Autour des groupes tolérants aux délais dans les flottes mobiles communicantes

Matthieu Barjon



THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

par **Matthieu Barjon**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

Autour des groupes tolérants aux délais dans les flottes mobiles communicantes

---

Thèse encadrée par

Arnaud Casteigts (co-encadrant), Serge Chaumette (co-directeur)
et Colette Johnen (co-directrice)

Soutenue le 1er décembre 2016

Après avis des rapporteurs :

Fouchal, Hacène	Professeur des Universités, CReSTIC
Roose, Philippe	Maître de conférences HDR, LIUPPA

Devant la commission d'examen composée de :

Casteigts, Arnaud	Maître de conférences, LaBRI	Co-encadrant
Chaumette, Serge	Professeur des Universités, LaBRI	Co-directeur
Devismes, Stéphane	Maître de conférences, VERIMAG Lab	Examinateur
Fouchal, Hacène	Professeur des Universités, CReSTIC	Examinateur
Johnen, Colette	Professeur des Universités, LaBRI	Co-directrice
Mosbah, Mohamed	Professeur des Universités, LaBRI	Président du jury
Roose, Philippe	Maître de conférences HDR, LIUPPA	Examinateur
Serfaty, Véronique	Docteur, DGA	Examinatrice

– 2016 –

Résumé Parmi les évolutions majeures de l'informatique, nous distinguons l'émergence des technologies mobiles sans fil. Le développement actuel de ces technologies permet de réaliser des communications ad-hoc directes entre de nombreux types d'entités mobiles, comme des véhicules, des robots terrestres ou des drones. Dans un réseau de tels équipements, l'ensemble des liens de communication qui existe à un instant donné dépend des distances entre les entités et la topologie du réseau change continuellement lorsque les entités se déplacent. Les hypothèses habituelles sur la connexité du réseau n'ont pas leur place ici, néanmoins, une autre forme de connexité appelée connexité temporelle est souvent disponible à travers le temps et l'espace. L'objectif de cette thèse a été de développer des algorithmes pour les flottes d'appareils dans le cas des réseaux tolérant aux délais (DTN). De manière simplifiée, les réseaux tolérants aux délais sont des réseaux pour lesquels certaines parties peuvent se retrouver isolées pendant un moment sans que cela pose problème. Nous nous intéressons, en particulier, au cas où ces appareils sont organisés sous la forme de groupes, et où la notion de groupe elle même survit à ces déconnexions transitoires. Ainsi, une grande partie de la thèse s'articule autour de la notion des groupes tolérant aux délais (groupe DTN). Dans notre cas cet éloignement est limité dans le temps et nous parlons alors de "diamètre temporel borné" au sein du groupe. Le fait de borner le diamètre temporel du groupe lui permet de distinguer entre l'éloignement temporaire d'un nœud et sa perte définitive (crash ou autre).

Title On Delay-Tolerant Groups in Communicating Mobile Fleets

Abstract Among the major developments in computer science, we distinguish the emergence of mobile wireless technologies. The current development of these technologies allows for direct ad-hoc communications between many types of mobile entities, such as vehicles, land robots or drones. In a network of such devices, the set of communication links that exists at a given instant depends upon the distances between the entities. As a result, the topology of the network changes continuously as the entities move. The common assumption on connectivity may not be relevant in this case, but another kind of connectivity called temporal connectivity is often alvailable over time and space. The goal of this thesis has been the development of algorithms for fleets of mobile devices in the case of delay-tolerant networks. In a simpler way, the delay-tolerant networks are networks where some parts can be isolated during a certain time without problems. We are interested, in particular, in the case where the devices are organised as groups, and where the notion of group itself survives to these deconnections. Hence, a big part of this thesis relates to the notion of delay-tolerant groups (DTN groups). In our case, these deconnections are limited in time and we speak of a "bounded temporal diameter" within the group. The fact of limiting the temporal diameter of the group enables it

to distinguish between temporary deconnections and final loss (crash or other) of some nodes.

Keywords Wireless networks, dynamic networks, time-varying graphs, delay-tolerant networks, temporal diameter, delay-tolerant groups

Mots-clés Réseau sans-fils, réseaux dynamiques, graphes dynamiques, réseaux tolérant aux délais, diamètre temporel, groupe tolérant aux délais

Laboratoire d'accueil LaBRI, Bâtiment A30, 351 cours de la Libération, F-33405 Talence Cedex

Remerciements

Avant tout chose, j'ai un certain nombre de personnes que je souhaite remercier pour leur soutien tout au long de cette thèse.

Je tiens à remercier Arnaud Casteigts, Serge Chaumette et Colette Johnen pour m'avoir encadré tout au long de ma thèse et pour toutes les discussions enrichissantes que l'on a pu avoir, que ce soit sur la thèse ou sur tous les autres sujets que l'on a pu aborder.

Je tiens ensuite à remercier la DGA et la région Aquitaine pour avoir accepté de financer cette thèse.

Je tiens aussi à remercier Hacène Fouchal et Phillippe Roose pour avoir accepté d'être raporteur de ma thèse et membre de mon jury de thèse. Je tiens également à les remercier pour leurs remarques pertinentes quant à ce manuscrit qui m'ont permis de l'améliorer.

Je tiens à remercier Mohamed Mosbah pour avoir présidé le jury de ma soutenance de thèse. Je remercie également Véronique Serfaty et Stéphane Devismes pour avoir été membre de mon jury et pour leurs questions qui m'ont permis d'envisager de nouvelles perspective à ces travaux.

Je remercie mes collègues de Bureau (Vincent Autefage, Vincent Klein, Martin Rosalie, Yessin Mohamed Neggaz, Christelle Al Hasrouty, Ema Falomir, Jigar Solanki, Sebastien Bindel, Jonathan Saludas, Ahmat-mahamat Daouda et tous ceux dont j'aurais pu oublier de citer le nom) pour la bonne humeur qu'il y a eu au sein de notre bureau et pour toutes les discusions très fructueuses que l'on a pu avoir au cours de ces trois années.

Je remercie enfin ma famille, mes parents, mon frère, mes oncles et mes tantes ainsi que mes grands-parents, qui m'ont poussé et soutenu tout au long de mes études et qui m'ont permis d'arriver là ou je suis aujourd'hui. Certains d'entre eux nous ont quitté avant la fin de cette thèse, mais leurs encouragements sont toujours restés dans mon cœur.

Table des matières

Introduction

Parmi les évolutions majeures de l'informatique, nous distinguons l'émergence des technologies mobiles sans fil. Le développement actuel de ces technologies permet de réaliser des communications *ad-hoc* directes entre de nombreux types d'entités mobiles, comme des véhicules, des smartphones, des robots terrestres, des drones ou des satellites. Dans un réseau de tels équipements, l'ensemble des liens de communication qui existe à un instant donné dépend des distances entre les entités et la topologie du réseau change continuellement lorsque les entités se déplacent. Non seulement les changements sont fréquents, mais en général ils partitionnent le réseau. Les hypothèses habituelles sur la connexité du réseau n'ont donc pas leur place ici, néanmoins, une autre forme de connexité appelée connexité temporelle est souvent disponible à travers le temps et l'espace. Enfin, dans les scénarios où la dynamique est la norme plus que l'exception, la vision classique des réseaux où la dynamique correspond à des *fautes* n'est pas adaptée.

Ces réseaux sont des systèmes distribués, qui peuvent être représentées de différentes manières. En ce qui concerne les communications, nous représentons le réseau à l'aide de graphes, à savoir que les entités (ou noeuds) sont représentées par des sommets et les possibilités de communication entre elles sont représentées par des arêtes. Comme évoqué, les systèmes que nous étudions sont dynamiques et par conséquent, le graphe qui les représente l'est aussi. Cela induit des changements de paradigme, en particulier le traitement des apparition et disparitions d'arêtes ainsi que de la connexité, qui impactent fortement les algorithmes s'exécutant sur de tels réseaux. Cela impacte également la faculté des appareils à détecter les pannes des autres appareils, car ils peuvent pendant de longues durées ne plus communiquer mais tout de même faire les tâches qui leur ont été assignées.

L'objectif de cette thèse a été de développer des algorithmes pour les flottes d'appareils dans le cas des réseaux tolérant aux délais (DTN). De manière simplifiée, les réseaux tolérants aux délais sont des réseaux pour lesquels certaines parties peuvent se retrouver isolées pendant un moment sans que cela pose problème. Nous nous intéressons, en particulier, au cas où ces appareils sont organisés sous la forme de groupes, et où la notion de groupe elle même survit à ces déconnexions transitoires. Ainsi,une grande partie de la thèse s'articule

autour de la notion des *groupes tolérant aux délais* (groupe DTN). Dans notre cas cet éloignement est limité dans le temps et nous parlons alors de « diamètre temporel borné » au sein du groupe. Le fait de borner le diamètre temporel du groupe lui permet de distinguer entre l'éloignement temporaire d'un noeud et sa perte définitive (crash ou autre).

À l'exception d'un chapitre, cette thèse s'intéresse à différents aspects de cette notion de groupe DTN, p.ex. test du diamètre temporel dans un groupe donné, exploration de graphes par des agents qui forment un groupe DTN, et développement d'un module d'obstacle qui permet de simuler des groupes dont la communication est temporairement interrompue. Les contributions de cette thèse sont organisées comme suit.

Dans un premier temps, dans le chapitre 1, nous faisons le lien entre les systèmes distribués et les graphes dynamiques. Nous introduisons ensuite les différentes notations et concepts relatifs aux graphes statiques et aux graphes dynamiques. En particulier, les différents modèles de graphes dynamiques : les graphes variant dans le temps et les graphes évolutifs. Nous introduisons formellement le concept des groupes tolérant aux délais (DTN) dans les graphes dynamiques car ces groupes sont au cœur de nos travaux. Nous présentons aussi certains problèmes centraux dans les systèmes distribués. Enfin, nous présentons certaines classes de graphes dynamiques qui correspondent à diverses hypothèses sur la dynamique du réseaux.

Nous présentons ensuite dans le chapitre 2 une méthode pour tester si la contrainte du diamètre temporel est respectée au cours du temps. Nous ramenons ce problème à celui de tester l'appartenance d'un graphe dynamique donné à une classe particulière : la classe des graphes temporellement connexes. Les graphes de cette classe ont comme garantie que tous les nœuds peuvent communiquer les uns avec les autres à l'aide de trajets (chemins temporels) au moins une fois. Pour savoir si un graphe est temporellement connexe ou pas nous utilisons un outil qui s'appelle la fermeture transitive des trajets. Nous présentons deux algorithmes existants que nous avons adaptés au calcul de la fermeture transitive des trajets et nous présentons aussi un algorithme dédié au calcul de cette fermeture transitive pour un graphe dynamique donné.

Nous nous sommes ensuite intéressés dans le chapitre 3 à la problématique de l'exploration de bâtiment par un groupe d'appareils. L'exploration d'un bâtiment consiste à visiter l'ensemble de ses pièces. Nous avons considéré une version abstraite du problème, où les bâtiments sont représentés par des graphes dont les sommets représentent les salles et les arêtes représentent les portes en ces salles. Les appareils sont représentés par des agents mobiles qui se promènent dans le graphe. Le problème de l'exploration de graphes par un groupe d'agents est un problème difficile (au sens de la complexité algorithmique). Dans notre cas nous nous sommes intéressés à l'exploration d'arbres car la plupart des bâtiments ont une structure de salle quasi-arborescente et il

se trouve que le problème est déjà difficile dans le cas des arbres. En revanche, nous nous sommes intéressé à contraindre les mouvements des agents de sorte à satisfaire un diamètre temporel borné par une valeur donnée (groupe DTN). Nous introduisons plusieurs algorithmes *offline*. Parmi eux, nous proposons un algorithme qui trouve la solution optimale, mais son temps de calcul est exponentiel. Nous définissons aussi un algorithme heuristique d'exploration *online* et nous présentons ensuite les résultats de simulation correspondant. Nous étudions enfin l'impact de différents facteurs tels que la valeur du diamètre temporel, le nombre d'agents et le degré maximal de l'arbre.

Moins lié à la notion de groupe DTN, mais considérant toujours le contexte des *réseaux DTN*, nous avons étudié dans le chapitre 4 le problème d'organiser les différentes composantes connexes qui résultent de la forte dynamicité du réseau. Pour cela nous avons voulu regrouper, au sein d'un même groupe, tous les appareils qui sont dans la même composante connexe, tout en permettant à ce groupe de s'organiser à l'aide d'un arbre couvrant. Il s'agit donc de maintenir une forêt d'arbres couvrants en garantissant toujours certaines propriétés, et ce, de manière distribuée (dans cette thèse nous utilisons de manière interchangeable les termes *distribué* et *online*). Ainsi, lorsque deux composantes fusionnent, leurs arbres doivent fusionner, et lorsqu'une composante éclate en plusieurs, autant d'arbres doivent être régénérés. Les propriétés garanties sont qu'à tout moment chaque nœud appartient à un unique arbre ; chaque arbre a exactement une racine qui se déplace dans l'arbre (qui peut donc être vue comme un leader mobile au sein du groupe) ; et il n'y a jamais de cycle.

Enfin, dans le chapitre 5, nous présentons une extension que nous avons développée à l'outil JBotSim permettant de mettre en place des obstacles qui coupent les communications entre les différents appareils et qui peuvent aussi impacter les mouvements des appareils. La motivation pour cette extension est de permettre d'étudier, dans les travaux futures, des problèmes liés aux groupes DTNs où la communication entre appareils peut être interrompue temporairement à cause d'obstacles. Par exemple, dans le cas du scénario d'exploration évoqué précédemment, il pourrait s'agir des murs entre les salles des batiments à explorer. Cette extension est accessible publiquement sous la forme d'une bibliothèque java.

Les chapitres de cette thèse sont organisés dans l'ordre qui nous a semblé le plus naturel et il est donc recommandé de les lires dans l'ordre d'apparition.

Chapitre 1

Définitions sur les graphes et l'algorithmique distribuée

Sommaire

Dans ce chapitre nous présentons notre vision des systèmes distribués. Nous rappelons notamment les définitions et les concepts relatifs aux graphes statiques. Nous présentons ensuite quelques formalismes de graphes dynamiques,

utilisés pour modéliser les topologies de réseaux considérées, ainsi que les différents concepts associés. Nous discutons de la manière dont cette dynamique impacte les problèmes relatifs aux systèmes distribués. Enfin, nous présentons plusieurs classes de graphes dynamiques, dont certaines sont étudiées plus précisément dans cette thèse.

1.1 Systèmes distribués

Dans le cadre de cette thèse, nous nous sommes intéressés aux systèmes distribués car, comme nous le verrons dans la section 1.1.1, ils sont la représentation d'un grand nombre de scénarios réels (réseaux de robots, de drones, de véhicules *etc.*).

1.1.1 Définitions

Dans le choix qui nous intéresse, un système distribué est constitué d'un ensemble d'entités autonomes. Ces entités peuvent être des ordinateurs, des téléphones portables, des satellites, des véhicules, des drones. Dans ce mémoire, ces entités seront appelées *nœuds*. Chacune de ces entités a des facultés de communication. Les communications se font sur des liens de communication identifiés par un couple ou une paire de nœuds. Ces liens de communication seront appelés *arêtes* lorsque les communications sont possibles dans les deux sens et *arcs* lorsque les communications ne sont possibles que dans un seul sens. Les nœuds travaillent de façon collaborative en effectuant des calculs internes et en s'échangeant des données sur les arêtes, le tout afin de résoudre un problème/une tâche donnée. Deux nœuds sont voisins lorsqu'il existe une arête les reliant. Un système distribué est alors représenté par son ensemble de nœuds et par sa topologie (son réseau de communication).

Les systèmes distribués permettent de représenter toutes sortes de systèmes et de réseaux. Ils peuvent servir de représentation pour les clusters de serveurs, les réseaux sociaux, les réseaux véhiculaires, les réseaux de smartphones, de capteurs et de drones.

Les avantages des systèmes distribués sont multiples :

— **partage de ressource** : Chaque nœud a un ensemble de ressources (mémoires, puissance de calcul, services). Dans un tel système, chaque nœud met ses ressources à disposition des autres nœuds. Un nœud peut alors faire appel à un service d'un autre nœud spécialisé et un problème peut être réparti entre différents nœuds pour faciliter et/ou accélérer sa résolution.

— **disponibilité** : Dans la vision classique d'un système où tous les calculs/services sont localisés sur le même processeur, lorsque ce processeur

tombe en panne, l'ensemble des services qu'il fournit devient indisponible. Dans un système distribué les calculs/services sont répartis sur différents nœuds et lorsqu'un nœud tombe en panne seuls certains services deviennent indisponibles, ou fonctionnent sous une forme dégradée. Ces systèmes permettent d'atteindre une certaine sûreté de fonctionnement.

— **extensibilité** : Contrairement à un système centralisé où l'évolution du système passe généralement par un changement de matériel, un système distribué peut être facilement étendu en y ajoutant où en y supprimant des nœuds. De plus un système distribué est généralement constitué de nœuds n'ayant pas forcément les même capacité ce qui implique que l'ajout de nouveaux nœuds au système n'est pas conditionné par le type de nœud.

1.1.2 Types de topologie

Les systèmes distribués peuvent être caractérisés par le type de leurs topologies. Une topologie peut être :

— **statique** : le graphe de communication est fixe, les arêtes entre les nœuds restent présentes en permanence.

— **dynamique** : le graphe de communication varie dans le temps, les nœuds et les arêtes entre les nœuds peuvent apparaître et disparaître au cours du temps.

Les systèmes distribués ayant une topologie statique ont un réseau de communication fixe. La perte d'un nœud ou d'une arête au sein de la topologie est alors considérée comme une panne/faute. Pour les systèmes ayant une topologie dynamique deux visions existent selon la fréquence des changements de la topologie. Dans le cadre d'une topologie faiblement dynamique, la plupart des algorithmes proposés sont des algorithmes auto-stabilisant et pour eux la perte d'un nœud reste une panne/faute et les algorithmes ont le temps de se stabiliser. Dans le cadre d'une topologie fortement dynamique, la perte d'un nœud ou d'une arête est considéré comme étant normale. Il devient par contre beaucoup plus difficile de détecter si le nœud s'est juste déconnecté ou s'il est tombé en panne. Les algorithmes doivent alors prendre en compte ce problème pour pouvoir fonctionner correctement.

Dans le cadre de ce mémoire, nous nous concentrons sur les systèmes distribués ayant une topologie fortement dynamique.

1.1.3 Modèles algorithmiques

Dans les systèmes distribués, deux visions algorithmiques coexistent :

— la vision **offline** : les calculs sont effectués par une entité extérieure au système distribué, entité qui peut avoir accès à l'ensemble des informations du système comme par exemple sa topologie. On parle aussi d'algorithme centralisé.

— la vision **online** : les calculs sont effectués localement par chaque nœud du système distribué. Généralement, il est supposé que les nœuds n'ont pas accès aux informations générales du système distribué. On parle alors d'algorithme distribué.

Dans ce mémoire nous proposerons des algorithmes *offlines* et *onlines* selon les problèmes étudiés.

1.1.4 Modèles d'activation des nœuds

Dans le cadre des algorithmes distribués l'activation des nœuds peut être réalisée de plusieurs façons.

— **synchrone** : dans ce cas, il existe une horloge globale avec laquelle tous les nœuds sont synchronisés. Les nœuds s'activent tous au même moment et font leurs calculs en même temps. Dans ce document chaque phase de calcul sera appelée *tour*.

— **semi-synchrone** : dans ce cas, il existe aussi une horloge globale avec laquelle tous les nœuds sont synchronisés mais seule une partie des nœuds s'active à chaque tour. Le choix des nœuds qui s'activent peut alors être décidé soit par un adversaire, soit de manière aléatoire, soit d'une autre façon.

— **asynchrone** : dans ce cas, les nœuds s'activent individuellement et à n'importe quel moment. Les nœuds peuvent rester arbitrairement longtemps sans s'activer. Le choix de l'activation peut par exemple être fait par un adversaire omniscient qui cherche à malmener l'algorithme qui s'exécute sur les nœuds.

Les algorithmes doivent être adaptés en fonction du modèle d'activation des nœuds. Par exemple, dans le modèle synchrone il est généralement considéré que les communications sont réciproques (un nœud et son voisin peuvent communiquer dans les deux sens) alors que dans le modèle asynchrone il n'est pas aussi évident de faire ce genre de supposition. Les algorithmes d'exploration (Chapitre 3) et de forêt couvrante (Chapitre 4) proposés dans cette thèse supposent un environnement synchrone.

1.1.5 Modèles de communication

Au sein des systèmes distribués différents modèles de communication existent selon le type du système étudié :

— **boîtes aux lettres** : Dans ce modèle, chaque nœud possède une zone mémoire dans laquelle ses voisins peuvent écrire des données. Si un nœud u veut communiquer avec un nœud v il doit écrire dans la zone mémoire de v et la réception de données par un nœud se fait en lisant cette zone.

— **registres** : Dans ce modèle, chaque arête se voit attribuer deux registres, un de chaque côté de l'arête. Pour communiquer avec un voisin un nœud écrit dans le registre situé de son côté de l'arête correspondant au nœud avec lequel il souhaite communiquer. Chaque nœud peut lire et écrire sur ses propres registres et lire les registres de ses voisins.

— **mémoires partagées** : Dans ce modèle, les nœuds qui souhaitent communiquer possèdent une mémoire commune. L'accès a cette mémoire est exclusif en écriture mais la lecture peut être concurrente.

— **passage de messages** : Dans ce modèle, les nœuds sont reliés par des canaux de communication et communiquent en s'échangeant des messages. Les canaux peuvent être unidirectionnels ou bidirectionnels.

Dans le chapitre 4, nous considérons le modèle à base de passage de messages pour des raisons de réalisme. En effet, le principe de l'algorithme que nous y étudions existait dans des modèles plus théoriques, mais qui ne peuvent pas être implémentés dans la vie réelle. Notre contribution est de réussir à transposer ce principe dans le modèle à base de passage de message.

1.2 Graphes statiques

Traditionnellement, nous utilisons le modèle des graphes statiques pour représenter les systèmes distribués ayant une topologie statique. De nouveaux modèles basés sur les graphes dynamiques sont maintenant utilisés pour les systèmes distribués ayant un topologie dynamique. Avant de définir les graphes dynamiques nous présentons les notions de base sur les graphes statiques.

Définition 1.1. Un **graphe orienté** est un couple $G = (V, E)$ composé d'un ensemble de **nœuds** V et d'un ensemble d'**arcs** $E \subseteq V \times V$ (E est un ensemble de couples de nœuds). Deux nœuds u et v sont les **extrémités** d'un arc si et seulement si $(u, v) \in E$, u est la source de l'arc et v est la destination. S'il existe un arc de u vers v il alors u peut communiquer avec v mais pas forcément l'inverse.

Définition 1.2. Un **graphe non-orienté** est un couple $G = (V, E)$ composé d'un ensemble de **nœuds** V et d'un ensemble d'**arêtes** $E \subseteq V \times V$ (E est un ensemble de paires de nœuds). Deux nœuds u et v sont les **extrémités** d'une arête si et seulement si $\{u, v\} \in E$.

Une **boucle** est une arête dont les deux extrémités sont identiques.
Les **arêtes multiples** sont plusieurs arêtes ayant les mêmes extrémités.
Un **graphe simple non-orienté** est un graphe simple n'ayant pas de boucles et pas d'arêtes multiples.

Dans la suite, le terme **graphe statique** (ou graphe) sera utilisé pour désigner un graphe simple non-orienté.

Définition 1.3. Soit deux graphes statiques $G = (V, E)$ et $G' = (V', E')$, G' est un **sous-graphe** de G si $V' \subseteq V$ et si pour toute arête $e \in E'$, on a $e \in V' \times V'$ et $e \in E$.

Définition 1.4. Soit deux graphes statiques $G = (V, E)$ et $G' = (V', E')$, G' est un **sous-graphe induit** de G si $V' \subseteq V$ et pour toute arête $e \in E$ si $e \in V' \times V'$ alors $e \in E'$.

Un sous-graphe induit d'un graphe statique est donc un sous-ensemble de nœuds de ce graphe statique ainsi que toutes les arêtes qui existent entre ces nœuds.

Définition 1.5. Soit $G = (V, E)$ un graphe statique orienté. On dit que deux arcs distincts e et e' sont **adjacents** si la destination de e est identique à la source de e'.
On appelle **chemin** entre deux nœuds u et v une suite d'arcs adjacente démarrant de u et arrivant sur v ; u est la **source** du chemin et v en est la **destination**.
La **longueur** d'un chemin est le nombre d'arcs le composant.

Définition 1.6. Soit $G = (V, E)$ un graphe statique non-orienté. On dit que deux arêtes distinctes e et e' sont **adjacentes** si elles partagent une extrémité en commun.
On appelle **chemin** entre deux nœuds u et v une suite d'arêtes adjacentes démarrant de u et arrivant sur v ; u est la **source** du chemin et v en est la **destination**.
La **longueur** d'un chemin est le nombre d'arêtes le composant.

Définition 1.7. Soit un graphe $G = (V, E)$, la **distance** entre deux nœuds u et v est la longueur du plus court chemin qui lie u à v dans le graphe G.

La notion de distance entre deux nœuds donne le nombre d'arêtes minimal à traverser pour passer d'un de ces nœuds à l'autre.

Définition 1.8. Soit un graphe $G = (V, E)$, le **diamètre** du graphe G est la plus grande distance entre toutes les paires de nœuds de V.

Définition 1.9. Un graphe $G = (V, E)$ est dit **complet** si et seulement si pour toutes les paires de nœuds u et v de V il existe une arête $e \in E$ ayant pour extrémité u et v.

Dans un graphe complet chaque nœud du graphe est directement connecté avec tous les autres nœuds.

Définition 1.10. Soit un graphe $G = (V, E)$, deux nœuds u et v de V sont dits **connectés** si et seulement s'il existe un chemin reliant u à v dans le graphe.
Un graphe est **connexe** si quels que soient les nœuds u et v de V, u et v sont connectés.
Une **composante connexe** est un sous ensemble de nœuds $V' \subseteq V$ tels que quelque soient les nœuds u et v de V', u et v sont connectés.

Un graphe peut être partitionné en plusieurs composantes connexes. Deux nœuds de différentes composantes connexes ne pourront jamais communiquer entre eux.

Définition 1.11. Soit un graphe $G = (V, E)$, un **cycle** est un chemin dont la source et la destination sont le même nœud.
Un graphe **acyclique** est un graphe sans cycle.
Un **arbre** est un graphe acyclique connexe.
Une **forêt** est un graphe dont toutes les composantes connexes sont des arbres.
Un **arbre couvrant** d'un graphe $G = (V, E)$ est un arbre qui contient tous les nœuds de V et un sous-ensemble $E' \in E$ d'arêtes.

Les graphes statiques de part la non-évolution des connexions entre les nœuds permet de représenter les systèmes distribués ayant une topologie statique qui n'évolue pas.

1.3 Graphes dynamiques

Nous allons maintenant nous intéresser aux systèmes distribués ayant des topologies dynamiques et que l'on peut représenter à l'aide des graphes dynamiques. Cette section introduit les modèles, concepts et notations propres à ce type de graphes.

Nous commençons avec le modèle des graphes dynamiques le plus expressif, il s'agit des graphes variants dans le temps (time-varying graphs) introduits par Casteigts, Flocchini, Quattrociocchi et Santoro dans l'article [27]. Ce modèle et les concepts spécifiques aux graphes dynamiques sont introduits dans les sous-sections suivantes. En particulier, nous introduisons la notion de groupe DTN qui est centrale dans cette thèse. Nous présentons aussi le modèle des graphes évolutifs introduit par Ferreira dans [35], qui sera utilisé dans le Chapitre 2. Les notations utilisées ci-dessous proviennent des différents articles présentés.

1.3.1 Graphes variant dans le temps (TVG)

Nous allons tout d'abord commencer par définir les graphes variant dans le temps avant d'enchaîner avec les concepts relatifs aux graphes dynamiques.

Définition 1.12. Soit un ensemble de nœuds V et un ensemble d'arêtes E entre ces nœuds, $E \subseteq V \times V$. Les relations entre les nœuds prennent place dans l'espace de temps $\mathcal{T} \subseteq \mathbb{T}$ appelé temps de vie du système. Le domaine temporel $\mathbb{T}$ peut être soit continu ($\mathbb{R}^+$), soit discret ($\mathbb{N}$). La dynamique du système est décrite par un tuple, appelé **graphe variant dans le temps**, ou time-varying graph (**TVG**), $\mathcal{G} = (V, E, \mathcal{T}, \rho, \zeta)$, où :

— $\rho : E \times \mathcal{T} \to \{0, 1\}$ est appelé fonction de *présence* et indique si l'arête donnée est présente au temps donné.

— $\zeta : E \times \mathcal{T} \to \mathbb{T}$ est appelé fonction de *latence* et indique le temps qu'il faut pour traverser l'arête donnée au temps donné.

Ce modèle peut être étendu en ajoutant une fonction de présence des nœuds $\psi : V \times \mathcal{T} \to \{0, 1\}$, les nœuds pouvant n'être présents que pendant certaines périodes. Il est aussi possible de rajouter une fonction de latence des nœuds $\varphi : V \times \mathcal{T} \to \mathbb{T}$ qui permet de représenter des temps de traitement sur les nœuds (par exemple la latence d'un routeur). Dans notre cas, nous considérerons le modèle initial sans ces ajouts. Dans le reste du document, $\mathcal{T}^-$ et $\mathcal{T}^+$ représenteront respectivement la première et la dernière date de $\mathcal{T}$.

Définition 1.13. Un sous-graphe $\mathcal{G}'$ d'un graphe dynamique $\mathcal{G}$ est obtenu en restreignant soit l'ensemble des nœuds $V' \subseteq V$, soit l'ensemble des arêtes $E' \subseteq E$, soit la plage de temps $\mathcal{T}' \subseteq \mathcal{T}$, il en résulte le tuple $(V', E', \mathcal{T}', \rho', \zeta')$ tel que :

— (V', E') est le sous-graphe de (V, E) induit (dans le sens statique du terme) par V'

— $\rho' : E' \times \mathcal{T}' \to \{0, 1\}$ où $\rho'(e, t) = \rho(e, t)$

— $\zeta' : E' \times \mathcal{T}' \to \mathbb{T}$ où $\zeta'(e, t) = \zeta(e, t)$

Si seule la plage de temps est restreinte sur un intervalle $\mathcal{I}$, alors le sous-graphe résultant est appelé sous-graphe temporel de $\mathcal{G}$ et est noté $\mathcal{G}_\mathcal{I}$.

Définition 1.14. Soit $\mathcal{G} = (V, E, \mathcal{T}, \rho, \zeta)$ un TVG, on appelle **graphe sous-jacent** de $\mathcal{G}$ le graphe statique (V, E).

Le graphe sous-jacent est donc le graphe représentant toutes les arêtes du graphe dynamique y compris si elles n'apparaissent jamais.

Définition 1.15. Soit $\mathcal{G} = (V, E, \mathcal{T}, \rho, \zeta)$ un TVG, on appelle **empreinte** de $\mathcal{G}$ le graphe statique (V, E') tel que $\forall e \in E$, on a $e \in E'$ si et seulement si $\exists t \in \mathcal{T}$ tel que $\rho(e, t) = 1$.

L'empreinte représente l'ensemble des arêtes qui existent à un moment ou un autre dans le graphe dynamique. Elle permet d'obtenir un certain nombre d'informations sur le graphe dynamique comme par exemple si les noeuds ont pu avoir la possibilité de tous communiquer les uns avec les autres lorsque l'empreinte est connexe. Néanmoins, cela ne garantit pas que cela est pu vraiment avoir lieu du fait de la notion de temps. Nous considérons dans la suite que chaque arête de l'ensemble E apparaît au moins une fois à un instant t. Dans notre cas le graphe sous-jacent et l'empreinte sont donc équivalents.

1.3.2 Concepts associés aux graphes dynamiques

Les notions introduites sur les graphes statiques nécessitent d'être adaptées aux graphes dynamiques. Nous présentons dans cette section un ensemble de définitions relatives aux graphes dynamiques.

1.3.2.1 Connexité

Dans le cadre des graphes dynamiques les notions de connexité et de chemin nécessitent d'être adaptées. On ne peut plus vraiment parler de chemin entre deux nœuds mais plutôt de chemin temporel ou trajet.

Définition 1.16. Soit $\mathcal{G} = (V, E, \mathcal{T}, \rho, \zeta)$ un TVG, un **trajet** (ou chemin temporel) est une séquence de couples $\mathcal{J} = \{(e_1, t_1), (e_2, t_2), \cdots, (e_k, t_k)\}$ telle que $\{e_1, e_2, \cdots, e_k\} \subseteq E$, telle que $\{t_1, t_2, \cdots, t_k\} \subseteq \mathcal{T}$ et telle que pour tout $1 \leqslant i \leqslant k$, $\rho(e_i, t_i) = 1$ et pour tout $1 \leqslant i < k$, $t_{i+1} \geqslant t_i + \zeta(e_i, t_i)$. Dans le cas où $\zeta(e_i, t_i) \neq 0$ il est nécessaire que $\forall t \in [t_i, t_i + \zeta(e_i, t_i)[$, $\rho(e_i, t) = 1$.

Dans le cadre d'un trajet $\mathcal{J}$ les notations $departure(\mathcal{J})$ et $arrival(\mathcal{J})$ représentent respectivement la date de départ t_1 et la date d'arrivée $t_k + \zeta(e_k, t_k)$ du trajet. La durée d'un trajet est la différence entre sa date d'arrivée et sa date de départ. Dans la suite de ce mémoire, $\mathcal{J}^*_{\mathcal{G}}$ représentera l'ensemble de tous les trajets du graphe $\mathcal{G}$ et $\mathcal{J}^*_{(u,v)} \subseteq \mathcal{J}^*_{\mathcal{G}}$ l'ensemble des trajets partant de u et terminant sur v. Dans la suite la notation $|\mathcal{J}|$ représentera le nombre d'arêtes traversées par le trajet $\mathcal{J}$. L'existence d'un trajet de u vers v est noté $u \rightsquigarrow v$.

La notion de connexité est alors être adaptée de la façon qui suivante :

Définition 1.17. Soit $\mathcal{G} = (V, E, \mathcal{T}, \rho, \zeta)$ un TVG, $\mathcal{G}$ est **temporellement connexe** si et seulement si pour tout couple de nœuds $(u, v) \in V \times V$ il existe un trajet de u vers v dans $\mathcal{G}$.

La connexité temporelle implique certaines contraintes sur le graphe dynamique. À partir de l'empreinte d'un graphe dynamique il est possible de dire

qu'un graphe dynamique n'est pas temporellement connexe lorsque l'empreinte n'est elle-même pas connexe. Dans ce cas, il existe un ensemble de nœuds dans le graphe dynamique qui n'aura jamais la possibilité de communiquer avec le reste des nœuds. Il n'est en revanche pas suffisant que l'empreinte soit connexe pour garantir que $\mathcal{G}$ soit temporellement connexe.

1.3.2.2 Distances et diamètre

En plus de la notion de trajet qui étend la notion de chemin, la notion de distance a été adaptée pour tenir compte de la notion de temps. La longueur des trajets en terme du nombre d'arêtes traversées ou en terme de temps peut varier selon la date de départ des trajets. Les définitions ci-dessous sont issues de [27]. Les concepts associés avaient fait l'objet d'un premier traitement dans [54].

Définition 1.18. Soit $\mathcal{G} = (V, E, \mathcal{T}, \rho, \zeta)$ un TVG, la **distance topologique** d'un nœud u à un nœud v à un instant t, notée $d_{u,t}(v)$, est définie comme $\min\{|\mathcal{J}| : \mathcal{J} \in \mathcal{J}^*_{(u,v)}, departure(\mathcal{J}) \geqslant t\}$.

La distance topologique entre deux nœuds u et v à l'instant t représente le nombre minimal d'arêtes traversées pour aller de u à v en utilisant un trajet dont la date de départ est supérieure ou égale à t.

Définition 1.19. Soit $\mathcal{G} = (V, E, \mathcal{T}, \rho, \zeta)$ un TVG, la **distance temporelle** d'un nœud u à un nœud v à un instant t, notée $\hat{d}_{u,t}(v)$, est définie comme $\min\{arrival(\mathcal{J}) : \mathcal{J} \in \mathcal{J}^*_{(u,v)}, departure(\mathcal{J}) \geqslant t\} - t$.

La distance temporelle entre deux nœuds u et v à l'instant t représente le temps minimal pour aller de u à v en utilisant un trajet dont la date de départ est supérieure ou égal à t.

Nous pouvons maintenant définir quelques notions s'appuyant sur la distance temporelle.

Définition 1.20. Soit $\mathcal{G} = (V, E, \mathcal{T}, \rho, \zeta)$ un TVG, l'**excentricité temporelle** d'un nœud $u \in V$ à un instant t, notée $\hat{\varepsilon}_t(u)$, est définie comme étant le $\max\{\hat{d}_{u,t}(v) : v \in V\}$.

L'excentricité temporelle d'un nœud u à l'instant t est le temps minimal qu'il faut au nœud u pour envoyer un message à tous les autre nœuds du graphe dynamique à l'instant t.

