Des aspects locaux dans les algorithmes distribués

Résumé : Dans cette thèse, nous étudions différents aspects liés à la localité des algorithmes distribués. D'abord, dans le modèle avec échange de messages, nous donnons des algorithmes déterministes sous linéaires en temps pour la construction de décompositions peu denses de graphes et des applications sous-jacentes. Nous donnons aussi des algorithmes ayant une complexité en temps mieux que sous linéaire pour la construction de sous graphes couvrants ayant peu d'arêtes et un petit facteur d'étirement. Ensuite, nous étudions le problème de la poignée de main distribuée (ou calcul de couplage en temps constant) dans le modèle avec agents mobiles ainsi que deux autres extentions de ce problème. Parmis nos résultats, nous obtenons de nouvelles idées pour améliorer les algorithmes existants dans le modèle avec échange de messages. Dans une approche plus formelle, nous montrons à travers plusieurs exemples comment on peut coder des algorithmes distribués complexes en utilisant le formalisme des systèmes de réétiquetage. Dans une approche plus pratique, nous exposons nos contributions dans le développement de la plateforme logicielle ViSiDiA pour la simulation et la visualisation d'algorithmes distribués.

Mots clés : Algorithmes distribués, échanges de messages, agents mobiles, systèmes de réétiquetage, décomposition de graphes, sous graphes couvrants, couplages, complexité en temps, localité, ViSiDiA.

Local aspects in distributed algorithms

Abstract: In this thesis, we study some locality aspects in distributed algorithms. In the $\mathcal{CONGEST}$ message passing model, we provide sublinear deterministic algorithms for constructing sparse graph decompositions and related applications. In the $\mathcal{LOCAL}$ message passing model, we provide deterministic algorithms for constructing sparse graph spanners with low stretch in time better than sublinear. In the mobile agent model, we provide randomized algorithms for the handshake problem, i.e., computing a matching in constant time. We also study two variants of this problem and obtain new ideas in order to improve existing algorithms in the classical message passing model. In the more formal model based on relabeling systems, we show how to encode and combine many basic building blocks in order to obtain a uniform and unified encoding of many distributed algorithms. Finally, we describe the ViSiDiA software plateform and give our main contribution for the visualisation and simulation of distributed algorithms.

Keywords: Distributed algorithms, message passing, mobile agents, relabeling systems, graph decompositions, graph spanners, matching, time complexity, locality, ViSiDiA.

N° d'ordre : 3305

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ BORDEAUX I

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

Par **Bilel Derbel**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

Local Aspects in Distributed Algorithms

Soutenue le : 7 Décembre 2006

Après avis des rapporteurs :

Pierre Fraigniaud	Directeur de Recherche CNRS
David Peleg	Professeur

Devant la commission d'examen composée de :

Pierre Fraigniaud	Directeur de Recherche CNRS	Rapporteur
Cyril Gavoille	Professeur	Examinateur
Guy Melançon ...	Professeur	Examinateur
Yves Métivier	Professeur	Co-Directeur
Mohamed Mosbah	Professeur	Directeur
David Peleg	Professeur	Rapporteur

– 2006 –

à Moncef, Jamila, Najet, Houda, Nizar et Mohamed.

Remerciements

Mes premiers remerciements vont tout naturellement à mes directeurs de thèse. À Mohamed Mosbah, pour son encadrement et sa bienveillance. À Yves Métivier, pour ses conseils précieux et sa confiance.

Cyril Gavoille m'a fait l'immense plaisir de présider mon jury de thèse. Bien avant cela, son incroyable énergie et son infatiguable curiosité scientifique n'ont cessées de me stimuler. Je saisie cette occasion pour le remercier très chaleureusement.

Je suis également profondément reconnaissant à David Peleg et à Pierre Fraigniaud qui m'ont fait l'honneur de lire ce manuscrit, puis d'assister à mon jury. En praticulier, David Peleg m'a fait l'immense bonheur de découvrir une personne admirablement gentille, que je ne connaissait auparavant qu'à travers un livre devenu aujourd'hui un compagnon très chèr.

Guy Melançon a immédiatement accepté d'être membre de mon jury. Je l'en remercie ainsi que pour l'intérêt qu'il a porté à mon travail.

Je suis fortement reconnaissant à Jean-Francois Marckert qui a consacré beacoup de temps à m'enseigner la rigueur et la subtilité des analyses probabilistes. Je remercie également Akka Zemmari pour son dévouement et son aide.

Je tiens aussi à remercier les équipes enseignantes, le personnel administratif et les collègues chercheurs à l'IUT de Bordeaux 1, à l'ENSEIRB et au LaBRI pour leur gentillesse et leur disponibilité. Aussi, les derniers mois de préparation de ce manuscrit auraient pu se prolonger sans compter sur l'acceuil amical de l'équipe MoVe du LIF à Marseille.

Merci à ma soeur Houda et à Olivier pour m'avoir aidé à soigner la rédaction de ce document.

Merci à tous mes amis Bordelais pour avoir tant donné, en particulier pour leur soutien sans faille lors de la soutenance. Un très grand merci à Rahim, Niäma et leur petit Rayan à Marseille. Merci à Soufienne, Asma et leur petit Anis à Grenoble.

Mes amis et mes proches en France et en Tunisie ont été d'un soutien inestimable. Rien que par leur présence, ils ont contribués à faire de cette thèse une aventure humaine des plus passionnates et des plus belles. Je les remercie infinement.

Enfin, je remercie du fond du coeur mon père et ma mère, ma tante, ma soeur, mes deux frères, mes deux belles soeurs, sans oublier ma petite nièce Najet et mon petit neveu Ayoub. Qu'ils sachent combien leur soutien et leur amour ont été déterminants.

Résumé

Dans cette thèse, nous adaptons quatre approches différentes et néanmoins complémentaire, pour étudier différents aspects liées à la nature locale des algorithmes distribués. La nature locale des algorithmes distribués est une conséquence du manque d'informations globales qui renseignent sur l'état globale du système distribué considéré.

D'abord, nous étudions la complexité en temps de la construction de structures locales de graphes. Les structures auxquelles nous nous intéressons peuvent être considérées comme étant des structures de données efficaces qui sont utilisées dans plusieurs applications distribuées tels que les synchroniseurs, le routage, le calculs de plus court chemins, etc. Nous donnons des algorithmes distribués déterministes sous linéaires en temps pour la construction de décompositions peu denses et de sous graphes couvrants ("spanners") en utilisant des messages de petite taille ($\mathcal{CONGEST}$ model). Dans le cas des spanners, nous présentons de nouvelles techniques permettant d'obtenir des algorithmes distribués avec une complexité en temps mieux que sous linéaire dans le cas déterministe et logarithmique en moyenne dans le cas probabiliste dans le modèle dit "local" de calculs distribués ($\mathcal{LOCAL}$ model).

Dans un cadre plus formel, nous étudions la conception d'algorithmes distribués de façon formelle, unifiée et compréhensible à travers les systèmes de réétiquetages. Notre contribution est de montrer à travers plusieurs exemples comment on peut combiner quelques techniques de bases pour modéliser des algorithmes plus complexes.

Dans le modèle avec agents mobiles, nous nous intéressons au problème du "handshake" (poignée de mains ou calcul de couplage en temps constant) et nous donnons une solution efficace basée sur k marches aléatoires de k agents mobiles. Des extensions et des applications sont présentées pour les systèmes de réétiquetages. Nous montrons comment émuler nos algorithmes dans le modèle avec échanges de messages. Nous obtenons alors des résultats qui améliorent les algorithmes connus existants. Ceci montre de façon pratique que le paradigme des agents mobiles peut aider à avoir une nouvelle vision des systèmes distribués et à trouver de nouvelles solutions et idées à des problèmes bien classiques.

Dans un cadre plus pratique, nous présentons nos travaux autour de la plate-forme logicielle ViSiDiA pour la simulation et la visualisation d'algorithmes distribués. Nos principales contributions sont la mise en place d'une version distribuée du logiciel, d'une version supportant le modèle synchrone de calculs distribués et enfin une version supportant le modèle avec agents mobiles. Ces différentes versions sont destinées aussi bien aux chercheurs qu'aux étudiants pour aider à l'étude et à l'expérimentation d'algorithmes distribués.

Abstract

In a distributed system many actors are operating in a cooperative manner in order to perform a specific task. Since no global view of the system is available, the actors must communicate together and exchange their knowledge. The main challenge when designing a distributed algorithm is to lean on local information in order to provide a global solution. In this thesis, we adopt four different approaches in order to acquire a better theoritical understanding of local aspects of distributed algorithms.

In the first part, we consider a network of processors which can communicate by exchanging messages. We focus on the time needed to construct network global data structures. In Chapter 1, we study the problem of partitioning the network into a set of clusters having a small radius and few inter-cluster edges. We obtain new sublinear time deterministic and randomized algorithms which improves previous constructions and related applications such as network synchronizers and graph spanners. The problem of constructing sparse low stretch spanners efficiently is studied more deeply in Chapter 2. In fact, for any n-node graph, we provide deterministic algorithms that construct constant stretch spanners with $o(n^2)$ edges in $o(n^\epsilon)$ time for any constant $\epsilon > 0$, against $O(n^c)$ time for previous constructions where $c < 1$ is a positive constant depending on the stretch. Our algorithms make use of an efficient clustering technique based on breaking the symmetry using independent dominating sets. Many logarithmic randomized implementations are also presented.

In the second part of this thesis, we consider the problem of encoding distributed algorithms in a formal and unified way independently of the underlying distributed system or communication structure. We use relabeling systems and local computation as a mathematical tool-box. Within this framework a distributed algorithm is encoded by mean of local relabeling rules, that is, the labels attached to nodes and edges are modified locally on a subgraph of fixed radius. We give an encoding of some basic traversal algorithms such as the broadcast, the convergecast and the PIF technique (Propagation of Information with Feedback). We show how these basic techniques can be combined in order to provide a formal and comprehensible encoding of sophisticated algorithms such as the Prim's MST (minimum spanning tree) algorithm. The work described in this part can be viewed as a first step toward a generic powerful framework for designing and proving distributed algorithms using relabeling systems.

In the third part, we adopt a mobile agent perspective. We consider a set of k agents scattered over a network. The agents are autonomous computation entities which are able

to move from node to node and to communicate together by writing and reading information using white-boards. In Chapter 4, we focus on the handshake problem which consists in computing a set a disjoint edges. The ruling measure of a handshake algorithm is the handshake number, that is the number of disjoint edges. We give a mobile agent algorithm for this problem and we study its efficiency. The algorithm is based on independent random walks of the agents. For general graphs, our handshake number improves all the previous results. We also obtain a handshake number which is optimal up to a constant factor for almost regular graphs. We show how to emulate our mobile agent algorithm in the message passing model without any loss of efficiency. We obtain a larger handshake number, which shows that using mobile agents can provide novel ideas to solve some well studied problems in the message passing model. Our handshake algorithm can be applied in order to resolve some symmetry breaking problems such as maximal matching and edge coloring. It can also be applied in order to implement relabeling systems in the basic edge local computation model. In Chapter 5, we extend our handshake algorithm for other basic local computation models, namely the closed star model and the open star model. We also give a performance study of our algorithm and obtain improved results when adapting them to a message passing model. Our mobile agent work is concluded by a generic framework for implementing any relabeling system. The main originality of our framework is to abstract the design of a distributed algorithm from its practical implementation.

In the last part of this thesis, we adopt a more educational and experimental approach of distributed algorithms. We present a software platform called *ViSiDiA* for the visualization and simulation of distributed algorithms. This tool provides a friendly graphical user interface and an easy to use library for writing, running and visualizing distributed algorithms in the asynchronous/synchronous message-passing/mobile-agent model. For researchers, *ViSiDiA* can be a software support to experiment new algorithms and ideas. For students, it can help to test the algorithms studied in class and it leads to a better understanding of the local nature of distributed algorithms.

Contents

Introduction

Le cadre général de cette thèse est l'étude de différents aspects liés à la *nature locale* des algorithmes distribués. La nature locale d'un algorithme distribué provient essentiellement du manque de vision globale concernant l'état du système lors de la résolution d'un problème. En effet, seule une information locale est disponible aux différentes entités de calculs pour produire une solution globalement cohérente. Par ailleurs, une information locale est souvent suffisante pour pouvoir résoudre de façon efficace des problèmes qui du point de vue des protocoles classiques nécessiteraient une information globale. En d'autres termes, une information globale n'est pas toujours nécessaire pour la résolution d'un problème distribué. Étant donnée l'utilisation toujours croissante des systèmes distribués actuels ainsi que le coût induit par les tâches de contrôle et de maintenance dans ses systèmes, l'étude de la nature locale des algorithmes distribués est un enjeu de taille qui peut avoir des conséquences importantes sur le développement et la mise en place de nouveaux systèmes distribués complexes.

Dans ce travail de thèse, nous essayons de contribuer à mieux *comprendre* certains aspects locaux des algorithmes distribués, à mieux les utiliser pour *concevoir des solutions robustes et performantes*, et à pouvoir les *appliquer* de façon pratique. Dans les paragraphes qui vont suivre, nous donnons d'abord un cadre formel et précis des différents modèles de calculs distribués que nous considérons. Ensuite, nous présentons nos différents travaux en donnant nos principaux résultats et contributions.

0.1 Modèles de calculs

En fonction du système distribué considéré et en fonction des objectifs poursuivis, on peut trouver une très grande variété de modèles de calculs. Leur très grand nombre est tel que dans chaque article publié, on trouve une section dédiée à la description du modèle considéré, et accompagnée en général d'une argumentation visant à motiver sa pertinence soit à un niveau théorique, soit à un niveau pratique. En effet, le modèle de calculs que l'on considère a une importance capitale lors de la conception et l'étude d'un algorithme distribué. Ce qui est vrai dans un modèle ne l'ai pas forcément dans un autre. Des détails qui peuvent paraître insignifiants dans un modèle centralisé prennent une ampleur surprenante en distribué et ont des conséquences étonnantes sur la nature des résultats que l'on peut obtenir.

Dans cette thèse nous considérons exclusivement deux types de systèmes distribués. Le premier concerne les réseaux d'interconnexions ou différentes entités sont connectées entre elles et communiquent par échanges de messages. Le deuxième concerne des réseaux à base d'agents mobiles. Dans la suite, nous décrivons ses deux types de systèmes distribués et nous précisons les différentes hypothèses que nous allons effectuer.

0.2 Modèles avec échanges de messages

0.2.1 Modèle de base

Considérons un réseau constitué d'un ensemble de machines, de processeurs, de processus, en général d'un ensemble d'*entités de calculs* pouvant communiquer en échangeant des messages. Nous utilisons alors un graphe $G = (V, E)$, comme celui représenté dans la figure 0.1, pour modéliser la topologie ou la structure du réseau. Les deux ensembles V et E correspondent respectivement aux sommets et aux arêtes du graphe. Nous considérons uniquement des graphes *connexes*, c'est à dire que pour deux sommets quelconque du graphe, on peut trouver un chemin constitué d'arêtes consécutives qui relie les deux sommets, *simples*, c'est à dire sans arêtes multiples reliant une même paire de sommets, et *sans boucles*, c'est à dire qu'aucune arête ne relie un sommet à lui même.

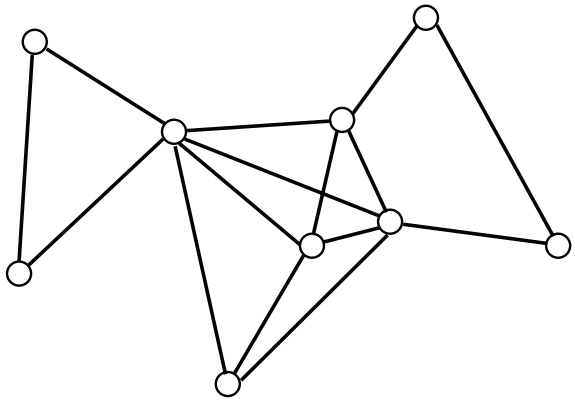


Figure 0.1: Un graphe modélisant un réseau d'interconnexion

Dans ce formalisme, un sommet (ou noeud) du graphe correspond à une entité de calcul et une arête du graphe correspond à un canal de communication bidirectionnel entre deux entités de calculs voisines. Seuls deux sommets reliés par une arête peuvent mutuellement s'envoyer des messages. Le voisinage d'un sommet donné est alors l'ensemble des sommets auxquels ce dernier est connecté par une arête.

Chaque sommet v du graphe G possède des *ports* lui permettant de communiquer avec ses voisins. Ils existent exactement autant de ports attachés au sommet v que d'arêtes le reliant

à ses voisins. Une arête $e = (u, v) \in E$ correspond alors à une paire $((u, i), (v, j))$ comme le montre le dessin de la figure 0.2. L'envoi d'un message de u vers v (resp. de v vers u) s'effectue alors en balançant le message sur le port i (resp. sur le port j). Le sommet v (resp. u) reçoit le message en le récupérant sur le port j (resp. sur le port i). Nous supposons que les ports d'un même sommet ont des numéros distincts et qu'un sommet peut distinguer le port sur lequel il a reçu un message donnée.

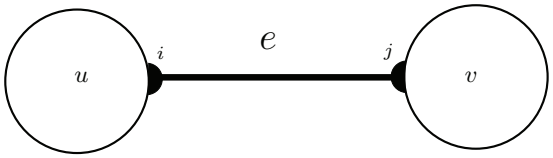


Figure 0.2: Zoom sur une arête $e = (u, v)$

Dans la suite, les hypothèses suivantes sont implicitement admises:

- Chaque sommet est une entité *autonome* de calcul, qui en plus d'envoyer et de recevoir des messages, est capable d'effectuer des calculs locaux. Ces calculs dépendent uniquement de l'état local du sommet et des messages échangés avec le voisinage.
- Les liens de communications, i.e., les arêtes, ainsi que les entités de calculs, i.e., les sommets, sont complètement *fiables*. En d'autres termes, aucune panne ne peut surgir sur une arête ou sur un sommet.
- On ne considère que des graphes statiques, c'est à dire que la structure du graphe reste la même du début à la fin. Aucun sommet, ni arête, ne peut se rajouter ou disparaître en cours d'exécution. En effet, aucun aspect lié à la dynamicité dans les réseaux d'interconnexions n'est traité.
- Tous les sommets du graphe sont actifs simultanément dès le début de l'algorithme. Autrement dit, toutes les entités du systèmes commencent à exécuter des calculs en même temps.

0.2.2 Opérations élémentaires

L'activité d'un sommet, i.e., celle qui correspond à l'exécution d'un algorithme, consiste en trois opérations élémentaires:

1. Envoyer des messages aux voisins;
2. Recevoir des messages;
3. Effectuer des calculs locaux;

L'ordre dans lequel ces trois opérations sont effectuées par deux sommets voisins est crucial pour le bon déroulement de l'algorithme. En effet, une difficulté principale lors de la conception d'un algorithme distribué avec échanges de messages est la bonne coordination des ordres d'envoi et de réception respectifs à deux sommets voisins.

0.2.3 Le synchronisme

Dans notre modèle de base, on met souvent l'accent sur la notion de temps et sur le niveau de synchronisme autorisé lors de la communication entre deux sommets (voir par exemple [Tel00, Pel00]). Nous allons utiliser deux modèles extrêmes que nous décrivons comme suit:

1. ***Le modèle synchrone***: Dans ce modèle, nous supposons que tous les sommets ont accès à une horloge globale qui produit des tops. En plus, si un message est envoyé pendant un top t, alors il arrive à destination avant le top $t + 1$. Cette hypothèse met l'accent sur le fait que les délais de communications entre les différentes entités du système sont bornés et que la valeur de la borne est connue des sommets. Ainsi, si un sommet attend un message qui n'arrive pas au bout d'un top, alors il peut en déduire qu'aucun message n'a été envoyé au top précédent.

2. ***Le modèle asynchrone***: Dans ce modèle, aucune horloge globale n'est autorisée. Les messages mettent un temps fini mais non déterminé (inconnu) pour arriver à destination. Il est impossible dans ce modèle de prédire (de façon locale) si un message est arrivé à destination ou s'il est encore en cours de route.

Ces deux modèles se situent aux deux extrémités de tout un panel de modèles décrivant des types de synchronismes différents. Nous motivons ce choix extrême par le fait que si un résultat d'impossibilité ou une borne inférieure sont démontrés pour un problème dans le modèle synchrone alors, en général, ils sont tout aussi corrects pour le modèle asynchrone ou pour les modèles présentant un niveau intermédiaire de synchronisme. Inversement, si nous disposons d'une solution à un problème dans le modèle asynchrone, alors notre solution reste toujours correcte pour des modèles ayant un niveau de synchronisme plus fort.

0.2.4 Mesures de complexité

Dans les modèles de calculs avec échanges de messages, on utilise le plus souvent deux mesures de complexité pour évaluer l'efficacité d'un algorithme distribué: le nombre de messages échangés et le temps d'exécution de l'algorithme.

Dans le cadre de cette thèse nous définissons la complexité en messages comme étant le nombre total de messages envoyés par les différents sommets. La définition de la complexité en temps pose quant à elle plus de difficultés et nécessite plus de précisions notamment dans le modèle asynchrone.

Le cas synchrone

Considérons d'abord le modèle synchrone, i.e., on suppose donc qu'il existe une horloge globale et qu'un message envoyé pendant un top arrive à destination avant le top suivant. La complexité en temps peut se définir comme étant le nombre total de tops émis par l'horloge globale pour la réalisation d'une tâche. Cependant, cette définition reste incomplète si on ne précise pas d'avantage le nombre de tops nécessaires pour réaliser une opération élémentaire.

On pourrait par exemple supposer que pendant un top d'horloge un sommet ne peut envoyer un message qu'à un nombre bien choisi de voisins, par exemple un seul. On pourrait aussi supposer que la réception d'un message consome un top d'horloge. De même, on peut préciser le nombre de tops nécessaires pour qu'un sommet puisse effectuer tel ou tel calcul local, e.g., trier des variables.

Dans la suite, une unité de temps correspond à un top d'horloge. En plus, nous mettons l'accent sur le *coût de la communication* plus que sur le coût des calculs locaux et nous supposons vraie l'hypothèse classique suivante:

($\mathcal{H}$): *Pendant une unité de temps, un sommet peut à la fois envoyer un message (différent) à chacun de ses voisins, recevoir des messages en provenance de chacun de ses voisins, et effectuer n'importe quel type de calculs locaux.*

Le cas asynchrone

Dans le cas asynchrone, le délai d'acheminement d'un message est fondamentalement inconnu d'où l'hypothèse de non existence d'horloge globale. On ne peut donc spécifier le temps mis par un message pour arriver à destination. Cependant, et pour des raisons d'évaluation purement théorique de la complexité, nous supposons que *le délai induit par un message pour arriver à destination est d'au plus une unité de temps.* En d'autres termes, on normalise tous les délais qui peuvent survenir lors de l'acheminement d'un message, mais en aucun cas, on suppose l'existence d'une borne connue pour l'acheminement des messages. En particulier, ceci permet de respecter la nature asynchrone du modèle considéré.

Maintenant, les mêmes paramètres que pour le modèle synchrone, à savoir le coût des opérations élémentaires, sont à prendre en considération. Dans la suite, nous admettons la validité des propriétés énoncées dans l'hypothèse $\mathcal{H}$ précédente.

0.2.5 Complexité en temps, taille des messages et modèles de calculs sous-jacents

Les éléments donnés dans les paragraphes précédents ainsi que la validité de l'hypothèse $\mathcal{H}$ permettent de définir la complexité en temps comme étant le nombre d'unités de temps écoulés depuis le début de l'algorithme jusqu'à sa fin.

Néanmoins, un autre paramètre important est à prendre en compte: *la taille maximale d'un message*. On peut par exemple se restreindre à des messages de taille B bits, avec B un

paramètre bien déterminé, ou au contraire autoriser n'importe quelle taille de messages (un sommet peut envoyer dans un seul message n'importe quelle quantité d'information). Il est évident qu'un algorithme autorisant une taille quelconque de messages peut être implémenté de façon triviale avec des messages de taille fixée B: en découpant les messages générés par l'algorithme en petits messages, chacun de taille au plus B. En revanche, la nombre d'unités de temps nécessaires pour acheminer ces messages sera plus grand.

Ce dernier paramètre a donné naissance aux deux modèles de calculs distribués suivants:

- ***Le modèle sans congestion***: Ce modèle a été introduit pour la première fois par Nathan Linial sous le nom de *modèle libre de calcul* (the free model of computation) et a ensuite été repris dans le livre de David Peleg sous le nom de *modèle local de calcul* (the $\mathcal{LOCAL}$ model of computation). Dans ce modèle, on suppose en plus de l'hypothèse $\mathcal{H}$, que la *taille des messages est non limitée*. En d'autres termes, en une unité de temps, un sommet peut transmettre toute l'information dont il dispose à tous ses voisins.

- ***Le modèle avec congestion***: connu sous le nom anglais de $\mathcal{CONGEST}$ model. Ce modèle appelé aussi *modèle* $\log(n)$*-borné* autorise seulement des messages de taille $O(\log(n))$, n étant le nombre de sommets dans le graphe. Comme son nom l'indique ce modèle prend en considération les difficultés inhérentes à l'utilisation de *petits messages* lors d'échanges d'information entre deux sommets.

Récapitulatif

Dans un système distribué par échanges de messages, nous utilisons un graphe pour modéliser les interconnexions entre les différentes entités de calculs. Dans le cas synchrone (resp. asynchrone), nous supposons qu'il existe (resp. n'existe pas) une horloge globale commune à tous les sommets. En une unité de temps, un sommet peut envoyer un message à tous ses voisins. Les calculs locaux effectués par un sommet prennent une quantité négligeable de temps. On parle de modèle sans congestion (resp. avec congestion) quand la taille des messages est illimitée (resp. bornée).

0.3 Modèles avec agents mobiles

Dans cette thèse nous nous intéressons aussi à des systèmes distribués où les calculs sont effectués par des entités mobiles appelées agents mobiles. Un agent mobile peut par exemple être un programme qui se déplace de machine en machine en exécutant des instructions sur chaque machine. Il peut aussi représenter un robot qui se déplace de pièce en pièce et qui effectue certaines commandes de contrôle. Les exemples sont nombreux et les modèles formels pour représenter un système distribué avec des agents mobiles le sont également.

Modèle de base

Dans cette thèse, nous considérons qu'un agent mobile est une *entité autonome de calcul* qui peut se *déplacer* d'un *site* à un autre. Afin de modéliser l'ensemble des sites présents dans le système, nous utilisons un graphe. Un sommet du graphe modélise un site pouvant héberger un agent. Une arête du graphe modélise une route que doit emprunter un agent pour aller d'un sommet à un autre. Comme précédemment, nous considérons uniquement le cas de graphes connexes, simples et sans boucles.

Nous admettons les hypothèses suivantes:

- À chaque sommet est affecté autant de *portes* que d'arêtes le reliant à ses voisins. On donne des numéros différents aux portes correspondant à un même sommet. En reprenant le schéma de la figure 0.2, pour se déplacer d'un sommet u à un sommet v voisin, un agent doit traverser l'arête $e = ((u, i), (v, j))$ qui relie les deux sommets u et v, i.e., l'agent quitte u à partir de la porte i et entre dans v par la porte j. Aucune "téléportation" n'est autorisée! En effet, quand un agent quitte un sommet donné, il atterrit forcément dans un sommet voisin.

- Chaque sommet est équipé d'*une ardoise* (appelé aussi *tableau blanc*) à laquelle peut accéder les agents présents dans le sommet. L'accès à l'ardoise d'un sommet s'effectue en lecture/écriture exclusive. Initialement, c'est à dire avant le début de tout calcul, les ardoises peuvent contenir des informations qui peuvent êtres différentes d'un sommet à un autre.

  D'un côté, les informations que contiennent les ardoises renseignent sur l'état des calculs effectués par les différents agents. D'un autre côté, elles permettent aux agents présents sur un même sommet de communiquer et de s'échanger de l'information.

- Tous les agents mobiles sont *fiables*. Aucune déviance du comportement attendu lors de la spécification d'un algorithme n'est autorisée pour les agents. De même, toutes les arêtes et tous les sommets sont fiables. Le contenu des ardoises ne peut être endommagé par des intervenants extérieurs. Un agent en transit sur une arête arrive bien à destination. Le comportement normal ainsi que l'état des agents qui s'exécutent sur un sommet ne peuvent être altérés par aucun intervenant extérieur.

- On ne considère que des graphes statiques dont la structure reste identique du début à la fin.

- Tous les agents sont actifs simultanément dès le début de l'algorithme.

Synchronisme

Dans le cas synchrone, on admet qu'il existe une horloge globale commune à tous les agents. Un agent qui quitte un sommet pendant un top donné t arrive à sa destination avant le top

$t+1$, i.e., un agent met au plus une unité de temps pour traverser une arête. Tous les calculs locaux effectués par un agent prennent un temps négligeable et ne sont pas pris en compte lors de l'évaluation de la complexité d'un algorithme. Seuls les déplacements des agents d'un sommet à un autre ont un coût.

Dans le cas asynchrone, on n'admet l'existence d'aucune horloge globale. Un agent met un temps fini mais non déterminé pour traverser une arête. Lors de l'évaluation théorique de la performance d'un algorithme nous supposons, comme dans le cas des messages, que le temps mis pour traverser une arête est borné par un. Cependant, nous n'aurons pas à traiter du cas d'algorithmes asynchrones avec agents mobiles dans le cadre de cette thèse.

Mémoire d'ardoises et mémoire d'agents

La taille mémoire autorisée pour une ardoise ou celle utilisée pour stocker les variables locales d'un agent peuvent jouer un rôle important lors de la conception d'un algorithme distribué. Les problèmes que nous traitons dans cette thèse ne sont pas affectés de façon significative par ce paramètre. Nous n'aborderons donc pas les difficultés que l'on peut rencontrer et les détails correspondant. Néanmoins, nous supposons que l'espace de la mémoire de stockage pour un agent ou pour une ardoise est *bornée*.

0.4 Contribution et structure de cette thèse

Les différentes particularités des modèles que nous venons de décrire traduisent déjà plusieurs aspects locaux propres aux systèmes distribués. Tant au niveau du synchronisme, qu'au niveau du type de communication entre les différentes entités du systèmes, ou encore au niveau de la nature même des entités de calculs considérées (agents mobiles ou entités fixes), on est confronté au manque d'informations globales et à la nécessité d'une bonne gestion et d'une bonne coordination des différentes entités lors de la conception d'un algorithme distribué. Dans cette thèse nous adoptons quatre approches différentes et à priori indépendantes qui contribuent à mieux comprendre les difficultés inhérentes aux aspects locaux des algorithmes distribués. Ces approches traitent respectivement de problématiques qui interviennent lorsqu'on réfléchit à des notions fondamentales telles que l'*efficacité et les structures de contrôles distribuées*, l'*unification et l'abstraction d'algorithmes distribués*, l'*équivalence entre agents mobiles et échanges de messages* et l'*expérimentation et l'apprentissage des algorithmes distribués*.

Cette thèse comporte donc quatre parties dont chacune correspond à une approche différente. Dans les paragraphes suivants nous exposons les problèmes abordés dans chaque partie et nous donnons un aperçu des principaux résultats obtenus.

Partie I: Une approche dédiée à l'efficacité dans les calculs distribués: Décompositions peu denses de graphes et applications

Dans cette partie, nous nous intéressons à la construction distribuée de structures et de représentations de graphes dites *préservant la localité* [Pel00], tels que les partitions et les sous graphes couvrants. En effet, ces structures permettent de capturer de façon locale des propriétés topologiques des réseaux sous-jacents. Elles sont également à la base de plusieurs applications distribuées tels que les synchroniseurs et le routage, et présentent une utilité récurrente lors de la résolution de nouveaux problèmes distribués. Les propriétés de ces structures sont intimement liées à la complexité des applications qui les utilisent. En général, améliorer la complexité des applications correspondantes revient à trouver de nouvelles structures avec de meilleures propriétés.

Dans ce contexte, nous nous sommes concentrés sur l'aspect *complexité en temps* des algorithmes permettant de construire ces structures locales de graphes dans le modèle avec échanges de messages. D'un côté, améliorer la complexité en temps de ces algorithmes permet en général d'améliorer la complexité des applications correspondantes. D'un autre côté, la complexité en temps est intimement liée à la notion fondamentale de la quantité d'information nécessaire pour résoudre un problème. En effet, le mieux q'un sommet puisse faire en t unités de temps est de récolter une information concernant des voisins à distance t. Donc, si un problème peut être résolu en t unités de temps alors toute l'information nécessaire à la résolution du problème se trouve à une distance au plus t de chaque sommet. Cet aspect est aussi lié à la question récurrente en algorithmique distribuée qui est celle de comment casser la symétrie dans les problèmes distribués de façon *efficace* c'est à dire en utilisant le moins d'information possible.

Chapitre 1: Construction complètement distribuée de décompositions peu dense de graphes

Dans ce premier chapitre, nous nous intéressons aux représentations d'un graphe en utilisant des "clusteurs"; un clusteur étant un ensemble de sommets du graphe. En particulier, une représentation permet de partitionner les noeuds d'un graphe en un ensemble disjoints de clusteurs tel que *(i)* le sous graphe induit par un clusteur est connexe, *(ii)* le rayon d'un clusteur est petit et *(iii)* le nombre moyen de clusteurs voisins à un clusteur fixé est lui aussi petit. Cette partition peut être considérée comme étant une *structure de base* qui a donné lieu à plusieurs autres types de partitions et plus généralement de décompositions [Awe85, AP90b, ABCP98, MS00].

Les algorithmes distribués existants et permettant de construire cette partition de base étaient semi-séquentiels, dans le sens ou les clusteurs sont construits un après l'autre mais jamais simultanément. Dans un premier temps, nous présentons un algorithme *complètement*

distribuée qui permet de construire les clusteurs de façon parallèle. Notre algorithme peut être implémenté simplement dans le modèle de calculs avec échanges de messages et avec congestion ($\mathcal{CONGEST}$ model). Dans un second temps, Nous montrons comment notre construction complètement distribuée peut être utilisée pour obtenir des algorithmes (déterministes et probabilistes) ayant une complexité *sous-linéaire* en temps. Nos algorithmes améliorent les anciens algorithmes qui eux avaient une complexité seulement linéaire.

Une première application importante de nos algorithmes est l'amélioration du temps du pré-calcul nécessaire à la mise en place d'un synchroniseur de type γ. La deuxième application est la construction de "spanners" (ou sous graphes couvrants) optimaux en temps sous-linéaire. Cette dernière application est décrite au début du deuxième chapitre qui lui traite de façon plus approfondie du calcul efficace de spanners.

Chapitre 2: La localité des sous-graphes couvrants

Dans ce chapitre, nous considérons le modèle de calculs avec échanges de messages et sans congestion ($\mathcal{LOCAL}$ model). Comme expliqué précédemment, ce modèle ne fixe aucune limite sur la taille des messages pouvant être échangés. Il permet de mettre de côté les difficultés inhérentes à l'utilisation de petits messages, et de se concentrer plutôt sur la notion fondamentale de la quantité d'information nécessaire pour résoudre un problème, c'est à dire la *localité* d'un problème.

Nous nous intéressons alors à la construction de spanners (ou sous graphes couvrants) ayant à la fois peu d'arêtes et un petit facteur d'étirement. Ce type de structures est largement étudié du fait de son incroyable utilité pour résoudre plusieurs autres problèmes tels que le routage, les oracles, le calculs de plus courts chemins, etc.

Le facteur d'étirement d'un spanner se définit naturellement comme étant le maximum du rapport de la distance entre deux sommets dans le sous graphe couvrant et de leur distance dans le graphe d'origine. Par exemple, si l'on inclut toutes les arêtes du graphe d'origine dans un spanner alors le facteur d'étirement est égale à un. On préserve ainsi toutes les distances, mais on prend trop d'arêtes. En revanche, si on considère un arbre couvrant alors le facteur d'étirement peut être aussi grand que le diamètre du graphe, mais on prend le minimum possible d'arêtes pour connecter tous les sommets. Un bon compromis est à la fois un bon facteur d'étirement et un bon nombre d'arêtes. C'est ce qui nous intéresse très particulièrement dans ce chapitre.

Nous utilisons une nouvelle technique basée sur les ensembles indépendants et maximaux. Cette technique nous permet à la fois de casser la symétrie de façon rapide et complètement distribuée, et de construire des spanners ayant de bonnes propriétés. Les algorithmes déterministes que l'on obtient ont une complexité en temps asymptotiquement meilleure que sous-linéaire, ce qui améliore toutes les anciennes constructions. De plus, les spanners que nous obtenons sont nouveaux de part leurs facteurs d'étirements. Nous décrivons aussi des

implémentations probabilistes de notre technique qui permettent d'obtenir une complexité en temps seulement logarithmique. Plusieurs résultats dans le cas des graphes avec des petits degrés sont aussi donnés dans les cas déterministe et probabiliste.

Partie II: Une approche formelle dans les calculs distribués: Les systèmes de réécritures de graphes

La deuxième partie de cette thèse comporte un seul chapitre dédié à l'étude de quelques aspects formels liés aux algorithmes distribués.

L'un des thèmes les plus importants en algorithmique en générale est l'étude de la correction d'un algorithme. En algorithmique distribuée, cet aspect prend une dimension toute particulière étant donné la multitude de modèles de calculs que l'on peut trouver, et la difficulté inhérente au non-déterminisme dans les calculs distribués.

Dans ce contexte, les systèmes de réétiquetages de graphes [LMS99, GMM04, CM05, CMZ06] constituent un outil formel pour concevoir et prouver de façon unifiée des algorithmes distribués. En effet, dans un système de réétiquetage, on considère un graphe *étiqueté* et un ensemble de *règles de réétiquetage*. Les étiquettes peuvent par exemple encoder les états ou les propriétés d'un sommet ou d'une arête. Les règles de réétiquetages dictent la façon avec laquelle les sommets et les arêtes sont réétiquetés en fonction de leurs anciennes étiquettes. Ces règles sont *locales* dans le sens où elles concernent des boules de rayon fixé (typiquement égal à 1), et ne modifient que les étiquettes attachées aux boules et non leurs structures. Dans ce cadre, on ne se préoccupe pas de la façon avec laquelle ses règles peuvent être implémentées dans des modèles pratiques, on se concentre plutôt sur comment concevoir les bonnes règles et comment prouver leur correction. Les systèmes de réétiquetages constituent donc un modèle abstrait qui permet de réfléchir sur des problèmes distribués fondamentaux.

En utilisant les systèmes de réétiquetages, nous montrons comment on peut encoder de façon très abstraite, mais très compréhensible, deux techniques simples que sont *la diffusion* et *la collecte d'information*. Ces deux techniques apparaissent comme étant la base de plusieurs autres algorithmes distribués plus compliqués. Partant de cette observation, nous montrons alors à travers plusieurs exemples comment ces différentes briques de bases peuvent être combinées pour encoder des algorithmes complexes de façon tout aussi abstraite et compréhensible. Nous commencons par donner les systèmes de réétiquetages correspondants aux techniques de "Propagation d'Infroamtion avec Feedback" (PIF), notamment à travers les exemples de *calculs de fonctions globales* et de *calcul d'arbres recouvrants BFS*. Ensuite, nous étudions l'exemple plus évolué de *calcul d'arbres recouvrants de poid minimum (MST)* en donnant les règles de réétiquetages correspondantes. Ce dernier exemple est particulièrement intéressant dans notre approche, puisqu'il représente une combinaison non triviale des principales techniques de bases utilisées lors de calculs distribués.

L'objectif principal poursuivit par les travaux présentés dans cette partie est de donner

un cadre formel et aussi général que possible pour la conception et la preuve d'algorithmes distribués. Les résultats que nous obtenons ouvrent des perspectives nombreuses dans l'étude des algorithmes distribués à la lumière des systèmes de réétiquetages.

Partie III: Une approche à base d'agents mobiles dans les calculs distribués: Le problème de la "poignée de main"

Dans la troisième partie, nous traitons de quelques aspects locaux propres aux agents. Nous nous intéressons notamment aux bénéfices que nous pouvons tirer de leur utilisation par rapport aux échanges de messages.

Chapitre 4: "Poignée de main" efficace en utilisant des agents mobiles

Dans le quatrième chapitre, nous étudions le problème de "la poignée de main" (*Handshake* [MSZ03, DHSZ06]) dans le modèle distribué avec agents mobiles. Ce problème consiste tout simplement à considérer une infinité de "round" et de calculer à chaque round un ensemble d'arêtes qui ne se recouvrent pas, i.e., un ensemble stable (ou indépendant) d'arêtes. En plus, nous sommes guidés par les deux contraintes suivantes: *(i)* calculer le plus d'arêtes que possible en un round, et *(ii)* la durée d'un round doit être aussi petite que possible (typiquement constante).

