E)$*, représentant tous les sommets et arêtes ayant existé, à un moment ou à un autre, durant la période de temps couverte par le graphe évolutif considéré.*

Nous utilisons ici des arêtes et non des arcs car nous ne considérons que des liens de communication bidirectionnels. Les définitions données dans cette section restent cependant correctes pour des graphes orientés.

4.1.2 Gestion du temps

Définition 4.2 ***(Domaine temporel** $\mathbb{T}$**)*** *Le domaine temporel* $\mathbb{T}$ *est l'ensemble de toutes les dates, durées, délais, etc. possibles. Selon les cas considérés, cet ensemble pourra être identifié à* $\overline{\mathbb{N}}$ *ou* $\overline{\mathbb{R}_+}$*, c.-à.-d. aux ensembles* $\mathbb{N}$ *ou* $\mathbb{R}_+$ *munis de* $-\infty$ *et* $+\infty$*.*

Dans le modèle des graphes évolutifs, chaque série d'événements topologiques survenant dans le réseau (apparition ou disparition d'un ou plusieurs nœuds, et/ou d'un ou plusieurs liens de communication) est associée à une date dans $\mathbb{T}$, date à laquelle il se produit. L'ensemble $\mathcal{S}_{\mathbb{T}} = t_0, t_1, ..., t_{last}$ représente la suite des dates correspondant à ces événements. A chacune de ces dates (excepté t_{last}) est associé un graphe G_i, sous-graphe du graphe sous-jacent, représentant l'état du réseau durant l'intervalle $[t_i, t_{i+1}[$.

Définition 4.3 ***(Séquence temporelle $\mathcal{S}_{\mathbb{T}}$)*** *Nous appelons Séquence temporelle $\mathcal{S}_{\mathbb{T}} = t_0, t_1, ..., t_{last}$ une suite ordonnée de dates représentant la séquence de vie d'un réseau. Dans cette séquence, t_0 est la date correspondant à l'état initial du réseau, t_{last} est la date correspondant à son état final, et chaque élément de $t_1, ..., t_{last-1}$ correspond à une date où un ou plusieurs événements topologiques s'y sont produits.*

Définition 4.4 ***(Séquence de sous-graphes $\mathcal{S}_G$)*** *Soit $G_i = (V_i, E_i)$ le graphe représentant les nœuds disponibles (V_i) et les liens disponibles (E_i) durant l'intervalle de temps $[t_i, t_{i+1}[$. La suite ordonnée de tous ces sous-graphes de G, appelée séquence de sous-graphes, est notée $\mathcal{S}_G = G_0, G_1, ..., G_{last-1}$*

4.1.3 Délais

Définition 4.5 ***(Délai de traversée d'une arête ζ)*** *On appelle $\zeta : E \times \mathbb{T} \rightarrow \mathbb{T}$ la fonction indiquant le temps de traversée, à un instant donné, de chaque arête de G.*

La représentation du délai de traversée des arêtes varie selon le modèle de graphe évolutif considéré. Dans [Jar05], chaque arête se voit associer un temps de traversée propre et constant dans le temps ; la propagation des messages dans les arêtes suit donc un modèle *FIFO*. Il est possible, sans modifier le modèle, de représenter des délais variant de manière discrète dans le temps pour chaque lien de communication, en dupliquant l'arête correspondante autant de fois qu'il y a de variations de son délai de traversée et en affectant aux différentes copies obtenues les dates de présence et les délais de traversée correspondants. Lorsque le délai est constant dans le temps, l'ensemble de départ donné définition 4.5 ($E \times \mathbb{T}$) se réduit à E.

4.1.4 Définition complète d'un graphe évolutif

Le système complet peut être donné par les définitions 4.6 ou 4.7, selon que l'on considère le délai de traversée des arêtes ou non. Dans la suite du présent document nous n'utiliserons que des graphes évolutifs répondant à la définition 4.7.

Définition 4.6 ***(Graphe évolutif avec délais)*** *Soit un graphe $G = (V, E)$, $\mathcal{S}_G$ une suite ordonnée de sous-graphes de G, $\mathcal{S}_{\mathbb{T}}$ une suite croissante de $\mathbb{T}$ contenant un élément de plus que $\mathcal{S}_G$ (élément correspondant à la dernière date du réseau). Soit ζ une fonction de $E \times \mathbb{T}$ vers $\mathbb{T}$. Le système $\mathcal{G} = (G, \mathcal{S}_G, \mathcal{S}_{\mathbb{T}}, \zeta)$ est appelé graphe évolutif avec délais.*

Définition 4.7 ***(Graphe évolutif sans délai)*** *Soit un graphe $G = (V, E)$, $\mathcal{S}_G$ une suite ordonnée de sous-graphes de G, $\mathcal{S}_{\mathbb{T}}$ une suite croissante de $\mathbb{T}$ contenant un élément de plus que $\mathcal{S}_G$. Le système $\mathcal{G} = (G, \mathcal{S}_G, \mathcal{S}_{\mathbb{T}})$ est appelé graphe évolutif sans délais, ou plus simplement graphe évolutif.*

4.2 Autres définitions sur les graphes évolutifs

Dans cette section, nous proposons une série de définitions et concepts portant sur les graphes évolutifs. Nous introduisons dans un premier temps la définition de la représentation étiquetée d'un graphe évolutif, souvent utilisée dans la littérature mais n'ayant pas, à notre connaissance, été déjà formalisée. Nous proposons également les notions de sous-graphes et de coupes temporelles d'un graphe évolutif (définitions 4.8 et 4.14). Enfin, nous réutilisons la définition existante des *trajets* dans les graphes évolutifs, pour définir les notions de *trajets strict* (définition 4.10), *trajet canoniques* (définition 4.11) et *destinations communes* (définition 4.13).

4.2.1 Représentation étiquetée

Un graphe évolutif $\mathcal{G} = (G, \mathcal{S}_G, \mathcal{S}_\mathbb{T})$ peut être représenté par un graphe étiqueté $(G, \lambda_\mathcal{T})$, avec G le graphe sous-jacent de $\mathcal{G}$, et $\lambda_\mathcal{T} : E(G) \to \mathcal{S}_\mathbb{T}^n$ la fonction qui associe à chaque arête de G une étiquette indiquant les intervalles de temps pour lesquels elle est disponible, c'est-à-dire tel que $\forall e \in E(G), \forall t_i \in \mathcal{S}_\mathbb{T}, t_i \in \lambda_\mathcal{T}(e) \Leftrightarrow e \in E(G_i)$. Le graphe évolutif ainsi obtenu sera noté $\mathcal{G} = (V_\mathcal{G}, E_\mathcal{G}, \lambda_\mathcal{T})$.

La figure 4.1 représente l'état d'un réseau aux quatre dates $t_0 = 1$, $t_1 = 2$, $t_2 = 3$ et $t_3 = 4$. La figure 4.2 page ci-contre donne la représentation étiquetée du graphe évolutif correspondant.

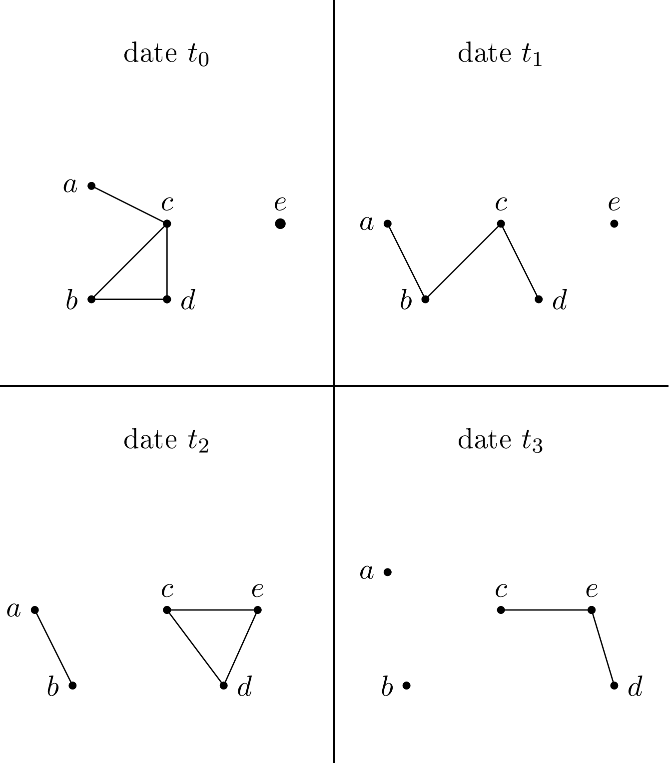


FIG. 4.1 – Suite de sous-graphes $\mathcal{S}_G$ de $\mathcal{G}$, indiquant la topologie du réseau à quatre dates différentes

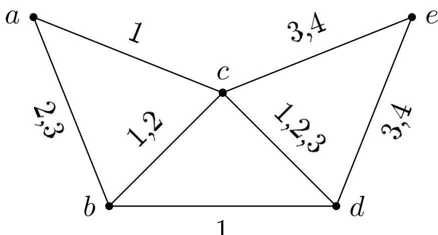


Fig. 4.2 – Représentation étiquetée du graphe évolutif de la figure 4.1

Dans la suite du chapitre nous utiliserons tantôt l'une ou l'autre des deux définitions d'un graphe évolutif ($\mathcal{G} = (V_\mathcal{G}, E_\mathcal{G}, \lambda_\mathcal{T})$ ou $\mathcal{G} = (G, \mathcal{S}_G, \mathcal{S}_\mathbb{T})$).

4.2.2 Sous-graphe évolutif

Soient L_1 et L_2 deux valeurs d'étiquette (indiquant chacune une suite de dates, par exemple "*1 3 4*"). Si toutes les dates de L_2 sont aussi présentes dans L_1, alors l'étiquette L_2 est dite *incluse* dans L_1. Cette inclusion est notée $L_2 \subseteq L_1$.

Soient $\mathcal{G}_1 = (V_{\mathcal{G}_1}, E_{\mathcal{G}_1}, \lambda_{\mathcal{T}_1})$ et $\mathcal{G}_2 = (V_{\mathcal{G}_2}, E_{\mathcal{G}_2}, \lambda_{\mathcal{T}_2})$ deux graphes évolutifs tels que $E_{\mathcal{G}_2} \subseteq E_{\mathcal{G}_1}$. Nous notons $\lambda_{\mathcal{T}_2} \subseteq \lambda_{\mathcal{T}_1}$ le fait que $\forall e \in E_{\mathcal{G}_2}, \lambda_{\mathcal{T}_2}(e) \subseteq \lambda_{\mathcal{T}_1}(e)$

Définition 4.8 ***(Sous-graphe évolutif)*** *Soient $\mathcal{G} = (V_\mathcal{G}, E_\mathcal{G}, \lambda_\mathcal{T})$ et $\mathcal{G}' = (V_{\mathcal{G}'}, E_{\mathcal{G}'}, \lambda'_\mathcal{T})$ deux graphes évolutifs. $\mathcal{G}'$ est appelé sous-graphe évolutif de $\mathcal{G}$ si et seulement si $V_{\mathcal{G}'} \subseteq V_\mathcal{G}$, $E_{\mathcal{G}'} \subseteq E_\mathcal{G}$ et $\lambda'_\mathcal{T} \subseteq \lambda_\mathcal{T}$. Nous notons cette relation $\mathcal{G}' \subseteq \mathcal{G}$.*

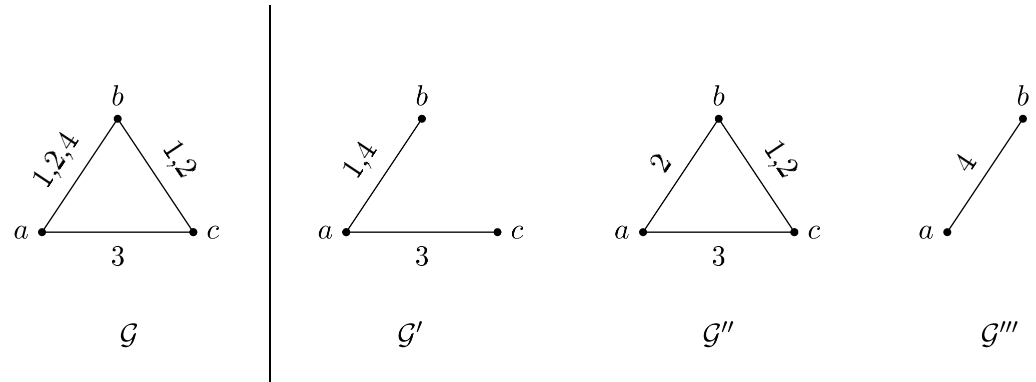


Fig. 4.3 – Un graphe évolutif $\mathcal{G}$ et trois de ses sous-graphes évolutifs $\mathcal{G}'$, $\mathcal{G}''$ et $\mathcal{G}'''$

4.2.3 Trajets

Un trajet peut être vu comme un chemin *dans le temps* d'un sommet vers un autre. Ci-dessous nous donnons la définition formelle d'un trajet, issue de [Fer04], et rajoutons

les définitions des trajets *stricts* et des trajets *canoniques* utilisés de manière récurrente dans la suite du chapitre.

Soit $\mathcal{S}_\mathbb{T}$ une suite de dates dans $\mathbb{T}$ et σ une date de $\mathbb{T}$ qui n'est pas forcément dans $\mathcal{S}_\mathbb{T}$. Nous notons $predecesseur(\sigma)$ la date $max(t_i \in \mathcal{S}_\mathbb{T} \mid t_i \leq \sigma)$ et $successeur(\sigma)$ la date $min(t_i \in \mathcal{S}_\mathbb{T} \mid t_i > \sigma)$.

Définition 4.9 ***(Trajet dans un graphe évolutif)*** *Soit $\mathcal{G} = (G, \mathcal{S}_G, \mathcal{S}_\mathbb{T})$ un graphe évolutif, soit P un chemin $P = e_1, e_2, ..., e_k \mid e_i \in E_G$. Soit P_σ une suite croissante de dates $P_\sigma = \sigma_1, \sigma_2, ..., \sigma_k \mid \sigma_i \in \mathbb{T}$. Le couple $\mathcal{J} = (P, P_\sigma)$ est un trajet dans $\mathcal{G}$ si et seulement si $\forall \sigma_i \in P_\sigma, e_i \in G_{predecesseur(\sigma_i)}$. Un trajet d'un sommet a vers un sommet b sera noté $\mathcal{J}_{(a,b)}$.*

Il est important de remarquer que la notion de trajet n'est pas symétrique : l'existence d'un trajet d'un sommet a vers un sommet b n'implique pas l'existence d'un trajet de b vers a.

Définition 4.10 ***(Trajet strict dans un graphe évolutif)*** *Un* ***trajet strict*** *est un trajet dont chaque date de traversée d'arête appartient à un intervalle de temps différent (dans $\mathcal{S}_\mathbb{T}$). Plus formellement, $\forall \sigma_1, \sigma_2 \in P_\sigma, predecesseur(\sigma_1) \neq predecesseur(\sigma_2)$. Un trajet strict d'un sommet a vers un sommet b sera noté $\mathcal{J}_{strict(a,b)}$.*

Définition 4.11 ***(Trajet canonique dans un graphe évolutif)*** *Nous appelons* ***trajet canonique*** *la suite des triplets $(source, destination, date)$, indiquant chacun que l'arête $(source, destination)$ sera traversée à une date σ telle que $predecesseur(\sigma) = date$ (voir figure 4.4). Cette notion permet de regrouper en une seule entité tous les trajets ayant, pour chaque arête traversée, le même intervalle de temps concerné. Un trajet canonique sera dit* ***strict*** *si les intervalles de traversée des arêtes sont strictement croissants.*

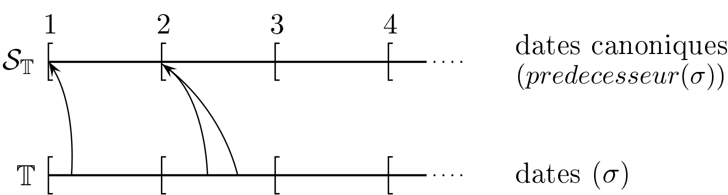


FIG. 4.4 – Dates dans les trajets canoniques

Dans ce document et à partir de cet endroit précis, nous ne manipulerons que des trajets canoniques, sauf mention contraire, que nous appellerons donc simplement trajets. La représentation des trajets sous forme de triplets dont les dates appartiennent toutes à $\mathcal{S}_\mathbb{T}$ permet d'assimiler chaque trajet dans $\mathcal{G}$ à un sous-graphe évolutif de $\mathcal{G}$, et donc de noter $\mathcal{J} \subseteq \mathcal{G}$.

Enfin, le nombre de triplets d'un trajet sera appelé sa *longueur*, et sera noté $|\mathcal{J}|$. Nous considérons également que pour tout sommet x de $V_\mathcal{G}$, $\mathcal{J}_{(x,x)} = \emptyset$ est un trajet strict de taille nulle.

Voici quelques exemples de trajets possibles dans le graphe évolutif de la figure 4.2 page précédente :

- $\mathcal{J}_{(a,d)} = \{(a,c,1),(c,d,1)\}$ est un trajet non strict de taille 2.
- $\mathcal{J}_{(a,e)} = \{(a,c,1),(c,b,1),(b,d,1),(d,c,3),(c,e,3)\}$ est un trajet non strict de taille 5.
- $\mathcal{J}_{(b,e)} = \{(b,c,1),(c,d,2),(d,e,3)\}$ est un trajet strict de taille 3, et sera noté $\mathcal{J}_{strict(b,e)}$.

4.2.4 Destinations communes

Définition 4.12 ***(Destinations d'un sommet)*** *Soit $\mathcal{G}$ un graphe évolutif. Soient deux sommets $u, v \in V_{\mathcal{G}}$, le sommet v fait partie des* ***destinations*** *du sommet u si et seulement si il existe un trajet de u vers v. Nous notons $v \in \mathcal{D}est(\mathcal{G}, u)$, ou plus simplement $v \in \mathcal{D}est(u)$ lorsque la désignation de $\mathcal{G}$ est contextuellement implicite.*

Observation 3 *Du fait de l'existence de trajets de taille nulle, chaque sommet fait partie de son propre ensemble de destinations.*

Définition 4.13 ***(Destinations communes à plusieurs sommets)*** *Soit $\mathcal{G}$ un graphe évolutif. Soient n sommets $u_1, u_2, ..., u_n \in V_{\mathcal{G}}$ et $v \in V_{\mathcal{G}}$. Le sommet v fait partie des* ***destinations communes*** *à $u_1, u_2, ...$ et u_n, et on note $v \in \mathcal{D}est(\mathcal{G}, u_1, u_2, ..., u_n)$ ou simplement $v \in \mathcal{D}est(u_1, u_2, ..., u_n)$, si et seulement si $\forall i \in 1..n, \exists \mathcal{J}_{(u_i,v)}$. Si un sommet v est une destination commune pour tous les sommets du graphe, nous notons $v \in \mathcal{D}est(\mathcal{G})$.*

La figure 4.5 donne quelques exemples de destinations communes dans un graphe évolutif.

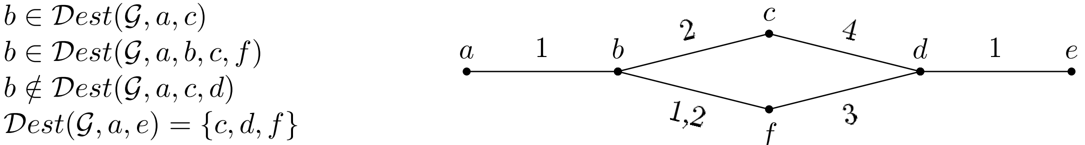


Fig. 4.5 – Destinations dans un graphe évolutif $\mathcal{G}$

4.2.5 Coupe temporelle

Définition 4.14 ***(Coupe temporelle)*** *La coupe temporelle d'un graphe évolutif $\mathcal{G} = (G, \mathcal{S}_G, \mathcal{S}_{\mathbb{T}})$, de la date t_1 à la date t_2, notée $\mathcal{G}_{[t_1,t_2[}$ est le graphe évolutif $\mathcal{G}' = (G', \mathcal{S}'_{G'}, \mathcal{S}'_{\mathbb{T}}) \subseteq \mathcal{G}$ tel que les trois propriétés suivantes sont satisfaites :*

1. $\forall t \in \mathcal{S}'_{\mathbb{T}}, t_1 \leq t < t_2$
2. $\forall t \in [t_1, t_2[, t \in \mathcal{S}_{\mathbb{T}} \implies t \in \mathcal{S}'_{\mathbb{T}}$
3. $\forall t \in \mathcal{S}'_{\mathbb{T}}, G'_t = G_t$.

Nous utiliserons également les notations suivantes :
- $\mathcal{G}_{[t_1,+\infty[}$ pour désigner $\mathcal{G}_{[t_1,max(\mathcal{S}_{\mathbb{T}})[}$
- $\mathcal{G}_{]-\infty,t_2[}$ pour désigner $\mathcal{G}_{[min(\mathcal{S}_{\mathbb{T}}),t_2[}$
- $\mathcal{G}_{]t_1,t_2[}$ pour désigner $\mathcal{G}_{[successeur(t_1),t_2[}$
- $\mathcal{G}_{[t_1,t_2]}$ pour désigner $\mathcal{G}_{[t_1,successeur(t_2)[}$

La figure 4.6 page suivante donne quelques exemples de coupes temporelles d'un graphe évolutif.