Nous introduisons maintenant le concept de diamètre temporel à l'aide de la définition suivante.

Définition 1.21. Soit $\mathcal{G} = (V, E, \mathcal{T}, \rho, \zeta)$ un TVG, le **diamètre temporel** d'un graphe $\mathcal{G}$ à un instant t, notée $\mathrm{diamT}_t(\mathcal{G})$, est défini comme étant le $\max\{\hat{\varepsilon}_t(u) : u \in V\}$.

Le diamètre temporel d'un graphe à un instant t est le temps minimum qu'il faut pour que chaque nœud diffuse un message à tous les autres nœuds.

1.3.3 Graphes évolutifs

Une autre manière de voir les graphes dynamiques est de les considérer comme une suite de graphes statiques. Les graphes évolutifs sont un modèle des graphes dynamiques basés sur une séquence de graphes statiques. L'idée de représenter un graphe dynamique comme une séquence de graphes statiques a été évoqué pour la première fois par Harary et Gupta dans [40]. Ce modèle a ensuite été formellement défini en 2004 par Ferreira dans l'article [35]. Il existe deux versions du modèle et nous les présentons ci-dessous.

1.3.3.1 Graphes évolutifs temporisés

Définition 1.22. Soient $G = (V, E)$ un graphe statique et $\mathcal{S}_G = \{G_1, \cdots, G_\delta\}$ une séquence ordonnée de sous-graphes $G_i(V, E_i)$ de G tel que $\cup_{i=1}^{\delta} G_i = G$. Soit $\mathcal{S}_T = \{t_0, t_1, \cdots, t_\delta\}$ une séquence de dates ordonnées. Le tuple $\mathcal{G} = (G, \mathcal{S}_G, \mathcal{S}_T)$, où chaque G_i est en place durant l'intervalle de temps $[t_{i-1}, t_i[$, est appelé **graphe évolutif temporisé**. Le graphe G est appelé **graphe sous-jacent**.

L'usage de dates permet de faire varier le temps de vie de chaque sous-graphe du graphe dynamique. Toutefois lorsque la durée de vie de chaque sous graphe est constante, le modèle des graphes dynamiques non-temporisés est plus facile à manipuler.

D'un point de vue mathématique, les graphes évolutifs sont presque équivalents aux graphes variants dans le temps. La seule différence est que les événements topologiques doivent être dénombrables (à cause de la représentation en séquence), alors que cela peut être arbitraire avec les graphes variant dans le temps. Cela conduit à de fortes différences dans l'expressivité des deux modèles et dans la puissance potentielle d'un adversaire dans les réseaux distribués ([22]). Du point de vue de l'utilisation, la différence principale est dans la commodité du formalisme (notations) qui fait que les graphes évolutifs sont souvent utilisés dans les cas discrets, alors que les TVGs sont souvent utilisés pour décrire les algorithmes distribués dans le cas continu (ou, comme nous le verrons, pour exprimer des propriétés générales sur la dynamique).

1.3.3.2 Graphes évolutifs non-temporisés

Définition 1.23. Soient $G = (V, E)$ un graphe statique et $\mathcal{S}_G = \{G_1, \cdots, G_\delta\}$ une séquence ordonnée de sous-graphes $G_i(V, E_i)$ de G tel que $\cup_{i=1}^{\delta} G_i = G$. Le tuple $\mathcal{G} = (G, \mathcal{S}_G)$ est appelé **graphe évolutif non-temporisé**. Chaque G_i est présent pendant la même durée.

Dans la suite de cette thèse, le terme **graphe évolutif** désignera un graphe évolutif non-temporisé. Dans cette thèse, soit le modèle des graphes évolutifs soit le modèle des graphes variant dans les temps seront utilisés et cela en fonction des problèmes étudiés.

1.3.3.3 Trajets stricts et trajets non-stricts

Dans le cadre des graphes évolutifs la notion de trajet peut être déclinée selon que l'on autorise ou pas un trajet à traverser plus d'une arête dans chaque sous-graphe. Cette notion a été indroduite dans [19].

Définition 1.24. Soit $\mathcal{G} = (G, \mathcal{S}_G)$ un graphe évolutif. Un **trajet non-strict** $\mathcal{J}$ est un trajet où plus d'une arête peut être traversée dans chaque sous-graphe de $\mathcal{S}_G$. Un **trajet strict** $\mathcal{J}_{st}$ est un trajet où au plus une arête peut être traversée dans chaque sous-graphe de $\mathcal{S}_G$. Pour compléter la notation existante sur les trajets non-stricts, l'existence d'un trajet strict de u vers v est noté $u \overset{st}{\rightsquigarrow} v$.

Dans la suite, lors de l'utilisation du modèle des graphes évolutifs, un trajet sans qualificatif sera un trajet non-strict.

1.4 Composantes connexes et groupes DTN

Dans l'article [14], Bhadra et Ferreira adaptent aux graphes dynamiques le concept des composantes connexes qui existe dans les graphes statiques. Nous le reformulons ici en utilisant le formalisme des TVGs.

Définition 1.25. Soit $\mathcal{G} = (V, E, \mathcal{T}, \rho, \zeta)$. Nous considérons comme étant la variante temporelle des composantes connexes, que l'on appelle *composantes*, les ensembles maximaux de nœuds $V' \subseteq V$ tels que pour tous nœuds $p, q \in V'$, $p \rightsquigarrow q$

Plusieurs variantes de cette définitions peuvent être envisagées, selon que l'on autorise les trajets à utiliser des nœuds en dehors de la composante. On parle alors de composante ouverte ou fermée. Cette définition a ensuite été étendue avec le concept des Δ-composantes.

1.4.1 Δ-composantes

Dans l'article [39], Gómez et al. introduisent de nouvelles propriétés sur les graphes dynamiques basées sur des trajets dont la durée est bornée dans le temps.

Définition 1.26. Soit $\mathcal{G} = (V, E, \mathcal{T}, \rho, \zeta)$ un TVG, un trajet $\mathcal{J}$ est un **Δ-trajet** si et seulement si $arrival(\mathcal{J}) - departure(\mathcal{J}) \leqslant \Delta$.

Les Δ-trajets sont donc l'ensemble des trajets dont la durée est inférieure ou égale à Δ. Les auteurs étendent ensuite la définition de composante connexe à ce type de trajet.

Définition 1.27. Soit $\mathcal{G} = (V, E, \mathcal{T}, \rho, \zeta)$ un TVG, une **Δ-composante connexe** est un sous-ensemble de nœuds $V' \in V$ tel que pour chaque $t \in [\mathcal{T}^-, \mathcal{T}^+ - \Delta]$, pour chaque couple de nœuds $(u, v) \in V' \times V'$, il existe un Δ-trajet de u vers v dans $\mathcal{G}_{[t,t+\Delta[}$.

Autrement dit, au sein de ces Δ-composantes connexes il existe dans toute fenêtre de temps Δ un trajet entre chaque paire de nœuds de la composante. On notera que la date de départ de ces Δ-trajets n'est pas forcément à l'instant t. De la même manière que pour les composantes générales, on peut envisager des Δ-composantes ouvertes ou fermées.

Une autre caractérisation de ces Δ-composantes est que leur diamètre temporel (c.f. définition 1.21) est borné par Δ.

1.4.2 Groupes DTN

Dans cette thèse, nous considèrerons très souvent le concept de groupe *tolérant aux délais* (groupe *DTN*). Cette notion est équivalente à une Δ-composante fermée, c'est à dire que tous les nœuds d'un groupe DTN donné doivent pouvoir se joindre (par des trajets) dans n'importe quelle fenêtre d'une durée donnée (diamètre temporel).

Cette notion est très utile car elle permet aux nœuds du groupe de faire la différence entre un nœud qui s'éloigne temporairement et un nœud qui est tombé en panne.

1.5 Problèmes principaux de l'algorithmique distribuée

Nous allons dans cette section présenter brièvement un certain nombre de problèmes étudiés dans les systèmes distribués et l'impact de la dynamique du réseau sur la définition même de ces problèmes. En particulier, le problème de la maintenance d'arbres couvrants sera considéré dans le Chapitre 4.

1.5.1 Diffusion

Le problème de la diffusion consiste à faire parvenir un message issu d'un nœud unique, que l'on appelle émetteur, à tous les autres nœuds du système. La figure 1.1 représente un exemple de diffusion. Dans le cas des systèmes distribués avec une topologie statique une solution simple est l'inondation. Dans le cadre de l'inondation, dès qu'un nœud reçoit le message que l'on cherche à

diffuser, il l'envoie à son tour à tous ses voisins même s'il l'ont déjà reçu. Cette méthode garantit que le message sera transmis à tous les nœuds en un temps $O(D)$ où D est le diamètre du graphe.

Dans le cadre des systèmes dynamiques, de par l'ajout de la contrainte temporelle sur la présence des arêtes, des optimisations sur les trajets sont apparues. Dans l'article [25] les auteurs proposent trois algorithmes de diffusion basés sur les travaux présentés dans [54] autour des différentes optimisations des trajets :

— La diffusion au plus tôt : le but est ici de sélectionner les trajets dont les dates de réception sont au plus tôt.
— La diffusion au plus rapide : le but est que les trajets de la diffusion soient les plus courts en terme de délais afin de minimiser l'écart entre les dates de départ et les dates d'arrivée des trajets.
— La diffusion au plus court : le but est ici de minimiser la longueur (le nombre d'arêtes utilisées) des trajets de la diffusion.

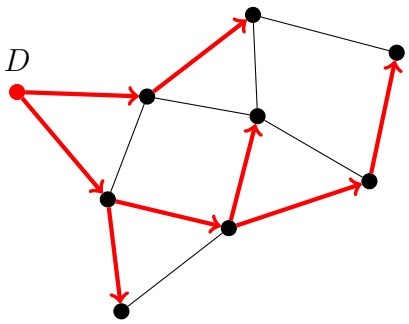


FIGURE 1.1 – Illustration du problème de la diffusion, le nœud D étant l'émetteur et les arcs représentant la diffusion du message.

1.5.2 Exploration

Un bref résumé du problème est présenté ici et dans le chapitre 3 nous présenterons le problème de façon plus approfondie. Pour parler du problème de l'exploration il faut tout d'abord présenter les agents mobiles qui sont des objets très souvent utilisés pour résoudre ce type de problèmes.

Définition 1.28. Un **agent mobile** est un objet capable d'effectuer plusieurs actions, il peut se déplacer sur les arêtes d'un graphe et effectuer des calculs sur le nœud sur lequel il est positionné.

En plus de ces actions, les agents peuvent avoir d'autres capacités :
— Les agents peuvent avoir le droit de modifier l'état du nœud sur lequel ils sont, ils peuvent avoir le droit de déposer un où plusieurs marqueurs (le concept de marqueur sera détaillé dans le chapitre 3).

— Ils peuvent avoir la capacité de communiquer avec les agents sur le même nœud ou avec ceux positionnés à distance 1 dans le graphe, voir même avec tous les agents quelles que soient leurs positions.

Le problème de l'exploration consiste alors à concevoir des algorithmes qui parcourent chaque nœud et/ou arête du graphe de tel sorte que tous les nœuds et/ou arêtes du graphe soient explorés par au moins un agent mobile.

1.5.3 Élection

Le problème de l'élection consiste à distinguer un nœud parmi l'ensemble des nœuds du graphe en le mettant dans l'état élu et tous les autres nœuds doivent être dans l'état non-élu. L'intérêt d'avoir un leader est qu'un certain nombre de tâches sont simplifiées car celui-ci peut exécuter un algorithme complètement différent des autres nœuds. La figure 1.2 représente un exemple d'élection dans les graphes statiques.

Dans le cas des graphes dynamiques le problème nécessite d'être repensé [21]. La dynamique du graphe, qui fait qu'à un instant donné le graphe dynamique n'est pas nécessairement connexe, rend fortement probable le fait que le graphe soit composé de plusieurs composantes connexes à chaque instant. Ces composantes connexes peuvent évoluer à travers le temps en fusionnant, en se séparant, où en changeant des arêtes internes.

Dans le cas où le graphe dynamique est temporelle connexe de manière récurrente, alors avoir un seul leader pour tout le graphe est une option exploitable. Ce leader peut prendre les décisions pour l'ensemble des nœuds et ensuite propager l'information vers tous les autres nœuds même si cela risque de prendre du temps. Cependant, dans le cas où les composantes connexes à travers le temps mettent beaucoup de temps à s'interconnecter, ce concept de leader pose problème, surtout dans les cas où les délais de transmission des informations doivent être courts. Dans ce cas là, une alternative est de maintenir en permanence un leader par composante connexe. Ces différents leaders sont plus à même de prendre les décisions pour l'ensemble des nœuds qui composent leurs composantes connexes. Les figures 1.3(a) et 1.3(b) représentent la différence entre les deux types d'élections distribuées. Ces concepts seront rediscutés dans le chapitre 4 où nous présentons une solution de maintien de forêt couvrante pouvant également servir de maintien de leader.

1.5.4 Arbre couvrant

Dans le cadre des systèmes statiques et donc des graphes statiques, le problème de l'arbre couvrant consiste pour un graphe statique donné $G = (V, E)$ à calculer un arbre $A = (V, E')$ sous-graphe de G contenant tous les nœuds de G et une partie de ses arêtes. Cette structure aide par exemple à faciliter le

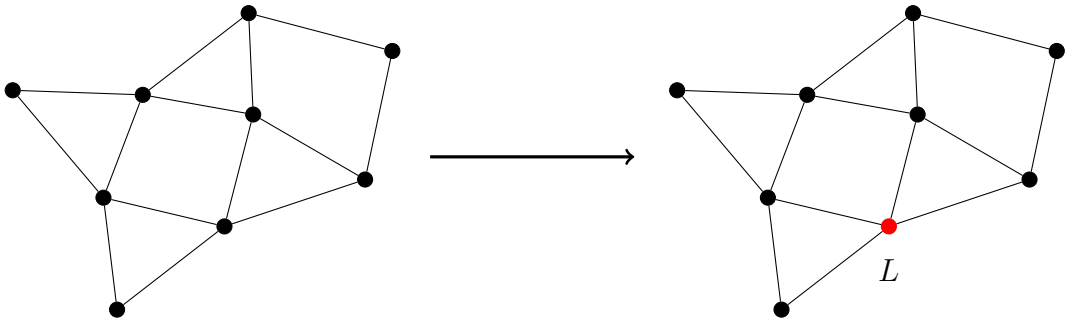
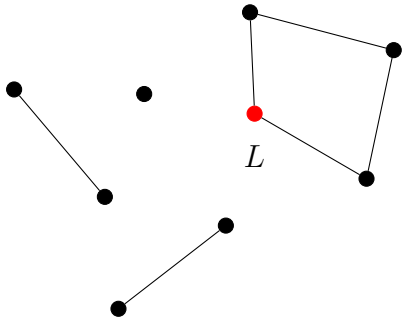
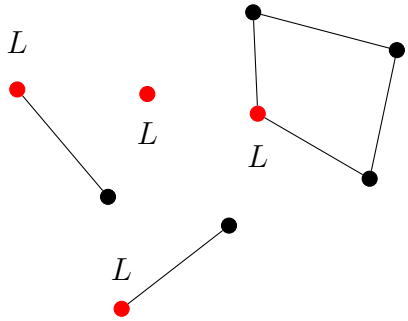


FIGURE 1.2 – Illustration du problème de l'élection, le nœud L étant le leader après la phase d'élection.



(a) Un leader pour l'ensemble du graphe



(b) Un leader par composante connexe du graphe

FIGURE 1.3 – Illustration du problème de l'élection dans le cadre des graphes dynamiques. L représente les nœuds leaders.

routage des messages entre les nœuds. Parfois, les algorithmes de construction d'arbre couvrant utilisent un leader (voir section 1.5.3) pour définir la racine de l'arbre et ensuite construire un arbre couvrant à partir de cette racine. La figure 1.4 représente un exemple d'arbre couvrant dans les graphes statiques.

Dans le cas des graphes dynamiques, le problème nécessite encore une fois d'être repensé [21]. Comme pour le problème de l'élection, la dynamique du graphe influe sur la connexité. De la même façon que pour le problème de l'élection deux visions du problème de l'arbre couvrant existent selon que le graphe à la propriété d'avoir ses arêtes qui apparaissent de façon récurrentes ou pas. Dans le cas où les arêtes apparaissent de façon récurrentes, l'arbre peut être calculé sur l'empreinte du graphe dynamique. La figure 1.5(a) représente un exemple d'arbre couvrant général à un instant t. Les arêtes en pointillés sont des arêtes récurrentes non présentes à ce moment là mais appartenant à l'arbre couvrant. Dans le cas où les arêtes ne sont pas récurrentes, l'idée est de maintenir un arbre couvrant par composante connexe ; on appelle cette construction une **forêt couvrante**. La figure 1.5(b) représente un exemple de

forêt couvrante à un instant t. Ce problème fait l'objet du chapitre 4 de cette thèse.

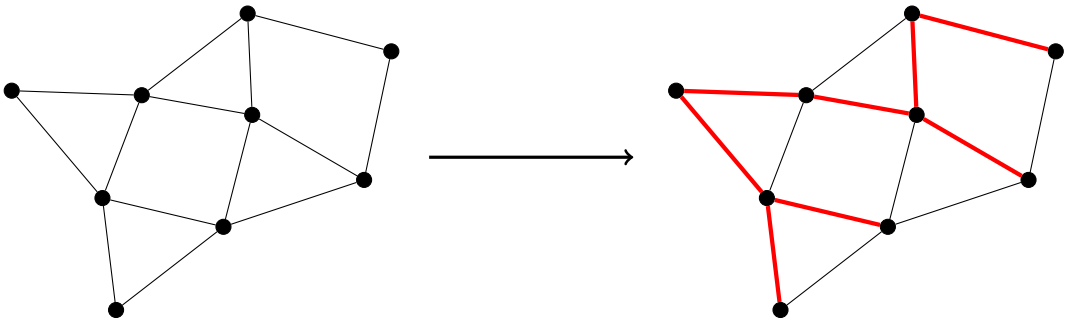
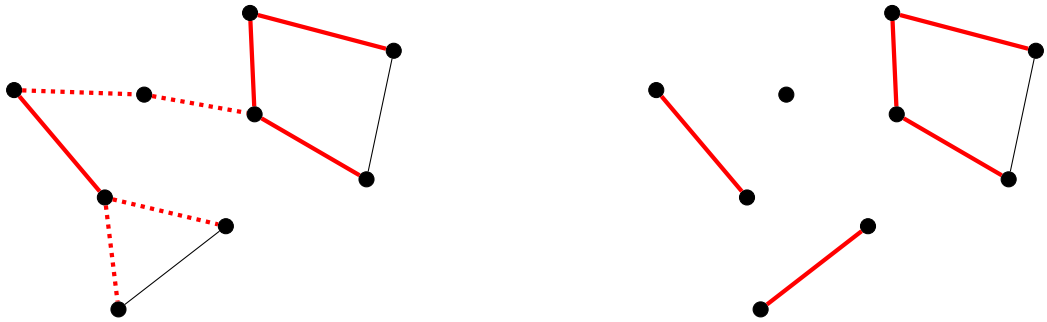


FIGURE 1.4 – Illustration du problème de l'arbre couvrant ; les arêtes en gras représentent les arêtes de l'arbre.



(a) Un arbre couvrant pour l'ensemble du graphe

(b) Un arbre couvrant par composante connexe du graphe

FIGURE 1.5 – Illustration du problème de l'arbre couvrant dans le cadre des graphes dynamiques.

1.6 Classes de graphes dynamiques

Dans le domaine des algorithmes distribués dans les graphes dynamiques, les auteurs de l'article [27] ont présenté le fait que les algorithmes ont besoin d'un certain nombre de conditions (propriétés) sur les graphes dynamiques pour atteindre leurs objectifs. Ces conditions (propriétés) peuvent être représentées à l'aide de classes de graphes dynamiques. Les auteurs introduisent aussi une hiérarchie des classes de graphes dynamiques. Ils ont ainsi défini un ensemble de classes et des relations entre ces différentes classes. L'ensemble de classes qu'ils introduisent n'est pas définitif et de nouvelles classes de graphes dynamiques feront leur apparition en fonction des nouvelles propriétés rencon-

trées. Néanmoins les classes proposées incluent les classes de graphes dynamiques les plus étudiées aujourd'hui.

D'un point de vue pratique, chaque classe correspond à des propriétés que la mobilité des noeuds peut offrir et qui permette à un algorithme d'augmenter l'ensemble d'hypothèses qu'il peut faire sur le réseau. Certaines classes correspondent d'ailleurs à des cas bien concrets comme les réseaux périodiques (satellites, transport public, etc.).

La figure 1.6 représente les liens d'inclusion entre les différentes classes de graphes dynamiques connues. Dans cette figure les flèches représentent une relation d'inclusion entre deux classes. Le sens de la flèche est le sens de l'inclusion ; ainsi la classe F_3 est incluse dans la classe F_1. Les différentes classes dynamiques de la figure sont présentées ci-dessous.

Puits (F_1) : Il existe un nœud dans le graphe dynamique pour lequel il existe un trajet de tous les autres nœuds du graphe vers lui.

Propriété 1.1. Un TVG $\mathcal{G} = (V, E, \mathcal{T}, \rho, \zeta)$ appartient à la classe F_1 si $\exists q \in V : \forall p \in V, \exists \mathcal{J}_{(p,q)}$.

Diffuseur (F_2) : Il existe un nœud dans le graphe dynamique pour lequel il existe un trajet de lui vers chaque autre nœud.

Propriété 1.2. Un TVG $\mathcal{G} = (V, E, \mathcal{T}, \rho, \zeta)$ appartient à la classe F_2 si $\exists p \in V : \forall q \in V, \exists \mathcal{J}_{(p,q)}$.

Connexité temporelle (F_3) : Pour chaque couple de nœuds dans le graphe dynamique il existe un trajet entre ces nœuds. Un groupe DTN appartient de part sa définition à cette classe car il nécessite que l'ensemble de ses nœuds puissent communiquer entre eux.

Propriété 1.3. Un TVG $\mathcal{G} = (V, E, \mathcal{T}, \rho, \zeta)$ appartient à la classe F_3 si $\forall (u, v) \in V \times V, \exists \mathcal{J}_{(p,q)}$.

Diffuseur strict (F_4) : Il existe un nœud dans le graphe dynamique pour lequel il existe un trajet strict de lui vers chaque autre nœud.

Propriété 1.4. Un graphe évolutif $\mathcal{G} = (G, S_G)$ appartient à la classe F_4 si $\exists p \in V : \forall q \in V, \exists \mathcal{J}_{st_{(p,q)}}$.

Connexité temporelle stricte (F_5) : Pour chaque couple de nœuds dans le graphe dynamique il existe un trajet strict entre ces nœuds.

Propriété 1.5. Un graphe évolutif $\mathcal{G} = (G, S_G)$ appartient à la classe F_5 si $\forall (p, q) \in V \times V, \exists \mathcal{J}_{st_{(p,q)}}$.

Diffuseur statique (F_6) : Il existe un nœud dans le graphe dynamique pour lequel il existe une arête de lui vers chaque autre nœud dans le graphe sous-jacent.

Propriété 1.6. Un TVG $\mathcal{G} = (V, E, \mathcal{T}, \rho, \zeta)$ appartient à la classe F_6 si $\exists p \in V : \forall q \in V \backslash \{p\}, (p, q) \in E$.

Connexité (F_7) : Pour chaque couple de nœuds dans le graphe dynamique il existe une arête entre ces nœuds dans le graphe sous-jacent.

Propriété 1.7. Un TVG $\mathcal{G} = (V, E, \mathcal{T}, \rho, \zeta)$ appartient à la classe F_7 si $\forall (p, q) \in V \times V, p \neq q \Rightarrow (p, q) \in E$.

Connexité temporelle avec retour (F_8) : Pour chaque couple de nœuds dans le graphe dynamique il existe un trajet aller-retour entre ces nœuds.

Propriété 1.8. Un TVG $\mathcal{G} = (V, E, \mathcal{T}, \rho, \zeta)$ appartient à la classe F_8 si $\forall (p, q) \in V \times V, \exists \mathcal{J}_{(p,q)}, \exists \mathcal{J}'_{(q,p)} : arrival(\mathcal{J}) \leqslant departure(\mathcal{J}')$.

Connexité temporelle récurrente (F_9) : Pour chaque couple de nœuds dans le graphe dynamique il existe un trajet récurrent dans le temps entre ces nœuds. Dans le cadre des groupes DTN la récurrence des trajets est primordiale car chaque nœud doit pouvoir communiquer avec tous les autres de façon répétée.

Propriété 1.9. Un TVG $\mathcal{G} = (V, E, \mathcal{T}, \rho, \zeta)$ appartient à la classe F_9 si $\forall (p, q) \in V \times V, \forall t \in \mathcal{T}, \exists \mathcal{J}_{(p,q)} : departure(\mathcal{J}_{(p,q)}) \geqslant t$.

Arêtes récurrentes (F_{10}) : Le graphe sous-jacent du graphe dynamique est connexe et chaque arête du graphe dynamique réapparaît infiniment souvent.

Propriété 1.10. Un TVG $\mathcal{G} = (V, E, \mathcal{T}, \rho, \zeta)$ appartient à la classe F_{10} si le graphe sous-jacent est connexe et $\forall e \in E, \forall t \in \mathcal{T}, \exists t' > t$ tel que $\rho(e, t') = 1$.

Arêtes récurrentes bornées dans le temps (F_{11}) : Le graphe sous-jacent du graphe dynamique est connexe et chaque arête du graphe dynamique réapparaît dans un temps borné.

Propriété 1.11. Un TVG $\mathcal{G} = (V, E, \mathcal{T}, \rho, \zeta)$ appartient à la classe F_{11} si le graphe sous-jacent est connexe et $\forall e \in E, \forall t \in \mathcal{T}, \exists \Delta \in \mathbb{T}, \exists t' \in [t, t + \Delta[$ tel que $\rho(e, t') = 1$.

Arêtes périodiques (F_{12}) : Le graphe sous-jacent du graphe dynamique est connexe et chaque arête du graphe dynamique réapparaît périodiquement.

Propriété 1.12. Un TVG $\mathcal{G} = (V, E, \mathcal{T}, \rho, \zeta)$ appartient à la classe F_{12} si le graphe sous-jacent est connexe et $\forall e \in E, \forall t \in \mathcal{T}, \exists p \in \mathbb{T}$ tel que $\forall k \in \mathbb{N}, \rho(e, t + kp) = \rho(e, t)$.

Graphe d'interaction complet avec arêtes récurrentes (F_{13}) : Pour chaque couple de nœuds dans le graphe dynamique il existe une arête récurrente dans le temps entre ces nœuds.

Propriété 1.13. Un TVG $\mathcal{G} = (V, E, \mathcal{T}, \rho, \zeta)$ appartient à la classe F_{13} si le graphe sous-jacent est complet et $\forall e \in E, \forall t \in \mathcal{T}, \exists t' > t$ tel que $\rho(e, t') = 1$.

Chemins récurrents (F_{14}) : Quel que soit la date i, pour chaque couple de nœuds du graphe dynamique il existe une date $j \geqslant i$ tels qu'il existe un chemin entre ces nœuds.

Propriété 1.14. Un graphe évolutif temporisé $\mathcal{G} = (G, S_\mathcal{G}, S_T)$ appartient à la classe F_{14} si $\forall (p, q) \in V \times V, \forall t_i \in S_T, \exists t_j \in S_T, t_j \geqslant t_i$ tel qu'il existe un chemin de p vers q dans G_j.

Connexité récurrente (F_{15}) : Le graphe dynamique est connexe (il existe pour tout couple de nœuds un chemin entre eux) de façon récurrente.

Propriété 1.15. Un graphe évolutif temporisé $\mathcal{G} = (G, S_\mathcal{G}, S_T)$ appartient à la classe F_{15} si $\forall t_i \in S_T, \exists t_j \in S_T$ tel que $t_j \geqslant t_i$ et G_j est connexe.

Connexité permanente (F_{16}) : Le graphe dynamique est connexe à chaque instant.

Propriété 1.16. Un graphe évolutif temporisé $\mathcal{G} = (G, S_\mathcal{G}, S_T)$ appartient à la classe F_{16} si $\forall t_i \in S_T$, G_j est connexe.

Sous-graphe couvrant stable durant T unités de temps (F_{17}) : À chaque instant t, il existe durant T unités de temps un sous graphe couvrant du graphe dynamique, identique pour les T unités de temps.

Propriété 1.17. Un graphe évolutif temporisé $\mathcal{G} = (G, S_\mathcal{G}, S_T)$ appartient à la classe F_{17} si $\exists T \in \mathbb{N}, \forall t_i \in S_T$ il existe un sous-graphe couvrant G' du graphe sous-jacent de $\mathcal{G}$ tel que $\forall t_j \in [t_i, t_{i+T-1}]$, G' est un sous graphe couvrant de G_j.

Comme dit précédemment, l'essentiel des travaux sur les graphes dynamiques sont réalisés sur ces classes de graphes. Par exemple, dans l'article [23] les auteurs étudient le problème de la diffusion dans un système distribué et montrent que la diffusion au plus court et la diffusion au plus rapide ne sont pas calculables dans la classe F_{10} alors que la diffusion au plus tôt y est calculable. Ils montrent ensuite que la diffusion au plus rapide n'est pas calculable dans la classe F_{11} mais que par contre dans la classe F_{12} elle l'est. Enfin ils présentent le fait que la diffusion au plus court est calculable dans la classe F_{11}.

Avec l'augmentation des traces de réseaux dynamiques, dû à l'augmentation significative du nombre d'objets connectés dans le monde, il est devenu important de pouvoir classer ces traces en fonction de leurs propriétés. Ces traces peuvent être représentées à l'aide de graphes dynamiques. Classer ces traces que l'on représente à l'aide de graphes dynamiques revient alors à placer

chaque trace dans la classe de graphes dynamiques qui la représente. Les algorithmes nécessitent un certain nombre de condition sur les graphes dynamiques pour pouvoir terminer. Grâce à ces conditions il est possible de dire sur quelles classes de graphes dynamiques les algorithmes vont pouvoir être utilisés.

La difficulté est de classer ces traces de manière automatique. Un certain nombre de travaux commencent sur ce sujet comme décrit par exemple dans l'article [28] où les auteurs introduisent un algorithme permettant de tester l'appartenance d'un graphe dynamique à la classe F_{17}. De notre côté nous avons voulu tester l'appartenance aux classes représentant la connexité temporelle.

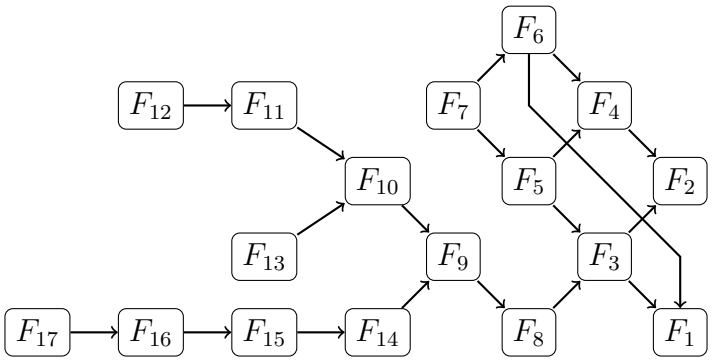


FIGURE 1.6 – Diagramme des relations d'inclusion entre les classes de graphes dynamiques issu de l'article [27].

1.7 Conclusion

Dans ce chapitre nous avons donné des définitions autour des systèmes distribués et de la réprésentation des réseaux à l'aide de graphes. Nous avons aussi présenté les différents modèles de graphes dynamiques et les différentes notions qui y sont associées. Nous avons ensuite discuté de l'impact de la dynamique du réseau sur les problèmes classiques de l'algorithmique distribuée. Enfin nous avons présenté un certain nombre de classes de graphes dynamiques qui capturent les différentes hypothèses que l'on peut vouloir faire sur l'évolution de la topologie.

Chapitre 2

Test de la connexité temporelle

Sommaire

Dans ce chapitre, nous nous intéresserons au problème de savoir si un graphe dynamique donné est temporellement connexe ou non. La motivation pour ce problème est de pouvoir vérifier, en restreignant la durée de vie à une fenêtre donnée, si un groupe de nœuds donné peut bel et bien communiquer dans cette fenêtre et consitue ainsi un groupe DTN comme défini précédemment.

Nous rappelons d'abord la définition de la connexité temporelle. Nous présentons ensuite la notion de fermeture transitive des trajets, qui est une notion clef permettant de détecter si un graphe est temporellement connexe. Enfin, nous proposons un ensemble d'algorithmes permettant de calculer la fermeture transitive des trajets (dans deux variantes, selon que les trajets sont stricts ou non-stricts). La complexité de ces algorithmes est comparée à celle d'algorithmes existants (ou plus précisément au détournement de ces algorithmes pour accomplir le même objectif). L'ensemble de ces travaux fait l'objet de deux publications, l'une en anglais lors de la conférence Aetos 2014 [12] et l'autre en français dans le cadre d'Algotel 2014 [11].

D'un point de vue pratique, ce chapitre permet de collecter des traces réelles de connexité dans le réseau et de tester, à postériori, si la contrainte de

diamètre temporel a bien été respectée. Ainsi, on peut vérifier si nos groupes DTN vérifient bien la condition nécessaire à distinguer un appareil éloigné temporairement d'un appareil perdu.

2.1 Connexité temporelle

Avant de revenir sur la raison de notre intérêt pour la connexité temporelle il faut tout d'abord prendre le temps d'expliciter deux termes relatifs à la connexité dans les graphes dynamiques. Il s'agit des termes *temporellement connexe* et *connexe à travers le temps*. Ces deux termes sont parfois utilisés pour désigner deux choses différentes. Le terme de *temporellement connexe* est généralement utilisé la connexité temporelle classique (*i.e.* l'existence d'au moins un trajet pour chaque couple de nœuds du graphe). La plupart du temps le terme *connexe à travers le temps* est utilisé pour désigner la même chose mais parfois il désigne la connexité temporelle récurrente qui est une contrainte plus forte sur le graphe dynamique (*i.e.* l'existence récurrente d'un trajet pour chaque couple de nœuds).

Définition 2.1. Soit $\mathcal{G}$ un graphe dynamique. $\mathcal{G}$ est **temporellement connexe** si et seulement si pour tout couple de nœuds (u, v) de $\mathcal{G}$ il existe un trajet de u vers v.

Dans la suite de ce chapitre nous nous intéresserons aux *graphes temporellement connexes* (classe F_3) et plus précisément nous chercherons à détecter si un graphe l'est ou pas. L'intérêt pour cette classe de graphes dynamiques vient du fait que la majorité des problèmes étudiés aujourd'hui nécessite au minimum cette propriété. Nous pouvons voir dans la figure 1.6 page 21 que la classe F_3 contient la majeure partie des classes de graphes dynamiques connues. Il s'agit d'une propriété charnière par le fait qu'elle assure que tous les nœuds peuvent communiquer deux à deux au moins une fois.

Nos travaux portent sur les groupes DTN qui ont un diamètre temporel borné (voir paragraphe 1.4.2 page 17) et il nous permet donc vérifier si le graphe dynamique représentant chaque groupe DTN respecte bien cette contrainte. Or, pour vérifier qu'un graphe dynamique $\mathcal{G}$ respecte bien la contrainte d'un diamètre temporel borné par un entier B il suffit de vérifier qu'à chaque instant t le sous-graphe de l'intervalle de temps $[t, t + B]$ est temporellement connexe.

Pour la détection de la connexité, la notion clé est la fermeture transitive des trajets.

2.2 Fermeture transitive des trajets

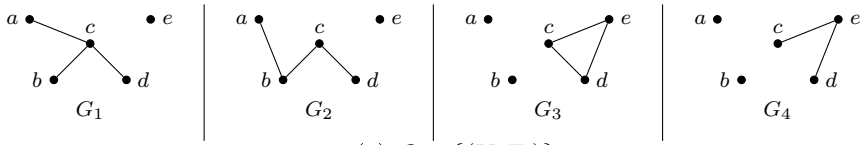
La fermeture transitive des trajets est un concept introduit dans l'article [54]. Il s'agit d'une extension du concept de la fermeture transitive des

chemins qui existe dans le cadre des graphes statiques. De la même façon que pour la fermeture transitive des chemins, le résultat est un graphe statique orienté (que le graphe dynamique soit orienté ou pas) dont chaque arc entre deux nœuds u et v représente l'existence d'un trajet issue de u et terminant sur v. L'orientation de ce graphe dans le cadre de graphes dynamiques non-orientés vient de la notion de temps qui implique que les trajets ne peuvent être parcourus que dans un seul sens. Deux variantes de la fermeture transitive des trajets coexistent selon que l'on considère des trajets stricts ou des trajets non-stricts (voir définition des trajets dans la section 1.3.3.3 page 16).

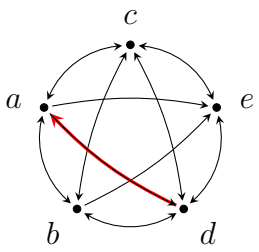
Définition 2.2. Soit $G = (V, E)$ un graphe statique et $\mathcal{G} = \{G, S_G\}$ un graphe dynamique, $\mathcal{G}^* = (V, E')$ est la fermeture transitive non-stricte de $\mathcal{G}$ si et seulement si pour tout trajet $\mathcal{J} \in \mathcal{J}_G^*$ partant d'un nœud u et arrivant sur un nœud v il existe un arc (u, v) dans $\mathcal{G}^*$.