Ce problème a été beaucoup étudié dans le modèle avec échanges de message puisqu'il permet à deux sommets d'avoir la garantie de communiquer exclusivement entre eux. Il est aussi à la base de plusieurs algorithmes distribués pour le calcul d'ensemble maximal d'arêtes indépendantes. Il permet également de résoudre de façon distribuée des problèmes liés à la coloration de graphe.

Dans un modèle à base d'agents mobiles, calculer un ensemble indépendant d'arêtes est facile si le système contient un seul agent. Si l'on s'autorise plusieurs agents, il faut préciser la façon avec laquelle les différents agents coordonnent leur actions pour calculer le plus d'arêtes possibles. Nous donnons alors un algorithme pour résoudre le problème avec n'importe quel nombre d'agents et en utilisant des rounds ayant une durée constante. Notre algorithme se base sur une marche aléatoire des différents agents. On donne une analyse probabiliste de notre algorithme dans le régime stationnaire et nous montrons comment créer les agents de façon efficace. Nous calculons notamment le nombre optimal (par rapport à notre technique) d'agents qui permet de maximiser le nombre moyen d'arêtes indépendantes calculées par notre algorithme. Pour des graphes particuliers tels que les graphes à degrés bornés et les graphes quasi d-régulier, notre algorithme est optimal à une petite constante multiplicative près.

Nous montrons comment émuler notre technique dans le modèle avec échanges de messages tout en gardant les mêmes performances de l'algorithme original avec agents. Ceci permet d'améliorer les anciens algorithmes et les applications correspondantes. En particulier, nous

obtenons un algorithme permettant d'implémenter les systèmes de réétiquetages sur des arêtes (Partie II) de façon plus efficace. Notre algorithme permet en effet de réétiqueter en parallèle et de façon indépendante un plus grand nombre d'arêtes. Cette application est donnée au début du Chapitre 5.

Chapitre 5: Implémentation des systèmes de réétiquetages de graphes en utilisant des agents mobiles

La première partie de ce cinquième chapitre est consacré à utiliser et à étendre les techniques de handshake pour mettre en place un cadre pratique pour l'implémentation de quelques systèmes de réétiquetages de bases. Nous nous intéressons en effet aux réétiquetages dans les étoiles fermées et dans les étoiles ouvertes. Dans le cas de plusieurs graphes, on obtient des algorithmes aussi efficaces, en terme de réétiquetages autorisés pendant un round, que dans le cadre d'échanges de messages. En plus, lorsque nous adaptons nos algorithmes avec agents au modèle synchrone avec messages, nous obtenons des solutions ayant une meilleure complexité en nombre de messages. Cela confirme le fait que réfléchir en terme d'agents mobiles peut contribuer à trouver de nouvelles solutions plus efficaces dans le modèle avec messages.

La deuxième partie de ce chapitre est quant à elle destinée à donner un cadre général pour l'implémentation d'un système de réétiquetage *quelconque* en utilisant des agents mobiles. Ici, nous ne sommes plus motivés par des considérations de performances. Nous cherchons simplement à montrer que les agents mobiles peuvent avoir un grand pouvoir d'expression, dans le sens où il peuvent nous aider à modéliser et à résoudre facilement des problèmes complexes et fastidieux à étudier dans un autre modèle tel que celui avec échanges de messages.

Ce chapitre donne pour la première fois une étude complète de l'implémentation des systèmes de réécritures dans un modèle distribué pratique à base d'agents mobiles et ouvre le champ à plusieurs autres travaux et études aussi bien au niveau théorique que pratique.

Partie IV: Une approche expérimentale et pédagogique dans les calculs distribués: la plateforme *ViSiDiA*

Cette dernière partie se distingue des trois autres par son caractère moins théorique et plus appliqué. En effet, nous y présentons une plateforme qui s'appelle *ViSiDiA* pour la visualisation et la simulation d'algorithmes distribués [ViS06]. *ViSiDiA* est un logiciel qui existe depuis déjà quelques années au sein du LaBRI. Avant le début de cette thèse, *ViSiDiA* permettait de simuler les algorithmes distribués dans le modèle asynchrone avec échanges de messages. Nos principales contributions consistent *(i)* à rendre possible la simulation d'algorithmes synchrones avec échanges de messages, *(ii)* à permettre la simulation d'algorithmes synchrones et asynchrones avec agents mobiles, *(iii)* et enfin à mettre en place une version distribuée du logiciel qui s'exécute sur plusieurs machines et qui est destinée à une simulation sur de grands

graphes.

En partant du principe que la vraie compréhension vient de la mise en pratique, la plateforme *ViSiDiA* permet aux étudiants d'implémenter, de tester et d'observer le comportement d'un algorithme distribué de façon pratique et conviviale. Elle leur permet ainsi d'acquérir une meilleure compréhension des algorithmes distribués et de tester leurs connaissances. *ViSiDiA* est aujourd'hui utilisée à l'université de Bordeaux 1 dans le cadre de cours d'algorithmique distribuée.

Pour les chercheurs, *ViSiDiA* est un moyen de mettre en oeuvre de nouvelles idées et de vérifier rapidement leurs validités notamment en effectuant des études expérimentales. Cela peut être très précieux avant de s'engager dans une étude théorique ou pour avoir une intuition sur la performance d'un algorithme.

Part I

An Efficiency Approach in Distributed Computing: Sparse Graph Decompositions and Applications

Chapter 1

Sublinear Fully Distributed Sparse Graph Decomposition

Abstract.

We present new efficient deterministic and randomized distributed algorithms for decomposing a graph with n nodes into a disjoint set of connected clusters with radius at most $k-1$ and having $O(n^{1+1/k})$ intercluster edges.

We show how to implement our algorithms in the distributed $\mathcal{CONGEST}$ model of computation, i.e., limited message size, which improves the time complexity of previous algorithms [MS00, Awe85, Pel00] from $O(n)$ to $O(n^{1-1/k})$.

We apply our algorithms for constructing network synchronizers and low stretch graph spanners with optimal size in sublinear deterministic time in the $\mathcal{CONGEST}$ model.

The challenge here is to resolve the problems araising due to the use of small messages.

Résumé.

Nous présentons des algorithmes déterministes et probabilistes permettant de décomposer de façon distribuée et efficace un graphe de taille n en un ensemble de clusteurs avec un rayon au plus $k-1$ et ayant seulement $O(n^{1+1/k})$ arêtes d'interconnexion.

Nous montrons comment implémenter nos algorithmes en utilisant des messages de petite taille, ce qui permet d'améliorer la complexité en temps des anciens algorithmes [MS00, Awe85, Pel00] de $O(n)$ à $O(n^{1-1/k})$.

Nous appliquons nos algorithmes pour construire des synchroniseurs et des sous graphes couvrants de façon déterministe en temps sous-linéaire.

Le principal défit dans cette étude est de résoudre les différentes difficultés inhérentes à l'utilisation de messages de petite taille.

1.1 Introduction

Due to the constant growth of networks, it becomes necessary to find new techniques to handle related global informations, to maintain and to update these informations in an efficient way. A *Locality-Preserving (LP) network representation* [Pel00] can be considered as an efficient data structure that captures topological properties of the underlying network and helps in designing distributed algorithms for many fundamental problems: synchronization [MS00, SS94, AP90a], Maximal Independent Set (MIS) [AGLP89], routing [AP92], mobile users [AP95], coloring [PS92] and other related applications [GM03, GKP98, KP98, BBCD02, AR93]. In order to provide *efficient* solutions for these problems, it is important to construct LP-representations in a distributed way while maintaining good complexity measures.

The main purpose of this chapter is to give an overview of some LP-representations of special interest and to show how to construct them efficiently in the distributed setting. More precisely, we focus on one important type of LP-representations called *clustered representations*. The main idea of a clustered representation is to decompose the nodes of a graph into many possibly overlapping regions called clusters. This decomposition allows to organize the graph in a particular way that satisfies some desired properties. In general, the clusters satisfy two types of qualitative criteria. The first criterion attempts to measure the *locality level* of the clusters. Some parameters like the *radius* or the *size* of a cluster are usually used to measure the locality level of a clustered representation. The second criterion attempts to measure the *sparsity level*. This criterion gives an idea about how the clusters are connected to each others. For instance, in the case of disjoint clusters, the number of intercluster edges is usually used to express the sparsity level. In the case of overlapping clusters, the average/maximum number of occurrences of a node in the clusters is usually used to express the sparsity (or the overlap) of the clustered representation.

In general, the locality and the sparsity levels of a clustered representation are tightly related and often go in an opposite way. For instance, one can take the whole graph to be one cluster C. In this case, the sparsity level is good (the degree of C is 0), but the locality level is bad (the radius of C is the radius of the whole graph). In opposite, one can take a representation in which each node forms a cluster. In this case, the locality level is good (the radius of each cluster is 0), but the sparsity level is bad (the degree of a cluster may be Δ where Δ is the maximum degree of the graph).

The complexity of many applications (using clustered representations as a communication structure) is also tightly related to the sparsity and locality levels. In fact, a good locality level implies in general a low time complexity, and a low sparsity level implies low message/memory complexity. All the clustered representations one can find always attempt to find a good compromise between the sparsity and the locality levels.

1.1.1 Goals and related works

In this chapter, we focus on an important clustered representation called *Basic Partition* ([Pel00] Chapter 11). Our interest in this *Basic Partition* comes from its good sparsity-locality compromise. In fact, given an n-node graph, the *Basic Partition* provides a set of *disjoint connected* clusters such that the radius of a cluster is at most $k-1$ and the number of intercluster edges is $O(n^{1+\frac{1}{k}})$ where k is a given integer parameter. Our goal is to design time efficient algorithms for constructing a *Basic Partition* of a graph in a distributed model of computation where *nodes can only communicate with their neighbors by exchanging messages of limited size*.

The *Basic Partition* was first used in [Awe85] in order to design efficient network synchronizers. The idea of producing a clustered representation satisfying a good compromise between the locality level and the sparsity level was then studied in [AP90b]. The results of [AP90b] inspired many other applications and generalizations [Cow93, ABCP96, ABCP98]. In particular, Awerbuch *et al* [ABCP96] studied two important types of clustered representations:

1. The first one called *network decomposition* aims at partitioning the network into *disjoint* colored clusters with either *weak* or *strong* small radius and using a small number of colors. For *weak*-network decompositions, a cluster does not necessarily need to be connected and its radius is computed using paths which may shortcut through neighboring clusters. For *strong*-network decompositions, a cluster must be connected and its radius is computed in the network induced by this cluster.

2. The second one called *network covers* constructs a set of *possibly overlapping* clusters with the property that for any node v, there exists a cluster which contains the t-neighborhood of v, i.e., the neighbors at distance at most t from v where t is an integer parameter. The quality of such covers is measured using the strong radius of clusters and the cluster overlap, i.e., the maximum number of clusters a node belongs to.

In addition to design new network decompositions satisfying some desirable properties, many works studied the problem of distributively constructing these representations in an efficient way. For instance, Awerbuch *et al* [ABCP96] gave a deterministic (resp. randomized) distributed algorithm to construct a $(k, t, O(kn^{1/k}))$-neighborhood cover in $O(tk \cdot 2^{c\sqrt{\log n}} + tk^2 \cdot 2^{4\sqrt{\log n}} \cdot n^{1/k})$ (resp. $O(tk^2 \cdot \log^2 n \cdot n^{1/k})$) time for some constant $c > 0$. A (k, t, d)-neighborhood is a set of possibly overlapping clusters such that *(i)* the strong diameter of a cluster is $O(kt)$, *(ii)* each node belongs to at most d clusters, and *(iii)* the t-neighborhood of each node is covered by at least one cluster. Moreover, a remark in [ABCP96] claims that it is possible to translate this neighborhood cover into a strong-network decomposition of comparable parameters by using some techniques from [Cow93, AP90b].

On one hand, the strong radius of the cover constructed in [ABCP96] is $2k-1$ which

is worst (by a factor 2) than the one of the *Basic Partition*. On the other hand, the distributed model considered there does not take into account the congestion created at various bottlenecks in the network (see Section 3.4 of [ABCP96]). In fact, the network model used in [ABCP96] is the Linial's *free* model [Lin87, Lin92] also known as the $\mathcal{LOCAL}$ model (see [Pel00] Chapter 2). The $\mathcal{LOCAL}$ model assumes that nodes can communicate by exchanging messages of *unlimited size*. This assumption focuses on the locality nature of distributed problems, i.e., what can be computed distributively provided that every node knows its whole neighborhood at some distance?

From a practical point of view, since clustered representations are in the basis of many practical applications, it is crucial to design fast algorithms to construct such representations in practical distributed models. From a more theoretical point of view, it is also interesting and challenging to design fast algorithms assuming only some weak distributed assumptions, e.g., see [PR00].

In [MS00], Moran and Snir gave a distributed algorithm that computes a *Basic Partition* in $O(n)$ time in a distributed model where the size of a message is at most $O(\log n)$ bits, i.e., $\mathcal{CONGEST}$ model (see [Pel00] Chapter 2). The algorithm of [MS00] improves the previous constructions of [Awe85, SS94], and allows to obtain more efficient algorithms for designing network synchronizers γ, γ_1 and γ_2. The algorithm of [MS00] is semi-sequential: Each cluster is constructed around some node in a distributed and layered fashion. Nevertheless, the clusters are constructed sequentially. In other words, the clusters are constructed one after the other: at each iteration, a new node is selected and the next cluster is constructed.

Moran and Snir end their paper [MS00] saying:

Q1 [MS00]:	are there *truly parallel algorithms* which construct a Basic Partition in *polylogarithmic or sublinear time complexity* in the $\mathcal{CONGEST}$ model?

1.1.2 Main results

In the following, we answer the [MS00] question. In fact, we give new sparse partition algorithms with $O(n^{1-1/k})$ time complexity, using messages of size at most $O(\log n)$.

More precisely, we give a fully distributed deterministic algorithm DIST_PART with no precomputation step. The idea is to let the clusters grow spontaneously in parallel in different regions of the graph, breaking ties using node identities. We give a detailed implementation of algorithm DIST_PART using small messages and we analyze its efficiency. The time complexity of algorithm DIST_PART is only linear. However, the technique of algorithm DIST_PART is used as a *black box* in order to design a new synchronous deterministic algorithm (SYNC_PART) with sublinear time complexity. The main idea to break the linear time barrier is to privilege the construction of clusters in the dense region of the graph which allows to finish the distributed construction in constant time once the graph becomes sparse. This idea is then

adapted in order to run in an asynchronous setting and we obtain algorithm Fast_Part. Our new asynchronous algorithm is even faster than the synchronous one for many particular graphs.

We also give a randomized distributed algorithm (Elect_Part) which is based on a local election technique (LE_k) in balls of radius k. This k-local election technique is a generalization of the algorithms given in [MSZ02] and can be of an independent interest. For general graphs, our randomized construction is also efficient, but its main strength is to provide improved bounds for many particular graphs. In fact, the analysis of algorithm Elect_Part enables us to express analytically the degree of parallelism of our construction and to compute the expected number of cluster constructed in parallel.

The basic partition can be applied for designing network covers, network synchronizers and also graph spanners. Hence, we obtain new fast algorithms for these two applications.

Outline

In Sections 1.2 and 1.3, we give some definitions and we review the Basic_Part algorithm for constructing the *Basic Partition* in a semi-sequential manner. In Section 1.4, we give a detailed implementation and analysis of the fully distributed algorithm Dist_Part. In Sections 1.5 and 1.6, we describe algorithms Sync_Part, Fast_Part and Elect_Part, and we analyze their time complexity.

The application of the basic partition to network covers and network synchronizers γ, γ_1 and γ_2 is given in Appendix A. In appendix B, we also give a constructive analysis of our algorithms in the case of *Circulant graphs* and we obtain logarithmic time complexity.

The application of our algorithms to the construction of graph spanners is presented as a preliminary result at the beginning of next chapter.

1.2 Model and definitions

We represent a network of n processes by an unweighted undirected connected graph $G = (V, E)$ where V represents the set of processes ($|V| = n$) and E the set of links between them. We consider the distributed model of computation used in [MS00, Awe85] and known as the $\mathcal{CONGEST}$ model. More precisely, we assume that a node can only communicate with its neighbors by sending and receiving messages of size $O(\log(n))$ bits. Each node processes messages received from its neighbors, performs local computations, and sends messages to its neighbors in negligible time. In a synchronous network, all nodes have access to a global clock which generates pulses. A message which have been sent in a given pulse arrives before the next pulse. In a synchronous network, the time complexity of an algorithm is defined as the worst-case number of pulses from the start of the algorithm to its termination. In an asynchronous network, there is no global clock and a message delay is arbitrary but finite. In the latter case, the time complexity is defined as the worst-case number of time units from

the start of the algorithm to its termination, assuming that a message delay is at most one time unit (this assumption is introduced only for the purpose of performance evaluation).

A cluster C is a subset of V such that the subgraph induced by C is connected. A cluster is always considered with a *leader* node and a BFS spanning tree rooted at the leader. We also assume that each node v of a graph G has a unique identity Id_v (of $O(\log(n))$ bits). The identity Id_C of a cluster C is defined as the identity of its leader.

For every pair of nodes u and v of a graph G, $d_G(u,v)$ denotes the distance between u and v in G (we also write $d(u,v)$ when G is clear from the context). For any node v of a graph G, $\mathcal{N}(v) = \{u \in V \mid d_G(u,v) \leqslant 1\}$ denotes the neighborhood of v. For any cluster C of a graph G, $\Gamma(C) = \bigcup_{v \in C} \mathcal{N}(v)$ denotes the neighborhood of C. For any cluster C of a graph G, $Rad(C)$ denotes the radius of the cluster C, i.e., the radius of the subgraph induced by C in G. Similarly, for any set $\mathcal{C}$ of clusters, $Rad(\mathcal{C}) = \max_{C \in \mathcal{C}} Rad(C)$ denotes the radius of $\mathcal{C}$.

In all our algorithms, clusters are constructed in a layered and concurrent fashion. In other words, a cluster may grow and explore a new layer but it may also lose its last layer. Some clusters may disappear because they lost all their layers and some others may be newly formed. A cluster is said *finished* if it belongs to the final decomposition that we are constructing. A node belonging to a finished cluster is also said finished. A node is said *active* if it is does not belong to a finished cluster.

1.3 A basic algorithm for constructing a sparse partition

```
1: Set 𝒞 := ∅
2: while V ≠ ∅ do
3:    Select an arbitrary vertex v ∈ V
4:    Set C := {v}
5:    while |Γ(C)| > n^{1/k}|C| do
6:       C := Γ(C)
7:    end while
8:    Set 𝒞 := 𝒞 ∪ C and V := V − C
9: end while
10: return 𝒞
```

Figure 1.1: Algorithm BASIC_PART

Let $k \geqslant 1$ be an integer parameter. Typically, k is taken to be small compared with n ($k \leqslant \log n$). Let us consider algorithm BASIC_PART (Fig. 1.1) as given in Peleg's book [Pel00] (Chapter 11, page 130). Algorithm BASIC_PART was first used in [Awe85] as a data structure for synchronizer γ, then some improvements were given in [SS94, MS00]. The algorithm operates in many phases. At each phase, a node is selected from the set of nodes which are

not yet covered by a cluster. Then a new cluster is constructed in many iterations according to the sparsity condition of line 5, i.e., $|\Gamma(C)| > n^{1/k}|C|$. It is important to note that the graph G changes in line 8 of the algorithm and the notations in the while loop correspond to the new graph G obtained after deletion of the corresponding nodes.

Algorithm Basic_Part constructs a *Basic Partition*. In fact, we have the following:

Theorem 1.3.1 ([Pel00]) *The output $\mathcal{C}$ of algorithm* Basic_Part *is a partition of G which satisfies the following properties:*

1. *$Rad(\mathcal{C}) \leqslant k-1$ (locality level)*
2. *There are at most $n^{1+1/k}$ intercluster edges (sparsity level)*

Proof On one hand, once the construction of a cluster C is finished, the nodes of C are definitely removed from the graph G. Thus, the clusters constructed by the algorithm are disjoint. On the other hand, the algorithm terminates once no node remains uncovered. Thus, the final output $\mathcal{C}$ is a partition of G.

Using the sparsity condition, if a cluster C adds i layers, then the size of C verifies $|C| > n^{i/k}$. Hence, a cluster can not add more than $k-1$ layers and the first property of the partition holds.

Let $G_{\mathcal{C}}$ be the graph induced by the clusters of the partition $\mathcal{C}$: the nodes of $G_{\mathcal{C}}$ are the clusters of $\mathcal{C}$ and there is an intercluster edge between two clusters if the clusters are at distance 1 from each others. Now, consider a cluster $C \in \mathcal{C}$. Once the construction of C is finished, there are at most $n^{1/k}|C|$ nodes in G at distance 1. Thus, there will be at most $n^{1/k}|C|$ neighboring clusters that will be constructed after C. Thus, there are at most $n^{1/k}|C|$ intercluster edges that can be added to the graph $G_{\mathcal{C}}$ after the construction of C is finished. Thus, the number of intercluster edges is bounded by $\sum_{C\in\mathcal{C}} n^{1/k}|C|$. Since $\mathcal{C}$ is a partition, $\sum_{C\in\mathcal{C}} |C| = n$ and the second property of the partition holds. ∎

There are many distributed implementations of the Basic_Part algorithm. All of these implementations are semi-sequential. First, they distributively elect a new leader in the network which corresponds to the center of a new cluster. Then, the cluster is constructed in a distributed way by adding the layers in many iterations. The construction of the cluster ends when there are no new layers to add or when the sparsity condition is no longer satisfied. Once the construction of the cluster is finished, a new leader is elected from unprocessed nodes and a new cluster grows up around this leader.

The main difficulty in these algorithms is to *distributively elect* the next leader. In [MS00], a preprocessing is used to overcome this difficulty. First, a spanning tree T of the graph G is constructed. Then, the next leader is elected by achieving a DFS traversal of T. This technique allows to improve the complexity bounds of the decomposition: $O(|E|)$ messages and $O(|V|)$ time.

In the next sections, we introduce a new algorithm with no precomputation step and no next leader election step.

1.4 A deterministic fully distributed basic partition algorithm

1.4.1 Overview of the algorithm

The main idea of algorithm DIST_PART is to allow clusters to grow in parallel in different regions. In fact, consider two nodes u and v such that $d_G(u, v) \geqslant 2k$ with k the same parameter than in algorithm BASIC_PART. Then, it is possible to grow two clusters respectively around u and v without any interference. Based on this observation, we initially let each node of the graph be a single-node cluster. Then, we allow the clusters to grow spontaneously. The main difficulty here is to guarantee that the clusters do not share any nodes.

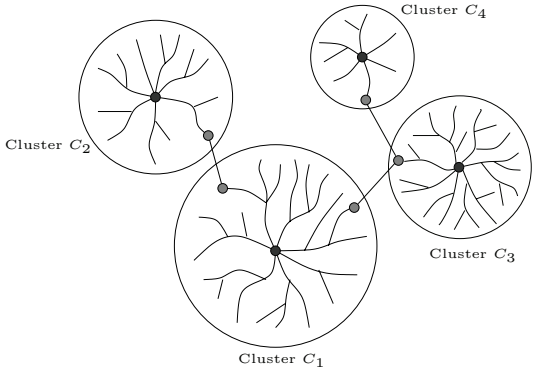


Figure 1.2: An example of conflicts between clusters at distance 1 or 2

We do not avoid cluster collisions but we try to manage the conflicts that can occur. For instance, consider some region of the graph and suppose that some clusters have independently grown as shown in Fig. 1.2. The clusters can not add a new layer simultaneously without overlapping. Thus, we make each cluster compete against its neighbors in order to win a new layer. There are two critical situations. Either, a cluster enters in conflict with an adjacent one or with another cluster at distance two. For instance, in the example of Fig. 1.2, cluster C_1 tries to invade some nodes that belong to cluster C_3 and C_2 at distance 1. Thus, the neighboring cluster C_1, C_2 and C_3 are in conflicts. Similarly, cluster C_4 tries to invade some nodes in cluster C_3. Nevertheless, these nodes are also required for the new layer of cluster C_1. Thus, the two clusters C_1 and C_4 (at distance 2) are also in conflicts. To resume, each cluster must compete against all clusters at distance 1 or 2 in order to add a layer. In addition, a layer not satisfying the sparsity condition of algorithm BASIC_PART must be rejected.

In order to manage the conflicts and the cluster growth, we use the following rules:

```
1: continue := True
2: while continue do
3:    execute the Exploration Rule
4:    if success of the Exploration Rule then
5:       add the new layer
6:       execute the Growth Rule
7:       if Non success of the Growth Rule then
8:          reject the last explored layer
9:          switch to a finished cluster
10:         continue := False
11:      end if
12:   else
13:      execute the Battle Rule
14:   end if
15: end while
```

Figure 1.3: Algorithm Dist_Part: code for a cluster

1. *Exploration Rule*: a cluster is able to add a new layer if its *identifier* is bigger than those of not finished neighboring clusters at distance one or two. If a cluster wins in exploring a new layer then it must apply the *Growth Rule*, otherwise it must apply the *Battle Rule*.

2. *Growth Rule*: If the sparsity condition is satisfied then a cluster adds the last explored layer and tries to apply the *Exploration Rule* once again. Otherwise, the cluster construction is finished and the cluster rejects the last explored layer. The nodes in the rejected layer are re-initialized to single node clusters with their initial identifiers.

3. *Battle Rule*: a cluster looses its *whole* last layer if at least a neighboring cluster at distance one has successfully applied the *Exploration Rule*. The nodes lost by a cluster are re-initialized to single node clusters with their initial identifiers.

Based on the three previous rules, we obtain the fully distributed algorithm Dist_Part described in a high level way in Fig. 1.3.

Remark 1.4.1 *It is important to choose a unique identifier for each cluster. For instance, the identifier of a cluster can be chosen to be the identity of its leader. This is implicitly assumed in the rest of this section. However, we can also choose the couple* $(|C|, Id_v)$ *as the identifier of a cluster* C *with a root* v*, and the lexicographical order to compare cluster identifiers.*

Example: Let us consider the concrete example of Fig. 1.4. We have five clusters 1, 2, 3, 4

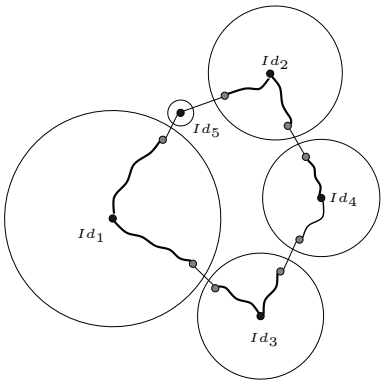
and 5 with identities $Id_1 > Id_2 > Id_3 > Id_4 > Id_5$. Assume that the identifier of each clusters corresponds to the identity of its leader node. When a new exploration begins, cluster 1 wins against clusters 5 and 3. Cluster 2 wins against clusters 4 and 5 but looses against cluster 1 which is at distance two. Thus, cluster 2 can not add a new layer. Cluster 4 loses against both clusters 2 and 3 but it will not be invaded because both clusters 2 and 3 can not grow. Cluster 3 wins against cluster 4 but loses against cluster 1. Cluster 3 will be invaded by cluster 1 which wins against all clusters at distance two (cluster 5, 2 and 3). Thus, cluster 3 will lose its last layer. The node connecting it with cluster 4 becomes a single node cluster with the its initial identity Id_6. The node connecting cluster 3 with cluster 1 becomes a leaf in cluster 1. Now, suppose that the sparsity condition for the new enlarged cluster 1 is not satisfied. Then, cluster 1 rejects the last explored layer and its construction is finished. Hence, the nodes in the rejected layer become single node clusters. Then, the remaining active clusters spontaneously continue new explorations. In our example, both cluster 2 and cluster 3 will succeed their explorations and add a layers. Note that in the other regions of the graph, there are other clusters which are fighting against each others. Hence, many clusters can grow in parallel.

1.4.2 Detailed description and implementation

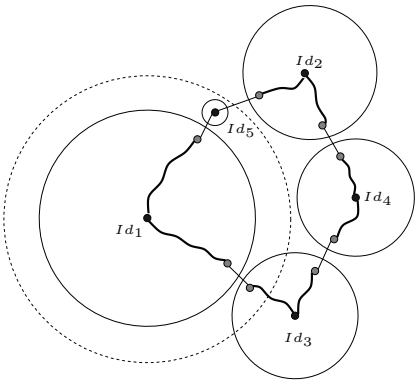
In this section, we give a complete description of how to implement the three rules of the DIST_PART algorithm using message passing. For the clarity of our algorithm, we assume that the identifier of a cluster is the identity of its root. The main difficulty when implementing the three rules of algorithm DIST_PART is that the center of a cluster can not know what is happening on the borders of its cluster, that's why, it must always wait for informations from the leaves (nodes at the border) before taking any decision. Similarly, the leaves can not know the global state of their cluster, that's why they must also wait for information from their root.

The nodes in a cluster collaborates in order to apply the three rules. They can be in five states *root*, *leaf*, *relay*, *orphan* or *final*. At the beginning, all nodes are orphans and they form *orphan clusters*, i.e., cluster with only one node. If a node is in a *final* state, then it belongs to a finished cluster and thus it does not make any computation. The other states define a precise role for each node in its cluster. Generally speaking, if a node v is in a *root* state, then it takes decisions. If v is in a *leaf* state, then it tries to invade new nodes and it informs its root. If v is in a *relay* state, then it forwards information from the leaves to the root.

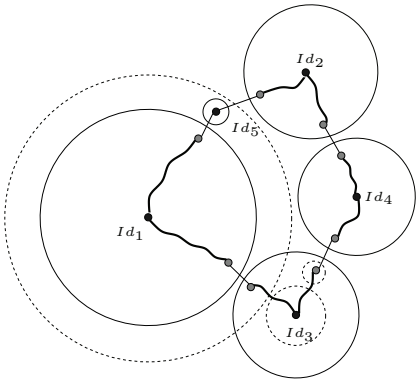
As long as new layers are added (resp. removed) to (resp. from) a cluster, the nodes in the cluster maintain a layered BFS spanning tree. The root of the tree corresponds to the root of the cluster, the leaves of the tree correspond to the leaves of the cluster and the nodes in the interior of the tree correspond to relay nodes. The decisions of adding or removing a layer are broadcasted by the root node according to the informations forwarded by the leaf nodes



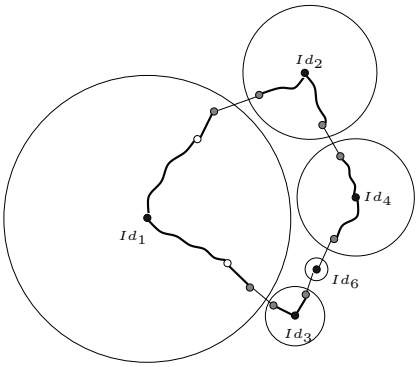
(a) starting configuration.



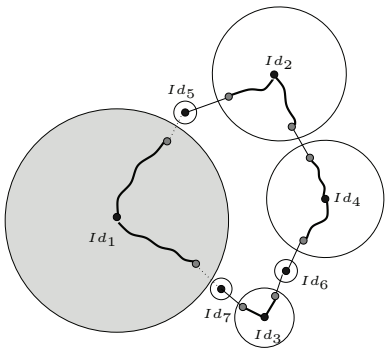
(b) Cluster 1: *Exploration rule* success.



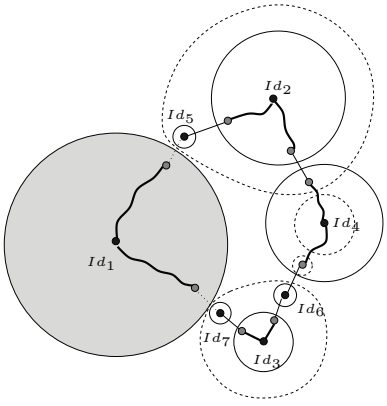
(c) Cluster 3: *Battle rule.*



(d) Cluster 1: *Growth rule*, Cluster 3: *Battle rule.*



(e) Cluster 1: *Growth rule.*



(f) Clusters 3,2: *Exploration rule.* Cluster 4: *Battle rule.*

Figure 1.4: An example of algorithm Dist_Part

all along the constructed BFS tree. Each time new nodes join a cluster, the BFS spanning tree is enlarged by making the new nodes choosing a father among the leaves of the already constructed tree.

In next paragraphs, we give a detailed description of the actions to be performed by each node according to its state. Notice that the state of a node can change several times. For instance, the state of a node can be relay at some time, then become leaf, after that orphan, and at last final.

Remark 1.4.2 *In the pseudo-code of our algorithms, a node uses the function* Send *to send a message to a (or some) neighbors. The function* Receive *allows a node to receive a message from a (or some) neighbors. The receive function is blocking, that is, a node can not execute the next instruction in the algorithm unless the receive action is terminated, i.e., all messages are arrived.*

Root nodes

The algorithm executed by a root node is given by Algorithm 1. First, the root verifies if the sparsity condition is satisfied and it informs the leaves (*Growth rule*). More specifically, if the sparsity condition is satisfied, then the root broadcasts a notification message *NEW* to the leaves in order to begin a new exploration. Otherwise, it broadcasts a *REJECT* message saying that the construction is finished.

After broadcasting a *NEW* message Second, the root waits for the response of its leaves. There are three possible cases:

1. If the root receives only *STOPPED* messages from its leaves, then the leaves do not find new nodes to explore. In this case, the root broadcasts a *STOP* message informing all the nodes that the cluster construction is finished.

2. If the root does not receive any *LOST* message, i.e., only *WIN* (or *STOPPED*) messages, then the new exploration was globally successful. Thus, the root broadcasts a *SUCCESS* message to the leaves. Then, the root waits to learn the size of the new enlarged cluster.

3. If the root receives at least one *LOST* message, then the new exploration was not successful (at least one leaf has lost against a neighboring cluster). Thus, the root informs the leaves by broadcasting a *FAILURE* message. Then, the root waits for the leaf responses. There are two cases:

   - At least one leaf is invaded by another cluster. Thus, the root receives at least a *BYE* message. In this case, the cluster must reject its last layer (*Battle rule*). Hence, it broadcasts a *DOWN* message asking the leaves to become *orphans*.

- All leaves have resisted to neighbor's attacks. Thus, the root receives only *SAFE* messages, i.e., none of the neighboring cluster has succeeded to invade the current cluster. In this case, the root broadcasts an *OK* message saying that the cluster is not invaded and asking for a new exploration.

```
1  Receive count From Sons; Compute |Γ(C)|;
2  if Γ(C) > n^{1/k}|C| then
3  |  Send NEW To Sons ;
4  |  Receive LOST_WIN_STOPPED From Sons;
5  |  if there exists at least one LOST message then
6  |  |  exploration_success := false ;
7  |  else if all messages are STOPPED then
8  |  |  cluster_stopped := true;
9  |  else
10 |  |  exploration_success := true ;
11 |  if cluster_stopped then
12 |  |  Send STOP To Sons ; State := Final ;
13 |  else
14 |  |  if exploration_success then
15 |  |  |  Send SUCCESS To Sons ;
16 |  |  |  h := h + 1;          /* the radius of the cluster */
17 |  |  else
18 |  |  |  Send FAILURE To Sons ;
19 |  |  |  Receive BYE_SAFE From Sons ;
20 |  |  |  if there exists at least one BYE message then
21 |  |  |  |  Send DOWN To Sons ;
22 |  |  |  |  h := h − 1 ;
23 |  |  |  |  if h = 1 then State := Orphan ;
24 |  |  |  else
25 |  |  |  |  Send OK To Sons ;
26 else
27 |  h := h − 1;
28 |  State := Final ;
29 |  Send REJECT To Sons ;
```

Algorithm 1 : Dist_Part: high level code for the *root node* of a cluster C

Leaf nodes

The algorithm executed by a leaf is given by Algorithm. 2.

Remark 1.4.3 *We remark that a leaf does not always belong to the last layer of a cluster. For instance, a leaf node may have only final neighbors belonging to finished clusters. Hence, it*

can not add new nodes to its cluster. Nevertheless, other leaves belonging to the same cluster can continue exploring new nodes. Therefore, the construction of the cluster can continue even if some leaves can not locally explore new nodes. In order to handle this situation, we make each node own a local variable h which corresponds to its width in the BFS-spanning tree of its cluster. If $h = 1$ then the leaf belongs to the last layer and can compete in order to add new layers, otherwise the leaf can not explore any new layer.

When a node u becomes a leaf in a new cluster, it does not know if the sparsity condition is verified for the new cluster (the exploration of a new layer is done before verifying the sparsity condition). Thus, node u sends 1 to its parent v in the new cluster and it waits for the decision of the root. The parent node v sends back the number of its children and so on. At the end, the root can compute the sparsity condition.

If the leaf u receives a *REJECT* message from its parent, then u must leave its new cluster (*Growth rule*) and it becomes an orphan cluster. Otherwise, u receives a *NEW* message from its parent. If u can not explore new regions, then u sends back a *STOPPED* message. Otherwise, u begins a new exploration using an election technique in a ball of radius two: First, u sends its cluster identifier to its neighbors. Symmetrically, it waits for the identifiers of the neighboring clusters. Second, u computes the maximum of the neighbor identifiers (including the identifier of its own cluster) and sends it again to the neighbors. Symmetrically, it waits for the maximum identifiers sent by neighboring leaves. If all the identifiers received by u are equal to the identifier of u's cluster, then u has locally succeed its exploration and it sends back a *WIN* message. Otherwise, u sends back a *LOST* message.

Remark 1.4.4 *Since the clusters have unique identifiers, then two neighboring leaves can easily decide whether they belong to the same cluster, e.g., when exchanging their identifiers in a new exploration.*

Once the exploration is finished, the leaf node u waits for the decision of its root. There are three cases :

1. If u receives a *STOP* message from the root, then none of the leaves can explore new nodes. Thus, u becomes a *final* node, i.e., the construction of the cluster is finished.

2. If u receives an *SUCCESS* message from the root, then all leaves have succeeded their local explorations, i.e., they won all neighbors at distance 1 or 2. Thus, u sends a *JOIN* message to neighboring leaves asking them to join its cluster. Then, u switches to a relay state.

3. If u receives a *FAILURE* message from the root, then at least one leaf has not succeeded the exploration. Thus, u sends a *STAY* message to neighboring leaves informing them that they will not be invaded by u's cluster. Then, u waits to know if the neighboring clusters succeeded their explorations. There are two cases:

- If u receives at least a *JOIN* message from a neighboring leaf (in a different cluster), then it sends back a *BYE* message to its roots, waits for an acknowledgment (*DOWN* message) and it joins the new cluster.
- Otherwise, if none of the neighboring cluster has succeeded in invading the leaf (*STAY* message), the leaf sends back to its root a *SAFE* message. At this stage of the algorithm, the leaves (except those who have received a *JOIN* message) do not know whether their cluster is being invaded or not (only the root globally knows what is happening at its frontiers). Thus, the leaves wait for either an *OK* or a *DOWN* message from the root. If a leaf receives an *OK* message, then it is still in the same cluster and it begins a new exploration once again. Otherwise, it receives a *DOWN* message and it becomes an orphan node.

Orphan nodes

The algorithm executed by an orphan node is given by Algorithm 3. An orphan node acts like a root and like a leaf node. In fact, it takes decisions for its single node cluster and it fights against neighboring nodes. If an orphan node succeeds an exploration, it becomes a root node in a new cluster of radius 1. If it is invaded by a cluster, it becomes a leaf. Otherwise, it re-tries to invade its neighbors (new exploration). If it has only neighbors belonging to finished clusters, then it switches to a final state. Here, the only difficulty is to send the good type of message to the neighbors.

Relay nodes

The algorithm executed by a relay node is given by Algorithm 4. The main role of a relay node is to forward informations from the root to the leaves. If a relay node receives a message from its father, it simply forwards it to its children. If the message is a *REJECT* or a *STOP* message, then the node knows that the cluster construction is finished and it switches to a final state. If the message is a *SUCCESS* message, then the node knows that there is a new layer joining. Thus, the width of the node is incremented by one ($h := h + 1$). If the message is a *DOWN* message then the relay node knows that their cluster was invaded and it has lost the last layer ($h := h - 1$). In this case, if a relay node belongs to the layer before the last one ($h = 2$) then the relay node becomes a leaf.

On the other hand, if a relay node receives a message from its children, it can deduce which step the leaves are executing (exploration of a new layer: *WIN*, *LOST* or *STOPPED* messages, resistance against neighbors attacks: *OK* or *BYE* messages, and computation of the sparsity condition: integer message). In all cases, the relay node can easily compute which type of message it must forward to its root.