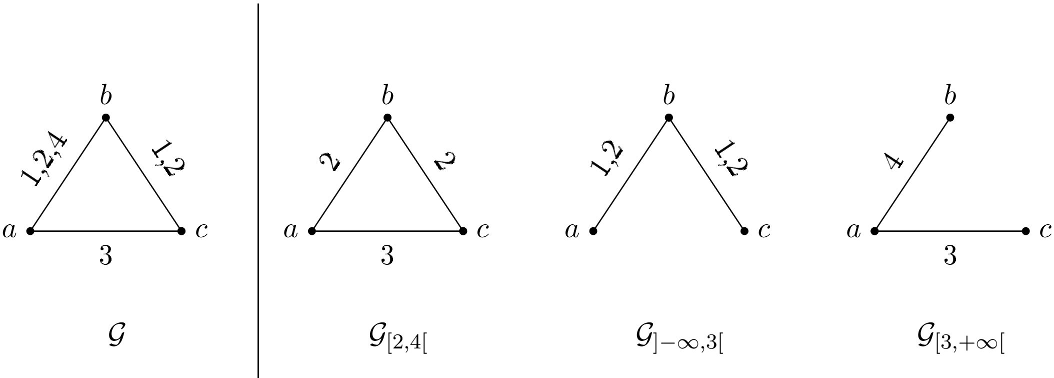


FIG. 4.6 – Un graphe évolutif $\mathcal{G}$ et trois de ses coupes temporelles

4.3 Outils pour l'analyse d'algorithmes distribués

Dans cette section, nous présentons les notions qui seront utilisées par la suite pour caractériser les conditions nécessaires ou suffisantes, relativement à la dynamique d'un réseau, pour garantir le succès ou l'échec d'un algorithme. Les notions introduites dans les deux premières parties de cette section sont illustrées par un schéma récapitulatif figure 4.7 page suivante, schéma auquel nous invitons le lecteur à se référer chaque fois que nécessaire.

4.3.1 Graphes évolutifs et graphes étiquetés

Pour rappel, nous représentons l'état du système à un moment donné par un graphe dont les sommets et les brins d'arêtes sont étiquetés. Ce graphe étiqueté est noté (G_λ), ou plus simplement $\mathbf{G}$ en l'absence d'ambiguïtés sur la désignation de l'étiquetage.

Définition 4.15 *Soit $\mathcal{G} = (G, \mathcal{S}_G, \mathcal{S}_\mathbb{T})$ un graphe évolutif, dont les éléments G_t de la suite $\mathcal{S}_G$ désignent chacun le graphe représentant le réseau à la date t. On note $\boldsymbol{G_t}$ le graphe G_t muni de son étiquetage à la date $t + \epsilon$, date succédant aux événements topologiques de la date t. En cas de suppression d'arête lors des événements topologiques de la date t, ce graphe a les brins correspondants marqués à* off *(comme défini à la section 2.3.3). De même, en cas d'ajout d'arête, les brins correspondants ont été étiquetés avec les valeurs par défaut spécifiées par l'algorithme.*

Définition 4.16 *Étant donné une date t de $\mathcal{S}_\mathbb{T}$, on appelle **graphe étiqueté précédant les événements de** t, noté $\boldsymbol{G_{t[}}$, le graphe étiqueté qui représente l'état global du système au moment où les événements de la date t vont se produire.*

Définition 4.17 *Étant donné un graphe évolutif $\mathcal{G} = (G, \mathcal{S}_G, \mathcal{S}_\mathbb{T})$, on définit un ensemble de fonctions, appelées fonctions d'événements qui, pour tout t de $\mathcal{S}_\mathbb{T}$, associent $\boldsymbol{G_t}$ à $\boldsymbol{G_{t[}}$. Nous utilisons la notation suivante :*

$$Ev_t(\boldsymbol{G_{t[}}) \;=\; \boldsymbol{G_t}$$

4.3.2 Réétiquetages

Pour rappel, nous notons $G_\lambda \; \mathcal{R} \; \mathcal{H}_\eta$ la relation de réécriture qui, au graphe étiqueté G_λ, associe le graphe étiqueté H_η. Nous rappelons également que les relations de réétiquetage

sont des relations de réécriture dont les graphes de départ et d'arrivée sont isomorphes. Les étapes de calcul pouvant être vues comme des instances de telles relations, nous les noterons $\mathcal{R}$ dans la suite du chapitre.

Définition 4.18 *Soit $\mathcal{R}_{\mathcal{A}}$ une étape de réétiquetage de l'algorithme $\mathcal{A}$. On appelle séquence de réétiquetage (de l'algorithme $\mathcal{A}$) de la date t à la date $t+1$, notée $\mathcal{R}_{\mathcal{A}_{[t,t+1[}}$, l'ensemble des réétiquetages ayant lieu entre les dates t et $t+1$ (temps pendant lequel le graphe est considéré comme statique). Cette séquence peut être vue comme une fonction sur le graphe étiqueté :*

$$\mathcal{R}_{\mathcal{A}_{[t,t+1[}}(\boldsymbol{G}_t) = \boldsymbol{G}_{t+1[}$$

Définition 4.19 *L'état du système de la date t_0 à la date t_{last} peut alors être décrit comme une succession d'événements topologiques et de réétiquetages qui débutent sur le graphe initial étiqueté $\boldsymbol{G}_{t_0}$:*

$$\mathcal{R}_{\mathcal{A}_{[t_{last-1},t_{last}[}} \circ Ev_{t_{last-1}} \circ \ \ldots \ \circ Ev_{t_i} \circ \mathcal{R}_{\mathcal{A}_{[t_{i-1},t_i[}} \circ \ \ldots \ \circ Ev_{t_1} \circ \mathcal{R}_{\mathcal{A}_{[t_0,t_1[}}(\boldsymbol{G}_{t_0})$$

La figure 4.7 récapitule les concepts et notations présentés jusqu'ici en ce qui concerne l'étiquetage et les réétiquetages des graphes évolutifs.

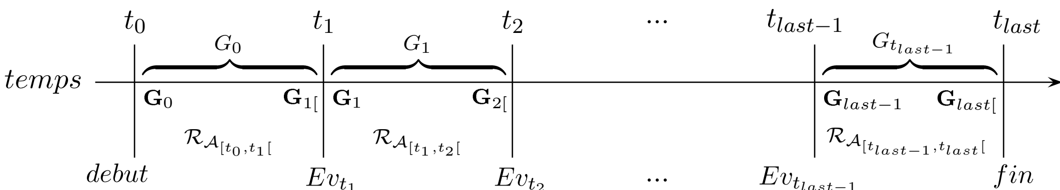


Fig. 4.7 – Schéma récapitulatif sur l'étiquetage des graphes évolutifs

4.3.3 Nature des objectifs d'un algorithme

On peut distinguer deux types fondamentaux d'objectifs qu'un algorithme peut chercher à remplir, selon qu'il s'agit de propriétés à maintenir tout au long de son exécution, ou de propriétés à atteindre à l'issue de son exécution.

Lorsque les objectifs de l'algorithme consistent à maintenir des propriétés au cours du temps, nous parlons d'**algorithme de maintien**. De tels objectifs peuvent être par exemple de mettre constamment à jour une structure couvrante, ou de maintenir une propriété de couverture données (arbre, k-connexité, *etc.*), ou de maintenir un leader pour chaque composante connexe du réseau, ou encore de maintenir sur chaque sommet une estimation du nombre de sommets de la composante connexe sous-jacente, *etc.*

Lorsque les objectifs de l'algorithme consistent à vérifier une propriété à l'issue de son exécution, nous parlons d'**algorithme à finalité**. De tels objectifs peuvent être par exemple de transmettre une information d'un sommet u vers un sommet v, vers tous les autres sommets, vers au moins 20% des sommets, de dénombrer les sommets, de mesurer les capacités de calcul cumulées sur l'ensemble du réseau, *etc.*

Définition 4.20 *Soit $\mathcal{G} = (G, \mathcal{S}_G, \mathcal{S}_\mathbb{T})$ le graphe évolutif représentant l'évolution du réseau durant l'exécution d'un algorithme de* **maintien** $\mathcal{A}$*. Soient t_0 et t_{last} la première et la dernière dates de $\mathcal{S}_\mathbb{T}$. Soit $\mathcal{P}$ la propriété qui doit être maintenue sur le graphe étiqueté. On dit que les objectifs de l'algorithme $\mathcal{A}$ sont atteints sur $\mathcal{G}$, et l'on note $\mathcal{O}_{\mathcal{A},\mathcal{G}}$, si et seulement si la propriété est rétablie entre chaque série d'événements topologiques :*

$$\mathcal{O}_{\mathcal{A},\mathcal{G}} \iff \forall t \in \mathcal{S}_\mathbb{T} \setminus \{t_0\}, \mathcal{P}(\boldsymbol{G}_{t[})$$

Ou, énoncé de manière différente :

$$\mathcal{O}_{\mathcal{A},\mathcal{G}} \iff \forall t \in \mathcal{S}_\mathbb{T} \setminus \{t_{last}\}, \mathcal{P}(\mathcal{R}_{\mathcal{A}_{[t,t+1[}} \circ Ev_t(\boldsymbol{G}_{t[}))$$

Définition 4.21 *Soit $\mathcal{G} = (G, \mathcal{S}_G, \mathcal{S}_\mathbb{T})$ le graphe évolutif représentant l'évolution du réseau durant l'exécution d'un algorithme* **à finalité** $\mathcal{A}$*. Soient t_0 et t_{last} la première et la dernière date de $\mathcal{S}_\mathbb{T}$. Soit $\mathcal{P}$ la propriété qui doit être atteinte durant l'exécution, on dit que les objectifs de l'algorithme $\mathcal{A}$ sont atteints si et seulement si $\mathcal{P}$ est vérifiée sur le dernier graphe étiqueté obtenu :*

$$\mathcal{O}_{\mathcal{A},\mathcal{G}} \iff \mathcal{P}(\boldsymbol{G}_{t_{last}[})$$

Les analyses présentées dans ce document se limiteront à quelques cas d'algorithmes simple à finalité, qui constituent un premier test pour le cadre d'analyse que nous proposons. L'étude d'algorithmes plus complexes, ainsi que d'algorithmes de maintien tels que l'algorithme de la forêt d'arbres couvrants (algorithme 1 page 25) est planifiée à terme.

4.3.4 Conditions sur le réseau

Dans un cadre statique, les conditions d'échec ou de succès d'un algorithme distribué dépendent essentiellement de propriétés sur le réseau sous-jacent. Il a été montré par exemple qu'on ne peut pas élire de manière déterministe un nœud parmi d'autres quand le réseau sous-jacent est anonyme (nœuds sans identifiant unique) et qu'il comporte certaines symétries (recouvrements) [Ang80]. Il est en revanche assez simple d'élire si l'une de ces propriétés n'est pas vérifiée, comme par exemple dans un arbre. Les caractérisations de ces cas de succès ou d'échec, sont aussi appelées résultats positifs ou négatifs. Le lecteur intéressé pourra notamment se référer aux travaux de Nancy Lynch, qui a effectué dans [Lyn89] une compilation de 100 résultats négatifs pour l'algorithmique distribuée.

Le cas des réseaux dynamiques est plus complexe car il nécessite la prise en compte, en plus des propriétés habituelles, de toutes les propriétés ayant trait à la dynamique du réseau. Pour certaines classes de réseaux, la connaissance des propriétés dynamiques de la topologie peuvent aider à prendre des décisions dans les algorithmes. C'est le cas par exemple des réseaux dont la dynamique est prévisible (*p.ex.* satellites LEO [WM97, FGP02]) ou de ceux dont la dynamique est contrôlée (*p.ex.* robots mobiles [SMR06]). Dans ce document, nous nous intéressons aux réseaux dont la dynamique n'est pas connue à l'avance ou, plus précisément, nous étudions des algorithmes qui n'ont aucune connaissance relative à cette l'évolution.

Il est néanmoins intéressant, une fois un algorithme conçu, d'étudier les conditions de mobilité sous lesquelles il atteindra ou non des objectifs donnés. C'est de ce type d'étude que traitent les paragraphes suivants.

4.3.5 Conditions nécessaires sur la dynamique du réseau

Définition 4.22 *Étant donnés un algorithme $\mathcal{A}$ et une propriété $\mathcal{P}$ définissant ses objectifs, on appelle* ***condition nécessaire*** *sur la dynamique du réseau, notée $\mathcal{C_N}$, toute condition portant sur un graphe évolutif, telle que :*

$$\forall \mathcal{G}, \neg \mathcal{C_N}(\mathcal{G}) \implies \neg \mathcal{O}_{\mathcal{A},\mathcal{G}}$$

Autrement dit, si la condition $\mathcal{C_N}$ n'est pas vérifiée sur le graphe évolutif représentant le réseau, aucune exécution ne permettra à l'algorithme $\mathcal{A}$ d'atteindre ses objectifs sur ce réseau. La mise en évidence d'une condition nécessaire constitue donc un résultat négatif.

4.3.6 Conditions suffisantes sur la dynamique du réseau

Définition 4.23 *Étant donnés un algorithme $\mathcal{A}$ et une propriété $\mathcal{P}$ définissant ses objectifs, on appelle* ***condition suffisante*** *sur la dynamique du réseau, notée $\mathcal{C_S}$, toute condition portant sur un graphe évolutif, telle que :*

$$\forall \mathcal{G}, \mathcal{C_S}(\mathcal{G}) \implies \mathcal{O}_{\mathcal{A},\mathcal{G}}$$

Autrement dit, si la condition $\mathcal{C_S}$ est vérifiée sur le graphe évolutif représentant le réseau, alors l'objectif de l'algorithme $\mathcal{A}$ sera atteint. La mise en évidence d'une condition suffisante constitue un résultat positif.

Le cas des conditions suffisantes est plus compliqué car il nécessite de fortes hypothèses sur le déroulement des synchronisations sous-jacentes aux calculs. En effet, aucun mode de synchronisation étudié, qu'il soit de type rendez-vous (section 3.1.3), ou à la demande (section 3.2), ne permet de garantir qu'une arête donnée, quelle que soit sa durée de présence, sera utilisée au bout d'un temps fini. En ce sens, aucune condition sur la dynamique du graphe ne peut réellement être qualifiée de suffisante. Il est donc important de préciser qu'étant donné un graphe évolutif $\mathcal{G} = (G, \mathcal{S}_G, \mathcal{S}_\mathbb{T})$, s'il s'agit de caractériser des conditions suffisantes, nous ferons toujours l'hypothèse suivante :

Hypothèse de progression 1 *Pour toute date t de $\mathcal{S}_\mathbb{T} \setminus \{t_{last}\}$, la période $[t, t+1[$ est « suffisamment longue » pour qu'une règle de réétiquetage au moins puisse être appliquée sur chaque arête présente, si ses préconditions sont déjà rassemblées au début de la période.*

Remarque : Intuitivement, le réalisme de cette hypothèse dépend du rapport entre les degrés des sommets du graphe aux instants considérés et la durée des périodes correspondantes. Aussi, pour diminuer l'impact de cette hypothèse, il peut être envisagé de fusionner des périodes dans le graphe évolutif et, pour chaque fusion, de ne conserver que les arêtes qui sont présentes sur l'ensemble des périodes fusionnées. Cette opération peut être répétée jusqu'à obtenir des probabilités de synchronisations réussies jugées convenables. En somme, cette transformation revient à ne conserver que les arêtes dont la durée de présence rend l'hypothèse 1 réaliste.

4.4 Analyse de quelques algorithmes simples

Dans cette section, nous analysons quelques algorithmes simples et caractérisons les cas où leur exécution réussit avec certitude, et les cas où leur exécution échoue avec certitude par le biais de propriétés nécessaires ou suffisantes sur les graphes évolutifs.

4.4.1 Propagation d'information

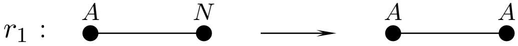
L'algorithme 2, déjà évoqué au second chapitre, a pour but de diffuser une information dans le réseau, de proche en proche, à partir d'un sommet émetteur.

L'algorithme

Pour rappel, les sommets sont étiquetés A s'ils possèdent l'information, N s'ils ne la possèdent pas. Initialement, tous les sommets sont étiquetés N, sauf un, l'émetteur, étiqueté A. Lorsqu'un sommet étiqueté A se retrouve synchronisé avec un sommet étiqueté N, il lui transmet l'information (application de la règle r_1). Le sommet informé peut alors communiquer à son tour l'information à ses voisins.

Rappel de l'algorithme 2 Propagation d'une information, version basique

Etats initiaux : un sommet distingué étiqueté A, les autres sommets étiquetés N

r_1 : 

Objectif 1

Nous aimerions caractériser ici les graphes évolutifs pour lesquels il existe au moins un sommet pouvant transmettre, de proche en proche, son information à tous les autres sommets. Avec cet objectif, l'algorithme 2 est un algorithme à finalité, dont la propriété à atteindre est la suivante :

$$\mathcal{P}_2(\mathbf{G}) \Longleftrightarrow \forall v \in V_{\mathcal{G}}, \lambda(v) = A$$

Comme défini dans la section précédente, l'objectif d'un algorithme à finalité est atteint dans un graphe $\mathcal{G} = (G, \mathcal{S}_G, \mathcal{S}_{\mathbb{T}})$ si et seulement si la propriété $\mathcal{P}_2$ est vérifiée à la fin de l'exécution sur le graphe étiqueté sous-jacent :

$$\mathcal{O}_{\mathcal{A}_2,\mathcal{G}} \Longleftrightarrow \mathcal{P}_2(\mathbf{G}_{t_{last}})$$

Soit $\mathcal{C}_1$ la condition telle qu'il existe au moins un sommet pouvant joindre tous les autres par un trajet dans le graphe évolutif :

$$\mathcal{C}_1 = \exists u \in V_{\mathcal{G}} \mid \forall v \in V_{\mathcal{G}} \setminus \{u\}, \exists \mathcal{J}_{(u,v)}$$

Proposition 4.1 *La condition $\mathcal{C}_1$ est nécessaire. Autrement dit s'il n'existe aucun sommet pouvant joindre tous les autres par un trajet, alors il ne sera pas possible d'informer tous les sommets, quel que soit l'émetteur choisi.*

Propriété intermédiaire 4.1.1 *Tout sommet informé, s'il n'est pas lui-même l'émetteur initial, a reçu l'information par un trajet. Énoncé formellement :*
$\forall v \in V_{\mathcal{G}} \mid \lambda_{t_0}(v) = N, \forall t \in \mathbb{T}_{]t_0,t_{last}]}$, // *les dates manipulées ici n'appartiennent pas forcément à* $\mathcal{S}_{\mathbb{T}}$
$\lambda_t(v) = A \implies \exists u_1 \in V_{\mathcal{G}} \mid \lambda_{t' \leq t}(u_1) = A$ *et* $\mathcal{J}_{(u_1,v)}$ *existe dans* $\mathcal{G}_{[predecesseur(t'),successeur(t)[}$

Preuve 4.1.1
(1) Si à un moment donné un sommet non émetteur a l'information, alors il l'a reçue à une date donnée, date à laquelle une arête existait entre lui et celui qui lui a transmis, ce dernier ayant déjà l'information.
$\forall v \in V_{\mathcal{G}} \mid \lambda_{t_0}(v) = N, \forall t \in \mathbb{T}_{]t_0,t_{last}]}, \lambda_t(v) = A \implies \exists t' \in \mathbb{T}_{]t_0,t]}, \exists x \in V_{\mathcal{G}} \mid \mathcal{R}1_{t'}(x,v)$
or $\mathcal{R}1_{t'}(x,v) \implies (x,v) \in E(G_{predecesseur(t')})$ *et* $\lambda_{t'}(x) = A$.
(2) Par répétition, il existe donc une suite de dates correspondant à une suite de réétiquetages sur des arêtes présentes à ces dates, depuis un sommet ayant l'information. Ce qui implique par définition l'existence, pour chaque sommet ayant reçu l'information, d'un trajet depuis un sommet l'ayant déjà :
$\lambda_t(v) = A \implies \exists d_1, ..., d_n \in \mathbb{T}_{]t_0,t]}, \exists u_1, ..., u_n \in V_{\mathcal{G}} \mid \forall i \in 1..n\text{-}1, (u_i, u_{i+1}) \in E(G_{predecesseur(d_i)})$ *et* $\lambda_{d_1}(u_1) = A$
d'où $\lambda_t(v) = A \implies \exists u_1 \in V_{\mathcal{G}} \mid \lambda_{t' \leq t}(u_1) = A$ *et* $\mathcal{J}_{(u_1,v)}$ *existe dans* $\mathcal{G}_{[predecesseur(t'),successeur(t)[}$ □

Preuve 4.1
(1) D'après la propriété 4.1.1, tout sommet ayant transmis l'information par un trajet $\mathcal{J}$, s'il n'est pas l'émetteur initial (i.e. le seul nœud ayant l'information au début de l'algorithme), l'a lui même reçue par un trajet. Par récurrence, il est donc évident qu'il existe une suite de trajets temporellement entraînables (et donc un trajet) depuis l'émetteur initial vers tout sommet ayant reçu l'information :
$\forall v \in V_{\mathcal{G}}, \lambda_{t_{last}}(v) = A \implies \exists \mathcal{J}_{(emetteur,v)}$ *et, par conséquent,* $\neg\mathcal{J}_{(emetteur,v)} \implies \neg(\lambda_{t_{last}}(v) = A)$.
(2) $\neg\mathcal{C}_1(\mathcal{G}) \implies \forall u \in V_{\mathcal{G}}, \exists v \in V_{\mathcal{G}} \mid \nexists \mathcal{J}_{(u,v)}$
$\implies \exists v \in V_{\mathcal{G}} \mid \nexists \mathcal{J}_{(emetteur,v)}$
par (1), $\implies \neg(\lambda_{t_{last}}(v) = A)$
$\implies \neg\mathcal{P}_2(\boldsymbol{G}_{t_{last}})$
$\implies \neg\mathcal{O}_{\mathcal{A}_2,\mathcal{G}}$ □

Soit $\mathcal{C}_2$ la condition telle qu'il existe au moins un sommet pouvant joindre tous les autres par un trajet **strict** dans le graphe évolutif :

$$\mathcal{C}_2 = \exists u \in V_{\mathcal{G}} \mid \forall v \in V_{\mathcal{G}} \setminus \{u\}, \exists \mathcal{J}_{strict(u,v)}$$

Proposition 4.2 *La condition $\mathcal{C}_2$ est suffisante. Autrement dit, s'il existe un sommet pouvant joindre tous les autres par un trajet strict, et si ce sommet est l'émetteur, alors tous les sommets seront informés.*

Preuve 4.2 $\exists \mathcal{J}_{strict(u,v)} \implies \exists d_1, d_2, ..., d_n \in \mathcal{S}_{\mathbb{T}}, \exists u_1, u_2, ..., u_n \in V_{\mathcal{G}} \mid \forall i \in 1..n\text{-}1, (u_i, u_{i+1}) \in E(G_{d_i})$ *et* $d_{i+1} > d_i$, *et donc (hypothèse de progression $\mathcal{H}$ 1 page 55)* $\lambda_{d_i}(u_i) = A \implies \lambda_{d_{i+1}}(u_{i+1}) = A$
donc, par répétition, $\exists \mathcal{J}_{strict(u,v)} \implies (\lambda_{t_0}(u) = A \implies \lambda_{t_{last}}(v) = A)$
d'où $\mathcal{C}_2(\mathcal{G}) \implies \exists u \in V_{\mathcal{G}} \mid \forall v \in V_{\mathcal{G}} \setminus \{u\}, (\lambda_{t_0}(u) = A \implies \lambda_{t_{last}}(v) = A) \implies \mathcal{P}_2(\boldsymbol{G}_{t_{last}})$ □

Objectif 2

Le deuxième objectif est le même que le premier, à savoir que tous les sommets doivent être informés à l'issue de l'exécution, mais nous désirons ici que l'émetteur puisse être n'importe lequel des sommets du graphe. Il est donc nécessaire qu'il y ait un trajet de n'importe quel émetteur potentiel vers tous les sommets du graphe, et suffisant qu'il y ait un trajet strict de n'importe quel émetteur, vers tous les sommets du graphe :

La condition $\mathcal{C}_3 = \forall u \in V_{\mathcal{G}}, \forall v \in V_{\mathcal{G}} \setminus \{u\}, \exists \mathcal{J}_{(u,v)}$ est une condition nécessaire.
La condition $\mathcal{C}_4 = \forall u \in V_{\mathcal{G}}, \forall v \in V_{\mathcal{G}} \setminus \{u\}, \exists \mathcal{J}_{strict(u,v)}$ est une condition suffisante.