Définition 2.3. Soit $G = (V, E)$ un graphe statique et $\mathcal{G} = \{G, S_G\}$ un graphe dynamique, $\mathcal{G}_{st}^* = (V, E')$ est la fermeture transitive stricte de $\mathcal{G}$ si et seulement si pour tout trajet strict $\mathcal{J}_{st} \in \mathcal{J}_G^*$ partant d'un nœud u et arrivant sur un nœud v il existe un arc (u, v) dans $\mathcal{G}_{st}^*$.

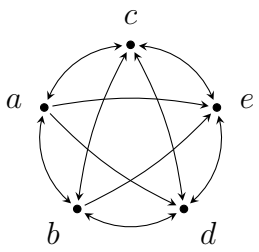
La figure 2.1 représente un graphe dynamique sous la forme d'un graphe évolutif (une séquence de graphes statiques) et ses deux fermetures transitives. L'arc en rouge gras, figure 2.1(b), représente le trajet supplémentaire possible entre d et a dans le cas où les trajets sont non-stricts.



(a) $\mathcal{G} = \{(V, E_i)\}$



(b) Non-stricte ($\mathcal{G}^*$)



(c) Stricte ($\mathcal{G}_{st}^*$)

FIGURE 2.1 – Les figures (b) et (c) représentent respectivement la fermeture transitive non-stricte et la fermeture transitive stricte du graphe dynamique non orienté donné dans la figure (a).

La raison d'être de ces deux définitions distinctes est de rendre compte de la problématique de latence sur un lien de communication individuel. En effet, les

trajets non-stricts sont ceux qui négligent la latence en permettant un nombre de saut non-limités sur une période de temps pourtant limitée. Dans le cas où cette hypothèse semble trop forte d'un point de vue pratique, la définition des trajets stricts (et donc des fermetures strictes) permet de limiter ce nombre de saut à 1. Si on voulait la limiter à k, il suffirait de dédoubler chaque graphe k fois, nous ne perdons donc pas (fondamentalement) de généralité ici.

De la structure de la fermeture transitive, il peut être déduit l'appartenance d'un graphe dynamique à plusieurs classes de graphes [19], en particulier l'appartenance à la classe des graphes temporellement connexes . Dans ce cas, la fermeture transitive non-stricte est un graphe complet (il existe un trajet non-strict entre tous les couples de nœuds du graphe dynamique).

Il est possible de tester l'appartenance à la classe des graphes temporellement strictement connexes à l'aide de la fermeture transitive stricte. La fermeture transitive stricte doit alors être un graphe complet, *i.e.* il doit exister un trajet strict pour chaque couple de nœuds dans le graphe dynamique.

A l'aide des deux fermetures transitives, il est également possible de vérifier l'appartenance aux classes *puits*, *diffuseur* et *diffuseur strict* (respectivement les classes F_1, F_2 et F_4 de la section 1.6) :

— La classe *puits* est identifiable lorsque tous les nœuds du graphe ont un arc vers le même nœud v dans la fermeture transitive non-stricte.

— La classe *diffuseur* est identifiable lorsque un nœud u a un arc vers tous les autres nœuds dans la fermeture transitive non-stricte.

— La classe *diffuseur strict* est identifiable lorsqu'un nœud u a un arc vers tous les autres nœuds dans la fermeture transitive stricte.

2.3 Calcul de la fermeture transitive

Comme nous venons de le voir, la fermeture transitive des trajets permet de déterminer si un graphe appartient à un certain nombre de classes de graphes dynamiques et donc de déterminer *in fine* quels algorithmes sont opérationnels sur ledit graphe.

Avant de présenter nos algorithmes, il faut noter que plusieurs algorithmes existants peuvent être adaptés pour calculer la fermeture transitive stricte $\mathcal{G}^*_{st}$, avec des différences de complexité, à la fois entre eux et avec notre algorithme, pour lequel nous caractérisons les graphes de prédilection. Pour discuter des complexités des différents algorithmes nous utiliserons les notations suivantes :

Soit $\mathcal{G} = \{(V, S_G)\}$ un graphe évolutif (*cf.* définition 1.23 page 15). On note $\delta = |S_G|$ (*i.e.* le nombre de sous-graphes). On distingue deux paramètres pour rendre compte du nombre d'arcs dans le graphe :

— μ est le nombre maximal d'arcs existant à une même étape, i.e. $\mu = max(|E_i|)$.

— m est le nombre total d'arcs pouvant exister au cours du temps, i.e. $m = |\cup E_i|$.

Bien sûr, quel que soit le graphe considéré, on a $m \geqslant \mu$, et même souvent $m \gg \mu$.

2.3.1 Adaptation de l'algorithme de diffusion au plus tôt

Trois algorithmes de diffusion offlines sont proposés dans l'article [54].

— Le premier calcule les trajets optimaux au plus tôt d'un sommet vers tous les autres.
— Le second calcule les trajets optimaux au plus court d'un sommet vers tous les autres.
— Le dernier calcule les trajets optimaux au plus rapide d'un sommet vers tous les autres.

Chacun de ces algorithmes prend un nœud en entrée qui sera le nœud émetteur et calcule l'arbre de diffusion selon le critère donné. Une exécution de l'un de ces algorithmes pour un nœud émetteur donné permet de connaître tous les nœuds atteignables à partir de ce nœud et cela grâce à l'arbre de diffusion. Si on exécute l'un de ces algorithmes pour tous les nœuds du graphe on obtient alors la fermeture transitive stricte du graphe. N'importe lequel de ces trois algorithmes peut être adapté au calcul de $\mathcal{G}^*_{st}$.

Le plus rapide des trois algorithmes est celui qui calcule les trajets au plus tôt. Il a un temps d'exécution en $O(m \log \delta + n \log n)$ pour une source. Étant donné que l'algorithme doit être exécuté pour tous les nœuds pour calculer la fermeture transitive, son temps total d'exécution est $O(n(m \log \delta + n \log n))$.

2.3.2 Adaptation de l'algorithme de calcul des graphes d'accessibilité dynamique

Un algorithme, calculant une généralisation de la fermeture transitive des trajets, a été proposé dans [53]. Cette généralisation, appelée *graphe d'accessibilité dynamique*, correspond à une fermeture transitive des trajets paramétrée par une date de départ minimale des trajets et une durée maximale pour les trajets.

Les graphes d'accessibilité dynamique sont définis sur les graphes dynamiques donnés sous la forme de TVG (*time-varying graphs*, voir définition 1.12, chapitre 1), à savoir un quintuplet $\mathcal{G} = (V, E, \mathcal{T}, \rho, \zeta)$. Le domaine temporelle du graphe dynamique est $\mathcal{T} = \mathbb{R}^+$. Dans ces travaux la fonction de

latence ζ retourne toujours le même durée pour toutes les arêtes du graphe dynamique.

Définition 2.4. Soient $\mathcal{G}$ un graphe dynamique, $\gamma \in \mathbb{R}^+$, $\mathcal{R}_\gamma$ est le **graphe d'accessibilité dynamique** de $\mathcal{G}$ pour une durée maximale γ.
Plus formellement, $\forall t, (u,v) \in \mathcal{R}_\gamma(t)$ si et seulement si $u \neq v$ et il existe à l'instant t de $\mathcal{G}$ un trajet (voir définition 1.16) $\mathcal{J}$ de u à v tel que $departure(\mathcal{J}) \geqslant t$ et $arrival(\mathcal{J}) \leqslant t + \gamma$.

Si on fixe $\gamma = \mathcal{T}^+$ et $t = 0$, alors le graphe d'accessibilité dynamique $\mathcal{R}_\gamma$ permet de savoir s'il existe un trajet entre toutes les paires de nœuds du graphe dynamique. Néanmoins, les auteurs se sont rendus compte que s'il est très facile d'obtenir le graphe d'accessibilité dynamique pour une durée maximale des trajets de 1, il est beaucoup plus compliqué d'obtenir ceux pour des durées plus grandes même en utilisant les graphes d'accessibilités de délais inférieurs. Les auteurs présentent une formule mathématique permettant d'obtenir un graphe d'accessibilité supérieur à partir de deux graphes inférieurs mais la formule se retrouve ne pas être calculable. Les auteurs sont alors revenus aux TVG η-réguliers définis ci-dessous pour simplifier le problème.

Définition 2.5. Soit $\mathcal{G} = (V, E, \mathcal{T}, \rho, \zeta)$ un TVG, $\mathcal{G}$ est un **TVG η-régulier** s'il existe $\eta > 0$ tel que $\forall k \in \mathbb{N}$, $k\eta < t_1 \leqslant t_2 < (k+1)\eta \Longrightarrow \mathcal{G}(t_1) = \mathcal{G}(t_2) \subseteq \mathcal{G}(k\eta)$.
η est appelé *résolution du système*.
L'intervalle $[k\eta, (k+1)\eta]$ est appelé *période*.

L'idée avec ce modèle est que dans les expérimentations qui sont réalisées les données sont généralement capturées toutes les secondes ou millisecondes et durant ce laps de temps il est fait la supposition que le système reste stable.

Sur ces graphes η-réguliers, les auteurs définissent un algorithme permettant de calculer une approximation fine du graphe d'accessibilité $\mathcal{R}_{\gamma+\lambda}$ en utilisant les approximations des graphes d'accessibilités $\mathcal{R}_\gamma$ et $\mathcal{R}_\lambda$. Ces approximations sont justes sur les extrémités des périodes mais peuvent contenir des erreurs sur les zones intermédiaires. Néanmoins, l'existence ou pas des trajets est bien détectée.

L'algorithme proposé peut donc être utilisé pour calculer $\mathcal{G}^*_{st}$ en calculant l'approximation de $\mathcal{R}_{\mathcal{T}^+}$. Si on fait la supposition que pendant une période le graphe dynamique ne change pas il est possible de représenter ces graphes dynamiques sous la forme d'une séquence de sous-graphes tels que chaque sous-graphe est l'état du graphe dynamique au début de chaque période. Comme pour les graphes évolutifs, δ représente alors le nombre de sous-graphes du graphe dynamique et $\mathcal{R}_\delta$ est alors le graphe d'accessibilité que l'on cherche à calculer. La complexité en temps pour une exécution de l'algorithme pour calculer le nouveau graphe d'accessibilité à partir des deux précédents est

$O(\delta mn \log n)$. Pour obtenir l'approximation de $\mathcal{R}_\delta$ il est nécessaire de faire $\log \delta$ appels à l'algorithme pour calculer tous les résultats intermédiaires donc la complexité en temps est en $O(\delta \log \delta \ mn \log n)$.

2.3.3 Nouveau algorithme pour le calcul de la fermeture transitive

Contrairement aux algorithmes précédents dont le but original n'était pas de calculer la fermeture transitive des trajets nous proposons dans cette section une approche dédiée au calcul de la fermeture transitive (d'abord stricte) des trajets d'un graphe évolutif non-temporisé orienté $\mathcal{G} = \{(V, E_i)\}$. Nos deux algorithmes sont de type *streaming* et sont capables de terminer sitôt la connexité temporelle atteinte. Un sous-produit de l'exécution est de rendre $\mathcal{G}^*_{st}$ et $\mathcal{G}^*$ disponibles pour d'éventuelles requêtes ultérieures de type st-connexité temporelle (*i.e.* existe-t-il un trajet entre deux nœuds ?), qui se réduisent alors à de simples requêtes de recherche d'incidence dans un graphe statique.

Nous proposons ci-dessous un algorithme de calcul de la fermeture transitive stricte $\mathcal{G}^*_{st}$ dans le cas général où $\mathcal{G}$ est orienté.

2.3.3.1 Calcul de la fermeture transitive stricte

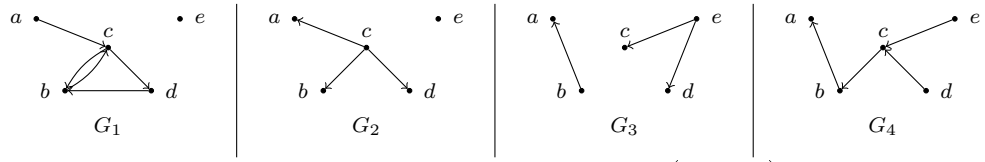
Le principe de l'algorithme est de construire, étape après étape (sous-graphe par sous-graphe), la liste de tous les prédécesseurs de chaque sommet, *i.e.*, pour un sommet v, l'ensemble $\{u : u \overset{st}{\rightsquigarrow} v\}$. Soit $\mathcal{P}(v,t)$ l'ensemble des prédécesseurs de v à l'issue des t premières étapes (*i.e.* en tenant compte des ensembles d'arêtes $E_1, ..., E_t$). Initialement $\mathcal{P}(v,0)$ contient le nœud v. A l'étape i, le cœur du traitement consiste à ajouter $\mathcal{P}(u, i-1)$ à $\mathcal{P}(v,i)$ pour chaque arête $(u,v) \in E_i$. En pratique, seules deux variables $\mathcal{P}(v)$ et $\mathcal{P}^+(v)$ sont maintenues pour chaque nœud v, où $\mathcal{P}^+(v)$ contient les nouveaux prédécesseurs de v (ajoutés durant l'étape courante). La figure 2.2 représente un graphe dynamique orienté et sa fermeture transitive des trajets stricts.

Lemme 2.1. Pour tout $v \in V$, $|\mathcal{P}(v)| \leq \delta\mu$, *i.e.*, un nœud ne peut avoir plus de $\delta\mu$ prédécesseurs.

Démonstration (*par l'absurde*)**.** S'il existe un nœud v tel que $|\mathcal{P}(v) \setminus v| > \delta\mu$, alors, par définition, il existe plus de $\delta\mu$ sommets u différents de v tels que $u \rightsquigarrow v$. Chacun de ces sommets est donc l'origine d'au moins un arc, ce qui implique que plus de $\delta\mu$ arcs distincts ont existé. □

Théorème 2.1. L'Algorithme 1 calculant la fermeture transitive stricte d'un graphe $\mathcal{G}$ a une complexité en temps en $O(\delta\mu n)$.

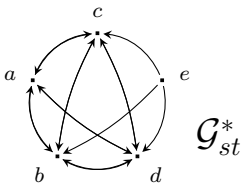
Démonstration. La boucle d'initialisation est linéaire en n. Vient ensuite la boucle principale (lignes 4-18), qui itère autant de fois qu'il y a de sous-graphes dans $\mathcal{G}$, i.e. δ fois. Elle comporte trois sous-boucles (lignes 6-8, 9-11 et 13-16), chacune étant dominée par $O(|E_i| \cdot n) = O(\mu n)$. La première sous-boucle parcourt tous les arcs présents dans le sous-graphe et fait une opération d'union sur au plus n nœuds car un nœud a au plus n prédécesseurs. La deuxième sous-boucle parcourt elle l'ensemble des nœuds pour lesquels on a modifié les prédécesseurs, il y a au plus autant de nœuds que le nombre d'arcs du sous-graphe. Elle fait ensuite une opération d'union sur au plus n nœuds car un nœud a au plus n prédécesseurs. La troisième sous-boucle reste quand à elle dominée par $O(\mu n)$. Enfin, la construction de la fermeture transitive, si cette dernière n'est pas complète avant la fin du parcourt des sous-graphes, consiste en une boucle qui, pour chaque nœud, itère sur ses prédécesseurs. Or, on sait que le nombre de prédécesseurs d'un nœud donné ne peut excéder $\delta\mu$ (Lemme 2.1). Cette dernière boucle est donc elle aussi contenue dans $O(\delta\mu n)$. □



(a) Représentation graphique d'un graphe dynamique avant (en haut) et après application de la fermeture transitive statique (en bas).

$\mathcal{P}_0(a) = \{a\}$	$\mathcal{P}_1(a) = \{a\}$	$\mathcal{P}_2(a) = \{a, c, b\}$		$\mathcal{P}_4(a) = \{a, c, b, d\}$
$\mathcal{P}_0(b) = \{b\}$	$\mathcal{P}_1(b) = \{b, c, d\}$	$\mathcal{P}_2(b) = \{b, c, d, a\}$		$\mathcal{P}_4(b) = \{b, c, d, a, e\}$
$\mathcal{P}_0(c) = \{c\}$	$\mathcal{P}_1(c) = \{c, a, b\}$	$\mathcal{P}_2(c) = \{c, a, b\}$	...	$\mathcal{P}_4(c) = \{c, a, b, e, d\}$
$\mathcal{P}_0(d) = \{d\}$	$\mathcal{P}_1(d) = \{d, c\}$	$\mathcal{P}_2(d) = \{d, c, a, b\}$		$\mathcal{P}_4(d) = \{d, c, a, b, e\}$
$\mathcal{P}_0(e) = \{e\}$	$\mathcal{P}_1(e) = \{e\}$	$\mathcal{P}_2(e) = \{e\}$		$\mathcal{P}_4(e) = \{e\}$

(b) Calcul des prédécesseurs



(c) Fermeture transitive des trajets stricts

FIGURE 2.2 – Exemple de calcul de la fermeture transitive des trajets stricts d'un graphe dynamique orienté.

2.3.3.2 Calcul de la fermeture transitive non-stricte

Dans cette section, nous nous intéressons au calcul de $\mathcal{G}^*$, *i.e.* la fermeture transitive des trajets où un nombre illimité d'arêtes peuvent être traversées à chaque étape (trajets non-stricts). Une simple observation nous permet de réutiliser l'Algorithme 1 de manière quasiment directe. En effet, la relaxation de la contrainte relative à l'aspect stricts des trajets implique qu'à chaque étape i, si un chemin (au sens classique) existe de u vers v, alors u peut joindre v à cette même étape. L'algorithme consiste donc à pré-calculer, à chaque étape, la fermeture transitive (au sens classique, statique du terme) des arcs présents dans G_i, résultant en un graphe G_i^* dont les arcs correspondent aux chemins dans G_i. L'Algorithme 1, appliqué ensuite au graphe dynamique $\{G_i^*\}$, produit ainsi la fermeture transitive $\mathcal{G}^*$ des trajets *non-stricts* de $\mathcal{G}$.
La figure 2.3 représente un graphe dynamique orienté, puis le graphe dynamique obtenu après l'application de la fermeture transitive des chemins pour chacun de ses sous-graphes et enfin la fermeture transitive des trajets non-stricts.

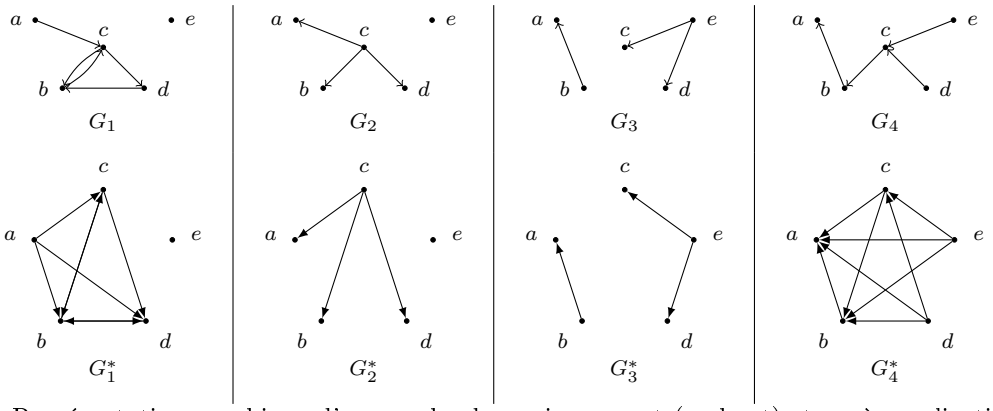
La complexité en temps de cet algorithme dépend essentiellement du coût requis pour calculer la fermeture transitive statique G_i^* des graphes G_i. Ce calcul peut être réalisé par une recherche en profondeur (DFS) ou en largeur (BFS), exécutée depuis chaque sommet dans G_i, chacune de ces exécutions ayant un coût en $O(|E_i|) = O(\mu)$. Ainsi, le surcoût engendré par ce traitement reste confiné dans le même ordre de grandeur que celui identifié précédemment, à savoir $O(\delta\mu n)$.

2.3.4 Comparaison de complexité

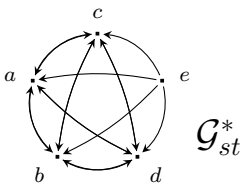
Cette section compare la complexité de notre algorithme à celle de la stratégie utilisant le calcul des trajets au plus tôt de [54]. Cette stratégie, qui revient à exécuter l'algorithme depuis chaque sommet, a une complexité totale en $O(n(m \log \delta + n \log n))$, où m est le nombre total d'arêtes pouvant exister au cours du temps, *i.e.* $|\cup E_i|$, et non μ.

Il ne sert à rien de se comparer à l'algorithme de [53] présenté dans la section 2.3.2 car il est toujours plus couteux en temps que l'algorithme de diffusion présenté dans la section 2.3.1 et que notre algorithme. La complexité en temps de l'utilisation de cet algorithme pour calculer la fermeture transitive est $O(\delta \log \delta\ mn \log n)$ et celle de notre algorithme est $O(\delta\mu n)$. On sait que généralement $m \geqslant \mu$ et de plus l'algorithme de [53] possède des facteurs $\log \delta$ et $\log n$ en plus par rapport à notre complexité.

Il s'agit donc de comparer cet ordre de grandeur à $O(\delta\mu n)$, ou après simplification par n, de comparer $O(\delta\mu)$ à $O(m \log \delta + n \log n)$. Ces grandeurs appartenant à un espace à quatre dimensions : μ, m, δ et n, il n'est pas aisé de



(a) Représentation graphique d'un graphe dynamique avant (en haut) et après application de la fermeture transitive statique (en bas).



(b) Fermeture transitive des trajets non stricts

FIGURE 2.3 – Exemple de calcul de la fermeture transitive des trajets non-stricts d'un graphe dynamique.

les comparer. Nous étudions asymptotiquement en n, en faisant varier les rapports entre μ, m et δ. Précisément, nous faisons varier les ordres de grandeur de μ et m (densité « instantanée » *vs.* densité « cumulée ») pour plusieurs ratios de valeurs possibles entre δ et n (*i.e.* nombre d'étapes dans $\mathcal{G}$ en fonction de n). Les tableau proposés (Table 2.1, Table 2.2 et Table 2.3) contiennent environ 50 résultats, dont une dizaine mettent en évidence là où a lieu le basculement de complexité entre les deux algorithmes. Pour simplifier la vérification de ces résultats, nous fournissons dans la colonne de droite une expression intermédiaire, obtenue après simple substitution de μ et m dans les deux expressions à comparer (*i.e.* $O(\delta\mu)$ et $O(m \log \delta + n \log n)$).

2.4 Conclusion

En résumé, nous avons présenté un algorithme permettant de calculer la fermeture transitive des trajets dans un graphe dynamique. Les tableaux 2.1, 2.2 et 2.3 confirment que notre solution se comporte d'autant mieux que l'écart entre densité « instantanée » et densité « cumulée » est élevé, ce qui n'est pas surprenant. Il n'est pas surprenant non plus, au vu de la présence des facteurs

δ versus $\log \delta$, que notre solution soit plus efficace lorsque le nombre d'étapes est relativement faible. En outre, le tableau révèle plusieurs éventails de valeurs pour lesquelles notre solution se comporte mieux, comme par exemple pour les trios (μ, m, δ) valant $(O(n), \Theta(n^2), O(n))$, ou $(O(\log n), \Omega(n \log n), O(n))$, ou encore $(O(\sqrt{n})), \Omega(n), O(\sqrt{n}))$.

Enfin, nous pensons que l'impact coûteux du paramètre δ dans la complexité théorique de notre algorithme doit être relativisé, eu égard au fait que l'algorithme termine dès que la connexité temporelle est atteinte. En effet, si on considère des modèles de graphes dynamiques aléatoires tels que les graphes à évolution arête-markovienne [29] (*Edge-Markovian Evolving Graphs*), la connexité temporelle s'établit avec forte probabilité après un nombre sous-logarithmique d'étapes. La performance de notre algorithme dans les scénarios représentés par ce type de modèle correspondrait donc, en réalité, à la colonne la plus à gauche du tableau. L'ensemble de ces travaux ont fait l'objet d'une publication à Algotel [11] ainsi que dans la conférence Aetos [12].

Input : Un graphe dynamique $\mathcal{G}$ donné sous la forme $(V, \{E_i\})$
Output : Un ensemble d'arcs E^* tel que $\mathcal{G}^*_{st} = (V, E^*)$

```
   // Initialisation
1  foreach v in V do
2  |  P(v) ← {v}        // Chaque nœud est son propre prédécesseur
3  |  P+(v) ← ∅
4  foreach E_i in {E_i} do
5  |  UpdateV ← ∅              // Contient tous les nœuds dont les
   |                           // prédécesseurs sont mis à jour
   |  // Liste les prédécesseurs induits par les arcs de E_i
6  |  foreach (u, v) in E_i do
7  |  |  P+(v) ← P+(v) ∪ P(u)
8  |  |  UpdateV ← UpdateV ∪ {v}
   |  // Ajout des prédécesseurs trouvés aux prédécesseurs déjà
   |     connus
9  |  foreach v in UpdateV do
10 |  |  P(v) ← P(v) ∪ P+(v)
11 |  |  P+(v) ← ∅
   |  // Teste si la fermeture transitive est complète et
   |     s'arrête si c'est le cas
12 |  isComplete ← true
13 |  foreach v in V do
14 |  |  if |P(v)| < |V| then
15 |  |  |  isComplete ← false
16 |  |  |  break
17 |  if isComplete then
   |  |  // L'algorithme s'arrête et retourne un graphe complet
   |  |     (les arcs)
18 |  |  return V × V \ {loops}
   // Construction de la fermeture transitive basée sur les
      prédécesseurs
19 E* ← ∅
20 foreach v in V do
21 |  foreach u in P(v) \ {v} do
22 |  |  E* ← E* ∪ (u, v)
23 return E*
```

Algorithme 1 : Algorithme de calcul de la fermeture transitive $\mathcal{G}^*_{st}$

$\mu = \Theta(.)$	$\delta = \Theta(\log n)$	$\delta = \Theta(\sqrt{n})$	$\delta = \Theta(n)$	$\delta = \Theta(n^2)$	Calcul intermédiaire $\Theta(.) \pm \Theta(.)$
$\log n$	n/a	n/a	$\approx$	$+$	$k \log n \pm n \log k + n \log n$
$\sqrt{n}$	n/a	$-$	$+$	$+$	$k\sqrt{n} \pm n \log k + n \log n$
n	$\approx$	$+$	$+$	$+$	$kn \pm n \log k + n \log n$

TABLE 2.1 – Comparaison de la complexité en temps de notre algorithme à l'adaptation de l'algorithme de [54] pour $m = \Theta(n)$. *Les symboles $-$ (resp $+$, $\approx$) indiquent les plages de paramètres pour lesquelles notre solution a une complexité asymptotique plus faible (resp. plus forte, du même ordre de grandeur). Le symbole* n/a *représente lui l'inexistence de valeur.*

$\mu = \Theta(.)$	$\delta = \Theta(\log n)$	$\delta = \Theta(\sqrt{n})$	$\delta = \Theta(n)$	$\delta = \Theta(n^2)$	Calcul intermédiaire $\Theta(.) \pm \Theta(.)$
$\log n$	n/a	n/a	$-$	$+$	$k \log n \pm (n \log n) \log k$
$\sqrt{n}$	n/a	n/a	$+$	$+$	$k\sqrt{n} \pm (n \log n) \log k$
n	$-$	$+$	$+$	$+$	$kn \pm (n \log n) \log k$
$n \log n$	$+$	$+$	$+$	$+$	$k \pm \log k$

TABLE 2.2 – Comparaison de la complexité en temps de notre algorithme à l'adaptation de l'algorithme de [54] pour $m = \Theta(n \log n)$. *Les symboles $-$ (resp $+$, $\approx$) indiquent les plages de paramètres pour lesquelles notre solution a une complexité asymptotique plus faible (resp. plus forte, du même ordre de grandeur). Le symbole* n/a *représente lui l'inexistence de valeur.*

$\mu = \Theta(.)$	$\delta = \Theta(\log n)$	$\delta = \Theta(\sqrt{n})$	$\delta = \Theta(n)$	$\delta = \Theta(n^2)$	Calcul intermédiaire $\Theta(.) \pm \Theta(.)$
$\log n$	n/a	n/a	n/a	$\approx$	$k \log n \pm n^2 \log k$
$\sqrt{n}$	n/a	n/a	n/a	$+$	$k\sqrt{n} \pm n^2 \log k$
n	n/a	n/a	$-$	$+$	$kn \pm n^2 \log k$
$n \log n$	n/a	n/a	$\approx$	$+$	$k(n \log n) \pm n^2 \log k$
$n\sqrt{n}$	n/a	$-$	$+$	$+$	$k(n\sqrt{n}) \pm n^2 \log k$
n^2	$+$	$+$	$+$	$+$	$kn^2 \pm n^2 \log k$

TABLE 2.3 – Comparaison de la complexité en temps de notre algorithme à l'adaptation de l'algorithme de [54] pour $m = \Theta(n^2)$. *Les symboles $-$ (resp $+$, $\approx$) indiquent les plages de paramètres pour lesquelles notre solution a une complexité asymptotique plus faible (resp. plus forte, du même ordre de grandeur). Le symbole* n/a *représente lui l'inexistence de valeur.*

Chapitre 3

Exploration d'arbres par des agents

Sommaire

De par l'augmentation des capacités des systèmes autonomes et de par la volonté de réduire les risques pour les personnes, les systèmes autonomes

sont de plus en plus utilisés pour explorer des bâtiments à risque (bâtiments en feux, centrales nucléaires, ...). Néanmoins, quand on parle d'exploration et plus précisément d'exploration d'un bâtiment, il faut se poser deux questions : qu'est ce qu'un bâtiment et que signifie explorer un bâtiment ?

La représentation d'un bâtiment dans le cadre de son exploration est la suivante. Un bâtiment est un ensemble de pièces, escaliers, couloirs qui sont reliés par des portes. Il est possible de passer d'une pièce à une autre si et seulement si il existe une porte entre les deux. Un bâtiment peut avoir plusieurs entrées. Dans notre cas nous considérons que le bâtiment a une entrée unique. Les couloirs et les escaliers sont également considérés comme des pièces.

L'exploration d'un bâtiment par une personne consiste à ce que cette personne visite toutes les pièces de ce bâtiment à partir de son point de départ. On peut transformer le problème de l'exploration d'un bâtiment en un problème d'exploration d'un graphe par un ou plusieurs agents. Chaque pièce du bâtiment est représentée par un nœud du graphe et chaque porte entre deux pièces est représentée par une arête entre les deux nœuds correspondants. Nous considérons que le déploiement des agents au point de départ se fait sur l'une des entrées du batiment (on pourrait utiliser plusieurs entrées, mais le déploiement en serait complexifié). La présence potentielle d'étages différents dans le batiment est prise en compte par l'intermédiaire des graphes que nous générons pour les simulations, qui sont des sous-ensembles de grilles en 3 dimensions.

De plus, nous supposons qu'il est possible à partir d'une pièce d'un bâtiment d'aller dans toutes les autres pièces de ce bâtiment. Les graphes correspondants aux bâtiments étudiés sont donc connexes.

La figure 3.1 représente un appartement sous la forme d'un graphe. La salle de bain (1) n'est accessible que par la chambre (2) qui n'est accessible que par le couloir (3) qui n'est accessible que par l'entrée (4). À contrario la cuisine (5) est accessible à la fois par l'entrée (4) et la salle à manger (6) qui est elle-même accessible depuis l'entrée (4). Il existe un cycle entre les pièces 4, 5 et 6. Cependant, bien qu'il puisse exister des cycles, la structure d'un bâtiment est, en général, quasiment arborescente.

Dans ce chapitre, nous nous intéressons à l'exploration collective de graphes avec contraintes. Le problème de l'exploration est un problème difficile (au sens de la complexité algorithmique) et les nouvelles contraintes que nous considérons le complique encore plus. Pour simplifier le problème, nous nous concentrons donc plus particulièrement sur le cas de l'exploration des *arbres*. Comme évoqué ci-dessus, cette hypothèse se justifie aussi en pratique par le fait que la structure des batiments est, en général, quasiment arborescente.

La section 3.1 présente un état de l'art sur le problème de l'exploration. La section 3.2 redécouvre certaines preuves de NP-complétude de l'exploration d'arbres et présente des algorithmes d'exploration d'arbres sans contrainte. La

section 3.3 présente les travaux effectués sur les algorithmes d'exploration sous la contrainte du diamètre temporel. Enfin la section 3.4 s'intéresse à la variante *online* du problème et présente des résultats de simulations liés. L'ensemble du contenu des sections 3.3 et 3.4 est original et n'a pas encore été publié.

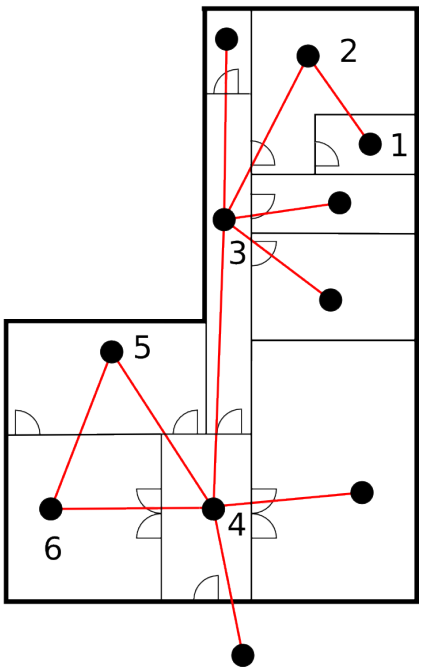


FIGURE 3.1 – Exemple d'appartement représenté à l'aide d'un graphe

3.1 Introduction

Dans ce qui suit, nous étudions le problème de l'exploration d'un graphe par des agents se déplaçant d'un sommet à l'autre dans le graphe. Un agent mobile est une entité dotée d'une mémoire qui est capable de se déplacer dans le graphe et de réaliser des actions. Ces actions peuvent être : se déplacer, communiquer avec un ou plusieurs agents, utiliser des marqueurs. Un marqueur est un objet propre à un agent qu'il peut déposer ou retirer d'un nœud. Suivant les modèles, un agent peut ne voir que ses marqueurs ou les voir tous.

L'exploration d'un graphe par un agent se déplaçant de sommet en sommet est un problème qui a fait l'objet de nombreuses recherches ces dernières décennies [31, 48]. La variante de ce problème où l'exploration est menée collectivement par plusieurs agents a suscité beaucoup d'intérêt récemment, motivé entre autres par le développement de la robotique et des capacités motrices des robots.

3.1.1 Résumé

Les articles [32, 51, 52] traitent du problème de l'exploration des graphes anonymes par un groupe d'agents. Un graphe anonyme est un graphe pour lequel les nœuds ne peuvent pas être différenciés par leurs identifiants. Plus précisément l'article [32] contient la preuve qu'il est impossible d'explorer de tels graphes sans l'usage de marqueurs. Dans cet article, un agent ne peut voir et déplacer que ses propres marqueurs. Dans [51] les auteurs présentent un algorithme en deux phases qui se répètent : explorations individuelles puis fusion des cartes. L'exploration d'un 'nouveau' nœud nécessite de vérifier qu'il est bien nouveau (il faut le distinguer des autres). Dans [52] les auteurs modifient l'algorithme de [51]. Ils permettent aux agents de ne pas chercher à déterminer immédiatement la nouveauté ou non d'un nœud découvert. Cette approche permet à un agent d'explorer le graphe un peu plus et peut permettre d'accélérer la distinction des nœuds (le fait de savoir s'il s'agit bien d'un nouveau nœud ou d'un nœud déjà visité).

Dans le domaine de l'exploration par un ensemble d'agents des graphes *avec identifiant* où les nœuds peuvent être différenciés, le problème peut être résolu sans marqueur. Néanmoins, les travaux [36, 37, 45, 46] contiennent des preuves que ce problème est NP-complet. Dans [37] les auteurs présentent un certain nombre d'algorithmes autour du problème de l'exploration de graphes à arêtes multiples entre nœuds pouvant être adaptés pour résoudre le problème de l'exploration d'un graphe. L'article [36] contient deux algorithmes différents, le premier où l'ensemble des agents peuvent communiquer sans aucune restriction et le deuxième où les agents ne communiquent que par des écritures et lectures des mémoires internes aux nœuds. Dans l'article [15], les auteurs présentent un algorithme à base de marqueurs où les marqueurs sont l'unique moyen de communication entre agents. Tous les marqueurs sont visibles par tous les agents. L'article [3] contient des algorithmes centralisés d'exploration de graphes par un nombre fini d'agents.

Les articles [48, 50] traitent du problème de l'exploration du "monde réel" par un robot muni de capteurs, le "monde réel" étant un espace en trois dimensions pourvu d'obstacles.