Remark 1.4.5 *Each node can easily know which of its neighbors belong to a finished cluster.*

```
1  Send 1 To father ; Receive msg From father;        /* msg is either NEW or REJECT */
2  if msg = NEW then
3      if there are new nodes to explore then
4          Send Id_C To neighbors in other active clusters;
5          Receive ∪_{C'} Id_{C'} From neighbors in other active clusters ;
6          max_1 := the maximum of ∪_{C'} Id_{C'} ;
7          if max_1 < Id_C then
8              Send Id_C To neighbors in other active clusters;
9          else
10             Send max_1 To neighbors in other active clusters;
11         Receive Ids From neighbors in other active clusters ;
12         max_2 := the maximum of received Ids;
13         if max_2 > Id_C then
14             Send LOST To father ;
15         else
16             Send WIN To father ;
17     else
18         Send STOPPED To father ;
19     Receive msg From father;        /* msg is either SUCCESS or STOP or FAILURE */
20     if msg = SUCCESS then
21         Send JOIN To neighbors in other active clusters ;
22         Receive messages From neighbors in other active clusters ;
23         Mark the new Sons in the BFS tree of C; h := h + 1; State := Relay;
24     else if msg = STOP then
25         State := Final ;
26     else
27         Send STAY To neighbors in other clusters ;
28         Receive messages From neighbors in other active clusters ;
29         if there exists at least one JOIN message then
30             Send BYE To father ;
31             Receive msg From father;        /* the message must be a DOWN message*/
32             choose a new father in the new winner cluster; h := 1 ;
33         else
34             Send SAFE To father; Receive msg From father ;
35             if msg = DOWN then
36                 if h = 1 then State := Orphan ;
37                 if h ≠ 1 then h := h − 1 ;
38 else
39     if h = 1 then State = Orphan ;
40     if h ≠ 1 then h := h − 1; State = Final ;
```

Algorithm 2 : DIST_PART: high level code for a *leaf node* in a cluster C

```
1  if there exist new nodes to explore then
2      Send Id_C To neighbors in other active clusters;
3      Receive ∪_{C'} Id_{C'} From neighbors in other active clusters;
4      max_1 := the maximum of ∪_{C'} Id_{C'} ;
5      if max_1 < Id_C then
6          Send Id_C To neighbors in other active clusters;
7      else
8          Send max_1 To neighbors in other active clusters;
9      Receive Ids From neighbors in other active clusters ;
10     max_2 := the maximum of received Ids;
11     if max_2 > Id_C then
12         Send STAY To neighbors in other active clusters ;
13         Receive msg From neighbors in other active clusters ;
14         if there exists at least one JOIN message then
15             choose a father in the new winner cluster ;
16             State := Leaf ; h := 1 ;
17     else
18         h := h + 1 ;
19         Send JOIN To neighbors in other active clusters ;
20         Receive msg From neighbors in other active clusters ;
21         Mark the new Sons in the BFS spanning tree of C;
22         State := Root ;
23 else
24     State := Final ;
```

Algorithm 3 : Dist_Part: code for the *orphan node* of an orphan cluster C

```
1  Receive MSG From any neighbor;
2  if there exists a message MSG from the father in the BFS tree of C then
3      Send MSG To Sons ;
4      if MSG = SUCCESS then
5          h := h + 1 ;
6      else if MSG = DOWN then
7          h := h − 1 ;
8          if h = 1 then State := leaf ;
9      else if MSG = REJECT then
10         h := h − 1; State := final ;
11     else
12         if MSG = STOP then State := final ;
13 if there exists a message MSG from Sons in the BFS tree of C then
14     if MSG = BYE or msg = OK then
15         Receive msg From all Sons ;
16         if there exists at least one BYE message then
17             Send BYE To father ;
18         else
19             Send SAFE To father ;
20     else if MSG = WIN or msg = LOST or msg = STOPPED then
21         Receive msg From all Sons ;
22         if there exists at least one LOST message then
23             Send LOST To father ;
24         else if there are only STOPPED messages then
25             Send STOPPED To father ;
26         else
27             Send WIN To father ;
28     else
29         Receive msg From All Sons;      /* msg is an integer message */
30         count := ∑ msg;
31         Send count To father ;
```

Algorithm 4 : DIST_PART: high level code of a *relay node* in a cluster C

It is sufficient to make each node (which becomes final) send a message to its neighbors to inform them. However, we can avoid these extra communication messages as following. When a node v is explored by a cluster C, it uses the Ids sent by neighbors in order to compute a set $\mathcal{F}_C$ of neighbors belonging to the layer before the last one. Then, if v receives a REJECT *message from the root of C, i.e., the sparsity condition for the last layer of C is not satisfied, then v marks its neighbors in $\mathcal{F}_C$ as finished.*

Now, consider two nodes u and v. Then, w.l.o.g., u must explore v before switching to a final state. Thus, v can always decide whether u is in a final state or not.

Remark 1.4.6 *Although our algorithm is completely asynchronous, we remark that there is a kind of synchronization in our implementation which is close to the one used in synchronizer γ (see Appendix A). In fact, the root nodes control the execution of the algorithm and give the starting signal for all the actions of the leaves. The neighboring leaves belonging to different clusters are synchronized since they wait to receive some information from each others. On one hand, the nodes inside a cluster synchronize their actions as if they were a super node. On the other hand, the super nodes synchronize their decisions too.*

1.4.3 Analysis of the algorithm

Theorem 1.4.7 *Algorithm* Dist_Part *terminates.*

Proof From the algorithm description, the cluster having the biggest identifier in the graph always succeeds the *Exploration rule*. Thus, it always succeeds adding new layers until the sparsity condition is violated. Thus, after at most $k-1$ layers, the nodes in the biggest cluster become in final states. Now, the remaining cluster with the biggest identifier always succeeds its new explorations and so on until all the nodes become in final states. ∎

Theorem 1.4.8 *Algorithm* Dist_Part *emulates the* Basic_Part *algorithm.*

Proof From the algorithm description, once the construction of a cluster is finished, the cluster can not be invaded by any other active cluster. Hence, the constructed clusters are disjoint.

In addition, a new layer is added if and only if it verifies the sparsity condition (*Growth rule*). Symmetrically, if a cluster is invaded, then it loses its whole last layer. Hence, the new cluster still satisfies the sparsity condition.

Thus, the constructed partition satisfies the sparsity and locality properties of algorithm Basic_Part. ∎

Theorem 1.4.9 *In the worst case, the time complexity of algorithm* Dist_Part *is:*

$$Time(\text{Dist\_Part}) = O(n)$$

Proof In the following proof, we consider the clusters in an increasing order of the time of their construction. Let C be a cluster in the final partition $\mathcal{C}$. Let r be the radius of C. Consider the first time t when the cluster C succeeds successively all its explorations until its construction is finished. Let $Time(C)$ be the number of time units from t to the end of C's construction. Let j be the radius of C at time t. For any $i \in \{j, \cdots, r\}$, we consider the time when C contains i layers, and we denote by r_{max_i} the maximum radius of the neighboring clusters of C.

In order to decide if a layer is added or not, the cluster C must be traversed at most a constant number of times. In addition, before a node joins a new cluster, it informs its previous root and waits for the acknowledgment of this root. Thus, $Time(C) \leqslant \sum_{0<i\leqslant r} O(i + r_{max_i})$. Using Theorem 1.3.1, we have $r \leqslant k-1$ and $r_{max_i} \leqslant k-1$. Thus, $Time(C) = O(kr)$.

In the worst case, two clusters are never constructed in parallel. Thus,

$$Time(\text{DIST\_PART}) = \sum_{C\in\mathcal{C}} Time(C)$$

Hence, using the fact that $\sum_{C\in\mathcal{C}} r \leqslant n$, we get

$$Time(\text{DIST\_PART}) = O(k \cdot n)$$

The previous analysis is not sufficient to prove the theorem if the parameter k is not a constant. Nevertheless, it gives us a precious remark. In fact, we remark that it is more interesting to take the couple $(Rad(C), Id_v)$ to be the identifier of a cluster C rooted at a node v and the lexicographical order to compare cluster identifiers (which do not change the overall implementation). In this case, we have $r_{max_i} \leqslant r$. Thus, for the relevant range of $k \leqslant log(n)$, we have:

$$|C| \geqslant n^{r/k} \;\Rightarrow\; r \leqslant \frac{k}{\log(n)} \log(|C|) \;\Rightarrow\; r \leqslant \log(|C|)$$

Thus,

$$\begin{aligned} Time(\text{DIST\_PART}) &= \sum_{C\in\mathcal{C}} \sum_{0<i\leqslant r} O(i + r_{max_i}) \leqslant \sum_{C\in\mathcal{C}} O(r^2) \\ &\leqslant \sum_{C\in\mathcal{C}} O(\log(|C|)^2) \\ &\leqslant \sum_{C\in\mathcal{C}} O(|C|) \end{aligned}$$

Since $\mathcal{C}$ is a partition, the theorem holds. ∎

Corollary 1.4.10 *In the worst case, the message complexity of algorithm* DIST_PART *is :*

$$Message(\text{DIST\_PART}) = O(n \cdot |E|)$$

Proof At each time unit, the nodes exchange at most $O(|E|)$ messages. Using Theorem 1.4.9, the algorithm terminates after at most $O(n)$ time units and the theorem holds. ∎

Remark 1.4.11 *The bound given by the previous corollary does not take into account the fact that a finished cluster stops communicating with its neighbors. In this work, we are mainly interested in the time complexity and we do not care too much about the message complexity.*

Remark 1.4.12 *One shall remark that our theoretical analysis is still sequential. We think that the complexity of our algorithm is better in practice. It would be interesting to compute the average message and time complexity. This is a hard task because the execution depends both on the graph topology and on the distribution of node identities. In Fig. 1.5, we give an example of a graph with a particular node distribution for which the previous complexity bounds are tight up to a constant factor. Nevertheless, one can show that in average (over all the permutations of node identities), the time complexity is logarithmic.*

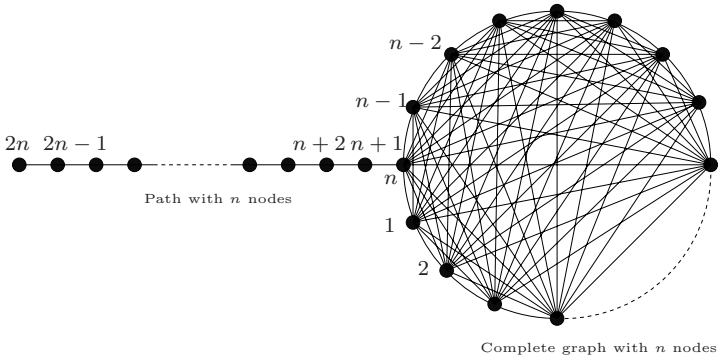


Figure 1.5: An example of bad node distribution

1.5 Sublinear deterministic distributed partition

In the following, we show how to improve algorithm Dist_Part in order to obtain sublinear time algorithms for constructing a basic partition. First, we describe and analyze a new synchronous algorithm called Sync_Part. Then, we show that the synchrony of the network is not important to achieve a sublinear time construction, and we provide a new asynchronous algorithm called Fast_Part.

In the remainder, we denote by V_f the set of finished nodes, i.e., nodes in a finished cluster. Furthermore, we are interested in active nodes in $V - V_f$, hence the degree of a node v is defined as its degree in the graph G_{V-V_f} induced by $V - V_f$.

1.5.1 A synchronous deterministic algorithm

In this section, we assume that the network is synchronous, i.e., there exists a global clock. At any time t, A_t denotes the set of active nodes (nodes not in V_f at time t), and $R_t = \{v \in A_t \mid d_v > n^{\frac{1}{k}}\}$ denotes the set of active nodes having high enough degrees at time t.

We remark that the sparsity condition for a single-node cluster rooted at some node v is $d_v > n^{\frac{1}{k}}$. Hence, a single-node cluster rooted at some node in $A_t \setminus R_t$ can not grow any layer. Thus, at any time t, we only let the nodes in R_t compete in order to grow some clusters. Once R_t becomes empty, we just let the remaining active nodes be finished single node clusters.

The new algorithm SYNC_PART works in two stages. The first stage is performed until time $\mathcal{T} = O(k^2 n^{1-1/k})$ is reached. The second stage begins at time $\mathcal{T}$ and lasts $O(1)$ time units.

In the next paragraphs, we give the details of algorithm SYNC_PART and discuss its correctness and its complexity.

First stage of the algorithm During this stage, all nodes execute algorithm DIST_PART with the following additional exploration rules:

- *If a node $v \in A_t$ is no longer in R_t, i.e., $v \in A_t \setminus R_t$, then v sets its identity to $-\infty$.*
- *Single-node clusters rooted at nodes in $A_t \setminus R_t$ do not explore any layer.*

Notice that the previous modifications are made only by single-node clusters that do not verify the sparsity condition. We use the same three rules of algorithm DIST_PART to manage the growth of other clusters rooted at any node in R_t.

Let us consider a single-node cluster rooted at $v \in A_t \setminus R_t$. Then, when applying the new rules, v sets its identity to $-\infty$. Hence, v has the lowest identity among all other possible identities. Therefore, node v will not stop the growth of another cluster rooted at a node of R_t. In fact, v can only be a part of other neighboring dense clusters (if it is asked to join). If the neighborhood of v is also in $A_t \setminus R_t$, then the cluster behaves as if it has the lowest identity, i.e., it does not explore any layer. In a practical implementation, a node needs to know whether it is in R_t or not. Since at any moment of algorithm DIST_PART a node is aware of its finished neighbors (see Remark 1.4.5), there are no further communications to be done by a node in order to know if it is still in R_t.

Second stage of the algorithm At time $\mathcal{T}$, all remaining active nodes in A_t stop computing and just decide to be finished single-node clusters.

Proposition 1.5.1 (Correctness) *Algorithm* SYNC_PART *emulates algorithm* BASIC_PART.

Lemma 1.5.2 *Let C be the cluster with the biggest identity among active nodes at some moment of the algorithm. We need $O(k^2)$ time in the worst case before the construction of C is finished.*

Proof On one hand, the communications performed by the algorithm are done using a broadcast convergecast process inside the BFS spanning tree of each cluster. Since a cluster has a radius at most $O(k)$, a broadcast (or a convergecast) costs at most $O(k)$ time units.

On the other hand, using the *Exploration rule*, cluster C always wins against its neighboring clusters and it always succeeds in exploring new layers. In the worst case, there will be at most $k-1$ new explored layers. Thus, it takes at most $O(k \cdot k)$ time before the construction of C is finished. ∎

Lemma 1.5.3 *At any time t such that $R_t \neq \varnothing$, $|A_{t+O(k^2)}| < |A_t| - n^{1/k}$*

Proof At time t such that $R_t \neq \varnothing$, we consider the node v in R_t with the biggest identity. Let us denote C the cluster rooted at v. Using Lemma 1.5.2, at time $t' = t + O(k^2)$, at least the $n^{1/k}$ nodes in the first layer of cluster C are not in $A_{t'}$. ∎

Lemma 1.5.4 *For $t = O(k^2 n^{1-1/k})$, $R_t = \varnothing$.*

Proof Using Lemma 1.5.3, at some time t, if $R_t \neq \varnothing$ then after a $O(k^2)$ time period p, the number of active nodes will decrease by at least $n^{1/k}$. By induction, after i ($\geqslant 0$) periods p, if $R_t \neq \varnothing$ then $|A_t| < n - i\, n^{1/k}$. Using the fact that $R_t \subseteq A_t$, we have at most $i = n^{1-1/k}$ periods p such that $R_t \neq \varnothing$. ∎

Since the first stage of the algorithm costs $\mathcal{T} = O(k^2 n^{1-1/k})$ time units and the second one is performed in $O(1)$ time units, we get the following theorem:

Theorem 1.5.5 (Time Complexity) *The time complexity of algorithm* SYNC_PART *is $O(k^2\, n^{1-1/k})$.*

Remark 1.5.6 *One can show that if we privilege the growth of clusters having the biggest couple (Radius,Id), then $\mathcal{T}$ can be chosen to be equal to $O(n^{1-1/k})$ for relevant range of $k < \log n$. This requires to show that a finished cluster of radius l has at least $n^{l/k}$ nodes and its construction costs at most $O(l^2)$ time. Then, by analyzing the function $n^{l/k}/l^2$, one can prove that the number of nodes becoming part of the biggest cluster is at least $\Omega(n^{1/k})$ each $O(1)$ time units. Thus, after $O(n^{1-1/k})$ time units, no cluster with radius $\geqslant 1$ can be constructed. Hence, $\mathcal{T}$ can be chosen to be $O(n^{1-1/k})$ and the factor k^2 can be removed in the time complexity. Since we assume that k is typically small comparing with $\log n$, we avoid going into the technical details.*

1.5.2 An asynchronous deterministic algorithm

Algorithm SYNC_PART uses the property that the system is synchronous to find a bound on the time $\mathcal{T}$ before no nodes can grow a non zero radius cluster. The time $\mathcal{T}$ informs all remaining active nodes that there are no more active dense clusters in the graph. This compels us to wait $\mathcal{T}$ time units even if the input graph is sparse. Furthermore, algorithm SYNC_PART can not be run in an asynchronous system without using any synchronizers (see,

e.g., Appendix A). In the following, we give a new asynchronous algorithm FAST_PART which does not use any global clock. The general idea of the algorithm is to allow sparse clusters to become finished without waiting until pulse $\mathcal{T}$. Our asynchronous algorithm shows that the key point for speeding up the construction does not rely on the global synchrony of the system, but rather on more local parameters.

Details of the algorithm Let us call a cluster C *dense*, if C has a radius at least 1 or if the single node v of C verifies $d_v > n^{\frac{1}{k}}$. We also define a *sparse cluster* to be a single-node cluster which is not dense (this corresponds to a node in $A_t \setminus R_t$ in algorithm SYNC_PART).

Algorithm FAST_PART uses the three rules of algorithm DIST_PART with the following modifications:

- *A dense cluster can explore a new layer if it has an identity bigger than those of its active dense neighbors at distance one or two.*
- *A sparse cluster is not allowed to explore a new layer.*
- *A sparse cluster declares itself finished single-node cluster if:*
  - *all its neighbors are sparse,*
  - *or if none of its dense neighbors has succeeded to explore a new layer.*

Using these new rules, a sparse node is allowed to declare itself finished if it is not explored by any neighboring cluster. This occurs if all neighbors are sparse or if the dense neighbors have not succeeded their explorations. This simple idea enables us to improve the time complexity of the previous synchronous algorithm.

It is obvious that the new modifications can be implemented using messages of size at most $O(\log(n))$ using the same techniques than in algorithm DIST_PART. For instance, we can use a couple *(Id,Dense)* for the cluster identifiers, where *Dense* is a boolean variable indicating whether a cluster is dense or sparse.

Proposition 1.5.7 (Correctness) *Algorithm* FAST_PART *emulates algorithm Basic_Part.*

Let Λ be the number of clusters of radius at least 1 at the end of algorithm FAST_PART. Then, the following theorem holds:

Theorem 1.5.8 (Time Complexity). *The worst case time complexity of algorithm* FAST_PART *satisfies:*

$$Time(\text{FAST\_PART}) = O(k^2\,\Lambda) = O(k^2\,n^{1-\frac{1}{k}})$$

Proof The new rules guarantee that a dense cluster is never stopped by a sparse one. In the worst case, no two dense clusters are constructed in parallel. Thus, let us consider the finished dense clusters in a *decreasing* order of their time construction.

The construction of a cluster costs at most $O(k^2)$. Thus, after at most $O(k^2\Lambda)$ time, it only remains active sparse clusters in the graph. In two rounds, all remaining sparse clusters detect that their neighbors are sparse. Thus, using the new rules, they become finished clusters and the algorithm terminates. Thus, the first part of the theorem holds. In addition, since the cluster are disjoint, it is obvious that for any graph and for any execution of the algorithm, Λ is bounded by $n^{1-\frac{1}{k}}$ which completes the proof. ∎

Remark 1.5.9 *Note that we can apply Remark 1.5.6 for the asynchronous algorithm* Fast_Part *in order to obtain a* $O(n^{1-\frac{1}{k}})$ *time complexity.*

Remark 1.5.10 *The bound* $O(k^2\,\Lambda)$ *becomes of special interest in the case of graphs where* Λ *can be shown to be small comparing with* $n^{1-\frac{1}{k}}$*, e.g., see Appendix B for the case study of Circulant graphs.*

Remark 1.5.11 *An important feature of algorithm* Fast_Part *is to focus on clustering dense regions of the graph. In fact, if we consider a graph with only some few dense regions, e.g., some cliques connected by some paths. Our algorithm will automatically capture the topology of the underlying graph and the clustering will have a high priority on dense regions. Moreover, the construction will be faster if the dense areas are far from each other.*

1.6 Sublinear randomized distributed partition

Although, the previous deterministic algorithms allow to construct cluster in parallel, their analysis is still sequential. In this section, we give a new randomized algorithm enabling us to compute a lower bound of the number of clusters constructed in parallel.

1.6.1 Randomized local elections

In [MSZ02], a randomized algorithm called L_2-*election* (LE_2 for short) is introduced in order to implement distributed algorithms described with relabeling systems (See Chapters 3 and 5). In algorithm LE_2, nodes are fighting to be centers of a ball of radius 1 so that the elected nodes can execute a computation step. Non elected nodes are part of at most one star which allows to execute local computations concurrently on closed balls of radius 1.

The authors in [MSZ02] studied the average number of nodes locally elected and they interpreted it as the degree of parallelism authorized by an algorithm. In the special case of our algorithm, that study gives an idea about the number of clusters constructed in parallel in one round and for $k = 2$. In fact, when taking $k = 2$ in the Basic_Part algorithm and using Theorem 1.1, the radius of the clusters produced by the decomposition can be either 0 (i.e., single-node clusters) or 1 (i.e., clusters containing a node v and its neighborhood $\mathcal{N}(v)$). Therefore, we can use algorithm LE_2 in order to implement algorithm Basic_Part. In fact, It is sufficient to run many rounds of algorithm LE_2 until there are no more active nodes in

the graph. Then, every time a node v is center of a star, it computes its neighborhood $|\mathcal{N}(v)|$ and decides to be either a radius 1 finished cluster or just a finished single-node cluster.

In the following, we use a generalization called LE_k of the local election algorithm of [MSZ02] in order to elect nodes which are centers of disjoint balls of radius $k \geqslant 2$. Our algorithm LE_k is used as a sub-procedure in algorithm ELECT_PART in order to construct the basic partition. The two algorithms are described in next paragraphs.

1.6.2 Description of algorithm Elect_Part

Algorithm ELECT_PART is depicted in Fig. 1.6 below. It runs in many phases until each node of the graph becomes part of a finished cluster. A phase of the algorithm is executed in two stages.

```
1: while There exist nodes not in a finished cluster do
2:    (0.) each node selects randomly an identity from a big set of integers.
3:    Stage 1: local election in balls of radius k
4:    (1.a) Each node v not in a finished cluster runs algorithm LE_k
5:    Stage 2: reinitialization
6:    (2.a) Each formed cluster C computes independently the sparsity condition for each layer j ⩽ k,
7:    if S contains a layer j violating the sparsity condition then
8:       (2.b) C releases all layers l ⩾ j and becomes a finished cluster,
9:       (2.c) nodes in released layers become single-node clusters.
10:   else
11:      if all neighbors are finished then
12:         (2.d) C becomes finished.
13:      end if
14:   end if
15:   (2.e) Break all non finished clusters and form new single-node clusters.
16: end while
```

Figure 1.6: Algorithm ELECT_PART

```
1: Round ← 0;
2: while Round < k do
3:    execute the Exploration Rule;
4:    Round ← Round + 1;
5:    if Non Success of the Exploration Rule then
6:       execute the Battle Rule;
7:    end if
8: end while
```

Figure 1.7: Algorithm LE_k: code for a cluster

In the first stage, we construct disjoint balls of radius at most k using algorithm LE_k depicted in Fig. 1.7. Algorithm LE_k is a variant of algorithm DIST_PART where the sparsity

condition does not matter: only the radius of elected clusters is important.

The second stage allows to compute finished clusters and to re-initialize the computations for a new phase. In fact, each cluster in the input of the second phase computes independently whether there is a layer not satisfying the sparsity condition (Step 2.a). This can be done distributively using convergecast and broadcast between the root and the leaves. If there exists a layer j violating the sparsity condition then the cluster rejects all layers $l \geqslant j$ and declares itself finished (Steps 2.b and 2.c). Otherwise, if all its neighbors are finished then the cluster can not grow any more and it also declares itself finished (Step 2.d). Finally, the remaining clusters are just broken into single-node clusters in order to run another phase (Step 2.e).

Remark 1.6.1 *Note that, algorithm LE_k grows balls of radius k whereas a radius $k-1$ suffices. This allows us to mark edges connecting a cluster with the nodes in the last rejected layer and thus avoiding the preferred edge election step needed for some applications (see the introduction of Chapter 2 for more explanations).*

1.6.3 Analysis of the algorithm

In this section, we compute a bound of the expected number of phases needed before algorithm Elect_Part terminates. The main idea of our analysis is to bound the number of nodes becoming part of a finished cluster in a phase, by using the number of clusters constructed in parallel in each phase.

In the sequel, we say that a node is *locally k-elected* if it succeeds the first stage of algorithm Elect_Part without losing against any other cluster, i.e., line 6 of algorithm LE_k is never executed by a locally k-elected node. We also use a parameter K such that: $\forall v \in V$, $\mathcal{N}_{2k}(v) \leqslant K$, where $\mathcal{N}_{2k}(v) = \{u \in V \mid d(u,v) \leqslant 2k\}$, i.e., K is an upper bound of the $2k$-neighborhood of any node.

Inspired by proofs in [MSZ02], the following proposition holds:

Proposition 1.6.2 *The expected number of nodes locally k-elected in a phase is lower bounded by:*

$$\sum_{v \in V - V_f} \frac{1}{\mathcal{N}_{2k}(v)} \;>\; \frac{|V - V_f|}{K}$$

Theorem 1.6.3 *Let T be the time complexity of algorithm* Elect_Part. *The expected value of T satisfies:*

$$\mathbb{E}(T) = O\left(k^2 \frac{\log(n)}{\log\left(\frac{K}{K-1}\right)}\right)$$

Proof Let $i \geqslant 0$ be a phase of the algorithm and $(G_i)_{i \geqslant 0}$ the sequence of graphs such that $G_0 = G$ and for all $i \geqslant 1$, G_i is the graph obtained by removing the nodes (and the

corresponding incident edges) belonging to a finished cluster from G_{i-1}. Obviously, G_i is the input graph of phase i.
Let X_i be the random variable which denotes the size of the graph G_i (the number of its nodes) for all $i \geqslant 0$, and let Y_i be the number of nodes locally k-elected in the i^{th} step. It is clear from Proposition 1.6.2 that we have the following inequality:

$$\mathbb{E}\,(Y_i \mid G_i) \geqslant X(G_i)/K$$

It is also easy to see that $X_{i+1} \leqslant X_i - Y_i$ for all $i \geqslant 0$. Thus,

$$\mathbb{E}\,(X_{i+1} \mid G_i) \leqslant X_i - \mathbb{E}\,(Y_i \mid G_i) \leqslant X_i(1 - \frac{1}{K})$$

For $i \geqslant 0$, we define a new r.v. Z_i by $Z_i = X_i/(1 - \frac{1}{K})^i$. Then, $\mathbb{E}\,(Z_{i+1} \mid G_i) \leqslant Z_i$. Thus, the r.v. Z_i is a super-martingale (see [Wil93]), and then

$$\mathbb{E}\,(Z_{i+1}) = \mathbb{E}\,(\mathbb{E}\,(Z_{i+1} \mid G_i)) \leqslant \mathbb{E}\,(Z_i)$$

A direct application of a theorem from [Wil93] chapter 9, yields $\mathbb{E}\,(Z_i) \leqslant Z_0 = n$. Thus

$$\mathbb{E}\,(X_i) = \left(1 - \frac{1}{K}\right)^i \mathbb{E}\,(Z_i) \leqslant n\left(1 - \frac{1}{K}\right)^i.$$

The algorithm terminates when $V_f = V$, i.e., $X_i = 1$. This implies that i is upper bounded by the ratio $\log(n)/\log(\frac{K}{K-1})$. Since both the first and the second stage of the algorithm take at most $O(k^2)$ time to be finished, the assertion in the theorem is proved. ∎

Remark 1.6.4 *The bound given by Theorem 1.6.3 does not take into account the size of the finished clusters at each phase but only the number of clusters constructed in parallel. Furthermore, the number of clusters constructed in parallel is just lower bounded using the variable K which corresponds to the initial graph G and not to the subgraph in the input of each phase. It would be very interesting to take all this features into account in order to get a better bound on the number of phases needed to terminate algorithm* ELECT_PART.

1.6.4 Improvements

In algorithm ELECT_PART, sparse nodes also participate in the computations and compete against other nodes in order to grow a ball. This slows down the construction because an elected sparse node will always form a finished single node cluster. Thus, we have to make some modifications in algorithm ELECT_PART. More specifically, *(i)* we prohibit that a sparse node stops the growth of a dense cluster, and *(ii)* we allow a sparse node to declare itself finished if it is not explored by any neighbor. Using these modifications, only the dense nodes are allowed to compete in order to grow a ball of radius k.

Similarly to algorithm FAST_PART, we let a dense node win against a sparse one using a couple $(Id, Dense)$. We also let a sparse node declare itself finished if it is not invaded by

any neighboring cluster, i.e., if dense neighbors loose their explorations or if all neighbors are sparse.

By considering the number of *dense nodes* at each phase and using the same arguments than in Theorem 1.6.3, we can find a bound on the expected number of phases needed to terminate the construction. Unfortunately, the theoretical analysis leads to the same bound than in Theorem 1.6.3.

This new modified version of algorithm Elect_Part is particularly interesting because it allows to express the high degree of parallelism of our method. For instance, consider a graph G such that $K = O(n^{\epsilon})$ with $\epsilon < 1$. This defines a large class of graphs for which we can achieve an improved time complexity, namely $O(\log(n)n^{\epsilon})$.

In Appendix B, we show that the expected running time of the modified algorithm is $O(\log(n))$ in the case of *Circulant* graphs.

1.7 Open questions

In this chapter, we focus on the time complexity of constructing sparse partitions in the practical $\mathcal{CONGEST}$ distributed model. One important motivation of our work is to understand and to study the effects of the congestion created by small messages into the overall time complexity of a distributed algorithm.

We left open the following questions:

1. Can we improve the time complexity of our algorithms from $n^{1-1/k}$ to $n^{1/k}$ in the $\mathcal{CONGEST}$ model? In particular, we remark that, in the case of small k $(2, 3, 4, 5)$, the locality level of the basic partition and the time complexity bound obtained using our technique are better than the bounds one can obtain by both assuming a more powerful distributed model, i.e., unlimited message size, and using techniques from [ABCP96]. This observation is intriguing and one can be interested in a lower bound on the time complexity of distributively computing the basic partition. Although, the case $k = 2$ seems hard to improve, we are optimistic that deterministic algorithms with better bounds exist for other values of k.

2. We have studied the sparse partition problem from a *locality* point of view. In other words, we only consider the problem of improving the time complexity. Can we improve the message complexity of our algorithms while maintaining the same time complexity?

The basic partition presented in this chapter is at the bottleneck of many distributed applications. In Appendix A, we apply our algorithms for network synchronizers. In the beginning of the next chapter, we show how we can apply our algorithms in order to construct *graph spanner* efficiently.

Chapter 2

On the Locality of Graph Spanners

Abstract.

This chapter concerns the efficient construction of sparse and low stretch spanners for unweighted arbitrary graphs with n nodes. All previous deterministic distributed algorithms, for constant stretch spanner of $o(n^2)$ edges, have a running time $\Omega(n^\epsilon)$ for some constant $\epsilon > 0$ depending on the stretch. Our deterministic distributed algorithms construct constant stretch spanners of $o(n^2)$ edges in $o(n^\epsilon)$ time for any constant $\epsilon > 0$.

More precisely, in the Linial's free model ($\mathcal{LOCAL}$ model), we construct in $n^{O(1/\sqrt{\log n})}$ time, for every graph, a $(3,2)$-spanner of $O(n^{3/2})$ edges. The result is extended to $(O(k^{2.322}), O(k^{2.322}))$-spanners with $O(n^{1+1/k})$ edges for every integer parameter $k \geqslant 1$. If the minimum degree of the graph is $\Omega(\sqrt{n})$, then, in the same time complexity, a 9-spanner with $O(n)$ edges can be constructed.

Résumé.

Dans ce chapitre, on considère un graphe sans poids de taille n. On s'intèresse à la construction efficace d'un "spanner" (sous graphe couvrant) ayant peu d'arêtes et un petit facteur d'étirement. Les meilleurs algorithmes distribués permettant de construire de façon déterministe un spanner avec $o(n^2)$ arêtes et un facteur d'étirement constant, ont une complexité en temps égale à $\Omega(n^\epsilon)$ où $\epsilon > 0$ et un paramètre qui dépend du facteur d'étirement. Les algorithmes que nous présentons dans ce chapitre permettent de construire de façon déterministe et distribuée des spanners avec les mêmes propriétés en un temps égale à $o(n^\epsilon)$ pour n'importe quel ϵ.

Dans le modèle de calcul sans congestion, nous construisons pour tout graphe un $(3,2)$-spanner avec $O(n^{3/2})$ arêtes en $n^{O(1/\sqrt{\log n})}$ unités de temps. Nous étendons également ce résultat pour construire un $(O(k^{2.322}, O(k^{2.322}))$-spanner avec $O(n^{1+1/k})$ arêtes en gardant la même complexité en temps pour tout paramètre entier $k \geqslant 1$. Dans le cas des graphes ayant un degrès minimum $\Omega(\sqrt{n})$, nous construisons, toujours avec la même complexité, un 9-spanner avec $O(n)$ arêtes.

2.1 Introduction

2.1.1 Motivations

This chapter deals with deterministic distributed construction of sparse and low stretch graph spanners. Intuitively, spanners can be thought of as a generalization of the concept of spanning trees. In fact, we look for a spanning subgraph such that the distance between any two nodes in the subgraph is bounded by some constant times the distance in the whole graph. More formally, H is a (α,β)-*spanner* of a graph G if H is a spanning subgraph of G, and if $d_H(u,v) \leqslant \alpha \cdot d_G(u,v) + \beta$ for all nodes u, v of G, where $d_X(u,v)$ denotes the distance from u to v in the graph X. The smallest pair (α,β) for which H is a (α,β)-spanner is called the *stretch* of H, and the *size* of H is the number of its edges. In the case $\beta = 0$ (resp. $\alpha = 0$), H is called a pure multiplicative (resp. additive) spanner. A pure multiplicative spanner with stretch $(\alpha, 0)$ is usually called an α-spanner. The quality of a spanner refers to the trade-off between the stretch and the size of the spanner.

Graph spanners are in the basis of various applications in distributed systems. For instance, Peleg and Ullman [PU89a] establish the relationship between the quality of spanners, and the time and message complexity of network synchronizers (see Appendix A). In addition, spanners are implicitly used for the design of low stretch routing schemes with compact tables [Cow01, EGP03, PU89b, RTZ02, TZ01], and appear in many parallel and distributed algorithms for computing approximate shortest paths and for the design of compact data-structures, a.k.a. distance oracles [BS04, GPPR04, RTZ05, TZ05, Coh98].

2.1.2 Preliminary results

It is well known that graph decompositions can be used in order to construct spanners. Let us first show how to efficiently construct graph spanner in the $\mathcal{CONGEST}$ distributed model using the sparse decomposition algorithms of Chapter 1. In fact, one immediate application of the basic partition of Chapter 1 (algorithm BASIC_PART) is the construction of a $(4k-3,0)$-spanner with $O(n^{1+\frac{1}{k}})$ edges for any n-node graph G. The spanner is obtained by considering the set of edges spanning each cluster and by selecting an inter-cluster edge for each pair of two neighboring clusters. The bounds on the stretch and the size of the spanner are a straightforward consequence of Theorem 1.3.1. In order to construct such a spanner *distributively*, we must first construct the basic partition, and second *select an edge between every two neighboring clusters*. However, we can both avoid this additional step of selecting preferred edges and at the same time improve the bound on the spanner size.

In fact, let us consider any cluster C under construction in algorithm DIST_PART. Before the construction of C is finished (just after the sparsity condition is no longer satisfied) and for every neighboring vertex u of C (u is on the last rejected layer of C), we select an edge from u to some v in the last layer of C and we add it to the spanner S. Moreover, we add

the BFS spanning tree of each cluster C to the spanner S.

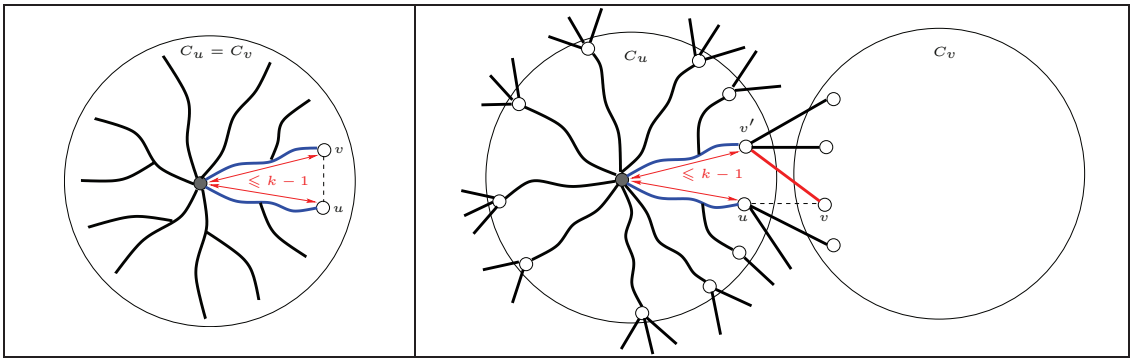


Figure 2.1: Stretch analysis using sparse partitions

Now, let us consider any edge (u, v) in the graph G. Using the fact that u and v must belong to some clusters, there exist two clusters C_u and C_v such that $u \in C_u$ and $v \in C_v$:

- If $C_u = C_v$ (first part of Fig. 2.1), then using the first property of Theorem 1.3.1, there is a path between u and v in the BFS spanning tree of C_u with size at most $2(k-1)$.
- Otherwise, if $C_u \neq C_v$, then w.l.o.g we can suppose that C_u was constructed before C_v. Thus, there exists an edge between v and a node $v' \in C_u$ which has been added to the spanner before finishing the construction of C_u (second part of Fig. 2.1). Using the first property of Theorem 1.3.1, there exists a path between u and v' in the BFS spanning tree of C_u with size $2(k-1)$. Hence, there exists a path between u and v in the spanner with size at most $2(k-1)+1 = 2k-1$.

Thus, for any edge (u, v), we have $d_S(u, v) \leqslant 2k-1$. Now, let us consider any pair of nodes w and w' (not necessarily neighbors), and a shortest path $\mathcal{P}$ in G between w and w'. Using the fact that for any edge (u, v) in the path $\mathcal{P}$, $d_S(u, v) \leqslant 2k-1$, it is straightforward that $d_S(w, w') \leqslant (2k-1)d_G(w, w')$. Thus, the stretch of the spanner S is $2k-1$. What about the size of S? On one hand, it is easy to see that the number of edges added by the BFS spanning trees is at most $n-1$. On the other hand, when adding the edges connecting the last rejected layer of some cluster C, the sparsity condition is no longer satisfied. Hence, for each cluster C we add at most $|\Gamma(C)| \leqslant n^{1/k}|C|$ edges. Summing over all clusters we add at most $n^{1/k}\sum_C |C|$ which is $n^{1+1/k}$ since the clusters are disjoint.

Notice that in all our sparse decomposition algorithms given in Chapter 1 the last rejected layer is always explored. Hence, the edges connecting a cluster with nodes in the last rejected layer are implicitly computed by our algorithms without any extra communications. Hence, we obtain the following results:

Theorem 2.1.1 *There is a deterministic algorithm that given a graph with n nodes and a fixed integer $k \geqslant 1$, constructs a $(2k-1, 0)$-spanner with $O(n^{1+1/k})$ edges in $O(n^{1-1/k})$ time in the worst case.*

Corollary 2.1.2 *There is a deterministic algorithm that given a graph with n nodes constructs a $(3,0)$-spanner with $O(n^{3/2})$ edges in $O(\sqrt{n})$ time in the worst case.*

2.1.3 Goals

To our knowledge, Theorem 2.1.1 provides the best time complexity for constructing $(2k-1,0)$-spanners with $O(n^{1+1/k})$ edges in a deterministic manner. Here, a natural question is the following: Can we provide distributed algorithms for constructing $O(k)$-spanners with better time complexity ? For instance, is it possible to construct spanners having good properties within a time that does not depend significantly on the stretch ? Generally speaking, we are interested in the *locality nature* of constructing graph spanners, that is:

Q2: what spanners can we compute assuming only some local knowledge?

Theoretically speaking, the *locality* of a distributed problem is often expressed in term of the *time* needed to resolve the problem. In fact, in the distributed setting, the best a node can do in $O(t)$ time units is to collect its t-neighborhood. For instance, $\Theta(\log^* n)$ time are necessary and sufficient to compute a maximal independent set for trees, bounded degree graphs, or bounded growth graphs with n nodes [CV86, GPS88, Lin92, KMNW05]. Results are known for other fundamental problems such as non-uniform coloring [Awe87, PS96], minimum spanning tree [Elk04a, Elk04b, LPSP01, LPSPP05, PR00], small dominating set [KP98, PV04], and maximal matching [KMW04, Lin92].