4.4.2 Comptage - version 1

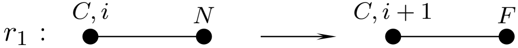
L'algorithme

L'algorithme 4, déjà évoqué au second chapitre, a pour but de compter le nombre maximal de participants dans un réseau (*i.e.* obtenir une borne inférieure sur le nombre de sommets du graphe).

Pour rappel, le fonctionnement de l'algorithme est le suivant : un sommet compteur (étiqueté C) se charge de compter les sommets qu'il rencontre. A chaque fois qu'il rencontre un sommet non compté (étiqueté N), il incrémente son nombre de sommets comptés (deuxième registre d'étiquette, nommé *counter*). Chaque sommet compté est alors marqué comme *compté* (étiquette F), pour ne pas être compté deux fois. Initialement, tous les sommets sont étiquetés N, sauf un, le compteur, étiqueté $C, 1$.

Rappel de l'algorithme 4 Comptage avec un sommet compteur distingué.

Etats initiaux : un sommet distingué étiqueté $(C, 1)$, les autres sommets étiquetés N

r_1 :


Objectif 1

On s'intéresse ici à caractériser les graphes évolutifs pour lesquels il existe au moins un sommet pouvant compter tous les autres. Comme montré section 2.4.3, l'objectif est atteint quand tous les sommets, le compteur excepté, sont étiquetés F :

$$\mathcal{P}_4(\mathbf{G}) \iff \forall v \in V_{\mathcal{G}} \setminus \{compteur\}, \lambda(v) = F$$

Soit $\mathcal{C}_5$ la condition telle qu'il existe au moins un sommet v ayant, au cours du temps, une arête commune avec chaque autre sommet, ce qui revient à dire que $\{v\}$ est un ensemble dominant dans le graphe sous-jacent G.

$$\mathcal{C}_5 = \exists u \in V_{\mathcal{G}} \mid \forall v \in V_{\mathcal{G}} \setminus \{u\}, (u, v) \in E_{\mathcal{G}}$$

Proposition 4.3 *$\mathcal{C}_5$ est une condition nécessaire. Autrement dit, s'il n'existe aucun sommet partageant au cours du temps une arête avec chaque autre sommet, alors les sommets ne pourront pas tous être comptés, et ce, quel que soit le sommet compteur.*

Preuve 4.3
$\neg\mathcal{C}_5(\mathcal{G}) \implies \forall u \in V_{\mathcal{G}}, \exists v \in V_{\mathcal{G}} \setminus \{u\} \mid (u, v) \notin E_{\mathcal{G}}$
$\implies \forall u \in V_{\mathcal{G}}, \exists v \in V_{\mathcal{G}} \setminus \{u\} \mid \nexists \mathcal{R}_1(u, v)$ // $\mathcal{R}_1(u, v)$ *désigne ici une application de la règle* r_1 *sur l'arête* (u, v)
$\implies \exists v \in V_{\mathcal{G}} \setminus \{u\} \mid \neg(\lambda_{t_{last}}(v) = F)$
$\implies \neg\mathcal{P}_4(\boldsymbol{G}_{t_{last}})$ □

Proposition 4.4 *$\mathcal{C}_5$ est une condition suffisante. Autrement dit, s'il existe un sommet partageant au cours du temps une arête avec chaque autre sommet, et si ce sommet est le compteur, alors tous les sommets seront comptés.*

Preuve 4.4
$\mathcal{C}_5 \implies \exists u \in V_{\mathcal{G}} \mid \forall v \in V_{\mathcal{G}} \setminus \{u\}, \exists t \in \mathcal{S}_{\mathbb{T}} \mid (u, v) \in E(G_t)$
et donc (hypothèse de progression $\mathcal{H}$ 1 page 55),
$\exists u \in V_{\mathcal{G}} \mid \forall v \in V_{\mathcal{G}} \setminus \{u\}, \exists t_1, t_2 \in \mathcal{S}_{\mathbb{T}} \mid (\lambda_{t_1}(u) = C \text{ et } \lambda_{t_1}(v) = N \implies \lambda_{t_2}(v) = F)$

$\implies \exists u \in V_{\mathcal{G}} \mid \forall v \in V_{\mathcal{G}}, (\lambda_{t_0}(u) = C \; et \; \lambda_{t_0}(v) = N \implies \lambda_{t_{last}}(v) = F)$
$\implies \mathcal{P}_4(\boldsymbol{G}_{t_{last}})$ □

Objectif 2

Le deuxième objectif est quasiment le même que le premier, à savoir que tous les sommets soient comptés par le compteur à l'issue de l'exécution, mais nous désirons ici que le compteur puisse être n'importe lequel des sommets du graphe. Les conditions qui devaient s'appliquer à un sommet au moins pour l'objectif 1 (existence, au cours du temps, d'une arête commune avec chaque autre sommet) doivent donc ici s'appliquer à tous les sommets (n'importe quel sommet doit avoir, au cours du temps, une arête commune avec chaque autre sommet) :

$\mathcal{C}_6 = \forall u \in V_{\mathcal{G}}, \forall v \in V_{\mathcal{G}} \setminus \{u\}, (u,v) \in E_{\mathcal{G}}$ est donc une condition nécessaire et suffisante pour l'objectif 2. Cette condition revient à exiger que le graphe G, graphe sous-jacent de $\mathcal{G}$, soit complet.

4.4.3 Comptage - version 2

Objectif de l'algorithme

L'algorithme 5, déjà évoqué au second chapitre, a pour fonction de compter le nombre maximal de participants du réseau. Contrairement à l'algorithme 4, il ne suppose pas l'existence d'un compteur distingué au début de l'algorithme. En effet, tous les sommets démarrent avec le même état, ce qui en fait un algorithme totalement décentralisé.

Pour rappel, le fonctionnement de l'algorithme est le suivant : chaque sommet possède deux registres d'étiquette. Le premier registre représente son état : *compteur*(C) ou *compté*(F). Le deuxième, si le sommet est compteur, représente le nombre de sommets qui ont été comptés par lui et par ceux qu'il a lui-même comptés (qu'on appelle *valeur de comptage*). Initialement tous les sommets sont étiquetés C et 1. A chaque fois que deux sommets compteurs appliquent la règle r_1, l'un des deux reste compteur, et l'autre devient compté. Le sommet compteur ajoute alors à sa valeur de comptage le nombre correspondant à la valeur de comptage de l'autre sommet.

Rappel de l'algorithme 5 Comptage avec multiples sommets compteurs qui fusionnent.

Etats initiaux : tous les sommets étiqueté $(C, 1)$

r_1 : (C,i) ●———● (C,j) ⟶ $(C,i+j)$ ●———● F

Objectif

On s'intéresse ici à trouver les propriétés nécessaires et suffisantes sur les graphes évolutifs pour qu'un sommet atteigne le décompte total du nombre de sommets. Comme montré section 2.4.4, l'objectif est atteint quand il n'y a plus qu'un seul sommet étiqueté C, et que tous les autres sont étiquetés F :

$$\mathcal{P}_5(\mathbf{G}) \iff \exists u \in V_{\mathcal{G}} \mid \lambda(u) = (C, i) \text{ et } \forall v \in V_{\mathcal{G}} \setminus \{u\}, \lambda(v) = F$$

Soit $\mathcal{C}_7$ la condition telle qu'il existe au moins un sommet pouvant être joint par tous les autres par un trajet.

$$\mathcal{C}_7 = \exists v \in V_{\mathcal{G}} \mid v \in \mathcal{D}est(\mathcal{G})$$

Proposition 4.5 *La condition $\mathcal{C}_7$ est nécessaire.*

Propriété intermédiaire 4.5.1 $\forall u \in V_{\mathcal{G}} \mid \lambda_{t_0}(u) = C, \exists u' \in \mathcal{D}est(u) \mid \lambda_{t_{last}}(u') = C$

Preuve 4.5.1
$\forall u \in V_{\mathcal{G}} \mid \lambda_{t_0}(u) = C, \nexists u' \in \mathcal{D}est(u) \mid \lambda_{t_{last}}(u') = C \implies \exists \mathcal{R}^*_{1(u,w)} \mid w \notin \mathcal{D}est(u)$ *et* $\lambda_{t_{last}}(w) = C$
($\mathcal{R}^*_{1(u,w)}$ désigne ici une suite de réétiquetages s'opérant sur une suite d'arêtes reliant u à w)
Or $\mathcal{R}^*_{1(u,w)} \implies \exists \mathcal{J}_{(u,w)} \implies w \in \mathcal{D}est(u)$
$\implies$ *contradiction*

Preuve 4.5
$\neg\mathcal{C}_7 \implies \forall v \in V_{\mathcal{G}}, \exists u \in V_{\mathcal{G}} \setminus \{v\} \mid v \notin \mathcal{D}est(u)$
Or, Lemme 4.5.1, $\forall u \in V_{\mathcal{G}} \mid \lambda_{t_0}(u) = C, \exists u' \in \mathcal{D}est(u) \mid \lambda_{t_{last}}(u') = C$
$\implies \forall v \in V_{\mathcal{G}}, \exists x \in V_{\mathcal{G}} \setminus \{v\} \mid \lambda_{t_{last}}(x) = C$ *(il y a donc au minimum 2 compteurs à la fin de l'exécution)*
$\implies \neg\mathcal{P}_5(\boldsymbol{G}_{t_{last}})$ □

Remarque : Nous étudions actuellement la définition d'une condition suffisante.

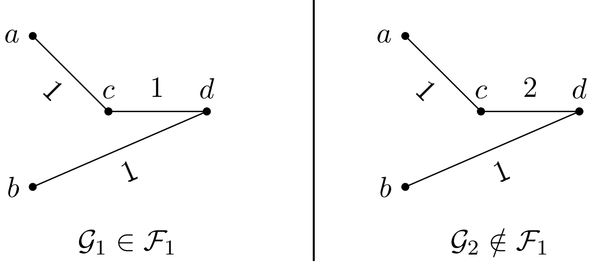
4.5 Vers une classification des réseaux dynamiques

Comme il a été vu dans la section précédente, les caractérisations des conditions de succès ou d'échec de divers algorithmes aboutissent à un ensemble de propriétés sur les graphes évolutifs. Chacune de ces propriétés $\mathcal{P}$ détermine, de par l'ensemble des graphes qui la vérifient, une classe particulière de graphes évolutifs $\mathcal{F}$, et donc de réseau dynamique. D'un point de vue plus formel, chaque propriété $\mathcal{P}$ définit une *classe d'équivalence* entre graphes évolutifs. Chaque classe de graphes évolutifs $\mathcal{F}$ peut ainsi être vue comme le *quotient* de tous les graphes évolutifs possibles par la classe d'équivalence que $\mathcal{P}$ définit. Dans cette section, nous rappelons dans un premier temps les différentes classes qui ont été obtenues par l'analyse de la section précédente, auxquelles nous ajoutons deux classes issues de la littérature. Une ébauche de classification est ensuite proposée sur la base de ces classes. Enfin, la section se termine par une discussion sur l'impact qu'une classification des graphes dynamiques peut avoir, entre autres, sur le domaine de l'algorithmique distribuée.

4.5.1 Classes résultant de l'analyse

Classe 1 *La classe $\mathcal{F}_1$ est caractérisée par la propriété $\mathcal{C}_1$ et un exemple de graphe est donné figure 4.8. Les graphes évolutifs qui ne sont pas dans cette classe ne permettent pas la diffusion d'une information depuis un sommet émetteur, quel qu'il soit, vers tous les autres sommets ($\mathcal{C}_1$ est une condition nécessaire pour l'objectif 1 de l'algorithme 2 page 27).*

$$\mathcal{C}_1 = \exists u \in V(\mathcal{G}) \mid \forall v \in V_{\mathcal{G}} \setminus \{u\}, \exists \mathcal{J}_{(u,v)}$$

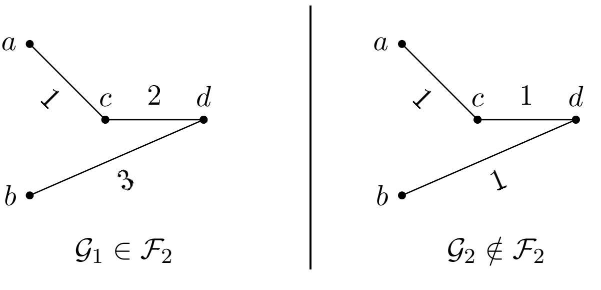


Il existe au moins un sommet pouvant joindre chaque autre sommet par un trajet.

FIG. 4.8 – Classe 1

Classe 2 *La classe $\mathcal{F}_2$ est caractérisée par la propriété $\mathcal{C}_2$ et un exemple de graphe est donné figure 4.9. Chacun de ces graphes évolutifs possède au moins un sommet pouvant, à coup sûr, informer tous les autres ($\mathcal{C}_2$ est une condition suffisante pour objectif 1 de l'algorithme 2 page 27).*

$$\mathcal{C}_2 = \exists u \in V(\mathcal{G}) \mid \forall v \in V_{\mathcal{G}} \setminus \{u\}, \exists \mathcal{J}_{strict(u,v)}$$

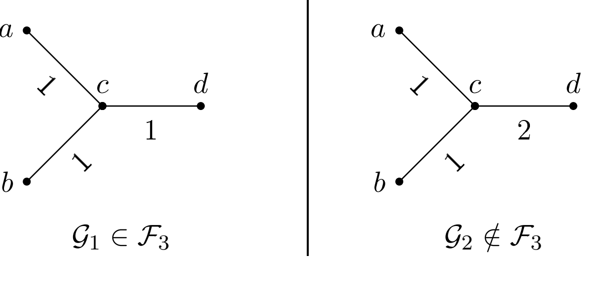


Il existe au moins un sommet pouvant joindre chaque autre sommet par un trajet strict.

FIG. 4.9 – Classe 2

Classe 3 *La classe $\mathcal{F}_3$ est caractérisée par la propriété $\mathcal{C}_3$ et un exemple de graphe est donné figure 4.10. Les graphes évolutifs n'appartenant pas à cette classe ne permettent pas à n'importe quel sommet de faire parvenir une information à tous les autres ($\mathcal{C}_3$ est une condition nécessaire pour l'objectif 2 de l'algorithme 2 page 27).*

$$\mathcal{C}_3 = \forall u, v \in V_{\mathcal{G}}, \exists \mathcal{J}_{(u,v)}$$



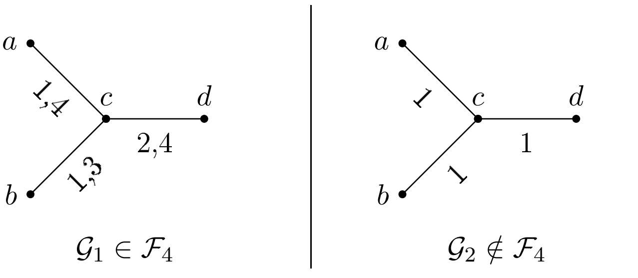
Il existe au moins un trajet de chaque sommet vers chaque autre sommet.

FIG. 4.10 – Classe 3

Classe 4 *La classe $\mathcal{F}_4$ est caractérisée par la propriété $\mathcal{C}_4$ et un exemple de graphe est donné figure 4.11. Dans les graphes évolutifs de cette classe, n'importe quel sommet peut*

faire parvenir une information à tous les autres ($\mathcal{C}_4$ est une condition suffisante pour l'objectif 2 de l'algorithme 2 page 27).

$$\mathcal{C}_4 = \forall u, v \in V_{\mathcal{G}}, \exists \mathcal{J}_{strict(u,v)}$$

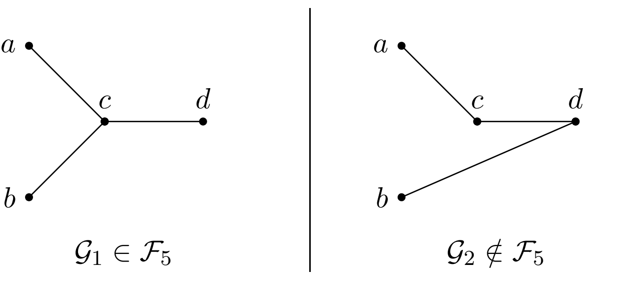


Il existe au moins un trajet strict de chaque sommet vers chaque autre sommet.

FIG. 4.11 – Classe 4

Classe 5 *La classe $\mathcal{F}_5$ est caractérisée par la propriété $\mathcal{C}_5$ et un exemple de graphe est donné figure 4.12. Dans les graphes évolutifs qui ne sont pas dans cette classe, aucun sommet ne peut compter tous les autres via l'algorithme 4 page 29. En revanche, dans les graphes évolutifs de cette classe, il existe un sommet pouvant compter tous les autres avec le même algorithme ($\mathcal{C}_5$ est une condition nécessaire <u>et</u> suffisante pour l'objectif 1 de l'algorithme 4).*

$$\mathcal{C}_5 = \exists u \in V_{\mathcal{G}} \mid \forall v \in V_{\mathcal{G}} \setminus \{u\}, (u, v) \in E_{\mathcal{G}}$$

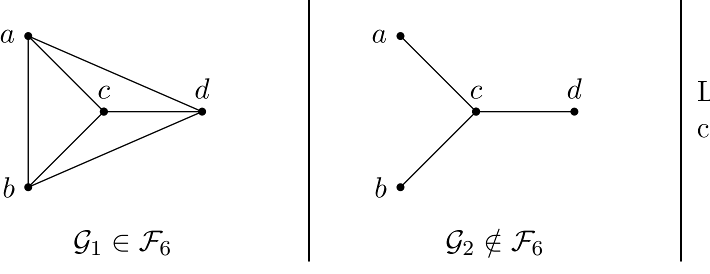


Il existe au moins un ensemble dominant de cardinalité 1 dans le graphe sous-jacent.

FIG. 4.12 – Classe 5

Classe 6 *La classe $\mathcal{F}_6$ est caractérisée par la propriété $\mathcal{C}_6$ et un exemple de graphe est donné figure 4.13. Les graphes évolutifs qui n'appartiennent pas à cette classe ne permettent pas à n'importe quel sommet de compter tous les autres via l'algorithme 4 page 29. En revanche, dans les graphes évolutifs de cette classe, n'importe quel sommet peut compter tous les autres par le même algorithme ($\mathcal{C}_6$ est une condition nécessaire <u>et</u> suffisante pour l'objectif 2 de l'algorithme 4).*

$$\mathcal{C}_6 = \forall u \in V_{\mathcal{G}}, \forall v \in V_{\mathcal{G}} \setminus \{u\}, (u, v) \in E_{\mathcal{G}}$$

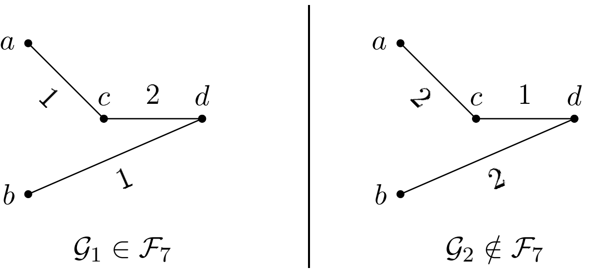


Le graphe sous-jacent est complet.

FIG. 4.13 – Classe 6

Classe 7 *La classe $\mathcal{F}_7$ est caractérisée par la propriété $\mathcal{C}_7$ et un exemple de graphe est donné figure 4.14. Dans les graphes évolutifs n'appartenant pas à cette classe, aucun sommet ne peut connaître le nombre total de participant via l'algorithme 5 page 30 ($\mathcal{C}_7$ est une condition nécessaire pour l'objectif de cet algorithme).*

$$\mathcal{C}_7 = \exists v \in V_{\mathcal{G}} \mid v \in \mathcal{D}est(\mathcal{G})$$



Il existe au moins un sommet qui est une destination commune à tous les sommets du graphe.