Nous décrivons maintenant plus en détail cet état de l'art

3.1.2 Graphes anonymes

Il est prouvé dans l'article [32] que l'exploration de graphes anonymes par des agents est impossible sans marqueur. Les papiers [32, 51, 52] présentent des algorithmes à base de marqueurs permettant d'explorer ces graphes. Dans [32] les auteurs présentent un algorithme explorant le graphe avec un seul agent et utilisant plusieurs marqueurs pour identifier les nouveaux nœuds explorés et

les arêtes. Les articles [51, 52] présentent des algorithmes multi-agents. Dans les deux algorithmes les agents ont une carte locale du graphe visité. Dans l'article [51], les robots alternent entre deux phases. La première phase consiste en l'exploration elle-même où les agents explorent les arêtes qu'ils considèrent comme n'étant pas explorées. À chaque fois qu'un agent arrive sur un nœud il doit détecter s'il s'agit d'un nouveau nœud où s'il est déjà venu dessus avant par l'intermédiaire d'une autre arête. Pour cela il utilise un marqueur qu'il place sur ce nouveau nœud, regarde le degré du nœud et visite de nouveau tous les nœuds qu'il connaît, ayant le même degré, pour regarder s'il trouve le marqueur et si c'est le cas il doit identifier les arêtes qui lui ont permis de visiter ce nœud. Cette identification des arêtes est nécessaire car la première fois que l'agent est arrivé sur le nœud il a fait une numérotation des arêtes en partant de celle d'où il est arrivé, lors de la seconde visite l'agent sait qu'il est forcément arrivé par une autre arête mais ne sait pas laquelle il doit alors l'identifier pour mettre à jour sa carte. Les agents disposent d'un temps T pour réaliser leur exploration. À la fin de ce temps l'ensemble des agents doit se retrouver sur un nœud qui a été choisi par eux tout au début de la phase parmi tous les nœuds déjà connus. À ce moment là, la deuxième phase de l'algorithme commence. Elle consiste en la fusion des cartes locales des agents en une seule carte identique pour tous les agents. Dans l'article [52], les auteurs améliorent les performances de l'algorithme précédent en permettant aux agents de repousser à plus tard certaines tâches visant à identifier correctement les nœuds. Pour chaque tâche de distinction l'agent va calculer le coût de la distinction et va décider s'il la remet à plus tard. Dans le cas où l'agent exécute une tache de distinction, il va recalculer le coût des distinctions qu'il n'a pas fait et reprendre une décision pour voir s'il les fait maintenant.

3.1.3 Graphes avec identifiant

Un graphe avec identifiant est un graphe où les nœuds sont immédiatement identifiables. Dans le papier [37] les auteurs présentent plusieurs algorithmes pouvant servir à l'exploration des graphes par un ensemble d'agents. Les problèmes étudiés tournent autour du problème des k voyageurs de commerce. Le problème initial du voyageur de commerce qui consiste à trouver pour un voyageur le chemin le plus court en terme de distance pour visiter toutes les villes positionner sur une carte. Le problème se représente à l'aide d'un graphe pondéré sur lequel chaque arête a une valeur équivalente à la distance entre les deux nœuds de ses extrémités. Il s'agit d'un problème NP-complet. Le problème des k voyageurs de commerce consiste à répartir des chemins de tailles équivalentes aux k voyageurs de commerce pour explorer l'ensemble du graphe. Les algorithmes présentés par les auteurs sont des algorithmes *onlines* (chaque agent prend ses propres décisions sans connaître toutes les informations). Les complexités maximales sont données et les auteurs prouvent la NP-complétude des

problèmes traités. Toutefois, il n'y a pas de contraintes sur le groupe d'agents comme nous l'étudions dans le cadre de cette thèse.

3.1.4 Arbres

Le problème de l'exploration des arbres par k agents est étudié depuis au moins 1996. Dans l'article [36], les auteurs présentent un algorithme d'exploration où les agents peuvent communiquer à tout moment quelle que soit leur distance. Dans l'article [3], les auteurs présentent plusieurs heuristiques centralisées pour résoudre ce problème. Les heuristiques cherchent à minimiser le déplacement des k agents avec une approximation de $(k-1)/(k+1)$. Dans l'article [15], les auteurs présentent un algorithme à base de marqueurs d'un seul type. Les agents peuvent poser autant de marqueurs qu'ils veulent pour s'informer les uns les autres mais c'est l'unique moyen de communication entre eux. Le temps d'exploration de cet algorithme est, au maximum, de deux fois le nombre d'arêtes.

L'article [36] présente aussi une preuve que le problème de l'exploration d'un arbre est NP-complet dans le cas général mais la preuve est basée sur le problème de 3-Partition [38]. La preuve fournie est assez compliquée et nous allons voir qu'il est possible d'en faire une plus simple à partir d'un autre problème.

Nous nous concentrons à partir de maintenant sur le cas particulier des arbres.

3.2 Exploration collective d'un arbre

Le problème de l'exploration collective d'un arbre est un cas particulier du problème de l'exploration collective d'un graphe. Dans cette section, nous établissons d'abord la NP-complétude de ce problème dans les arbres, en reconstituant des preuves dont on connait l'existence, mais qui n'étaient malheureusement pas accessibles jusqu'à présent. Nous discutons ensuite de la résolution de ce problème sans contrainte de diamètre temporel.

3.2.1 Définition du problème

Dans le cas général, le problème de l'exploration collective d'un graphe (puis d'un arbre) se définit comme suit :

Exploration d'un graphe par k-agents
Instance : Soit $G = (V, E)$ un graphe non-orienté. Soient $r \in V$ un nœud, $k > 0$ un nombre d'agents et B une borne entière positive sur le nombre d'arêtes qu'un agent parcourt lors de son circuit.

Question : Existe-t-il des circuits $C_1, \ldots, C_k$ tels que $\bigcup_{i=1}^{k} C_i = E$, chaque circuit commence et termine au nœud r et $\max\{|C_i| : i = 1, \ldots, k\} \leqslant B$?

L'article [46] mentionne l'existence de réductions à partir du problème MPS (Multi-Processor Scheduling), mais les preuves se trouvent dans le mémoire [45]. Nous avons contacté l'auteur pour en obtenir une copie mais il ne dispose plus de la version électronique. Le mémoire en version papier est néanmoins disponible dans la bibliothèque universitaire de l'université de West Virginia (États-Unis). N'ayant pas eu accès au mémoire mais supposant fortement que les réductions à partir de MPS sont plus simples, nous avons entrepris de retrouver ces résultats par nous-mêmes. Nos preuves sont présentées dans la section 3.2.2.

Dans les travaux qui suivent, le contexte est défini de cette façon : l'arbre n'est pas anonyme (chaque nœud a un identifiant unique) et les agents ont une connaissance complète de l'arbre. Pour des raisons techniques nous nous sommes intéressés à l'exploration d'arbres de hauteur $B/2$. Pour rappel, la hauteur de l'arbre est la distance maximale entre la racine et les feuilles. Nous y présentons plusieurs algorithmes qui améliorent progressivement le nombre de mouvements nécessaires pour explorer l'arbre.

Nous allons maintenant présenter les preuves de NP-complétude.

3.2.2 Preuves de NP-complétude

Dans cette sous-section, nous montrons que le problème de l'exploration collective d'un arbre quelconque est NP-complet, puis nous montrons que le problème reste NP-complet même lorsque le degré dans l'arbre est borné. En fait, nous montrons que même si l'arbre est binaire, le problème reste NP-complet.

Pour prouver que ces problèmes sont NP-complet nous allons montrer que le problème de l'ordonnancement de tâches sur un multi-processeur s'y réduit. Le problème de l'ordonnancement de tâches sur un multi-processeur est un problème NP-complet. Dans ce problème, il existe un ensemble de tâches de durées variables et on cherche à les répartir entre les processeurs en faisant en sorte que le temps d'exécution de chaque processeur ne dépasse pas une borne B donnée. Le problème se définit comme suit :

Ordonnancement de tâches sur un multi-processeur(MPS(B))
Instance : Soient J un ensemble de δ tâches où chaque tâche (job) $j \in J$ a une durée l_j, k le nombre de processeurs et B la borne entière positive sur le temps de travail maximal de chaque processeur.

Question : Existe-t-il une partition disjointe des tâches $J_1, \ldots, J_k$ où $\bigcup_{i=1}^{k} J_i =$

J et telle que $\max\{\sum_{j\in J_i} l_j : i = 1, \ldots, k\} \leqslant B$?

Moins formellement, cela revient à savoir s'il existe une partition des tâches en k ensembles, telle que la durée totale des tâches de chaque ensemble est inférieure à B.

3.2.2.1 Exploration d'un arbre de degré quelconque

Le problème de l'exploration d'un arbre quelconque par k agents (***k*-RE(B)**) est défini de la même façon que le problème de l'exploration d'un graphe. Le graphe exploré est alors un arbre où le degré de chaque nœud est libre (pas de degré maximal fixé).

L'idée de la preuve est de montrer que chaque circuit d'exploration de l'arbre représente la file de tâches d'un processeur et que si un circuit a une longueur $2B$ alors le temps d'exécution pour le processeur correspondant est B. Chaque tâche est représentée par une branche dans l'arbre généré, voir Figure 3.2. S'il existe un algorithme permettant de résoudre k-RE(B), cet algorithme peut donc être utilisé pour résoudre MPS. Nous détaillons maintenant la preuve.

Théorème 1. MPS(B) se réduit à k-RE(B_{RE}) en temps avec $B_{RE} = 2B$

Démonstration. Soit une instance du problème MPS(B). Nous construisons un arbre correspondant au problème de la façon suivante : pour chaque tâche $j \in J$ on construit la branche T_j partant de la racine $(0, 0)$ et contenant l_j nœuds nommés respectivement $(j, 1), \ldots, (j, l_j)$. Cette construction est en temps polynomial sur la somme des longueurs des tâches du problème MPS(B). La figure 3.2 représente la forme de l'arbre construit.

Si nous avons une solution positive au problème k-RE(B_{RE}), nous avons donc sur l'arbre précédemment construit un ensemble de k circuits C_i couvrants tel que $\max\{|C_i| : i = 1, \ldots, k\} \leqslant B_{RE}$. Chaque branche T_j de l'arbre est complètement visitée dans au moins l'un des circuits. Dans le cas où une branche a été complètement visitée dans plusieurs circuits, nous l'associons à un seul des circuits. Nous définissons J_i comme étant l'ensemble des tâches du problème MPS(B) pour lesquels la branche T_j a été complètement visitée dans le circuit C_i. Nous avons $\sum_{j\in J_i} l_j \leqslant B$ car chaque branche T_j est visitée deux fois, une fois lors de la descente depuis la racine jusqu'à la feuille de la branche et une deuxième lors de la remontée jusqu'à la racine. Nous avons ainsi construit une réponse positive au problème MPS(B).

Si nous avons une solution positive au problème MPS(B), c'est qu'il existe une partition des tâches en k ensembles J_i telle que $\max\{\sum_{j\in J_i} l_j, i : 1, \ldots, k\}$ $\leqslant B$. Nous pouvons à partir de ces k ensembles créer k circuits $C_i,\ i \in [1, k]$. Le circuit C_i est l'exploration des branches T_j associées à la tâche j appartenant

à l'ensemble J_i. Lors de l'exploration d'une branche T_j, ses arêtes sont visitées deux fois, on a $\max\{|C_i| : i = 1, \ldots, k\} \leqslant B_{RE}$. Nous avons alors construit une réponse positive au problème k-RE(B_{RE}).

Le problème MPS(B) se réduit donc au problème k-RE(B_{RE}) et cette réduction prend un temps polynomial.

□

Corollaire 1. k-RE(B) est NP-complet.

Démonstration. Nous avons montré avec le théorème 1 que le problème MPS (B) se réduit à k-RE(B_{RE}), hors MPS(B) est NP-difficile donc k-RE(B) est NP-difficile. Par ailleurs, il est facile de vérifier qu'une solution de k-RE(B) est valide en temps polynomial. Cela implique que k-RE(B) est dans la classe NP. k-RE(B) est à la fois dans la classe NP et NP-difficile donc il est NP-complet.
□

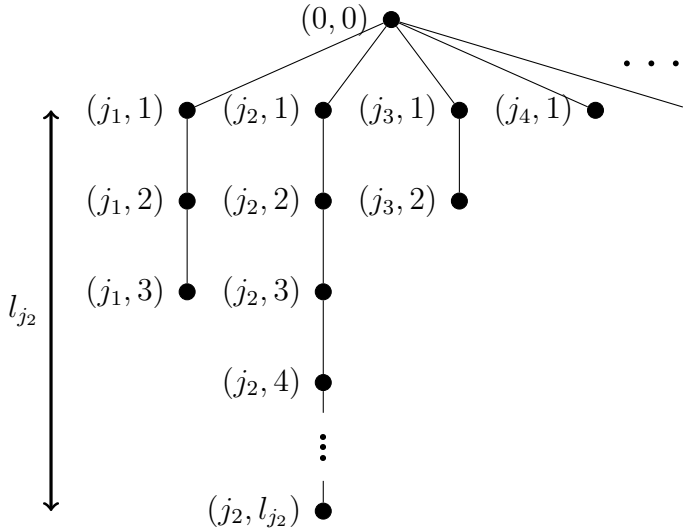


FIGURE 3.2 – Exemple de construction d'un arbre à partir d'un problème MPS

3.2.2.2 Exploration d'un arbre binaire

Le problème de l'exploration d'un arbre binaire par k agents (***k*-BRE(B)**) est défini de la même façon que le problème de l'exploration d'un graphe. Le graphe exploré est alors un arbre binaire où chaque nœud a un parent (hormis la racine) et au maximum deux fils. Nous montrons que même dans ces conditions le problème est NP-complet.

L'idée de la preuve est du même type que pour la preuve de NP-complétude du problème de l'exploration d'un arbre quelconque par k agents, mais la

construction est plus compliquée. Un arbre binaire contenant δ feuilles, toutes de même hauteur est d'abord créé. A chaque feuille de l'arbre on accroche une branche qui va représenter une tâche du problème MPS, voir Figure 3.3.

Théorème 2. MPS(B) se réduit à k-BRE(B_{BRE}) en temps polynomial avec $B_{BRE} = 2\delta\lceil\log\delta\rceil(B+1)$.

Démonstration. Soit une instance du problème MPS(B). Nous construisons l'arbre correspondant de la façon suivante : premièrement nous construisons un arbre binaire supérieur AS contenant δ feuilles toute placées à distance $\lceil\log\delta\rceil$ de la racine. La hauteur de l'arbre AS est donc $\lceil\log\delta\rceil$. Chacune des δ feuilles de l'arbre AS est complétée par une branche T_j correspondant à la tâche j. La branche T_j a pour longueur $l_j\delta\lceil\log\delta\rceil$. Nous obtenons alors un arbre binaire A. La valeur $\delta\lceil\log\delta\rceil$ vient du fait que nous voulons que le temps passé à se déplacer dans l'arbre supérieur AS soit équivalent au temps passé à explorer une branche de taille minimale. Cette valeur fait en sorte que le choix de répartition des branches entre les agents ne dépend pas de leurs positions relatives dans l'arbre A, un agent faisant au plus $2\delta\lceil\log\delta\rceil$ déplacements dans l'arbre supérieur AS s'il explore toutes les branches. La figure 3.3 représente la forme de l'arbre construit.

Si nous avons une solution positive au problème k-BRE(B_{BRE}), c'est que nous avons un ensemble de k circuits C_i couvrants tel que $\max\{|C_i| : i = 1,\ldots,k\} \leqslant B_{BRE}$. Chaque branche T_j de l'arbre T est complètement visitée dans au moins l'un des circuits. Si elle l'est dans plusieurs circuits, nous l'associons à un seul de ces circuits. Nous définissons J_{C_i} comme étant l'ensemble des tâches, du problème MPS(B), pour lesquels leur branche T_j est associée au circuit C_i.

Il est important de noter que le coût de l'exploration complète d'une branche T_j est de $2l_j\delta\lceil\log\delta\rceil$, c'est-à-dire deux fois la taille de la branche. La longueur de l'exploration des branches d'un circuit C_i est donc égale à $2\delta\lceil\log\delta\rceil(\sum_{j\in J_{C_i}} l_j)$. A cette longueur s'ajoute les déplacements dans l'arbre supérieur AS qui sont au minimum de 2. Supposons maintenant qu'il existe un circuit C_i tel que J_{C_i} vérifie $\sum_{j\in J_{C_i}} l_j \geqslant B+1$. Cela implique que le circuit C_i a une longueur d'exploration de branches supérieure ou égale à $2\delta\lceil\log\delta\rceil(B+1) = B_{BRE}$. En rajoutant les déplacements dans AS, le circuit a une longueur totale au minimum de $B_{BRE}+2$. C'est impossible et donc $\sum_{j\in J_{C_i}} l_j \leqslant B$. Nous avons ainsi construit une réponse positive au problème MPS(B).

Si nous avons une solution positive au problème MPS(B), c'est qu'il existe une partition des tâches en k ensembles J_i telle que $\max\{\sum_{j\in J_i} l_j, i : 1,\ldots,k\}$ $\leqslant B$. Nous définissons alors le circuit C_{T_j} comme le circuit qui part de la racine et explore la branche T_j associée à la tâche j. La longueur du circuit C_{T_j} est $2\log\delta(\delta l_j + 1)$. Nous allons maintenant construire k circuits $C_i,\ i \in [1,k]$. Le circuit C_i est l'agrégation des circuits C_{T_j} avec $j \in J_i$. La longueur d'un circuit

C_i est alors $2\log\delta(\delta B + |J_i|)$. De plus, nous avons $|J_i| \leqslant \delta$. Donc la longueur d'un circuit C_i est inférieure ou égale à $2\log\delta(\delta B + \delta) = B_{BRE}$. Nous avons ainsi construit une réponse positive au problème k-BRE(B_{BRE}).

Le problème MPS(B) se réduit donc au problème k-BRE(B_{BRE}) et cette réduction prend un temps polynomial.

□

Corollaire 2. k-BRE(B) est NP-complet.

Démonstration. Nous avons montré avec le théorème 2 que le problème MPS (B) se réduit à k-BRE(B_{BRE}), hors MPS(B) est NP-difficile donc k-BRE(B) est NP-difficile. Par ailleurs, il est facile de vérifier qu'une solution de k-BRE(B) est valide en temps polynomial. Cela implique que k-BRE(B) est dans la classe NP. k-BRE(B) est à la fois dans la classe NP et NP-difficile donc il est NP-complet.

□

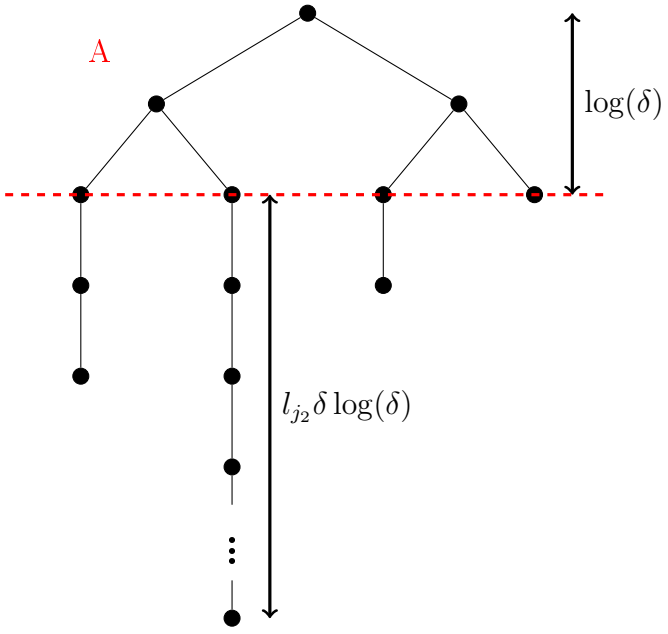


FIGURE 3.3 – Exemple de construction de l'arbre binaire à partir d'un problème MPS

Le problème de l'exploration d'arbre étant NP-complet, nous nous intéresserons surtout dans la suite à des heuristiques.

3.2.3 Heuristiques dans les explorations sans contrainte

Dans cette sous-section, nous étudions l'approche *offline* qui ne comporte pas de contrainte sur le groupe (les déplacements des agents sont complètement

libres) et la borne inférieure du surcoût que génère une exploration *online* (c'est à dire distribuée). Dans un premier temps nous présentons les différents critères d'évaluation d'un algorithme d'exploration puis nous présentons un algorithme *offline* sans contrainte. Enfin, nous présentons une borne inférieure existante sur le surcoût d'un algorithme *online* par rapport à un algorithme *offline*.

3.2.3.1 Critères d'évaluation d'un algorithme d'exploration

La qualité d'un algorithme est déterminée en fonction de plusieurs éléments : le temps de calcul, l'espace mémoire utilisé et la qualité du résultat (quel est l'écart entre le résultat obtenu et l'optimal). Dans notre cas la qualité du résultat dépend du temps que vont mettre les agents pour explorer le graphe. Il y a donc trois points de comparaison ; regardons maintenant ce qu'ils évaluent précisément que ce soit dans le cas *offline* ou dans le cas *online* :

- **la complexité en temps de calcul de l'algorithme** :
  - il s'agit du nombre d'opérations de calcul (pas de calcul) nécessaire pour obtenir la solution complète dans le cas *offline*.
  - il s'agit du nombre d'opérations de calcul pour déterminer le prochain mouvement dans le cas *online*.
- **la complexité en terme d'occupation mémoire** :
  - il s'agit de l'espace mémoire (nombre de bits) nécessaire pour calculer et stocker la solution dans le cas *offline*.
  - il s'agit de l'espace mémoire (nombre de bits) nécessaire pour calculer la prochaine étape dans le cas *online*.
- **la complexité en terme de temps d'exploration** (qualité du résultat) :

  Quel que soit le cas (*offline* ou *online*), il s'agit du temps nécessaire pour explorer l'arbre. Ce temps est souvent exprimé en terme de nombre d'arêtes traversées car il est généralement considéré qu'un agent traverse au plus une arête en une unité de temps.

Dans la suite de ces travaux nous nous sommes principalement intéressés à la complexité en terme de temps d'exploration.

3.2.3.2 Algorithme *offline* sans contrainte

Dans cette sous-section nous présentons une borne inférieure sur le temps d'exploration d'un arbre par un ensemble d'agents, nous y présentons ensuite un algorithme d'exploration *offline*.

Borne inférieure sur le temps d'exploration de l'algorithme *offline* sans contrainte

Il existe une borne inférieure "fine" pour le problème de l'exploration d'un graphe par k agents (cette borne est considérée comme folklore et nous n'avons pas pu remonter à son origine). Cette borne peut également être étendue au problème de l'exploration d'arbre. Les arguments sont les suivants. Au moins un agent doit se rendre à une distance du nœud de départ de l'ordre du diamètre D (au moins $D/2$). De plus, un tel algorithme explore n nœuds mais avec seulement k agents qui ne peuvent donc explorer collectivement qu'au plus k nœuds en une seule étape. Ainsi il faut au moins $\lceil n/k \rceil$ étapes pour explorer tous les nœuds. La borne inférieure est donc $\Omega(D + n/k)$.

Algorithme *offline* sans contrainte

Nous étudions maintenant un algorithme d'exploration d'un graphe présenté par Dominik Pająk dans sa thèse de doctorat [47]. Comme pour la borne inférieure il s'agit d'un algorithme considéré comme folklore et nous n'avons pas pu remonter à une origine antérieur à celle de la thèse de Dominik Pająk. Il s'agit d'un algorithme centralisé qui nécessite une connaissance complète du graphe et qui s'approche à un facteur constant près de l'optimal. Prenons un graphe $G = (V, E)$ ayant comme diamètre D. L'algorithme commence par calculer un arbre couvrant A du graphe G à l'aide d'un parcours en largeur. La construction de l'arbre A par un parcours en largeur (Breath First Search ou BFS) permet de s'assurer que la hauteur de l'arbre est égale à D. L'algorithme calcule ensuite un parcours en profondeur (Depth Fisrt Search ou DFS) sur l'arbre A, nommé DFS. Ce parcours a une longueur de $2n-2$ car chaque arête de l'arbre est parcourue deux fois et il y a $n-1$ arêtes. Ce parcours DFS est un parcours eulérien car il commence et se termine sur la racine de l'arbre. Une fois ce parcours calculé, l'algorithme suit les trois phases suivantes :

1 – Les agents se placent à équidistance dans le parcours DFS en un temps au plus D en partant de la racine. Chaque agent i prend position sur le nœud à distance $\lfloor \frac{2n-2}{k} \rfloor \times i$ du parcours DFS de A. Les agents se déplacent uniquement sur les arêtes de A. Étant donné que la hauteur de l'arbre A est D, le nombre de déplacement est bien borné par D.

2 – Chaque agent i, placé sur le nœud à la position $\lfloor \frac{2n-2}{k} \rfloor \times i$, suit le parcours DFS jusqu'à arriver sur le nœud à distance 1 du nœud de départ de l'agent $i+1$ et chaque agent se déplace au plus $\lceil \frac{2n-2}{k} \rceil$ fois.

3 – Pour finir, chaque agent retourne à la racine en au plus D pas.

Cet algorithme s'applique également à l'exploration d'arbre de profondeur D, le parcours en largeur n'a néanmoins pas besoin d'être exécuté car il s'agit déjà d'un arbre.

La figure 3.4 représente le positionnement des agents (sous la forme d'un rond, d'un losange et d'un carré) après le calcul du DFS et elle montre aussi la portion de DFS que fait chaque agent. Le coût de l'algorithme en temps est donc $O(2D + (2n-2)/k) = O(D + n/k)$. Nous avons vu que la borne

inférieure est $\Omega(D + n/k)$ dans la section précédente. Donc le temps pour réaliser le problème est $\Theta(D + n/k)$.

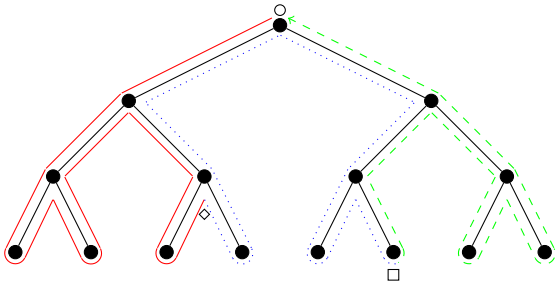
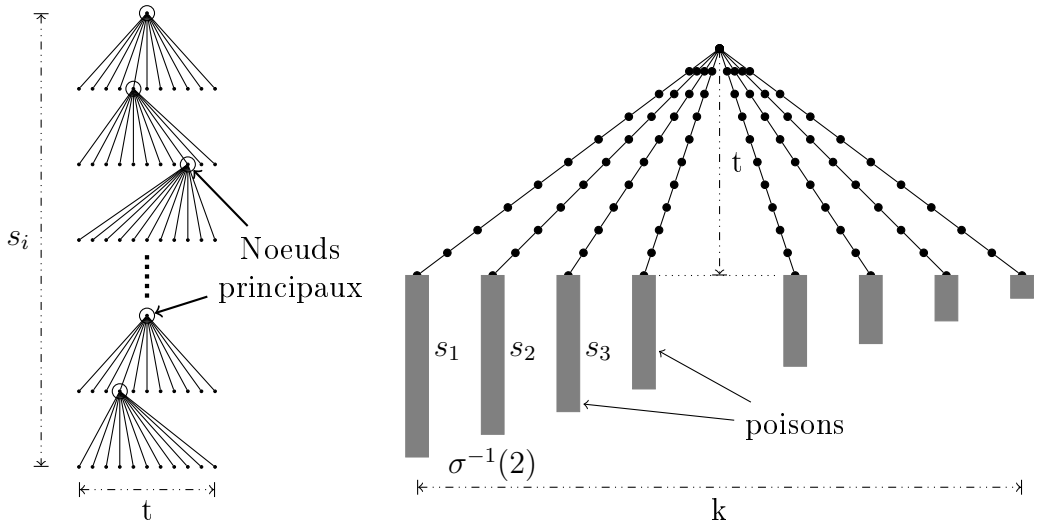


FIGURE 3.4 – Exemple d'exploration par l'algorithme *offline* sans contrainte avec trois agents, seule la partie concernant le parcours du DFS est représentée. Chaque type de pointillé représente le parcours d'un agent.

3.2.3.3 Algorithme *online* sans contrainte

Dans cette sous-section, nous présentons un résultat de [33] qui démontre le surcoût minimal d'un algorithme *online* par rapport à un algorithme *offline*. Plus précisément, les auteurs de [33] montrent qu'il existe certains graphes pour lesquels le surcoût en terme de temps d'un algorithme *online* par rapport à un algorithme *offline* est $\Omega(\log k / \log \log k)$, où k est le nombre d'agents. Ce surcoût est vrai même en considérant que les agents mobiles peuvent toujours communiquer directement quel que soit leur localisation. Dans l'article, les auteurs prouvent cette borne en construisant un **arbre méduse** (*jellyfish tree*). L'arbre méduse est représenté dans la figure 3.5 et est défini de la façon suivante.

Définition 3.1. Soit $t > k$ un entier et soit σ une permutation de l'ensemble 1 à k. L'**arbre méduse** $J(k, t, \sigma)$ est constitué de k sous-arbres (**tentacules**) numérotés de 1 à k et connectés à la racine (une tentacule par agent).
Chaque tentacule est constitué d'un **poison** constitué d'un ou plusieurs étages qui est attaché au bout d'une chaîne de taille t.
Chaque étage du poison est constitué de t nœuds tous rattachés à un unique nœud de l'étage précédent. Ce nœud est appelé **nœud principal** de l'étage et quel que soit l'algorithme utilisé il sera visité toujours après tous les autres nœuds de l'étage. Ceci est possible car les auteurs utilisent des algorithmes déterministes dont on peut déterminer à l'avance les mouvements et il est donc possible de modifier à l'avance la structure des poisons pour garantir que l'algorithme choisisse toujours en dernier ce nœud à chaque étage.
La **hauteur** (le nombre d'étages) du poison du tentacule $\sigma(i)$ est $s_{\sigma(i)} = \left\lceil \frac{k}{\log k} \cdot \frac{1}{i} \right\rceil$



(a) poison d'ID $\sigma^{-1}(i)$ (b) arbre méduse

FIGURE 3.5 – Exemple d'arbre méduse

Les auteurs de l'article utilisent cet arbre et surtout la permutation des branches pour forcer les algorithmes à perdre du temps dans les petites branches au détriment des plus grandes (les algorithmes sont déterministes). Avec une certaine permutation un algorithme déterministe ne va pas avoir exploré, après tr unités de temps, les poisons de taille supérieure à $(2\log k)^r$. Le temps total pour explorer la totalité de l'arbre est de la forme tR et l'algorithme aura exploré les nœuds à distance $t + (2\log k)^R$ de la racine. Sachant que la hauteur du plus grand poison est $\frac{k}{\log k}$ nous avons $(2\log k)^R \geqslant \frac{k}{\log k}$. Cela implique que $R = \Omega\left(\frac{\log k}{\log\log k}\right)$ et que le temps d'exploration de cet algorithme est donc de $\Omega\left(t\frac{\log k}{\log\log k}\right)$ étapes.

De plus, le nombre de nœuds de l'arbre est $kt + \sum_{i=1}^{k} t\left\lceil\frac{k}{i\log k}\right\rceil$ et d'après le calcul suivant il est en $O(kt)$.

$$\begin{aligned} kt + \sum_{i=1}^{k} t\left\lceil\frac{k}{i\log k}\right\rceil &\leqslant kt + kt + t\sum_{i=1}^{k}\frac{k}{i\log k} \\ &\leqslant 2kt + \frac{tk}{\log k}\sum_{i=1}^{k}\frac{1}{i} \\ &\leqslant 3kt \end{aligned}$$

Sachant cela, les auteurs montrent que le temps d'exploration de ce type

d'arbre est en $O(t)$ lorsqu'il est exploré par un algorithme *offline*. Le surcoût du passage d'un algorithme *offline* à un algorithme *online* est donc $\Omega\left(\frac{\log k}{\log\log k}\right)$.

3.3 Exploration avec contrainte du diamètre temporel

En plus de l'exploration d'arbre classique, des travaux commencent à apparaître où les auteurs rajoutent une contrainte sur le groupe d'agents. Dans l'article [30], les auteurs lui imposent un diamètre maximum : le diamètre d'un groupe d'agents est la distance maximale qui existe entre les agents dans le graphe. La distance entre deux agents est le nombre d'arêtes du chemin le plus court entre ces deux agents. Deux algorithmes sont présentés dans [30]. Dans le premier, les auteurs supposent qu'ils ont un nombre d'agents infini. L'algorithme peut faire apparaître de nouveaux agents lorsqu'il en a besoin et le sous graphe représenté par les nœuds sur lesquels sont les agents est connexe. Le deuxième algorithme limite le nombre d'agents au nombre de feuilles de l'arbre tout en bornant la distance maximale entre les agents. Néanmoins, cette limitation fait que le sous-graphe représenté par les nœuds sur lesquels sont les agents ne sera plus connexe. Le problème de la détection de la perte d'un agent n'est pas traité.

Nous étudions une contrainte différente, il s'agit du diamètre temporel du groupe d'agents, qui correspond au fait d'explorer un batîment par un groupe DTN d'agents (c.f. Chapitre 1). Tout le contenu de cette section est original et n'a pas encore été publié. La plupart des algorithmes présentés ici ont été évalués par simulation. Les résultats correspondants font l'objet d'une section dédiée (Section 3.4).

3.3.1 Diamètre temporel

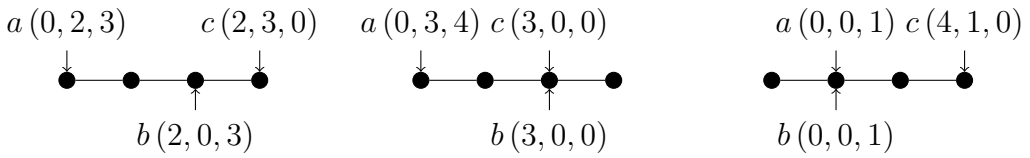
Le diamètre temporel est une notion issue du domaine des graphes dynamiques. La définition 1.21 du chapitre 1 définit ce qu'est le diamètre temporel et la définition 1.27 permet de définir ce qu'est un groupe de nœuds ayant un diamètre temporel borné.

Dans le cadre de ces travaux, la notion de diamètre temporel se transforme en une contrainte sur les déplacements des agents qui oblige chaque agent à être en contact direct ou indirect avec les autres agents dans un laps de temps borné par B. Plus précisément, il doit exister un trajet possible entre toue paire d'agents dans tout intervalle de durée B Si un agent a n'a pas entendu parlé d'un agent b dans un délais B, alors a considère b comme étant tombé en panne. Ainsi, cette contrainte permet de détecter la perte d'un agent tout en autorisant aux agents de s'éloigner ce qui est plus souple que de les forcer

à rester connexes en permanence.

Réalisation distribuée de cette contrainte

Dans l'article [24], les auteurs présentent une implémentation de ce problème dans le cas du temps continu (les "temporal views"). Dans notre cas nous proposons une version discrète simplifiée. Chaque agent a contient un ensemble de durées $\Lambda_a = \{d_1, d_2, \cdots, d_k\}$, où chaque d_i représente le temps écoulé depuis le départ d'un trajet en provenance de i. Au début de chaque étape chaque agent commence par mettre à jour l'ensemble de ses durées en ajoutant 1 à toutes les durées sauf la sienne qui reste à 0. Lorsque deux agents a et b ayant comme durées $\Lambda_a = \{d_{a_1}, d_{a_2}, \cdots, d_{a_k}\}$ et $\Lambda_b = \{d_{b_1}, d_{b_2}, \cdots, d_{b_k}\}$ sont sur le même nœud de l'arbre ils s'échangent leurs ensembles de durées et procèdent à une mise à jour : pour chaque agent i, d_i prend la valeur minimum entre d_{a_i} et d_{b_i}.



(a) à la fin de t_α (b) à la fin de $t_{\alpha+1}$ (c) à la fin de $t_{\alpha+2}$

FIGURE 3.6 – Exemple d'évolution des ensembles des durées des agents au cours du temps

La sous-section suivante présente quelques fonctions utilitaires nécessaire pour la suite. Nous introduisons après de nouveaux algorithmes d'exploration.

3.3.2 Exploration *offline* d'un arbre de hauteur h

Nous allons présenter dans cette section des algorithmes permettant de visiter un arbre de hauteur h par k agents. Le groupe d'agents a un diamètre temporel de taille B au maximum. Le nombre de feuilles de l'arbre est noté F dans la suite des explications. Dans le cas des algorithmes *offlines*, les déplacements des agents sont pré-calculés à l'avance et les mouvements sont ensuite exécutés par les agents eux-mêmes.

Les algorithmes *offlines* présentés dans les sections 3.3.2.1 et 3.3.2.2 utilisent le concept de *phase*. Une phase est une période de temps B telle que tous les agents sont positionnés, au début et à la fin de la phase, sur la racine de l'arbre. L'utilisation de phases est le moyen le plus simple de garantir la contrainte du diamètre temporel. Ces algorithmes ne fonctionnent que sur des arbres dont la hauteur h est inférieure à $B/2$ (aller/retour jusqu'à une feuille

au moins par un agent) Néanmoins, il existe une façon d'explorer des arbres ayant une hauteur plus grande présentée dans la section 3.3.2.3. La hauteur de l'arbre que l'on explore étant h, le temps qu'un agent met à visiter une des feuilles et à remonter à la racine est au maximum B.