In this chapter, we give new results about the *locality* of graph spanners. The distributed model of computation we will be concerned with is the Linial's free model [Lin87], also known as $\mathcal{LOCAL}$ model in [Pel00]. In this model, we assume that communication is completely synchronous and reliable. At every time unit, each node may send (resp. receive) a message of unlimited size to (resp. from) all its neighbors, and can locally compute any function. The model also assumes that each node is equipped with a unique identifier. Much as PRAM algorithms in parallel computing give a good indication of parallelism, the $\mathcal{LOCAL}$ model gives a good indication of the *locality and distributed time*.

2.1.4 Related Works

Best known results for deterministic $O(k)$-spanners: Sparse and low stretch spanners can be constructed from (d,c)-decomposition of Awerbuch and Peleg [AP90b], that is a partition of the graph into clusters of diameter at most d such that the graph obtained by contracting each cluster can be properly c-colored. There are several deterministic algorithms for constructing (d,c)-decompositions [ABCP93, ABCP96, ABCP98, PS96] (see also the introduction of Chapter 1). The resulting distributed algorithms provide $O(k)$-spanners of size $O(n^{1+1/k})$, for any integral parameter $k \geqslant 1$. However, these algorithms run in $\Omega(n^{1/k+\epsilon})$

time, where $\epsilon = \Omega(1/\sqrt{\log n})$, and provide a stretch $s \geqslant 4k-2$. Better stretch-size trade-offs exist but with an increasing time complexity. In particular, our sparse partition algorithms (as stated in Theorem 2.1.1 and Corollary 2.1.2) provide the best trade-offs.

Other related deterministic and randomized results: Elkin [Elk01] discussed some distributed issues of constructing $(1+\epsilon, \beta(\epsilon,\rho))$-spanner of size $O(n^{1+\rho})$, where ϵ can be made arbitrarily small and β is independent of n but grows super-polynomially in ρ^{-1} and ϵ^{-1}. The deterministic algorithm given there has $O(n^{1+\rho})$ running time in the $\mathcal{CONGEST}$ model. Although, the running time of the algorithm can be showed to be better in the $\mathcal{LOCAL}$ model, the algorithm uses some distributed procedures having high time complexity, e.g., BFS spanning tree of the whole graph and DFS traversals. In addition, a network cover (similar to those of [ABCP98, ABCP93, Coh98]) is at the bottleneck of the algorithm. Therefore, the running time of the algorithm in the $\mathcal{LOCAL}$ model is at least equal to the one needed for constructing such covers. Unfortunately, the best known deterministic distributed cover algorithms are still slow and depend significantly on the stretch.

More recently, the result of [Elk01] was improved in [EZ04], and the authors gave a *randomized* $O(n^{\rho})$ time distributed algorithm to construct spanners with essentially the same properties (ρ still depends on the stretch). In fact, there exist some fast *randomized* algorithms for constructing sparse spanners with good properties. For instance, Baswana et al. [BS03, BKMP05] gave a randomized algorithm which computes an optimal $(2k-1)$-spanner with expected size $O(n^{1+1/k})$ in $O(k)$ time. The latter stretch-size trade-off is optimal since, according to an Erdös Conjecture, there are graphs with $\Omega(n^{1+1/k})$ edges and girth $2k+2$ (the length of the smallest induced cycle), thus for which every s-spanner requires $\Omega(n^{1+1/k})$ edges if $s < 2k+1$. The conjecture was proved for $k = 1, 2, 3, 5$ [Wen91].

As mentioned in [ABCP96], a randomized solution might not be acceptable in some cases, especially for distributed computing applications. In the case of graph spanners, deterministic algorithms that *guarantee* a high quality spanner are more than of a theoretical interest. Indeed, one cannot just run a randomized distributed algorithm several times to guarantee a good spanner, since it is impossible to check efficiently the global quality of the spanner in the distributed model.

In general, the *randomized* algorithms one can find use a technique based on sampling the graph nodes with a given probability. The sampling step implies *with high probability* a data structure having some desirable properties, e.g., a good expected size. There exist some theoretical methods to derandomize the sampling, e.g., [AS92]. However, no fast deterministic *distributed* implementations of these methods are known.

Related sequential algorithms: Many algorithms for computing approximated shortest paths [Elk01, DHZ00, CZ01, BGS05], routing [EGP98, ABNLP90, ABNLP89, PU88, AP92] and approximate distance oracles [RTZ05, TZ01, BS04] can be used at the aim of constructing graph spanners with various properties. Most of these algorithms are designed in the sequential or parallel settings. Those who can be *deterministically* adapted to distributed model of

computation could even not be implemented in linear time. The other *randomized* ones give no guarantees on the properties of the spanner, i.e., expected size.

For instance, Thorup and Zwick [TZ01] gave a randomized algorithm for computing a $(2k-1)$-approximate distance oracle with expected size $O(kn^{1+1/k})$ in $\widetilde{O}(kmn^{1/k})$ expected sequential time, where $|E| = m$. Recently, Roditty, Thorup and Zwick [RTZ05] gave a deterministic construction of such oracles with only a logarithmic loss in the time complexity and improves the results of many previous works, e.g., [BS04]. In particular, the technique in [RTZ05] allows to derandomize the construction of spanners in [BS03] and enables the deterministic construction of $(2k-1)$-spanners with size $O(kn^{1+1/k})$ in $O(km)$ sequential time. The algorithm of [RTZ05] uses a powerful technique that allows to efficiently compute the q-nearest neighbors and close dominating sets. However, that technique can not trivially work in the distributed setting without drastically losing its efficiency. For instance, given a set of nodes U, and in order to efficiently compute the set containing the 1-nearest nodes from every node to the set U, the authors in [RTZ05] add a virtual source node s to G and connect it to all nodes in U, then they run a single shortest path algorithm rooted at s. This technique can not be implemented distributively by an efficient algorithm.

Related parallel algorithms: In [LB96], the authors gave a parallel algorithm for constructing a $O(k^t)$-spanner with $O((\frac{n}{k^t})^{1+1/k}+n)$ edges in $O(\frac{n}{k^t}\log n)$ time on a CRCW PRAM machine. They also give a deterministic poly-logarithmic parallel algorithm for constructing a $2k$-spanner with $O(\min\{m, n^2/k\})$ which is interesting only for large values of k. In [Coh98], Cohen gives a randomized EREW PRAM parallel algorithm for constructing k-spanners with size $O(n^{1+(2+\epsilon)/k})$ in $O(\beta^2\log^2 n)$ time and $O(n^{1/\beta}m\beta\log^2 n)$ work where $\beta = k/(2+\epsilon/2)$. She also gives a deterministic (resp. randomized) sequential algorithm that runs in $O(mn^{c\cdot(2+\epsilon)/k})$ (resp. $\widetilde{O}(mn^{(2+\epsilon)/k})$) where $c = 1+\log_n m$. The algorithms given in [Coh98] are based on network covers.

2.1.5 Main results

We consider unweighted connected graphs with n nodes. All previous deterministic distributed algorithms for $O(1)$-spanner of size $o(n^2)$ have a running time $\Omega(n^\delta)$ for some constant $\delta > 0$ depending on the stretch. In this chapter, we construct constant stretch spanner of size $o(n^2)$ in $o(n^\epsilon)$ time for any constant $\epsilon > 0$.

More precisely, in the $\mathcal{LOCAL}$ model we construct in $n^{O(1/\sqrt{\log n})}$ time and for every graph a $(3,2)$-spanner of $O(n^{3/2})$ edges. The result is extended to larger stretch spanners of size $O(n^{1+1/k})$ for every $k \geqslant 1$. More precisely, we obtain stretches $s = (\alpha(k), \beta(k))$ which surprisingly depend on the positions of the first two leading 1's in the binary written of k. A detailed analysis is made for any parameter k and we show that $\alpha(k)$ and $\beta(k)$ are essentially bounded by $O(k^{\log_2 5})$. Furthermore, for any nodes u and v, the stretch bound depends on whether $d_G(u,v)$ is even or odd.

We also show that if the minimum degree of the graph is $\Omega(\sqrt{n}\,)$, then, in the same time complexity, spanners with small constant stretch and $O(n)$ edges can be constructed.

The previous algorithms have simple randomized versions with improved performances, i.e., $O(\log n)$ time complexity. In particular, we cans compute a 5-spanner of size $O(n \log^2 n)$ in $O(\log n)$ time if the minimum degree is $\Omega(\sqrt{n}\,)$.

The main idea to break the $O(n^\delta)$ time barrier is to abandon the optimality on the stretch-size trade-off. We show that constant stretch spanners can be constructed on the basis of a maximal independent set, i.e., a set of pairwise non-adjacent nodes, maximal for inclusion. This can be deterministically computed in $n^{O(1/\sqrt{\log n}\,)}$ time [ABCP96, PS96]. Therefore, the time complexity to construct our spanners is improved by a factor of $n^{1/k}$.

The generic algorithm is described in Section 2.2 and analyzed in Section 2.3, where a distributed implementation is presented.

We mainly reduce the problem to the computation of an *independent ρ-dominating set*, that is a set X of pairwise non-adjacent nodes such that every node of the graph is at distance at most ρ from X. Using the terminology of [Pel00], an independent ρ-dominating set if nothing else than a (ρ, s)-ruling set for some $s > 1$. Actually, in order to optimize the stretch, the main algorithm combines two strategies in a way depending on the binary written of k.

In Section 2.4, we present the main results about constant stretch spanners for general graphs. Observing that for $\rho = 1$ an independent ρ-dominating set is a maximal independent set, we conclude that our generic algorithm can be implemented to run in $n^{O(1/\sqrt{\log n}\,)}$ time for $\rho = 1$. Several optimizations are then proposed including randomization and graphs of large minimum degree.

2.2 A Generic Algorithm

2.2.1 Definitions

Let us consider an unweighted connected graph $G = (V, E)$. Given an integer $t \geqslant 1$, the t-th power of G, denoted by G^t, is the graph obtained from G by adding an edge between any two nodes at distance at most t in G. For a set of nodes H, $G[H]$ denotes the subgraph of G induced by H. For $X, Y \subseteq V$, let $d_G(X, Y) = \min\{d_G(x, y) \mid x \in X \text{ and } y \in Y\}$.

We associate with each $v \in V$ a *region*, denoted by $R(v)$, that is a set of nodes containing v and inducing a connected subgraph of G. Given $U \subseteq V$, G_U denotes the graph whose node set is U, and there is an edge between u and v in U if $d_G(R(v), R(u)) = 1$. We denote by $R^+(v) = \{u \in V \mid d_G(u, R(v)) \leqslant 1\}$ and by $R^+_U(v) = \{u \in U \mid d_G(R(u), R(v)) \leqslant 1\}$.

The *eccentricity* of a node v in G is defined as $\max_{u \in V}\{d_G(u, v)\}$. For a node $v \in X$, we denote by $\mathrm{BFS}(v, X)$ a Breadth First Search spanning tree in X rooted at v. We define $\mathrm{IDS}(G, \rho)$ as any independent ρ-dominating set of G. Finally, we define the integer $\ell(x)$ as

follows:

$$\ell(x) = \begin{cases} -1 & \text{if } x \leqslant 0, \\ \lfloor \log_2 x \rfloor & \text{otherwise.} \end{cases}$$

2.2.2 Description of the algorithm

The main idea of the algorithm is to find an efficient clustering of dense regions in the graph. A high level description of the algorithm, named SPANNER, is given in Fig. 2.2. Intuitively, i_0 represents the relative position of the first two leading 1's in the binary written of k.

SPANNER

Input: a graph $G = (V, E)$ with $n = |V|$, and integers $\rho, k \geqslant 1$
Output: a spanner S

1. $i_0 := \ell(k) - \ell(k - 2^{\ell(k)})$; $U := V$; $r = 0$; $\forall v \in V$, $R(v) := \{v\}$, and $c(v) := v$
2. **for** $i := 1$ **to** $\ell(k) + 1$ **do: if** $i = i_0$ **then** STRATEGY 1 **else** STRATEGY 2

Figure 2.2: The algorithm SPANNER.

The algorithm works in many phases, where new regions are formed at each phase. There are two types: the light regions (L) and the heavy regions (H). At a given phase, some of the heavy regions are selected and enlarged by including nodes from other neighboring regions. One important observation is that each new enlarged region is connected and the new constructed regions are mutually disjoint.

At each phase of the algorithm, one of the two strategies depicted in Fig. 2.3 and Fig. 2.4 applies. The main idea behind the two strategies is the same: choose some well selected dense regions and merge them with the other ones in order to form new larger regions. The main difference is that the density of a region is computed in a different way. The stretch of the output spanner depends on the way the radius of the regions increases and on the total number of phases of the algorithm, depending on the volume of the regions. And, radius and volume increase very differently.

On one hand, in STRATEGY 1, a region is dense if its neighborhood is $n^{1/k}$ times greater than its size. Applying only STRATEGY 1 allows to obtain small stretch for small values of k. However, asymptotically, the stretch is exponential in k. On the other hand, in STRATEGY 2, a region is dense if the number of its neighboring regions is $n^{1/k}$ times greater than its size which provides an exponential growth of the size of a region. Applying only STRATEGY 2 allows to obtain asymptotically stretches polynomial in k.

The main idea of algorithm SPANNER is to switch from one strategy to an other at each phase in order to obtain the smallest possible stretch. A full analysis shows that, by alternating

Strategy 1 and Strategy 2, the best stretch can be obtained by applying Strategy 1 only once at a well chosen phase i_0. Typically, $i_0 = p - q$ for $k = 2^p + 2^q$ with $p > q$.

We associate with each region $R(v)$ an active node, called *center*, and the set of centers forms U. Initially, each node is the center of the region formed by itself. Each phase $i \in \{1, \ldots, \ell(k) + 1\}$ can be decomposed in seven parts we briefly sketch.

1. $L := \left\{v \in U,\ |R^+(v)| \leqslant n^{1/k} \cdot |R(v)|\right\}$ and $H := U \setminus L$;
2. $\forall (u, v) \in L \times V$ such that $\exists$ edge e between $R(u)$ and v, $S := S \cup \{e\}$
3. $X := \mathrm{IDS}(G^{2(r+1)}[H], \rho)$
4. $\forall z \in V$, if $d_G(z, X) \leqslant (2\rho + 1)r + 2\rho$, then set $c(z)$ to be its closest node of X, breaking ties with identities.
5. $\forall v \in X$, $R(v) := \{z \in V \mid c(z) = v\}$
6. $\forall v \in X$, $S := S \cup \mathrm{BFS}(v, R(v))$
7. $U := X$ and $r := (2\rho + 1)r + 2\rho$

Figure 2.3: Strategy 1.

1. $L := \left\{v \in U,\ |R^+_U(v)| \leqslant n^{1/k} \cdot |R(v)|\right\}$ and $H := U \setminus L$
2. $\forall (u, v) \in L \times U$ such that $\exists$ edge e between $R(u)$ and $R(v)$, $S := S \cup \{e\}$
3. $X := \mathrm{IDS}((G_U)^2[H], \rho)$
4. $\forall u \in U$, if $d_{G_U}(u, X) \leqslant 2\rho$, then set $c(u)$ to be its closest node of X in G_U, breaking ties with identities.
5. $\forall v \in X$, $R(v) := \{R(u) \mid u \in U \text{ and } c(u) = v\}$
6. $\forall v \in X$, $S := S \cup \mathrm{BFS}(v, R(v))$
7. $U := X$ and $r := (4\rho + 1)r + 2\rho$

Figure 2.4: Strategy 2.

In Step 1, we compute the two sets H and L corresponding respectively to heavy and light regions. In Step 2, a light region is connected with some neighboring nodes. This step is crucial in the stretch bound analysis. If Strategy 1 is applied, then each light region is

connected with each neighboring node in V, i.e., $\forall u \in L, R^+(u)$ is spanned. If STRATEGY 2 is applied, then each light region is connected with every neighboring region. Note that at the beginning of a given phase, every region is spanned by a BFS tree constructed in Step 6 of the previous phase.

The nodes in H are then processed at the aim of constructing new regions with a set of new centers. The key point of our construction is to efficiently merge *all* the regions defined by the set H into *more dense*, *connected* and *disjoint* regions. In order to guarantee that the algorithm terminates quickly, the dense regions must grow enough. More precisely, if a dense region $R(v)$ is enlarged it must contain at least its neighborhood $R^+(v)$ when STRATEGY 1 is applied or its neighborhood in the graph G_U if STRATEGY 2 is applied. It is clear that two regions at distance one or two (in G or in G_U depending on the strategy 1 or 2) cannot grow simultaneously without overlapping. Thus, the difficulty is to elect in an efficient way the centers of regions that are allowed to grow in parallel.

In Step 3, we compute an independent ρ-dominating set X in the graph $G^{2(r+1)}[H]$ if STRATEGY 1 is applied (resp. $(G_U)^2[H]$ for STRATEGY 2), where r is a radius that grows at each phase. The set X defines the set of nodes allowed to grow in parallel.

In order to guarantee that nodes in non selected regions in Step 3 (the set $H \setminus X$) will be spanned by the output spanner, we must merge them with nodes in the selected regions. Thus, in Step 4, we define a coloring strategy allowing a correct merge process. In fact, in order to ensure that the new regions are disjoint, we let nodes choose their new region in a consistent manner, i.e., a node chooses to be in the region of the closest node in X breaking ties using identities. If STRATEGY 1 is applied then each node chooses by itself its new dominator, i.e., its new region. However, once a node u chooses its new dominator node v, and in order to ensure that the new formed regions are connected, we include all the nodes in the shortest path between u and v, even those in non dense region. If STRATEGY 2 is applied then, the center of each region chooses a new region and merge its whole region with the new chosen region.

In Step 5, the new regions are formed according to the coloring step. Note that as soon as the new region are formed, they are spanned in Step 6. Finally, in Step 7, the set U and the variable r are updated for the next phase.

2.2.3 Examples for $\rho = 1$

In the example of Fig. 2.5, we show how to connect a sparse region with neighboring ones using STRATEGY_1. In the example of Fig. 2.6, we show how to merge the regions using STRATEGY_1 in the case $\rho = 1$ and all regions are dense.

In the example of Fig. 2.7, we show how to connect a sparse region with neighboring ones using STRATEGY_2. In the example of Fig. 2.8, we show how to merge the regions using STRATEGY_2 in the case $\rho = 1$ and all regions are dense.

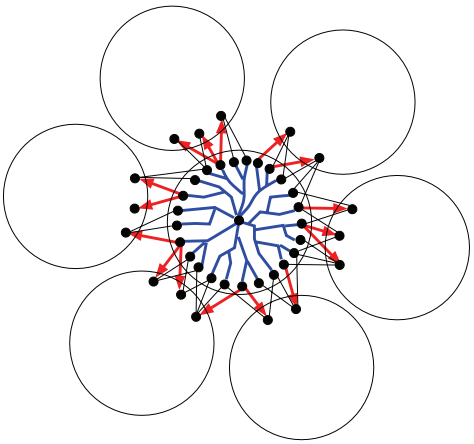
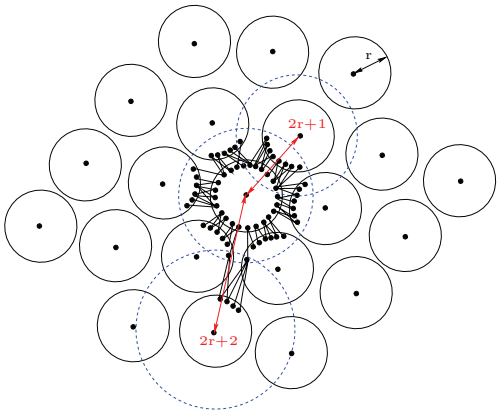
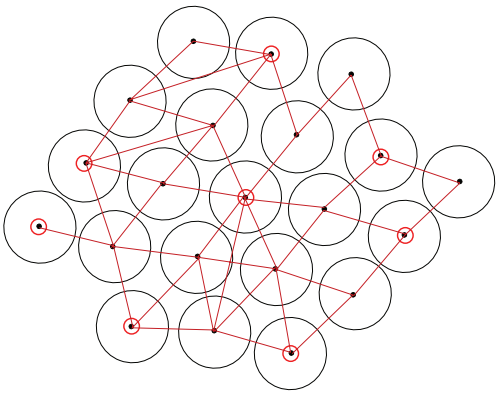


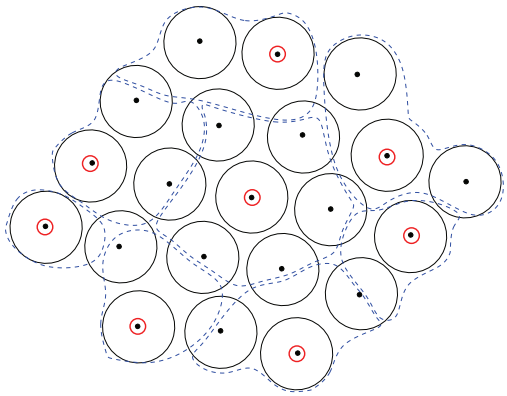
Figure 2.5: An example of spanning a sparse region with $\textsc{Strategy}_1$ (Steps 2 and 1)



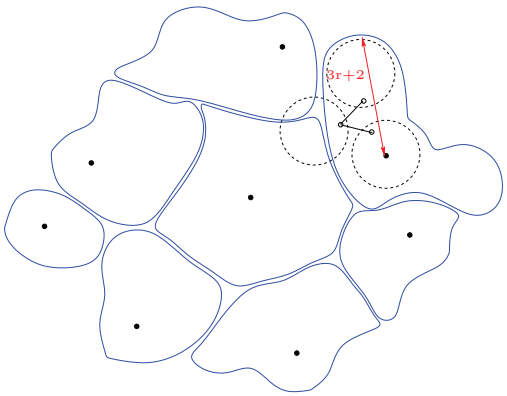
(a) Starting configuration.



(b) Constructing the IDS (Step 3).



(c) Merging the new regions (Step 4).



(d) The regions are constructed (Step 5).

Figure 2.6: An example of merging dense regions with $\textsc{Strategy}_1$

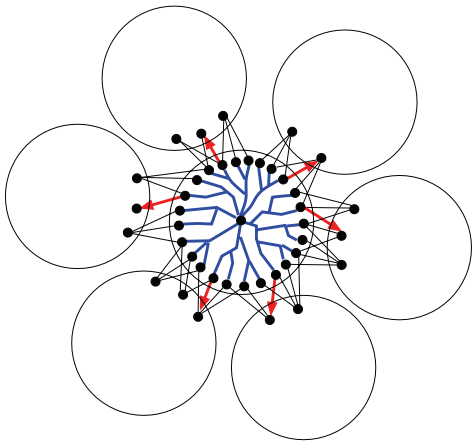
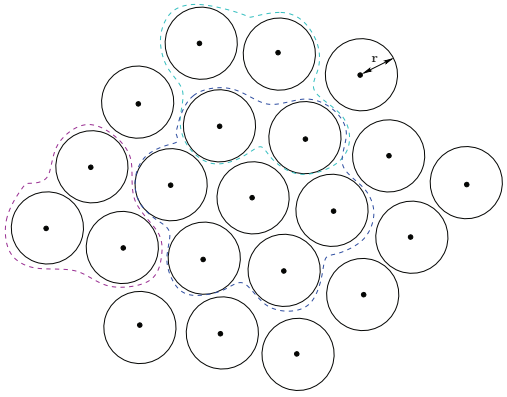
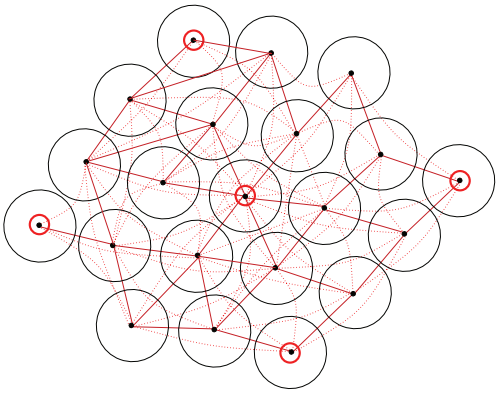


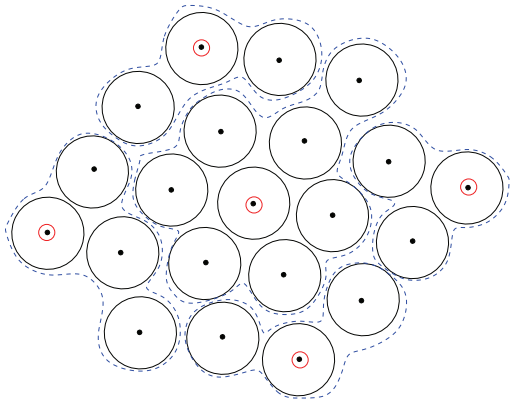
Figure 2.7: An example of spanning a sparse region with STRATEGY$_2$ (Steps 2 and 6)



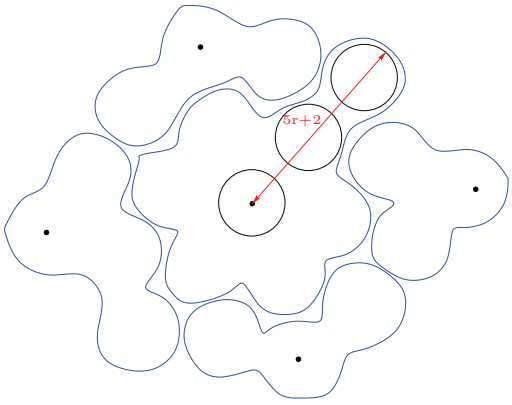
(a) Starting configuration.



(b) Constructing the IDS (Step 3).



(c) Merging the new regions (Step 4).



(d) The regions are constructed (Step 5).

Figure 2.8: An example of merging dense regions with STRATEGY$_2$

2.3 Analysis of the Algorithm

For every phase i, we denote by H_i (resp. X_i and L_i) the set H (resp. X and L) computed during phase i, i.e., after Steps 1 and 3 of phase i. Similarly, we denote by $c_i(z)$ the color of z assigned during phase i, i.e., after Step 4 of phase i. We denote by U_i the set U at the beginning of phase i, and r_i denotes the value of r at the beginning of phase i. For a node $v \in U_i$, we denote by $R_i(v)$ the region of v at the beginning of phase i. In the following we need the four important properties.

Lemma 2.3.1 *At the beginning of phase i, every $v \in U_i$ is of eccentricity at most r in $G[R_i(v)]$.*

Proof We prove the lemma by induction. The lemma is clearly true for $i = 1$. Let us consider a node $v \in U_i$ and a node $z \in R_i(v)$ at a given phase $i > 1$. From Step 7 of the algorithm, U_i corresponds to the set X_{i-1}. Thus, the region $R_i(v)$ was computed in Step 5 of phase $i-1$.

- *Case 1*: Strategy 1 was used at phase $i-1$. Hence, using Step 5, for every node $z \in R_i(v)$, z was colored v at phase $i-1$. Thus, $\forall z \in R_i(v), c_{i-1}(z) = v$. Thus, from the coloring step, there exists a shortest path $P = (z = z_1, z_2, z_3, \ldots, z_l, v)$ connecting z and v in G, with $l \leqslant 2(r_{i-1}+1)\rho + r_{i-1}$. Let us consider $z_j \in P$ with $1 \leqslant j \leqslant l$. Let us first note that $d_G(z_j, v) \leqslant 2(r_{i-1}+1)\rho + r_{i-1}$ and $v \in X_{i-1}$. Hence, z_j has also chosen a color at phase $i-1$. Using the fact that the coloring is consistent, $c_{i-1}(z_j) = v$. Thus, using Step 5, $P \subseteq R_i(v)$. Hence, $d_{G[R_i(v)]}(z, v) \leqslant 2(r_{i-1}+1)\rho + r_{i-1} = r_i$.

- *Case 2*: Strategy 2 was used at phase $i-1$. Hence, using Step 5, there exists $u \in U_{i-1}$ such that $c_{i-1}(u) = v$ and $z \in R_{i-1}(u)$. Using Step 4, there exists a path P in $G_{U_{i-1}}$ such that $P = (u = u_1, u_2, ..., u_l = v)$ with $l \leqslant 2\rho$. From now, it is convenient for the proof of the lemma to view the path P as a directed path where u_1 is the leftmost node and u_l is the rightmost one. From the definition of the graph $G_{U_{i-1}}$, for every pair of successive nodes u_j and u_{j+1}, there exists a pair of neighboring nodes z_j and z_{j+1} such that $z_j \in R_{i-1}(u_j)$, $z_{j+1} \in R_{i-1}(u_{j+1})$ and $(z_j, z_{j+1}) \in E$. Thus, $d_{R_{i-1}(u_j) \cup R_{i-1}(u_{j+1})}(u_j, u_{j+1}) \leqslant d_{R_{i-1}(u_j)}(u_j, z_j) + 1 + d_{R_{i-1}(u_{j+1})}(u_{j+1}, z_{j+1})$. Thus, using the induction hypothesis, $d_{R_{i-1}(u_j) \cup R_{i-1}(u_{j+1})}(u_j, u_{j+1}) \leqslant 2r_{i-1} + 1$. Therefore, $d_{\cup_{1 \leqslant j \leqslant l} R_{i-1}(u_j)}(u_1, u_l) \leqslant 2\rho \cdot (2r_{i-1} + 1) = 4\rho \cdot r_{i-1} + 2\rho$. Because we have $z \in R_{i-1}(u = u_1)$, then $d_{\cup_{1 \leqslant j \leqslant l} R_{i-1}(u_j)}(z, u_l) \leqslant 4\rho \cdot r_{i-1} + 2\rho + r_{i-1} = (4\rho+1)r_{i-1} + 2\rho$.

  Notice that $\forall 1 \leqslant j \leqslant l$, $d_{G_{U_{i-1}}}(u_j, v) \leqslant 2\rho$. Hence, u_j has also chosen a color in Step 4 of phase $i-1$. Using the fact that the coloring is consistent, $c_{i-1}(u_j) = v$. Thus, $\forall 1 \leqslant j \leqslant l$, $R_{i-1}(u_j) \subset R_i(v)$. Therefore, $d_{R(v)}(z, v) \leqslant (4\rho+1)r_{i-1} + 2\rho$. By Step 7, we have $r_i = (4\rho+1)r_{i-1} + 2\rho$ which completes the proof. ∎

Lemma 2.3.2 *At the beginning of phase i, for every two nodes $u \neq v \in U_i$, $R_i(u) \cap R_i(v) = \varnothing$.*

Proof We prove the lemma by induction. The lemma is clearly true for $i = 1$. Let us consider a phase $i > 1$. Let us consider two nodes $u, v \in U_i$ and $u \neq v$. From Step 7 of the algorithm, the set U_i corresponds to the set X_{i-1}. Hence, $u, v \in X_{i-1}$.

- *Case 1*: STRATEGY 1 was used at phase $i-1$. Suppose that there exists $z \in R_i(u) \cap R_i(v)$. From Step 5 of phase $i-1$, we have: $c_{i-1}(z) = u$ and $c_{i-1}(z) = v$. Thus, $u = v$ which is a contradiction.

- *Case 2* STRATEGY 2 was used at phase $i-1$. Suppose that there exists $z \in R_i(u) \cap R_i(v)$. From Step 5 of phase $i-1$, there exist $w_1, w_2 \in U_{i-1}$ such that $c_{i-1}(w_1) = u$ and $c_{i-1}(w_2) = v$ and $z \in R_{i-1}(w_1) \cap R_{i-1}(w_2)$. Using the induction hypothesis, $w_1 = w_2$. Thus, $c_{i-1}(w_1) = c_{i-1}(w_2) = u = v$ which is a contradiction.

Therefore for every two nodes $u, v \in U_i$ such that $u \neq v$, $R_i(u) \cap R_i(v) = \varnothing$, which completes the proof. ∎

Lemma 2.3.3 *At the beginning of phase $i \neq i_0$, if $|R_i(v)| \geqslant \mathcal{V}_i$ for every $v \in U_i$, then $|R_{i+1}(v)| \geqslant n^{1/k} \cdot \mathcal{V}_i^2$ for every $v \in U_{i+1}$.*

Proof First, because $i \neq i_0$, the STRATEGY 2 is applied at phase i. Notice also that the set U_{i+1} corresponds to the set X_i ($\subseteq H_i \subseteq U_i$) computed at phase i. Let us consider a node v in X_i. Consider a node $u \in R^+_{U_i}(v)$. Clearly, $u \in U_i \setminus X_i$, otherwise the independence of set X_i is violated. Suppose that there exists a node v' at distance 1 from u in the graph G_{U_i}. Thus, v' is at distance 1 from v in $G^2_{U_i}$. Thus, $v' \in U_{i-1} \setminus X_{i-1}$, otherwise the independence of set X_i is violated. Therefore, v is the closest node in X_i to u. Thus, $c_i(u) = v$ and hence, $R_i(u) \subseteq R_{i+1}(v)$. Therefore, using Lemma 2.3.2,

$$|R_{i+1}(v)| > |R^+_{U_i}(v)| \cdot \min\{R_i(u) \mid u \in U_i \text{ and } c_i(u) = v\} \ .$$

Thus from Step 1 of phase i, we have

$$\begin{aligned} |R_{i+1}(v)| &> n^{1/k} \cdot |R_i(v)| \cdot \mathcal{V}_i \\ &> n^{1/k} \cdot \mathcal{V}_i^2 \end{aligned}$$ ∎

Lemma 2.3.4 *For every node $u \in V$, there exists a phase i and a node $v \in V$ such that:*

- *at the beginning of phase i, $v \in U_i$ and $u \in R_i(v)$; and*
- *v is in the set L_i computed in Step 1 of phase i.*

Proof Let us denote by $i_1 = \ell(k)$ and $i_2 = \ell(k - 2^{\ell(k)})$, i.e., $i_0 = i_1 - i_2$. Let $\mathcal{V}_i$ the minimum size of any region corresponding to a node in U_i.

First, let us show that

Claim Every node in $U_{\ell(k)+1}$ is in $L_{\ell(k)+1}$, i.e. $U_{\ell(k)+1} = L_{\ell(k)+1}$.

Proof of the claim

- *Case 1*: $i_2 = -1$. Hence, $i_0 = \ell(k) + 1$ and $k = 2^{\ell(k)}$. By induction and using Lemma 2.3.3, at the beginning of phase i_0, the size of the region of any node in U_{i_0} is at least $\mathcal{V}_{i_0} \geqslant n^{(2^{i_0-1}-1)/k} = n^{(k-1)/k}$ (because $\mathcal{V}_1 = 1$). Hence, for every $v \in U_{i_0}$,

$$n^{1/k} \cdot |R_{i_0}(v)| \geqslant n^{1/k} \cdot n^{(k-1)/k} = n \geqslant |R^+_{i_0}(v)|$$

  Because, Strategy 1 is applied at phase i_0, after Step 1, $U_{\ell(k)+1} = L_{\ell(k)+1}$.

- *Case 2*: $i_2 \geqslant 0$. We have $\mathcal{V}_{i_0} \geqslant n^{(2^{i_0-1}-1)/k}$. At phase i_0, we apply Strategy 1. Thus, using the same arguments than in Lemma 2.3.3, the new enlarged regions constructed at phase i_0 contain all their neighborhood (in G) otherwise the independence of set X_{i_0} is violated. Thus, $\mathcal{V}_{i_0+1} \geqslant n^{1/k} \cdot \mathcal{V}_{i_0} \geqslant n^{2^{i_0-1}/k}$.

  - *Subcase 2.1*: $i_2 = 0$. Then, $i_0 = i_1 = \ell(k)$ and $k = 2^{\ell(k)} + 1$. Thus,

$$n^{1/k} \cdot \mathcal{V}^2_{\ell(k)+1} \geqslant n^{(1+2^{\ell(k)})/k} = n$$

  - *Subcase 2.2*: $i_2 \neq 0$. Then, by applying Lemma 2.3.3 to phase i going from $i_0 + 1$ to $\ell(k) + 1$ (excluding $\ell(k) + 1$), we have

$$\begin{aligned}\mathcal{V}_{\ell(k)+1} &\geqslant n^{1/k} \cdot \mathcal{V}^2_{\ell(k)} \geqslant n^{1/k} \cdot \left(n^{1/k} \cdot \mathcal{V}^2_{\ell(k)-1}\right)^2 \geqslant \ldots \geqslant n^{(1+2^1+\cdots+2^{j-1})/k} \cdot \mathcal{V}^{2^j}_{\ell(k)+1-j} \\ &\geqslant n^{\sum_{j=0}^{(\ell(k)+1)-(i_0+1)-1} 2^j/k} \cdot \mathcal{V}^{2^{(\ell(k)+1)-(i_0+1)}}_{i_0+1} \geqslant n^{(2^{\ell(k)-i_0}-1+2^{\ell(k)-1})/k}\end{aligned}$$

    Thus,

$$n^{1/k} \cdot \mathcal{V}^2_{\ell(k)+1} \geqslant n^{(1+2(2^{\ell(k)-i_0}-1+2^{\ell(k)-1}))/k} \geqslant n^{(2^{\ell(k)-i_0+1}-1+2^{\ell(k)})/k}$$

    Note that $\ell(k) - i_0 + 1 = i_1 - i_0 + 1 = i_2 + 1$. Thus,

$$2^{\ell(k)-i_0+1} - 1 + 2^{\ell(k)} = 2^{i_2+1} + 2^{i_1} - 1 \geqslant k$$

    Thus, we get $n^{1/k} \cdot \mathcal{V}^2_{\ell(k)+1} \geqslant n$.

  Hence, in both subcases, we have

$$\frac{n}{\mathcal{V}_{\ell(k)+1}} \leqslant n^{1/k} \cdot \mathcal{V}_{\ell(k)+1}$$

  Now, let us take a node $v \in U_{\ell(k)+1}$. Because Strategy 2 is applied at phase $\ell(k)+1$, v is in $L_{\ell(k)+1}$ iff $|R^+_{U_{\ell(k)+1}}(v)| \leqslant n^{1/k} \cdot |R_{\ell(k)+1}(v)|$. Let us show that, in fact v is in $L_{\ell(k)+1}$. There is at least $\mathcal{V}_{\ell(k)+1}$ nodes in a region, thus using Lemma 2.3.2, we have

$$|R^+_{U_{\ell(k)+1}}(v)| \leqslant \frac{n}{\mathcal{V}_{\ell(k)+1}} \leqslant n^{1/k} \cdot \mathcal{V}_{\ell(k)+1} \leqslant n^{1/k} \cdot |R_{\ell(k)+1}(v)|$$

  Thus, all nodes of $U_{\ell(k)+1}$ are in $L_{\ell(k)+1}$.

Therefore, in both cases, $U_{\ell(k)+1} = L_{\ell(k)+1}$. ∎

We are now ready to prove Lemma 2.3.4.

Let us consider a node $u \in V$. At the beginning of the algorithm, the node u is also in U. If $R(u)$ does not satisfy the condition of Step 1 of the algorithm then $u \in L_1$. Hence, the lemma is true. Otherwise, u participates in Step 3 and u is at distance at most 2ρ from a node in X_1. Thus, u joins the region of some other node $u_1 \in X_1 \subset U_2$ (possibly equal to u) and $u \in R_2(u_1)$. Let us call u_1 the new dominator of u. At the next phase $i = 2$, if u_1 is in L_2 then the lemma holds. Otherwise, u_1 participates in Step 3. There are two cases:

- *Case 1*: Suppose that STRATEGY 1 is applied at phase 2. Then, u_1 is at distance at most $2(r_2+1)\rho$ from a node in set X_2 in the G. Hence, u is at distance at most $2(r_2+1)\rho+r_2$ from a node in set X_2 in the graph G. Hence u will be colored in phase 2 and it will join the region of a new dominator $u_2 \in X_2 \subset U_3$.
- *Case 2*: Suppose that STRATEGY 2 is applied at phase 2. Then, u_1 is at distance at most 2ρ from a node in set X_2 in the graph G_{U_2}. Hence, the region of u_1 joins the region of a new dominator $u_2 \in X_2 \subset U_3$. Because u is in the region of u_1, u will join the region of the new dominator u_2.

In both cases, u will be in the region of a new dominator $u_2 \in U_3$. It is not difficult to show by induction that, if the $(i-1)$-th dominator of u is still not in L_i at phase i, then u will join the region of a new dominator u_i.

In the worst-case, the $\ell(k)$-th dominator of u will be in $L_{\ell(k)+1}$ in phase $\ell(k)+1$. Thus, there must exist some node v such that u is in the region of v at the beginning of some phase i, i.e., v is the $(i-1)$-th dominator of u, and v is in the set L_i computed in Step 1 of phase i. ∎

2.3.1 Stretch analysis

In the following, we denote $i_1 = \ell(k)$ and $i_2 = \ell(k - 2^{\ell(k)})$, i.e., $i_0 = i_1 - i_2$.