FIG. 4.14 – Classe 7

4.5.2 Autres classes

Classe 8

Graphe d'interaction complet et infini : à tout moment on a la garantie qu'il existera dans le futur une arête entre chaque paire de sommets. On notera que dans ce cas précis, la suite $\mathcal{S}_{\mathbb{T}}$ est infinie. Cette classe a été utilisée par Angluin et al. dans [AAD+06]. Toujours selon les mêmes auteurs, elle peut représenter par exemple un petit enclos de volailles dotées de capteurs de température où chaque volaille évolue en permanence au sein de l'enclos.

$$\mathcal{C}_8 = \forall u, v \in V_{\mathcal{G}}, \forall t \in \mathcal{S}_{\mathbb{T}}, (u, v) \in E_{\mathcal{G}_{[t,+\infty[}}$$

Classe 9

Graphe d'interaction connexe dans le temps, à l'infini : à tout moment, on a la garantie qu'il existera dans le futur un trajet entre chaque paire de sommets. Cette classe est fréquemment utilisée dans le domaine du routage, lorsque l'on suppose qu'aucun participant ne quitte le réseau. Pour cette classe, la suite $\mathcal{S}_{\mathbb{T}}$ est également infinie.

$$\mathcal{C}_9 = \forall u, v \in V_{\mathcal{G}}, \forall t \in \mathcal{S}_{\mathbb{T}}, \exists \mathcal{J}_{(u,v)} \subseteq \mathcal{G}_{[t,+\infty[}$$

4.5.3 Relations entre classes

Pour rappel, nous définissons chaque classe de graphe dynamique par une propriété sur les graphes évolutifs de sorte que tout graphe évolutif vérifiant cette propriété appartient à la classe. Étant donné une classe $\mathcal{F}$, on désigne par $\mathcal{P}_{\mathcal{F}}$ sa propriété caractérisante.

Définition 4.24 ***(Relation d'inclusion entre classes)*** *Nous définissons une relation binaire d'inclusion entre classes de graphes évolutifs. Cette relation est définie à la manière usuelle de la théorie des ensembles : une classe $\mathcal{F}'$ est dite* ***incluse*** *(ou* ***contenue****) dans la classe $\mathcal{F}$, et l'on note $\mathcal{F}' \subset \mathcal{F}$, si et seulement si tout élément de $\mathcal{F}'$ est dans $\mathcal{F}$. Dans notre cas, cela revient également à dire que $\mathcal{P}_{\mathcal{F}'} \implies \mathcal{P}_{\mathcal{F}}$. Par définition, cette relation est transitive (si $\mathcal{F}'' \subset \mathcal{F}'$ et $\mathcal{F}' \subset \mathcal{F}$, alors $\mathcal{F}'' \subset \mathcal{F}$).*

La suite de cette section propose quelques relations d'inclusion entre les différentes classes rencontrées sections 4.5.1 et 4.5.2.

Proposition 4.6 $\mathcal{F}_2 \subset \mathcal{F}_1$ *(Il existe un trajet strict d'au moins un sommet vers tous les autres $\implies$ il existe un trajet d'au moins un sommet vers tous les autres)*

Preuve 4.6
Pour l'objectif 1 de l'algorithme 2, $\mathcal{C}_2$ est une condition suffisante et $\mathcal{C}_1$ est une condition nécessaire, tout graphe vérifiant $\mathcal{C}_2$ vérifie donc aussi $\mathcal{C}_1$, indépendamment de l'algorithme 2.
donc $\forall \mathcal{G}, \mathcal{C}_2(\mathcal{G}) \implies \mathcal{C}_1(\mathcal{G})$
Il était également possible de prouver cette inclusion en utilisant le fait qu'un trajet strict est un trajet. □

Proposition 4.7 $\mathcal{F}_4 \subset \mathcal{F}_3$ *(Il existe un trajet strict de n'importe quel sommet vers tous les autres $\implies$ il existe un trajet de n'importe quel sommet vers tous les autres)*

Preuve 4.7
Pour l'objectif 2 de l'algorithme 2, $\mathcal{C}_4$ est une condition suffisante et $\mathcal{C}_3$ est une condition nécessaire, tout graphe vérifiant $\mathcal{C}_4$ vérifie donc aussi $\mathcal{C}_3$, indépendamment de l'algorithme 2.
donc $\forall \mathcal{G}, \mathcal{C}_4(\mathcal{G}) \implies \mathcal{C}_3(\mathcal{G})$
Il était également possible de prouver cette inclusion en utilisant le fait qu'un trajet strict est un trajet. □

Proposition 4.8 $\mathcal{F}_3 \subset \mathcal{F}_1$ *(N'importe quel sommet peut joindre tous les autres par un trajet $\implies$ Il existe un sommet pouvant joindre chaque autre sommet par un trajet)*

Preuve 4.8
$\mathcal{C}_1 = \exists u \in V_{\mathcal{G}} \mid \forall v \in V_{\mathcal{G}} \setminus \{u\}, \exists \mathcal{J}_{(u,v)}$
$\mathcal{C}_3 = \forall u \in V_{\mathcal{G}}, \forall v \in V_{\mathcal{G}} \setminus \{u\}, \exists \mathcal{J}_{(u,v)}$
$\forall \mathcal{G}, \mathcal{C}_3(\mathcal{G}) \implies \mathcal{C}_1(\mathcal{G})$ *de manière évidente.* □

Proposition 4.9 $\mathcal{F}_4 \subset \mathcal{F}_2$ *(N'importe quel sommet peut joindre tous les autres par un trajet strict $\implies$ Il existe un sommet pouvant joindre chaque autre sommet par un trajet strict)*

Preuve 4.9
$\mathcal{C}_2 = \exists u \in V_{\mathcal{G}} \mid \forall v \in V_{\mathcal{G}} \setminus \{u\}, \exists \mathcal{J}_{strict(u,v)}$
$\mathcal{C}_4 = \forall u \in V_{\mathcal{G}}, \forall v \in V_{\mathcal{G}} \setminus \{u\}, \exists \mathcal{J}_{strict(u,v)}$
$\forall \mathcal{G}, \mathcal{C}_4(\mathcal{G}) \implies \mathcal{C}_2(\mathcal{G})$ *de manière évidente.* □

Proposition 4.10 $\mathcal{F}_6 \subset \mathcal{F}_5$ *(Il existe au cours du temps une arête entre chaque paire de sommets $\implies$ Il existe un sommet ayant, au cours du temps, une arête commune avec chaque autre sommet)*

Preuve 4.10
$\mathcal{C}_5 = \exists u \in V_{\mathcal{G}} \mid \forall v \in V_{\mathcal{G}} \setminus \{u\}, (u,v) \in E_{\mathcal{G}}$
$\mathcal{C}_6 = \forall u \in V_{\mathcal{G}}, \forall v \in V_{\mathcal{G}} \setminus \{u\}, (u,v) \in E_{\mathcal{G}}$
$\forall \mathcal{G}, \mathcal{C}_6(\mathcal{G}) \implies \mathcal{C}_5(\mathcal{G})$ *de manière évidente.* □

Proposition 4.11 $\mathcal{F}_5 \subset \mathcal{F}_2$ *(Il existe un sommet ayant, au cours du temps, une arête commune avec chaque autre sommet* $\implies$ *Il existe un sommet pouvant joindre chaque autre sommet par un trajet strict)*

Preuve 4.11
$\mathcal{C}_5 = \exists u \in V_{\mathcal{G}} \mid \forall v \in V_{\mathcal{G}} \setminus \{u\}, (u,v) \in E_{\mathcal{G}}$
$\mathcal{C}_2 = \exists u \in V_{\mathcal{G}} \mid \forall v \in V_{\mathcal{G}} \setminus \{u\}, \exists \mathcal{J}_{strict(u,v)}$
$(u,v) \in E_{\mathcal{G}} \implies \exists t \in \mathcal{S}_{\mathbb{T}} \mid (u,v) \in E(G_t) \implies \exists \mathcal{J}_{strict(u,v)} = \{(u,v,t)\}$
(une arête est un trajet strict)
donc $\forall \mathcal{G}, \mathcal{C}_5(\mathcal{G}) \implies \mathcal{C}_2(\mathcal{G})$ □

Proposition 4.12 $\mathcal{F}_6 \subset \mathcal{F}_4$ *(Tout sommet a, au cours du temps, une arête commune avec chaque autre sommet* $\implies$ *Chaque sommet peut joindre tous les autres par un trajet strict)*

Preuve 4.12
$\mathcal{C}_6 = \forall u, v \in V_{\mathcal{G}}, (u,v) \in E_{\mathcal{G}}$
$\mathcal{C}_4 = \forall u, v \in V_{\mathcal{G}}, \exists \mathcal{J}_{strict(u,v)}$
$(u,v) \in E_{\mathcal{G}} \implies \exists t \in \mathcal{S}_{\mathbb{T}} \mid (u,v) \in E(G_t) \implies \exists \mathcal{J}_{strict(u,v)} = \{(u,v,t)\}$
(une arête est un trajet strict)
donc $\forall \mathcal{G}, \mathcal{C}_6(\mathcal{G}) \implies \mathcal{C}_4(\mathcal{G})$ □

Proposition 4.13 $\mathcal{F}_3 \subset \mathcal{F}_7$ *(Il existe un trajet entre chaque paire de sommet* $\implies$ *Chaque sommet peut être joint par tous les autres par un trajet)*

Preuve 4.13
$\mathcal{C}_3 = \forall u, v \in V_{\mathcal{G}}, \exists \mathcal{J}_{(u,v)}$
$\mathcal{C}_7 = \exists v \in V_{\mathcal{G}} \mid v \in \mathcal{D}est(\mathcal{G})$
$\mathcal{C}_3 \implies \exists v \in V_{\mathcal{G}} \mid \forall u \in V_{\mathcal{G}}, \exists \mathcal{J}_{(u,v)} \implies \mathcal{C}_7$
(Si tout le monde peut joindre tout le monde, alors il existe un sommet pouvant être joint par tout le monde)
donc $\forall \mathcal{G}, \mathcal{C}_3(\mathcal{G}) \implies \mathcal{C}_7(\mathcal{G})$ □

Proposition 4.14 $\mathcal{F}_8 \subset \mathcal{F}_9$ *(A tout moment, il existe une arête, dans le futur, entre chaque paire de sommet* $\implies$ *A tout moment, il existe un trajet, dans le futur, entre chaque paire de sommet)*

Preuve 4.14
$\mathcal{C}_8 = \forall u, v \in V_{\mathcal{G}}, \forall t \in \mathcal{S}_{\mathbb{T}}, (u,v) \in E_{\mathcal{G}_{[t,+\infty[}}$
$\mathcal{C}_9 = \forall u, v \in V_{\mathcal{G}}, \forall t \in \mathcal{S}_{\mathbb{T}}, \exists \mathcal{J}_{(u,v)} \subset \mathcal{G}_{[t,+\infty[}$
$\forall u, v \in V_{\mathcal{G}}, \forall t \in \mathcal{S}_{\mathbb{T}}, (u,v) \in E_{\mathcal{G}_{[t,+\infty[}} \implies \exists t' \in (\mathcal{S}_{\mathbb{T}} \cup [t,+\infty[) \mid \mathcal{J}_{(u,v)} = \{(u,v,t')\} \subset \mathcal{G}_{[t,+\infty[}$
$\implies \exists \mathcal{J}_{(u,v)} \subset \mathcal{G}_{[t,+\infty[}$
(une arête est un trajet)
donc $\forall \mathcal{G}, \mathcal{C}_8(\mathcal{G}) \implies \mathcal{C}_9(\mathcal{G})$ *de manière évidente.* □

Proposition 4.15 $\mathcal{F}_8 \subset \mathcal{F}_6$ *(A tout moment, il existe une arête, dans le futur, entre chaque paire de sommet* $\implies$ *Il existe une arête, au cours du temps, entre chaque paire de sommet)*

Preuve 4.15
$\mathcal{C}_8 = \forall u, v \in V_{\mathcal{G}}, \forall t \in \mathcal{S}_{\mathbb{T}}, (u,v) \in E_{\mathcal{G}_{[t,+\infty[}}$
$\mathcal{C}_6 = \forall u, v \in V_{\mathcal{G}}, (u,v) \in E_{\mathcal{G}}$
$\mathcal{C}_8 \implies \forall u, v \in V_{\mathcal{G}}, (u,v) \in E_{\mathcal{G}_{[t_0,+\infty[}} \implies \mathcal{C}_6$
donc $\forall \mathcal{G}, \mathcal{C}_8(\mathcal{G}) \implies \mathcal{C}_6(\mathcal{G})$. □

Proposition 4.16 $\mathcal{F}_9 \subset \mathcal{F}_4$ *(A tout moment, il existe un trajet, dans le futur, entre chaque paire de sommet $\implies$ Il existe un trajet strict entre chaque paire de sommet)*

Preuve 4.16
$\mathcal{C}_9 = \forall u, v \in V_{\mathcal{G}}, \forall t \in \mathcal{S}_{\mathbb{T}}, \exists \mathcal{J}_{(u,v)} \subset \mathcal{G}_{[t,+\infty[}$
$\mathcal{C}_4 = \forall u, v \in V_{\mathcal{G}}, \exists \mathcal{J}_{strict(u,v)}$
$\mathcal{C}_9 \implies \forall u_1, u_n \in V_{\mathcal{G}} \mid \exists \mathcal{J}_{(u_1,u_n)} = \{(u_1, u_2, t_1), (u_2, u_3, t_2), ...(u_{n-1}, u_n, t_{n-1})\} \subset \mathcal{G}$
Or, $\mathcal{C}_9 \implies \forall t \in \mathcal{S}_{\mathbb{T}}, \exists t' \in \mathcal{S}_{\mathbb{T}} \mid t' > t$ *et,* $\forall i \in 1..n\text{-}1, \exists \mathcal{J}_{(u_i,u_n)} \subset \mathcal{G}_{[t',+\infty[}$
$\implies \exists t_1, t_2, ..., t_{n-1} \in \mathcal{S}_{\mathbb{T}} \mid \forall i \in 1..n\text{-}1, t_{i+1} > t_i$ *et* $(u_i, u_{i+1}) \in E_{\mathcal{G}_{[t_i,+\infty[}}$
$\implies \exists \mathcal{J}_{strict(u_1,u_n)} \subset \mathcal{G}$
donc $\forall \mathcal{G}, \mathcal{C}_3(\mathcal{G}) \implies \mathcal{C}_7(\mathcal{G})$ □

La figure 4.15 récapitule toutes les inclusions proposées. Chaque flèche de cette figure peut être lue "est incluse dans". En regardant cette figure on peut constater que l'ensemble des classes étudiées est connexe par l'inclusion. Il s'agit là d'une propriétée non prémédité.

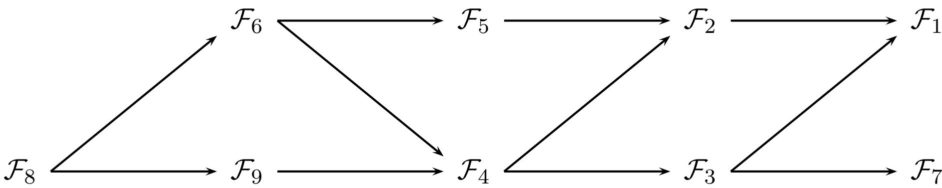


FIG. 4.15 – Relations d'inclusion entre quelques classes de graphes évolutifs

4.5.4 Intérêt d'une classification

Dans la section 4.5.3, nous avons proposé une ébauche de classification portant sur une dizaine de classes de graphes dynamiques. Cette ébauche de classification, basée sur les seules notions d'arêtes et de trajets dans le temps, ne représente très certainement qu'une infime partie de ce qui peut être fait. Les domaines d'application des graphes dynamiques sont nombreux, et peuvent intéresser des chercheurs de disciplines très variées, allant de l'algorithmique distribuée à la neurologie, ou de l'étude des colonies de fourmis à celle des réseaux de télécommunication. Une classification plus poussée, ainsi que l'adoption d'un outil théorique commun, tel que le graphe évolutif, aiderait probablement les chercheurs de ces disciplines à se positionner les uns par rapport aux autres, et pourquoi pas à partager des résultats. Nous pensons que ce parallèle entre différents travaux représente aujourd'hui un axe de recherche intéressant, mais vaste. Nous nous limiterons donc dans ce document à discuter des avantages qu'une classification peut avoir dans le contexte de nos travaux, c'est-à-dire pour l'algorithmique distribuée dans les réseaux dynamiques.

Un algorithme distribué est généralement conçu pour un type de réseau cible donné. On peut citer par exemple les réseaux mobiles *ad hoc* (ou MANets), les réseaux *ad hoc* non mobiles (parfois appelés à tort MANets), les réseaux statiques, ou encore les réseaux de capteurs (généralement *ad hoc*). Pour chacun de ces types de réseaux, le fonctionnement d'un algorithme peut être soumis à des hypothèses particulières, comme l'existence d'un identifiant unique sur chaque nœud ou la connaissance d'une borne sur le nombre de nœuds, *etc.* Tout cela crée une grande diversité de résultats qui sont difficiles à comparer les uns aux autres.

Usage de la classification pour l'algorithmique

La classification présentée figure 4.15, bien que limitée, apporte déjà quelques enseignements. Si, par exemple, la solution à un problème donné nécessite que le graphe soit dans la classe $\mathcal{F}_7$, c'est-à-dire qu'il vérifie la propriété caractéristique de $\mathcal{F}_7$, alors on sait que tout graphe issu des classes $\mathcal{F}_3, \mathcal{F}_4, \mathcal{F}_6, \mathcal{F}_8$ ou $\mathcal{F}_9$ vérifiera également cette propriété caractéristique. Il en va de même pour les conditions suffisantes : si par exemple la condition $\mathcal{C}_4$ est une condition suffisante au succès d'un algorithme donné, alors les graphes issus des classes $\mathcal{F}_6, \mathcal{F}_8$ et $\mathcal{F}_9$ verront également l'algorithme atteindre son objectif.

Par ailleurs, s'il est montré qu'une classe de graphes dynamiques $\mathcal{F}_a$ est incluse dans une classe de graphes dynamiques $\mathcal{F}_b$, alors il est également possible d'affirmer qu'un algorithme à destination de $\mathcal{F}_b$ est moins contraignant sur la mobilité du système qu'un algorithme à destination de $\mathcal{F}_a$, et de dire qu'il est, en ce sens, plus général. Ce raisonnement peut ainsi aider un concepteur d'algorithmes à choisir ses hypothèses de départ quant à la mobilité du système.

Usage de l'algorithmique pour la classification

L'apport fonctionne bien entendu dans les deux sens, et l'analyse d'algorithmes peut, comme nous l'avons montré, servir de base à la caractérisation de nouvelles classes. L'étude de résultats existants, retranscrits dans le formalisme de nos travaux, est une piste de recherche envisagée.

Chapitre 5

Développements logiciels

Sommaire

5.1 Simulateur de réétiquetage de graphes dynamiques 70

5.1.1 Édition d'un algorithme 70

5.1.2 Contrôle de l'exécution 72

5.1.3 Quelques mots sur les aspects internes 74

5.2 Éditeur de graphes évolutifs 76

5.2.1 Format du fichier texte 76

5.2.2 Manipulations 77

5.3 Vérificateur de propriétés sur les graphes évolutifs 77

5.3.1 Propriétés supportées 77

5.3.2 Fonctionnement 79

5.4 Convertisseur *DGS* vers et depuis les graphes évolutifs 79

5.4.1 Madhoc 79

5.4.2 Description du format *Dynamic Graph Stream (DGS)* 79

5.4.3 Conversion 80

5.5 Récapitulatif de la chaîne logicielle 83

Dans cette section, nous présentons les développements logiciels liés à nos travaux. Ces développements comprennent quatre outils majeurs. Le premier est un simulateur de réétiquetage de graphes dynamiques (*simuDAGRS* [Cas07]), permettant l'édition et la visualisation d'algorithmes distribués dans un contexte topologique instable. Certains éléments de ce simulateur sont réutilisés dans le deuxième outil, un éditeur de graphes évolutifs interactif. Le troisième outil est un convertisseur permettant de passer de notre représentation des graphes évolutifs vers un autre format de représentation de graphes dynamiques (*Dynamic Graph Stream* [PD]), et *vice versa*. Le quatrième et dernier outil développé durant cette thèse est un vérificateur de propriétés pour graphes évolutifs. Cet outil permet à l'utilisateur de vérifier par exemple que des propriétés nécessaires et suffisantes relatives à un algorithme sont, ou non, satisfaites dans le graphe évolutif fourni en entrée. Enfin, le chapitre se termine par un schéma de synthèse regroupant ces outils et montrant de quelle manière ils peuvent être combinés les uns avec les autres.

5.1 Simulateur de réétiquetage de graphes dynamiques

Afin de pouvoir aisément tester et visualiser des algorithmes à base de réétiquetages sur des graphes dynamiques, nous avons développé un outil de visualisation s'inspirant de ce que propose la plateforme ViSiDiA[MMZG] dans un cadre statique. Cet outil est essentiellement utilisé comme support de réflexion pour la conception d'algorithmes, et non comme plateforme de test automatisée (la visualisation sur une centaine de sommets reste cependant confortable). Il permet notamment d'éditer des algorithmes, puis d'en tester le comportement sur une topologie dynamique, dont l'évolution est pilotable par l'utilisateur. Il est ainsi possible de casser ou de créer des liens de communication (arêtes), en ajoutant, déplaçant ou supprimant des nœuds du réseau (sommets du graphe) en quelques clics de souris, et ce, pendant l'exécution de l'algorithme étudié. Ce type de manipulation permet de tester les réactions de l'algorithme à différents événements, et donc de faire émerger rapidement d'éventuels défauts de conception.

5.1.1 Édition d'un algorithme

L'édition d'un algorithme se fait à travers deux onglets : l'onglet de spécification des états manipulés et l'onglet de spécification des règles de réétiquetage de l'algorithme. Les algorithmes ainsi édités peuvent être exportés puis réimportés dans le simulateur.

Édition des états

L'édition des états manipulés au travers de l'onglet montré figure 5.1 permet de spécifier les noms et les valeurs initiales des étiquettes présentes sur chaque sommet du graphe. A noter que le nombre de ces étiquettes (aussi appelées « registres d'étiquette ») n'est pas limité *a priori*. Il est également possible, si l'algorithme le nécessite, de spécifier les valeurs qui doivent être affectées à un sommet particulier, dit « sommet distingué », au début de l'exécution de l'algorithme (ce sommet sera alors automatiquement créé et placé en haut à gauche de la topologie au début de chaque exécution).