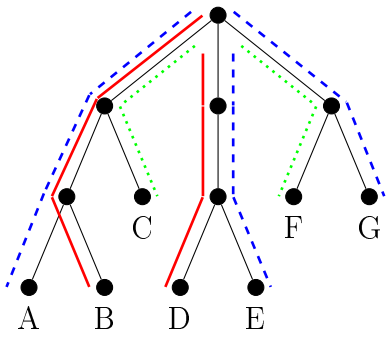
Les sous-sections suivantes présentent des algorithmes spécialisés ainsi qu'une analyse de leur efficacité.

3.3.2.1 Algorithme basé sur la répartition des feuilles

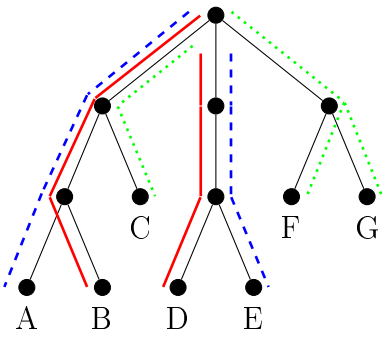
Le premier algorithme envisagé repose sur le fait que comme la hauteur de l'arbre est $h \leqslant B/2$, on sait qu'un agent a le temps de faire un aller-retour jusqu'à n'importe quelle feuille depuis la racine. Le temps de l'aller retour correspond alors au temps d'une phase. Pour chaque phase de l'algorithme, chaque agent se voit attribuer une feuille qu'il doit explorer. Le nombre de phase de l'algorithme est $\left\lceil \frac{F}{k} \right\rceil$, avec F étant le nombre de feuilles de l'arbre.

Il existe bien sûr des topologies pour lesquelles l'algorithme n'est pas optimal : citons par exemple le cas où un agent peut visiter tout un sous arbre (plusieurs feuilles) dans le temps donné et pas seulement une seule feuille. La figure 3.7(a) montre une répartition où l'agent vert aurait pu se voir attribuer tout le sous-arbre de droite (*i.e.* les feuilles F et G), qui est visité par l'agent bleu. Dans ce cas l'agent bleu n'aurait pas eu besoin d'une troisième phase. L'exploration se serait fait en seulement deux phases.

L'algorithme dans la section qui suit va chercher à améliorer l'attribution aux agents des parties de l'arbre à visiter.



(a) Exemple de répartition des feuilles de l'arbre en trois phases



(b) Exemple de partition de l'arbre en deux phases

FIGURE 3.7 – Exemples d'exploration d'un arbre pour trois agents avec $B = 6$

3.3.2.2 Algorithme basé sur une partition en sous-arbre

L'idée retenue pour cet algorithme est de pré-partitionner l'arbre de départ en sous-arbres qu'un agent a le temps de visiter en au maximum B mouve-

ments.Le point de rendez-vous à la fin de chaque phase reste la racine de l'arbre (comme précédemment).

Soit r la racine d'un arbre. Le temps de parcours de l'arbre (un DFS) par cet agent doit être inférieur ou égal au diamètre temporel. Si ce n'est pas le cas alors cet arbre doit être partitionné. Pour commencer, pour chaque fils s de r, un sous arbre A_s est créé. L'algorithme est ensuite appelé sur chacun des sous-arbres pour voir s'il faut de nouveau les partitionner. Il faut diminuer de deux le temps maximal d'exploration à chaque fois que l'on descend d'un niveau dans l'arbre. Cette diminution sert à compenser le temps qu'il faudra pour aller de la racine de l'arbre jusqu'à la racine du sous-arbre pour l'aller et le retour.

L'algorithme 2 est une implémentation récursive de cet algorithme : `current` représente la racine du sous arbre que l'on doit étudier ; `maxTime` est le budget de temps dont on dispose et `roots` est (à la fin) la liste des racines des sous-arbres. Cet algorithme utlise une fonciton décrite ci-dessous.

La fonction `durationOfExploration` implémente le calcul de la durée d'exploration par un seul agent du sous-arbre enraciné sur le nœud donné. La durée d'exploration d'un sous-arbre par un agent correspond à la durée qu'il faut à l'agent pour réaliser un DFS dans ce sous-arbre. Le DFS (Depth First Search ou parcours en profondeur) est un algorithme optimal d'exploration d'un arbre par un agent. La durée de parcours d'un DFS dans un arbre donné est deux fois le nombre d'arêtes. Il est aussi possible de construire un algorithme récursif pour calculer cette durée. Pour un nœud donné la durée d'exploration du sous arbre est la durée d'exploration de tous les sous-arbres de ses enfants plus deux fois le nombre d'enfants (cela correspond à la durée qu'il faut pour faire l'aller/-retour sur chacune des arêtes correspondantes). La durée d'exploration d'une feuille est de 0. Lors de l'exécution de l'algorithme durationOfExploration la durée d'exploration pour tous les sous-arbres du nœud donné est calculée. Il est donc possible d'exécuter cet algorithme une seul fois en partant de la racine de l'arbre et d'enregistrer les résultats au fur et à mesure pour chaque nœud une fois la valeur connue. Obtenir la durée d'exploration se résume alors à récupérer la valeur précédemment calculée.

```
1 procedure Partition(Node current, int maxTime, List<Node> roots)
2 |  if durationOfExploration(current) > maxTime then
3 |  |  foreach local child do
4 |  |  |  Partition(child,maxTime-2,roots)
5 |  else
6 |  |  add(roots, current)
```

Algorithme 2 : Fonction de partition de l'arbre en sous-arbres.

```
1 function durationOfExploration(Node current)
2 |   if isLeaf(current) then
3 |   |   return 0
4 |   duration = 0
5 |   foreach local child do
6 |   |   duration ← duration + durationOfExploration(child) + 2
7 |   return duration
```

Fonction durationOfExploration(*Node* `current`)

Une fois que la partition est effectuée, chaque sous-arbre est attribué à un agent. Soit P l'ensemble des sous-arbres de la partition, $|P|$ représente le nombre de ces sous-arbres. Chaque agent se voit attribuer x sous-arbres avec $\left\lfloor \frac{|P|}{k} \right\rfloor \leqslant x \leqslant \left\lceil \frac{|P|}{k} \right\rceil$. Une fois les sous-arbres attribués, l'exploration de l'arbre commence et se fait par phases, une phase ayant une durée B. Au début d'une phase tous les agents sont sur la racine de l'arbre. Pour un agent donné, s'il n'a pas visité tous les sous-arbres qui lui sont attribués alors il descend dans l'arbre jusqu'à la racine d'un des sous-arbres non visités dont il a la charge. Il visite ce sous-arbre en faisant un DFS, puis remonte à la racine de l'arbre et attend la fin de la phase. Si l'agent n'a plus de sous-arbres à visiter, il ne bouge pas de la racine durant la phase. La figure 3.7(b) montre un exemple de partition pour un arbre donné et on voit très clairement le gain en terme de temps d'exploration de cette approche grâce à la figure 3.7(a).

Cet algorithme, bien que meilleur que le premier, n'est pas toujours optimal, en terme de temps d'exploration. Par exemple la figure 3.8 représente un arbre qui va être partitionné en un grand nombre de sous-arbres. L'algorithme va créer un sous-arbre par fils de la racine et les attribuer ensuite aux k agents. Le nombre de phases pour explorer l'arbre est donc $\left\lceil \frac{F}{k} \right\rceil$, avec F étant le nombre de feuilles de l'arbre. Le problème vient du fait qu'un seul agent va utiliser tout le temps qu'on lui donne dans une phase pour explorer un des sous-arbres, celui en charge de la branche la plus longue. Le reste du temps cet agent utilise deux unités de temps pour explorer un sous-arbre puis attend jusqu'à la fin de la phase sur la racine. Les autres agents vont juste utiliser deux unités de temps pour explorer un sous-arbre puis attendre jusqu'à la fin de la phase sur la racine. Il aurait été plus judicieux de donner plusieurs sous-arbres à faire aux agents pendant une phase. Cette extension est laissée ouverte dans le présent document. Nous nous concentrons plutôt sur le cas où la hauteur de l'arbre ne permet pas de revenir sur la racine à chaque intervalle de largeur B.

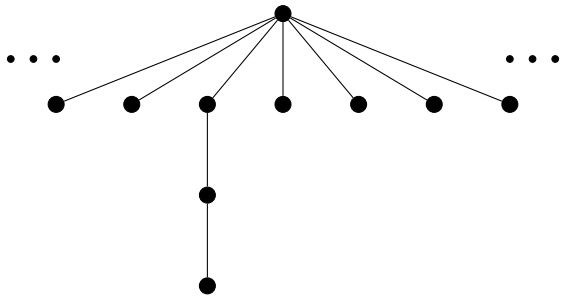


FIGURE 3.8 – Exemple de graphe pour lequel l'algorithme génère une partition non optimale, $B = 6$

3.3.2.3 Extension aux arbres de hauteur quelconque

Comme dit précédemment, les algorithmes présentés dans les deux sections précédentes fonctionnent sur des arbres de hauteur $h = B/2$. Néanmoins, pour les arbres de hauteur $H > h$, il existe une méthode simple permettant tout de même d'utiliser ces algorithmes. Pour cela il suffit de faire un DFS sur l'arbre de hauteur H avec tous les agents. Lorsque les agents arrivent à un nœud dont la hauteur du sous-arbre est inférieure ou égale à $h = B/2$, ils se séparent et explorent le sous-arbre avec un des algorithmes spécialisés précédemment présentés. Lorsque l'exploration du sous-arbre est terminée, les agents reprennent le parcours du DFS en ignorant les nœuds du sous-arbre exploré. Cela implique que la hauteur du sous-arbre est possiblement virtuellement modifiée pour le parent du nœud dont le sous-arbre vient d'être exploré. S'il y a une modification de la hauteur, les hauteurs des sous-arbres des ascendants peuvent également avoir diminué. Les agents en reprenant le DFS vont remonter sur le parent du nœud du sous-arbre exploré et vont étudier sa hauteur. Si sa hauteur est supérieur à h alors les agents continuent le DFS tous ensembles en descendant sur l'un des enfants non explorés du nœud. Dans le cas contraire, les agents remontent dans l'arbre de parent en parent jusqu'à trouver le nœud ayant la hauteur la plus près de h tout en lui étant inférieur ou égale. Une fois qu'ils y sont, ils se séparent et recommencent à explorer avec un des algorithmes spécialisés. La figure 3.9 montre un exemple d'exploration d'un tel arbre. Le coût d'une telle exploration est dépendant de la structure de l'arbre et va être dépendant du nombre de feuille de l'arbre. L'expression exacte de ce coût en fonction de divers paramètres est laissée ouverte.

3.3.3 Algorithme exact pour l'exploration d'arbre *offline* avec diamètre temporel borné

Dans cette section, nous présentons un algorithme qui trouve la solution optimale au problème de l'exploration d'un arbre A par k agents qui respectent

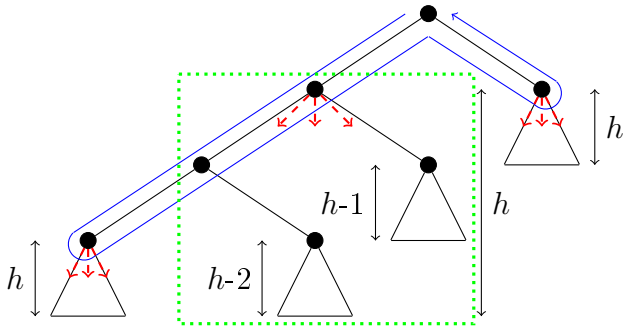


FIGURE 3.9 – Exemple d'exploration d'un arbre de hauteur supérieure à la moitié du diamètre temporel. La flèche continue (bleu) représente le parcours du DFS. Les flèches discontinues (rouges) représentent les parties où les agents se séparent pour explorer un sous-arbre. Le cadre en pointillé (vert) représente l'un des sous-arbres que les agents explorent en se séparant.

la contrainte du diamètre temporel. Le problème étant suspecté NP-difficile (même si nous ne l'avons pas prouvé dans cette variante), le moyen retenu pour trouver la solution optimale est d'étudier toutes les explorations possibles en retenant la meilleure en terme de temps d'exploration.

Nous représentons l'ensemble des explorations possibles par un méta-arbre. Chaque méta-nœud du méta-arbre représente l'état global d'une exploration à un moment donné, c'est-à-dire la position des agents dans l'arbre A, les nœuds visités et pour chaque agent une liste de durées tels que chaque durée est associée à un agent et correspond au temps qu'il s'est écoulé depuis le dernier échange avec cet agent. À chaque fois que plusieurs agents sont sur le même nœud, les durées de séparation sont toutes mises à jour. Cela revient à adapter la règle de synchronisation distribuée du diamètre temporel présentée dans la section 3.3.1 (la figure 3.6 représente ce qui se passe au niveau des durées).

La méta-racine est la racine du méta-arbre, elle représente le début de toute exploration, où tous les agents sont sur la racine de l'arbre A. Un méta-nœud v est un enfant du méta-nœud u si et seulement si les agents se sont déplacés au plus d'un nœud dans l'arbre A et il y a au moins un agent qui s'est déplacé. La figure 3.11 contient une représentation partielle du méta-arbre pour l'arbre de la figure 3.10.

Algorithme BFS

L'exploration de toutes les solutions se fait alors en réalisant un BFS sur le méta-arbre jusqu'à trouver une solution respectant la contrainte du diamètre temporel. Uns solution de l'exploration est une chaîne partant de la racine du méta-arbre et arrivant sur un méta-nœud où tous les nœuds de l'arbre à explorer sont explorés (le retour à la racine n'est pas un problème ici). La longueur

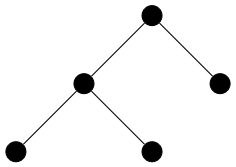


FIGURE 3.10 – Exemple simple d'arbre à explorer

de la chaîne donne alors le temps d'exploration de l'arbre pour cette solution (sans inclure le retour). Il est important de noter que, de par sa construction, le méta-arbre est un graphe infini, les agents pouvant faire des allers et retours en permanence. Toutefois le BFS permet de parcourir le meta-arbre étage par étage jusqu'à trouver une solution. Si une solution est trouvé à une certaine profondeur et qu'aucune autre solution n'a été trouvé avant alors, de fait, cette solution est une solution optimale en terme de temps d'exploration.

Complexité de l'algorithme

La complexité en temps du DFS est de deux fois le nombre de méta-nœuds du méta-arbre. Pour chaque méta-nœud du méta-arbre, son nombre d'enfants est paramétré par le nombre d'agents explorant l'arbre et par le degré max d_{max} de l'arbre A : chaque méta-nœud de l'arbre a par construction au plus $(d_{max}+1)^k - 1$ descendants, chaque agent pouvant se déplacer sur un des voisins du nœud sur lequel il se trouve ou rester sur place (le -1 s'explique du fait qu'au moins un agent doit se déplacer). Nous savons qu'il existe une solution pour explorer un arbre avec un seul agent qui est le parcours DFS de l'arbre. Le temps d'exploration d'un DFS est $2(n-1)$, le BFS sur le méta-arbre trouvera donc forcément une solution à la profondeur $2(n-1)$ car ce sera alors la solution de faire un DFS dans l'arbre. Le nombre de méta-nœuds visités est donc au plus de $((d_{max}+1)^k)^{2(n-1)} = O^*((d_{max}+1)^{2kn})$. Le temps maximal d'exploration est donc aussi de $O^*((d_{max}+1)^{2kn})$.

Optimisations de l'algorithme

L'algorithme précédant peut être optimisé en n'explorant pas certaines parties du méta-arbre et en coupant les branches du méta-arbre. En effet, si sur un méta-nœud la contrainte du diamètre temporel n'est plus respectée par les agents, alors il n'est alors pas nécessaire de continuer à explorer le sous-arbre de ce méta-nœud. La détection du non respect de cette contrainte se fait à l'aide des durées de séparation (incluses dans les états). Précisément, si l'une des durées dépasse le maximum autorisé par le diamètre temporel, alors la contrainte n'est pas vérifiée sur le méta-nœud et la branche peut être coupée.

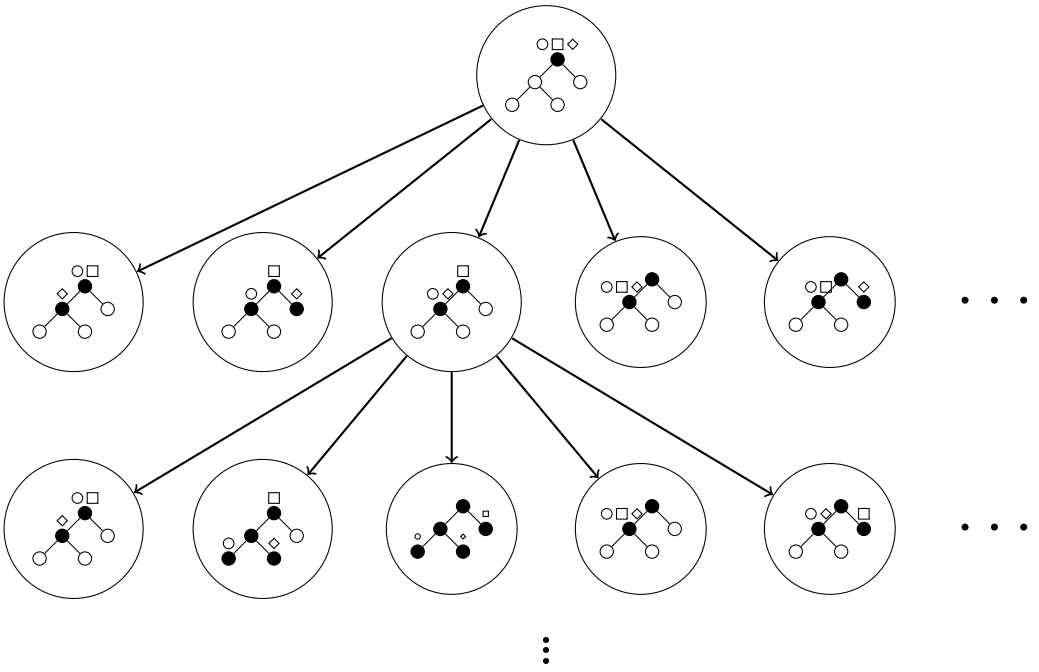


FIGURE 3.11 – Exemple partiel d'un méta-arbre pour l'exploration de l'arbre de la figure 3.10 par trois agents représentés par des pastilles en forme de rond, carré et losange. Les nœuds noirs représentent les nœuds visités par les agents. Seule une partie des 26 descendants de la méta-racine est dessinée. Les durées de séparation ne sont pas représentées sur ce graphe.

3.3.4 Algorithme d'exploration *online* avec diamètre temporel borné

Nous présentons dans cette section un algorithme *online* (distribué) d'exploration générique d'un arbre A par un groupe de k agents qui doit avoir un diamètre temporel inférieur ou égal B. Cet algorithme est générique car il fonctionne quel que soit l'arbre A, quel que soit le nombre d'agents k et quel que soit le diamètre temporel B.

Cet algorithme étant *online*, les agents ne connaissent pas l'arbre et vont le découvrir au fur et à mesure de l'exploration. Il est important de noter que chaque agent a sa propre carte de l'arbre. De plus, les agents ont seulement la possibilité de communiquer lorsqu'ils sont sur le même nœud de l'arbre et ils se déplacent de manière synchrone : les déplacements de tous les agents ont lieu en même temps et chaque agent se déplace à travers au plus une arête à chaque étape (il peut décider de rester sur place, s'il le veut). On suppose que B et k sont donnés.

3.3.4.1 Modèle et variables de l'algorithme

Le modèle sur lequel est basé notre algorithme est le suivant. Le système est synchrone, tous les agents s'activent en même temps. Chaque nœud de l'arbre a trois états possibles dans la carte d'un agent :

— *visité* : l'agent sait qu'un agent est allé sur ce nœud.
— *exploré* : l'agent sait que le nœud ainsi que tout son sous-arbre a été visité.
— *none* : l'agent connaît l'existence de ce nœud non visité.

On considère que l'arbre est totalement exploré lorsque tous les nœuds sont dans l'état *exploré* dans les cartes de tous les agents et que tous les agents sont sur la racine.

Les agents commencent par une phase d'expansion du groupe dans laquelle ils commencent tous sur le même nœud appelé centre de l'extension et vont explorer les nœuds en se séparant ; ils vont ainsi connaître la topologie de l'arbre. Au bout d'un temps égal au diamètre temporel tous les agents se retrouvent sur le centre de l'expansion et ils échangent les cartes obtenues pour avoir une vision globale de l'arbre. Les nœuds choisissent alors un nouveau nœud comme centre de l'extension et se déplacent jusqu'à lui. Une fois sur ce nœud les agents recommencent la phase d'expansion décrite ci-dessus. Ils alternent les deux phases jusqu'à ce que l'arbre soit complètement exploré.

Chaque agent possède un ensemble de variables :

— `current` qui contient le nœud sur lequel se trouve l'agent, cette variable est initialisée avec la racine de l'arbre.
— `phase` qui contient l'information sur la phase actuelle de l'agent soit *exploration*, soit *selectionCenter* ; sa valeur initiale est *exploration*.
— `time` qui est le compteur du nombre de déplacements après la séparation du groupe. Cette variable est utilisée pour s'assurer que la contrainte du diamètre temporel est vérifiée. Ce compteur est initialisé à 0.
— `center` qui contient le nœud "centre de l'exploration". Comme pour la variable `current`, elle est initialisée avec la racine de l'arbre.
— `newCenter` qui contient le nœud qui est choisi comme nouveau centre de la phase d'expansion. Elle est initialisée avec $\perp$.
— `tree` qui contient l'état de l'arbre qui est exploré, c'est-à-dire s'il est totalement exploré ou pas. Initialement, sa valeur est *notExplored*.

Chaque agent possède également une carte locale de l'arbre. Cette carte contient l'ensemble des nœuds dont il a connaissance et les arêtes connues entre ces nœuds et vers les nœuds inconnus. Cette carte est mise à jour au fur et à mesure de l'exploration.

Chaque étape est composé de trois parties : la première partie est l'envoi de messages, la deuxième la réception des messages et enfin la troisième est la

partie dédiée aux calculs et décisions.

Lors de l'envoi de message, chaque agent envoie, à tous les autres agents situés sur le même nœud que lui, son identifiant, sa carte locale et ses durées de séparation.

Lors de la partie de réception des messages, chaque agent met à jour les durées de séparation en fonction de celles reçues en prenant les valeurs les plus petites (*cf.* section 3.3.1, page 56). Il met aussi à jour sa carte locale en fonction des cartes qu'il a reçu de la part des autres agents.

La partie dédiée aux calculs est décrite dans la section suivante.

3.3.4.2 Description de l'algorithme

Nous présentons maintenant le fonctionnement général de l'algorithme. Le détail des deux phases de l'algorithme (Algorithme 3, page 68) est présenté ci-dessous.

La phase d'expansion

Elle correspond aux lignes 2 à 21. Lors de la première étape de la phase d'expansion tous les agents sont forcément sur le même nœud qui est le centre de l'expansion, noté *center* et tous les agents ont la même carte de l'arbre. Au début de l'algorithme la carte contient seulement la racine de l'arbre A sur laquelle sont les agents et ses arêtes sortantes.

Au début de chaque étape de la phase d'expansion, avant de faire un mouvement, chaque agent regarde les messages qu'il a reçus dans cette étape et met à jour sa carte s'il a reçu des informations provenant d'autres agents (ligne 3). Les agents se séparent alors en allant explorer les sous-arbres du centre de l'expansion *center*. S'il y a plusieurs agents sur un nœud, ils se séparent pour explorer chacun un sous-arbre non exploré du nœud sur lequel ils sont. La ligne 13 représente le choix du nœud enfant à explorer ; la fonction `chooseUnexploredChild` retourne un enfant du nœud `current` non exploré s'il en existe un. Le choix de l'enfant s'il y en a plusieurs se fait grâce aux informations reçues dans les messages tels que les identifiants des agents positionnés sur le nœud et l'état de la carte locale après la mise à jour à partir des cartes des autres agents. La version la plus simple consiste à distribuer les agents sur les enfants non explorés en fonction de l'ordre de leurs identifiants.

Les agents vont descendre dans l'arbre tant qu'ils ont le temps de remonter jusqu'à *center* avant la fin de la phase d'expansion. La fonction `hasTime` ligne 12 retourne *true* dans le cas où l'agent a assez de temps pour continuer l'exploration avant de retourner au centre. Cette fonction calcule la durée qu'il faudrait à l'agent pour aller visiter un fils de plus et remonter sur le centre de l'expansion et compare cette durée avec la durée de temps qu'il lui reste

avant de devoir être sur le centre ($B - \texttt{time}$). Si l'agent a le temps de visiter un nœud de plus alors la fonction retourne *true*.

Si tout le sous-arbre du nœud sur lequel se trouve l'agent est exploré et que ce nœud n'est pas le centre de l'expansion (*center*), l'agent remonte sur le nœud parent et reprend son exploration. Si l'agent n'a plus le temps d'explorer l'arbre alors il retourne sur *center*. Au bout de B unités de temps, tous les agents sont de nouveau sur le nœud *center*.

Dans le cas où le centre de l'expansion *center* n'a qu'un seul enfant non exploré c' et que les agents sont au début de la phase d'expansion (ligne 8), tous les agents se déplacent sur cet enfant c' qui devient le centre de l'expansion de la nouvelle phase d'expansion (on arrête prématurément la phase d'expansion courante et on en commence une nouvelle).

Si, au cours de la phase d'expansion, tous les agents se retrouvent sur le même nœud alors ils arrêtent la phase d'expansion et passent à la phase de sélection du nouveau centre (ligne 6).

La phase de sélection du nouveau nœud centre de l'expansion

Elle correspond aux lignes 22 à 33. Durant cette phase les agents ne vont pas modifier leur carte car tous les déplacements se font dans la partie de l'arbre déjà explorée.

Le nouveau centre de l'expansion est un nœud (s'il existe) du sous-arbre du nœud centre de l'expansion précédente (*center*) vérifiant les propriétés suivantes : il doit être visité mais pas exploré. Si plusieurs nœuds sont éligibles (plusieurs noeuds du sous arbres sont visités mais pas explorés) le choix du nœud peut dépendre des options/critères tels que la profondeur, etc. Si aucun nœud n'est éligible (aucun nœud visité mais pas exploré) dans le sous-arbre de *center* alors on choisit un nœud éligible dans l'arbre A s'il existe. Dans ce cas, le choix de ce nœud peut aussi être conditionné par différents critères tels que la distance par rapport à *center* et la profondeur de ce nœud. Si aucun nœud n'est éligible alors tous les nœuds sont visités et explorés et les agents vont se repositionner sur la racine s'il ne le sont pas déjà. Notons que la distance à la racine peut être supérieure à B, mais puisqu'à ce stade tous les agents sont situés sur le même nœud, ils peuvent rester groupés tout le long de ce retour.

La ligne 25 représente le choix du nouveau centre. La fonction `chooseNewCenter` retourne le nouveau centre selon des critères qu'on lui fixe et dans le cas où il n'y a pas de centre éligible, elle retourne la racine de l'arbre A.

```
1  while tree ≠ explored do
2      if phase == exploration then
3          mergeNeighborsMaps()
4          if time == B then
5              phase ← selectionCenter
6          else if allHere() AND time > 0 then
7              phase ← selectionCenter
8          else if oneChild() AND current == center AND time == 0 then
9              current ← getUnexploredChild()
10             center ← current
11         else
12             if hasTime() then
13                 child ← chooseUnexploredChild()
14                 if child ≠ null then
15                     current ← child
16                 else if current ≠ center then
17                     current ← parent(current)
18             else
19                 if current ≠ center then
20                     current ← parent(current)
21             time ← time + 1
22     if phase == selectionCenter then
23         if time ≠ 0 then
24             time ← 0
25             newCenter ← chooseNewCenter()
26         if current == newCenter then
27             if current == racine AND allChildrenExplored() then
28                 tree ← explored
29             else
30                 center ← current
31                 phase ← exploration
32         else
33             current ← nextHopToTarget(newCenter)
```

Algorithme 3 : Algorithme d'exploration sur un agent

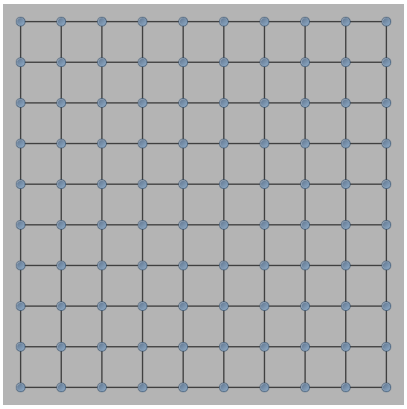
3.4 Simulation

Nous présentons ici les résultats de simulation correspondants aux algorithmes présentés dans la section précédente. Ces simulations ont été réalisées en java à l'aide de la bibliothèque JBotSim [18], qui permet de simuler des graphes statiques et/ou des graphes dynamique de façon simple et d'implémenter rapidement toutes sortes d'algorithmes distribués. Le code correspondant à ces simulations peut être fourni sur demande.

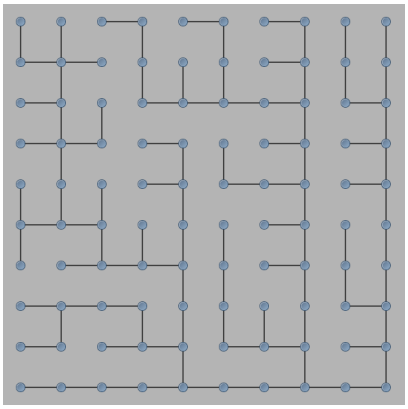
Pour chaque jeu de paramètre étudié les simulations ont été répétées plusieurs fois (le nombre exact sera donné lorsque nous parlerons de chaque simulation). De plus, la valeur pour chaque point affiché sur les courbes est la moyenne des résultats de toutes les exécutions, pour chaque point nous affichons aussi l'intervalle de confiance à 95%.

3.4.1 Génération d'arbres aléatoires

Chaque exécution de notre algorithme s'est faite sur un arbre différent généré soit à partir d'une grille 3D dont la taille est fixée à l'avance, soit sans structure sous-jacente (pour faire varier les degrés librement).



(a) Grille 2D de 10 lignes et 10 colonnes.



(b) Arbre généré à partir de la grille.

FIGURE 3.12 – Exemple de génération d'un arbre à partir d'une grille, ici en 2D seulement avec 100 nœuds (nos simulations ont utilisé une variante 3D de ce principe).

Le cas où l'arbre est un sous-graphe d'une grille 3D rend compte de contrainte physique réelles (par exemple les salles d'un bâtiment). Dans ce cas la génération de ce type d'arbre se passe comme suit, on choisit d'abord un nœud aléatoire dans la grille que l'on met dans l'arbre. Ensuite tant qu'il reste des nœuds de la grille qui ne sont pas dans l'arbre on sélection un nœud de la grille

aléatoirement dans la frontière de l'arbre déjà créé et on l'ajoute à l'arbre (ainsi que l'arête correspondante). La figure 3.12 montre un exemple de génération dans le cadre d'une grille 2D à 100 nœuds répartis sur 10 colonnes et 10 lignes. Cette figure n'a pour but que d'illustrer le principe, nos simulations ayant utilisé des grilles 3D.

Comme évoqué ci-dessus, nous générons aussi des arbres sans structure sous-jacente, afin de pouvoir faire varier les degrés librement. Dans ce cas, nous avons directement généré des arbres qui ont n sommets et un degré max Δ. Un tel arbre est construit par un parcours en largeur qui détermine au fur et à mesure le nombre d'enfants de chaque nœud : son nombre d'enfants est tiré aléatoirement entre 0 et Δ. Dans le cas où il n'y a plus qu'un nœud à traiter dans le parcourt en largeur et que le nombre de nœuds voulu n'est pas atteint, le nombre tiré sera forcément supérieur à zero. Cette génération nous permet d'avoir des arbres ayant des nœuds de degré au plus Δ.

```
1  procedure generateTree(Node root, int maxDegree, int nbNodesMax)
2      nbNodes ← 1 ;
3      degree ← randInt(1,maxDegree +1) ;
4      for val ∈ {1, ··· , degree} do
5          neighbor ← createNewNode() ;
6          createLink(root, neighbor) ;
7          nbNodes ← nbNodes +1 ;
8          queue.add(neighbor) ;
9      while not queue.isEmpty() do
10         node ← queue.pop() ;
11         if nbNodes < nbNodesMax then
12             if queue.isEmpty() then
13                 degree ← randInt(1,maxDegree) ;
14             else
15                 degree ← randInt(0,maxDegree) ;
16             for val ∈ {1, ··· , degree} do
17                 neighbor ← createNewNode() ;
18                 createLink(node, neighbor) ;
19                 nbNodes ← nbNodes +1 ;
20                 queue.add(neighbor) ;
21                 if nbNodes == nbNodesMax then
22                     break ;
```

Algorithme 4 : Algorithme de génération d'arbres sans structure sous-jacente

L'algorithme 4 est l'algorithme utilisé pour cette dernière forme de généra-

tion de l'arbre. La variable `queue` est une file de donnée dont la fonction `add` ajoute la valeur donnée à la fin et la fonction `pop` récupère la première valeur et la supprime de la file. La fonction `randInt(min, max)` retourne un entier compris dans l'ensemble $[min, max[$. Le pseudo-code des lignes 3-8 concerne la génération des fils de la racine, le nombre aléatoire est tiré entre 1 et le degré max compris car la racine doit au moins avoir un fils et au plus autant que le degré max. Pour les autres nœuds de l'arbre le nombre d'enfants sera un nombre aléatoire compris entre 0 et et le degré max non compris car ces nœuds ont déjà une connexion avec un nœud père.

3.4.2 Impact du diamètre temporel sur le temps d'exploration d'un arbre

Nous avons étudié l'impact du diamètre temporel sur le temps d'exploration d'un arbre donné avec notre algorithme *online*. Pour chacun des jeux de paramètres que l'on va présenter ci-dessous les simulations on été exécutées 1000 fois.

Pour ces simulations l'arbre a été généré aléatoirement au sein d'une grille 3D de taille 10x10x10, c'est-à-dire de 1000 nœuds.

Pour un groupe de taille donnée nous avons étudié l'impact du diamètre temporel. Quatre tailles ont été étudiées pour les groupes d'agents : 2, 4, 8, 16. Pour chaque taille, nous avons fait varier le diamètre temporel de 10 à 100 par pas de 10.

La figure 3.13 représente le temps d'exploration moyen de l'algorithme. L'intervalle de confiance à 95% des valeurs est aussi représenté mais difficilement visible du fait qu'il est très proche de ces valeurs. Notons que le temps optimal d'exploration avec un seul agent est de 1998 étapes ; il s'agit du temps nécessaire pour parcourir toutes les arêtes de l'arbre à l'aide d'un DFS. La hauteur moyenne de l'arbre généré est pour, toutes les simulations de cette section, de 33.31 avec 95% des valeurs comprises entre 32.53 et 34.09.

La figure 3.13 montre que le diamètre temporel a un impact non négligeable sur le temps d'exploration des arbres. Plus le diamètre temporel est grand, plus le temps d'exploration diminue. Cela s'explique par le fait que les agents ont plus de temps pour accumuler de l'information et qu'ils passent moins de temps à se regrouper.

Néanmoins il est important de noter qu'augmenter le nombre d'agents n'apporte pas un gain important. Lorsqu'on multiplie le nombre d'agents par deux, le temps d'exploration est réduit au maximum d'un tiers. Notez aussi qu'à chaque fois que le nombre d'agents est multiplié par deux le gain de temps obtenu diminue. Par exemple dans le cas où le diamètre temporel est 100, le gain du passage de 2 à 4 agents est $\approx 33\%$ alors que le gain du passage de 4 à 8

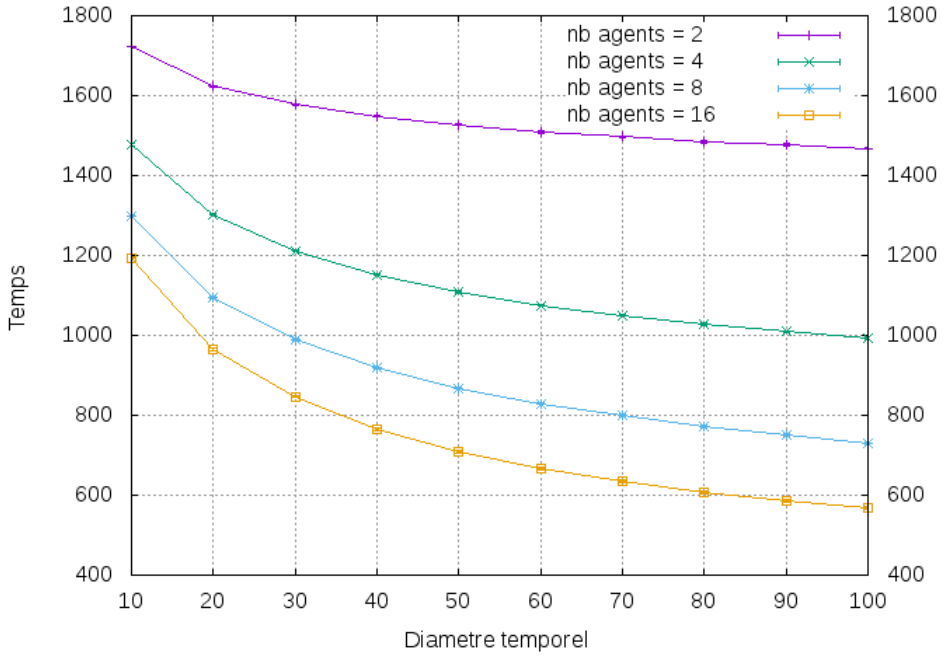


FIGURE 3.13 – Variation du temps d'exploration moyen de l'algorithme *online* en fonction du diamètre temporel et du nombre d'agents.

agents est $\approx 27\%$. La table 3.1 contient les valeurs moyennes du temps d'exécution de l'algorithme d'exploration pour les diamètres temporels extrêmes (10 et 100) et pour toutes les tailles de groupes étudiées.