Lemma 2.3.5 *For any integer $\rho \geqslant 1$, if $i_0 = \ell(k)+1$, i.e., $k = 2^{\ell(k)}$, then for every phase i such that $1 \leqslant i \leqslant \ell(k)+1$, we have:*

$$r_i \;=\; \frac{1}{2} \cdot \big((4\rho+1)^{i-1} - 1\big)$$

Proof From the initialization step of the algorithm, we have $r_1 = 0$. Since STRATEGY$_2$ is applied until phase i_0, for every $1 < i \leqslant i_0$, we have $r_i = (4\rho+1)r_{i-1} + 2\rho$. Thus, the lemma holds by induction. ∎

Lemma 2.3.6 *For any integer $\rho \geqslant 1$, if $i_0 < \ell(k)+1$ then for every phase i such that $1 \leqslant i \leqslant \ell(k)+1$, we have:*

$$r_i = \begin{cases} \frac{1}{2}\left((4\rho+1)^{i-1}-1\right) & \text{if } 1 \leqslant i \leqslant i_0, \\ \frac{1}{2}(2\rho+1)\cdot\left((4\rho+1)^{i_0-1}-1\right)+2\rho & \text{if } i = i_0+1, \\ \frac{1}{2}(4\rho+1)^{i-i_0-1}\Big((2\rho+1)(4\rho+1)^{i_0-1}+2\rho\Big)-\frac{1}{2} & \text{if } i_0+1 < i \leqslant \ell(k)+1. \end{cases}$$

Proof From the initialization step of the algorithm, we have $r_1 = 0$. Since Strategy$_2$ is applied until phase i_0, for every $1 < i \leqslant i_0$, we have $r_i = (4\rho+1)r_{i-1}+2\rho$. Thus, by induction, if $1 \leqslant i \leqslant i_0$, we have $r_i \leqslant \frac{1}{2}\cdot\left((4\rho+1)^{i-1}-1\right)$.

In phase i_0, Strategy$_1$ is applied. Thus, $r_{i_0+1} = (2\rho+1)r_{i_0}+2\rho$. Thus,

$$r_{i_0+1} = \frac{1}{2}(2\rho+1)\cdot\left((4\rho+1)^{i_0-1}-1\right)+2\rho \tag{2.1}$$

Now, for other phases i such that $i_0+1 < i \leqslant \ell(k)+1$, Strategy$_2$ is applied until phase $\ell(k)+1$. Thus, for $i_0+1 < i \leqslant \ell(k)+1$, we have $r_i = (4\rho+1)r_{i-1}+2\rho$. Thus, by induction, for every $i_0+1 < i \leqslant \ell(k)+1$,

$$r_i = (4\rho+1)^{i-i_0-1}\cdot r_{i_0+1}+\frac{1}{2}((4\rho+1)^{i-i_0-1}-1) \tag{2.2}$$

By replacing r_{i_0} by its corresponding value from Eq. 2.1, the lemma holds. ∎

Lemma 2.3.7 *Let z and z' be two nodes such that $d_G(z_1, z'_1) = 1$, i.e., (z, z') is an edge of G.*

For any integer $k, \rho \geqslant 1$, the output spanner S of algorithm Spanner *satisfies*

$$d_S(z,z') \leqslant \begin{cases} (4\rho+1)^{\ell(k)}-1 & \text{if } k = 2^{\ell(k)}, \\ 2(2\rho+1)(4\rho+1)^{\ell(k)-1}+4\rho(4\rho+1)^{\ell(k-2^{\ell(k)})}-1 & \text{otherwise.} \end{cases}$$

Proof Using Lemma 2.3.4, there exists a phase $j \leqslant \ell(k)+1$ (resp. $j' \leqslant \ell(k)+1$) and a node v (resp. v') such that $v \in U_j$ (resp. $v' \in U_{j'}$), $z \in R_j(v)$ (resp. $z' \in R_{j'}(v')$) and $v \in L_j$ (resp. $v' \in L_{j'}$). We take v (resp. v') to be the first dominator of z, i.e., node in U whose region contains z, (resp. z') that fall into set L. In fact, one can see that node z (or z') can be in a sparse region at a phase and switch to a dense region at the next phase, because either

- its sparse region has been merged with a neighboring dense one (if Strategy 2 is applied),
- it is in the neighborhood of a dense region,
- or it is on a shortest path leading to a dense region.

W.l.o.g., we suppose that $j \leqslant j'$.

- *Case 1*: $k = 2^{\ell(k)}$, i.e., $i_2 = -1$ and $i_0 = \ell(k) + 1$ and. By induction and using Lemma 2.3.3, at the beginning of phase i_0, the size of the region of any node in U_{i_0} is at least $n^{(2^{i_0-1}-1)/k} = n^{(k-1)/k}$. Note that we apply STRATEGY 1 at phase i_0. Thus, every node in U_{i_0} will be in L_{i_0}.

  - *Subcase 1.1*: $j \leqslant j' < i_0$ (see Fig. 2.9). Thus, using Step 6, a BFS tree spanning

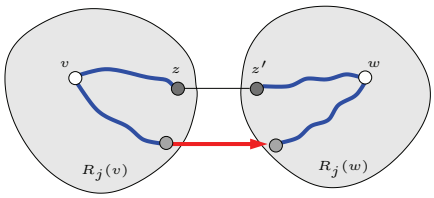


Figure 2.9: Stretch analysis for distance 1: *Subcase* 1.1

    $R_j(v)$ is added to the output spanner at phase $j-1$. In addition, one can easily show that there exists a node $w \in U_j$ such that $z' \in R_j(w)$. Hence, a BFS tree spanning $R_j(w)$ is added to the output spanner at phase $j-1$. Using Step 2 of STRATEGY$_2$, there exists an edge $e \in S$ connecting $R_j(v)$ and $R_j(w)$. Thus, $d_S(z, z') \leqslant 4r_j + 1$. Using Lemma 2.3.5, we have $d_S(z, z') \leqslant 2(4\rho+1)^{j-1} - 1$

  - *Subcase 1.2*: $j \leqslant j' = i_0$. (see Fig. 2.10). In this subcase, STRATEGY 1 is applied

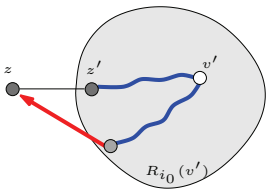


Figure 2.10: Stretch analysis for distance 1: *Subcase* 1.2

    at phase j'. Using Step 2, $R_j^+(v)$ is spanned. In addition, by Step 6, a BFS tree spanning $R_{j'}(v')$ is added to the output spanner at phase $j'-1$. Thus, because $z \in R_{j'}^+(v')$, $d_S(z, z') \leqslant 2r_{j'} + 1 = 2r_{i_0} + 1$. Using Lemma 2.3.5, we have $d_S(z, z') \leqslant (4\rho+1)^{i_0-1}$.

  Finally, because $\rho > 0$, in both subcases, the stretch is bounded by $(4\rho+1)^{\ell(k)}$.

- *Case 2*: $k \neq 2^{\ell(k)}$, i.e., $i_2 \geqslant 0$.

  - *Subcase 2.1*: $j \neq i_0$. Thus, it easy to show that there exists a node $w \in U_j$ such that $z' \in R_j(w)$. Using Step 6, $R_j(w)$ and $R_j(v)$ were spanned by a BFS tree at

phase $j-1$. In addition, because Strategy 2 is applied at phase j, an edge e connecting $R_j(v)$ and $R_j(w)$ is added at phase j (Step 2). Thus, $d_S(z,z') \leqslant 4r_j+1$.

– *Subcase* 2.2: $j = i_0$. Thus, Strategy$_1$ is applied at phase j and $R_j^+(v)$ is spanned by a BFS tree. Since $z' \in R_j^+(v)$, we have $d_S(z,z') \leqslant 2r_j + 1 = 2r_{i_0} + 1$.

Thus, the stretch is bounded by $4r_{\ell(k)+1} + 1$. Using Lemma 2.3.5, we have:

$$\begin{aligned} 4r_{\ell(k)+1} + 1 &= 4\left((4\rho+1)^{i_2} \cdot r_{i_0+1} + \tfrac{1}{2}((4\rho+1)^{i_2} - 1)\right) + 1 \\ &= 2(2\rho+1)(4\rho+1)^{i_1-1} + 4\rho(4\rho+1)^{i_2} - 1 \quad \blacksquare \end{aligned}$$

It is easy to see that the previous stretch bounds also hold for any two nodes (not necessarily at distance 1). Thus, we obtain a general bound for the stretch of the output spanner S of algorithm Spanner. However, in the next lemmas, we give a different analysis which provides improved bounds.

Lemma 2.3.8 *Let z_1 and z'_1 (resp. z_2 and z'_2) be two nodes such that $d_G(z_1,z'_1) = 2$ (resp. $d_G(z_2,z'_2) = 3$).*

For any integers $k, \rho \geqslant 1$, if $i_0 = \ell(k)+1$, i.e., $k = 2^{\ell(k)}$, then the output spanner S of algorithm Spanner *satisfies:*

$$\begin{cases} d_S(z_1,z'_1) & \leqslant & (4\rho+1)^{\ell(k)} + 1 \\ d_S(z_2,z'_2) & \leqslant & 2(4\rho+1)^{\ell(k)} + 1 \end{cases}$$

Proof In the following proof, we shall keep in mind that in the case $k = 2^{\ell(k)}$, Strategy$_1$ is applied in the last phase $\ell(k)+1 = i_0$.

First let us study the stretch for the two nodes z_1 and z'_1 satisfying $d_G(z_1,z'_1) = 2$. Let us consider a path (z_1, z, z'_1) of length 2 in G. Using Lemma 2.3.4, there exists a phase j (resp. j_1 and j'_1) and a node v (resp. v_1 and v'_1) such that $v \in U_j$ (resp. $v_1 \in U_{j_1}$ and $v'_1 \in U_{j'_1}$), $z \in R_j(v)$ (resp. $z_1 \in R_{j_1}(v_1)$ and $z'_1 \in R_{j'_1}(v'_1)$) and $v \in L_j$ (resp. $v_1 \in L_{j_1}$ and $v'_1 \in L_{j'_1}$).

We take v (resp. v_1 and v'_1) to be the first dominator of z (resp. z_1 and z'_1), i.e., node in U whose region contains z (resp. z_1 and z'_1), that fall into set L.
In the following, we analyze of all possible cases.

- *Case* 1: $j \neq i_0$, i.e., nodes z is in a light region before the last phase $\ell(k)+1 = i_0$.

  – *Subcase* 1.1: $j \leqslant j_1, j'_1$ (see Fig. 2.11).
  Let $R_j(w)$ and $R_j(w')$ the regions containing z_1 and z'_1 at phase j. Since the regions $R_j(v)$, $R_j(w)$ and $R_j(w')$ are neighbors and using Step 2 of Strategy$_2$, there exist nodes y, y_1, y' and y'_1 (respectively in regions $R_j(v)$, $R_j(w)$, $R_j(v)$ and $R_j(w')$) such that $e_1 = (y, y_1)$ is an edge in the spanner S connecting $R_j(v)$ with $R_j(w)$ and $e'_1 = (y', y'_1)$ is an edge in the spanner S connecting $R_j(v)$ with $R_j(w')$.

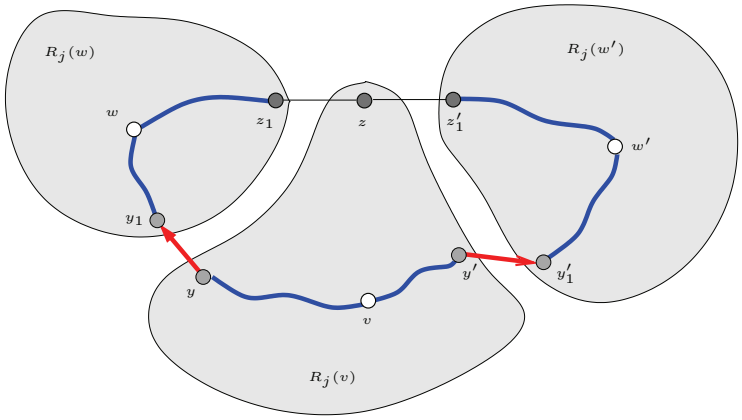


Figure 2.11: Stretch analysis for distance 2: *Subcase* 1.1

In addition, using Step 6, the regions $R_j(v)$, $R_j(w)$ and $R_j(w')$ were spanned by a BFS tree in phase $j-1$. Thus, we have:

$$\begin{aligned} d_S(z_1, z'_1) &\leqslant d_S(z_1, v_1) + d_S(v_1, y_1) + d_S(y_1, y) + d_S(y, v) + d_S(v, y') \\ &\quad + d_S(y', y'_1) + d_S(y'_1, v'_1) + d_S(v'_1, z'_1) \\ &= r_j + r_j + 1 + r_j + r_j + 1 + r_j + r_j \\ &\leqslant 6r_{i_0-1} + 2 \end{aligned}$$

– *Subcase* 1.2: $j > j_1, j'_1$. W.l.o.g. assume that $j_1 \leqslant j'_1$. (see Fig. 2.12).

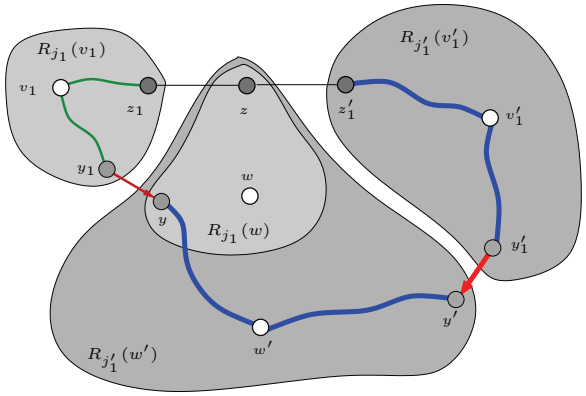


Figure 2.12: Stretch analysis for distance 2: *Subcase* 1.2

By construction, there exists a node w such that $z \in R_{j_1}(w)$, i.e., $R_{j_1}(w)$ is the region containing z in phase j_1. Since $R_{j_1}(v_1)$ and $R_{j_1}(w)$ are neighbors and using Step 2, there exist nodes $y_1 \in R_{j_1}(v_1)$ and $y \in R_{j_1}(w)$ such that (y, y_1) is an edge in the spanner S connecting $R_{j_1}(w)$ and $R_{j_1}(v_1)$. Thus, using Step 6, we have:

$$d_S(z_1, y) \leqslant 2r_{j_1} + 1 \leqslant 2r_{i_0-1}$$

Let us suppose $j'_1 > j_1$ and let us focus on the region $R_{j_1}(w)$ containing both z and y. We only apply Strategy$_2$ in phases before phase $j'_1 \leqslant j < i_0$. Thus, at each phase where Strategy$_2$ is applied, and using Steps 4 and 5, the whole region $R_{j_1}(w)$ is entirely merged with neighboring ones in order to form a new enlarged region. Then, the new enlarged region is is entirely merged with other regions and so on. Thus, nodes z and y will always belong to the same region (which is connected). In particular, there exists a region $R_{j'_1}(w')$ such that both z and y are in $R_{j'_1}(w')$. Since $R_{j'_1}(v'_1)$ and $R_{j'_1}(w')$ are neighbors, there exist nodes $y' \in R_{j'_1}(w')$ and $y'_1 \in R_{j'_1}(w')$ such that (y', y'_1) is an edge of the output spanner S connecting $R_{j'_1}(w')$ and $R_{j'_1}(w')$. Thus, we have:

$$d_S(z'_1, y') \leqslant 2r_{j'_1} + 1 \leqslant 2r_{i_0-1} + 1$$

Using Step 6, $R_{j'_1}(w')$ is spanned by a BFS tree. Thus, we get:

$$d_S(y, y') \leqslant 2r_{j'_1} \leqslant 2r_{i_0-1}$$

Thus,

$$d_S(z_1, z'_1) \quad \leqslant \quad 6r_{i_0-1} + 2$$

If $j'_1 = j_1$, then it is easy to see that the region $R_{j'_1}(v'_1)$ is connected to $R_{j_1}(w)$ using an edge in the output spanner. Hence, it is easy to find a path in S such that $d_S(z_1, z'_1) \leqslant 2r_{j_1} + 1 + 2r_{j_1} + 1 + 2r_{j_1} = 6r_{i_0-1} + 2$.

– *Subcase* 1.3: $j_1 \leqslant j \leqslant j'_1$ (see Fig. 2.13) which is symmetric to $j'_1 \leqslant j \leqslant j_1$.

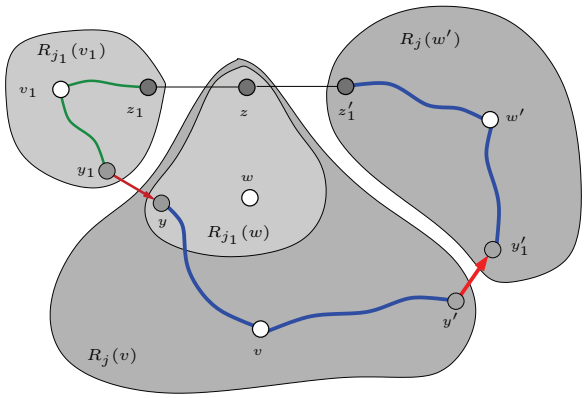


Figure 2.13: Stretch analysis for distance 2: *Subcase* 1.3

Here, the only difference with subcase 1.2 is that the region $R_j(v)$ becomes sparse before $R_{j'_1}(v'_1)$. It is straightforward from the analysis of subcase 1.2 and Fig. 2.13 that:

$$d_S(z_1, z'_1) \quad \leqslant \quad 6r_{i_0-1} + 2$$

- *Case 2*: $j = i_0$ (see Fig. 2.14). Thus, STRATEGY$_1$ is applied at phase j. It is clear by Lemma 2.3.4 and the analysis there that $R_j(v) \in L_{j=i_0}$. Thus, the neighborhood $R_j^+(v)$ of $R_j(v)$ is spanned by a BFS tree. Since, $z_1, z_1' \in R_j^+(v)$, we get the following:

$$d_S(z_1, z_1') \leqslant 2r_{i_0} + 2$$

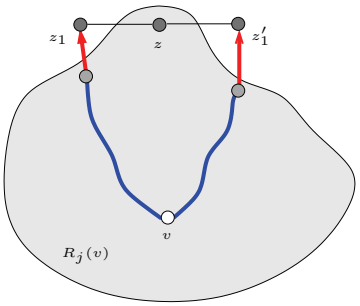


Figure 2.14: Stretch analysis for distance 2: *Case* 2

Thus, in all cases the following is true:

$$d_S(z_1, z_1') \leqslant \max\left\{ \quad 2r_{i_0} + 2 \quad , \quad 6r_{i_0-1} + 2 \quad \right\}$$

But $r_{i_0} = (4\rho + 1)r_{i_0-1} + 2\rho$. Hence,

$$\begin{aligned} d_S(z_1, z_1') &\leqslant \max\left\{ \quad 2(4\rho+1)r_{i_0-1} + 4\rho + 2 \quad , \quad 6r_{i_0-1} + 2 \quad \right\} \\ &= \qquad\qquad 2r_{i_0} + 2 \end{aligned}$$

Using Lemma 2.3.5, we have

$$\begin{aligned} d_S(z_1, z_1') &\leqslant 2 \cdot \left(\frac{1}{2} \cdot \left((4\rho+1)^{i_0-1} - 1 \right) \right) + 2 \\ &= (4\rho+1)^{\ell(k)} + 1 \end{aligned}$$

This demonstrates the lemma for the case of two nodes at distance 2.

Now, let us study the stretch for the two nodes z_2 and z_2' satisfying $d_G(z_2, z_2') = 2$. Let us consider a path (z_2, z, z', z_2') of length 3 in G. Using Lemma 2.3.4, there exists a phase j (resp. j', j_2 and j_2') and a node v (resp. v', v_2 and v_2') such that $v \in U_j$ (resp. $v' \in U_{j'}$, $v_2 \in U_{j_2}$ and $v_2' \in U_{j_2'}$), $z \in R_j(v)$ (resp. $z \in R_{j'}(v')$, $z_2 \in R_{j_2}(v_1)$ and $z_2' \in R_{j_2'}(v_2')$) and $v \in L_j$ (resp. $v' \in L_{j'}$, $v_2 \in L_{j_2}$ and $v_2' \in L_{j_2'}$). We take v (resp. v', v_2 and v_2') to be the first dominator of z (resp. z', z_2 and z_2'), i.e., node in U whose region contains z (resp. z', z_2 and z_2'), that fall into set L.

Hereafter, the analysis is more difficult because there more cases to analyze, but the general idea is the same as for two nodes at distance two.

- *Case 1*: $j \neq i_0$ and $j' \neq i_0$.
  - *Subcase 1.1*: $j_2 \leqslant j \leqslant j' \leqslant j'_2$.

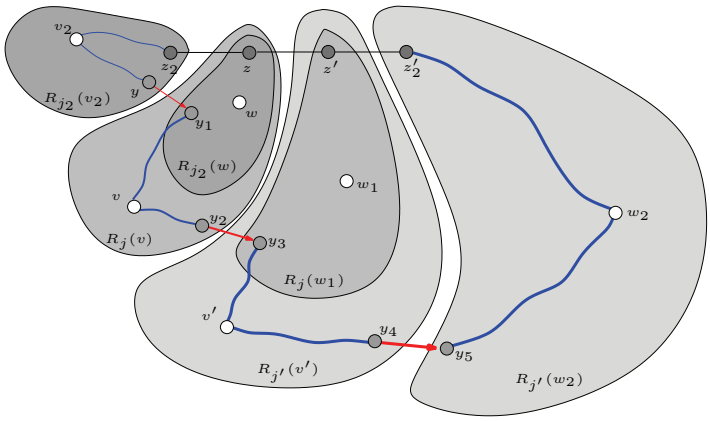


Figure 2.15: Stretch analysis for distance 3: *Subcase* 1.1

Let us first recall the following fact: when applying StrategY$_2$, a dense region is never broken, but it is entirely merged with another one. Now, using Step 2 of StrategY$_2$, one can show that there exist nodes w, w_1 and w_2, with corresponding regions $R_{j_2}(w)$, $R_j(w_1)$ and $R_{j'}(w_2)$ such that (see Fig. 2.15):

- $z \in R_{j_2}(w)$, $R_{j_2}(w) \subseteq R_j(v)$ and there exists and edge $(y, y_1) \in S$ such that $y \in R_{j_2}(v_2)$ and $y_1 \in R_{j_2}(w)$.
- $z' \in R_j(w_1)$, $R_j(w_1) \subseteq R_{j'}(v')$ and there exists and edge $(y_2, y_3) \in S$ such that $y_2 \in R_j(v)$ and $y_3 \in R_j(w_1)$.
- $z'_2 \in R_{j'}(w_2)$ and there exists and edge $(y_4, y_5) \in S$ such that $y_4 \in R_{j'}(v')$ and $y_3 \in R_{j'}(w_2)$.

Now, using the fact that each region is always spanned by a BFS tree in Step 6, it is easy to see that

$$\begin{aligned} d_S(z_2, z'_2) &\leqslant 2r_{j_2} + 1 + 2r_j + 1 + 2r_{j'} + 1 + 2r_{j'} \\ &\leqslant 8r_{i_0-1} + 3 \end{aligned}$$

  - *Subcase 1.2*: $j_2 \geqslant j$ and $j \leqslant j' \leqslant j'_2$ (see Fig. 2.16). Using the same arguments than in subcase 1.1, it is straightforward that

$$d_S(z_2, z'_2) \leqslant 8r_{i_0-1} + 3$$

  - *Subcase 1.3*: $j_2 \geqslant j$, $j \leqslant j'_2 \leqslant j'$ (see Fig. 2.17). Using the same arguments than in subcase 1.1, it is straightforward that

$$d_S(z_2, z'_2) \leqslant 8r_{i_0-1} + 3$$

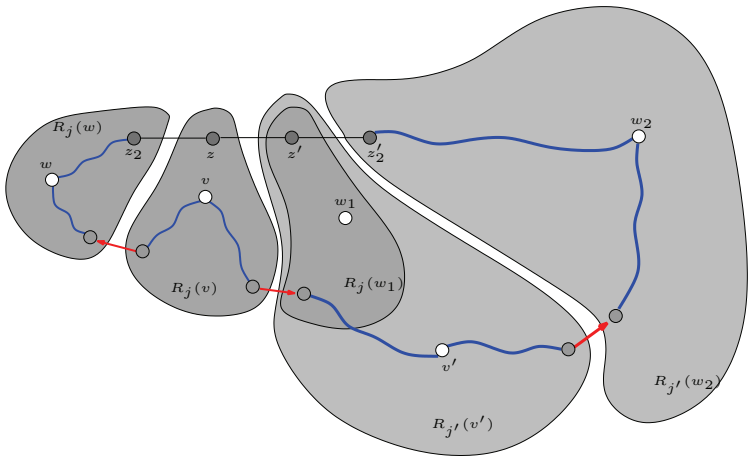


Figure 2.16: Stretch analysis for distance 3: *Subcase* 1.2

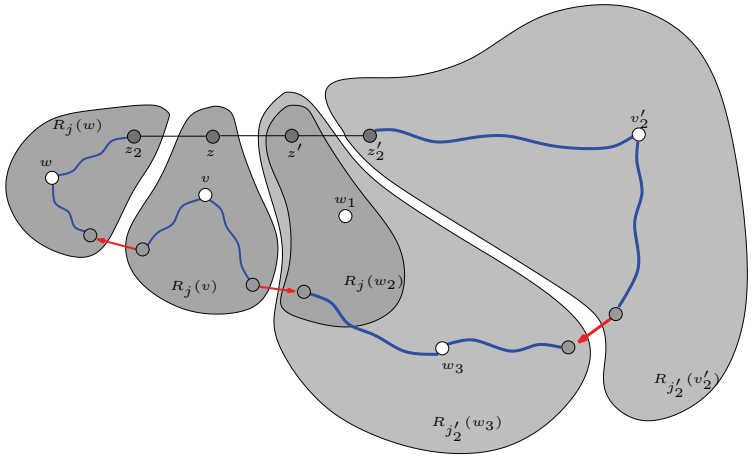


Figure 2.17: Stretch analysis for distance 3: *Subcase* 1.3

– *Subcase* 1.4: $j_2, j'_2 \leqslant j, j'$ (see Fig. 2.18). Note that the cases $j \leqslant j'$ and $j' \leqslant j$ are symmetric. In this subcase, it is also clear that we have

$$d_S(z_2, z'_2) \quad \leqslant \quad 8r_{i_0-1} + 3$$

– *Subcase* 1.5: all other subcases are perfectly symmetric to one of the three previous subcases.

- *Case* 2: $j = i_0$, i.e., STRATEGY$_1$ is applied at phase j. Thus, the neighborhood $R_j^+(v)$ is spanned by Step 6. Thus, $d_S(z_2, z') \leqslant 2r_{i_0} + 2$

  – *Subcase* 2.1: $j'_2 \neq i_0$ (see Fig. 2.19). Based on the previous analysis, it is easy to

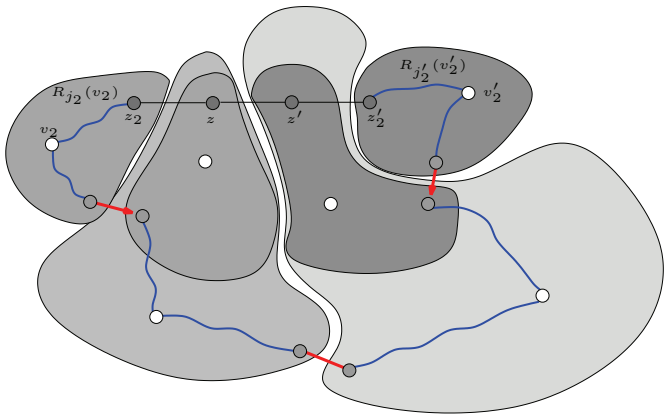


Figure 2.18: Stretch analysis for distance 3: *Subcase* 1.4

see that: $d_S(z', z'_2) \leqslant 4r_{\min\{j', j'_2\}} + 1 \leqslant 4r_{i_0-1} + 1$.
Thus, $d_S(z_2, z'_2) \leqslant 2r_{i_0} + 4r_{i_0-1} + 3$.

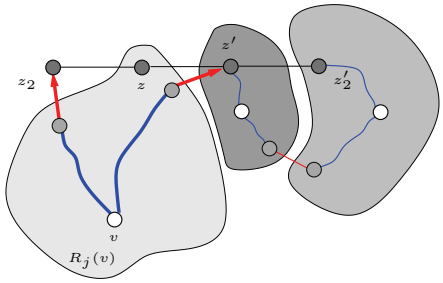


Figure 2.19: Stretch analysis for distance 3: *Subcase* 2.1

– <u>*Subcase* 2.2</u>: $j'_2 = i_0$ (see Fig. 2.20). Because in this case $R^+_{j'_2}(v'_2)$ is spanned, we have: $d_S(z', z'_2) \leqslant 2r_{i_0} + 1$. Thus,

$$d_S(z_2, z'_2) \leqslant 2r_{i_0} + 2 + 2r_{i_0} + 1 = 4r_{i_0} + 3$$

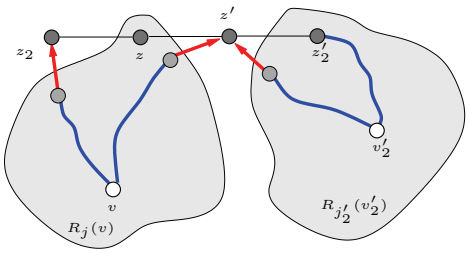


Figure 2.20: Stretch analysis for distance 3: *Subcase* 2.2

- *Case 3*: $j' = i_0$. By symmetry, we obtain the same bounds than in Case 2.

Thus, for all cases, we get the following:

$$d_S(z_2, z_2') \leqslant \max\{ \quad 2r_{i_0} + 4r_{i_0-1} + 3, \quad 4r_{i_0} + 3, \quad 8r_{i_0-1} + 3 \quad \}$$

But $r_{i_0} = (4\rho+1)r_{i_0-1} + 2\rho$. Hence,

$$\begin{array}{rcll} d_S(z_2, z_2') & \leqslant & \max\{ & 2(4\rho+1)r_{i_0-1} + 4\rho + 4r_{i_0-1} + 3, \\ & & & 4(4\rho+1)r_{i_0-1} + 8\rho + 3 \\ & & & 8r_{i_0-1} + 3 \quad \} \\ & = & & 4r_{i_0} + 3 \\ & = & & 2(4\rho+1)^{i_0-1} + 1 \end{array}$$

This completes the proof of the lemma. ∎

Lemma 2.3.9 *Let z_1 and z_1' (resp. z_2 and z_2') two nodes such that $d_G(z_1, z_1') = 2$ (resp. $d_G(z_2, z_2') = 3$).*

For any integers $k, \rho \geqslant 1$, if $i_0 < \ell(k)+1$, then the output spanner S of algorithm SPANNER *satisfies:*

$$\begin{cases} d_S(z_1, z_1') & \leqslant & 3(2\rho+1)(4\rho+1)^{\ell(k)-1} + 6\rho(4\rho+1)^{\ell(k-2^{\ell(k)})} - 1 \\ d_S(z_2, z_2') & \leqslant & 4(2\rho+1)(4\rho+1)^{\ell(k)-1} + 8\rho(4\rho+1)^{\ell(k-2^{\ell(k)})} - 1 \end{cases}$$

Proof First let us study the stretch for the two nodes z_1 and z_1' which satisfy $d_G(z_1, z_1') = 2$. Let us consider a path (z_1, z, z_1') of length 2 in G. Using Lemma 2.3.4, there exists a phase j (resp. j_1 and j_1') and a node v (resp. v_1 and v_1') such that $v \in U_j$ (resp. $v_1 \in U_{j_1}$ and $v_1' \in U_{j_1'}$), $z \in R_j(v)$ (resp. $z_1 \in R_{j_1}(v_1)$ and $z_1' \in R_{j_1'}(v_1')$) and $v \in L_j$ (resp. $v_1 \in L_{j_1}$ and $v_1' \in L_{j_1'}$). We take v (resp. v_1 and v_1') to be the first dominator of z (resp. z_1 and z_1'), i.e., node in U whose region contains z, (resp. z_1 and z_1') that fall into set L.

- *Case 1*: $j_1 \leqslant j \leqslant j_1'$. Here the critical case is when $j_1 < i_0 < j \leqslant j_1'$ (see Fig. 2.21).

  In fact, we can find two nodes $y_1 \in R_{j_1}(v_1)$ and $y_2 \in R_{j_1}(w)$ where $R_{j_1}(w)$ is the region containing z in phase j_1 such that the edge (y_1, y_2) is in the spanner. Then, the region $R_{j_1}(w)$ is entirely merged with other regions until phase i_0. At phase $i_0 < j$, it may happen that the region $R_{j_1}(w)$ becomes broken into many parts and the nodes y_1 and y_2 become in two new different regions. Thus, it may happen that $R_{j_1}(w) \not\subseteq R_j(v)$ (This is the main difference with the case $k = 2^{\ell(k)}$). Nevertheless, we can find two nodes $y_3 \in R_j(v)$ and $y_4 \in R_j(w')$, where $R_j(w')$ is the region containing z_1' at phase j, such that the edge (y_3, y_4) is in the output spanner. Thus,

$$\begin{array}{rcl} d_S(z_1, z_1') & \leqslant & 4r_{j_1} + 1 + 4r_j + 1 \\ & \leqslant & 4r_{i_0-1} + 1 + 4r_{\ell(k)+1} + 1 \end{array}$$

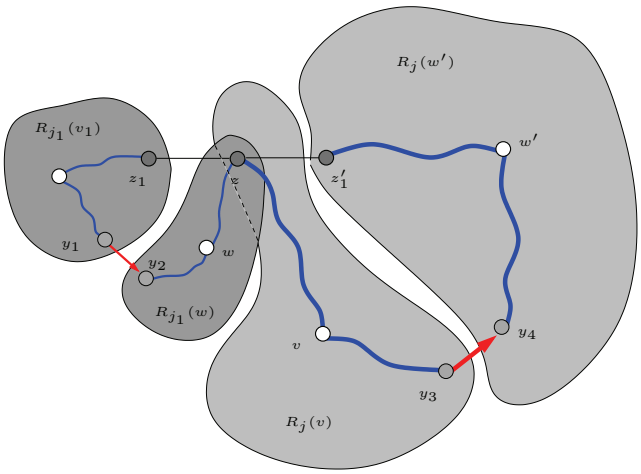


Figure 2.21: Stretch analysis for distance 2: $j_1 < i_0 < j \leqslant j_2$

In the other cases, we have essentially the same analysis as in the case $k = 2^{\ell(k)}$ and it is not difficult to show that $d_S(z_1, z'_1) \leqslant 6r_{\ell(k)+1} + 2$.

Thus, we obtain:

$$d_S(z_1, z'_1) \leqslant \max\left\{6r_{\ell(k)+1} + 2\ ,\ 4r_{i_0-1} + 4r_{\ell(k)+1} + 2\right\}$$

- *Case 2*: $j_1 \leqslant j'_1 \leqslant j$. Similarly to Case 1, the critical case here is for $j_1 < i_0 < j'_1 \leqslant j$ and we obtain the same bounds.
- *Case 3*: $j \leqslant j_1 \leqslant j'_1$. Here, it is obvious that $d_S(z_1, z'_1) \leqslant 6r_{\ell(k)+1} + 2$
- Other cases. By symmetry we obtain the same upper bounds as in previous cases.

Using Eq. 2.2 (in Lemma 2.3.6), we have:

$$\begin{aligned}
r_{\ell(k)+1} &= (4\rho+1)^{\ell(k)-i_0} \cdot r_{i_0+1} + \tfrac{1}{2}((4\rho+1)^{\ell(k)-i_0} - 1) \\
&= (4\rho+1)^{\ell(k)-i_0} \cdot ((2\rho+1)r_{i_0} + 2\rho) + \tfrac{1}{2}((4\rho+1)^{\ell(k)-i_0} - 1) \\
&= (4\rho+1)^{\ell(k)-i_0} \cdot ((2\rho+1)\left((4\rho+1)r_{i_0-1} + 2\rho\right) + 2\rho) + \tfrac{1}{2}((4\rho+1)^{\ell(k)-i_0} - 1) \\
&= (4\rho+1)^{\ell(k)-i_0} \cdot ((2\rho+1)(4\rho+1)r_{i_0-1} + 2\rho(2\rho+1) + 2\rho) + \tfrac{1}{2}((4\rho+1)^{\ell(k)-i_0} - 1)
\end{aligned}$$

Hence, we have:

$$r_{\ell(k)+1} = (2\rho+1)(4\rho+1)^{\ell(k)-i_0+1} r_{i_0-1} + 2\rho(2\rho+2+\frac{1}{2})(4\rho+1)^{\ell(k)-i_0} - \frac{1}{2} \qquad (2.3)$$

Thus, $\left(6r_{\ell(k)+1} + 2\right) - \left(4r_{i_0-1} + 4r_{\ell(k)+1} + 2\right) =$

$$\begin{aligned}
2r_{\ell(k)+1} - 4r_{i_0-1} &= \left(2(2\rho+1)(4\rho+1)^{\ell(k)-i_0+1} - 4\right) r_{i_0-1} + 4\rho(2\rho+2+\tfrac{1}{2})(4\rho+1)^{\ell(k)-i_0} - 1 \\
&\geqslant 0
\end{aligned}$$

Thus, in all the cases, we have:

$$d_S(z_1, z_1') \leqslant 6r_{\ell(k)+1} + 2$$

Using Lemma 2.3.6, we have:

$$\begin{aligned} 6r_{\ell(k)+1} + 2 &= 6\left(\frac{1}{2}(4\rho+1)^{\ell(k)-i_0}\Big((2\rho+1)(4\rho+1)^{i_0-1}+2\rho\Big) - \frac{1}{2}\right) + 2 \\ &= 3(4\rho+1)^{\ell(k)-i_0}\Big((2\rho+1)(4\rho+1)^{i_0-1}+2\rho\Big) - 1 \\ &= 3(2\rho+1)(4\rho+1)^{\ell(k)-1} + 6\rho(4\rho+1)^{\ell(k-2^{\ell(k)})} - 1 \end{aligned}$$

Thus, the lemma is true for the nodes z_1 and z_1' at distance 2 in G.

Now, let us study the stretch for the two nodes z_2 and z_2' satisfying $d_G(z_2, z_2') = 2$. Let us consider a path (z_2, z, z', z_2') of length 3 in G. Using Lemma 2.3.4, there exists a phase j (resp. j', j_2 and j_2') and a node v (resp. v', v_2 and v_2') such that $v \in U_j$ (resp. $v' \in U_{j'}$, $v_2 \in U_{j_2}$ and $v_2' \in U_{j_2'}$), $z \in R_j(v)$ (resp. $z \in R_{j'}(v')$, $z_2 \in R_{j_2}(v_1)$ and $z_2' \in R_{j_2'}(v_2')$) and $v \in L_j$ (resp. $v' \in L_{j'}$, $v_2 \in L_{j_2}$ and $v_2' \in L_{j_2'}$). We take v (resp. v', v_2 and v_2') to be the first dominator of z (resp. z', z_2 and z_2'), i.e., node in U whose region contains z (resp. z', z_2 and z_2'), that fall into set L.

There are too many cases to analyze in details. Thus, we will just give the bound obtained for each case. We do not give the details of the cases which use the same technical arguments as for the case of two nodes at distance 2. In fact, using the same ideas as previously, the reader can easily guess the path of S allowing to obtain the corresponding bound.

- Case 1: $j_2 \leqslant j \leqslant j' \leqslant j_2'$. The critical case is when $j_2 \leqslant i_0 \leqslant j$ or $j \leqslant i_0 \leqslant j'$. In fact, we have:
  - If $j_2 < i_0 \leqslant j$ (see Fig. 2.22), then $d_S(z_2, z_2') \leqslant 4r_{j_2} + 1 + 2r_j + 1 + 2r_{j'} + 1 + 2r_{j'}$. Thus, $d_S(z_2, z_2') \leqslant 4r_{i_0-1} + 6r_{\ell(k)+1} + 3$.
  - If $j < i_0 \leqslant j'$ (see Fig. 2.23), then $d_S(z_2, z_2') \leqslant 2r_{j_2}+1+2r_j+1+2r_j+2r_{j'}+1+2r_{j'}$. Thus, $d_S(z_2, z_2') \leqslant 6r_{i_0-1} + 4r_{\ell(k)+1} + 3$.
  - Otherwise, one can see that $d_S(z_2, z_2') \leqslant 8r_{\ell(k)+1}+3$. For instance, if $j_2 \leqslant j \leqslant j' \leqslant j_2' < i_0$ or if $i_0 < j_2 \leqslant j \leqslant j' \leqslant j_2'$, then we have the situation of Fig. 2.24 (which is the same than for the case k power of 2).