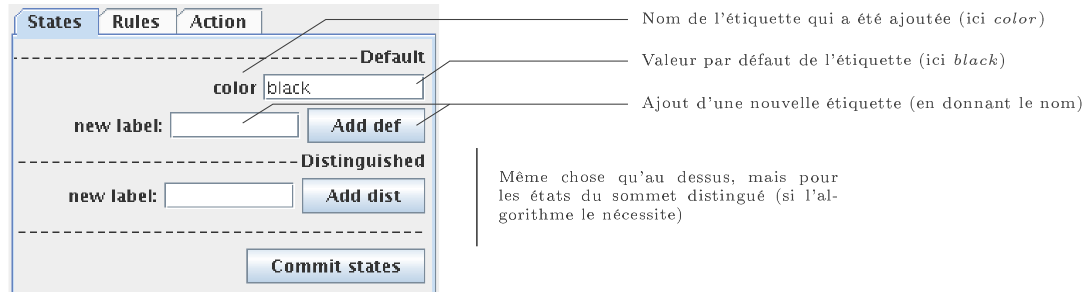


FIG. 5.1 – Édition des états manipulés par l'algorithme

Édition des règles de réétiquetage

L'édition des règles de réétiquetage au travers de l'onglet montré figure 5.2 permet de spécifier les règles qui composent l'algorithme, en donnant pour chacune les *préconditions*

et les *actions* correspondantes. Rappelons que pour une règle donnée, si les *préconditions* sont vérifiées, alors les *actions* correspondantes sont exécutées. Les priorités entre les règles *normales*, c'est-à-dire les règles qui ne sont pas des règles de réactions à la rupture, sont déterminées par leur ordre d'apparition dans l'algorithme (le cas des règles de réactions à la rupture est traité plus loin dans ce chapitre). Les deux sommets prenant part à l'application d'une règle *normale* sont contextuellement désignés par $v1$ et $v2$, les registres d'étiquette manipulés sont notés comme des attributs sur ces sommets, c'est-à-dire en utilisant une notation postfixée (*p. ex.* le registre $v1.monetiq1$ désigne le registre *monetiq1* du sommet $v1$). Deux registres sont automatiquement fournis pour les brins d'arêtes : *edgelabel*, qui représente l'étiquette d'état algorithmique du brin, et *edgestate*, qui vaut *off* si l'arête correspondante a été rompue, *on* sinon. La désignation de l'arête ($v1$,$v2$) est implicite et ces deux registres (*edgelabel* et *edgestate*) sont donc désignés comme attributs du sommet correspondant (*p. ex.* le registre $v1.edgelabel$ désigne l'étiquette du brin d'arête sortant de $v1$ vers $v2$). Dans la version actuelle du simulateur, les brins d'arête ne peuvent pas avoir d'autres registres que *edgelabel* et *edgestate*. Nous étudions actuellement la possibilité de leur donner, comme pour les sommets, un nombre de registres illimité.

FIG. 5.2 – Édition des règles de réétiquetage de l'algorithme

Chaque précondition est de la forme suivante :

$$registre\ comparateur\ operande\ [+\ modificateur]$$

Le comparateur est un caractère pris parmi $\{=, !, <, >\}$, signifiant respectivement *égal, différent, strictement inférieur, strictement supérieur*. L'opérande est soit un autre registre, soit une valeur. Les types de valeurs supportés pour les registres et les opérandes sont les entiers et les chaînes de caractères. Le modificateur est une valeur numérique (éventuellement négative) pouvant être ajoutée (opérateur +) à un opérande numérique. La vérification de la cohérence des types employés est, à ce jour, laissée à la charge de l'utilisateur.

Enfin, les préconditions peuvent être combinées les unes avec les autres à l'aide des opérateurs *et (&)* et *ou (|)*, et d'un parenthésage adapté, comme dans l'exemple suivant :

$$precondition\ 1\ \&\ (precondition\ 2\ |\ precondition\ 3)$$

Les actions suivent les mêmes règles de formation que les préconditions, à ceci près que la position du comparateur est occupée par l'opérateur d'affectation (=) et qu'elles ne se combinent les unes avec les autres qu'en utilisant des *et (&)*, par exemple :

$$(v1.label1\ =\ v2.label1\ +\ 2)\ \&\ (v1.label2\ =\ chaine)\ \&\ (v2.edgelabel\ =\ 3)$$

Un exemple de correspondance entre un algorithme de réétiquetage (forêt couvrante) et son codage dans l'éditeur est donné figure 5.3.

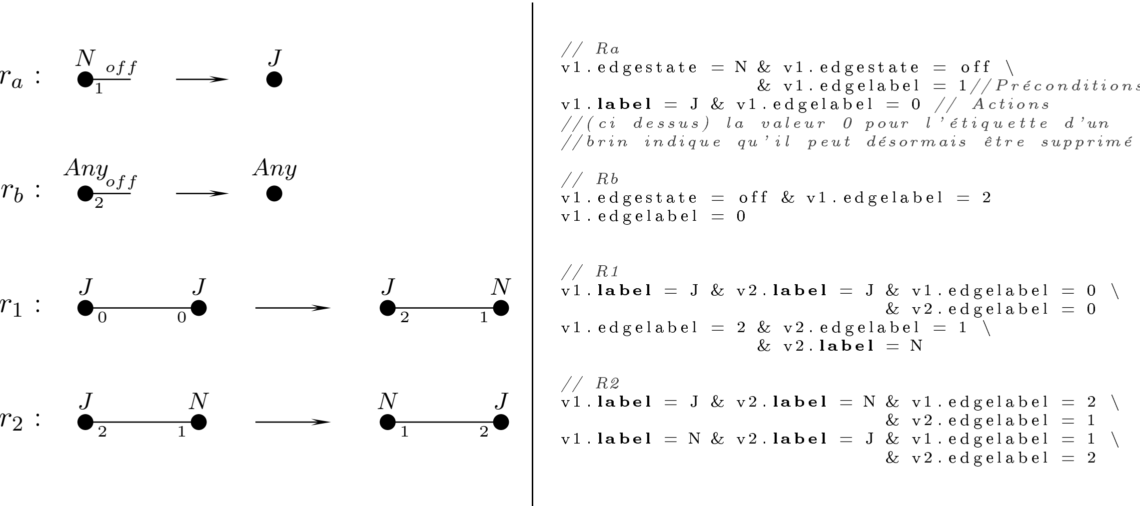


FIG. 5.3 – Représentation d'un algorithme dans le simulateur

Importation / exportation des algorithmes

Les algorithmes peuvent être importés et exportés depuis et vers des fichiers *via* les menus *>File>Save Algorithm* ou *>File>Open Algorithm*. La figure 5.4 décrit le format de fichier utilisé pour stocker les algorithmes.

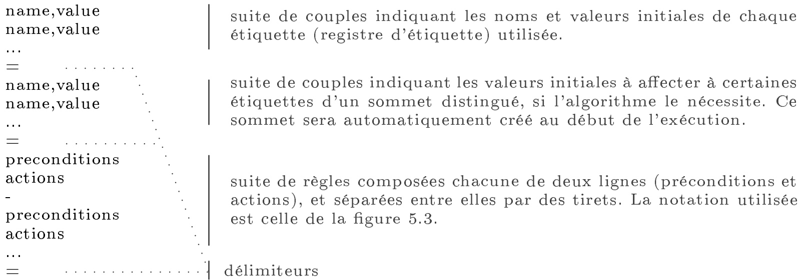


FIG. 5.4 – Structure d'un fichier d'algorithme pour le simulateur

5.1.2 Contrôle de l'exécution

L'utilisateur dispose de différentes options pour contrôler le déroulement d'un algorithme pendant son exécution. En premier lieu, il peut contrôler la topologie sur laquelle ont lieu les calculs. Il peut également modifier la cadence des calculs, en les suspendant ou en faisant varier leur vitesse jusqu'à la valeur maximale supportée par la machine sous-jacente. Enfin, il peut modifier l'état de tout sommet pendant le calcul, afin de produire ou reproduire des configurations particulières.

Contrôle de la topologie

A tout moment, l'utilisateur peut ajouter, déplacer ou supprimer des sommets. L'ajout se fait en cliquant avec le bouton gauche de la souris à l'emplacement souhaité pour le nouveau sommet, ce dernier étant initialisé avec les états initiaux prévus par l'algorithme. Les déplacements se font en effectuant un glisser déposer du sommet cible, de l'ancienne position à la nouvelle. Enfin, un clic droit sur un sommet a pour effet de le supprimer de la topologie. Les arêtes du graphe sont quant à elles gérées automatiquement en fonction des distances qui séparent les sommets : une arête relie deux sommets si et seulement si la distance entre ces deux sommets est inférieure à la porté radio définie. Cette portée peut être redéfinie à tout moment à l'aide d'une barre glissante. Enfin, un moteur d'événements aléatoires peut être activé ou désactivé à l'aide du bouton *{start/stop} rando*. Ce moteur ajoute, déplace ou supprime des sommets selon un algorithme ne se rapportant à aucun modèle de mobilité particulier (au sens des modèles étudiés par exemple dans la thèse de Hogie [Hog07]). L'algorithme utilisé ici décide de manière aléatoire s'il ajoute, supprime, ou déplace des sommets.

Vitesse de calcul

Comme indiqué ci-dessus, la vitesse d'exécution de l'algorithme peut être réglée à l'aide d'une barre glissante. La valeur ainsi fixée détermine le temps d'attente entre deux opérations. Le minimum correspond à une attente nulle et le maximum à une pause de l'exécution de l'algorithme. Cette fonctionnalité permet essentiellement d'observer visuellement les calculs lorsqu'ils sont trop rapides.

Modification des états

La modification des états se fait *via* le troisième onglet d'édition, montré figure 5.5. Après avoir sélectionné le sommet cible à l'aide du bouton central de la souris, on peut effectuer sur lui des actions de réétiquetage. L'onglet affiche aussi toutes les informations d'état relatives aux sommets, à savoir les valeurs de ses étiquettes (ici *counter* vaut *3* et *color* vaut *black*), et celles des étiquettes de tous ses brins d'arêtes (*le brin sortant vers le voisin 0 est étiqueté 2, et est opérationnel, celui sortant vers le voisin 1 est étiqueté 1, et est opérationnel*). L'utilisation de cet onglet ne suspend pas l'exécution.

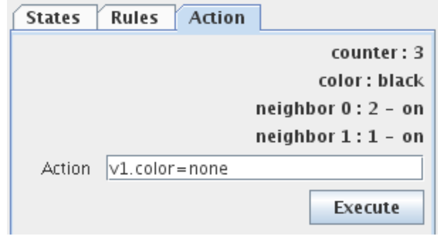


Fig. 5.5 – Édition d'états de sommets durant l'exécution de l'algorithme

5.1.3 Quelques mots sur les aspects internes

Ordonnancement des calculs

Contrairement à ce qui est en œuvre dans la plateforme ViSiDiA, dans notre outil chaque sommet du graphe ne possède pas son propre fil d'exécution (*Thread*). Les différentes unités de calcul sont toutes simulées par un seul et même fil d'exécution. Nous nous autorisons cette simplification dans la mesure où notre outil ne sert pas à fournir de statistiques sur les temps et les répartitions des calculs, mais simplement à permettre de les visualiser. En ce sens, notre outil est plus un outil de visualisation qu'un simulateur à proprement parler. La procédure de synchronisation actuellement utilisée est détaillée par l'algorithme 13. Cette procédure se substitue, dans le simulateur, aux procédures étudiées au chapitre 3.

Algorithme 13 Procédure de synchronisation utilisée par le simulateur

Répéter à l'infini les actions suivantes :

1. *Sélectionner aléatoirement un sommet v ;*
2. ***Pour chaque** brin d'arête de v étiqueté off, **Faire** :*
   - ***Si** l'étiquette du brin vaut 0, **Faire** :*
     - *supprimer le brin ;*
   - ***Sinon, Faire** :*
     - *Trouver la 1ère règle de rupture ayant l'état du brin pour préconditions ;*
     - *//(Les règles sont parcourues dans l'ordre donné par l'algorithme).*
     - *Appliquer cette règle ;*
   - ***Fin***

   Fin
3. ***Si** v a des voisins, **Faire** :*
   - *Sélectionner aléatoirement un sommet u parmi les voisins de v ;*
   - ***Pour chaque** règle r_i, **tant qu'**aucune règle n'a été appliquée, **Faire** :*
   - *//(Les règles sont parcourues dans l'ordre donné par l'algorithme).*
     - *$v_1 = v$; $v_2 = u$;*
     - ***Si** v_1 et v_2 vérifient les préconditions de r_i, **Faire** :*
       - *Appliquer les actions de r_i sur v_1 et v_2 ;*
       - *Terminer la procédure, qui redémarrera à l'étape 1 ;*
     - ***Sinon, Faire** :*
       - *Échanger v_1 et v_2, et réessayer cette même règle ;*
     - ***Fin***
   - ***Fin***

   Fin

Moteur de réétiquetage

Les règles de réétiquetage spécifiées par les algorithmes sont interprétées par le moteur de réétiquetage. Pour ce faire, les préconditions de chaque règle sont transformées en arbre binaire dont chaque feuille représente une précondition élémentaire, et dont les nœuds

internes portent des opérateurs *et* ou *ou*, comme illustré figure 5.6. Lors de l'exécution de l'algorithme, les vérifications de préconditions sont effectuées sur cet arbre, par un algorithme récursif qui remonte les évaluations des feuilles vers la racine.

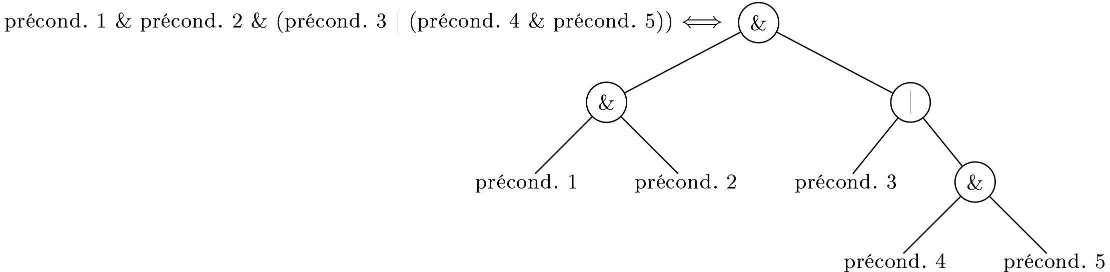


Fig. 5.6 – Exemple d'arbre binaire pour les préconditions

Pour des raisons de simplicité, la même structure est utilisée pour stocker les actions, bien que ces dernières ne soient combinées qu'en utilisant des *et* (&).

Disparition de liens

Lorsqu'un lien disparaît, il faut garder en mémoire le brin correspondant afin de permettre au moteur de réétiquetage d'appliquer les règles de réaction à la rupture, puis le supprimer lorsque ces traitements sont terminés. Une suppression d'arête se fait donc en plusieurs étapes. Tout d'abord, le démon chargé de surveiller l'évolution de la topologie marque les deux brins de l'arête disparue à off (étiquette *edgestate*). Les brins ayant conservé leur étiquette *edgelabel* dans l'état initial (valeur 0) sont ensuite définitivement supprimés par la procédure de synchronisation (algorithme 13 page ci-contre). Pour les autres, une règle de réaction à la rupture sera appliquée, et le brin correspondant sera définitivement supprimé quand l'algorithme lui aura affecté la valeur 0.

Aide à la visualisation

Afin de faciliter la visualisation du réseau, des informations graphiques sur l'état des sommets et des arêtes ont été ajoutées. Lorsqu'au moins un brin d'une arête possède une étiquette différente de 0 (la valeur 0 désignant par convention une arête sans intérêt particulier pour l'algorithme), l'arête correspondante est mise en relief avec un trait gras. De même, lorsqu'un des registres d'étiquette a pour nom *color*, le simulateur utilisera automatiquement sa valeur pour colorier le sommet. Ces effets, ainsi qu'une vue d'ensemble du simulateur, sont présentés figure 5.7 page suivante.

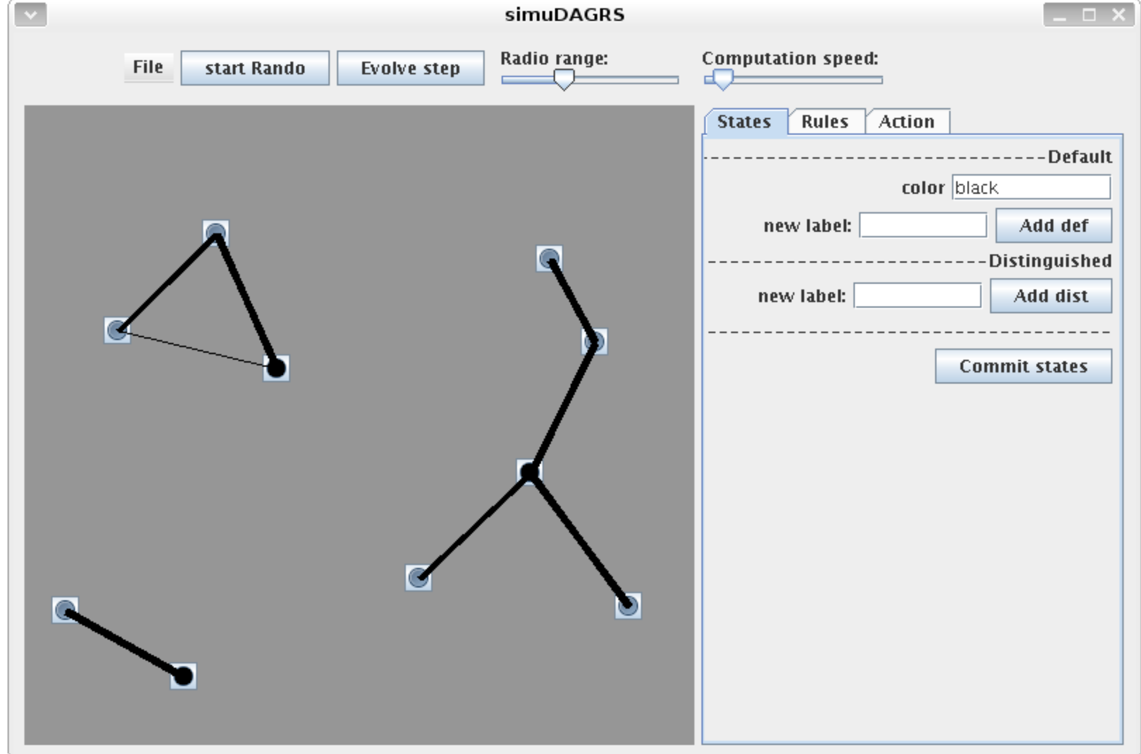


FIG. 5.7 – Vue d'ensemble du simulateur

5.2 Éditeur de graphes évolutifs

L'éditeur de graphes évolutifs présenté ici est un outil permettant de créer des graphes évolutifs (*c.f.* définition 4.7 page 47) de petite taille de manière interactive en n'utilisant que la souris. Les graphes ainsi créés peuvent être exportés sous forme de fichiers texte et sous forme d'images au format Encapsulated PostScript. Pour des raisons essentiellement pratiques, notamment pour réutiliser certaines classes de manipulation de la topologie, nous avons décidé d'intégrer cet éditeur au simulateur de réétiquetage de graphes (présenté à la section précédente).

5.2.1 Format du fichier texte

Le format utilisé pour exporter les graphes évolutifs dans des fichiers textes est décrit figure 5.8. Ce format s'inspire de celui qu'ont utilisé Monteiro, Goldman et Ferreira pour des simulations avec *NS2*, dans le cadre d'une étude de performance d'algorithmes de routage pour réseaux dynamiques [MGF06]. La liste des arêtes est précédée d'une liste des sommets, afin que les nœuds n'ayant pas de lien de communication soient tout de même consignés dans le fichier. Les sommets sont considérés comme présents du début à la fin (leur période d'absence peut être simulée par l'absence d'arêtes adjacentes durant cette période). Enfin, une section *dimensions*, qui est facultative, a été ajoutée pour faciliter la conversion du graphe en image.

Format	Explications	Exemple
dimensions dim_x dim_y ↵	Abscisse et ordonnée maximales parmi les coordonnées de tous les sommets, afin de faciliter la conversion du graphe évolutif en PostScript (facultatif).	dimensions 346 258
vertices id_1 $[pos_{x_1}$ $pos_{y_1}]$ ... id_i $[pos_{x_i}$ $pos_{y_i}]$ ... ↵	Liste complète des sommets ayant appartenu au graphe. Pour chaque sommet, des coordonnées fixes (facultatives) peuvent être fournies pour indiquer la position à lui affecter dans l'image PostScript.	vertices v1 346 50 v2 0 258 v3 97 0
edges id_{v_1} id_{v_2} d_1 d_2 d_3... ...	Liste complète des arêtes ayant appartenu au graphe. Chaque arête est identifiée par ses deux extrémités et est accompagnée d'une liste de dates (ou intervalles) indiquant ses périodes d'existence.	edges v1 v3 1 3 5-8 v2 v3 2-3

FIG. 5.8 – Format, explications et exemple de fichier stockant un graphe évolutif

5.2.2 Manipulations

Les manipulations permettant l'édition d'un graphe évolutif se limitent à quelques actions effectuées à la souris. La procédure, illustrée par la figure 5.9 est la suivante : dans un premier temps, l'utilisateur dispose les sommets de sa configuration initiale sur l'espace à deux dimensions de l'éditeur de topologie (figure 5.9(a)). Il peut également régler la portée radio régissant l'existence des arêtes entre ces sommets. A l'aide du bouton gauche (ajout ou déplacement), et du bouton droit (suppression), il fait varier cette topologie à sa guise. A chaque clic sur le bouton *Evolve step*, l'état du réseau pour la date courante est consigné, et la date courante est incrémentée (figures 5.9(b), 5.9(c) et 5.9(d)). Après avoir consigné l'état final souhaité, l'utilisateur peut exporter le graphe *via* le menu *File>Export evolving graph*. Cette action crée les fichiers *evolving.eps* (figure 5.9(e)) et *evolving.txt* (figure 5.9(f)) dans le répertoire *export* du simulateur, chacun de ces fichiers contenant une représentation du graphe évolutif. Les coordonnées utilisées sont les dernières en date ayant été enregistrées pour chaque sommet. Les sommets ayant été supprimés en cours de manipulation seront tout de même représentés dans le graphe exporté.