Diamètre temporel / **Nombre d'agents**	10	100
2	1722	1468
4	1478	993
8	1299	730
16	1194	569

TABLE 3.1 – Temps d'exécution moyen en nombre d'étapes de l'algorithme d'exploration *online* en fonction du diamètre temporel et du nombre d'agents.

3.4.3 Impact du degré moyen sur le temps d'exploration d'un arbre

Nous nous sommes intéressés à l'impact du degré moyen des nœuds sur le temps d'exploration. Comme évoqué ci dessus, les arbres utilisés pour nos

simulations ont été générés de manière aléatoire sur des grilles à 3 dimensions ce qui permet aux nœuds d'avoir un degré maximal égal à 6.

Pour augmenter le degré maximal de nos arbres à plus de 6, nous avons voulu créer des grilles de plus grande dimension $R > 3$. Lors des tests de génération des arbres, nous nous sommes rendus compte que cette construction prend trop de temps lorsque le nombre de nœuds dans les grilles à R dimensions dépasse 100 000, le temps de génération d'un seul arbre dépassant une heure.

Suite à ce constat, nous avons décidé de construire directement les arbres sans passer par une structure sous-jacente. Nous utilisons alors l'algorithme présenté précédemment dans la section 3.4.1. Chaque arbre est généré aléatoirement et pour ne pas avoir de biais dans les résultats lié à un arbre particulier nous avons fait nos simulations cent fois en générant à chaque fois un nouvel arbre pour chaque jeu de valeurs de nos paramètres. Nous avons fait varier le degré max Δ entre 10 et 100 compris par pas de 10 et nous avons aussi pris la valeur 6 correspondant au degré maximal qui était atteint dans les simulations portant sur le diamètre temporel. Nous avons étudié l'impact du degré maximal sur différents scénarios. Ces scénarios consistent à faire varier différents paramètres :

— le nombre de nœuds de l'arbre
— le nombre d'agents
— le diamètre temporel du groupe d'agents

Degré maximal / **Nombre d'agents**	6	100
2	1524	1056
4	1105	558
8	874	311
16	727	186

TABLE 3.2 – Temps d'exécution moyen de l'algorithme d'exploration *online* en fonction du degré maximal et du nombre d'agents pour un diamètre temporel de 10 et 1000 nœuds.

La figure 3.14 représente l'évolution du temps d'exploration en fonction du degré maximal de l'arbre et de la taille du groupe d'agents pour un arbre à 1000 nœuds et pour un diamètre temporel du groupe d'agents de 10. On y remarque que plus le degré maximal est grand plus le temps d'exploration est court. Cela s'explique par le fait que la hauteur de l'arbre à explorer diminue. Il est aussi important de noter que le temps d'exploration évolue de manière similaire et dans les mêmes proportions quel que soit le nombre d'agents du groupe. La table 3.2 contient les valeurs moyennes du temps nécessaire à l'exploration en fonction du nombre d'agents et du degré maximal.

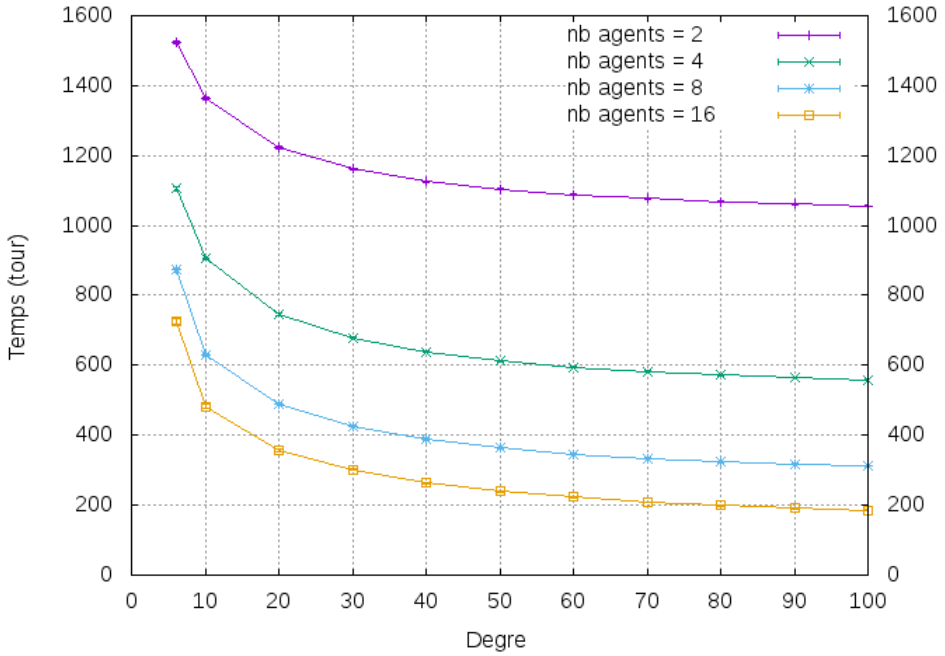


FIGURE 3.14 – Variation du temps d'exploration moyen de l'algorithme *online* en fonction du degré maximum et du nombre d'agents pour 1000 nœuds et un diamètre temporel de 10.

Nombre de nœuds \ **Degré maximal**	6	100
1000	12.98	4.13
10000	17.99	5.79

TABLE 3.3 – Hauteur moyenne de l'arbre en fonction du nombre de nœuds et du degré maximal.

La figure 3.15 représente l'évolution de la hauteur de l'arbre en fonction du degré maximal et du nombre de nœuds. La table 3.3 représente les valeurs maximales de la hauteur en fonction du nombre de nœuds et du degré maximal de l'arbre.

Notez que lorsque l'on étudie les valeurs pour le degré 6 du tableau 3.2, elles sont très en dessous des valeurs obtenues lors des simulations de la section 3.4.2 dans le tableau 3.1 pour le diamètre temporel de 10. Cette variation s'explique par le fait que les méthodes de génération des arbres sont différentes et que les hauteurs moyennes sont très différentes.

La figure 3.16 représente l'évolution du temps d'exploration en fonction du degré maximal de l'arbre et du nombre d'agents pour un arbre à 1000 nœuds dont le diamètre temporel est 100. On peux noter que les effets de

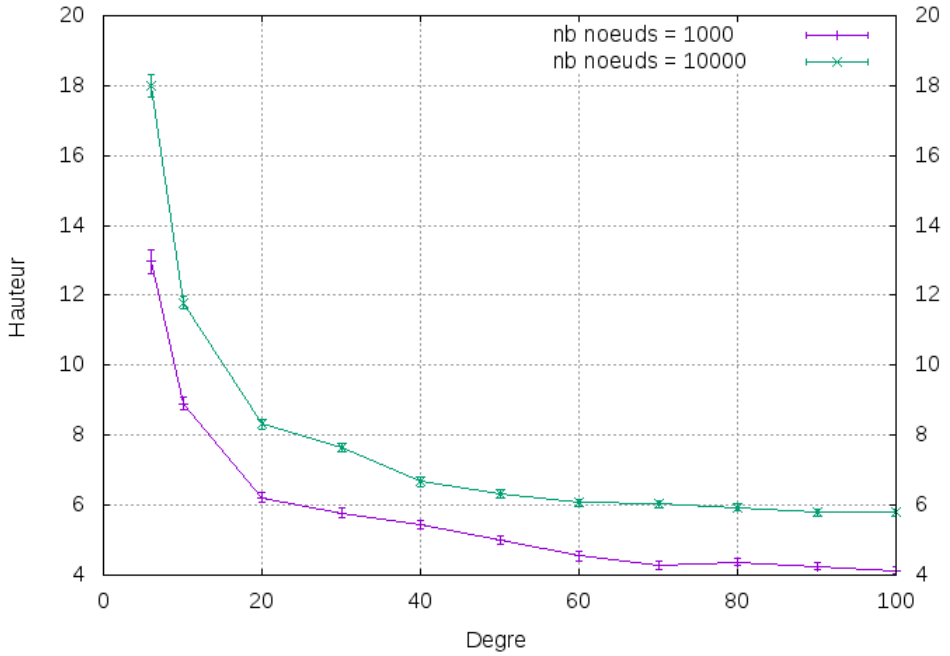


FIGURE 3.15 – Variation de la hauteur moyenne de l'arbre en fonction du degré maximum et du nombre de nœuds.

l'augmentation du degré maximal se réduisent beaucoup plus rapidement que pour les courbes précédentes. Lorsque que le degré maximal est 40, le temps d'exploration de l'arbre est similaire à celui constaté lorsque le degré maximal est de 100 mais le diamètre temporel est de 10. La table 3.4 contient les valeurs moyennes du temps d'exploration en fonction du nombre d'agents et du degré maximal.

La figure 3.17 représente l'évolution du temps d'exploration en fonction du degré maximal de l'arbre pour un arbre à 10000 nœuds et pour un diamètre temporel du groupe d'agents de 10. On peut dire de façon immédiate que multiplier le nombre de noeuds par 10 a simplement multiplié par 10 le temps d'exploration. Les courbes des figures 3.14 et 3.17 évoluent de la même façon. Tous les commentaires que nous avons fait sur la figure 3.14 s'appliquent donc à la figure 3.17. La table 3.5 contient les valeurs moyennes du temps d'exploration en fonction du nombre d'agents et du degré maximal.

La figure 3.18 représente l'évolution du temps d'exploration en fonction du degré maximal de l'arbre et de la taille du groupe pour un arbre à 10000 nœuds et pour un diamètre temporel du groupe d'agents de 100. Comme pour les figures 3.14 et 3.17, les courbes 3.16 et 3.18 évoluent de la même façon et les commentaires sont identiques. La table 3.6 contient les valeurs moyennes du temps d'exploration en fonction du nombre d'agents et du degré maximal.

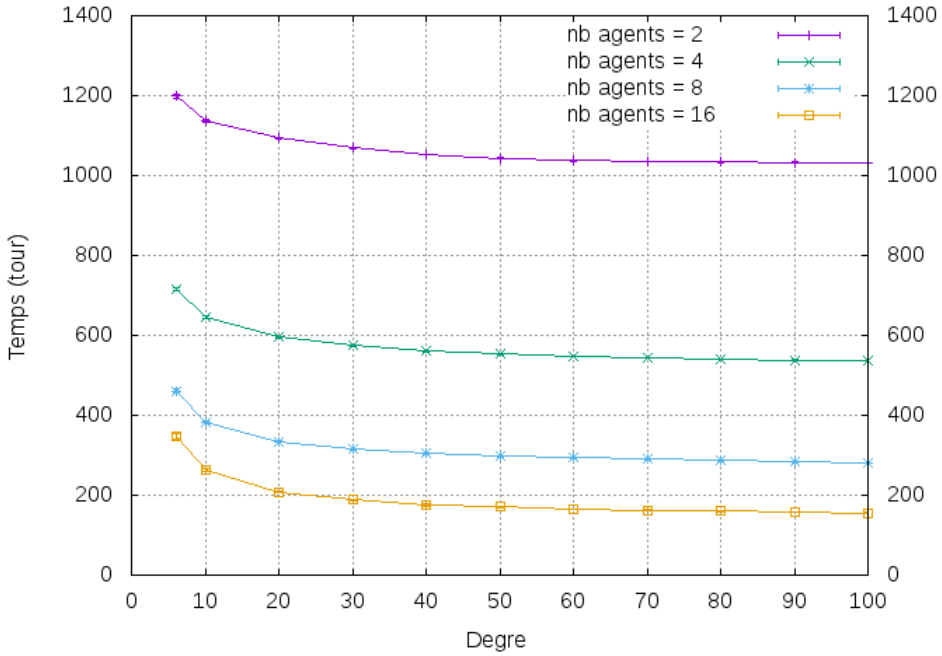


FIGURE 3.16 – Variation du temps d'exploration moyen de l'algorithme *online* en fonction du degré maximum et du nombre d'agents pour 1000 nœuds et pour un diamètre temporel de 100.

La figure 3.19 montre très clairement que le temps d'exploration des arbres est proportionnel au nombre de nœuds de l'arbre. Cette figure représente l'évolution du temps d'exploration en fonction du nombre de nœuds pour un diamètre temporel de 10 et de 16 agents. Chaque courbe représente l'évolution du temps d'exploration pour un degré maximal donné. Le tableau 3.7 donne les valeurs pour le temps d'exploration pour les deux extremums.

De toutes ces expérimentations nous pouvons tirer les conclusions suivantes :

— comme on pourrait s'y attendre, l'augmentation du nombre d'agents ou du diamètre temporel ou du degré maximal de l'arbre (pour un nombre de nœuds fixé) réduit le temps d'exploration de nos agents avec l'algorithme *online*. Cependant cette amélioration de performance devient moins significative au fur et à mesure de l'augmentation de ces valeurs.

— l'augmentation du nombre de nœuds de l'arbre augmente également le temps d'exploration mais cette fois de façon proportionnelle. Cela est cohérent avec le fait que le temps d'exploration d'un arbre croît linéairement avec le nombre de nœuds.

Nombre d'agents \ Degré maximal	6	100
2	1199	1031
4	717	536
8	460	281
16	352	155

TABLE 3.4 – Temps d'exécution moyen de l'algorithme d'exploration *online* en fonction du degré maximal et du nombre d'agents pour un diamètre temporel de 100 et pour 1000 nœuds.

Nombre d'agents \ Degré maximal	6	100
2	15305	10568
4	11085	5577
8	8769	3081
16	7277	1829

TABLE 3.5 – Temps d'exécution moyen de l'algorithme d'exploration *online* en fonction du degré maximal et du nombre d'agents pour un diamètre temporel de 10 et pour 10000 nœuds.

3.5 Conclusion et perspectives

En résumé, dans ce chapitre, nous avons motivé le problème d'exploration d'un arbre par un ou plusieurs agents et montré que dans le cas général la variante avec plusieurs agents est NP-complète. Nous avons ensuite présenté plusieurs algorithmes d'exploration d'arbres avec ou sans la contrainte du diamètre temporel borné. Le dernier de ces algorithmes est un algorithme *online*. Nous avons enfin présenté les résultats de simulations de notre algorithme et nous avons exploré l'impact de la borne sur le diamètre temporel, l'impact du nombre d'agents, du nombre de nœuds et du degré maximal de l'arbre. De nombreuses extensions à ces différentes questions peuvent être envisagées, comme la conception d'un algorithme exact plus efficace ou la preuve que le problème reste NP-complet même avec la contrainte de diamètre temporel. Il aurait été également intéressant de comparer l'algorithme online avec l'algorithme *offline* exact pour étudier son surcoût. Enfin, il serait très intéressant d'effectuer des simulations dans un modèle plus réaliste prenant en compte l'espace et la morphologie d'un batiment réelle plutôt que sa représentation abstraite sous forme de graphes. De même, une piste intéressante serait d'étudier le cas d'agents hétérogènes, par exemple, ayant des vitesses ou capacités différentes.

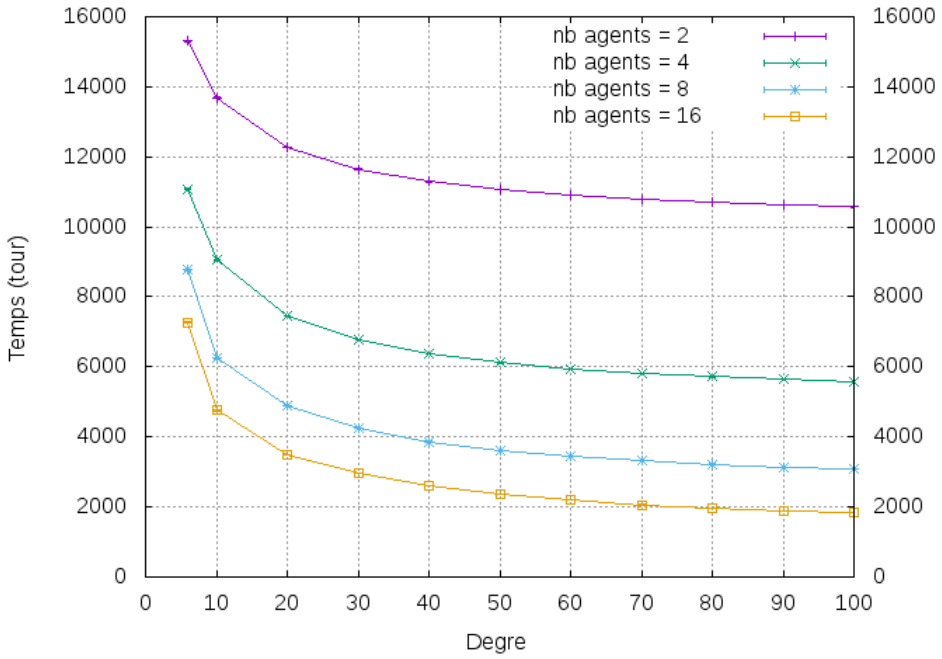


FIGURE 3.17 – Variation du temps d'exploration moyen de l'algorithme *online* en fonction du degré maximum et du nombre d'agents pour 10000 nœuds et pour un diamètre temporel de 10.

Degré maximal / **Nombre d'agents**	6	100
2	11974	10303
4	7121	5349
8	4487	2807
16	3002	1512

TABLE 3.6 – Temps d'exécution moyen de l'algorithme d'exploration *online* en fonction du degré maximal et du nombre d'agents pour un diamètre temporel de 100 et pour 10000 nœuds.

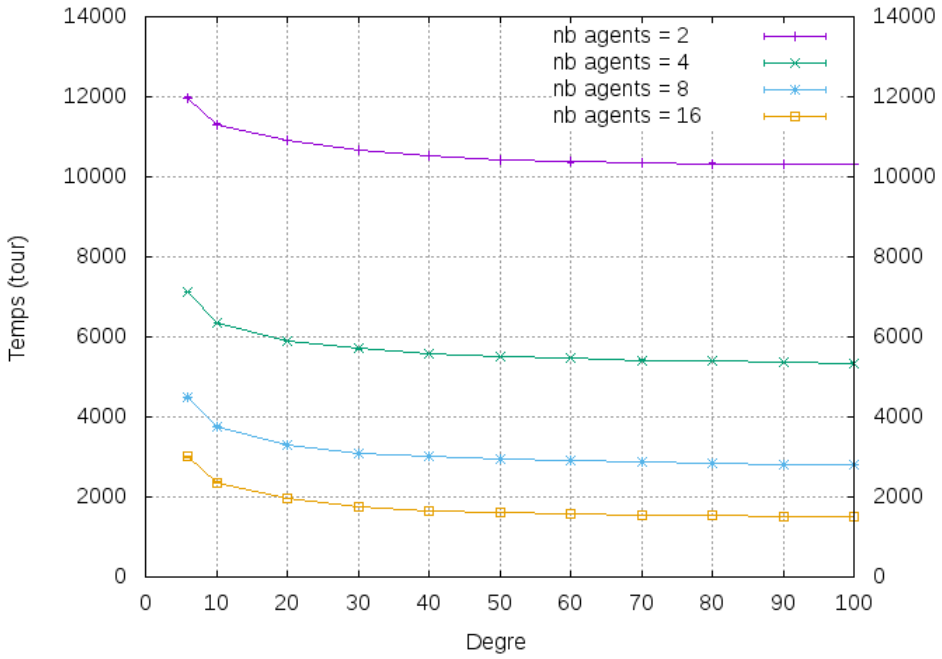


FIGURE 3.18 – Variation du temps d'exploration moyen de l'algorithme *online* en fonction du degré maximum et du nombre d'agents pour 10000 nœuds et pour un diamètre temporel de 100.

Degré \ **Nombre de nœuds**	1000	10000
6	734	7288
10	481	4774
20	353	3502
30	300	2981
40	263	2605
50	238	2369
60	221	2195
70	208	2060
80	200	1974
90	190	1899
100	185	1830

TABLE 3.7 – Temps d'exécution moyen de l'algorithme d'exploration *online* en fonction du degré maximal et du nombre de nœuds pour un diamètre temporel de 10 et pour 16 agents.

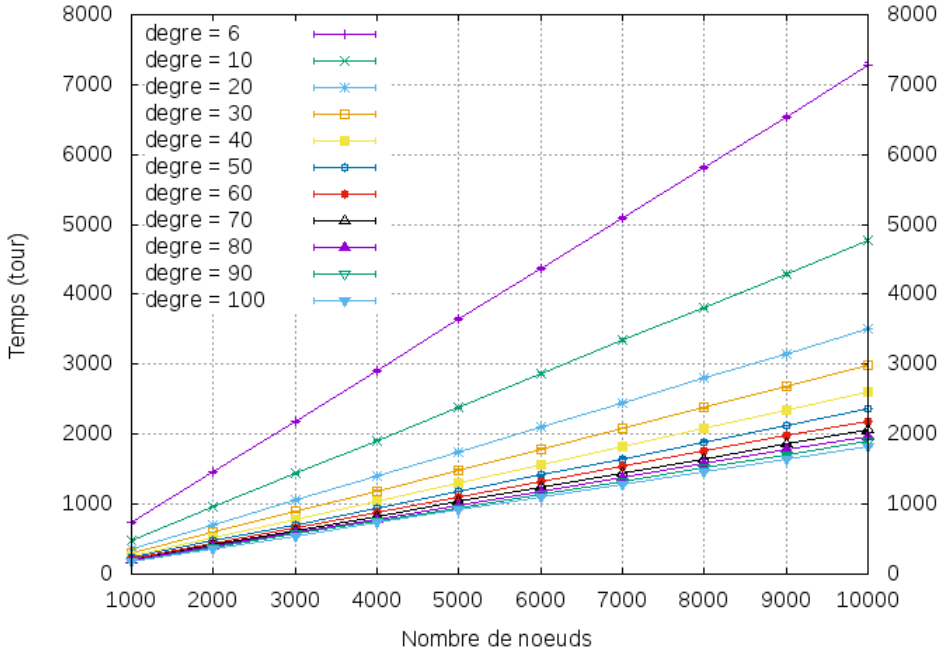


FIGURE 3.19 – Variation du temps d'exploration moyen pour l'algorithme *online* en fonction du degré maximum et du nombre de nœuds pour un diamètre temporel de 10 et pour 16 agents.

Chapitre 4

Maintien d'une forêt couvrante

Sommaire

Nous avons vu dans le chapitre précédent des algorithmes d'exploration à l'aide d'un groupe d'agents ne contenant pas de leader. Néanmoins, dans de nombreux cas, la présence d'un leader dans le groupe facilite la prise de décision. Cela n'est pas seulement vrai pour les algorithmes d'exploration mais pour la plupart des algorithmes distribués. Il est aussi souhaitable que chaque nœud connaisse un moyen contacter le leader. Pour cela de nombreux travaux se sont basés sur les arbres couvrants qui permettent d'avoir un leader (la racine de l'arbre) et de connaître facilement un moyen de contacter la racine à partir de tous les nœuds de l'arbre. Ce chapitre présente un algorithme de maintenance de forêt couvrante dans les réseaux DTN. On cherche ici à maintenir idéalement un arbre par composante connexe (au sens classique du terme, non temporel). La notion de groupe couverte n'est donc pas elle-même DTN comme dans le reste de la thèse. Les résultats de ce chapitre ont fait l'objet d'une publication à OPODIS 2014 [9] et sont actuellement soumis en version longue dans une revue.

4.1 Introduction

Comme discuté dans le chapitre 1, la dynamique des réseaux impacte les problèmes que nous étudions. Par exemple dans le cas de la construction d'arbres couvrants. Un algorithme doit-il construire un arbre unique et global dont les arêtes logiques existent par intermittence, ou doit-il construire et maintenir une forêt d'arbres qui s'efforcent de couvrir collectivement toutes les composantes à chaque instant ? Les deux points de vue ont du sens et sont fortement étudiés depuis assez peu dans le domaine du calcul distribué (voir par exemple [5, 26] pour les arbres temporels et [4, 20] pour la maintenance d'arbre).

Nous nous intéressons à la seconde interprétation qui reflète une variété de scénario où la sortie attendue de l'algorithme est en relation avec la configuration instantanée (par exemple les réseaux sociaux, les flottes de drones, les convois de véhicules sur la route). Une particularité de ces algorithmes est qu'ils ne terminent jamais. Plus spécifiquement, dans les réseaux hautement dynamiques, on ne suppose même pas que les algorithmes vont se stabiliser dans un état optimal (ici, un seul arbre par composante connexe) tant que les changements topologiques ne s'arrêtent pas, ce qui n'arrive jamais. Cela exclut, en particulier, toutes les approches où le calcul d'une nouvelle solution requiert que le calcul de la solution précédente soit terminé.

Ce chapitre est une tentative de compréhension de ce qui peut être calculé (et donc garanti) quand aucune hypothèse ne peut être faite sur les réseaux dynamiques considérés, ni sur la fréquence des changements, comme sur leurs simultanéités, ni sur la connectivité globale. Dans ce contexte apparemment chaotique, nous présentons un algorithme qui s'efforce de créer et de maintenir le moins d'arbres couvrants par composantes connexes, tout en garantissant *en permanence* certaines propriétés sur ces arbres.

4.1.1 État de l'art

De nombreux travaux se sont intéressés au problème de l'arbre couvrant dans les réseaux dynamiques, avec différents objectifs et hypothèses. Burman et Kutten [16] et Kravchik et Kutten [42] considèrent une approche autostabilisante dans lequel l'état légitime correspond à avoir un (seul) arbre couvrant minimum et où les fautes sont induites par des changements topologiques. La stratégie consiste à recalculer l'intégralité de l'arbre quand un changement a lieu. Cette approche générale, parfois appelée approche "blast away", a du sens si une période de temps pendant lequel le réseau est stable existe, ce qui n'est pas le cas ici.

Beaucoup d'algorithmes de construction d'arbres couvrants s'appuient sur les marches aléatoires pour leur élégance et leur simplicité, ainsi que pour

le paradigme localisé inhérent qu'elles offrent. En particulier, les approches qui impliquent de multiples marches aléatoires coalescentes permettent une initialisation uniforme (chaque nœud commence dans le même état) et une indépendance topologique (la même stratégie peut être utilisée quel que soit le graphe). Les premières études impliquant de tels procédés sont celles de Bar-Ilan et Zernik [7] (pour le problème de l'élection et de l'arbre couvrant), d'Israeli et Jalfon [41] (pour l'exclusion mutuelle), et le chapitre 14 de Aldous et Fill [2] (pour l'analyse générale).

Le principe d'utiliser des marches aléatoires coalescentes pour construire des arbres couvrants dans les réseaux faiblement dynamiques est utilisé par Baala et al. [6] et Abbas et al. [1] ; les jetons y annexent les territoires en se capturant les uns les autres. Au regard de la dynamique, les deux algorithmes requièrent que les nœuds connaissent une borne supérieure sur le temps de couverture de la marche aléatoire, cela pour pouvoir regénérer un jeton s'ils ne sont pas visités durant une assez longue période de temps, grâce à une horloge. Au-delà de la force de cette hypothèse (qui revient à connaître le nombre de nœuds n, ou la taille des composantes dans notre cas), l'efficacité de la stratégie de l'horloge, qui consiste à avoir une horloge sur chaque noeud, diminue fortement avec le taux des changements topologiques. En particulier, s'ils sont plus fréquents que le temps de couverture (lui-même en $O(n^3)$), alors l'arbre est constamment fragmenté en pièces "mortes" auxquelles il manque une racine et donc un leader.

Un autre algorithme basé sur les marches aléatoires est proposé par Bernard et al. [13]. Dans cette approche, l'arbre est constamment redéfini lorsque le jeton se déplace. Comme le jeton se déplace seulement sur les arêtes présentes et de part le mécanisme utilisé dans l'algorithme, les arêtes qui ont disparu sont naturellement retirées de l'arbre tant que la marche continue. Par conséquent, l'algorithme tolère la perte d'arêtes de l'arbre. Néanmoins, il continue de souffrir du problème de la détection de la disparition des jetons en utilisant des horloges basées sur le temps de couverture, technique qui, on l'a vu, ne fonctionne que lorsque la dynamique est faible.

Un travail récent fait par Awerbuch et al. [4] s'intéresse à la maintenance d'arbres couvrants minimum dans les réseaux dynamiques. L'article introduit le fait qu'une solution au problème peut être adaptée après un changement topologique en utilisant $O(n)$ messages (et en temps $O(n)$), contrairement aux $O(m)$ messages de l'approche "blast away" qui était considérés comme optimales. D'ailleurs, ceci démontre l'intérêt de *mettre à jour* une solution plutôt que de tout recalculer depuis le début dans le cas des arbres couvrants minimum. L'algorithme a de bonnes propriétés pour les réseaux hautement dynamiques. Par exemple, il considère comme naturel le fait que les composantes se séparent et fusionnent perpétuellement. De plus, il tolère la survenue de nouveaux événements topologiques quand une opération de mise à jour est en

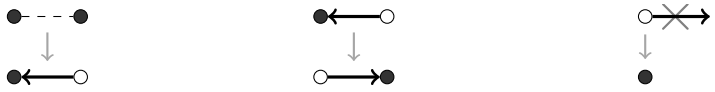
cours d'exécution. Dans ce cas, les opérations de mises à jour sont mises dans une file et sont exécutées l'une après l'autre. Alors que ce mécanisme autorise un nombre arbitraire d'événements topologiques, il requiert toujours que ces rafales de changements soient seulement épisodiques et que le réseau reste stable à terme pour (au moins) un temps linéaire relativement au nombre de nœuds. Cela permet que les opérations de mises à jour terminent et que l'état logique de l'arbre soit consistant avec la réalité physique.

Tous les algorithmes mentionnés ci-dessus assument que ces opérations de *mise à jour globale* (e. g. mécanisme de vague) peuvent être exécutées simultanément, ou du moins ultimement, ou il existe quelques nœuds qui peuvent collecter une *information globale* sur la structure de l'arbre. En ce qui concerne la dynamique, ceci interdit les changements arbitraires et non anticipés dans le réseau.

4.1.2 Le principe de la forêt couvrante

Un mécanisme purement localisé a été proposé par Casteigts [17] et étendu dans [20] pour la maintenance d'une forêt couvrante (non minimale) dans les réseaux dynamiques non restreints. Il utilise un modèle d'interaction atomique entre voisins inspiré des systèmes de ré-étiquetage de graphes [44]. Il peut être décrit informellement comme suit. Initialement, chaque nœud contient un jeton et est la racine de son propre arbre. Quand deux racines se retrouvent sur les deux extrémités d'une arête (voir la règle de fusion figure 4.1), une des deux détruit son jeton et sélectionne l'autre racine comme parent (les arbres ont fusionné). Le reste du temps, chaque jeton effectue une marche aléatoire dans son propre arbre à la recherche d'autres opportunités de fusions (règle de circulation). Les relations parent/enfant dans l'arbre sont échangées pour rester en accord avec le fait que le jeton est la racine de l'arbre. Le fait que la marche aléatoire soit confinée dans l'arbre sous-jacent est crucial et est différent de ce qui est fait dans les algorithmes discutés ci-dessus, pour lesquels les marches sont libres d'aller n'importe où sans restriction. Cette particularité implique des propriétés très intéressantes pour les réseaux hautement dynamiques. En particulier, quand une arête de l'arbre disparaît, le côté enfant de l'arête sait immédiatement qu'il n'y a plus de jeton dans son sous-arbre. Il peut alors *instantanément* regénérer un jeton (c.-à-d. devenir une racine), sans concertation globale, ni collecte d'information supplémentaire. En conséquence, la fusion et le fractionnement des arbres sont gérés d'une façon purement localisée.

Ce mécanisme très simple garantit que le réseau reste couvert par une forêt couvrante à tout moment, et que 1) aucun cycle ne peut apparaître, 2) les sous-arbres maximums sont toujours des arbres enracinés orientés (avec un jeton à la racine), et 3) chaque nœud appartient à un tel arbre quel que soit le chaos dans les changements topologiques. En revanche, il n'est pas attendu que l'algorithme atteigne un état optimal où chaque arbre couvre exactement



(a) Règle de fusion

(b) Règle de circulation

(c) Règle de regénération

FIGURE 4.1 – Principe de la forêt couvrante (figure de Barjon et al. [9]). *Les nœuds noirs sont ceux ayant un jeton. Les flèches épaisses et noires représentent la relation enfant-parent. Les flèches verticales grises représentent les transitions.*

une composante connexe. Même si le réseau se stabilise, la solution ne va pas forcément converger rapidement vers la solution optimale. Savoir si ce principe général peut être transposé dans le modèle à base de passage de message reste une question ouverte.

4.1.3 Notre contribution

Ce chapitre fournit une implémentation du principe de la forêt couvrante tel que décrit ci-dessus dans le modèle par passage de messages *synchrones*. Du fait de la perte de l'atomicité et de l'exclusivité des interactions, l'algorithme est devenu beaucoup plus sophistiqué que son homologue original. Bien qu'il reflète toujours le même principe haut niveau, il fait face à de nouveaux problèmes qui nécessitent des différences conceptuelles. En particulier, le modèle original autorise à un nœud d'éviter de simultanément sélectionner un parent et être sélectionné comme parent, ce qui permet d'éviter la création de cycles. L'un des ingrédients dans le nouvel algorithme utilisé pour contourner ce problème est une technique originale (appelée la technique du *score unique*) qui consiste à maintenir, à travers le réseau, un ensemble de variables `score` qui reste toujours une permutation de l'ensemble des identifiants des nœuds. Ce mécanisme nous permet de casser la symétrie et d'éviter la formation de cycle dans un contexte où les identifiants seuls ne le permettent pas et sera expliqué de façon plus approfondie plus loin dans ce chapitre. Ce chapitre est organisé de la façon suivante. Dans la section 4.2 nous présentons le modèle et les notations que nous utiliserons. Ensuite, la section 4.3 introduit l'algorithme. Enfin la section 4.4 présente des résultats expérimentaux qui valident notre algorithme.

4.2 Modèle et notations

Le réseau est représenté par un graphe évolutif non daté [34] $\mathcal{G} = (G_1, G_2, \ldots)$, tel que $G_i = (V, E_i)$, où V est un ensemble fixé de nœuds et E_i est un ensemble changeant d'arêtes non orientées. Comme Kuhn et al. [43], nous consi-

dérons un modèle de calcul synchrone (qui a donc des tours), où dans chaque tour i, un adversaire choisit l'ensemble des arêtes présentes dans E_i. Dans notre cas, cet ensemble est arbitraire (c.-à-d. que l'adversaire n'est pas restreint). Au début de chaque tour, chaque nœud envoie un message qu'il a préparé à la fin du tour précédant. Ce message est envoyé à tous ses voisins dans E_i, bien qu'ils ne soient pas connus par le nœud. Ensuite, chaque nœud reçoit tous les messages envoyés par ses voisins (dans le même tour), et enfin calcule son nouvel état et son prochain message. Par conséquent, chaque tour correspond à trois phases `(envoyer, recevoir, calculer)`, qui correspondent à une rotation du modèle originel de [43] où les phases sont `(calculer, envoyer, recevoir)`. Cette adaptation nous permet d'observer l'état de nœuds après chaque fin de tour et non pas au milieu des tours.

Nous supposons que les nœuds ont un identifiant unique pris dans un ensemble totalement ordonné. Pour deux nœuds u et v, cela veut dire que $ID(u) > ID(v)$ ou que $ID(u) < ID(v)$. Un nœud peut spécifier à quel voisin son message est destiné (bien que tous ses voisins vont le recevoir) en modifiant le champ `target` de son message. De manière symétrique l'ID de l'émetteur du message peut être lu dans le champ `sender` du message. Comme les arêtes ne sont pas orientées, si u reçoit un message de v à un tour i alors v reçoit aussi un message de u dans ce tour. Nous appelons cette propriété le *principe de réciprocité* et il s'agit d'un point important pour la correction de notre algorithme.

L'utilisation de tours synchrones nous permet de représenter la progression de l'exécution par une séquence de *configurations* $(C_0, C_1, C_2, \ldots, C_i)$, où chaque C_i correspond à l'état du système après le tour i (à l'exception de C_0 qui est l'état initial). Chaque *configuration* consiste en l'union des variables de tous les nœuds, comme défini plus loin.