    All the other cases are very similar and are based on the observation that if two nodes belong to the same region at phase $i < i_0$ (resp. $i > i_0$), then in any phase i' such that $i < i' \leqslant i_0$ (resp. $i < i'$) the two nodes will still belong to a common region, i.e., a region is never broken by STRATEGY$_2$.

- Case 2: $j_2 \leqslant j \leqslant j_2' \leqslant j'$. Here the critical cases are when $j_2 < i_0 \leqslant j$ or $j < i_0 \leqslant j_2'$. And we obtain the same bounds as for Case 1.

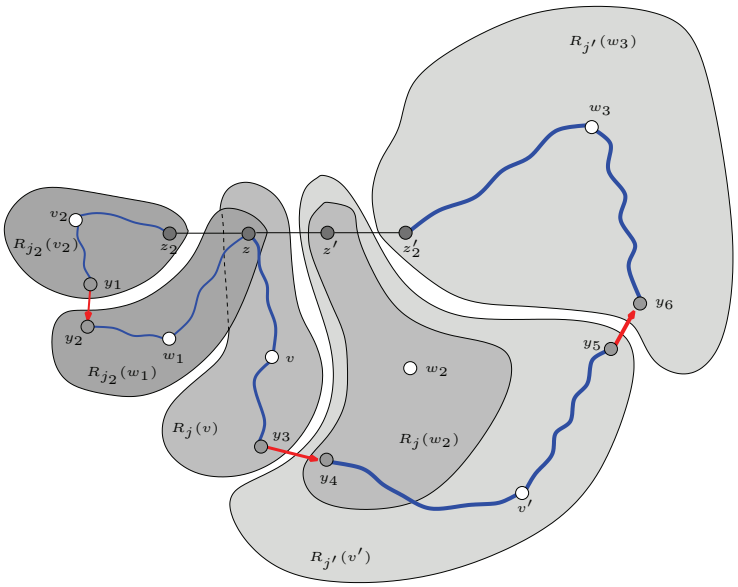


Figure 2.22: Stretch analysis for distance 3 (k not power of 2): $j_2 < i_0 \leqslant j \leqslant j' \leqslant j'_2$

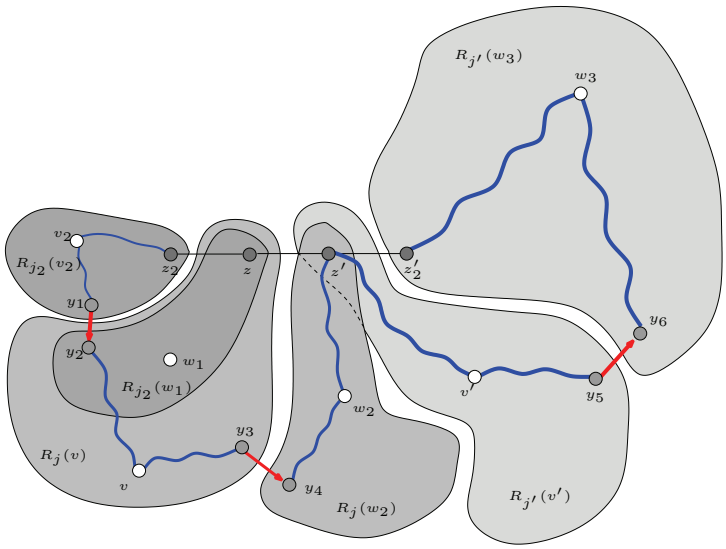


Figure 2.23: Stretch analysis for distance 3 (k not power of 2): $j_2 \leqslant j < i_0 \leqslant j' \leqslant j'_2$.

- Case 3: $j_2 \leqslant j'_2 \leqslant j' \leqslant j$. Here, the critical cases are
  - $j'_2 < i_0$ and we have the situation of Fig. 2.25. Thus, we obtain:

$$\begin{aligned} d_S(z_2, z'_2) &\leqslant 4r_{j_2} + 1 + 4r_{j'} + 1 + 4r_{j_2} + 1 \\ &\leqslant 8r_{i_0-1} + 4r_{\ell(k)+1} + 3 \end{aligned}$$

  - $j_2 < i_0 < j'_2$ and we obtain $d_S(z_2, z'_2) \leqslant 4r_{i_0-1} + 6r_{\ell(k)+1} + 3$.

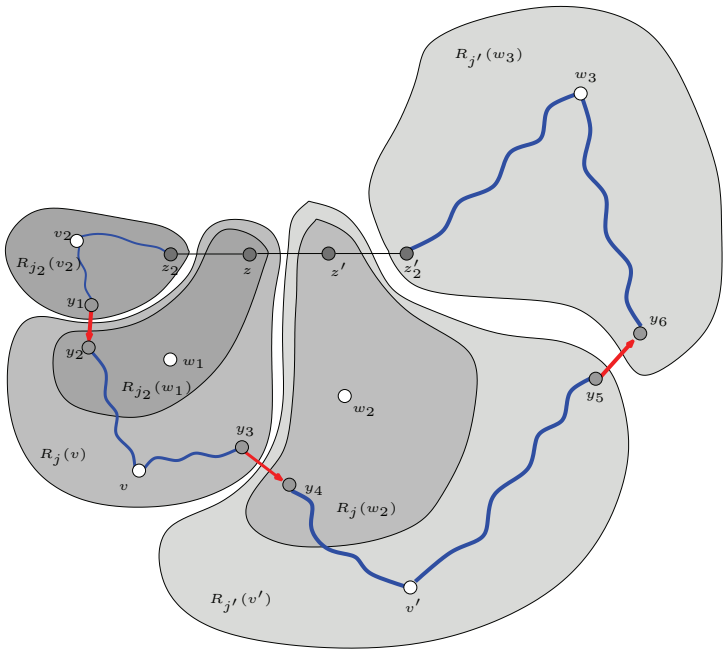


Figure 2.24: Stretch analysis for distance 3 (k not power of 2): $j_2 \leqslant j \leqslant j' \leqslant j'_2 < i_0$ or $i_0 < j_2 \leqslant j \leqslant j' \leqslant j'_2$.

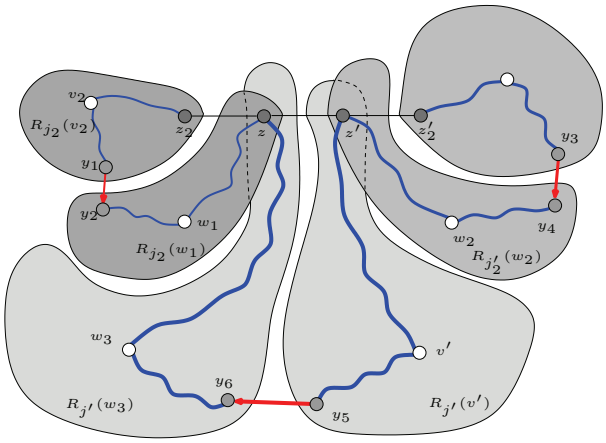


Figure 2.25: Stretch analysis for distance 3 (k not power of 2): $j_2 \leqslant j'_2 < i_0 \leqslant j' \leqslant j$.

- Case 4: $j_2 \leqslant j'_2 \leqslant j \leqslant j'$. The analysis is very similar to the one of Case 3 and we obtain the same bounds.

- Other cases: By studying all the reamining cases, we always get similar situtations than in previous cases, and it is not diffuclt to show that we obtain the same upper bounds.

In all cases, we have:

$$\begin{aligned} d_S(z_2, z_2') &\leqslant \max\Big\{ \; 8r_{\ell(k)+1}+3 \quad , \quad 6r_{i_0-1}+4r_{\ell(k)+1}+3 \quad , \\ &\qquad\quad 4r_{i_0-1}+6r_{\ell(k)+1}+3 \quad , \quad 8r_{i_0-1}+4r_{\ell(k)+1}+3 \; \Big\} \\ &= \max\Big\{ \; 8r_{\ell(k)+1}+3 \quad , \quad 4r_{i_0-1}+6r_{\ell(k)+1}+3 \quad , \\ &\qquad\quad 8r_{i_0-1}+4r_{\ell(k)+1}+3 \; \Big\} \end{aligned}$$

Using Eq. 2.3, we have $8r_{\ell(k)+1}+3-(4r_{i_0-1}+6r_{\ell(k)+1}+3) = 2r_{\ell(k)+1}-4r_{i_0-1} =$

$$(2(2\rho+1)(4\rho+1)^{\ell(k)-i_0+1}-4)r_{i_0-1}+4\rho(2\rho+2+\frac{1}{2})(4\rho+1)^{\ell(k)-i_0}-1 \geqslant 0$$

In addition, $8r_{\ell(k)+1}+3-(8r_{i_0-1}+4r_{\ell(k)+1}+3) = 4r_{\ell(k)+1}-8r_{i_0-1} =$

$$(4(2\rho+1)(4\rho+1)^{\ell(k)-i_0+1}-8)r_{i_0-1}+8\rho(2\rho+2+\frac{1}{2})(4\rho+1)^{\ell(k)-i_0}-2 \geqslant 0$$

Thus,

$$d_S(z_2, z_2') \leqslant 8r_{\ell(k)+1}+3$$

Now, using Lemma 2.3.6, we have:

$$\begin{aligned} 8r_{\ell(k)+1}+3 &= 8\left(\frac{1}{2}(4\rho+1)^{\ell(k)-i_0}\Big((2\rho+1)(4\rho+1)^{i_0-1}+2\rho\Big)-\frac{1}{2}\right)+3 \\ &= 4(4\rho+1)^{\ell(k)-i_0}\Big((2\rho+1)(4\rho+1)^{i_0-1}+2\rho\Big)-1 \\ &= 4(2\rho+1)(4\rho+1)^{\ell(k)-1}+8\rho(4\rho+1)^{\ell(k-2^{\ell(k)})}-1 \end{aligned}$$

Thus, the lemma is true for the nodes z_2 and z_2' at distance 3 in G. ∎

2.3.2 Size analysis

Lemma 2.3.10 *For any integer $k, \rho \geqslant 1$, the size of the output spanner S of algorithm* Spanner *is $O(\log k \cdot n^{1+1/k})$.*

Proof Let us fix a phase i of the algorithm. The output spanner S is updated in Steps 2 and 6 of each phase. Let us consider two consecutive phases i and $i-1$ and the edges added by Step 6 at phase $i-1$ and the edges added by Step 2 at phase i.

- *Case 1*: Strategy 1 is applied at phase i. Thus, the number of edges is bounded by:

$$\begin{aligned} \sum_{v\in L_i} |\mathrm{BFS}(v, R_i(v))| + \sum_{v\in L_i}\sum_{z\in R^+(v)} 1 \quad &< \quad n+\sum_{v\in L_i}|R^+(v)| \\ &\leqslant \quad n+\sum_{v\in L_i} n^{1/k}|R_i(v)| \\ &\leqslant \quad n+n^{1+1/k} \end{aligned}$$

- *Case 2*: STRATEGY 2 is applied at phase i. Thus, the number of edges added is bounded by:

$$\begin{aligned}\sum_{v\in L_i} |\mathrm{BFS}(v, R_i(v))| \quad + \quad \sum_{v\in L_i} |R^+_{U_i}(v)| \quad &\leqslant \quad n + n^{1/k} \sum_{v\in L_i} |R_i(v)| \\ &\leqslant \quad n + n^{1+1/k}\end{aligned}$$

Since there are $O(\log k)$ phases in the algorithm, the lemma is true. ∎

2.3.3 Distributed implementation and time complexity

In the $\mathcal{LOCAL}$ model, distributed computation of some distributed procedure A on $G^t[H]$ can be easily simulated on G as follows, charging the overall time by a factor of t. Hereafter, we assume that each node $u \in G$ can determine if it belongs or not to H. Indeed, consider one communication step in A running on some node u of $G^t[H]$ followed by one local computation step. In G, an original message in A is sent from u with a counter initialized to $t-1$ as an extra field. Now, each node $v \in G$, upon the reception of a message with some counter in its header: 1) decrements the counter; 2) stores this message if $v \in H$; and 3) forwards the incoming message with the updated counter to all its neighbors in G if the updated counter is non-null (if many messages are received during a round, then they are concatenated before being sent). After t communication rounds in G, every node $u \in H$ starts the local computation step of A on the base of all received messages during the last t communication rounds.

Similarly, given $U \subseteq V$, the computation of some distributed procedure A on G_U can be simulated on G as follows, charging the overall time by a factor $O(r)$ where r is an upper bound of the eccentricity of a node $v \in U$ in $G[R(v)]$. At each time procedure A requires for a node v of G_U to send a message to a neighbor, v broadcasts the message in $G[R(v)]$ (which is connected). The nodes at the frontier of $R(v)$, i.e., nodes having neighbors in different regions, also broadcasts the message out of their region. Symmetrically, upon the reception of messages from different regions, messages are concatenated and a convergecast is performed to v. The time overhead for one step of A is at most $2r + 1$.

Relying on the above discussions, running procedure A on $G^{2(r+1)}[H]$ or on $(G_U)^2[H]$ can be simulated on G within a factor of $O(r)$ on the time complexity.

Lemma 2.3.11 *For any integer $k, \rho \geqslant 1$,* SPANNER *can be implemented with a deterministic distributed algorithm in $O(\log k \cdot \rho^{\log k} \cdot \tau)$ time, where τ is the time complexity to compute an independent ρ-dominating set in a graph of at most n nodes.*

Proof Let us first remark that in the $\mathcal{LOCAL}$ model, we can make the nodes know their entire p-neighborhood in $O(p)$ time. Therefore, any task which requires only information about p-neighborhood can be solved within $O(p)$ time.

Let us consider a fixed phase $i \leqslant \ell(k)$. For every $v \in U$, Steps 1 and 6 of the algorithm can be implemented, using classical broadcast-convergecast, by traversing $R^+_i(v)$ a constant number of times which is $O(r_i)$ time consuming. Note that the BFS trees constructed in

Step 6 can be maintained at each phase, by adding the new nodes being merged at each phase. The time needed to run procedure IDS on $G^{2(r_i+1)}[H]$ blows up by a factor of $2(r_i+1)$ as explained before. Thus, if Strategy 1 is applied, then Step 3 is $O(r_i \cdot \tau)$ time consuming. Similarly, the time complexity of procedure IDS on $G_U[H]$ blows up by a factor $2r_i$. Thus, if Strategy 2 is applied, Step 3 is $O(r_i \cdot \tau)$ time consuming.

Steps 4 and 5 can be implemented by letting each node exploring its $O(2(r_i+1)\rho + r_i)$-neighborhood which is $O(r_i)$ time consuming for fixed ρ. Finally, Step 7 is $O(1)$ time consuming. Summing up among all steps, every phase $i \leqslant \ell(k)$ is $O(r_i \cdot \tau)$ time consuming.

At phase $\ell(k)+1$, the set H is empty and thus phase $\ell(k)+1$ is $O(r_{\ell(k)+1})$ time consuming.

Using Lemma 2.3.1 (and the radius bounds given in the proof of Lemma 2.3.7) and summing up among all phases, the time complexity of the algorithm is $O(\ell(k) \cdot (4\rho+1)^{\ell(k)} \cdot \tau)$ which completes the proof. ∎

2.4 Applications to low stretch spanners

2.4.1 Constant stretch spanners with sub-quadratic size

Let $\mathrm{MIS}(n)$ denote the time complexity for computing, by a deterministic distributed algorithm, a maximal independent set (MIS) in a graph with at most n nodes. The fastest deterministic algorithm [PS96] shows that $\mathrm{MIS}(n) \leqslant n^{O(1/\sqrt{\log n})}$. It is also known that $\mathrm{MIS}(n) \geqslant \Omega(\sqrt{\log n / \log\log n})$ [KMW04].

It is not difficult to check that a set X is an independent 1-dominating set if and only if X is a maximal independent set (cf. [Pel00, pp. 259, Ex. 4]). Thus, using the fast distributed MIS algorithm as a subroutine in algorithm Spanner, we obtain:

Theorem 2.4.1 *There is a deterministic distributed algorithm that given a graph G with n nodes and any fixed integer $k = 2^p$ with $p \geqslant 0$, constructs a spanner for G with $O(n^{1+1/k})$ edges in $O(\mathrm{MIS}(n))$ time, with the following stretch properties, $\forall u, v \in V$:*

- *If $d_G(u,v)$ is even, then: $d_S(u,v) \leqslant \frac{1}{2}\left(k^{\log 5}+1\right) d_G(u,v)$.*
- *Otherwise, $d_S(u,v) \leqslant \frac{1}{2}\left(k^{\log 5}+1\right) d_G(u,v) + \frac{1}{2}\left(k^{\log 5}-1\right)$.*

Proof Size and time are direct consequences of lemmas 2.3.3 and 2.3.11 for a fixed k and for $\rho = 1$. Note also that $\ell(k) = p = \log k$.

If $d_G(u,v) = 2d'$ for some integer $d' \geqslant 0$, then we consider a path $\mathcal{P}$ between u and v in G. The path $\mathcal{P}$ can be viewed as the sum of d' segments of length 2 each. Now, using Lemma 2.3.8, the stretch of each segment is $5^{\log k}+1 = k^{\log 5}+1$. Thus $d_S(u,v) \leqslant (k^{\log 5}+1)d'$ and the stretch bound for even distances holds.

If $d_G(u,v) = 2d'+1$ for some integer $d' > 0$, then we consider a path $\mathcal{P}$ between u and v in G. The path $\mathcal{P}$ can be viewed as the sum of $d'-1$ segments (of length 2 each), and a segment

of length 3. Now, using Lemma 2.3.8, the stretch of each even segment is $k^{\log 5}+1$ and the stretch of the odd segment is $2k^{\log 5}+1$. Thus, $d_S(u,v) \leqslant (k^{\log 5}+1)(d'-1)+2k^{\log 5}+1$ and the stretch bound for odd distances holds.

If $d_G(u,v)=1$ ($d'=0$), then using Lemma 2.3.7, the stretch bound is still good. ∎

Theorem 2.4.2 *There is a deterministic distributed algorithm that given a graph G with n nodes and any fixed integer $k=2^p+2^q-1$ with $p \geqslant q > 0$, constructs a spanner for G with $O(n^{1+1/k})$ edges in $O(\mathrm{MIS}(n))$ time, with the following stretch properties, $\forall u,v \in V$:*

- *If $d_G(u,v)=1$, then: $d_S(u,v) \leqslant 6\ 5^{p-1}+4\ 5^{q-1}-1$.*
- *If $d_G(u,v)$ is even, then: $d_S(u,v) \leqslant \frac{1}{2}\left(9\ 5^{p-1}+6\ 5^{q-1}-1\right)\ d_G(u,v)$.*
- *Otherwise, $d_S(u,v) \leqslant \frac{1}{2}\left(9\ 5^{p-1}+6\ 5^{q-1}-1\right)\ d_G(u,v) - \frac{1}{2}\left(3\ 5^{p-1}+2\ 5^{q-1}-1\right)$.*

Proof Size and time are direct consequences of lemmas 2.3.3 and 2.3.11 fixing k and $\rho=1$.

First, if $p=q$, then $k=2^{p+1}-1=\sum_{j=0}^{p}2^j$. Hence, $\ell(k)=p$ and $\ell(k-2^{\ell(k)})=\ell(2^{p+1}-1-2^p)=\ell(2^p-1)=p-1$. If $p\neq q$, then $k=2^p+\sum_{j=0}^{q-1}2^j$. Hence, $\ell(k)=p$. In addition, $\ell(k-2^{\ell(k)})=\ell(2^p+2^q-1-2^p)=\ell(2^q-1)=q-1$.

If $d_G(u,v)=1$, the stretch bound is given by Lemma 2.3.7 for $\rho=1$.

If $d_G(u,v)=2d'$ for some $d'\geqslant 0$, then by viewing the shortest path between u and v as a sum of segments of length 2 and using Lemma 2.3.9, we have $d_S(u,v)\leqslant(9\cdot5^{p-1}+6\cdot5^{q-1}-1)d'$. Hence, the stretch bound for even distance holds.

Otherwise, if $d_G(u,v)=2d'+1$ for some $d'>0$, then using Lemma 2.3.9, we have $d_S(u,v)\leqslant(9\cdot5^{p-1}+6\cdot5^{q-1}-1)(d'-1)+(12\cdot5^{p-1}+8\cdot5^{q-1}-1)$. Hence, the stretch bound for even distances also holds. ∎

Corollary 2.4.3 *For every integer k such that $k=2^p+2^q-1$, where $p\geqslant q\geqslant 0$, there is a deterministic distributed algorithm that given a graph G with n nodes, constructs a spanner for G with $O(n^{1+1/k})$ edges in $O(\mathrm{MIS}(n))$ time, such that for every $u,v\in V$, $d_S(u,v)\leqslant \alpha[k]\ d_G(u,v)+\beta[k]$ where $(\alpha[k],\beta[k])$ is given by Table 2.1.*

(p,q)	$(0,0)$	$(1,0)$	$(1,1)$	$(2,0)$	$(2,1)$	$(2,2)$	$(3,0)$	$(3,1)$
k	1	2	3	4	5	7	8	9
$2k-1$	1	3	5	7	9	13	15	17
$d_G(u,v)=1 \Rightarrow (\alpha[k],\beta[k])$	$(1,0)$	$(5,0)$	$(9,0)$	$(25,0)$	$(33,0)$	$(49,0)$	$(125,0)$	$(153,0)$
$d_G(u,v)\equiv 0[2] \Rightarrow (\alpha[k],\beta[k])$	$(1,0)$	$(3,0)$	$(7,0)$	$(13,0)$	$(25,0)$	$(37,0)$	$(63,0)$	$(115,0)$
$d_G(u,v)\equiv 1[2] \Rightarrow (\alpha[k],\beta[k])$	$(1,0)$	$(3,2)$	$(7,-2)$	$(13,12)$	$(25,-8)$	$(37,-12)$	$(63,62)$	$(115,-38)$
i_0	1	2	1	3	2	1	4	3

Table 2.1: Stretch and Strategy examples for $k=2^p+2^q-1$.

2.4.2 Graphs with large minimum degree

It is known that sparser spanners exist whenever the minimum degree increases (cf. the concluding remark of [BKMP05]). In this paragraph, we show that graphs with minimum degree large enough enjoy an $O(1)$-spanner with only $O(n)$ edges, moreover computable with a fast deterministic distributed algorithm.

Let us first note that if a graph G has a ρ-dominating set X (not necessarily independent), then G has a $(O(\rho), O(\rho))$-spanner with at most $n+|X|^2/2$ edges. Assuming we are given such a dominating set, the spanner can be constructed distributively in $O(\rho)$ time by first clustering the nodes of the graph around the nodes in the dominating set, and then by connecting every two neighboring clusters using one edge. The stretch analysis is based on the same techniques used for algorithm Spanner.

Proposition 2.4.4 *For every parameter $\rho \geqslant 1$, there exists a deterministic distributed algorithm that given a graph G with n nodes and a ρ-dominating set X, constructs a spanner S for G with at most $n + |X|^2/2$ edges in $O(\rho)$ time, such that, $\forall u, v \in V$:*

- *If $d_G(u, v) = 1$, then $d_S(u, v) \leqslant 4\rho + 1$.*
- *If $d_G(u, v)$ is even, then $d_S(u, v) \leqslant (3\rho + 1)\ d_G(u, v)$.*
- *If $d_G(u, v)$ is odd and $d_G(u, v) \neq 1$, then $d_S(u, v) \leqslant (3\rho + 1)\ d_G(u, v) - \rho$.*

This proposition can be combined with the observation that if G has minimum degree $\delta \geqslant \sqrt{n \log n}$, then G has a 1-dominating set X of size $O(\sqrt{n \log n})$. Indeed, this can be proved using the following greedy algorithm [Lov75]: one starts with $X = \varnothing$ and with the set of all radius-1 balls, $\mathcal{B} = \{N[v] \mid v \in V\}$, where $N[v] = \{u \in V \mid d_G(u, v) \leqslant 1\}$. Then, while $\mathcal{B}$ is nonempty, one selects a node $x \in V$ for X that belongs to the maximum number of balls in the current set $\mathcal{B}$. The set $\mathcal{B}$ is updated by removing all balls containing x. The constructed set X is a 1-dominating set and it can be shown that $|X| \leqslant n(1 + \ln n)/\min_{v \in V} |N[v]|$ which is at most $O(\sqrt{n \log n})$ if $\delta \geqslant \sqrt{n \log n}$. Thus, the problem is to efficiently compute such 1-dominating set.

Unfortunately, no deterministic distributed implementation of the greedy algorithm faster than $O(|X|)$ is known. A small ρ-dominating set can be computed much more efficiently in $O(\rho \log^* n)$ time by the algorithm of [KP98]. Unfortunately, its guaranteed size for X is only of $O(n/\rho)$. Finally, no algorithm is known to run in $o(\sqrt{n \log n})$ time for this problem.

However, using our algorithm, we obtain a spanner with only $O(n)$ edges, moreover with a better time complexity.

Theorem 2.4.5 *There exists a deterministic distributed algorithm that given a graph G with n nodes and minimum degree $\delta \geqslant \sqrt{n}$, constructs a spanner for G with at most $3n/2$ edges in $O(\mathrm{MIS}(n))$ time, such that, $\forall u, v \in V$:*

- *If* $d_G(u,v) = 1$*, then:* $d_S(u,v) \leqslant 9$*.*
- *If* $d_G(u,v)$ *is even, then:* $d_S(u,v) \leqslant 7\ d_G(u,v)$*.*
- *If* $d_G(u,v)$ *is odd and* $d_G(u,v) \neq 1$*, then:* $d_S(u,v) \leqslant 7\ d_G(u,v) - 2$*.*

Proof The algorithm consists in two stages. First, we construct an MIS for G^2. Then, each node of the MIS constructs its region using the coloring technique of algorithm SPANNER. The spanner is obtained by considering the edges spanning the regions and the edges connecting every two adjacent regions.

The stretch bounds are just a corollary of Proposition 2.4.4. ∎

2.4.3 Randomized distributed implementation issues

In [Lub86], Luby gives a simple and efficient randomized PRAM algorithm for computing an MIS in $O(\log n)$ expected time. Luby's algorithm can be turned to run in the distributed $\mathcal{LOCAL}$ model, and we obtain a distributed algorithm for computing an independent 1-dominating set which terminates within $O(\log n)$ expected time. We remark that upon termination of the algorithm, the constructed 1-dominating set is always correct, the randomization is only on the running time, i.e., it is a *Las Vegas* algorithm.

Thus, we obtain the following randomized version of Theorems 2.4.1 and 2.4.2:

Theorem 2.4.6 *There is a (*Las Vegas*) randomized distributed algorithm that given a graph* G *with* n *nodes and any fixed integer* $k = 2^p$ *with* $p \geqslant 0$*, constructs a spanner for* G *with* $O(n^{1+1/k})$ *edges in* $O(\log n)$ *expected time, with the same stretch properties than in Theorem 2.4.1.*

Theorem 2.4.7 *For every fixed integer* $k \geqslant 3$*, there is a (*Las Vegas*) randomized distributed algorithm that given a graph* G *with* n *nodes and any fixed integer* $k = 2^p + 2^q - 1$ *with* $p \geqslant q > 0$*, constructs a spanner for* G *with* $O(n^{1+1/k})$ *edges in* $O(\log n)$ *expected time, with the same stretch properties than in Theorem 2.4.2.*

Our (*Las Vegas*) randomized algorithms guarantee the stretch and the size bounds for the constructed spanners, while the $O(k)$ time (*Monte Carlo*) randomized algorithms [BS03] do not give any guarantee on the spanner size. This is of course achieved at the price of increasing the stretch factor of the spanner.

Recently, in [KMW06], Khun et al. show that every packing problem can be approximated by a constant factor with high probability in $O(\log n)$ time in the $\mathcal{LOCAL}$ model. Therefore, the (*Monte Carlo*) algorithm of [KMW06] implies a randomized constant approximation algorithm for the minimum 1-dominating set problem with $O(\log n)$ time. Thus, using Proposition 2.4.4, we obtain the following result (to be compared with Theorem 2.4.5 and [BS03]):

Theorem 2.4.8 *There exists a (*Monte Carlo*) randomized distributed algorithm that given a graph G with n nodes of minimum degree $\delta \geqslant \sqrt{n}$, constructs a spanner for G in $O(\log n)$ time. The size is $O(n \log^2 n)$ edges with high probability, and $\forall u, v \in V$:*

- *If $d_G(u,v) = 1$, then: $d_S(u,v) \leqslant 5$.*
- *If $d_G(u,v)$ is even, then: $d_S(u,v) \leqslant 4\ d_G(u,v)$.*
- *If $d_G(u,v)$ is odd and $d_G(u,v) \neq 1$, then: $d_S(u,v) \leqslant 4\ d_G(u,v) - 1$.*

More generally, for a minimum degree δ graph, we obtain a spanner with $O(n + (n \log n/\delta)^2)$ edges.

Let us remark that, in Theorem 2.4.8, 5 is the best possible bound on the stretch if $\delta \geqslant w(n^{1/4} \log n)$. In fact, there exist graphs with minimum degree $c\sqrt{n}$ (for some constant $c > 0$) and girth 6 (the length of its smallest cycle). Thus, the deletion of any edge implies a stretch of at least 5 for its endpoints. Therefore, any spanner with size less than $\frac{1}{2}cn\sqrt{n}$ have stretch at least 5, and $O(n + (n \log n/\delta)^2) = o(n\sqrt{n})$ if $\delta \geqslant w(n^{1/4} \log n)$.

2.5 Open questions

In this chapter we have considered deterministic distributed algorithm to construct low stretch and sparse spanners of unweighted arbitrary graphs. In particular, we have shown that $(3,2)$-spanner with $O(n^{3/2})$ edges can be constructed in $n^{O(1/\sqrt{\log n})}$ time. Let us observe that $\log n < n^{1/\sqrt{\log n}}$ only for $n > 2^{4^2}$. In other words, deterministic distributed $n^{1/\sqrt{\log n}}$ time algorithms might be competitive[1] over randomized $\log n$ time algorithms for distributed system up to $n \leqslant 32656$ processors. We left open the two following problems:

1. Reduce the stretch from $(3,2)$ to optimal stretch 3, without increasing the size of the spanner and the running time. More generally, is it possible, for every $k \geqslant 1$, to compute with a deterministic distributed algorithm a $(2k-1)$-spanners of size $O(n^{1+1/k})$ in $O(\text{MIS}(n))$ time?
2. Reduce the time complexity to $o(\text{MIS}(n))$, possibly with some small stretch and size increasings. More precisely, is it possible to compute with a deterministic distributed algorithm a constant stretch spanner with $o(n^2)$ edges in $o(\text{MIS}(n))$ time? Using our approach, it suffices to show that there is a constant ρ for which an independent ρ-dominating set can be computed in $o(\text{MIS}(n))$ time for every graph.

[1] This obviously depends on the constants hidden in the O-notation.

Part II

A Formal Approach in Distributed Computing: Relabeling Systems and Local Computations

Chapter 3

Relabeling Systems: a Formal Tool-Box for Distributed Algorithms

Abstract.

Graph traversals are fundamental for many distributed algorithms. In this chapter, we use graph relabelling systems to encode two basic graph traversals which are the broadcast and the convergecast. We obtain formal, modular and simple encoding for many sophisticated distributed algorithms, such as the layered breadth-first spanning tree algorithm, and the minimum spanning tree algorithm.

The method and the formalism we use in this chapter allow to focus on the correctness of a distributed algorithm rather than on the implementation and the communication details in the network.

Résumé.

Les parcours de graphes sont la bases de la plus part des algorithmes distribués. Dans ce chapitre, nous utilisons les systèmes de réétiquetage de graphes pour coder deux techniques de bases: la dissémation et la collecte d'information. Ceci nous permet d'obtenir une description formelle, modulaire et simple de plusieurs algorithmes distribués sophistiqués, tels que, le calcul d'un arbre recouvrant de plus courts chemins, et le calcul d'un arbre recouvrant de poids minimum.

La méthode ainsi que le formalisme que nous utilisons dans ce chapitre permettent de se concentrer sur la correction d'un algorithme distribuè plutôt que sur les aspects d'implémentation ou de communications entre les différentes entités du réseau.

3.1 Introduction

One of the most important challenges in distributed computing is to design rigorous distributed algorithms and to prove them in a precise and formal way. In general, due to the interaction between the process variables and the communication procedures, it is hard to design a distributed algorithm in such a way it can be understood, implemented and proved easily. In addition, many algorithms depend strongly on the distributed model, i.e., message-passing, shared memory, synchronous, asynchronous etc. A distributed algorithm which is designed in a given model can not always work in another model. Even though it is possible, one has often to re-adapt or to re-encode the algorithm depending on the model assumptions.

Generally speaking, the purpose of this chapter is to give a general methodology in order to help designers to encode distributed algorithms. More precisely, in this work we try to understand and to answer the following general problem:

Q3:	How can we design distributed algorithms in a *unified*, *formal* and *comprehensive* way?

In order to achieve this goal, we use graph relabeling systems and local computations [LMS95, GMM04] as a tool-box. A graph relabeling system is based on a set of relabeling rules which are executed locally and concurrently on balls of fixed radius. These rules are described using mathematical and logic tools which enables to give a rigorous mathematical formalization of a distributed algorithm.

The starting point of our work is the following simple observation: *"many distributed algorithms appear as the compositions of some basic network traversals"*. These basic traversals are generally based on the broadcast (or propagation) of information and the convergecast (or echo) of information [Cha82, Tel00, Lyn96, Seg83]. Our goal is to show how to encode and to combine the broadcast and the convergecast techniques in order to obtain a formal encoding of a large class of *wave and traversal* algorithms. Our approach is purely theoretical and provides many theoretical examples for encoding in a formal way the main classical techniques used in distributed computing. We do not care about the practical implementation issues of relabeling systems or the efficiency issues of such implementations. We just focus on the expressiveness of relabeling systems for distributed algorithms from a theoretical point of view.

The rest of this chapter is organized as follows:

- In Section 3.2, we first give an intuitive definition of relabeling systems.
- In Section 3.3, we give an encoding of the broadcast and convergecast techniques using graph relabeling systems. These two techniques are the basis of all the other algorithms given in this chapter.

- In Section 3.4, we show how to encode the classical layered BFS tree algorithm using relabeling systems. We also give a general method to encode distributed algorithms for computing global functions having particular properties (commutativity and associativity). These two applications illustrate how to encode distributed algorithms based on the more general technique of *"Propagation of Information with Feedback"* (PIF for short) [Seg83].

- In Section 3.5, we use a combination of our basic traversal algorithms in order to derive the graph relabeling system encoding the classical Prim's distributed algorithm for computing a minimum spanning tree (MST for short) [Tel00, Pri57]. This example shows how to combine the basic graph traversal techniques in order to obtain a simple and formal encoding of more sophisticated algorithms.

3.2 Model and notations

In this section, we give a practical definition of relabeling systems and we explain the conventions under which we will describe then later. A detailed description of the mathematical foundations of graph relabeling systems and local computations can be found in [GMM04, LMS99, CM05].

A network is represented as a connected, undirected graph $G = (V, E)$ where nodes denote processors and edges denote direct communication links. Labels (or states) are attached to nodes and edges. For every two nodes $u, v \in V$, we denote by $d_G(u, v)$ the unweighted distance between u and v in G, i.e., the number of hops from u to v. For every integer $\ell \geqslant 0$, and for every node $v \in V$, the ball $B_G(v, \ell)$ of center v and radius ℓ is a subgraph of G, where the nodes of $B_G(v, \ell)$ are defined by the set $V(B_G(v, \ell)) = \{u \in V \mid d_G(u, v) \leqslant \ell\}$ and the edges of $B_G(v, \ell)$ are defined by the set $E(B_G(v, \ell)) = \{(u, w) \in E \mid d_G(v, u) \leqslant \ell, d(v, w) \leqslant \ell\}$. When the graph G is clear from the context, the ball of center v and radius ℓ is denoted by $B(v, \ell)$. We also write $u \in B(v, \ell)$ to denote a node u of the ball $B(v, \ell)$.

Throughout the chapter we will consider graphs where nodes and edges are labeled with labels from a set $\mathcal{L}$. A graph labeled over $\mathcal{L}$ is denoted by (G, λ), where G is the graph and $\lambda \colon V \cup E \to \mathcal{L}$ is the labeling function.

Intuitively, a distributed algorithm can be encoded by means of local relabeling rules which modify the labels of nodes and edges. The rules satisfy the following constraints, that arise naturally when describing distributed computations with decentralized control:

($C1$) they *do not change the underlying graph but only the labeling* of its components (node and/or edge), the final labeling being the result of the computation,

($C2$) they are *local*, that is, each relabeling step changes only a connected subgraph of a fixed size in the underlying graph,

($C3$) they are *locally generated*, that is, the applicability of a relabeling rule only depends on the *local context* of the relabeled subgraph.

More formally, a relabeling system is a triple $\mathcal{R} = (\mathcal{L}, \mathcal{I}, \mathcal{P})$ where:

- $\mathcal{L} = \mathcal{L}_v \cup \mathcal{L}_e$: where $\mathcal{L}_v$ is a finite set of node labels, and $\mathcal{L}_e$ is a finite set of edge labels,
- $\mathcal{I} \subseteq \mathcal{L}$: a finite set of initial labels verifying some properties,
- $\mathcal{P}$: a finite set of relabeling rules.

For each relabeling rule, we consider a generic node v_0 and the corresponding generic ball $B(v_0, \ell)$ of center v_0 and radius ℓ for some integer $\ell \geqslant 0$. Within these conventions, a relabeling rule $R \in \mathcal{P}$ is given by:

1. a precondition on the labels of the nodes and edges of $B(v_0, \ell)$,
2. a relabeling of the nodes and edges of $B(v_0, \ell)$.

Since a relabeling rule R is entirely defined by the precondition and the relabeling on the generic ball $B(v_0, \ell)$, R is called *ℓ-local*. A relabeling system $\mathcal{R} = (\mathcal{L}, \mathcal{I}, \mathcal{P})$ is called *ℓ-local*, if for every relabeling rule $R \in \mathcal{P}$ there exists an integer $\ell' \leqslant \ell$ such that R is *ℓ'-local*.

Let us consider a relabeling rule $R \in \mathcal{P}$ and a generic ball $B(v_0, \ell)$. Let v be a node of the generic ball $B(v_0, \ell)$. We denote by $\lambda(v)$ the label of v in the precondition, and by $\lambda'(v)$ the label of v in the relabeling (the new label of v). The label of a node can be composed of c components (with c a given integer). In this case, we denote by $\mathcal{L}_v = \big(L_1 \times L_2 \times ... \times L_c\big)$ the set of labels where L_i $(1 \leqslant i \leqslant c)$ is the set of possible values of the i^{th} component. We denote by $\lambda(v).L_i$ the value of the i^{th} component of $\lambda(v)$. We use the same notations for edge labels. For instance, $\lambda(v_0, v)$ denotes the labels of the edge $e = (v_0, v)$ in the precondition and $\lambda'(v_0, v)$ denotes its label in the relabeling. The precondition and the relabeling are encoded as logic formulas. We use the logic symbols $\wedge$, $\vee$, $\exists$, $\exists!$ and $\forall$ to denote respectively the logic operators *"and"*, *"or"*, *"there exist"*, *"there exists a unique"* and *"for all"*. In the case of a weighted graph, if $e = (u, v)$ is an edge then we denote by $\mathcal{W}(u, v)$ the weight of e.

A *relabeling step* corresponds to an *application* of a relabeling rule. An application of a relabeling rule R consists in:

1. finding a ball in the graph G where the precondition of rule R is verified, i.e., finding an *occurrence* of rule R in the graph G,
2. applying the relabeling corresponding to rule R to the ball.

Local computations can then be defined by the computations corresponding to the relabeling steps. A *sequential view* of local computations consists in applying the relabeling rules sequentially, i.e., a relabeling step at once. In opposite, since many relabeling rules can

be applied in parallel if the corresponding balls do not overlap, a *distributed view* of local computations consists in applying some rules in parallel on a non-overlapping set of balls.

An *execution* of a *distributed algorithm* in our formalism corresponds to a *sequence* of relabeling steps.

The three basic local computation models

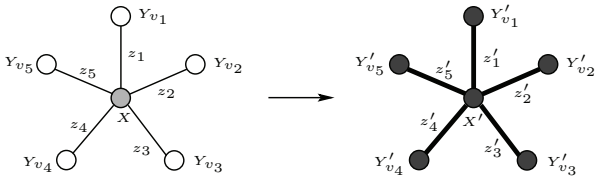
For the special case of 1-local relabeling systems, one can find three particular types of local computation models:

1. *The Closed Star (CS) model*: in this model, the labels attached to the nodes and the edges of a star are allowed to be relabeled according to their previous values.