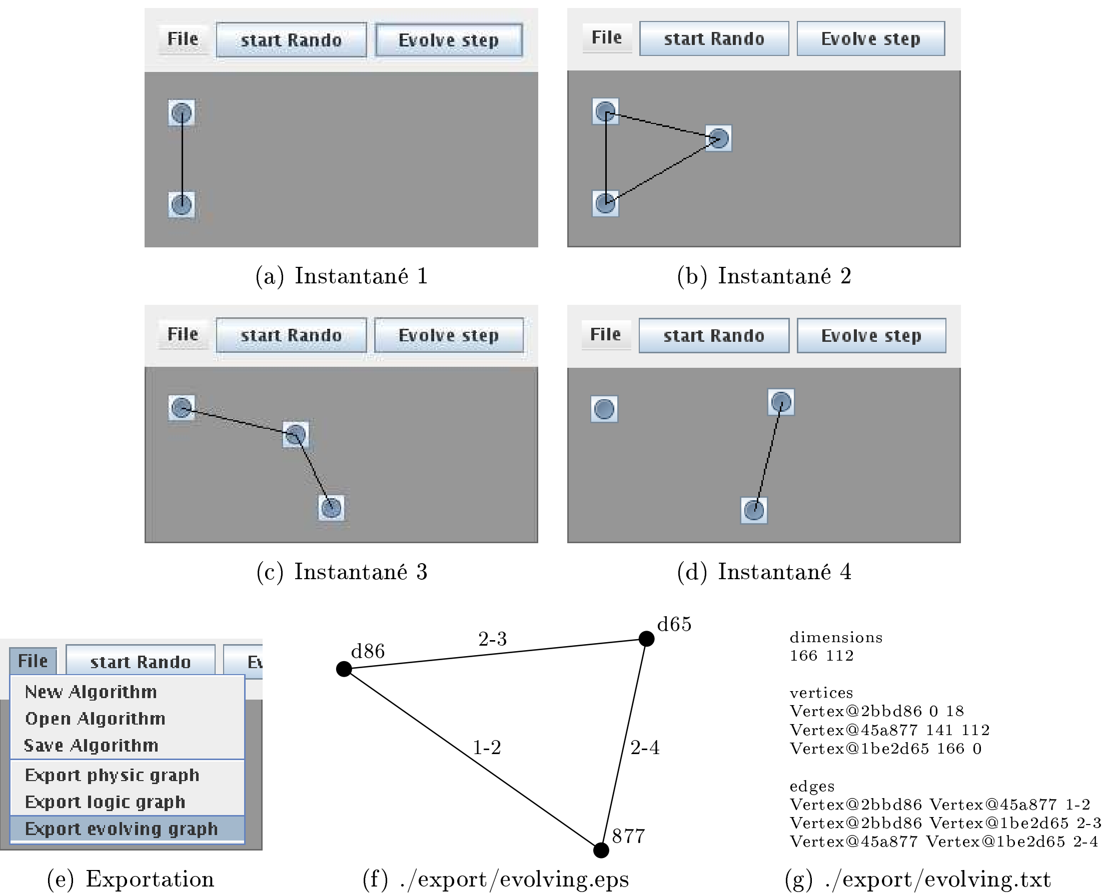
5.3 Vérificateur de propriétés sur les graphes évolutifs

Le chapitre 4 de ce document, dédié à l'analyse d'algorithmes, a permis de mettre en évidence différentes classes de réseaux dynamiques à partir de propriétés sur les graphes évolutifs. L'idée sous-jacente à l'outil présenté dans cette section est de pouvoir tester automatiquement ces propriétés sur des graphes évolutifs fournis en entrée.

5.3.1 Propriétés supportées

Les propriétés actuellement supportées sont les suivantes :

1. $\mathcal{P}_1 = \exists u \in V(\mathcal{G}) \mid \forall v \in V_{\mathcal{G}} \setminus \{u\}, \exists \mathcal{J}_{(u,v)}$
   (Il existe au moins un sommet pouvant joindre tous les autres par un trajet)



(a) Instantané 1

(b) Instantané 2

(c) Instantané 3

(d) Instantané 4

(e) Exportation

(f) ./export/evolving.eps

(g) ./export/evolving.txt

FIG. 5.9 – Édition d'un graphe évolutif

2. $\mathcal{P}_2 = \exists u \in V(\mathcal{G}) \mid \forall v \in V_{\mathcal{G}} \setminus \{u\}, \exists \mathcal{J}_{strict(u,v)}$
   (Il existe au moins un sommet pouvant joindre tous les autres par un trajet strict)

3. $\mathcal{P}_3 = \forall u, v \in V_{\mathcal{G}}, \exists \mathcal{J}_{(u,v)}$
   (Chaque sommet peut joindre tous les autres par un trajet)

4. $\mathcal{P}_4 = \forall u, v \in V_{\mathcal{G}}, \exists \mathcal{J}_{strict(u,v)}$
   (Chaque sommet peut joindre tous les autres par un trajet strict)

5. $\mathcal{P}_5 = \exists u \in V_{\mathcal{G}} \mid \forall v \in V_{\mathcal{G}} \setminus \{u\}, (u, v) \in E_{\mathcal{G}}$
   (Il existe au moins un sommet ayant, au cours du temps, une arête commune avec chaque autre sommet)

6. $\mathcal{P}_6 = \forall u \in V_{\mathcal{G}}, \forall v \in V_{\mathcal{G}} \setminus \{u\}, (u, v) \in E_{\mathcal{G}}$
   (Chaque sommet a, au cours du temps, une arête commune avec chaque autre sommet)

7. $\mathcal{C}_7 = \exists v \in V_{\mathcal{G}} \mid v \in \mathcal{D}est(\mathcal{G})$
   (Il existe un sommet pouvant être joint par tous les autres au cours du temps)

5.3.2 Fonctionnement

Le vérificateur prend en entrée un nom de fichier correspondant au graphe évolutif que l'on désire tester. Ce fichier doit être dans le format de la figure 5.8 page 77, *c.-à.-d.* dans celui utilisé par l'éditeur de graphes évolutifs pour ses exportations. Pour chacune de ces propriétés, le vérificateur répond *vrai* ou *faux*. Par combinaison des résultats obtenus, l'utilisateur peut ranger le graphe dynamique testé dans sa classe d'appartenance.

5.4 Convertisseur *DGS* vers et depuis les graphes évolutifs

Dans le cadre de recherches menées au Havre par l'équipe de Frédéric Guinand autour des graphes dynamiques, un format de description dédié a été élaboré. Ce format, Dynamic Graph Stream [PD], permet de décrire les évolutions topologiques d'un réseau en utilisant un flux d'événements consignés les uns à la suite des autres. Ce format est également utilisé par le simulateur Madhoc [HC] pour exporter (et à terme importer) des graphes dynamiques générés durant ses simulations. Après avoir brièvement décrit le simulateur Madhoc et décrit plus en détail le format Dynamic Graph Stream (DGS), nous présentons les algorithmes conçus et implémentés pour convertir ce format vers notre format de graphe évolutif, et *vice versa*.

5.4.1 Madhoc

Au sein d'un partenariat entre les universités du Havre et du Luxembourg, Luc Hogie a développé un simulateur de réseaux mobiles ad hoc, appelé Madhoc [HC](Metropolitan ad hoc network simulator). Madhoc compte à ce jour une dizaine de groupes de chercheurs utilisateurs ainsi qu'une dizaine de contributeurs. Ce simulateur, dédié aux réseaux mobiles sans fil, se positionne comme l'un des plus flexibles parmi l'offre actuelle (Network Simulator [NS2], GloMoSim [GLO] et OPNet [OPN], *etc.*). Ses cas d'utilisation vont de l'optimisation de protocoles à l'émulation de réseaux IEEE802.11b, en passant par l'étude d'agents mobiles ou la génération de graphes dynamiques. C'est cette génération de graphes dynamiques qui nous intéresse ici tout particulièrement, ainsi que la possibilité d'exporter les graphes générés vers le format DGS. Différents modèles de mobilité peuvent être utilisés dans Madhoc, comme par exemple le *random mobility model* [Bet01], le *random waypoint mobility model* [BRS03], ou encore un modèle personnalisable appelé *human mobility model* [Hog07]. Il est ainsi possible de disposer indirectement d'une large gamme de graphes dynamiques issus de modèles différents.

5.4.2 Description du format *Dynamic Graph Stream (DGS)*

Le format Dynamic Graph Stream (DGS) a été spécifié à l'université du Havre par Dutot, Guinand et Pigné (de la même équipe que *Madhoc*). Ce format est accompagné d'une bibliothèque Java du même nom permettant de le manipuler [PD]. Le principe de ce format est de décrire un graphe dynamique à travers le flux d'événements qui s'y produit (ajouts, suppressions de sommets et d'arêtes, changements d'état, *etc.*).

L'organisation d'un fichier au format DGS est la suivante : au début du fichier se trouve un en-tête donnant des informations sur la version de DGS utilisée, puis le nom du graphe, le nombre d'événements et le nombre d'étapes qui constituent le flux lui-même. Une étape

regroupe les événements qui se produisent entre deux relevés topologiques, ou *snapshots*, du réseau. L'en-tête comprend également des informations relatives à l'étiquetage des sommets et des arêtes. Le fichier contient ensuite les étapes et les événements eux-mêmes, de manière séquentielle, comme illustré figure 5.10.

```
DGS002                                          version DGS de référence
mongraphe 40 90                                 nom du graphe, nb d'étapes, nb d'évènements
nodes x :number y :number values :vector  }     décrit les étiquettes que peuvent posséder les
edges weight :number                      }     sommets et les arêtes du graphe (facultatif)
st 0
an "A" ... [autre étiquettes]     suite de primitives
an "B" ...                        st  : step           // numéro de l'étape
ae "AB" "A" "B"                   an  : add node       // ajoute un sommet
⋮                                 dn  : delete node    // efface un sommet
                                  ae  : add edge       // ajoute une arête
st 40                             de  : delete edge    // efface une arête
de "AB"                           cn  : change node    // change la valeur des étiquettes des sommets
dn "A"                            ce  : change edge    // change la valeur des étiquettes des arêtes
```

FIG. 5.10 – Exemple de fichier au format DGS

5.4.3 Conversion

Nous avons développé deux convertisseurs (un dans chaque sens) entre le format DGS (figure 5.10) et le format de graphes évolutifs que nous utilisons (figure 5.8 page 77). Dans cette section nous présentons les algorithmes réalisant ces conversions.

Conversion *DGS* → graphe évolutif

L'outil que nous avons développé pour convertir les *DGS* vers notre format de graphes évolutifs utilise la bibliothèque *GraphStream* mentionnée précédemment. Le traitement réalisé et décrit par l'algorithme 14 page ci-contre consiste à répertorier les sommets et arêtes prenant part au graphe, et à transformer les événements d'ajouts/suppressions d'arêtes en intervalles de temps de présence.

Conversion Graphe évolutif → DGS

La conversion d'un fichier contenant un graphe évolutif vers un fichier au format DGS se fait en deux temps. Un objet contenant le graphe évolutif est d'abord instancié au sein du convertisseur, puis utilisé pour récupérer les modifications topologiques. Ces dernières sont alors transformées en une suite d'événements, puis consignées dans le fichier de sortie. Ces dernières opérations sont illustrées par l'algorithme 15 page 82. Notre représentation des graphes évolutifs ne prenant en compte que les changements relatifs aux arêtes, les sommets sont tous ajoutés dès la première étape d'événements DGS.

Algorithme 14 Conversion DGS → Graphe évolutif

sommets, arêtes, débuts, fins, périodes ← ∅ ;
date ← 0 ;
Ouvrir le fichier DGS et sauter l'en-tête ;
Tant que ligne ← lireligne(), **Faire** :
  Si ligne==nouvelle étape, **Faire** :
    date ← date + 1 ;
  Sinon, Si ligne==ajout d'un sommet **s**, **Faire** :
    sommets ← sommets ∪ s ;
  Sinon, Si ligne==ajout d'une arête **e**, **Faire** :
    // (e est représenté par sa paire de sommets)
    arêtes ← arêtes ∪ e ;
    débuts[e] ← débuts[e] ∪ date ;
  Sinon, Si ligne==suppression de l'arête **e**, **Faire** :
    fins[e] ← fins[e] ∪ date ;
  Fin
Fin
Fermer le fichier DGS ;
Pour chaque arête **e**, **Faire** :
d_deb, d_fin ← 0 ;
  Tq d_deb ← datesuivante(débuts,e,d_deb), **Faire** :
    d_fin ← datesuivante(fins,e,d_fin) ;
    Si d_fin==null, **Faire** :
      d_fin ← date+1 ;
    Fin
    périodes[e] ← périodes[e] ∪ (d_deb,d_fin) ;
  Fin
Fin
Ouvrir le fichier de sortie en écriture ;
Écrire "vertices :" puis la liste des sommets ;
Écrire "edges :" puis la liste des arêtes, accompagnées de leurs périodes de présence ;
Fermer le fichier de sortie ;

Exemple pour une arête (a,b)

st 1
ae "AB" "A" "B"
...
st 3
de "AB"
...
st 5
ae "AB" "A" "B"
...
st 6
de "AB"
...
st 8
ae "AB" "A" "B"
...
↓
débuts[{A,B}]=(1,5,8)
fins[{A,B}]=(3,6)

↓

périodes[{A,B}]=
(1-3),(5-6),(8-lastdate)
//où lastdate est la dernière date de vie du réseau(date+1)

↓

edges :
...
A B 1 2 3 5 8 9 ... lastdate

Algorithme 15 Conversion Graphe évolutif → DGS

events[] ← ∅ ;
datemax ← dernière date recensée dans le graphe évolutif ;
Pour chaque *arête e,* ***Faire*** *:*
  Pour chaque *début d'intervalle de présence de e,* ***Faire*** *:*
    events[début] ← events[début] + "ae " + e ;
  Fin
  Pour chaque *fin d'intervalle de présence de e,* ***Faire*** *:*
    events[fin] ← events[fin] + "de " + e ;
  Fin
Fin
Ouvrir le fichier de sortie ;
Écrire l'en-tête ;
Écrire "st 0" ;
Pour chaque *sommet,* ***Faire*** *:*
  Écrire "an " + sommet ;
Fin
Écrire events[0] (si non vide) ;
Pour chaque *date de 1 à datemax,* ***Faire*** *:*
  Écrire "st " + date ;
  Écrire events[date] (si non vide) ;
Fin
Fermer le fichier de sortie ;

5.5 Récapitulatif de la chaîne logicielle

Les outils qui ont été présentés dans ce chapitre sont très variés dans leur nature (édition, test, visualisation, conversion). Ils s'articulent tous cependant autour d'un seul et même but : étudier les algorithmes distribués dans le cadre des réseaux dynamiques. Le simulateur de réétiquetage permet entre autres d'assister la conception d'algorithmes. Une fois les algorithmes conçus, leur analyse (non automatisée) met en évidence les conditions nécessaires et suffisantes au bon déroulement de leur exécution (*c.f.* chapitre 4). Ces conditions, exprimées sous forme de propriétés sur les graphes évolutifs, peuvent ensuite être testées sur des graphes de petite taille issus de l'éditeur ou sur des graphes plus importants issus des simulations de *Madhoc* et convertis pour l'occasion. Les résultats du vérificateur de propriétés sur ces graphes et ces propriétés permettent enfin d'avoir de bonnes indications sur le fait qu'un algorithme donné est adapté, ou non, à un contexte de mobilité donné. La figure 5.11 récapitule les possibilités de combinaison de ces outils.

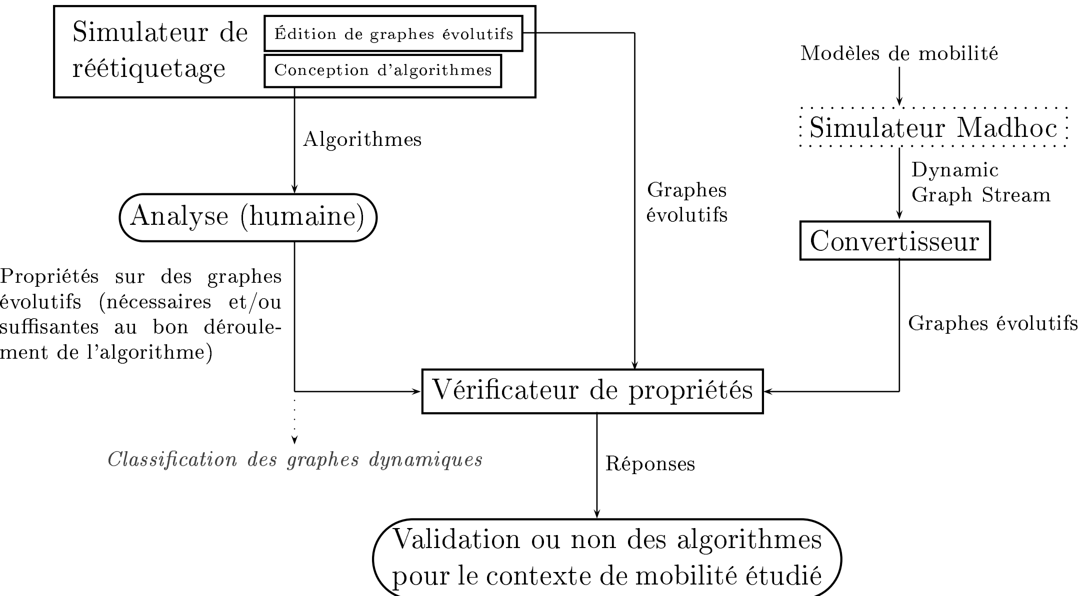


FIG. 5.11 – Liens possibles entre les outils logiciels développés

Chapitre 6

Assistance au développement d'applications réelles

Sommaire

6.1 Principe général **86**
6.2 Détail de l'architecture et canonicité du développement **86**
6.2.1 Lien entre la synchronisation et les réétiquetages 87
6.2.2 Lien entre les réétiquetages et l'application 87
6.3 Illustration de la généricité du développement à travers deux exemples **89**
6.3.1 Exemple 1 : Algorithme de propagation 89
6.3.2 Exemple 2 : Algorithme de la forêt couvrante 90
6.3.3 Réflexions sur la généricité des algorithmes 90
6.4 Extension du modèle à la non-atomicité des réétiquetages . . . **92**
6.4.1 Problématique du monde réel 92
6.4.2 Au niveau algorithmique 93
6.4.3 Au niveau applicatif 93

Dans ce chapitre nous proposons une architecture distribuée à trois niveaux qui permet à une application d'utiliser un algorithme de réétiquetage de graphes comme base de développement. En utilisant cette architecture, l'application délègue aux réétiquetages la gestion de l'organisation du réseau et se focalise sur les traitements de haut niveau qui lui sont spécifiques.

L'architecture proposée est tout d'abord présentée de manière synthétique, pour donner au lecteur l'intuition qui lui facilitera la lecture de la suite de ce chapitre, suite dans laquelle nous décrivons le fonctionnement des différentes couches et détaillons les liens que l'application entretient avec les réétiquetages. Ces liens seront illustrés à travers deux exemples récurrents dans ce document : la propagation d'information et la forêt d'arbres couvrants. Ces deux exemples servent également de base à une discussion sur la généricité qu'offrent par nature les réétiquetages de graphes. Enfin, une extension est proposée pour prendre en compte le caractère interruptible des opérations de calculs. Ce problème se pose dès lors qu'on ne considère plus les réétiquetages comme étant atomiques, hypothèse que nous avons faite dans le cadre de nos travaux théoriques.

6.1 Principe général

Les nœuds du réseau exécutent un algorithme de réétiquetage de manière distribuée, ce qui suppose qu'ils sont équipés d'une couche de synchronisation et d'un moteur de réétiquetage tous deux embarqués. A chaque fois qu'une règle est jouée, l'application est prévenue par le moteur de réétiquetage, qui lui indique la règle exécutée, le rôle joué dans cette règle (sommet de gauche ou sommet de droite), ainsi que l'identifiant (au sens réseau, et non algorithmique) du voisin concerné. L'application exécute alors un traitement de haut niveau propre à la règle appliquée. A chaque règle de l'algorithme de réétiquetage correspond donc un traitement effectif dans l'application.

L'architecture complète, illustrée pour deux nœuds figure 6.1, est composée de 3 parties. Au niveau le plus bas se trouve la couche de synchronisation, dont la principale fonction est de permettre aux nœuds du réseau de s'appairer le temps d'une étape de calcul commune. Cette couche gère aussi les détection de nouveaux voisins ou de voisins perdus. Sur cette couche de synchronisation repose un moteur de réétiquetage distribué qui se charge d'appliquer les règles de l'algorithme. A chaque fois qu'une règle est appliquée, le moteur de réétiquetage prévient l'application en invoquant une méthode spécifiée par le développeur (une pour chaque règle de l'algorithme).

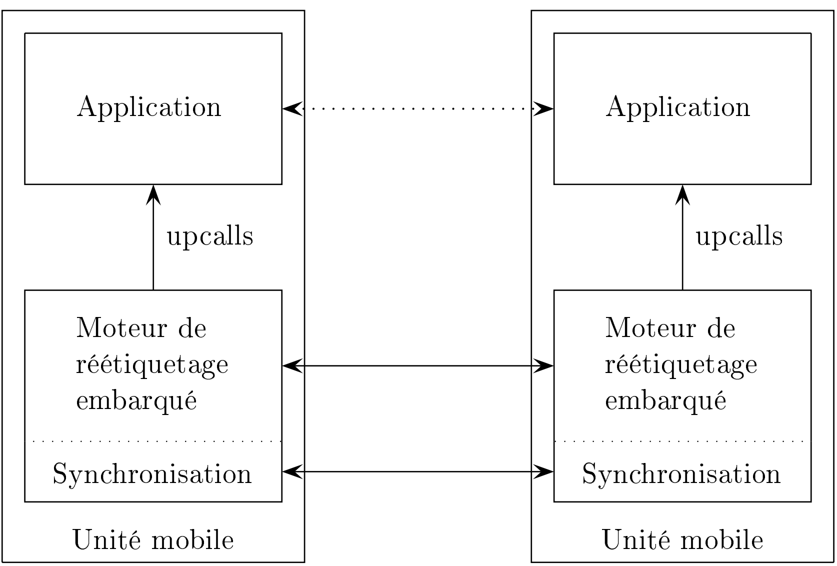


FIG. 6.1 – Architecture du système

6.2 Détail de l'architecture et canonicité du développement

L'objectif du chapitre étant de traiter des aspect génie logiciel de notre modèle de calcul, nous nous sommes limités, pour la synchronisation et le fonctionnement interne du moteur de réétiquetage, à donner les principes de fonctionnement par le biais de procédures

générales. Les liens que le moteur de réétiquetage entretient avec l'application sont, en revanche, plus détaillés.