4.3 L'algorithme de la forêt couvrante

Dans cette section, nous décrivons en détail l'algorithme de la forêt couvrante. Ayant été impliqué dans différents aspects de ce travail (conception, simulation), mais pas dans les preuves théoriques qui représentent plus d'une dizaine de pages, nous nous limitons ici à décrire l'algorithme et les simulations effectuées, et renvoyons le lecteur intéressé à la version longue de l'article [10] pour les preuves détaillées.

4.3.1 État des variables

Derrière la variable `ID` que l'on suppose initialisée par un agent externe, chaque nœud a un ensemble de variables qui reflètent sa situation dans l'arbre : `status` représente la possession du jeton (`T` s'il a le jeton, `N` s'il ne l'a pas) ;

`parent` contient l'`ID` de son nœud parent ($\perp$ s'il n'en a pas) ; `children` contient l'ensemble des `ID`s des enfants du nœud ($\emptyset$ s'il n'en a pas). On peut observer que les variables `status` et `parent` sont redondantes, car dans le principe de la forêt couvrante (voir section 4.1.2) la possession d'un jeton équivaut à être une racine. Notre algorithme garantit cette équivalence mais nous gardons les deux variables séparées pour simplifier la description de l'algorithme et rendre les choses plus intuitives. La variable `neighbors` d'un nœud contient l'ensemble des `ID`s des nœuds dont il a reçu un message dans la dernière phase de réception. Ces voisins peuvent appartenir ou pas au même arbre que le nœud courant. La variable `contender` contient l'`ID` d'un voisin que le nœud courant envisage de sélectionner comme parent dans le prochain tour (ou $\perp$ s'il n'en existe pas). Enfin, la variable `score` est l'ingrédient principal de notre mécanisme d'évitement de cycle ; son rôle sera décrit plus bas.

Valeurs initiales : Tous les nœuds sont uniformément initialisés. Ils sont initialement la racine de leur propre arbre (c.-à-d. `status` $= T$, `parent` $= \perp$ et `children` $= \emptyset$). Aucun des nœuds ne connaît ses voisins (`neighbors` $= \emptyset$),ni n'a de prétendant (`contender` $= \perp$). Le `score` de chacun est initialisé avec son `ID`.

4.3.2 Structure d'un message (et des variables associées)

Les messages sont composés d'un certain nombre de champs : `sender` est l'`ID` du nœud émetteur ; `senderStatus` est son statut (soit T, soit N) ; `score` est son score au moment où le message est préparé. Le champ `action` contient l'une des valeurs suivantes : {`FLIP`,`SELECT`,`HELLO`}. Informellement, les messages `SELECT` sont envoyés par un nœud racine à un autre nœud racine pour lui signifier qu'il "l'adopte" comme parent (opération de fusion) ; les messages `FLIP` sont envoyés par un nœud racine à un de ses enfants pour faire circuler le jeton (opération de circulation) ; les messages `HELLO` sont envoyés par défaut, quand aucun autre message n'est envoyé, pour signaler à ses voisins la présence du nœud et son statut. Enfin, `target` est l'`ID` du voisin vers lequel le message `FLIP` ou `SELECT` est dirigé ($\perp$ pour les messages `HELLO`).

Les messages reçus sont stockés dans la variable `mailbox` du nœud. Il s'agit d'une table associative dont les *clefs* sont les `ID`s des émetteurs (c.-à-d. un message dont l'`ID` de l'émetteur est u peut être accédé en tant que `mailbox`$[u]$). À chaque tour, l'algorithme utilise une fonction `RECEIVE()` qui purge la boîte-aux-lettres et la remplit avec les messages reçus durant ce tour (un pour chacun de ses voisins). Un nœud peut ensuite mettre à jour son ensemble de voisins en regardant les *clefs* pour lesquelles il existe une entrée dans sa boîte-aux-lettres. De la même manière, il peut supprimer de sa liste de enfants tous les nœuds qui ne sont plus ses voisins.

Comme mentionné ci-dessus, chaque nœud prépare, à la fin de chaque

tour, le message qui sera envoyé au début du tour suivant. Ce message est stocké dans la variable `outMessage`. Nous nous autorisons l'écriture suivante $m \leftarrow (a, b, c, d, e)$ pour définir un nouveau message m dont a est le nœud émetteur (ayant comme statut b et comme score e), d est le nœud cible et c est l'action.

Valeur initiales : La boîte-aux-lettres de chaque nœud est initialement vide ($\texttt{mailbox} = \emptyset$) et sa variable `outMessage` est initialisée avec (`ID`, `T`, `HELLO`, $\perp$, `ID`).

4.3.3 Description informelle de l'algorithme

L'algorithme implémente le mécanisme général présenté dans la section 4.1.2. Dans cette section nous expliquons tout d'abord comment sont réalisées les trois opérations principales (*fusion*, *circulation* et *regénération*). Nous discuterons ensuite en détails de l'opération de fusion et du problème qui apparaît lors de son enchevêtrement avec l'opération de circulation du jeton, problème dû à la perte de l'atomicité du fait du modèle de passage de messages. La solution résultante est substantiellement plus sophistiquée que le mécanisme original et pourtant, elle reflète fidèlement le même principe de haut niveau. Commençons avec quelques généralités. Lors de chaque tour, chaque nœud diffuse à ses voisins un message contenant, entre autres, son statut (`T` ou `N`) et l'action concernée (`SELECT`, `FLIP` ou `HELLO`). Que le message soit ou pas, adressé à une cible spécifique (ce qui est le cas pour les messages `SELECT` et `FLIP`), tous les nœuds qui reçoivent ce message vont utiliser ces informations pour leurs propres décisions. En se basant sur les informations reçues et sur son état local, chaque noeud calcule son nouvel état et la structure locale de son arbre (les variables `children` et `parent`) à la fin du tour, puis il prépare son prochain message à envoyer.

Nous allons maintenant décrire ces trois opérations. Tout au long des explications, le lecteur est invité à se référer à la figure 4.2, où un exemple d'exécution (utilisant toutes les opérations) est montré. Tous les détails sont aussi donnés dans les algorithmes 5 et 6.

Fusion : Si une racine v (c.-à-d. un nœud ayant un jeton), détecte l'existence d'une racine voisine ayant un `score` plus élevé que le sien, alors elle considère ce nœud comme un `contender` possible, c.-à-d. comme un nœud qu'il peut sélectionner comme parent au prochain tour. Si plusieurs racines de ce type existent, alors celle (u) ayant le plus haut score est sélectionnée. Au début du tour suivant, le nœud v envoie un message `SELECT` à u pour l'informer qu'il l'a choisi comme parent. Deux cas sont alors possibles : soit l'arête considérée entre u et v est toujours présente lors de ce tour, soit elle a disparu entre les deux tours. Si elle est toujours présente, alors u reçoit bien le message et ajoute v à sa liste de enfants (Ligne 16). En ce qui concerne v, il met sa

variable `parent` à u et `status` à `N` (Lignes 8 et 9). Si l'arête a disparu, alors u ne reçoit pas le message qui est donc perdu. Toutefois, de part la réciprocité de l'échange des messages, v ne reçoit pas alors de message de u et donc n'exécute pas les changements correspondants. Ainsi, à la fin du tour, soit les arbres sont correctement fusionnés, soit ils sont correctement séparés.

Circulation : Si une racine v ne détecte pas d'autre racine avec un score plus grand, alors elle sélectionne un de ses enfants, si elle en a un, de façon aléatoire (voir Ligne 27), sinon elle fait rien.

L'aspect aléatoire dans notre algorithme est un choix que nous avons fait mais le remplacer par n'importe quelle stratégie déterministe n'affectera pas sa correction. Une fois un enfant choisi (u), la racine prépare un message `FLIP` à son attention et le lui envoie au début du tour suivant. Deux cas sont encore possibles, selon que l'arête $\{u, v\}$ est toujours présente ou pas lors de ce tour. Si elle est toujours présente, alors u reçoit le message, met à jour `status` à `T` et ajoute v à `children` la liste de ses enfants (Lignes 15 et 16). En ce qui concerne v, il met à jour sa variable `parent` à la valeur u et `status` à `N` (Ligne 8 et 9). Si l'arête a disparu alors v le détecte de la même manière que précédemment et ne fait aucune modification. Le nœud u détecte lui aussi que l'arête vers son parent actuel a disparue et il regénère alors un jeton (discuté prochainement dans la section). On pourra noter qu'en l'absence d'une opportunité de fusion, un nœud recevant le jeton dans le tour i va immédiatement préparer un message `FLIP` pour le faire circuler au prochain tour. Tant que l'arbre est composé d'au moins deux nœuds, le jeton va circuler à chaque tour. Pour que les jetons restent détectables dans ce cas-là, le statut annoncé dans les messages `FLIP` est `T` (alors qu'il est `N` pour les messages `SELECT`). Cela est fait pour permettre la détection de la racine de cet arbre par une autre racine.

Regénération : La première chose que fait un nœud non racine après avoir reçu les messages du tour courant est de tester si l'arête vers son parent actuel existe toujours (un message a-t-il été reçu ?). Si cette arête a disparu, alors le nœud regénère directement une racine (Ligne 7). Une bonne propriété de la forêt couvrante est que la regénération ne peut pas arriver deux fois dans le même arbre. Si un arbre est coupé en plusieurs pièces simultanément, alors chacun des sous-arbres résultant des cassures aura exactement un nœud exécutant cette action.

La technique du score unique : Contrairement au modèle de graphe haut niveau décrit dans [20] basé sur le re-étiquetage de graphe, dans lequel les opérations de fusions impliquent deux nœuds de manière *exclusive*, le caractère non atomique du passage de message autorise une *chaîne* de sélection qui peut impliquer une séquence arbitrairement longue de nœuds. Une chaîne de sélection est une suite de sélection définie comme suit : a sélectionne b, b sélectionne c, etc. Cela a à la fois des avantages et des inconvénients. Cela permet un processus initial de fusions très rapide (voir tour 1 et 2 dans la

figure 4.2 pour un exemple). En revanche, cela peut conduire à des cycles que nous évitons en introduisant des scores.

En effet, se baser seulement sur une comparaison des `ID`s n'est pas suffisant pour éviter les cycles. Considérons une chaîne de sélection au tour i qui se termine sur un nœud u. Rien ne permet d'éviter à u de passer le jeton à un nœud enfant ayant un identifiant plus petit, disons v, au tour précédant $i-1$ (le même tour où le statut `T` de u a été entendu par l'avant dernière racine de la chaîne). Maintenant rien ne permet d'éviter à v de sélectionner un des nœuds de la chaîne de sélection dans le tour i et ainsi créer un cycle. Le mécanisme de score évite une telle situation en faisant respecter le fait suivant après chaque `FLIP`, la nouvelle racine à un score plus grand que son prédécesseur (voir Lignes 9 et 13 dans l'algorithme 6).

Le mécanisme de score garantit aussi que l'ensemble courant des scores (à travers le réseau) est toujours une permutation de l'ensemble initial des scores (grâce aux Lignes 9 et 13 dans l'algorithme 6). Par conséquent les scores sont toujours uniques et permettent d'éviter les cycles.

Un mot sur la convergence : Chaque jeton fait une marche aléatoire dans son propre arbre. Par conséquent, si le réseau arrête de changer et à moins que quelques uns des arbres soient dans une configuration bipartie où toutes les racines sont sur le même coté, la configuration va ultimement (et avec forte probabilité) se stabiliser à un seul arbre par composante connexe. Bien que la convergence ne soit pas notre préoccupation première ici, nous précisons que les scénarios "pathétiques" où certains des arbres sont bipartis peuvent être facilement évités, en faisant s'arrêter les jetons sur les nœuds pendant un nombre aléatoire de tours additionnels (une technique appelée marche *paresseuse*), cela permettant de casser la symétrie.

Ayant été impliqué dans l'implémentation et l'expérimentation de cet algorithme, mais pas dans ses preuves, le lecteur est renvoyé à la version longue du papier [10] pour les preuves de corrections.

4.4 Simulation sur des traces réelles (traces Infocomm 2006)

Afin de vérifier l'applicabilité de notre algorithme à des situations du monde réel, l'algorithme a été implémenté dans le simulateur JBotSim [18] et testé sur le jeu de données `Infocomm06` [49]. Ce jeu de données est un enregistrement des interactions possibles entre les personnes présentes durant la conférence Infocomm'06. Chacun des conférenciers a été équipé d'un équipement Bluetooth qui enregistre les interactions possibles. Le graphe résultant a les caractéristiques suivantes : le nombre de nœuds est de 78 et le degré moyen d'un nœud

```
1  while True do
2  |  SEND(outMessage);
3  |  mailbox ← RECEIVE();// Réception des messages (indexés
   |    par les IDs des émetteurs)
4  |  neighbors ← mailbox.keys();// Tous les IDs des émetteurs
5  |  children ← children ∩ neighbors
   |  // Regénération du jeton si le lien vers le parent a
   |    disparu
6  |  if status=N ∧ parent ∉ neighbors then
7  |  |  BECOME_ROOT();
   |  // Teste si le FLIP ou SELECT sortant (s'il existe) est
   |    réussi
8  |  if outMessage.action ∈ {FLIP,SELECT} ∧ outMessage.target
   |  ∈ neighbors then
9  |  |  ADOPT_PARENT(outMessage)
   |  // Traitement des messages reçus
10 |  contender ← ⊥;
11 |  contenderScore ← 0;
12 |  foreach message ∈ mailbox do
13 |  |  if message.target = myID then
14 |  |  |  if message.action = FLIP then
15 |  |  |  |  BECOME_ROOT();
16 |  |  |  ADOPT_CHILD(message);// appelé pour FLIP et SELECT
17 |  |  else
18 |  |  |  if message.status = T ∧ message.score
   |  |  |  > contenderScore then
19 |  |  |  |  contender ← message.ID;
20 |  |  |  |  contenderScore ← message.score;
   |  // Préparation du message à envoyer
21 |  outMessage ← ⊥
22 |  if status = T then
23 |  |  if contenderScore > score then
24 |  |  |  PREPARE_MESSAGE(SELECT, contender);
25 |  |  else
26 |  |  |  if children ≠ ∅ then
27 |  |  |  |  PREPARE_MESSAGE(FLIP, random(children));
28 |  if outMessage = ⊥ then
29 |  |  PREPARE_MESSAGE(HELLO, ⊥);
```

Algorithme 5 : Algorithme principal

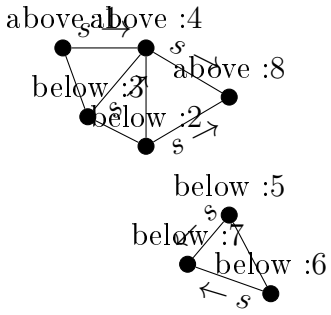
```
1  procedure BECOME_ROOT
2  |  status ← T;
3  |  parent ← ⊥;

4  procedure ADOPT_PARENT(outMessage)
5  |  status ← N;
6  |  parent ← outMessage.target;
7  |  if outMessage.action = FLIP then
8  |  |  children ← children\parent;
9  |  |  score ← min(score, mailbox[parent].score);

10 procedure ADOPT_CHILD(message)
11 |  children.add(message.ID);
12 |  if message.action = FLIP then
13 |  |  score ← max(score, message.score);

14 procedure PREPARE_MESSAGE(action, target)
15 |  switch action do
16 |  |  case SELECT
17 |  |  |  outMessage ← (ID, N, SELECT, target, score);
18 |  |  case FLIP
19 |  |  |  outMessage ← (ID, T, FLIP, target, score);
20 |  |  case HELLO
21 |  |  |  outMessage ← (ID, status, ⊥, ⊥, score);
```

Algorithme 6 : Fonctions appelées dans l'Algorithme 5.

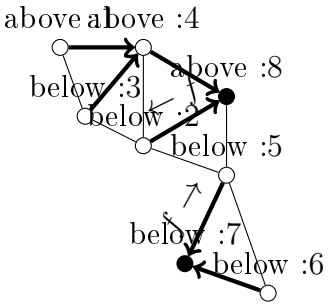
est de 1,3. Une arête peut apparaître à tout moment mais on notera le fait que sa présence n'est testée que toutes les 120 secondes ; cela veut dire que le temps de présence d'une arête est considéré comme étant un multiple de 120 secondes. Deux cas ont été considérés lors de nos simulations, en fonction du nombre de tours que l'on suppose s'exécuter chaque seconde. Les résultats montrent le nombre moyen d'arbres par composante connexe, sur 100 exécutions. Dans le premier cas (Figure 4.3), nous supposons que 10 tours peuvent être réalisés chaque seconde, ce qui semble réaliste. Dans le second cas, nous abaissons nos attentes en supposant qu'un tour seulement peut s'exécuter chaque seconde (Figure 4.4). Ces résultats montrent que le nombre d'arbres par composante connexe, à travers le temps, est très près de 1 (environ 1,027 dans le premier cas et 1,080 dans le deuxième). De plus, l'algorithme arrive dans une configuration optimale d'un arbre par composante connexe environ 47% du temps dans le premier cas et 32,68% dans le deuxième cas, ce qui montre que notre algorithme converge tout de même un tiers du temps même avec l'hypothèse défavorable. Ces résultats valident aussi l'intérêt de notre algorithme pour les scénarios du monde réel.

4.5 Conclusion

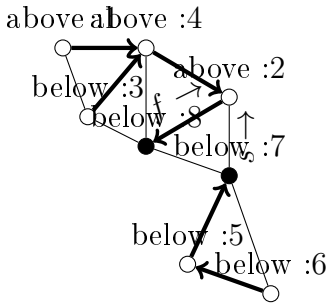
Dans ce chapitre nous avons présenté une adaptation d'un algorithme décrit sous le modèle de ré-étiquetage de graphes dans le modèle de passage de messages synchrones. Cet algorithme permet de créer une forêt couvrante dans les graphes fortement dynamiques. Grâce à des opérations de fusion, l'algorithme cherche à agréger les arbres couvrants lorsque cela est possible. Dans le cas où le graphe dynamique se stabilise l'algorithme finit par atteindre un arbre couvrant par composante connexe. Nous avons enfin fait tourner notre algorithme sur une trace d'un réseau dynamique réel et nous avons pu remarquer que notre algorithme obtient tout de même un nombre d'arbres couvrants par composante connexe proche de un. Ces résultats ont été publiés au sein de l'article [9], les preuves sont disponibles dans la version longue de l'article [10].



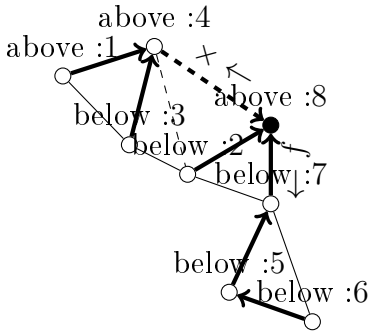
(a) tour 1



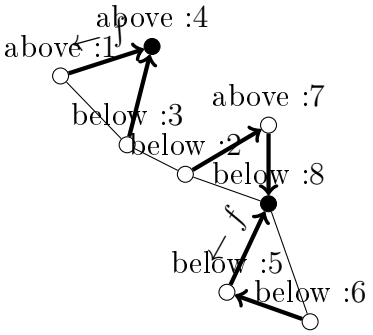
(b) tour 2



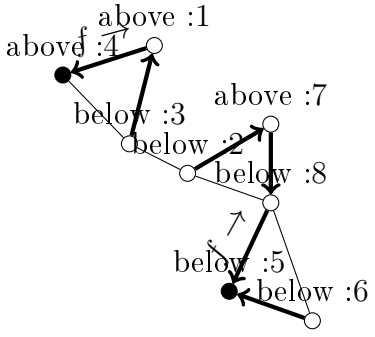
(c) tour 3



(d) tour 4



(e) tour 5



(f) tour 6

FIGURE 4.2 – Exemple d'exécution de l'algorithme qui illustre tous les types d'opérations : sélection d'un parent ($s \rightarrow$), circulation du jeton ($f \rightarrow$) et ~~déconnexion de l'arbre ($\times \leftarrow$). *Les deux premiers symboles représentent les messages SELECT ou FLIP qui sont envoyés dans le prochain tour. Les nœuds noirs (resp. blanc) sont ceux qui ont (resp. n'ont pas) un jeton au début du tour courant. Les arêtes de l'arbre sont représentées par des arcs en gras. Les arêtes en pointillées sont celles qui viennent juste de disparaître.*

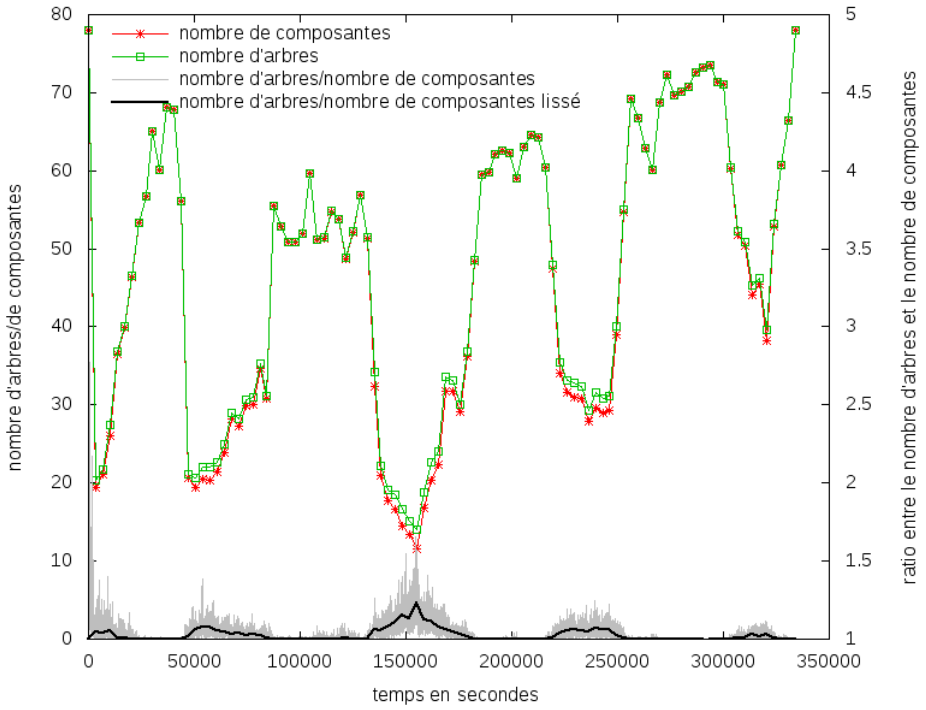


FIGURE 4.3 – Nombre de racines par composante connexe, en supposant 10 tours par seconde.

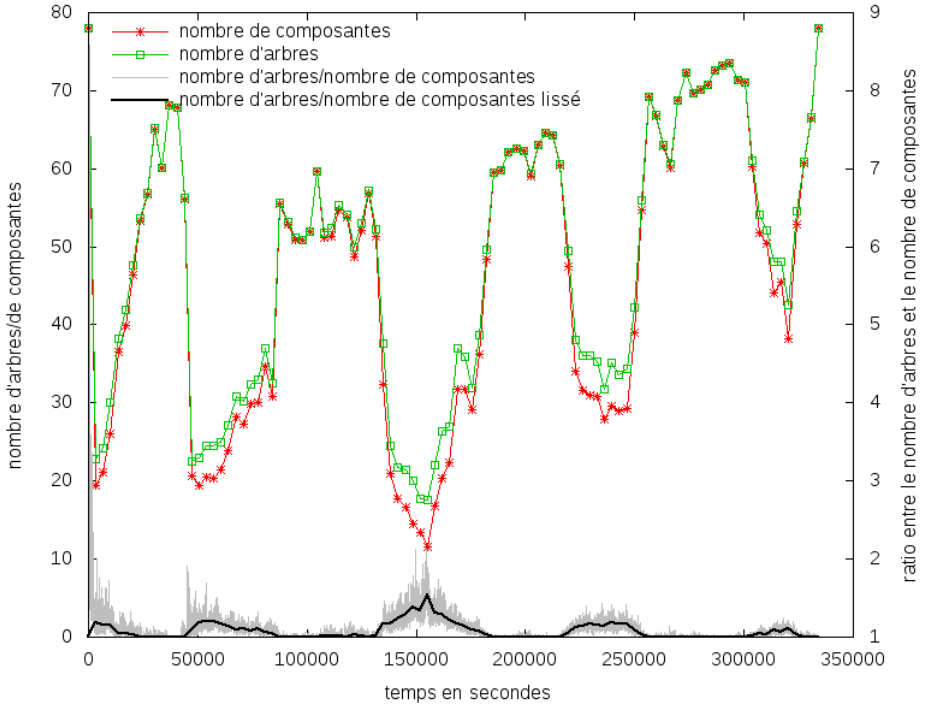


FIGURE 4.4 – Nombre de racines par composante connexe, en supposant 1 tour par seconde.

Chapitre 5

Module `Obstacle`

Sommaire

Dans ce chapitre, nous nous intéressons à un module permettant de générer des obstacles au sein du simulateur JBotSim [18]. Nous présentons d'abord les différentes raisons qui nous ont poussé à l'élaboration de ce module. Puis nous indiquons comment utiliser ce module. Enfin, nous en présentons le fonctionnement interne avec notamment un exemple d'utilisation.

5.1 Motivation

Au délà de la conception et de l'analyse théorique des algorithmes d'explorations présentés dans le chapitre 3, nous avons voulu rendre possible (sans pour autant les faire nous-même) des simulations d'exploration dans les espaces réels 2D ou 3D (et non seulement dans les graphes). L'idée générale est de pouvoir représenter des structures réelles (bâtiment, salles, *etc.*) que des appareils (robots/drones) doivent explorer en groupe en tenant compte des contraintes que ces obstacles imposent sur les possibilités de communications.

Par exemple, lorsque deux appareils sont séparés par un mur et/ou sol et/ou plafond, ils ne peuvent pas établir de lien de communication.

Nous avons donc développé un module `Obstacle` pour le simulateur JBotSim. Par défaut, JBotSim permet de simuler des appareils (nœuds) qui évolue dans un espace en deux ou trois dimensions et dont les communications évoluent selon la distance des nœuds. Pour respecter la contrainte que l'on s'est donné, notre module permet de couper les communications entre les différents nœuds lorsqu'un obstacle (mur d'une pièce, ou sol) se trouve entre eux. Nous avons aussi fait en sorte que les nœud puisse détecter la présence d'obstacles afin d'y réagir, notamment en évitant les collision avec ces derniers.

Le module est publiquement accessible sur le dépot Bitbucket [8].

5.2 Utilisation du module `Obstacle`

Cette section décrit l'utilisation du module `Obstacle`, c'est à dire comment l'initialiser, comment ajouter ou supprimer des obstacles et comment faire en sorte que les nœuds soient avertis de la présence d'obstacles. Un exemple concret d'utilisation est fourni dans la section 5.4.

5.2.1 Initialisation du module

Le module `Obstacle` contient deux fonctionnalités importantes : 1) il coupe la communication entre deux nœuds lorsqu'un obstacle se trouve entre eux, 2) il peut notifier un nœud lorsque celui-ci contient un ou plusieurs obstacles dans sa zone de détection.

Pour activer les fonctionnalités du module il faut d'abord créer une topologie puis initialiser le module Obstacle sur cette topologie. Pour cela on utilise la classe **ObstacleManager** et on exécute le code suivant :

```
Topology topology = new Topology();
ObstacleManager.init(topology);
```

Une fois cette phase d'initiation réalisée le module est prêt à être utilisé.

Il est également possible lors de l'initialisation de mettre en place la partie graphique du module, pour cela il faut exécuter le code suivant :

```
Topology topology = new Topology();
JTopology jTopology = new JTopology(topology);
ObstacleManager.init(topology,jTopology);
```

5.2.2 Ajout et suppression d'obstacles dans la topologie

Une fois que le module est initialisé sur une topologie une première fonctionnalité du module est activée : si un obstacle se trouve entre deux nœuds alors la communication entre ces nœuds devient impossible. Il est aussi maintenant possible d'ajouter ou de supprimer des obstacles dans cette topologie. Pour cela il faut tout d'abord créer un obstacle.

5.2.2.1 Types d'obstacles implémentés

Trois types d'obstacles sont implémentés au sein de la version actuelle du module, deux sont des obstacles définis en deux dimensions. Le troisième type d'obstacle est un obstacle en trois dimensions.

Les obstacles 2D implémentés sont des obstacles pour lesquels on considère qu'ils ont une hauteur infinie pour pouvoir les utiliser que se soit en 2D ou en 3D avec JBotSim.

Le premier des deux obstacles 2D est un obstacle circulaire défini par un centre et un rayon. Il est important de noter qu'aucun nœud n'a le droit de se trouver à l'intérieur de ce type d'obstacle. Cet obstacle peut être vu comme un pilier de hauteur infinie.

Le deuxième obstacle 2D est lui une suite de segments continus défini par une liste de points. Par exemple la liste de points (A, B, C) définit les segments AB et BC. Il peut être représenté comme une succession de murs de hauteurs infinies.

Enfin, l'obstacle 3D est une facette rectangulaire définie par trois points A, B, C de telle sorte que le segment AB est perpendiculaire au segment AC. Cette facette est un mur de taille finie qui peut être incliné dans n'importe quel sens. Si lors de la création de l'obstacle les segments AB et AC ne sont pas perpendiculaires alors l'obstacle n'est pas généré et l'exception **NotPerpendicularException** est retournée.

⚠ Pour la création des deux obstacles 2D il faut fournir des **Point2D** (classe standard de Java) mais pour l'obstacle 3D il faut des points en trois dimensions, la classe à utiliser est donc **Point3D** (classe issue de JBotSim).

Voici un exemple de création d'instance de chacun des trois types d'obstacles :

— pour le cercle plein

```
Point2D center = new Point2D.Double(400, 400);
CircleObstacle circle=new CircleObstacle(center, 10);
```

— pour la suite de segments

```
List<Point2D> points = new ArrayList<Point2D>();
points.add(new Point2D.Double(20,80));
points.add(new Point2D.Double(200.8,600.1));
LinesObstacle lines = new LinesObstacle(points);
```

— pour la facette rectangulaire

```
Point3D a = new Point3D(20, 20, 0);
Point3D b = new Point3D(20, 200, 0);
Point3D c = new Point3D(20, 20, 500.5);
RectangularFacetObstacle facet = new
    RectangularFacetObstacle(a,b,c);
```

5.2.2.2 L'ajout et la suppression d'obstacles au sein d'une topologie

Pour ajouter un obstacle dans une topologie il faut utiliser la fonction `addObstacle(Obstacle obstacle, Topologie topology)` de la classe **ObstacleManager**. Le code suivant montre comment l'utiliser en considérant que les obstacles sont déjà définis et que le module est initialisé sur la topologie. Dans les exemples suivants `obs` est un objet de la classe **Obstacle**.

```
ObstacleManager.addObstacle(obs,topology);
```

Pour supprimer un obstacle dans une topologie il faut utiliser la fonction `removeObstacle(Obstacle obstacle, Topologie topology)` de la classe **ObstacleManager**. Le code suivant montre comment l'utiliser en considérant que les obstacles sont déjà définis et que le module est initialisé sur la topologie.

```
ObstacleManager.removeObstacle(obs,topology);
```

5.2.2.3 Création de types d'obstacles

Il est possible de créer de nouveaux types d'obstacles en implémentant l'interface **Obstacle**, cette interface définit trois méthodes :

— `boolean obstructLink(Node node1, Node node2)` qui doit retourner *true* si l'obstacle se trouve entre les deux nœuds et empêche l'établissement d'une communication entre eux.

— `Point3D pointAtMinimumDistance(Node node)` qui retourne le point de l'obstacle à distance minimum du nœud.

— `void paint(Graphics g)` qui contient le code pour afficher l'obstacle sur l'objet g de la classe Graphics.

Si ces trois méthodes sont implémentées au sein du nouveau type d'obstacles alors ce type d'obstacles pourra être utilisé comme les autres.

5.2.3 Activer ou désactiver la détection d'obstacle

Cette fonctionnalité consiste à permettre aux nœuds d'une topologie d'être avertis lorsqu'un ou plusieurs obstacles se trouvent dans leur zone de détection (sensing range). Cette fonctionnalité nécessite deux choses : l'implémentation de l'interface **ObstacleListener** par le nœud et qu'il signale à **ObstacleManager** qu'il souhaite être informé de la présence des obstacles dans sa zone de détection.

5.2.3.1 Implémenter ObstacleListener

Pour cela il faut définir une classe qui hérite de **Node** (comme il est d'usage pour coder un algorithme dans JBotSim) et qui implémente l'interface **ObstacleListener**. L'interface **ObstacleListener** définit une seule méthode à implémenter. Il s'agit de la méthode `onDetectedObstacles(List<Obstacle> obstacles)`. Elle est appelée par le module lorsqu'un ou plusieurs obstacles se retrouvent dans la zone de détection du nœud. L'argument `obstacles` contient la liste de ces obstacles.

```
public class MonNode extends Node implements ObstacleListener {

    public MonNode(){
        ...
    }

    public void onDetectedObstacles(List<Obstacle> obs){
        <votre code>
    }
}
```

5.2.3.2 Activation de la détection des obstacles

Pour signaler qu'un nœud souhaite détecter les obstacles proches il suffit ensuite d'appeler `addObstacleListener(ObstacleListener listener, Topology topology)` sur l'**ObstacleManager**, le listener est le nœud qui souhaite être notifié de la présence des obstacles et topology est la topologie à laquelle l'ObstacleListener doit être . Pour activer la détection, il faut connaître la topologie du nœud, deux solutions existent. Soit à la création du nœud on transmet la topologie sur lequel il va être accroché. Le code dans ce cas-là doit ressembler à :

```
public class MonNode extends Node implements ObstacleListener {
```

```
    public MonNode(Topology topology){
        ObstacleManager.addObstacleListener(this,topology);
    }

    public void onDetectedObstacles(List<Obstacle> obs){
        <votre code>
    }
}
```

Lorsqu'un nœud est ajouté par l'interface graphique, le constructeur du nœud appelé par JBotSim n'a pas d'argument donc il n'y a pas moyen de lui transférer la topologie à laquelle va appartenir le nœud. Dans ce cas, le seul moyen de connaître la topologie est de surcharger la méthode `onStart()` de la classe **Node**. Cette méthode est appelée par JBotSim au moment où le nœud est ajouté dans la topologie. Dans ce cas le code se résume à :

```
public class MonNode extends Node implements ObstacleListener {

    public MonNode(){...}

    public void onStart(){
        ObstacleManager.addObstacleListener(this, this.getTopology
            ());
    }

    public void onDetectedObstacles(List<Obstacle> obs){
        <votre code>
    }
}
```

Cette deuxième approche est à privilégier.

5.2.3.3 Désactivation de la détection des obstacles

Pour désactiver la détection des obstacles pour un nœud implémentant **ObstacleListener**, il faut appeler la méthode `removeObstacleListener(ObstacleListener listener, Topology topology)` de la classe **ObstacleManager**.

L'appel à cette méthode doit obligatoirement être fait au moment où le nœud, qui était informé de la présence d'obstacles, est supprimé. Cela peut être fait en surchargeant la méthode `onStop()` de la même manière que pour l'activation de la détection.

```
public void onStop(){
    ObstacleManager.removeObstacleListener(this, this.getTopology
        ());
}
```

5.2.4 Activer l'affichage des obstacles

Comme dit dans la section 5.2.1, il est possible d'activer automatiquement l'affichage des obstacles lors de l'initialisation. Néanmoins il est aussi possible de le faire manuellement en signalant à la **JTopology** que l'on veut utiliser la classe **ObstaclePainter** pour afficher des informations sur le background.

Il suffit d'utiliser le code suivant pour obtenir l'affichage des obstacles en plus des nœuds et des liens :

```
JTopology jTopology = new JTopology();
jTopology.addBackgroundPainter(new ObstaclePainter());
```

5.3 Fonctionnement interne du module `Obstacle`

Le fonctionnement du module va être vu étape par étape en commençant par étudier le cœur du module (incluant aussi la détection des obstacles par les nœuds), puis la partie graphique et enfin les types d'obstacles déjà implémentés (cercles et segments en 2D, plans en 3D).

5.3.1 Le cœur du module

Le cœur du module est la partie qui contient le code de l'ensemble des fonctionnalités du module à l'exception de la partie graphique. Il se trouve dans le package *jbotsimx.obstacle.core*.

5.3.1.1 L'interface Obstacle

Cette interface définit les trois seules méthodes qu'un obstacle doit implémenter. Ces trois fonctions sont :

- `boolean obstructLink(Node node1, Node node2)` qui doit retourner *true* si l'obstacle se trouve entre les deux nœuds et empêche l'établissement d'une communication entre eux.
- `Point3D pointAtMinimumDistance(Node node)` qui doit retourner le point de l'obstacle à distance minimum du nœud.
- `void paint(Graphics g)` qui contient le code pour afficher l'obstacle sur l'objet g de la classe Graphics.

5.3.1.2 L'interface ObstacleListener

Cette interface doit être implémentée par tout nœud qui souhaite être informé de la présence d'obstacles dans sa zone de détection. Elle définit une seule méthode à implémenter, il s'agit de `onDetectedObstacles(List<Obstacle>`

`obstacles)`. Cette méthode est appelée par le module dès qu'un ou plusieurs obstacles sont dans la zone de détection du nœud.