   In particular, if $Pred(v)$ (resp. $Pred(u,v)$) denotes a predicate using the label of node u (resp. edge (u,v)), then the general shape of a relabeling rule in the CS model is as follows:

R_{cs} : *Precondition :*
   - $\lambda(v_0) = X$
   - $\forall v \in B(v_0,1),\ \lambda(v) = Y_v$
   - $\forall v \in B(v_0,1),\ \lambda(v_0,v) = z_v$

   Relabeling :
   - $\lambda'(v_0) := X'$
   - $\forall v \in B(v_0,1)\ (Pred(v) \Longrightarrow \lambda'(v) := Y'_v)$
   - $\forall v \in B(v_0,1)\ (Pred(v_0,v) \Longrightarrow \lambda'(v_0,v) := z'_v)$

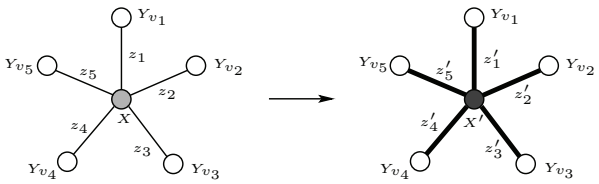


2. *The Open Star (OS) model*: in this model, only the labels attached to the center and the edges of a star are allowed to be relabeled according to the previous values of the labels of the star.

   In particular, if $Pred(u,v)$ denotes a predicate using the label of an edge (u,v), then the general shape of a relabeling rule in the OS model is as follows:

R_e : *Precondition :*
   - $\lambda(v_0) = X$
   - $\forall v \in B(v_0,1),\ \lambda(v) = Y_v$
   - $\forall v \in B(v_0,1),\ \lambda(v_0,v) = z_v$

   Relabeling :
   - $\lambda'(v_0) := X'$
   - $\forall v \in B(v_0,1)\ (Pred(v_0,v) \Longrightarrow \lambda'(v_0,v) := z'_v)$



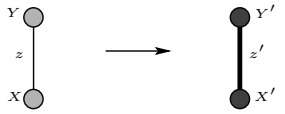
3. *The Edge model*: in this model, only the labels attached to an edge are allowed to be relabeled. The general shape of a relabeling rule in the edge local computation model can be written as follows:

R_{os} : *Precondition :*

- $\lambda(v_0) = X$
- $\lambda(v) = Y$
- $\lambda(v_0, v) = z$

Relabeling :

- $\lambda'(v_0) = X'$
- $\lambda'(v) = Y'$
- $\lambda'(v_0, v) = z'$



The previous basic local computation models are widely studied. In particular, many works are devoted to study and to compare their computation power. For instance, it is clear that a relabeling system in the OS model trivially holds for the CS model, whereas it was proved that the CS model is strictly more powerful than the OS model. In this work we are not interested in the computation power of relabeling systems. Our goal is to show how to use them in order to design formal and unified distributed algorithms. Therefore, we choose to use the CS model as the default model for all our relabeling systems. However, it is important to remark that all the techniques presented in this chapter can be adapted in less powerful local computation models.

Remark 3.2.1 *The three basic local computation models can be implemented in practical distributed systems, i.e., message passing/mobile agents. The idea is to run many rounds where a set of disjoint stars (or edges) are distributively elected in each round. This set defines the set of regions where a relabeling can be applied. As discussed before, we are not interested in the implementation issues in this chapter. More details are given in Chapter 5.*

3.3 Basic building blocks

Many basic distributed algorithms can be described as a combination of many couples of broadcast and convergecast [Tel00, Pel00, Lyn96]:

- The broadcast is usually used to deliver a given information, e.g., the value of a variable, the beginning of a new computation step, to all the nodes of the network.
- The convergecast is in general used to collect some information into one single node. This information is for example used to start some new computation step.

In the next two subsections, we give the relabeling systems corresponding to the broadcast and convergecast operations. We do not care about the information to be broadcasted or collected. We only give the intuitive method to encode these two basic techniques using relabeling systems.

3.3.1 The broadcast technique

The broadcast operation can be defined as *"the dissemination of some information from a source node to all nodes in the network"* [Pel00]. It can be encoded with the relabeling system $\mathcal{R}_b = (\mathcal{L}_b, \mathcal{I}_b, \mathcal{P}_b)$ defined by:

- $\mathcal{L}_b = \mathcal{E} \cup \{0,1\}$ where $\mathcal{E} = \{A, S, O\}$, i.e., a node can be labled A or S or O, and an edge can be labeled 0 or 1.
- $\mathcal{I}_b = \{A, O\} \cup \{0\}$. Initially, one source node is labeled A, all other nodes are labeled O and all the edges are labeled 0.
- $\mathcal{P}_b = \{R_b^1, R_b^2\}$ where the two rules R_b^1 and R_b^2 are described below.

In the following rules, the label S encodes the fact that the broadcast process has reached some node.

R_b^1 : **Broadcast : initial step**

Precondition :

- $\lambda(v_0).\mathcal{E} = A$

Relabeling :

- $\forall v \in B(v_0, 1)\ \Big(\lambda(v).\mathcal{E} = O \Longrightarrow \Big(\lambda'(v) := S, \lambda'(v, v_0) := 1\Big)\Big)$

R_b^2 : **Broadcast**

Precondition :

- $\lambda(v_0).\mathcal{E} = S$

Relabeling :

- $\forall v \in B(v_0, 1)\ \Big(\lambda(v).\mathcal{E} = O \Longrightarrow \Big(\lambda'(v) := S, \lambda'(v, v_0) := 1\Big)\Big)$

In practical, rule R_b^1 (resp. R_b^2) can be applied as follows. If a star center v_0 is labeled A (resp. S) then for each node v in the star $B(v_0, 1)$, if v is labeled O then it becomes labeled S and the edge (v_0, v) becomes labeled 1.

As a basic application of the relabeling system $\mathcal{R}_b$, once the broadcast is terminated, we obtain a spanning tree by considering the edges with labels 1. Fig. 3.1 shows an example of the broadcast algorithm using the relabeling system $\mathcal{R}_b$. Notice that the rules in our example are applied by many nodes belonging to disjoint stars.

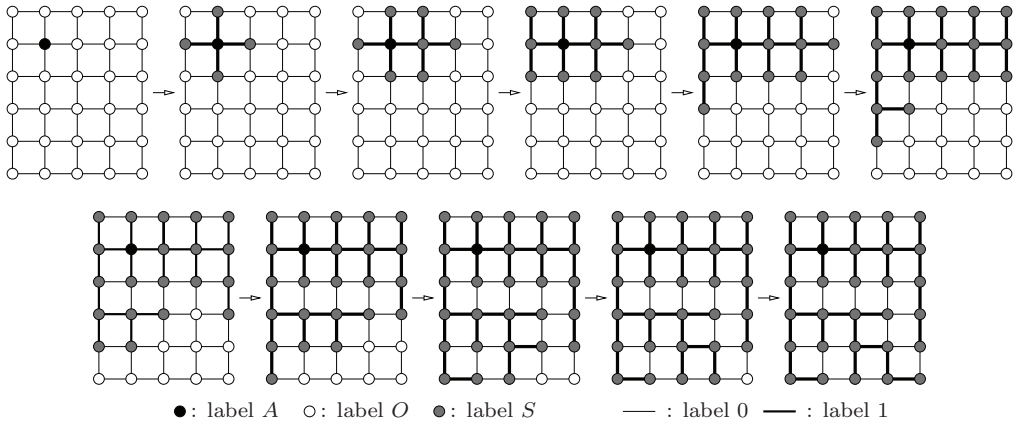


Figure 3.1: An example of a broadcast using the relabeling system $\mathcal{R}_b$.

3.3.2 The convergecast technique

The convergecast operation consists in *"collecting information upwards on a rooted spanning tree T"* [Pel00]. The most fundamental example is to let a source node which broadcasted some information, detect that the information has reached all the nodes. In order to detect the "broadcast termination", a convergecast process can be performed as follows. First, each leaf of the tree T which is reached by the broadcast sends an acknowledgment to its parent. Upon receipt of an acknowledgment from all its children, a node sends an acknowledgment to its parent and so on. When the source node receives an acknowledgment from all its children then the source node knows that the broadcast has reached all the nodes of the graph. This example can be generalized when we want to collect some other information in some root node.

We assume that we have a precomputed rooted spanning tree (recall that the relabeling system $\mathcal{R}_b$ enables us to construct such a tree). Then, the convergecast operation can be encoded using the relabeling system $\mathcal{R}_c = (\mathcal{L}_c, \mathcal{I}_c, \mathcal{P}_c)$ defined by:

- $\mathcal{L}_c = \mathcal{E} \cup \{0,1\}$ where $\mathcal{E} = \{A, S, F, T\}$, i.e., $\mathcal{E}$ is the set of node labels and $\{0,1\}$ is the set of edge labels.
- $\mathcal{I}_c = \{A, S\} \cup \{0,1\}$. Initially, the source node is labeled A, all other nodes are labeled S, an edge belonging to the spanning tree is labeled 1 and all other edges are labeled 0.
- $\mathcal{P}_c = \{R_c^1, R_c^2\}$, where the two rules R_c^1 and R_c^2 are described below.

Notice that in the following rules, if a node becomes labeled F then it has finished the convergecast. Once the source node A becomes labeled T then the convergecast process is terminated.

R_c^1 : **A node becomes a leaf**

Precondition :

- $\lambda(v_0).\mathcal{E} = S$
- $\exists!\, v_1 \in B(v_0, 1)\ \Big(\Big(\lambda(v_1).\mathcal{E} = S \vee \lambda(v_1).\mathcal{E} = A\Big) \wedge \lambda(v_0, v_1) = 1\Big)$

Relabeling :

- $\lambda'(v_0).\mathcal{E} := F$

R_c^2 : **Termination detection**

Precondition :

- $\lambda(v_0).E = A$
- $\forall v \in B(v_0, 1)\ \Big(\lambda(v).\mathcal{E} = F\Big)$

Relabeling :

- $\lambda'(v_0).\mathcal{E} := T$

In Fig. 3.2, we show an example of the execution of the convergecast algorithm using the relabeling system $\mathcal{R}_c$.

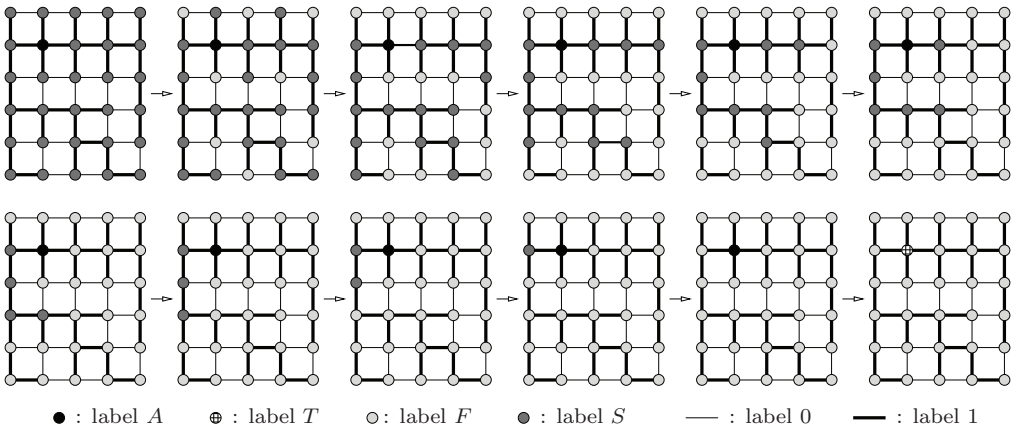


Figure 3.2: An example of a convergecast using the relabeling system $\mathcal{R}_c$.

3.4 The PIF technique: two basic applications

3.4.1 Layered BFS tree construction

Many distributed algorithms can be implemented by performing many phases of broadcast and convergecast. Each phase corresponds to a propagation of some information with feedback (PIF). The broadcast can be viewed as the beginning of a new stage of the algorithm and the convergecast corresponds to the termination of that stage. This technique is fundamental when designing distributed algorithms because, in the distributed setting, there is no centralized entity which supervises in a global way the execution of an algorithm. In the following, our main goal is to show how to encode by graph relabeling systems the PIF operation by using the basic example of the Dijkstra's layered BFS tree [CLR90, Tel00, Pel00] algorithm.

We recall that a BFS tree of a graph G with respect to a root node r is a spanning tree such that for every node v, the path leading from the root r to v in the tree has the minimum (unweighted) possible length. A classical algorithm to construct a BFS tree is to grow the tree from the root in a layered fashion. The algorithm works in many phases. At each phase, a new layer is explored and added to the tree. The main difficulty here is to start adding a new layer only when the previous layer has been completely added.

The classical layered BFS tree construction can be encoded using the graph relabeling system $\mathcal{R}_t = (\mathcal{L}_t, \mathcal{I}_t, \mathcal{P}_t)$ defined by:

- $\mathcal{L}_t = \big(\mathcal{E} \times i\big) \cup \{0, 1\}$, where $\mathcal{E} = \{A, S, F, T, O\}$ and $i = \{-1, 1\}$,
- $\mathcal{I}_t = \{(O, -1), (A, -1)\} \cup \{0\}$. Initially, a pre-distinguished node (the root) is labeled $(A, -1)$ i.e., active. All other nodes are labeled $(O, -1)$, i.e., outside the BFS tree. All edges are labeled 0.
- $\mathcal{P}_t = \{R_t^1, R_t^2, R_t^3, R_t^4, R_t^5, R_t^6, R_t^7\}$ where the seven rules are explained below.

In the remainder, by BFS tree, we mean the fragment which is being enlarged. If an edge becomes labeled 1 then it is part of the tree. The A-labeled node is the initiator of a new phase of the algorithm, i.e., the construction of a new layer. If a node is labeled $(S, 1)$, then the node must broadcast some information, i.e., the construction of a new layer. In contrast, if a node is labeled $(S, -1)$ then it waits for the acknowledgment of its children. If a node is labeled F, then the node has finished the convergecast and it is waiting for an order from its parent. Finally, if a node is labeled T, then the node has locally terminated, i.e., it can not contribute any more in the tree construction.

Rule R_t^1 starts the computation of the BFS tree by adding the first layer. It also encodes the beginning of a new phase of the algorithm. In fact, once the neighbors of the node with labeled A become labeled F (or T), the A-node knows that a new layer was added and the construction of a new layer can start. Thus, the labels of all F-neighbors are set to $(S, 1)$ in order to begin the broadcast up to the leaves.

R_t^1 : **Beginning the construction of a new layer**

Precondition :

- $\lambda(v_0).\mathcal{E} = A$ `/*the root node*/`
- $\forall v \in B(v_0, 1)\ \Big(\lambda(v).\mathcal{E} \neq S\Big)$ `/*broadcast-convergecast finished*/`
- $\exists v_1 \in B(v_0, 1)\ \Big(\lambda(v_1).\mathcal{E} \neq T\Big)$ `/*the computation is not over*/`

Relabeling :

- $\forall v \in B(v_0, 1)\ \Big(\lambda(v).\mathcal{E} \neq T \Longrightarrow \Big(\lambda'(v) := (S, 1) \wedge \lambda'(v_0, v) := 1\Big)\Big)$

If an $(S, 1)$-labeled node u is in the interior of the BFS tree, then it just informs its children by setting their labels to $(S, 1)$ and it becomes $(S, -1)$ labeled (Rule R_t^2). Otherwise, if u is a leaf, then either u adds the neighbors with label O to the tree, and it becomes F-labeled (Rule R_t^3), or there are no new nodes to add and u becomes T-labeled, i.e., terminated state (Rule R_t^4).

R_t^2 : **Broadcast**

Precondition :

- $\lambda(v_0) = (S, 1)$ `/*broadcast in progress*/`
- $\exists v_1 \in B(v_0, 1)\ \Big(\lambda(v_0, v_1) = 1 \wedge \lambda(v_1).\mathcal{E} = F\Big)$ `/*children are not informed*/`

Relabeling :

- $\lambda'(v_0) := (S, -1)$
- $\forall v \in B(v_0, 1)\ \Big(\Big(\lambda(v).\mathcal{E} = F \wedge \lambda(v_0, v) = 1\Big) \Longrightarrow \lambda'(v) := (S, 1)\Big)$

R_t^3 : **Construction of the next layer**

Precondition :

- $\lambda(v_0) = (S, 1)$
- $\exists v_1 \in B(v_0, 1)\ \Big(\lambda(v_1).\mathcal{E} = O\Big)$ `/*some neighbors are not in the tree*/`

Relabeling :

- $\lambda'(v_0) := (F, -1)$
- $\forall v \in B(v_0, 1)\ \Big(\lambda(v).\mathcal{E} := O \Longrightarrow \Big(\lambda(v) := (F, -1) \wedge \lambda'(v_0, v) := 1\Big)\Big)$

R_t^4 : **No nodes to add to the next layer**

Precondition :

- $\lambda(v_0) = (S, 1)$
- $\forall v \in B(v_0, 1)\ \Big(\lambda(v).\mathcal{E} \neq O\Big)$ `/*all neighbors are in the tree*/`
- $\exists!\, v_1 \in B(v_0, 1)\ \Big(\lambda(v_0, v_1) = 1\Big)$ `/*`v_0 `is a leaf*/`

Relabeling :

- $\lambda'(v_0) := (T, -1)$

After the broadcast step in Rule R_t^2, a node u which becomes $(S, -1)$-labeled waits for the acknowledgment of its children. When these children become F-labeled, then u knows that a new layer has been added by the leaves of the subtree rooted at it. Thus, it becomes $(F, -1)$-labeled in order to inform its parent (Rule R_t^5). Note that if all the children of u become T-labeled, then u knows that no new nodes can be added in the subtree rooted at it. Thus, it becomes T-labeled (Rule R_t^6).

R_t^5 : **Convergecast: waiting for the next broadcast**

Precondition :

- $\lambda(v_0) = (S, -1)$
- $\exists!\, v_1 \in B(v_0, 1)\ \Big(\Big(\lambda(v_1).\mathcal{E} = S \vee \lambda(v_1).\mathcal{E} = A\Big) \wedge \lambda(v_0, v_1) = 1\Big)$
  `/*all children have received an acknowledgment*/`
- $\exists v_2 \in B(v_0, 1)\ \Big(\lambda(v_2).\mathcal{E} = F \wedge \lambda(v_0, v_2) = 1\Big)$
  `/*some children have not yet terminated the algorithm*/`

Relabeling :

- $\lambda'(v_0) := (F, -1)$

R_t^6 : **Convergecast: the tree construction is locally finished**

Precondition :

- $\lambda(v_0) = (S, -1)$
- $\exists!\, v_1 \in B(v_0, 1)\ \Big(\Big(\lambda(v_1).\mathcal{E} = S \vee \lambda(v_1).\mathcal{E} = A\Big) \wedge \lambda(v_0, v_1) = 1\Big)$
- $\forall v \in B(v_0, 1)\ \Big(\Big(v \neq v_1 \wedge \lambda(v_0, v) = 1\Big) \Longrightarrow \lambda(v).\mathcal{E} = T\Big)$

Relabeling :

- $\lambda'(v_0) := (T, -1)$

The construction of the BFS tree is terminated when all the neighbors of the A-labeled node become T-labeled. In this case, the A-labeled node becomes T-labeled (Rule R_t^7).

R_t^7 : **Termination detection**

Precondition :

- $\lambda(v_0).\mathcal{E} = A$
- $\forall v \in B(v_0, 1)\ \Big(\lambda(v).\mathcal{E} = T\Big)$

Relabeling :

- $\lambda'(v_0).\mathcal{E} = T$

Remark 3.4.1 *Note that only the A-labeled node detects the global termination of the algorithm. By adding a new broadcast rule, we are able to let all nodes detect the global termination of the BFS tree construction.*

3.4.2 Global function computation

In many distributed algorithms, the convergecast and the broadcast are used in order to compute some functions of the graph. Suppose for instance that we want a source node to compute a global function $f(X_{v_1}, X_{v_2}, ..., X_{v_n})$ where X_v is an input stored in each node v. Suppose that f verifies the following properties (we adopt the same notations as in [Pel00] page 36) :

- f is well-defined for any subset of the inputs.
- f is associative and commutative.

For simplicity, let us assume that we have a precomputed spanning tree (obtained for example by using the relabeling system $\mathcal{R}_t$). Such a function f, also called *semi-group function*, can be computed in a distributed manner by performing a convergecast process using the precomputed tree.

In fact, $f(X_{v_1}, X_{v_2}, ..., X_{v_n})$ can be computed using the relabeling system $\mathcal{R}_f = (\mathcal{L}_f, \mathcal{I}_f, \mathcal{P}_f)$ defined by:

- $\mathcal{L}_f = (\mathcal{E} \times \mathcal{X} \times \mathcal{Y}) \cup \{0, 1\}$, where $\mathcal{E} = \{S, F, T\}$, $\mathcal{X}$ is the set of the possible input values of function f, and $\mathcal{Y}$ is a set of intermediate partial values of function f,
- $\mathcal{I}_f = \{S\} \cup \{0, 1\}$. Initially, all the nodes are labeled S. An edge with label 1 is part of the precomputed spanning tree. Edges with label 0 are not in the tree.
- $\mathcal{P}_f = \{R_f^1, R_f^2\}$ where the two rules R_f^1 and R_f^2 are described below.

By *"local value of f"* in rule R_f^1, we mean the value of f computed on the subtree T_{v_0} rooted at node v_0. The variable Y_1 in rule R_f^1 contains the value of f applied to all the entries X_{v_i} with $v_i \in T_{v_0}$ and $v_i \neq v_0$. The local value of f computed by v_0 enables the parent u of v_0 to compute its own local value of f. Each time a node applies Rule R_f^1, it becomes F-labeled which means that it has finished to compute the local value of f. At the end of the convergecast process, it remains only one node with label S. Thus, this node applies rule R_f^2 and it obtain the global value $f(X_{v_1}, X_{v_2}, ..., X_{v_n})$.

R_f^1 : **Convergecast: computation of the local value of f**

Precondition :

- $\lambda(v_0).\mathcal{E} = S$
- $\exists!\, v_1 \in B(v_0,1)\ \Big(\lambda(v_1).\mathcal{E} = S \wedge \lambda(v_0,v_1) = 1\Big)$

Relabeling :

- $Y_1 := f\big(\bigcup_{v_i \in B(v_0,1)(\lambda(v_i).\mathcal{E}=F \wedge \lambda(v_0,v_i)=1)} \lambda(v_i).\mathcal{Y}\big)$.
- $\lambda'(v_0).\mathcal{E} := F$
- $\lambda'(v_0).\mathcal{Y} := f(\lambda(v_0).\mathcal{X}, Y_1)$

R_f^2 : **Convergecast: global computation of f and termination detection**

Precondition :

- $\lambda(v_0).\mathcal{E} = S$
- $\forall v \in B(v_0,1)\ (\lambda(v).\mathcal{E} = F)$

Relabeling :

- $Y_1 := f\big(\bigcup_{v_i \in B(v_0,1)(\lambda(v_0,v_i)=1)} \lambda(v_i).\mathcal{Y}\big)$.
- $\lambda'(v_0).\mathcal{E} := T$
- $\lambda'(v_0).\mathcal{Y} := f(\lambda(v_0).\mathcal{X}, Y_1)$

Remark 3.4.2 *Note that the convergecast process used in rule R_f^2 does not end at a pre-distinguished node but at some node which is elected at random depending on the algorithm execution.*

Remark 3.4.3 *Using the two generic rules R_f^1 and R_f^2, we can encode in a formal way some classical distributed algorithms. For instance, in order to compute the maximum of some values stored by the network nodes, we take $f := max$; for the sum over the node inputs, we take $f := +$. This technique can also be used to design distributed algorithms for computing some logical functions. For instance, let $Pred(v)$ some predicate using some local*

*variables of node v, and suppose that we want to design a distributed algorithm to decides whether the formula $\{\exists v \in V \mid Pred(v)\}$ is true or false (i.e., is there some node v such that $Pred(v) = true$). This can be done by defining for each node v the variable X_v to be 1 if $Pred(v) = true$ and 0 otherwise, and by taking f to be the logical $\vee$ (*or*) operator. Similarly, the formula $\{\exists v \in V \mid Pred(v)\}$ can be evaluated by taking f to be the logical $\wedge$ (*and*) operator. Now, by combining with the PIF technique, we can encode more sophisticated functions.*

3.5 Distributed minimum spanning tree: Prim's algorithm

3.5.1 Preliminaries

The goal of this section is to show that by combining some basic relabeling systems, we can design more sophisticated algorithms in a detailed and comprehensive way. We choose the minimum spanning tree in order to illustrate our approach. In fact, the distributed solution of this problem uses a careful and non trivial combination of the basic techniques described in the previous sections.

Recall that given a weighted graph G, a minimum spanning tree of G (MST for short) is a spanning tree T such that the sum of the weights of the edges of T is the minimum over all possible spanning trees of G. The problem has been heavily studied and many features of the MST problem, such as the distributed computability of such a tree or the time complexity for constructing it, were studied under many assumptions in past works. In this work we only want to use relabeling systems in order to design a distributed MST algorithm. In particular, we assume that the edge weights are unique, real and positive. Under this assumption, it is well known that there exists a unique minimum spanning tree of G (see [CLR90, Tel00, Pel00] and references there).

One of the most basic algorithms for computing a MST is Prim's algorithm [Pri57, Tel00] (see Fig. 3.3). Starting from one node, Prim's algorithm grows a fragment of the MST by adding at each phase the minimum outgoing edge (MOE for short) to this fragment.

Input: a weighted graph $G = (V, E)$.

Step 1: Initially, set E_T (a set of edges) empty.
Set V_T (a set of nodes) empty.
Step 2: Select an arbitrary node in V, and add it to V_T.
Step 3: Find the lowest weight edge $e = (u, v)$ such that $u \in V_T$ but $v \notin V_T$. Add v to V_T, and e to E_T.
Step 4: Repeat *Step* 3 until V_T equals V.

Output: A minimum spanning tree $T = (V, E_T)$.

Figure 3.3: Prim's Algorithm

The classical distributed implementation of this algorithm works in many phases where

each phase can be decomposed in two stages. In the first stage, the nodes in the fragment cooperate to compute the weight of the MOE. This is performed by a convergecast using the constructed fragment. In the second stage, the MOE is added by broadcasting the weight of the MOE to all nodes in the fragment. When learning about the weight of the MOE, a node adds the new edge to the fragment if the MOE is incident to it.

The main difficulty here is to combine many broadcast and convergecast operations (PIF) with the MOE computation (global function). By combining $\mathcal{R}_t$ and $\mathcal{R}_{f=min}$, Prim's algorithm can be encoded by the graph relabeling system $\mathcal{R}_m$=$(\mathcal{L}_m, \mathcal{I}_m, \mathcal{P}_m)$ defined by:

- $\mathcal{L}_m$=$\big(\mathcal{E} \times w_{subtree} \times w_{local} \times i\big) \cup \{0,1\}$ where $\mathcal{E} = \{S, F, T, O\}$, $i = \{-1, 1\}$, and $w_{subtree}$ and w_{local} are subsets of $\in \mathbb{R}^2_+ \cup \perp$,
- $\mathcal{I}_m = \{(O, \perp, \perp, -1),\ (S, \perp, \perp, -1)\} \cup \{0\}$. Initially, there is a distinguished node with label $(S, \perp, \perp, -1)$ which is the first node in the fragment. All other nodes are labeled $(O, \perp, \perp, -1)$. All edges are labeled 0.
- $\mathcal{P}_m$=$\{R^1_m, R^2_m, R^3_m, R^4_m,\ R^5_m\}$ where the five rules are described below.

If the value of some label is equal to $\perp$, then it means that this value has not been computed yet. If an edge becomes labeled 1 then it is part of the MST.

Remark 3.5.1 *Similarly to the relabeling system $\mathcal{R}_t$, if the value of the attribute i is equal to 1 (resp. -1), then an S-labeled node knows that it is in the broadcast (resp. convergecast) stage.*

3.5.2 Computing the weight of the MOE

The nodes of the fragment must cooperate in order to compute the MOE, i.e., convergecast from the leaves up to an elected node (rule R^1_m). Each node must first compute the attribute $w_{subtree}$ which is the weight of the minimum outgoing edge of the subtree rooted at it. Note that, during the convergecast process, each node also stores the attribute w_{local} which is the weight of its local minimum outgoing edge. In the broadcast stage, the w_{local} enables a node to know whether its local minimum outgoing edges corresponds to the global one or not (recall that we have assumed that edge weights are unique).

R^1_m : **Convergecast: computing the minimum outgoing edge**

Precondition :

- $\lambda(v_0) = (S, \perp, \perp, -1)$ `/*convergecast stage*/`
- $\forall v \in B(v_0, 1)\ \Big(\Big(\lambda(v).\mathcal{E} = S \wedge \lambda(v_0, v) = 1\Big) \Longrightarrow \lambda(v).i = -1\Big)$
- $\exists!\, v_1 \in B(v_0, 1)\ \Big(\lambda(v_1).\mathcal{E} = S \wedge \lambda(v_0, v_1) = 1\Big)$
  `/*all children have received an acknowledgment */`

Relabeling :

- $w := \min\Big\{\big\{\mathcal{W}(v_0,v) \mid v \in B(v_0,1) \wedge \lambda(v).\mathcal{E} = O\big\} \cup \big\{+\infty\big\}\Big\}$ `/*local MOE*/`
- $w_{min} := \min\Big\{\big\{\lambda(v).w_{subtree} \mid v \in B(v_0,1) \wedge (\lambda(v_0,v)=1) \wedge (\lambda(v).\mathcal{E} = F)\big\} \cup \big\{w\big\}\Big\}$
  `/*the MOE of the subtree rooted at` v_0`*/`
- $(w_{min} = +\infty) \Longrightarrow \lambda'(v_0).\mathcal{E} := T$ `/*local termination*/`
- $(w_{min} \neq +\infty) \Longrightarrow \lambda'(v_0) := (F, w_{min}, w, -1)$

At the end of the convergecast, the weight w_{min} of the MOE is computed at some elected node (rule R_m^2). This node sets its label to $(S, w_{min}, w, 1)$ in order to begin the broadcast phase. (Note that rule R_m^2 also enables to initialize the MST construction).

R_m^2 : **End of the convergecast: election of a node**

Precondition :

- $\lambda(v_0) = (S, \perp, \perp, -1)$
- $\forall v \in B(v_0,1)\Big(\lambda(v_0,v) = 1 \Longrightarrow \lambda(v).\mathcal{E} \neq S\Big)$

Relabeling :

- $w := \min\Big\{\big\{\mathcal{W}(v_0,v) \mid v \in B(v_0,1) \wedge (\lambda(v).\mathcal{E} = O)\big\} \cup \big\{+\infty\big\}\Big\}$
- $w_{min} := \min\Big\{\big\{\lambda(v).w_{subtree} \mid v \in B(v_0,1) \wedge (\lambda(v_0,v)=1) \wedge (\lambda(v).\mathcal{E} = F)\big\} \cup \big\{w\big\}\Big\}$
- $(w_{min} = +\infty) \Longrightarrow \lambda'(v_0).\mathcal{E} := T$
- $(w_{min} \neq +\infty) \Longrightarrow \lambda'(v_0) := (S, w_{min}, w, 1)$

Remark 3.5.2 *Rules R_m^1 and R_m^2 also allow to detect the termination of the MST construction. In fact, if the weight of the MOE is equal to $+\infty$ then there is no node with label O at the frontier of the fragment and thus all the nodes of the graph are in the fragment.*

3.5.3 Finding and adding the MOE

At the end of the convergecast process (rule R_m^2), there is an elected node with label $(S, w, w', 1)$ that begins the broadcast (attribute i is equal to 1). Thus, a node u with label $(S, w, w', 1)$ first compares w and w':

- If $w{=}w'$ then the MOE is incident to u. Hence, the minimum edge is added and u sets its attribute i to -1 in order to start a new stage (rule R_m^4).
- If $w \neq w'$ then there must exist a children v_1 with label $(F, w, w'', -1)$ from which u has inherited its w value and the MOE must be in the subtree rooted at that children. Thus, the F-labeled children v_1 becomes $(S, w, w'', 1)$-labeled (rule R_m^3). The other F-labeled

children become $(S, \perp, \perp, 1)$-labeled in order re-initialize the computation of a new MOE (rule R_m^5).

R_m^3 : **Broadcasting the weight of the MOE**

Precondition :

- $\lambda(v_0) = (S, w, w', 1)$
- $w \neq w' \wedge (w \neq +\infty) \wedge (w \neq \perp)$
- $\exists!\, v_1 \in B(v_0, 1)\ \Big(\lambda(v_0, v_1) = 1 \wedge (\lambda(v_1).w_{subtree} = w)\Big)$
  `/*weights are unique*/`

Relabeling :

- $\lambda'(v_0) := (S, \perp, \perp, -1)$
- $\lambda'(v_1).E := S, \lambda'(v_1).i := 1$
- $\forall v \in B(v_0, 1)\ \Big(\Big(v \neq v_1 \wedge (\lambda(v_0, v) = 1) \wedge (\lambda(v).\mathcal{E} = F)\Big) \Longrightarrow \lambda'(v) := (S, \perp, \perp, 1)\Big)$

R_m^4 : **Adding the MOE**

Precondition :

- $\lambda(v_0) = (S, w, w, 1)$
- $(w \neq +\infty) \wedge (w \neq \perp)$
- $\exists!\, v_1 \in B(v_0, 1)\ \Big((\lambda(v_0, v_1) = 1) \wedge (\lambda(v_1).\mathcal{E} = O) \wedge (\mathcal{W}(v_0, v_1) = w)\Big)$

Relabeling :

- $\lambda'(v_0) := (S, \perp, \perp, -1)$
- $\forall v \in B(v_0, 1)\ \Big(\Big((v \neq v_1) \wedge (\lambda(v_0, v) = 1) \wedge (\lambda(v).\mathcal{E} = F)\Big) \Longrightarrow \lambda'(v) := (S, \perp, \perp, 1)\Big)$
- $\lambda'(v_1) := (S, \perp, \perp, -1)$
- $\lambda'(v_0, v_1) := 1$

R_m^5 : **Reinitialization**

Precondition :

- $\lambda(v_0) = (S, \perp, \perp, 1)$ `/*the MOE is not in the subtree rooted at` v_0`*/`

Relabeling :

- $\lambda'(v_0) = (S, \perp, \perp, -1)$
- $\forall v \in B(v_0, 1)\ \Big(\Big((\lambda(v).\mathcal{E} = F) \wedge (\lambda(v_0, v) = 1)\Big) \Longrightarrow \lambda'(v) := (S, \perp, \perp, 1)\Big)$

3.6 Future works

In this chapter, we gave a modular methodology that enables to encode a large class of distributed algorithms by mean of relabeling systems. The main strength of our approach is the expressiveness of the algorithms we obtain and their high level nature. However, some other features concerning the proof techniques and the implementation issues remains to be studied in future works:

1. In order to prove the correctness of an algorithm encoded by a relabeling system, the main technique is to exhibit *(i)* some *invariant properties* associated with the relabeling system, i.e., some properties of the graph labeling which are satisfied by the initial labeling and preserved after a relabeling step and *(ii)* some properties of *irreducible* graphs [MMS02], i.e., graph that can not be relabeled any more. The formal correctness of the algorithms given in this chapter can be proved using the previous technique. Nevertheless, since our relabeling systems are clearly expressed as a combination of some basic procedures, we think that it would be more interesting to use the correctness proofs of the basic procedures as building blocks in order to get a global proof. In order to achieve this goal, it seems necessary to provide a more sophisticated *formal scheme* that allows to *combine* some basic relabeling systems *automatically*. In our future work, we plan to provide specific *logical tools* which, given a distributed algorithm expressed in term of some basic procedures, allows to define the corresponding relabeling system automatically. This will also enable us to *provide a formal proof automatically* by *only* using the correctness proofs of the basic procedures.

2. In this work, we have assumed the CS local computation model. Nevertheless, our relabeling systems could have been encoded in less powerful models. Therefore, our work can be viewed as a first step towards a more general methodology which allows to encode a distributed algorithm in any local computation model. Once again, it would be very interesting to provide high level logical tools allowing to combine the basic techniques automatically. Then, by only encoding the basic techniques in different local computation models, we will be able to encode more sophisticated algorithms *automatically* in the corresponding local computation models. In particular, it would be very nice to apply our approach in the local computation model studied in [CM05]. In fact, that model provides a rigorous and theoretical formalization of the distributed messages passing model. The resulting framework would be very precious in order to design and to prove practical message passing algorithms in a very formal way.

Part III

A Mobile Agents Approach in Distributed Computing: the Handshake Problem

Chapter 4

Efficient Distributed Handshake using Mobile Agents

Abstract.

There is a handshake between two nodes in a network, if the two nodes are communicating with one another in an exclusive mode. In this chapter, we give a mobile agent algorithm that allows to decide whether two nodes realize a handshake. Our algorithm can be used in order to solve some other classical distributed problems, e.g., local computations, maximal matching and edge coloring. We give a performance analysis of our algorithm and we compute the optimal number of agents maximizing the mean number of simultaneous handshakes.

In addition, we show how to emulate our mobile agent algorithm in the classical message passing model while maintaining the same performances. Comparing with previous message passing algorithms, we obtain a larger number of handshakes, which shows that using mobile agents can provide novel ideas to efficiently solve some well studied problems in the message passing model.

Résumé.

On dit que deux sommets dans un résau réalisent une *"poignée de main"*, si les deux sommets communiquent l'un avec l'autre de façon exclusive. Dans ce chapitre, nous proposons un algorithme distribué utilisant des agents mobiles, qui permet de décider si deux sommets réalisent une *"poignée de main"*. Notre algorithme peut être utilisé pour résoudre d'autres problèmes distribués classiques, tels que, l'implémentation de règles de réétiquetages, la coloration d'arêtes, et le calcul d'ensemble maximal d'arêtes indépendantes. Nous donnons une étude théorique de la performance de notre algorithme et nous calculons le nombre optimal d'agents qui optimise le nombre moyen de *"poignée de main"* simultanées.

En outre, nous montrons comment émuler notre algorithme dans le modèle avec messages, tout en gardant les mêmes performances. En comparaison avec les algorithmes existants, notre algorithme est plus efficace. Ceci montre que l'utilisation d'agents mobiles peut aider à trouver des idées nouvelles pour résoudre de façon efficace des problèmes bien étudiés dans les modèles avec messages.

4.1 Introduction

4.1.1 Goals and Motivations

In this chapter, we present new efficient handshake algorithms in the distributed model of computation. Generally speaking, a handshake algorithm enables the establishment of safe communications between two nodes, which guarantees that both the two nodes are being communicating with one another in exclusive mode. Distributed solutions of this problem are known in networks supporting message passing [MSZ00, MSZ02, MSZ03, DHSZ06]. What happens if we consider a distributed system based on mobile agents? In particular, we focus on the following question:

Q4:	can we solve the handshake problem using mobile agents while maintaining good performances?

More generally, the growing demand for distributed applications, makes a case for the comparative study of the performances of systems based on mobile agents and systems based on more classical network communications. Many works in the last few years were intended to understand the computational power of mobile agents and to solve new specific problems raised by their use. In this work, we are mainly interested in the *performance aspects* of mobile agents. More precisely, we show how to *efficiently* solve the handshake problem by using mobile agents. Surprisingly, our mobile agent approach also leads to improved solutions and new ideas in the more classical message passing setting. From a general point of view, this work can be viewed as a part of a larger study concerning the complexity power of mobile agents and the benefit they may provide.

4.1.2 Models and notations

We model a network by a connected graph $G = (V, E)$ where V is the set of nodes and E the set of edges. We denote by Δ (resp. δ) the maximum (resp. minimum) degree of G and by $n = |V|$ (resp. $m = |E|$) the number of nodes (resp. edges). For each node $v \in V$, we denote by d_v the degree of v and by $\mathcal{N}(v)$ the neighbors of v, i.e, $\mathcal{N}(v) = \{u \in V \mid d_G(u, v) = 1\}$ where $d_G(u, v)$ is the distance between u and v in G.

In *the mobile agent model*, an agent (or robot) is an autonomous entity of computation able to move from a node to another and equipped with an internal memory. We assume the following:

- each node v is equipped with a white-board $\mathcal{WB}(v)$, which can be viewed as a memory place where agents can write and read information in a mutual exclusion manner.
- the outgoing edges around each node are labeled, that is each node has a numbering of the ports connecting it with its neighbors.

- each agent knows the port from which it is arrived in a given node.
- we only consider the synchronous case where agents have access to a global clock which generates pulses.
- agents can read and write in a white-board in negligible time.
- it takes one time unit to an agent to move from a node to a neighboring one.

For the clarity of our algorithm, we use a local generic function $\textsc{Write}(v) \in \{true, false\}$ which can be applied by each agent at any node v. At a given pulse, if many agents apply $\textsc{Write}(v)$ in node v, then $\textsc{Write}(v)$ returns $true$ for only one agent and $false$ for all others. In this case, the agent for which $\textsc{Write}(v) = true$ has instantaneously a read/write access to $\mathcal{WB}(v)$, i.e., it has access to $\mathcal{WB}(v)$ before the other agents.