6.2.1 Lien entre la synchronisation et les réétiquetages

Nous supposons ici que la couche de synchronisation fonctionne selon le mode de synchronisation à la demande présenté chapitre 3, dans sa version basique (section 3.2 page 37), à savoir que chaque nœud émet ou reçoit des demandes de synchronisation, qui peuvent être acceptées ou refusées de part et d'autre. Des couples de nœuds synchronisés émergent ainsi régulièrement, prêts à appliquer une étape de calcul commune. Lorsqu'un nouveau voisin est détecté, qu'un ancien voisin ne répond plus, ou qu'une synchronisation est réussie, la couche de synchronisation déclenche la procédure suivante dans le moteur de réétiquetage :

Algorithme 16 Procédure interne du moteur de réétiquetage

Si *un nouveau voisin v a été détecté,* ***Faire*** *:*
-Créer un brin local pour représenter l'arête vers v ;
-Affecter à ce brin la valeur par défaut prévue par l'algorithme (c.f. section 2.3.3) ;
Sinon*, si un voisin v a disparu du voisinage,* ***Faire*** *:*
-Positionner à off *le brin qui représentait l'arête vers v ;*
-Appliquer la première règle de réaction à la rupture dont les préconditions correspondent, et en avertir l'application.
Sinon*, si une synchronisation est réussie avec un voisin v,* ***Faire*** *:*
-Envoyer à v l'état du sommet local et du brin local correspondant.
-Recevoir ces mêmes informations de v.
-Appliquer, si elle existe, la première règle dont les préconditions correspondent, et en avertir l'application.
(si les deux sommets peuvent jouer le rôle de chaque côté de la règle, alors celui dont l'identifiant réseau est alpha-numériquement le plus petit jouera, par convention, le côté gauche.)
Fin

6.2.2 Lien entre les réétiquetages et l'application

Nous supposons ici que le moteur de réétiquetage est écrit en langage *Java*, qu'il se présente à l'application sous la forme d'une classe *REngine* et qu'il peut prendre un algorithme en paramètre de constructeur. L'utilisation d'un algorithme $\mathcal{A}$ par une application se fait en plusieurs étapes que nous décrivons maintenant.

Préalablement au développement de l'application, une interface (au sens *objet* du terme) est générée à partir de l'algorithme que l'on désire utiliser. Cette interface $\mathcal{I}$ comporte autant de méthodes qu'il y a de règles dans l'algorithme. Pour les règles « normales », la signature comporte deux arguments : *Node n* et *boolean left*. La classe *Node* est une classe qui comporte, au minimum, l'adresse réseau du voisin avec qui la règle est appliquée. *left* indique le côté de la règle joué par le sommet sous-jacent (gauche si *true*, droit sinon). Pour les règles de réaction à la rupture, la signature ne comporte que l'élément *Node n*, qui

représente le voisin perdu. L'interface générée est illustrée figure 6.2. L'algorithme qui sert d'exemple dans cette figure est composé d'une règle normale r_1 et une règle de réaction à la rupture r_a.

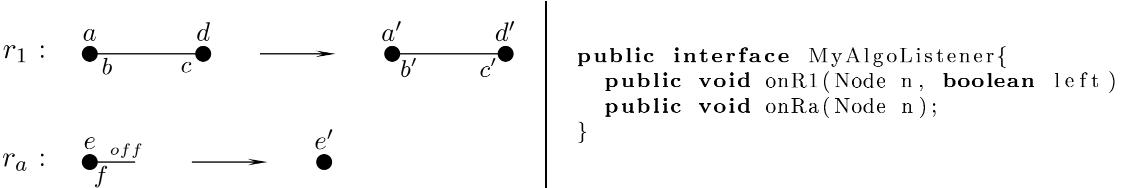


FIG. 6.2 – Interface générée depuis un algorithme

L'utilisation effective de l'algorithme se fait ensuite en deux temps. Le développeur doit tout d'abord fournir le code spécifique qui sera exécuté lorsque chaque règle est jouée. Cela se fait en *implémentant* dans une classe de l'application l'interface générée. Le lien avec le moteur de réétiquetage est ensuite finalisé à l'exécution quand, après avoir instancié un objet de type *REngine*, cette classe s'enregistre comme *Listener* auprès de l'instance du moteur. En d'autres termes, l'application souscrit aux événements de l'algorithme. Cette utilisation est illustrée figure 6.3.

Observation 4 *Nous avons fait le choix de ne pas permettre à l'application d'accéder aux états internes de la couche de réétiquetage, dont elle fait abstraction. Par ailleurs, du fait que chaque réétiquetage engendre l'invocation d'une méthode, l'application peut maintenir elle-même toutes les informations dont elle a besoin.*

```
public class MyApplication
              implements MyAlgoListener{
  // déclaration du moteur de réétiquetage
  private REngine re;
  MyApplication{
    // création du moteur, avec
    // l'algorithme en paramètre
    re = new REngine(``./myalgo.dagrs'');
    // souscription aux événements de l'algorithme
    re.setListener(this);
    re.start();
  }
  public void onR1(Node n, boolean left){
    // code à exécuter lorsque r1 est appliquée
  }
  public void onRa(Node n){
    // code à exécuter lorsque ra est appliquée
  }
}
```

FIG. 6.3 – Utilisation du moteur de réétiquetage dans une application

6.3 Illustration de la généricité du développement à travers deux exemples

Dans cette section, nous présentons deux exemples d'application reposant directement sur un algorithme à base de réétiquetage de graphe. Nous menons ensuite une réflexion sur la généricité offerte par ce type d'algorithme.

6.3.1 Exemple 1 : Algorithme de propagation

L'algorithme de propagation présenté ici est une variante de l'algorithme 2 page 27. Cet algorithme est constitué d'une seule règle qui représente la transmission d'une information entre deux nœuds voisins. Si la règle est appliquée de manière répétée, alors cela représente la propagation d'une information sur un réseau, de proche en proche, par un processus de type inondation. L'algorithme de propagation et son interface associée sont présentés figure 6.4.

r_1 : A —0—0— N ⟶ A —1—2— A

```
public interface PropagationListener{
  public void onR1(Node n, boolean left);
}
```

FIG. 6.4 – Algorithme de propagation et interface associée

L'utilisation effective de cet algorithme est donné figure 6.5. Il est important de remarquer ici que l'organisation du réseau est gérée de manière transparente par les couches de synchronisation et de réétiquetage, en accord avec l'algorithme fourni. Le développeur de l'application se consacre ainsi uniquement au code effectif des envois et des réceptions.

```
public class MyApplication
             implements PropagationListener{
  private REngine re;
  MyApplication{
    re = new REngine(''./propagation.dagrs'');
    re.setListener(this);
    re.start();
  }
  public void onR1(Node n, boolean left){
    if (left==true){ // si côté gauche
      Socket s = new Socket(n.address, port);
      ... // instructions d'envoi
    }else{ // si côté droit
      ServerSocket ss = new ServerSocket(port);
      Socket s = ss.accept();
      ... // instructions de réception
    }
  }
}
```

FIG. 6.5 – Utilisation de l'algorithme de propagation dans une application

6.3.2 Exemple 2 : Algorithme de la forêt couvrante

L'algorithme dont nous parlons dans cette section construit et maintient une forêt d'arbres couvrants sur un réseau. Il a été étudié tout au long de ce document et son fonctionnement est décrit section 2.3.4 page 23. La figure 6.6 montre l'interface qui y est associée.


```java
public interface SpanningForestListener{
    public void onRa(Node n);
    public void onRb(Node n);
    public void onR1(Node n, boolean left);
    public void onR2(Node n, boolean left);
}
```

FIG. 6.6 – Algorithme de la forêt d'arbres couvrants et interface associée

Dans l'implémentation que nous fournissons figure 6.7, l'objectif est simplement de transmettre à l'application la connaissance de l'arbre entretenu par le moteur de réétiquetage. Le code fourni ici maintient donc à jour un tableau de voisins dans l'arbre, avec une référence particulière pour le *père*.

Le détail du code est le suivant : lorsque la règle r_1 est appliquée, le voisin correspondant est ajouté dans la liste des parents, et si le nœud sous-jacent joue le rôle du sommet côté droit de la règle, le voisin est également enregistré comme *père*. Lorsque la règle r_2 est appliquée, le voisinage ne change pas mais la relation *père/fils* est inversée entre les deux nœuds considérés. Lors de l'application des règles r_a et r_b, la référence au voisin perdu est simplement enlevée du tableau de parents, en indiquant pour r_a que le nœud sous-jacent n'a plus de père.

Comme pour l'exemple 1, il est important de remarquer que quelques lignes de code (une vingtaine) suffisent pour mettre à la disposition du reste de l'application une gestion automatisée d'un arbre couvrant. C'est-à-dire qu'à tout moment, et de manière transparente, l'application peut disposer d'un arbre (qui tend à couvrir sa composante connexe dans le réseau) et que cet arbre, grâce à l'algorithme de réétiquetage fourni, est considéré par l'application comme « auto-entretenu ».

6.3.3 Réflexions sur la généricité des algorithmes

Nous avons vu dans ce chapitre, et surtout à travers les deux exemples précédents, qu'un jeu de règles abstraites de réétiquetage pouvait être complété par un ensemble de traitements concrets, apportant ainsi une signification effective à l'algorithme. Un algorithme de réétiquetage peut ainsi être vu comme un mécanisme abstrait de gestion du réseau, abstrait dans le sens où il ne spécifie pas les traitements effectifs que son exécution déclenche. En ce sens, un algorithme de réétiquetage est générique, et il peut servir de base à différentes applications de haut niveau.

Par exemple, l'algorithme de l'exemple 1 (figure 6.4 page précédente) que nous avons jusqu'à présent nommé « algorithme de propagation », a en fait, de même que son interface

```
public class MyApplication implements SpanningForestListener{

  private REngine re;
  public Vector neighbors;
  public Node father;
  public Node me;

  MyClass{
    re = new REngine (``./spanforest.dagrs'');
    re.setListener (this);
    re.start();
    neighbors = new Vector();
  }
  public void onRa(Node n){
    neighbors.remove(n);
    father = null;
  }
  public void onRb(Node n){
    neighbors.remove(n);
  }
  public void onR1(Node n, boolean left){
    neighbors.add(n);
    if (!left){
      father = n;
    }
  }
  public void onR2(Node n, boolean left){
    if (left){
      father = n;
    }else{
      father = null;
    }
  }
}
```

FIG. 6.7 – Utilisation de l'algorithme de la forêt couvrante dans une application

associée, une vocation bien plus générale que celle de propager de l'information. Le même algorithme pourrait par exemple être utilisé dans un réseau statique pour construire un arbre couvrant, en supposant que les étiquettes A (resp. N) signifient que le sommet correspondant est déjà (resp. pas encore) intégré à l'arbre, et que l'étiquette 1 (resp. 2) désigne un lien partant vers un père (resp. un fils). Un même algorithme peut ainsi donner lieu à différents traitements, selon le sens que lui donne l'application.

Plusieurs significations peuvent également être données à l'algorithme de l'exemple 2 (figure 6.6 page 90), que nous avons jusqu'ici nommé « algorithme de la forêt couvrante ». Il peut par exemple servir d'algorithme d'élection probabiliste, en imaginant que les sommets étiquetés J sont les sommets élus, avec la garantie d'un et un seul sommet élu par arbre (le nombre d'arbres tendant vers 1 par composante connexe du graphe). Lorsque la connexité est rompue, un nouveau leader est généré. Si l'on désire utiliser ce type de propriétés, par exemple pour régir l'exclusion mutuelle à une ressource donnée, le seul développement à effectuer consiste à placer le code utilisant la ressource dans la méthode onR_2 (pour $left$ valant $faux$) d'une classe utilisant le moteur.

6.4 Extension du modèle à la non-atomicité des réétiquetages

6.4.1 Problématique du monde réel

Dans ce chapitre, nous avons proposé une architecture pour l'utilisation concrète d'un algorithme de réétiquetage au sein d'une application. Cette architecture permet de spécifier un code à exécuter pour chaque règle de l'algorithme. Afin que l'état du système de réétiquetage soit constamment en phase avec l'état de l'application, ces codes doivent être exécutés pendant les réétiquetages, *c.-à-d.* que l'exécution de chaque code doit commencer après le début, et se terminer avant la fin du réétiquetage correspondant dans le moteur de réétiquetage, comme indiqué figure 6.8.

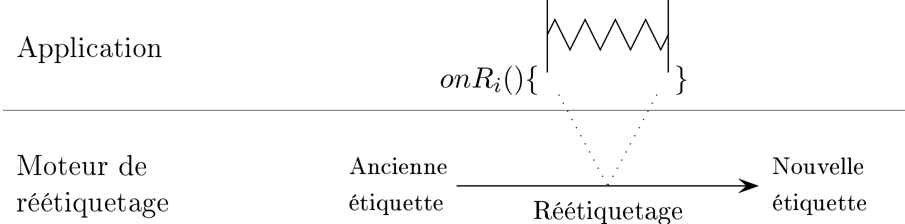


FIG. 6.8 – Déroulement d'un réétiquetage avec exécution de code

Nous avons dans un cadre théorique considéré les réétiquetages comme étant des opérations atomiques. En revanche, les traitements associés aux règles de réétiquetage ne sont pas, eux, atomiques. La question qui vient à l'esprit est alors de savoir ce qui peut être fait pour aider à gérer les ruptures des liens de communication qui se produisent pendant l'application d'une règle sur l'arête correspondante. Dans la suite, nous nommons de tels événements « ruptures à chaud ».

r_1 : a d b c $\longrightarrow$ a' d' b' c' $||$ a'' *off* b'' *off* d'' c''

Fig. 6.9 – Règle de réétiquetage munie d'états de secours

6.4.2 Au niveau algorithmique

S'il n'y avait que les réétiquetages à prendre en considération, alors une solution basique pour les ruptures à chaud serait de restaurer les états antérieurs au réétiquetage courant sur les nœuds correspondants. Cette solution ne convient évidemment pas si l'on tient compte des traitements qui s'exécutent au niveau supérieur. En effet, pour que le système reste cohérent en pareil cas, il faudrait que l'exécution de chaque méthode invoquée depuis le moteur soit réversible, ce qui représente pour le développeur une contrainte considérable.

La solution que nous proposons, plus souple, consiste à rajouter optionnellement une troisième partie aux règles. Cette partie indique les états qui doivent être affectés aux nœuds en cas de rupture à chaud. L'arête n'existant plus, ces états dits « de secours » sont gérés par chacun des deux sommets, indépendamment l'un de l'autre, et le concepteur doit prévoir le cas où un seul des deux sommets considère avoir été interrompu, l'autre ayant terminé ses traitements. Les règles sont donc maintenant constituées de triplets *(préconditions, actions, actions de récupération)*, que l'on représente comme sur la figure 6.9.

Au niveau du moteur de réétiquetage, l'état de secours est en fait considéré comme un état intermédiaire normal entre l'ancien état et le nouvel état. Le déroulement d'un réétiquetage est alors le suivant : quand le réétiquetage commence, les états de secours sont affectés aux nœuds sous-jacents, puis la méthode correspondant à la règle appliquée est invoquée. Si la méthode termine normalement, le nouvel état stable est affecté au nœud, sinon l'état intermédiaire est conservé (figure 6.10). Dans les deux cas, le cycle de réétiquetage courant est terminé. Il est ensuite du ressort de l'algorithme de faire en sorte que chaque état de secours figure dans les préconditions d'au moins une règle.

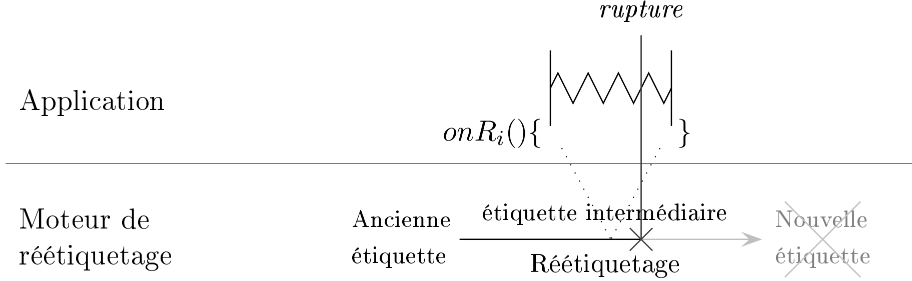


Fig. 6.10 – Interruption d'un réétiquetage avec états de secours

6.4.3 Au niveau applicatif

La rupture d'un lien de communication pendant l'application d'une règle, et donc pendant l'exécution du code correspondant, *c.à.d.* une rupture à chaud, ne pose pas systématiquement problème. En effet, si les deux nœuds ont terminé les communications relatives

à l'étape courante, ou si cette étape ne nécessite aucune communication, alors ils peuvent continuer chacun de leur côté l'exécution de la méthode correspondante, sans se soucier du lien de communication commun.

La plupart des langages de programmation à vocation générale permettent, à plus ou moins haut niveau, de gérer des erreurs de communication réseau. En *C* par exemple, cela correspond à des codes d'erreurs retournés par les fonctions *connect*, *read*, *write*, *etc.* En *Java* des exceptions peuvent être levées lors de l'invocation des méthodes correspondantes. Notre proposition est la suivante : lorsqu'une de ces erreurs survient, l'application réagit par le traitement souhaité puis décide, selon l'état d'avancement de ses communications, si l'objectif du réétiquetage a été, ou non, atteint. S'il a été atteint, alors la méthode invoquée doit retourner *vrai*, sinon elle doit retourner *faux* pour prévenir le moteur de réétiquetage.

La gestion des « ruptures à chaud » proposée ici modifie ainsi (de manière minime) la génération des interfaces objet associées aux algorithmes de réétiquetage. Cette modification consiste à remplacer le type de retour indéterminé (*void*), par le type booléen (*boolean*). Au niveau du moteur de réétiquetage, l'état intermédiaire, s'il a été prévu, est affecté au sommet sous-jacent au début du réétiquetage, puis la méthode est invoquée. Si la méthode retourne *vrai*, le nouvel état est affecté, sinon l'état intermédiaire (ou l'état initial si aucun état intermédiaire n'est prévu) est conservé.

Chapitre 7

Directions de recherches

Les travaux effectués dans le cadre de cette thèse ont ouvert un certain nombre de pistes de recherche, aussi bien dans un cadre théorique que pratique. Le présent chapitre regroupe quelques unes de ces pistes, certaines d'entre elles déjà très précises et d'autres plus générales devant être affinées.

Perspectives issues du chapitre 2 page 13 : Modèles de calculs et formalismes associés

1) Modèle de calcul de l'étoile « ouverte »

Le chapitre 2 a présenté quelques-uns des modèles de calculs locaux les plus répandus. Parmi ces modèles, présentés figure 2.1 page 17, nous nous sommes restreints à l'étude de ceux qui n'impliquaient que deux sommets à chaque étape de calculs (modèles (c) et (d)), car les procédures de synchronisations associées aux modèles (a) et (b) impliquant plus de deux sommets ne nous semblaient pas réalistes dans un contexte fortement dynamique. Pour rappel, ces procédures synchronisent tous les sommets d'une boule de rayon 1 (étoile ouverte) ou d'une boule de rayon 2 (étoile fermée) pour chaque étape de calcul effectuée. Une variante du modèle de l'étoile ouverte (figure 7.1) pourrait cependant avoir de bonnes propriétés dans le cadre des réseaux sans fils.

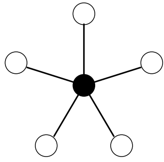


Fig. 7.1 – Modèle de calcul de l'étoile ouverte

En effet, si l'on accepte que deux opérations puissent se dérouler en parallèle lorsque les deux boules concernées ont des centres disjoints (et non plus des centres éloignés d'une distance de 2), alors ce modèle ne nécessite aucune synchronisation. Il suffit à chaque sommet de diffuser son état à intervalle régulier, et de modifier son état en fonction des états reçus de ses voisins. Nous pensons que ce modèle, bien qu'ayant une puissance de calcul assez faible, peut être adapté à certains types de traitements (comme par exemple

la propagation d'information), et mérite donc d'être étudié.

2) Algorithmes

Nous avons proposé quelques algorithmes pour illustrer les réétiquetages de graphes dynamiques. Il serait intéressant pour certains d'entre eux, et notamment ceux utilisant des circulations de jeton(s), de compléter les explications que nous avons fournies par une étude probabiliste, en déclinant différents contextes de mobilités et différentes propriétés sur les arbres (taille et forme notamment). Comme nous l'avons mentionné, l'équipe RI2C du Havre travaille actuellement sur le sujet. Une autre piste intéressante serait de porter ces algorithmes sur de vrais terminaux mobiles communicants, afin de connaître les temps de convergences (des arbres, des compteurs, des candidats, *etc.*) effectifs dans le contexte technologique actuel (*p.ex.* avec *802.11* ou *802.15*).

Perspectives issues du chapitre 3 page 33 : Synchronisation entre voisins

Dans le troisième chapitre de ce document nous avons proposé un mode de synchronisation à la demande. Dans la procédure associée, chaque sommet attend d'éventuelles demandes de synchronisation provenant de ses voisins pendant une durée aléatoire. Si aucune demande ne lui a été adressée à l'issue de cette période il effectue lui-même une demande à un de ses voisins. L'impact de la durée d'attente choisie, et plus précisément de l'intervalle au sein duquel cette durée est tirée aléatoirement, est actuellement étudiée au sein de notre équipe. Les premiers résultats sont donnés figure 7.2. On constate notamment qu'une valeur d'attente trop petite dégrade les performances, du fait de l'encombrement du réseau (radio) d'une part et du nombre plus important de sommets occupés lorsque des demandes leurs sont adressées d'autre part. Les intervalles permettant le plus grand nombre de synchronisations sont ceux allant de [100,200] à [140,280] millisecondes (cellules entourées d'un cadre gris sur la figure).