5.3.1.3 La classe ObstacleDetector

La classe **ObstacleDetector** gère l'information de la présence d'obstacles des **ObstacleListener**. Elle surveille tous les mouvements des nœuds de la topologie et, dès qu'un **ObstacleListener** se retrouve à proximité d'un ou plusieurs obstacles, elle notifie l'**ObstacleListener** en appelant sa méthode `onDetectedObstacles(List<Obstacle> obstacles)`.

5.3.1.4 La classe ObstacleLinkResolver

La classe **LinkResolver** de JBotSim est la classe qui gère par défaut le test de communication entre deux nœuds. Pour pouvoir tester si un obstacle empêche la communication entre deux nœuds nous avons créé la classe **ObstacleLinkResolver**. Cette classe hérite de **LinkResolver** et surcharge la méthode de test de communication `boolean isHeardBy(Node arg0, Node arg1)`. Il y est rajouté un test pour détecter si un des obstacles de la topologie bloque la communication en appelant la méthode `boolean obstructLink(Node node1, Node node2)` sur chaque obstacle de la topologie.

5.3.1.5 La classe ObstacleManager

La classe **ObstacleManager** est une classe statique, elle s'occupe d'initialiser le module Obstacle sur chaque topologie qu'on lui fournit en créant un **ObstacleDetector** pour cette topologie. Elle change aussi la classe de test de communication entre les nœuds de la topologie, qui est **LinkResolver**, par **ObstacleLinkResolver**, qui est celle du module. Enfin elle initialise le stockage des obstacles dans la topologie en stockant une liste d'obstacles dans les propriétés de la topologie sous le nom "obstacles".

Cette classe gère aussi l'ajout et la suppression des obstacles au sein des topologies grâce aux fonctions `void addObstacle(Obstacle o, Topology tp)` et `void removeObstacle(Obstacle o, Topology tp)`.
Enfin elle gère le stockage de chaque **ObstacleDetector** créé et est capable de retourner celui associé à une topologie donnée grâce à la fonction `ObstacleDetector getObstacleDetector(Topology tp)`, tout cela grâce à une **HashMap**.

5.3.2 La partie graphique

Nous allons maintenant étudier comment est géré l'affichage des obstacles au sein du module. L'ensemble du code gérant l'affichage se trouve dans le

package *jbotsimx.obstacle.ui*. L'affichage des obstacles se fait à l'aide de la classe **ObstaclePainter**.

5.3.2.1 La classe ObstaclePainter

Cette classe implémente l'interface **BackgroundPainter** fournie par JBotSim et implémente la méthode d'affichage `paintBackground(Graphics2D graphics2D, Topology topology)`. Elle permet d'insérer des dessins sur le rendu d'une topologie avant que JBotSim n'y dessine les nœuds et les arêtes. L'affichage se fait en parcourant la liste des obstacles de la topologie et en invoquant la méthode `void paint(Graphics g)` décrite précédemment sur chaque objet de type **Obstacle**.

5.3.3 Les exemples d'obstacles

Dans le cadre de test du module un certain nombre d'obstacles ont été implémentés, ces obstacles sont regroupés au sein du module dans le package *jbotsimx.obstacle.exemple*. Deux catégories d'obstacles sont présentes, obstacle 2D et obstacle 3D.

5.3.3.1 Notions mathématiques.

Pour implémenter les obstacles suivants un certain nombre de notions mathématiques ont été utilisées dont les vecteurs et les opérations sur les vecteurs. Nous décrivons ici les prérequis mathématiques pour comprendre les opérations que nous avons implémentées. Un vecteur est représenté de la façon suivante $\overrightarrow{A} = \begin{pmatrix} x \\ y \end{pmatrix}$ dans un espace 2D et $\overrightarrow{A} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$ dans un espace 3D, avec x, y, z étant trois réels.

La norme d'un vecteur. La norme d'un vecteur est la taille du vecteur. Elle se calcule dans le cadre d'un espace 2D par la formule suivante $\|\overrightarrow{A}\| = \sqrt{x^2 + y^2}$ et dans un espace 3D par $\|\overrightarrow{A}\| = \sqrt{x^2 + y^2 + z^2}$

Le produit vectoriel. Il s'agit d'une opération sur les vecteurs qui calcule un nouveau vecteur à partir des vecteurs suivants $\overrightarrow{A} = \begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix}$ et $\overrightarrow{B} = \begin{pmatrix} x_2 \\ y_2 \\ z_2 \end{pmatrix}$. La formule permettant de calculer le nouveau vecteur est

$$\overrightarrow{A} \wedge \overrightarrow{B} = \begin{pmatrix} y_1 z_2 - z_1 y_2 \\ z_1 x_2 - x_1 z_2 \\ x_1 y_2 - y_1 x_2 \end{pmatrix}$$

.Le vecteur obtenu est perpendiculaire aux deux autres vecteurs.

Le produit scalaire. Il s'agit d'une opération sur les vecteurs qui retourne un scalaire (réel) à partir de 2 vecteurs $\overrightarrow{A} = \begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix}$ et $\overrightarrow{B} = \begin{pmatrix} x_2 \\ y_2 \\ z_2 \end{pmatrix}$.
La formule retournant le scalaire est $\overrightarrow{A} \cdot \overrightarrow{B} = x_1 * x_2 + y_1 * y_2 + z_1 * z_2$.Il est important de noter que lorsque le produit scalaire de deux vecteurs est nul cela signifie que les vecteurs sont perpendiculaires.

5.3.3.2 Les obstacles 2D

Les obstacles 2D qui ont été implémentés sont des obstacles dont la hauteur est considérée comme infinie. Concrètement quelle que soit l'altitude de deux nœuds si l'obstacle se trouve entre eux sur les axes x et y alors la communication sera coupée entre ces nœuds.

Deux obstacles ont été implémentés, le premier est un obstacle circulaire et le deuxième est une suite de segments définie par une suite de points.

La classe CircleObstacle. Cette classe définit l'obstacle circulaire. Pour l'instancier il faut définir son centre C par un Point2D et son rayon par un nombre réel.

Pour tester si l'obstacle se trouve entre deux nœuds A et B et empêche la communication entre ces nœuds, on commence tout d'abord par regarder si la droite passant par A et B coupe l'obstacle. La droite coupe l'obstacle si et seulement si le produit vectoriel $\overrightarrow{AB} \wedge \overrightarrow{AC}$ est strictement inférieur au produit du rayon du cercle par la norme du vecteur $\overrightarrow{AB}$. Si la droite coupe l'obstacle alors on cherche à déterminer si le segment coupe l'obstacle. C'est le cas si les deux produits scalaires suivants sont positifs : $\overrightarrow{AB} \cdot \overrightarrow{AC}$ et $\overrightarrow{BA} \cdot \overrightarrow{BC}$.

Pour calculer le point, sur l'obstacle, à distance minimum d'un nœud A on calcule l'angle de la droite CA par rapport à la droite parallèle à l'axe des abscisses passant par le centre C de l'obstacle. Ensuite le point à distance minimum étant sur le cercle, on a seulement à calculer la position du point en calculant $x = C_x + rayon * cos(angle)$ et $y = C_y + rayon * sin(angle)$ avec (C_x,C_y) étant les coordonnées du centre C de l'obstacle.

La classe LinesObstacle. Cette classe définit un obstacle par une suite de segments. Pour l'instancier il faut lui fournir une liste de points $p_1, p_2, ..., p_n$. Les segments sont alors représentés par un couple de point (p_i, p_{i+1}).

Pour tester si l'obstacle se trouve entre deux nœuds A et B et empêche la communication entre ces nœuds, il faut tester chaque segment de l'obstacle pour savoir si l'un d'entre eux empêche la communication. Pour chaque segment CD on commence par tester si les boîtes englobantes du segment CD et du segment AB se coupent. Si c'est le cas alors c'est qu'il peut y avoir d'intersection. Ce test permet de réduire les calculs car ceux qui suivent sont plus longs. Si on a eu une intersection entre les boîtes englobantes il faut alors tester l'intersection entre les segments. Pour cela, la seule manière de faire est de tester s'il y a une intersection entre la droite représentant le segment CD et le segment AB et de tester aussi s'il y a une intersection entre la droite représentant le segment AB et la droite représentant le segment CD. Si et seulement si les deux intersections ont lieu alors il y a intersection entre le segment CD et le segment AB donc l'obstacle se trouve entre les deux nœuds et la communication entre les nœuds est coupée.
Le calcul de l'intersection entre la droite CD et le AB se fait par le calcul de deux produits scalaires, celui entre $\overrightarrow{CD}$ et $\overrightarrow{CA}$ et celui entre $\overrightarrow{CD}$ et $\overrightarrow{CB}$. Les produits scalaires permettent de savoir de quel côté du vecteur (droite) $\overrightarrow{CD}$ se trouvent les points A et B. Si le produit des deux produits scalaires est négatif alors c'est que les produits scalaires sont de signe contraire. Les points A et B sont alors de chaque côté de la droite CD. Il y a alors intersection car le segment AB coupe la droite CD.

Pour calculer le point, sur l'obstacle, à distance minimale d'un nœud P, il faut calculer le point à distance minimale sur chaque segment de l'obstacle et retourner celui qui est le plus proche de P. Pour trouver ce point sur un segment CD, il faut faire une projection du point P sur l'axe passant par C et D, qui a pour origine le point C et est représenté par le vecteur $\overrightarrow{CD}$. Cette projection est représentée par un nombre réel p qui représente le nombre de fois qu'il faut multiplier le vecteur $\overrightarrow{CD}$ pour obtenir le projeté de P sur la droite CD à partir du point C. Si ce nombre p est inférieur ou égal à 0 alors le point le plus proche de P sur le segment CD sera le point C. S'il est supérieur ou égal à 1 ce sera le point D. Cela est dû au fait que le projeté de P n'est pas sur le segment CD et que le point le plus proche se retrouve être l'une des extrémités. Dans le cas où p se trouve entre 0 et 1 alors le point à distance minimale est le projeté de P sur le segment CD. Il est calculé grâce à p qui permet lorsque l'on multiplie le vecteur $\overrightarrow{CD}$ par lui d'obtenir un nouveau vecteur. Lorsque ce nouveau vecteur est appliqué à partir de C on obtient alors les coordonnées du projeté de P sur le segment CD.

5.3.3.3 L'obstacle 3D

Un seul obstacle 3D a été implémenté, il s'agit d'une facette rectangulaire. Étant donné que la majorité des calculs impliquent de travailler avec des vecteurs lorsque que l'on est en 3D, une classe Vector3D a été créée. De plus une exception dédiée à l'obstacle a elle aussi été ajoutée.

La classe Vector3D. Cette classe implémente des vecteurs 3D, trois constructeurs ont été créés le premier prend trois nombres réels en entrée pour créer le vecteur, le deuxième prend deux **Point3D** (classe issue de JBotSim) A et B pour créer le vecteur $\overrightarrow{AB}$ le troisième prend un **Vector3D** en entrée et crée une copie de celui-ci.

Les opérations de produit vectoriel, de produit scalaire, de somme de deux vecteurs, de multiplication par un scalaire, de division par un scalaire ou de norme ont été implémentées pour faciliter les calculs.

De plus la méthode `Point3D getNewPointFrom(Point3D n)` permet d'obtenir un nouveau point après application du vecteur à partir du nœud n donné.

La classe NotPerpendicularException. Cette exception est levée par **RectangularFacetObstacle** lorsque les points donnés par l'utilisateur au constructeur de **RectangularFacetObstacle** ne forme pas deux droites perpendiculaires. Il s'agit principalement d'une classe d'avertissement.

La classe RectangularFacetObstacle. Cette classe implémente l'obstacle 3D qui est une facette rectangulaire. Pour l'instancier il faut lui fournir trois points E, F, G de telle sorte que les segments EF et EG soient perpendiculaires. Lors de sa création l'obstacle va aussi calculer son vecteur normal $\overrightarrow{N}$ comme s'il s'agissait d'un plan. Ce vecteur est perpendiculaire à la facette et il est important pour les calculs. Si lors de l'instanciation les segments EF et EG ne sont pas perpendiculaires alors l'exception **NotPerpendicularException** est levée.

Pour tester si l'obstacle se trouve entre deux nœuds A et B et empêche la communication entre ces nœuds, on calcule le produit scalaire entre $\overrightarrow{AB}$ et $\overrightarrow{N}$. Si la valeur retournée par ce produit est 0 alors le segment AB est perpendiculaire au vecteur normal de la facette. Vu que le vecteur normal de la facette est par définition perpendiculaire à la facette, on a alors le segment AB perpendiculaire à la facette. Si en plus le produit scalaire entre $\overrightarrow{EA}$ et $\overrightarrow{N}$ vaut 0 alors le segment se trouve sur le même plan que la facette.

Il faut alors tester qu'aucun des points du segment AB ne se trouvent sur la facette, ce qui couperait la communication. Pour cela il faut projeter la position des deux extrémités sur les axes EF et EG avec E comme origine des

axes. À partir de ces coordonnées il suffit de regarder leurs positions. Si les deux points sont à l'intérieur ou si les deux points coupent l'un des segments représentant le bord de la facette alors il n'y a pas de communication. Sinon la communication est possible.

Si la facette et le segment ne sont pas parallèles alors deux cas existent. Soit le point d'intersection entre la droite AB et le plan représentant la facette se trouve entre les points A et B (c'est-à-dire sur le segment AB) et alors il peut y avoir une intersection avec la facette. Soit il est en dehors du segment et il n'y a pas d'intersection possible entre la facette et le segment donc la communication entre A et B existe.

Dans le cas où le point d'intersection se trouve sur le segment AB il faut ensuite vérifier que le point d'intersection se trouve sur la facette grâce à la projection du point d'intersection sur les axes EF et EG avec E comme origine des axes. Enfin, il suffit juste de vérifier si les coordonnées obtenues sont bien à l'intérieur de la facette. Si c'est bien le cas alors l'obstacle empêche la communication entre A et B, sinon la communication existe entre les nœuds A et B.

Pour calculer le point, sur l'obstacle, à distance minimum d'un nœud P, on projette la position du nœud sur les axes EF et EG avec E comme origine des axes. On obtient alors les coordonnés 2D du nœud P sur ce plan. Si le point 2D P' est dans la facette alors on peut calculer le vecteur $\overrightarrow{EP'}$ dans le plan puis le convertir dans l'espace. Il suffit ensuite d'appliquer ce vecteur à partir de E pour obtenir les coordonnées de P' en 3D.

Si le point obtenu n'est pas dans la facette alors il suffit de regarder où se trouve le point grâce à ses coordonnés 2D pour trouver sur quelle bordure de la facette se trouve le point à distance minimal. On peut alors calculer les coordonnés 2D du point à distance minimale, puis enfin les convertir en coordonnés 3D.

5.4 Exemple d'utilisation du module

Nous présentons ici un exemple concret d'utilisation du module `Obstacle`, à travers un certain nombre de classes. Dans la classe **Main** (code 5.1) plusieurs obstacles sont crées : un sous la forme d'une suite de segments et trois sous la forme de cercle. L'ensemble est ensuite ajouté au sein de la topologie. Nous ajoutons également quelques nœuds du type **ObstacleNode** (code 5.2). Ces nœuds se déplacent vers une destination et changent de destination (aléatoirement) s'ils rencontrent un obstacle. Dans le cas où il y a plusieurs obstacles à proximité, seul l'obstacle le plus proche est considéré. La figure 5.1 est une capture d'écran montrant le résultat graphique de l'implémentation présentée ci-dessous.

Ces classes illustrent la plupart des fonctionnalités qui sont utilisables au sein du module, la création des obstacles 2D du module et leurs ajouts au sein d'une topologie ainsi que la partie concernant la gestion de la détection d'obstacles au sein d'un nœud implémentant **ObstacleListener**.

```
public class Main {

    public static void main(String[] args) {
        Topology tp = new Topology(800,600);
        tp.setDefaultNodeModel(ObstacleNode.class);
        JTopology jtp = new JTopology(tp);

        ObstacleManager.init(tp, jtp);

        List<Point2D> points = new ArrayList<>();
        points.add(new Point2D.Double(100, 100));
        points.add(new Point2D.Double(200, 400));
        points.add(new Point2D.Double(500, 500));
        points.add(new Point2D.Double(600, 300));

        LinesObstacle lo = new LinesObstacle(points);
        ObstacleManager.addObstacle(lo, tp);

        CircleObstacle co = new CircleObstacle(new Point2D.Double
            (400, 400), 10);
        ObstacleManager.addObstacle(co, tp);
        co = new CircleObstacle(new Point2D.Double(200, 200), 20);
        ObstacleManager.addObstacle(co, tp);
        co = new CircleObstacle(new Point2D.Double(600, 200), 80);
        ObstacleManager.addObstacle(co, tp);

        tp.pause();
        Node n = new ObstacleNode();
        tp.addNode(400, 500, n);
        n = new ObstacleNode();
        tp.addNode(200,500,n);
        n = new ObstacleNode();
        tp.addNode(400,300,n);
        tp.resume();
        new JViewer(jtp);
    }

}
```

Classe 5.1 – Classe Main

```
public class ObstacleNode extends Node implements ObstacleListener
    {

    private static Random r = new Random();
```

```
    public ObstacleNode() {
        setProperty("target", new Point(r.nextInt(800), r.nextInt
            (600)));
        setSensingRange(25);

    }

    @Override
    public void onStart() {
        ObstacleManager.addObstacleListener(this, getTopology());
    }

    @Override
    public void onStop() {
        ObstacleManager.removeObstacleListener(this, getTopology()
            );
    }

    @Override
    public void onDetectedObstacles(List<Obstacle> obstacles) {
        Point3D p = obstacles.get(0).pointAtMinimumDistance(this);
        Point3D tmp = new Point3D(this.getX()+Math.cos(this.
            getDirection()), this.getY() + Math.sin(this.
            getDirection()),0);
        double distance = p.distance(tmp);

        for (Obstacle o : obstacles){
            Point3D ptmp = o.pointAtMinimumDistance(this);
            if (ptmp.distance(tmp) < distance){
                p = ptmp;
                distance = p.distance(tmp);
            }
        }
        Point3D tmp2;
        Point3D n = new Point3D(this.getX(),this.getY(),this.getZ
            ());
        if(p.distance(tmp) < this.getSensingRange() && p.distance(
            tmp) < p.distance(n)){
            do {
                setProperty("target", new Point(r.nextInt(800), r.
                    nextInt(600)));
                Point target = (Point)getProperty("target");
                double direction = Math.atan2(target.getX() - this
                    .getX(), - (target.getY() - this.getY())) -
                    Math.PI/2;
                tmp2 = new Point3D(this.getX() + Math.cos(
                    direction), this.getY() + Math.sin(direction)
                    ,0);
            }
            while(p.distance(tmp2) < p.distance(n));
        }
    }
```

```
    @Override
    public void onClock() {
        Point target = (Point)getProperty("target");
        setDirection(target);
        move();
        if (distance(target) < 15)
            setProperty("target", new Point(r.nextInt(800), r.
                nextInt(600)));
    }

}
```

Classe 5.2 – Classe ObstacleNode

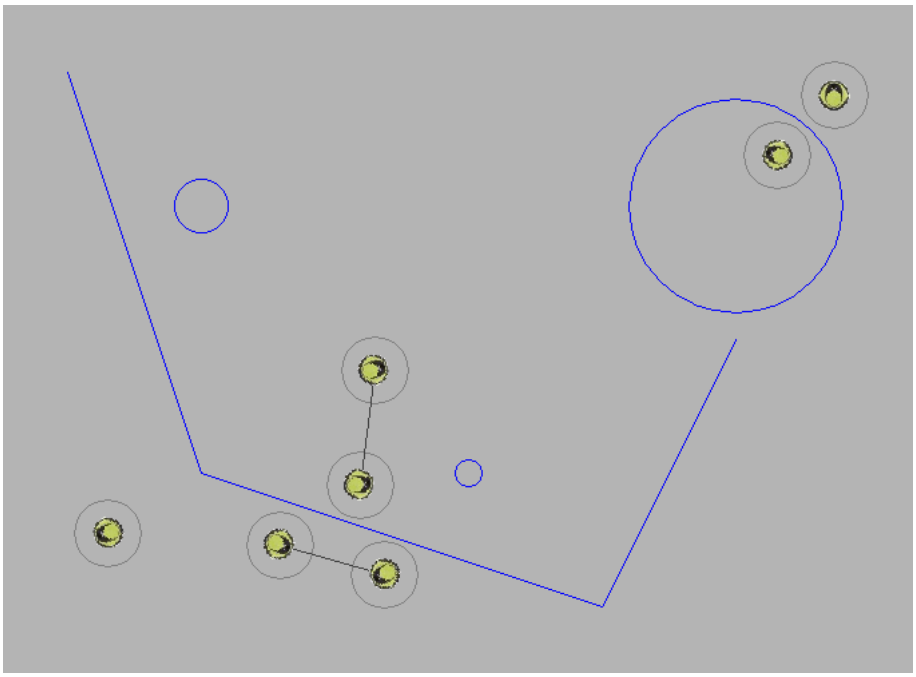


FIGURE 5.1 – Capture d'écran d'un exemple de scénario utilisant des obstacles.

5.5 Conclusion

Le module obstacle est disponible en téléchargement sur le dépot Bitbucket [8] sous la forme d'une librairie (.jar) utilisable directement avec JBotSim. Le code source est également disponible au téléchargement. Ce module est actuellement utilisé dans le cadre d'un cours de master 2 à l'Université de Bordeaux.

Conclusion

Dans cette thèse, nous nous sommes intéressés aux problèmes liés aux systèmes distribués ayant une topologie dynamique. Nous avons représenté la topologie de ces systèmes à l'aide de graphes dynamiques. Dans le cadre de ces graphes dynamiques, nous avons étudié des problèmes évoluant autour de la contrainte du diamètre temporel borné dans le temps qui permet de distinguer la perte d'appareils et les déconnexions temporaires (groupe DTN). Cette contrainte oblige les nœuds à communiquer régulièrement entre eux et donc contraint les mouvements de ces nœuds. Cette considération a été récurrente dans nos travaux. Voici un résumé des contributions que nous avons apportées.

Connexité temporelle

Dans le chapitre 2, nous avons présenté un algorithme pour calculer la fermeture transitive des trajets. Cette dernière nous permet de détecter si un graphe dynamique est temporellement connexe ou pas. Comme nous l'avons montré, il est possible, à l'aide des fermetures transitives des trajets non-stricts d'un ensemble précis de sous-graphes dynamiques d'un graphe dynamique, de vérifier si ce dernier respecte la contrainte du diamètre temporel. Cette contrainte impose qu'à chaque instant t il existe un trajet pour tous les couples de nœuds dont la durée est bornée.

Nous avons proposé deux algorithmes dédiés au calcul des fermetures transitives strictes et non-strictes d'un graphe dynamique nous permettant de dire si le graphe dynamique est temporellement connexe ou pas. Nous avons aussi présenté une analyse des complexités en temps de nos algorithmes et les avons comparées à deux algorithmes existants que nous avons adaptés pour l'occasion.

Exploration d'arbres

Dans le chapitre 3, nous avons étudié le problème de l'exploration d'arbres à l'aide d'agent mobiles et cela sous la contrainte du diamètre temporel borné. Nous avons présenté un certain nombre d'algorithmes *offlines* non-optimaux d'exploration d'arbres sous cette contrainte. Nous avons aussi présenté un nou-

vel algorithme permettant de calculer la solution optimale en examinant l'ensemble des explorations possibles de l'arbre sous la contrainte du diamètre temporel. Néanmoins, le coût en temps de cet algorithme est exponentiel en le nombre de nœuds de l'arbre et en le nombre d'agents. Nous avons ensuite introduit un algorithme *online* d'exploration d'arbres sous la contrainte du diamètre temporel. Enfin, nous avons simulé cet algorithme en faisant varier des paramètres tels que le nombre d'agent et le nombre de nœuds pour étudier ses performances.

Forêt couvrante

Dans le chapitre 4, notre contribution concerne le passage d'un algorithme existant dans un modèle de calcul abstrait sur les graphes vers un modèle de passage de messages synchrone. L'idée de l'algorithme est de maintenir une forêt couvrante au sein du réseau dynamique. Cet algorithme permet de s'assurer que chaque nœud du graphe dynamique appartient à tout instant à un groupe de nœuds (l'arbre auquel il appartient) et que chaque groupe possède à tout instant un leader (la racine de l'arbre). Nous avons également présenté des résultats de simulation de cet algorithme sur des traces existantes, qui montrent qu'il est exploitable en contexte réel.

Module obstacle

Dans le chapitre 5, nous avons présenté le module `Obstacle` que nous avons créé et implémenté au sein du simulateur JBotSim. Ce module nous permet d'intégrer des obstacles qui empêchent les communications entre les nœuds. Ce module est capable de gérer les obstacles 2D et 3D et peut informer les nœuds de la présence d'un obstacle à proximité. Ce module est mis à disposition de la communauté en libre téléchargement (licence LGPL).

Questions ouvertes et travaux futurs

Dans le cadre de l'exploration sous la contrainte du diamètre temporel nous nous sommes concentrés sur le problème de l'exploration d'arbres. Nous y avons présentés un certain nombre d'algorithme d'exploration mais il serait intéressant d'étudier si d'autres méthodes d'exploration sont plus intéressantes. Le problème de l'exploration peut aussi être étendu aux graphes généraux, où l'anonymité des nœuds pose des problèmes dans la version distribuée qui ne se posaient pas dans le cas des arbres (p.ex. apparition de cycles non détectables et de symmétries dans le réseau).

Enfin, la question se pose de savoir comment (et si) nos algorithmes pourraient être transposés dans un contexte d'exploration d'espace réel (et non seulement de graphes). Le module `obstacle` que nous avons présenté peut

servir à simuler de tels scénarios au sein de JBotSim, ce qui représente donc une étape dans cette direction.

Bibliographie

[1] S. Abbas, M. Mosbah, and A. Zemmari. Distributed Computation of a Spanning Tree in a Dynamic Graph by Mobile Agents. In *2006 IEEE International Conference on Engineering of Intelligent Systems*, pages 1–6, 2006.

[2] D. Aldous and J. Fill. *Reversible Markov Chains and Random Walks on Graphs*. 2002.

[3] I. Averbakh and O. Berman. (p - 1)(p + 1)-approximate algorithms for p-traveling salesmen problems on a tree with minmax objective. *Discrete Applied Mathematics*, 75(3) :201–216, 1997.

[4] B. Awerbuch, I. Cidon, and S. Kutten. Optimal Maintenance of a Spanning Tree. *J. ACM*, 55(4) :18 :1–18 :45, Sept. 2008.

[5] B. Awerbuch and S. Even. Efficient and Reliable Broadcast is Achievable in an Eventually Connected Network. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, pages 278–281. ACM, 1984.

[6] H. Baala, O. Flauzac, J. Gaber, M. Bui, and T. El-Ghazawi. A self-stabilizing distributed algorithm for spanning tree construction in wireless ad hoc networks. *Journal of Parallel and Distributed Computing*, 63(1) :97–104, Jan. 2003.

[7] J. Bar-Ilan and D. Zernik. Random leaders and random spanning trees. In *Distributed Algorithms*, number 392 in Lecture Notes in Computer Science, pages 1–12. Sept. 1989.

[8] M. Barjon. Module obstacle JBotSim. `https://bitbucket.org/mbarjon/jbotsimobstaclemodule`.

[9] M. Barjon, A. Casteigts, S. Chaumette, C. Johnen, and Y. M. Neggaz. Maintaining a Spanning Forest in Highly Dynamic Networks : The Synchronous Case. In *Principles of Distributed Systems*, number 8878 in Lecture Notes in Computer Science, pages 277–292. Springer, Dec. 2014.

[10] M. Barjon, A. Casteigts, S. Chaumette, C. Johnen, and Y. M. Neggaz. Maintaining a spanning forest in highly dynamic networks : The synchronous case. *CoRR*, abs/1410.4373, 2014.

[11] M. Barjon, A. Casteigts, S. Chaumette, C. Johnen, and Y. M. Neggaz. Testing Temporal Connectivity in Sparse Dynamic Graphs. *arXiv :1404.7634 [cs]*, Apr. 2014. arXiv : 1404.7634.

[12] M. Barjon, A. Casteigts, S. Chaumette, C. Johnen, and Y. M. Neggaz. Un algorithme de test pour la connexité temporelle des graphes dynamiques de faible densité. *arXiv :1405.0170 [cs]*, May 2014. arXiv : 1405.0170.

[13] T. Bernard, A. Bui, and D. Sohier. Universal adaptive self-stabilizing traversal scheme : Random walk and reloading wave. *Journal of Parallel and Distributed Computing*, 73(2) :137–149, 2013.

[14] S. Bhadra and A. Ferreira. Complexity of Connected Components in Evolving Graphs and the Computation of Multicast Trees in Dynamic Networks. In *Ad-Hoc, Mobile, and Wireless Networks*, number 2865 in Lecture Notes in Computer Science, pages 259–270. Oct. 2003.

[15] P. Brass, F. Cabrera-Mora, A. Gasparri, and J. Xiao. Multirobot Tree and Graph Exploration. *IEEE Transactions on Robotics*, 27(4) :707–717, 2011.

[16] J. Burman and S. Kutten. Time Optimal Asynchronous Self-stabilizing Spanning Tree. In *Distributed Computing*, number 4731 in Lecture Notes in Computer Science, pages 92–107. Sept. 2007.

[17] A. Casteigts. Model driven capabilities of the DA-GRS model. In *Proc. of 1st Intl. Conference on Autonomic and Autonomous Systems (ICAS'06)*, pages 24–32, 2006.

[18] A. Casteigts. Jbotsim : a tool for fast prototyping of distributed algorithms in dynamic networks. In *Proceedings of the 8nd international ICST conference on simulation tools and techniques, SIMUTools'15, Athens, Greece*, 2015.

[19] A. Casteigts, S. Chaumette, and A. Ferreira. Characterizing topological assumptions of distributed algorithms in dynamic networks. In *Proc. of SIROCCO'09*, pages 126–140. Springer, 2009.

[20] A. Casteigts, S. Chaumette, F. Guinand, and Y. Pigné. Distributed Maintenance of Anytime Available Spanning Trees in Dynamic Networks. In *Ad-hoc, Mobile, and Wireless Network*, number 7960 in Lecture Notes in Computer Science, pages 99–110. July 2013.

[21] A. Casteigts and P. Flocchini. Deterministic algorithms in dynamic networks : Problems, analysis, and algorithmic tools. Technical report, Commissioned by Defense Research and Development Canada (DRDC), 2013.

[22] A. Casteigts, P. Flocchini, E. Godard, N. Santoro, and M. Yamashita. Expressivity of time-varying graphs. In *19th International Symposium on Fundamentals of Computation Theory (FCT)*, Liverpool, United Kingdom, 2013.

[23] A. Casteigts, P. Flocchini, B. Mans, and N. Santoro. Deterministic Computations in Time-Varying Graphs : Broadcasting under Unstructured Mobility. In *Theoretical Computer Science*, number 323 in IFIP Advances in Information and Communication Technology, pages 111–124. Sept. 2010.

[24] A. Casteigts, P. Flocchini, B. Mans, and N. Santoro. Measuring temporal lags in delay-tolerant networks. *IEEE Transactions on Computers*, 63(2), 2014.

[25] A. Casteigts, P. Flocchini, B. Mans, and N. Santoro. Shortest, fastest, and foremost broadcast in dynamic networks. *Int. Journal of Foundations of Computer Science*, 26(4) :499–522, 2015.

[26] A. Casteigts, P. Flocchini, B. Mans, and N. Santoro. Shortest, Fastest, and Foremost Broadcast in Dynamic Networks. *International Journal of Foundations of Computer Science*, 26(04) :499–522, 2015.

[27] A. Casteigts, P. Flocchini, W. Quattrociocchi, and N. Santoro. Time-varying graphs and dynamic networks. *International Journal of Parallel, Emergent and Distributed Systems*, 27(5) :387–408, Oct. 2012.

[28] A. Casteigts, R. Klasing, Y. M. Neggaz, and J. G. Peters. Efficiently Testing T-Interval Connectivity in Dynamic Graphs. In *Algorithms and Complexity*, number 9079 in Lecture Notes in Computer Science, pages 89–100. May 2015.

[29] A. E. Clementi, C. Macci, A. Monti, F. Pasquale, and R. Silvestri. Flooding Time in edge-Markovian Dynamic Graphs. In *Proceedings of the Twenty-seventh ACM Symposium on Principles of Distributed Computing*, PODC '08, pages 213–222. ACM, 2008.

[30] J. Czyzowicz, A. Pelc, and M. Roy. Tree exploration by a swarm of mobile agents. In *Principles of Distributed Systems*, volume 7702 of *Lecture Notes in Computer Science*, pages 121–134. 2012.

[31] S. Das. Mobile agents in distributed computing : Network exploration. *Bulletin of EATCS*, 1(109), 2013.

[32] G. Dudek, M. Jenkin, E. Milios, and D. Wilkes. Robotic exploration as graph construction. *Robotics and Automation, IEEE transactions on*, 7(6) :859–865, 1991.

[33] M. Dynia, J. Łopuszański, and C. Schindelhauer. Why Robots Need Maps. In G. Prencipe and S. Zaks, editors, *Structural Information and Communication Complexity*, number 4474 in Lecture Notes in Computer Science, pages 41–50. Springer Berlin Heidelberg, June 2007.

[34] A. Ferreira. On models and algorithms for dynamic communication networks : The case for evolving graphs. In *Proc. ALGOTEL*, 2002.

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

[36] P. Fraigniaud, L. Gasieniec, D. R. Kowalski, and A. Pelc. Collective tree exploration. *Networks*, 48(3) :166–177, 2006.

[37] G. Frederickson, M. Hecht, and C. Kim. Approximation algorithms for some routing problems. *SIAM Journal on Computing*, 7(2) :178–193, 1978.

[38] M. R. Gary and D. S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-completeness*. WH Freeman and Company, New York, 1979.

[39] C. Gómez-Calzado, A. Casteigts, M. Larrea, and A. Lafuente. A connectivity model for agreement in dynamic systems. In *21st Int. Conference on Parallel Processing (EUROPAR)*, volume 9233 of *Lecture Notes in Computer Science*, pages 333–345, 2015.

[40] F. Harary and G. Gupta. Dynamic graph models. *Mathematical and Computer Modelling*, 25(7) :79–87, 1997.

[41] A. Israeli and M. Jalfon. Token Management Schemes and Random Walks Yield Self-stabilizing Mutual Exclusion. In *Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing*, PODC '90, pages 119–131, New York, NY, USA, 1990. ACM.

[42] A. Kravchik and S. Kutten. Time Optimal Synchronous Self Stabilizing Spanning Tree. In *Distributed Computing*, number 8205 in Lecture Notes in Computer Science, pages 91–105. Oct. 2013.

[43] F. Kuhn, N. Lynch, and R. Oshman. Distributed Computation in Dynamic Networks. In *Proceedings of the Forty-second ACM Symposium on Theory of Computing*, STOC, pages 513–522. ACM, 2010.

[44] I. Litovsky, Y. Métivier, and E. Sopena. Graph relabelling systems and distributed algorithms. *Handbook of graph grammars and computing by graph transformation*, pages 1–56, 2001.

[45] I. Muslea. The k-Vehicle Routing Problem in Trees. M.S. thesis, Department of Statistics and Computer Science, West Virginia University, 1996.

[46] I. Muslea. The very offline k-vehicle routing problem in trees. In *Computer Science Society, 1997. Proceedings., XVII International Conference of the Chilean*, pages 155–163, Nov. 1997.

[47] D. Pajak. *Algorithms for deterministic parallel graph exploration*. PhD thesis, 2014. Thèse de doctorat dirigée par Klasing, Ralph et Kosowski, Adrian Informatique Bordeaux 2014.

[48] N. S. Rao, S. Kareti, W. Shi, and S. S. Iyengar. Robot navigation in unknown terrains : Introductory survey of non-heuristic algorithms. Technical Report ORNL/TM-12410, Citeseer, Oak Ridge National Laboratory, 1993.

[49] J. Scott, R. Gass, J. Crowcroft, P. Hui, C. Diot, and A. Chaintreau. CRAWDAD dataset cambridge/haggle (v. 2009-05-29), 2009. Published : Downloaded from http ://crawdad.org/cambridge/haggle/20090529/imote traceset : imote.

[50] S. Thrun. Robotic mapping : A survey. In G. Lakemeyer and B. Nebel, editors, *Exploring artificial intelligence in the new millennium*, volume 135. Morgan Kaufmann, 2002.

[51] H. Wang, M. Jenkin, and P. Dymond. Graph exploration with robot swarms. *International Journal of Intelligent Computing and Cybernetics*, 2(4) :818–845, Nov. 2009.

[52] H. Wang, M. Jenkin, and P. Dymond. It can be beneficial to be "lazy" when exploring graph-like worlds with multiple robots. In *Proceedings of the IASTED International Conference*, volume 1, page 200, 2009.

[53] J. Whitbeck, M. Dias de Amorim, V. Conan, and J.-L. Guillaume. Temporal reachability graphs. In *Proc. of MOBICOM'12*, pages 377–388. ACM, 2012.

[54] B. B. Xuan, A. Ferreira, and A. Jarry. Computing shortest, fastest, and foremost journeys in dynamic networks. *International Journal of Foundations of Computer Science*, 14(02) :267–285, 2003.