Un rapport de recherche à paraître [ACC07] présentera des mesures effectuées en faisant varier d'autres paramètres tels que la durée des calculs opérés par la couche supérieure entre deux synchronisations, que nous avons fixée à 10 millisecondes dans ces simulations, et le délai d'attente des réponses (*timeout*), que nous avons fixé à 50 millisecondes. Ce rapport comparera également les résultats obtenus avec ceux qu'obtiennent d'autres modes de synchronisation comme l'algorithme du *Rendezvous* présenté section 3.1.3 page 35.

Perspectives issues du chapitre 4 page 45 : Analyse d'algorithmes

1) Algorithmes analysés

Dans le quatrième chapitre nous avons analysé quelques algorithmes simples *à finalité* (algorithmes qui doivent atteindre un objectif précis à l'issue de leur exécution). Il serait intéressant d'effectuer le même type d'analyse pour des algorithmes plus compliqués, tel que l'algorithme de propagation de document (algorithme 3 page 29), ou pour des algorithmes de *maintien*, tel que celui de la forêt d'arbres couvrants (algorithme 1 page 25). Une question intéressante serait par exemple de savoir s'il existe une condition suffisante sur la dynamique du réseau (*c.-à.-d.* une propriété sur un graphe évolutif) pour que chaque

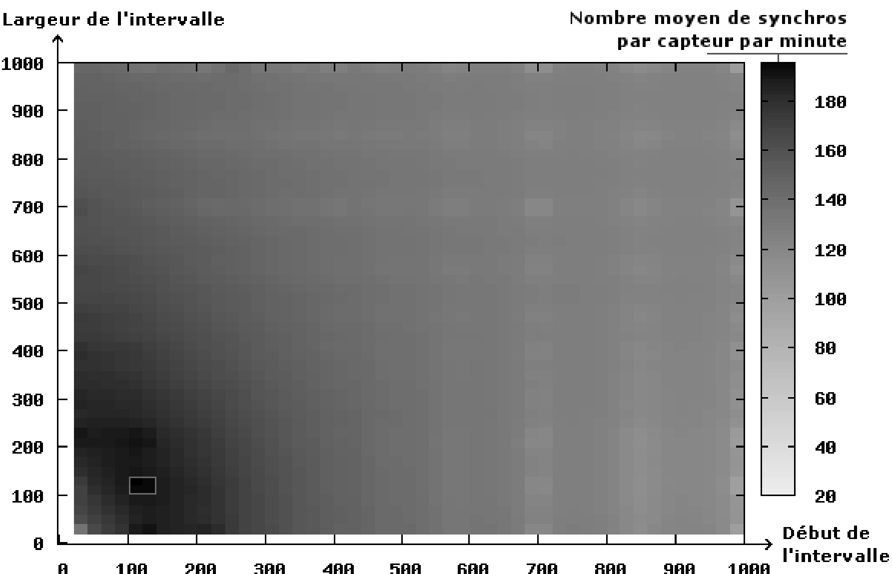


FIG. 7.2 – Impact du délai d'attente dans la procédure de synchronisation à la demande

composante connexe soit couverte par un unique arbre, et ce, en un nombre borné de périodes suivant chaque événement topologique.

2) Impact de la synchronisation sur l'analyse

Les caractérisations que nous avons effectuées dans le cadre de l'analyse du chapitre 4 ne font aucune hypothèse sur la synchronisation sous-jacente aux calculs. Nous pensons que le mode de synchronisation à la demande avec critères (voir section 3.3 page 38) peut avoir un impact important sur les conditions nécessaires et suffisantes. Il serait par exemple intéressant de regarder, pour les algorithmes de comptage présentés, si certains critères de synchronisation permettent d'obtenir des conditions moins contraignantes que celles que nous avons déjà obtenues.

3) Classification

L'élaboration d'une classification des réseaux dynamiques basée sur l'analyse d'algorithmes distribués et les graphes évolutifs représente pour nous un axe de recherche prioritaire. Nous envisageons dans un premier temps d'analyser par le biais de graphes évolutifs un certain nombre d'algorithmes existants. La classification pourra être alimentée, dans un second temps, par des travaux issus d'autres domaines, tels que l'étude des réseaux de neurones ou des réseaux d'interactions sociales observées chez l'homme ou chez la fourmi.

Perspectives issues du chapitre 5 page 69 : Développements logiciels

1) Simulateur de réétiquetages

Plusieurs pistes de travaux, plus pratiques que théoriques, peuvent être envisagées autour du simulateur de réétiquetages de graphes que nous avons développé. Il serait

par exemple intéressant de pouvoir intégrer dans les simulations des options concernant la synchronisation, et notamment de permettre l'utilisation de la synchronisation à la demande avec critères. La question de l'intégration de notre outil à la plateforme ViSiDiA s'est également posée de manière récurrente pendant cette thèse. Ce sont les importantes différences d'objectif (réseaux dynamiques) et de conception (*Threads* pour les sommets, choix de l'interprétation ou de l'exécution pour les algorithmes, *etc.*) entre notre plateforme et ViSiDiA qui ont conduit à ne pas réaliser, pour l'instant, cette intégration. Cette piste devra néanmoins être étudiée par la suite.

2) Vérification automatique de propriétés sur les graphes évolutifs

Le vérificateur de propriétés décrit au chapitre 5 pourra progressivement être étendu aux propriétés que la classification de graphes dynamiques fera émerger.

Perspectives issues du chapitre 6 page 85 : Assistance au développement d'applications réelles

L'architecture décrite dans le sixième chapitre n'a pas été validée expérimentalement. Son implémentation sur de vrais terminaux mobiles communicants pourrait fournir des résultats intéressant, tant sur le plan de la synchronisation et des algorithmes que sur celui de l'ingénierie logicielle.

Conclusion

Avant de conclure ce document, il convient de repositionner la contribution de cette thèse dans le contexte général de l'algorithmique distribuée. Cette discipline a fait l'objet de nombreux travaux ces dernières décennies et on peut noter deux approches fondamentalement différentes dans la manière d'aborder le sujet. La première de ces approches consiste à se positionner dans un contexte technologique donné, et à y résoudre différents problèmes. La seconde, symétrique, consiste à sélectionner des problèmes fondamentaux donnés, et à étudier leurs solutions dans différents contextes. D'un point de vue théorique, la notion de contexte se traduit généralement par des hypothèses qui sont faites sur le réseau sous-jacent. Certains problèmes fondamentaux comme l'élection, le dénombrement ou encore la construction de structures couvrantes ont souvent été utilisés dans ce cadre et ont permis à la communauté d'élaborer progressivement un ensemble d'hypothèses récurrentes dans le domaine de l'algorithmique distribuée. On peut citer parmi les plus classiques de ces hypothèses celles qui ont trait à la forme du réseau, à sa taille, à sa connexité ou encore aux connaissances initiales que possèdent les nœuds. L'étude de ce type de problèmes dans le cadre des réseaux dynamiques, en regard des différentes hypothèses (seconde approche), n'a pas encore été très explorée. Cette thèse est une contribution aux outils théoriques facilitant ce type d'étude. Nous espérons à terme faire émerger des hypothèses récurrentes portant sur la dynamique des réseaux, complétant ainsi la gamme des hypothèses généralement considérées.

Autour du modèle de calcul

Le chapitre 2 a proposé une adaptation du modèle des calculs locaux au domaine des graphes dynamiques. Le choix de ce modèle comme point de départ s'est imposé de lui-même pour différentes raisons. Tout d'abord, les calculs locaux font abstraction du modèle de communication sous-jacent, ce qui permet de s'affranchir d'un contexte technologique donné. Ensuite, ils ne permettent que des étapes de calcul impliquant des voisins directs dans le réseau, ce qui nous semble être la seule approche réaliste dans un contexte dynamique général. L'adaptation proposée permet, en plus des opérations normales entre voisins, d'ajouter à un algorithme la capacité de réagir à l'apparition ou à la disparition d'un voisin. Ce nouveau modèle de calcul a été illustré par un certain nombre d'algorithmes ne faisant aucune hypothèse sur le réseau. L'algorithme de la forêt couvrante est d'ailleurs, à notre connaissance, le premier algorithme totalement décentralisé à ne faire aucune hypothèse sur la dynamique du réseau ou sur l'existence d'identifiants uniques pour les sommets. La convergence rapide d'un tel algorithme, qui est une question fondamentale, n'est évidemment pas assurée, mais en revanche un certain nombre de propriétés sont

garanties quoi qu'il se produise dans le réseau.

Dans le chapitre 3, nous nous sommes attachés à proposer un mode de synchronisation adapté aux réseaux dynamiques. La synchronisation (qui peut être abstraite selon le niveau d'abstraction auquel on se place) est le processus qui régit la répartition des calculs entre les différents voisins dans le réseau. Le mode de synchronisation proposé a deux propriétés intéressantes. La première est qu'il tolère les ruptures de liens de communication, condition nécessaire dans un contexte dynamique. La seconde est qu'il permet au concepteur d'algorithmes d'influer sur la répartition des calculs en favorisant certains liens de communication plutôt que d'autres. Les critères de sélection pour ces liens peuvent être variés, et nous avons proposé des mécanismes permettant d'en gérer plusieurs et éventuellement de les combiner.

Dans les chapitres 5 et 6, nous nous sommes intéressés à des aspects plus pratiques de notre modèle de calcul. Le chapitre 5 a, entre autres, présenté le simulateur de réétiquetage de graphes que nous avons développé. Ce simulateur permet d'éditer des algorithmes décrits dans notre modèle de calcul et de visualiser leur exécution sur une topologie dynamique dont les évolutions sont contrôlées par l'utilisateur. Dans le chapitre 6, nous avons proposé une architecture logicielle permettant à une application de se décharger de l'organisation du réseau sous-jacent en utilisant un algorithme décrit dans notre modèle. Cette architecture nous a également donné l'occasion de nous intéresser aux ruptures de liens de communication qui surviennent en cours de réétiquetage. Une extension du modèle prenant en compte ce facteur a été proposée en fin de chapitre.

Autour de l'analyse

Le chapitre 4 est central à nos travaux. Il présente un nouveau cadre théorique pour l'analyse de problèmes fondamentaux en contexte dynamique. Ce cadre d'analyse ne prend pas en compte les performances ou la complexité des algorithmes, mais permet la caractérisation des hypothèses nécessaires ou suffisantes, en terme de dynamique, à la réalisation de leurs objectifs. L'outil combinatoire qui sert de support à ce cadre d'analyse est le graphe évolutif. Ce type de graphe est utilisé à tous les étages de l'analyse, à savoir pour la définition des objectifs d'un algorithme, pour la déclinaison de ses exécutions possibles, et pour l'expression des propriétés nécessaires et suffisantes qui résultent de l'analyse. Ces propriétés, à l'instar des hypothèses usuelles dans les graphes statiques, induisent des classes de graphes bien particulières regroupant chacune les graphes dynamiques pour lesquels un algorithme donné fonctionne, ou ne fonctionne pas. Les exemples d'analyse donnés, portant sur un algorithme de propagation et deux algorithmes de dénombrement, nous ont permis de mettre en évidence une dizaine de ces classes, que nous avons hiérarchisées. L'ensemble des outils et des méthodes que nous avons développés pour l'analyse semble ainsi pouvoir contribuer à l'élaboration d'une classification des réseaux dynamiques. La poursuite de cette classification est un axe de recherche privilégié dans la suite de ces travaux.

Bibliographie

[AAD$^+$06] D. Angluin, J. Aspnes, Z. Diamadi, M.J. Fischer, and R. Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed Computing*, pages 235–253, March 2006.

[ACC07] J. Albert, A. Casteigts, and S. Chaumette. Experimental comparison of different synchronization procedures on a sensor network. Technical Report 1434-07, LaBRI, Université Bordeaux 1, Juillet 2007.

[Ang80] D. Angluin. Local and global properties in networks of processors. In *Proceedings of the 12th Symposium on theory of computing*, pages 82–93, 1980.

[BBC$^+$93] C.H Bennett, G. Brassard, C. Crépeau, R. Jozsa, A. Peres, and W. Wootters. Téléportation quantique. http ://www.cs.mcgill.ca/ crepeau/tele.html, 1993.

[Bet01] C. Bettstetter. Smooth is better than sharp : A random mobility model for simulation of wireless networks. In *ACM MSWiM*, 2001.

[BRS03] C. Bettstetter, G. Resta, and P. Santi. The node distribution of the random waypoint mobility model for wireless ad hoc networks. *IEEE Trans. Mobile Computing*, 2(3) :257–269, July–September 2003.

[Cas06] A. Casteigts. Model Driven Capabilities of the DA-GRS Model. In *Intl. Conf. on Autonomic and Autonomous Systems (ICAS'06).* IEEE, 2006.

[Cas07] A. Casteigts. SimuDAGRS, a platform for design and visualization of localized distributed algorithms based on dynamic graph relabelings. available at http ://www.labri.fr/perso/casteigt/simulator.html, 2007.

[CC05] A. Casteigts and S. Chaumette. Dynamicity Aware Graph Relabeling Systems - a model to describe MANet algorithms. In *17th Intl. Conf. on Parallel and Distributed Computing and Systems (PDCS'05)*, 2005.

[CC06] A. Casteigts and S. Chaumette. Dynamicity Aware Graph Relabeling Systems and the Constraint Based Synchronization : a Unifying Approach to Deal With Dynamic Networks. In *Intl. Conf. on Wireless Algorithms, Systems and Applications (WASA'06).* LNCS, 2006.

[Cha06] J. Chalopin. *Algorithmique distribuée, calculs locaux et homomorphismes de graphes.* PhD thesis, University of Bordeaux I, France, 2006.

[CM05] J. Chalopin and Y. Métivier. A bridge between the asynchronous message passing model and local computations in graphs. In *Mathematical Foundations of Computer Science (MFCS 2005)*, volume 3618 of *Lecture Notes in Computer Science*, pages 212–223. Springer-Verlag, August 2005.

[CMZ04] J. Chalopin, Y. Métivier, and W. Zielonka. Election, naming and cellular edge local computations. In *second international conference on graph transformation*, pages 242–256. Lectures Notes in Computer Science, 2004.

[CPW] Article *Calcul des Prédicats*, sur Wikipedia. http ://en.wikipedia.org/wiki/Calcul_des_prédicats.

[Fer04] A. Ferreira. Building a reference combinatorial model for manets. *IEEE Network*, 18(5) :24–29, 2004.

[FGP02] A. Ferreira, J. Galtier, and P. Penna. Topological design, routing and handover in satellite networks. pages 473–493, February 2002.

[FISR07] H. Frey, F. Ingelrest, and D. Simplot-Ryl. Localized minimum spanning tree based multicast routing with energy-efficient guaranteed delivery in ad hoc and sensor networks. Technical Report RT-0337, INRIA, 2007.

[FSF] Free Software Foundation, General Public Licence (GPL). http ://www.gnu.org/copyleft/gpl.html.

[GÖ3] F.C. Gärtner. A survey of self-stabilizing spanning-tree construction algorithms, 2003.

[GL93] F. Glover and M. Laguna. Tabu search. In C. Reeves, editor, *Modern Heuristic Techniques for Combinatorial Problems*, Oxford, England, 1993. Blackwell Scientific Publishing.

[GLO] Global Mobile Information Systems Simulation Library (GloMoSim). http ://pcl.cs.ucla.edu/projects/glomosim/.

[HC] L. Hogie and Contributeurs. The Madhoc Simulator - Rapport Technique, Université du Havre, 2005. http ://www-lih.univ-lehavre.fr/~hogie/madhoc/.

[HMR+06] A. El Hibaoui, Y. Métivier, J.M. Robson, N. Saheb-Djahromi, and A. Zemmari. Analysis of a randomized dynamic timetable handshake algorithm. Technical Report 1402-06, LaBRI, Université Bordeaux 1, May 2006.

[Hog07] L. Hogie. *Mobile Ad Hoc Networks : Modelling, Simulation and Broadcast-based Applications (chapitre 6)*. PhD thesis, University of Luxembourg, 2007.

[Jar05] A. Jarry. *Connexité dans les réseaux de télécommunications (chapitres 4,5,6)*. PhD thesis, University of Nice Sophia-Antipolis, 2005.

[JMB01] D. Johnson, D. Maltz, and J. Broch. *DSR The Dynamic Source Routing Protocol for Multihop Wireless Ad Hoc Networks*, pages 139–172. Addison-Wesley, 2001.

[LLW] Article *Leslie Lamport*, sur Wikipedia. http ://en.wikipedia.org/wiki/Leslie_Lamport.

[LMS95] I. Litovsky, Y. Métivier, and E. Sopena. Different local controls for graph relabeling systems. *Mathematical Systems Theory*, 28(1) :41–65, 1995.

[Lyn89] N. Lynch. A hundred impossibility proofs for distributed computing. In *Proceedings of the 8th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 1–28, New York, NY, 1989. ACM Press.

[MGF06] J. Monteiro, A. Goldman, and A. Ferreira. Performance evaluation of dynamic networks using an evolving graph combinatorial model. In *Proceedings of the 2nd IEEE International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob'06)*, pages 173–180, Jun 2006.

[MMOS04] Y. Métivier, M. Mosbah, R. Ossamy, and A. Sellami. Synchronizers for local computations. In *International conference on graph transformation*, volume 3256 of *Lecture Notes in Comput. Sci.*, pages 271–286. Springer-Verlag, 2004.

[MMZG] M. Mosbah, Y. Métivier, A. Zemmari, and E. Godard. ViSiDiA. http ://www.labri.fr/projet/visidia/.

[MO04] M. Mosbah and R. Ossamy. A programming language for local computations in graphs : Computational completeness. In IEEE, editor, *Proceedings of the 5th. Mexican International Conference in Computer Science Colima Mexico 20-24 September*, pages 12–19. Computer Society, 2004.

[MSZ00] Y. Métivier, Nasser Saheb, and Akka Zemmari. Randomized rendezvous. In *Colloquium on mathematics and computer science : algorithms, trees, combinatorics and probabilities*, Trends in mathematics, pages 183–194. Birkhäuser, 2000.

[MSZ02] Y. Métivier, N. Saheb, and A. Zemmari. Randomized local elections. *Inf. Process. Lett.*, 82(6) :313–320, 2002.

[MSZ03] Y. Métivier, N. Saheb, and A. Zemmari. Analysis of a randomized rendezvous algorithm. *Inf. Comput.*, 184(1) :109–128, 2003.

[NS2] Network Simulator (ns-2). http ://www.isi.edu/nsnam/ns/.

[OPN] OPNET Simulator. http ://www.opnet.com/.

[PC97] V.D. Park and M.S. Corson. A Highly Adaptive Distributed Routing Algorithm for Mobile Wireless Networks. In *INFOCOM (3)*, pages 1405–1413, 1997.

[PD] Y. Pigné and A. Dutot. The DGS (Dynamic Graph Stream) file format, Université du Havre. http ://graphstream.sourceforge.net/dgs.html.

[Per97] C. Perkins. Ad-hoc On-demand Distance Vector routing. In *MILCOM '97 panel on Ad Hoc Networks*, 1997.

[RTH+03] R. Riggs, A. Taivalsaari, J. Huopaniemi, M. Patel, J. VanPeursem, and A. Uotila. *Programming Wireless Devices with Java 2 Platform, Micro Edition.* Addison-Wesley Professional, June 2003. Second edition.

[SMR06] M. Schwager, J. McLurkin, and D. Rus. Distributed coverage control with sensory feedback for networked robots. In *Proceedings of Robotics : Science and Systems*, Philadelphia, PA, August 2006.

[TS01] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems : Principles and Paradigms.* Prentice Hall PTR, 2001.

[TWB03] A. Troël, F. Weis, and M. Banâtre. Découverte automatique entre terminaux mobiles communicants. Technical Report 4979, INRIA, 2003.

[WM97] M. Werner and G. Maral. Traffic flows and dynamic routing in LEO intersatellite link networks. In 5th *International Mobile Satellite Conference (IMSC'97)*, June 1997.

Contribution à l'algorithmique distribuée
dans les réseaux mobiles ad hoc
-
Calculs locaux et réétiquetages de graphes dynamiques

Résumé :
Les réseaux mobiles ad hoc sont par nature instables et imprévisibles. De ces caractéristiques découle la difficulté à concevoir et analyser des algorithmes distribués garantissant certaines propriétés. C'est sur ce point que porte la contribution majeure de cette thèse. Pour amorcer cette étude, nous avons étudié quelques problèmes fondamentaux de l'algorithmique distribuée dans ce type d'environnement. Du fait de la nature de ces réseaux, nous avons considéré des modèles de calculs locaux, où chaque étape ne fait collaborer que des nœuds directement voisins. Nous avons notamment proposé un nouveau cadre d'analyse, combinant réétiquetages de graphes dynamiques et graphes évolutifs (modèle combinatoire pour les réseaux dynamiques). Notre approche permet de caractériser les conditions de succès ou d'échec d'un algorithme en fonction de la dynamique du réseau, autrement dit, en fonction de conditions nécessaires et/ou suffisantes sur les graphes évolutifs correspondants. Nous avons également étudié la synchronisation sous-jacente aux calculs, ainsi que la manière dont une application réelle peut reposer sur un algorithme de réétiquetage. Un certain nombre de logiciels ont également été réalisés autour de ces travaux, notamment un simulateur de réétiquetage de graphes dynamiques et un vérificateur de propriétés sur les graphes évolutifs.

Mots-clés : Réseaux mobiles ad hoc, Graphes évolutifs, Réétiquetages de graphes dynamiques, Algorithmique distribuée

---

Contribution to distributed algorithmics
in mobile ad hoc networks
-
Local computations and dynamic graph relabelling systems

Abstract :
Mobile ad hoc networks are by nature unpredictable and unstable. These characteristics make it difficult to design and analyze distributed algorithms which ensure given properties. This problem has been the main concern of the present thesis, in which we have first studied some classical distributed problems in this context. Due to the nature of the network, we have considered local computation models, in which each step involves direct neighbors only. We have designed a new analysis framework, based on graph relabellings, that allows us to characterize algorithms according to the hypothesis they require on the network evolution, and more precisely by using necessary and/or sufficient conditions on the corresponding evolving graphs (a combinatorial model for dynamic networks). We have also proposed a new synchronization procedure, and studied the use of local computation models in real applications. At last, a number of softwares have been developed throughout this work, among which a simulator for dynamic graph relabellings and a tool to check properties on evolving graphs.

Keywords : Mobile ad hoc networks, Evolving graphs, Dynamic graph relabelling systems, distributed algorithmics