In *the message passing model*, a node is an autonomous entity of computation that can communicate with its neighbors by sending and receiving messages. We assume that each node performs computations in negligible time. In the synchronous model, we assume that all nodes have access to a global clock that generates pulses. We assume that messages sent in a given pulse reach their destination before the beginning of the next pulse. In the asynchronous model, there is no global clock and a message delay is arbitrary but finite.

4.1.3 Problem Definition

One can think of several formulations of the handshake problem depending on the distributed model. We have based our work on the following general definition: "a handshake algorithm is a distributed procedure that enables a pair of adjacent nodes (u, v) to communicate *exclusively* with one another *at some time*". In other words, if a handshake occurs between nodes u and v at some time, then u (resp. v) has the guarantee that v (resp. u) does not communicate with any other neighbors.

In general, distributed algorithms solving the handshake problem work in infinitely many *rounds*. At each round, some handshakes occur distributively between pairs of neighboring nodes. Then, communications take place between nodes where a handshake occurs. In practice, there are new rounds as long as some communication between pairs of nodes is required. The handshake problem can then be formulated more practically in terms of *matching: "Given a graph G, at each round, find a set of disjoint edges of E"*. It is clear that the set of these edges defines the nodes that can communicate with each other in an exclusive manner at each round. The number of edges computed at each round is called the *handshake number*. The handshake number is the ruling performance measure of a handshake algorithm. Our goal is to design an algorithm providing the highest possible handshake number.

4.1.4 Related Works

All handshake algorithms in the literature use randomization and message passing. For instance, in the asynchronous message passing model and in the algorithm presented in [MSZ03, MSZ00, RS82], each node repeats forever the following three steps *(i)* choose randomly a neighbor, *(ii)* send him 1, and *(iii)* send 0 to all other neighbors. Then, there is a handshake if two neighboring nodes have sent 1 to each others, and a handshake between two nodes occurs with a given probability. The authors in [MSZ03] studied many probabilistic properties of the above algorithm for many graphs. In the general case, their handshake number is $\Omega(m/\Delta^2)$. Very recently, the authors in [HMR$^+$06] gave a new efficient handshake algorithm and they prove that their algorithm is more efficient than the one of [MSZ03]. However, they assume a fully synchronous message passing model where nodes have access to a continuous real-valued global clock, and *communications take no time*, i.e., if a message is sent at time $t \in \mathbb{R}$, then it arrives at time t. This assumption is the key point of the correctness and the analysis of their algorithm. Therefore, the results in [HMR$^+$06] are fundamentally different from those in this chapter.

Independently of its theoretical interest, the handshake problem can be applied in many settings. For instance, the authors in [DHSZ06] use the handshake algorithm of [MSZ03] in order to efficiently solve the problem of broadcasting information under a restricted model of communication. In [MSZ02], the authors apply the handshake problem in order to practically implement the basic edge local computation model described in Section 3.2 (see also [LMS99, CM05, KY96, Maz97, Ang80, AAER05, Cha05, CM04, CMZ06, GM02] for more results about related local computations models).

The handshake problem is also tightly related to the fundamental problem of breaking the symmetry in distributed networks where nodes have to make decisions depending only on their local views. Typical problems where breaking the symmetry is essential are finding a maximal independent set (MIS), a coloring and a maximal matching. For instance, a maximal matching (see, e.g., [HKP98] for a definition) can be computed using handshakes by deleting the edges computed at each round, and by iterating until the graph is empty. The same idea can be applied for distributed edge coloring by giving a legal color to the edges computed at each round independently and in parallel.

4.1.5 Main results

In this chapter, we give an efficient algorithm for the handshake problem in the mobile agent model. Our algorithm is based on random walks of the agents. We give a probabilistic analysis of the performance of our algorithm. In particular, we compute the optimal (with respect to our method) number of agents that allows a maximal handshake number in expectation. We show that our algorithm is efficient for general graphs, and provides $\Omega(m\delta/\Delta^2)$ handshakes per round. It also becomes of special interest for many graph classes. For instance, for almost

```
1: choose randomly 0 or 1 with probability 1/2;
2: if 0 then
3:    do not move.
4: else
5:    choose at random (equally likely) an outgoing edge e = (v, v');
6:    move to v'.
7: end if
```

Figure 4.1: Algorithm Random Step: code for an agent at node v

Δ-regular graphs, i.e., graphs such that $\delta = \Theta(\Delta)$ (which includes bounded degree graphs), the handshake number is drastically reduced to $\Omega(n)$ which is optimal up to a constant factor.

We also show how we can turn back to the asynchronous message passing model and emulate our algorithm to this model while maintaining the same performances. The technique is based on simulating agents using tokens. From a practical point of view, we obtain new improved message passing handshake algorithms. From a more general point of view, since the simulation technique is independent of the handshake problem, our results show that solving a problem using mobile agents can provide new ideas to design new efficient algorithms in other distributed models.

The rest of this chapter is organized as follows. In Section 4.2 we give our main Basic Agent Handshake algorithm and we analyze its performance. In Section 4.3 we show how to transform our algorithm in the message passing model. In Section 4.4 we discuss how to initialize the agents efficiently.

4.2 Handshake using mobile agents

In this section, we consider the mobile agent model and we assume that the white-board of each node v contains a single boolean variable b. We write $\mathcal{WB}(v) = true$ when $b = true$ and $\mathcal{WB}(v) = false$ when $b = false$. We assume that for every $v \in V$, $\mathcal{WB}(v)$ is initially equal to $false$ and that the network contains k agents. We do not make any assumptions concerning the initial positions of agents.

At pulse 0, each agent begins to execute algorithm Basic Agent Handshake (see Fig. 4.2). The algorithm consists of many rounds. At each round, agents in a node v first try to make a handshake randomly on a given edge. Once the handshake trial is finished, the agents in v move to an equally likely chosen neighboring node (see algorithm Random Step Fig. 4.1). Then, a new round starts.

Remark 4.2.1 *Note that in Fig. 4.2, t_0 denotes the pulse at which a given round begins and it is not used by the agents in order to make any computation.*

Let us consider a round which begins at pulse $t_0 = 3t$ (with $t \geqslant 0$) and let us consider an

Line pulse	The Algorithm
	1: **while** *true* **do**
	2: **if** WRITE(v) **then**
	3: $\mathcal{WB}(v) := true$;
	4: Choose at random (equally likely) an outgoing edge $e = (v, u)$;
	5: Move from v to u;
t_0+1	6: **if** WRITE(u) **then**
	7: **if** $\mathcal{WB}(u) = false$ **then**
	8: Handshake Success;
	9: **end if**
	10: **end if**
	11: Move back from u to v;
t_0+2	12: $\mathcal{WB}(v) := false$;
	13: **else**
	14: **repeat**
	15: wait;
	16: **until** $\mathcal{WB}(v) = false$
	17: **end if**
t_0+2	18: execute algorithm RANDOM STEP;
	19: **end while**

Figure 4.2: Algorithm BASIC AGENT HANDSHAKE: code for an agent at node v

agent $\mathcal{A}$ at some node v. It may happen that many agents are in v at t_0. Only one agent in v is allowed to try to make a handshake. Hence, the agents first "fight" in order to mark the white-board of v. The agent which succeeds in marking $\mathcal{WB}(v)$ is chosen to try a handshake, i.e., line 2 of the algorithm. If agent $\mathcal{A}$ is not chosen to make the handshake, then it just waits (for two pulses) in v until the chosen agent comes back (line 15). Otherwise, agent $\mathcal{A}$ moves to a neighboring node u (line 5). In this case, at pulse $t_0 + 1$ agent $\mathcal{A}$ arrives at u, and three cases arise:

1. $\mathcal{WB}(u) = true$: node u was marked at pulse t_0. Thus, there was an agent in u at pulse t_0, and the handshake fails.

2. $\mathcal{WB}(u) = false$: there were no agents in u at pulse t_0, and no other agents arrive in u at pulse $t_0 + 1$. Thus, WRITE(u) always returns *true* and the handshake succeeds.

3. $\mathcal{WB}(u) = false$: there were no agents in u at pulse t_0, and at least another agent arrives in u at pulse $t_0 + 1$. Thus, if WRITE(u) returns *true* then $\mathcal{A}$ succeeds the handshake.

Remark 4.2.2 *It is important to note that once an agent in a node v executes line* 14 *of algorithm* BASIC AGENT HANDSHAKE*, the white-board of v verifies $\mathcal{WB}(v) = true$, i.e., there is another agent for which* WRITE(v) *in line* 2 *returns true, and which instantaneously writes true in $\mathcal{WB}(v)$.*

To summarize, when an agent is at some node v, we say that it succeeds a handshake, if it can write firstly the white-board of v and secondly the white-board of some unmarked node $u \in \mathcal{N}(v)$. In this case, we also say that a handshake is assigned to edge (u, v). It is clear that our handshake algorithm is correct, that is at each round, the edges where a handshake is assigned are disjoint.

4.2.1 The stationary regime

Let $(\mathcal{A}_i)_{\{i=1,\cdots,k\}}$ denotes the set of all agents. For every integer pulse $t \geqslant 0$ and for every $i \in \{1, \cdots, k\}$, let $\mathcal{A}_i(t) \in V$ denotes the position of agent $\mathcal{A}_i$ at pulse $t \geqslant 0$. Let $\mathbb{P}_G(\mathcal{A}_i(3t) = v)$ denotes the probability that agent $\mathcal{A}_i$ is in node v at pulse $3t$, i.e., the beginning of a round.

From the description of the algorithm, each round takes 3 time units. Thus, each 3 time units, each agent makes a step of a random walk. A classical result from Markov chain theory [Lov96] claims that there exists a unique stationary distribution for random walks on graphs (under some additional assumptions of aperiodicity which is satisfied by line 1 of algorithm RANDOM STEP). The stationary distribution is π the probability measure on G defined by:

$$\pi(v) = \frac{d_v}{2m}, \forall v \in V$$

In other words, if the starting point of a random walk is chosen according to π, then at each time the position of the random walk is still π-distributed. We recall that whatever is the distribution of the starting point, the random walk converges to the stationary distribution, that is, for every $i \in \{1, \cdots, k\}$, when $t \longrightarrow +\infty$, $\mathbb{P}_G(\mathcal{A}_i(3t) = v) \longrightarrow \pi(v)$.

In addition, we assume that each agent aims a proper random generator, and the agents execute algorithm RANDOM STEP independently. Hence, we use the following definition:

Definition 4.2.3 *We say that the k agents are under the stationary regime, if for every $v \in V$ and for every pulse t, we have:*

$$\mathbb{P}_G(\mathcal{A}_i(3t) = v) \;=\; \frac{d_v}{2m} \;=\; \pi(v)$$

and the positions of agents are independent: for any $(v_i)_{i\in\{1,\cdots,k\}} \in V^k$, we have:

$$\mathbb{P}_G\Big((\mathcal{A}_i(3t))_{i\in\{1,\cdots,k\}} = (v_i)_{i\in\{1,\cdots,k\}} \Big) \;=\; \prod_{i=1}^{k} \pi(v_i)$$

Let us consider a given fixed round and let us denote by N^v the number of agents in the node v at the beginning of the round.

Remark 4.2.4 *It is easy to check that under the stationary regime, the r.v. $(N^v)_{v\in V}$ has a multinomial distribution that is, for any family of positive integers $(j_v)_{v\in V}$ such that*

$\sum_{v\in V} j_v = k$,

$$\mathbb{P}\big((N^v = j_v)_{v\in V}\big) = \frac{k!}{\prod_{v\in V} j_v!} \cdot \prod_{v\in V} \pi(v)^{j_v}$$

Notice also that the distribution of the $(N^v)_{v\in V}$ is preserved by a step of the random walks.

The handshake number depends on the graph G, on the number of agents k, on the round and on the initial positions of agents. For the sake of analysis, we only assume the stationary regime and we focus on the expected handshake number $\mathbb{E}(H_k(G))$. We will see in Section 4.4 that assuming the stationary regime is more than of a theoretical interest.

4.2.2 General case analysis: a lower bound

Let us consider an edge $(u, v) \in E$, and let us denote by ω_1 the event *"an agent moves from u to v in line 5"* and by ω_2 the event *"no agent moves from $\mathcal{N}(v) \setminus \{u\}$ to v in line 5"*. We denote by $p(u \rightsquigarrow v) = \mathbb{P}(N^u \geqslant 1, N^v = 0, \omega_1, \omega_2)$ the probability that $\{N^u \geqslant 1\}$, $\{N^v = 0\}$, w_1 and w_2 arise altogether. Similarly, let $p_i(u \rightsquigarrow v) = \mathbb{P}(N^u = i, N^v = 0, \omega_1, \omega_2)$ for any $i \in \{1, \cdots, k\}$.

Lemma 4.2.5 *Under the stationary regime and at any round, the following holds:*

1. *For every edge $(u, v) \in E$ and for every $i \in \{1, \cdots, k\}$, we have:*

$$p_i(u \rightsquigarrow v) \geqslant \frac{1}{d_u} \cdot \binom{k}{i} \cdot \pi(u)^i \cdot (1 - \pi(u) - 2\pi(v))^{k-i}$$

2. *For every edge $(u, v) \in E$, we have:*

$$p(u \rightsquigarrow v) \quad \geqslant \quad \frac{1}{d_u} \cdot \Big((1 - 2\pi(v))^k \ - \ (1 - \pi(u) - 2\pi(v))^k\Big)$$

Proof By the definition of $p_i(u \rightsquigarrow v)$, we have:

$$p_i(u \rightsquigarrow v) = \mathbb{P}(\omega_2 \mid N^u = i, N^v = 0, \omega_1) \cdot \mathbb{P}(\omega_1 \mid N^u = i, N^v = 0) \cdot \mathbb{P}(N^u = i, N^v = 0) \quad (4.1)$$

Using the definitions of N^u and N^v, one has:

$$\mathbb{P}(N^u = i, N^v = 0) = \binom{k}{i} \cdot \pi(u)^i \cdot (1 - \pi(u) - \pi(v))^{k-i} \quad (4.2)$$

Since in line 4 of algorithm BASIC AGENT HANDSHAKE an agent chooses equally likely a node to move to, we have for any $i \in \{1, \cdots, k\}$:

$$\mathbb{P}(\omega_1 \mid N^u = i, N^v = 0) = \frac{1}{d_u} \quad (4.3)$$

It remains to compute $\mathbb{P}(\omega_2 \mid N^u = i, N^v = 0, \omega_1)$. First of all, we note that $\mathbb{P}(\omega_2 \mid N^u = i, N^v = 0, \omega_1) = \mathbb{P}(\omega_2 \mid N^u = i, N^v = 0)$ because knowing $N^u = i$ and $N^v = 0$, the two events w_1 and w_2 are independent.

Remark 4.2.6 *the event w_2 depends on whether or not there are some agents in the neighborhood of v. Hence, w_2 depends on the r.v. $(N^w)_{w\in\mathcal{N}(v)\setminus\{u\}}$. On one hand, using the lemmas given in Section 4.2.4, the r.v. $(N^w)_{w\in\mathcal{N}(v)\setminus\{u\}}$ (knowing $N^u = i$ and $N^v = 0$) can be shown to be a multinomial like r.v.. On the other hand, if $N^w > 0$ for some $w \in \mathcal{N}(v) \setminus \{u\}$, an agent moves to v with probability $1/d_w$ independently of the value of N^w. Thus, we can obtain an exact formula of $\mathbb{P}(\omega_2 \mid N^u = i, N^v = 0, \omega_1)$.*

In the following, we just give a lower bound of $\mathbb{P}(\omega_2 \mid N^u = i, N^v = 0, \omega_1)$, by relaxing the fact that there is only one agent that tries to make the handshake at a given node w.

In fact, assume the following hypothesis $\mathcal{H}$: "in line 5, every agent in a node $w \in \mathcal{N}(v)\setminus\{u\}$ is allowed to move to a neighboring node *independently* of the other agents in w (in other words, not only one agent in w is allowed to move but all of them)". Denote by w_2' the event: "no agent in $w \in \mathcal{N}(v) \setminus \{u\}$ moves to u assuming $\mathcal{H}$ is true". Then, it is clear that $\mathbb{P}(w_2 \mid N^u = i, N^v = 0) \geqslant \mathbb{P}(w_2' \mid N^u = i, N^v = 0)$. Now, it is easy to compute $\mathbb{P}(w_2' \mid N^u = i, N^v = 0)$. In fact, if we assume that $\mathcal{H}$ is true, then each agent in a node $w \in \mathcal{N}(v) \setminus \{u\}$ moves to v with probability $1/d_w$. Thus, we get the following:

$$\mathbb{P}(\omega_2 \mid N^u = i, N^v = 0, \omega_1) \quad \geqslant \quad \left(1 - \sum_{w\in\mathcal{N}(v)\setminus\{u\}} \pi^{(i)}(w) \cdot \frac{1}{d_w}\right)^{k-i}$$

where $\pi^{(i)}(w)$ is the probability that a fixed agent is in $w \in \mathcal{N}(v) \setminus \{u\}$ knowing that $N^u = i$ and $N^v = 0$. It is easy to check that $\pi^{(i)}(w) = \frac{\pi(w)}{1-\pi(u)-\pi(v)}$.

Thus, we have:

$$\begin{aligned}\mathbb{P}(\omega_2 \mid N^u = i, N^v = 0, \omega_1) \quad &\geqslant \quad \left(1 - \frac{1}{1-\pi(u)-\pi(v)} \cdot \sum_{w\in\mathcal{N}(v)\setminus\{u\}} \frac{d_w}{2m} \cdot \frac{1}{d_w}\right)^{k-i} \\ &= \quad \left(1 - \frac{1}{1-\pi(u)-\pi(v)} \cdot \frac{d_v - 1}{2m}\right)^{k-i}\end{aligned}$$

Note that $\frac{d_v-1}{2m} \leqslant \frac{d_v}{2m} = \pi(v)$. Thus,

$$\mathbb{P}(\omega_2 \mid N^u = i, N^v = 0, \omega_1) \geqslant \left(\frac{1-\pi(u)-2\pi(v)}{1-\pi(u)-\pi(v)}\right)^{k-i} \tag{4.4}$$

Hence, the first part of the lemma holds by putting together (4.1), (4.2), (4.3) and (4.4). The second part of the lemma is a consequence of the first part and can be checked using the Newton's formula. ∎

Note that in line 2 of the algorithm, it does not matter for which agent the WRITE function returns *true*. It only matters that exactly one agent is chosen to move. In addition, in line 6, if there is exactly one agent in the corresponding node, then the WRITE function always returns *true* for this agent. Thus, we get the following:

Fact 4.2.7 *The handshake number verifies:* $\mathbb{E}(H_k(G)) \geqslant \sum_{(u,v)\in E} p(u \rightsquigarrow v) + p(v \rightsquigarrow u)$.

Using Fact 4.2.7 and Lemma 4.2.5, we obtain a general lower bound of the expected handshake number per round, which depends on G and k. In particular, we obtain the following:

Theorem 4.2.8 *Let $G(m)$ be a sequence of graphs such that $G(m)$ has m edges and $\Delta/m \to 0$ when $m \to +\infty$. Then, there exists $k = \Theta(m/\Delta)$ such that under the stationary regime $\mathbb{E}(H_k(G(m))) = \Omega(m\delta/\Delta^2)$.*

Proof For all $v \in V$, we have $\delta \leqslant d_v \leqslant \Delta$ and $\frac{\delta}{2m} \leqslant \pi(v) \leqslant \frac{\Delta}{2m}$. Hence, for all $(u,v) \in E$, $1 - \pi(u) - 2\pi(v) \geqslant 1 - \frac{3\Delta}{2m}$. Thus, for $i \in 1, k$, we have:

$$p_i(u \rightsquigarrow v) \geqslant \frac{1}{\Delta} \cdot \binom{k}{i} \left(\frac{\delta}{2m}\right)^i \cdot \left(1 - \frac{3\Delta}{2m}\right)^{k-i}$$

Thus,

$$p(u \rightsquigarrow v) \geqslant \frac{1}{\Delta} \cdot \left[\left(1 - \frac{3\Delta - \delta}{2m}\right)^k - \left(1 - \frac{3\Delta}{2m}\right)^k\right]$$

Hence, using Fact 4.2.7, we have:

$$\mathbb{E}(H_k(G)) \geqslant \frac{2m}{\Delta} \left[\left(1 - \frac{3\Delta - \delta}{2m}\right)^k - \left(1 - \frac{3\Delta}{2m}\right)^k\right]$$

Let $f(k)$ be the function defined by $f(k) = (\mathbb{E}_1)^k - (\mathbb{E}_2)^k$ where $\mathbb{E}_1 = 1 - \frac{3\Delta-\delta}{2m}$ and $\mathbb{E}_2 = 1 - \frac{3\Delta}{2m}$. Hence, the optimal k which maximizes $f(k)$ is obtained for:

$$k_{opt} = \frac{\log\left(\frac{\log \mathbb{E}_2}{\log \mathbb{E}_1}\right)}{\log\left(\frac{\mathbb{E}_1}{\mathbb{E}_2}\right)}$$

On the other hand, we have: $f(k) = \exp(k \log(\mathbb{E}_2)) \cdot \left(\exp\left(k \cdot \log\left(\frac{\mathbb{E}_1}{\mathbb{E}_2}\right)\right) - 1\right)$. Hence,

$$\begin{aligned} f(k_{opt}) &= \exp(k_{opt} \log(\mathbb{E}_2)) \cdot \left(\exp\left(\log\left(\frac{\log \mathbb{E}_2}{\log \mathbb{E}_1}\right)\right) - 1\right) \\ &= \exp(k_{opt} \log(\mathbb{E}_2)) \cdot \left(\frac{\log \mathbb{E}_2}{\log \mathbb{E}_1} - 1\right) \end{aligned}$$

In the rest of this poof, we suppose that $\Delta = o(m)$. Thus, we have:

$$\log \mathbb{E}_1 \sim -\frac{3\Delta - \delta}{2m} \quad , \quad \log \mathbb{E}_2 \sim -\frac{3\Delta}{2m} \quad , \quad \log\left(\frac{\mathbb{E}_1}{\mathbb{E}_2}\right) \sim \frac{\delta}{2m} \quad \text{and} \quad \frac{\log \mathbb{E}_2}{\log \mathbb{E}_1} \sim \frac{3\Delta}{3\Delta - \delta}$$

Hence,

$$k_{opt} \sim \log\left(\frac{3\Delta}{3\Delta - \delta}\right) \cdot \frac{2m}{\delta} \qquad \text{and} \qquad k_{opt} \log(\mathbb{E}_2) \sim \log\left(1 - \frac{\delta}{3\Delta}\right) \cdot \frac{3\Delta}{\delta}$$

Using the fact that for any x such that $0 < x \leqslant 1/3$, $(\log(1-x))/x = \Theta(1)$, we have $k_{opt} = \Theta(\frac{m}{\Delta})$ and $f(k_{opt}) = \Theta(\frac{\delta}{\Delta})$. Thus, $\mathbb{E}(H_{k_{opt}}(G)) = \Omega(\frac{m\delta}{\Delta^2})$. ∎

Remark 4.2.9 *We note that the performance of the algorithm is not very sensitive to the value of k. For instance, if the value of k varies by a multiplicative constant close to 1, then the handshake number is up to a constant factor the same.*

The previous theorem has to be compared with the previous best known handshake number which is $\Omega(m/\Delta^2)$ (in the asynchronous message passing model). Our lower bound becomes of special interest for graphs having high minimum degree. For instance, if $\delta = \Theta(\Delta)$, then $\mathbb{E}(H_k(G)) = \Omega(n)$ for $k = \Theta(n)$ which is optimal up to a constant factor. In fact, the maximal theoretical handshake number is $n/2$. Note also that for these graphs our algorithm drastically improves the one presented in [MSZ03, MSZ00] by a factor $\Omega(\Delta)$. In particular, if $\delta = \Theta(\Delta) = \Theta(n^\epsilon)$ with $\epsilon \in (0,1)$, we obtain $\Theta(n)$ handshakes against $\Omega(n^{1-\epsilon})$ in [MSZ03, MSZ00]. If $\epsilon = 1$, i.e., almost complete graphs, we obtain $\Omega(n)$ handshakes against $\Omega(1)$. In the next section, we give a different approach to the problem which provides exact bounds for d-regular graphs.

4.2.3 Regular graph analysis: asymptotic tight bound

In this part, we consider a d-regular graph $G_n = (V_n, E_n)$ where d is a fixed integer. For every $v \in V_n$, we suppose given an ordering of the neighbors of v from 1 to d. Let us consider a given fixed round. For every $j \in \{1, \cdots, d\}$, let the r.v. N_j^v be the number of agents in the j-th neighbor of v at the beginning of the given round. Let $\mathcal{N}(v) \rightsquigarrow v$ be the event: *"an agent moves from at least a node in $\mathcal{N}(v)$ to v in line 5"*. Let $\mathbb{P}_n\left(\mathcal{N}(v) \rightsquigarrow v, N^v = 0\right)$ the probability that $\{\mathcal{N}(v) \rightsquigarrow v\}$ and $\{N^v = 0\}$ in G_n. In the remainder, we will make the following assumption:

$$\mathcal{Q} = \Big(n \longrightarrow +\infty,\ k = k(n) \longrightarrow +\infty,\ k(n)/n \longrightarrow c \text{ and } c \in (0, +\infty)\Big)$$

When the graph G_n is regular and under the stationary regime, agents choose equally likely each node v. By symmetry of G_n, the distribution of $(N^v, N_1^v, N_2^v, \cdots, N_d^v)$ does not depend on v. Thus, in the following two lemmas, the node v may be seen as a generic node in V_n, or as the first node for any ordering on the nodes of G_n, or even as a node chosen randomly.

Lemma 4.2.10 *Assume $\mathcal{Q}$. Let v be a generic node in V_n. Under the stationary regime,*

1. *the following convergence in distribution holds*

$$(N^v, N_1^v, N_2^v, \cdots, N_d^v) \quad \xrightarrow[n\to+\infty]{(law)} \quad (X_0, X_1, X_2, \cdots, X_d)$$

where the r.v. X_j are i.i.d and follow a Poisson distribution with parameter c, that is:

$$\mathbb{P}(X_j = \ell) = e^{-c}\frac{c^\ell}{\ell!} \quad \forall \ell \geqslant 0$$

2. for any round, we have

$$\mathbb{P}_n\left(\mathcal{N}(v) \rightsquigarrow v, N^v = 0\right) \quad \xrightarrow[n\to+\infty]{} \quad e^{-c} \cdot \left(1 - \left(1 - \frac{1}{d} + \frac{e^{-c}}{d}\right)^d\right)$$

Proof of the first part of the lemma (sketch): Let v be a generic node. Recall that the agent positions are independent and uniform on all the nodes (for regular graphs). Consider some fixed integers $j_0, j_1, \cdots, j_d$. We write $s = j_0 + j_1 + \cdots j_d$. Using Sterling's formula ($\ell! \sim \sqrt{2\pi\ell}\left(\frac{\ell}{e}\right)^\ell$)), we have $\mathbb{P}_n\left((N^v, N_1^v, \cdots, N_d^v) = (j_0, j_1, \cdots, j_d)\right) =:$

$$\begin{aligned}
& \tbinom{k(n)}{j_0, j_1, \cdots, j_d, k(n)-s} \left(\frac{1}{n}\right)^s \left(1 - \frac{d+1}{n}\right)^{k(n)-s} \\
= \quad & \frac{k(n)!}{j_0!\; j_1!\; \cdots j_d!\; \cdot (k(n)-s)!} \; \frac{1}{n^s} \; \left(1 - \frac{d+1}{n}\right)^{k(n)-s} \\
\sim \quad & \frac{1}{j_0!\; j_1!\; \cdots j_d!} \; \frac{k(n)!}{(k(n)-s)!\; n^s} \; e^{-k(n)\frac{d+1}{n}} \\
\sim \quad & \frac{1}{j_0!\; j_1!\; \cdots j_d!} \; \frac{\sqrt{2\pi k(n)} \; \left(\frac{k(n)}{e}\right)^{k(n)}}{\sqrt{2\pi(k(n)-s)} \; \left(\frac{k(n)-s}{e}\right)^{k(n)-s}} \; \frac{1}{n^s} \; e^{-k(n)\frac{d+1}{n}} \\
\sim \quad & \frac{1}{j_0!\; j_1!\; \cdots j_d!} \; \frac{e^{-s}}{\left(1 - \frac{s}{k(n)}\right)^{k(n)-s}} \; \frac{k(n)^s}{n^s} \; e^{-k(n)\frac{d+1}{n}}
\end{aligned}$$

Using $k(n)/n \longrightarrow c$, this converges to:

$$\frac{1}{j_0!\; j_1!\; \cdots j_d!} c^s e^{-c(d+1)} \quad = \quad \prod_{i=0}^{d} \mathbb{P}(X_i = j_i)$$

where the r.v. X_i are Poisson(c)-distributed. Hence, the first part of the lemma holds.

Proof of the second part of the lemma: Let $\mathcal{A}(v)$ denotes the number of neighbors of v in which there is at least an agent. Obviously, we have:

$$\mathbb{P}_n\left(\mathcal{N}(v) \rightsquigarrow v, N^v = 0\right) \quad = \quad \sum_{j=1}^{d} \mathbb{P}(\mathcal{N}(v) \rightsquigarrow v \mid N^v = 0, \mathcal{A}(v) = j) \cdot \mathbb{P}(N^v = 0, \mathcal{A}(v) = j)$$

One one hand, we have:

$$\mathbb{P}_n(\mathcal{N}(v) \rightsquigarrow v \mid N^v = 0, \mathcal{A}(v) = j) = 1 - \left(1 - \frac{1}{d}\right)^j$$

On the other hand, when $n \to +\infty$, using the first part of the current lemma, we have:

$$\mathbb{P}_n(N^v = 0, \mathcal{A}(v) = j) \longrightarrow \binom{d}{j} \cdot \left(1 - e^{-c}\right)^j \cdot \left(e^{-c}\right)^{d-j} \cdot e^{-c}$$

Thus,

$$\mathbb{P}_n\left(\mathcal{N}(v) \rightsquigarrow v, N^v = 0\right) \longrightarrow \sum_{j=1}^{d} \left(1 - \left(1 - \frac{1}{d}\right)^j\right) \cdot \binom{d}{j} \cdot \left(1 - e^{-c}\right)^j \cdot \left(e^{-c}\right)^{d-j} \cdot e^{-c} \qquad (*)$$

Using the Newton formula, the right hand side of (*) can be explicitly computed. One finds

$$\mathbb{P}_n\left(\mathcal{N}(v) \rightsquigarrow v, N^v = 0\right) \longrightarrow e^{-c} \cdot \left(1 - \left(1 - \frac{1}{d} + \frac{e^{-c}}{d}\right)^d\right)$$

∎

As a straightforward consequence of Lemma 4.2.10, we get the following:

Theorem 4.2.11 *Assume $\mathcal{Q}$. For every d-regular graph G_n with d an integer constant, under the stationary regime, the handshake number verifies:*

$$\frac{\mathbb{E}(H_k(G_n))}{n} \quad \xrightarrow[n\to+\infty]{} \quad e^{-c} \cdot \left(1 - \left(1 - \frac{1}{d} + \frac{e^{-c}}{d}\right)^d\right) \qquad (4.5)$$

One can numerically find the optimal constant c that maximize the limit given in Theorem 4.2.11. By taking $c = \log(2)$, the right hand side of (4.5) is larger than 0.196... for any d. Thus, our bound is optimal up to a small multiplicative constant factor ($\simeq 5/2$).

4.2.4 General case analysis: a tight bound

In this section, we give a general generic formula that allows to compute the exact handshake number of algorithm Basic Agent Handshake in the general case.

For any event ω, we define $\mathbb{1}_\omega$ as follows:

$$\mathbb{1}_\omega = \begin{cases} 1 & \text{if } \omega \text{ is true,} \\ 0 & \text{if } \omega \text{ is false.} \end{cases}$$

It is easy to show that under the stationary regime and at any round, the handshake number verifies the following:

$$\mathbb{E}(H_k(G)) \;=\; \sum_{v\in V} \mathbb{P}(N^v = 0) \;-\; \sum_{v\in V} \mathbb{P}(\overline{\mathcal{N}(v) \rightsquigarrow v},\ N^v = 0) \qquad (4.6)$$

where $\overline{\mathcal{N}(v) \rightsquigarrow v}$ is the complementary of the event $\mathcal{N}(v) \rightsquigarrow v$. It is clear that for any $v \in V$, $\mathbb{P}(N^v = 0) = (1 - \pi(v))^k$. Thus, it remains to compute $\mathbb{P}(\overline{\mathcal{N}(v) \rightsquigarrow v}, N^v = 0)$ for any $v \in V$.

Let us consider a node $v \in V$ and the set of its neighbors $\cup_{1\leqslant i\leqslant d_v} u_i$. Let $J_v = (0, j_1, j_2, \cdots, j_{d_v})$ where $(j_i)_{1\leqslant i\leqslant d_v}$ is a set of integers. We denote by $p(J_v) = \mathbb{P}\left(\left(N^v, N_1^v, N_2^v, \cdots, N_{d_v}^v\right) = J_v\right)$ the probability that the number of agents at the beginning of a round in node $u_i \in \mathcal{N}(v)$ is equal to j_i for every $i \in \{1, \cdots, d_v\}$, and that there are no agents in v.

Lemma 4.2.12 *For every $v \in V$, we have:*

$$\mathbb{P}(\overline{\mathcal{N}(v) \rightsquigarrow v}, N^v = 0) \quad = \sum_{\substack{0 \leqslant j_1, j_2, \cdots, j_{d_v} \leqslant k, \\ j_1 + j_2 + \cdots + j_{d_v} \leqslant k}} \left(p(J_v) \cdot \prod_{i=1}^{d_v} \left(1 - \frac{\mathbb{1}_{(j_i \geqslant 1)}}{d_{u_i}} \right) \right)$$

and the r.v. $\left(N^v, N_1^v, N_2^v, \cdots, N_{d_v}^v\right)$ is a multinomial like r.v.. More precisely, for every positive integers $(j_1, j_2, \cdots, j_{d_v})$ such that $j_1 + j_2 + \cdots + j_{d_v} \leqslant k$,we have:

$$p(J_v) \quad = \quad \binom{k}{j_1, j_2, \cdots, j_{d_v}} \cdot \prod_{i=1}^{d_v} \pi(u_i)^{j_i} \cdot \left(1 - \pi(v) - \sum_{i=1}^{d_v} \pi(u_i) \right)^{k - j_1 - j_2 \cdots - j_{d_v}}$$

By putting together the previous formulas, we obtain an exact general formula of the expected handshake number for any graph G. It seems hard to deduce a *simple* general formula *for any graph*. Nevertheless, for particular graphs such as trees, our general formula could be very helpfull.

4.3 Handshake in the message passing model

In this section, we show how to simulate our agent based algorithm in the asynchronous message passing model. The general outline of the method consists in using tokens to simulate agents. A similar idea appears in [BFFS03] (see also [CGMO06]) in order to give a necessary condition for deterministic election in anonymous networks.

Initially, we suppose that there are k tokens scattered at some nodes. Each time a node v has one or more tokens, v locally executes the algorithm that the agents are supposed to execute in node v. Each white-board can be simulated using the local variables of the corresponding node. The agent movements can be simulated by sending the tokens from a node to another (If many agents in the original algorithm choose to move to the same direction, then the corresponding tokens are concatenated). Because we have only considered the synchronous mobile agent model, the above simulation method will automatically provide synchronous algorithms in the message passing model. However, by using a classical synchronization technique, we obtain algorithm DISTRIBUTED HANDSHAKE which works in the asynchronous case.

Algorithm DISTRIBUTED HANDSHAKE is given by Algorithm 5. We assume that the function *sendTo* (resp. *sendAll*) allows to send a message to a specified (resp. all) neighbor(s), and the function *receiveFrom* allows to receive a message from a specified incoming edge (if there are no messages then the node waits until a message arrives). The variable #tokens(v) corresponds to the number of agents in node v in the original handshake algorithm. All tokens are given the value 1. When many tokens are sent to the same direction, we simply send their sum.

```
1  while true do
2      if #tokens(v) > 0 then
3          Hs-trial := false;
4          choose an outgoing edge i at random;
5          sendTo(i, 1);                                   /* send a request to one neighbor */
6          for j ∈ [1, d_v] and j ≠ i do sendTo(j, 0);/* send synchronization msg to other neighbors */
7          for j ∈ [1, d_v] do
8              receiveFrom(j);                             /* receive neighbor's request or synchronization msg */
9              sendTo(j, 0);                               /* send a reject or synchronization msg */
10         for j ∈ [1, d_v] do
11             Msg := receiveFrom(j);                      /* receive response of the request */
12             if j = i and Msg = 1 then Hs-trial := true;/* handshake success if response of i is 1 */
13         #moves := 0; move := int [1, d_v];              /* tabular initialized with 0: where to move tokens */
14         for int ℓ = 1 to #tokens(v) do
15             choose 1 or 0 with probability 1/2;
16             if 1 then
17                 choose randomly an outgoing edge i;
18                 #moves ++; move[i]++;
19         #tokens(v) -= #moves;                           /* update tokens: those who stay at the node */
20         for int ℓ = 1 to d_v do sendTo(ℓ, move.[ℓ]); /* move other tokens to the chosen directions */
21     else
22         sendAll(0);                                     /* send synchronization msg */
23         request := boolean [1, d_v];                    /* tabular initialized with false */
24         for j ∈ [1, d_v] do
25             Msg := receiveFrom(j);
26             if Msg = 1 then request.[j] := true;        /* neighbor j is requesting a handshake */
27         if ∃j such that request.[j] = true then
28             choose at random i ∈ [1, d_v] such that request.[i] = true;
29             sendTo(i, 1);                               /* accept request from neighbor i */
30             for every ℓ ≠ i do sendTo(ℓ, 0);            /* reject the others */
31         else
32             sendAll(0);                                 /* synchronization msg */
33         for j ∈ [1, d_v] do Msg := receiveFrom(j);      /* synchronization msg */
34         sendAll(0);                                     /* synchronization msg */
35     for j ∈ [1, d_v] do
36         Msg := receiveFrom(j);                          /* receive incoming tokens */
37         #tokens(v) += Msg;                              /* update the number of tokens */
```

Algorithm 5 : asynchronous DISTRIBUTED HANDSHAKE: code for a node v

Theorem 4.3.1 *Algorithm* DISTRIBUTED HANDSHAKE *is correct and the number of handshakes at a given round is equal to the number of handshakes in algorithm* BASIC AGENT HANDSHAKE.

Using the previous theorem, it makes sense to compare the performance results of Section 4.2 in the mobile agent model and the performances of other algorithms in the message passing model (such as those in [MSZ03, MSZ00]).

Remark 4.3.2 *Another benefit one can obtain from using mobile agents is to reduce the global number of computation entities in the network from n in the message passing model to k in the mobile agent model. We think that this observation defines new criteria allowing to compare distributed solutions of a problem in the two models. For instance, given a distributed problem, one can be interested in finding an algorithm that provides a good compromise between the global resources used in the whole network in order to solve the problem and the performances of the algorithm.*

4.4 Distributed initialization of agents

In Section 4.2, we have assumed the stationary regime in our analysis. In fact, this is relevant if the agents have been moving randomly from a node to another in the network for a sufficiently long time before the computation of some tasks begins. For instance, this assumption can be realistic in the case of some distributed systems where the agents have been created in the past and have been waiting to do some tasks. In this case, the initial positions of agents do not matter and the previous analysis still holds.

If the computations begin before the stationary regime, our algorithms are still correct, only the analysis is different and depends on the initial positions of agents. For instance, if the agents have the same initial departure node, then the number of handshakes will be 1 at the first round and then it increases with time. In opposite, if the agents are well distributed over the network, then intuitively, there will be more handshakes at the first round and it will take less time to reach the stationary regime. In practice, the agents must have been created at some time before any computation begins, but we have not made any assumption on how the agents are created. In the following, we show how to create agents in such a way they are immediately under the stationary regime. The idea is to make the nodes start some agents locally and by their own such that the global number of agents is almost the optimal one since the first round.

First suppose that m is known and let k be the optimal number of agents computed in Section 4.2. Then, *at time* 0, *each node* v *creates a random number* N^v *of agents according to a Poisson law with parameter* $\frac{d_v}{2m} \cdot k$. Let K denotes the total number of agents effectively created by the nodes. Let us first describe the joint distribution of the N^v's knowing that

***Input**: a constant parameter x.*
repeat for ever:
1: create an agent $\mathcal{A}$ with probability x,
2: agent $\mathcal{A}$ tries to make a handshake with a neighboring node chosen at random,
3: agent $\mathcal{A}$ commits suicide.

Figure 4.3: Dist_Bernoulli: code for a node v

$K = \ell$. Let $(j_v)_{v\in V}$ a sequence of integers such that $\sum_{v\in V} j_v = \ell$. Thus, we have:

$$\mathbb{P}((N^v = j_v)_{v\in V} \mid K = \ell) \quad = \quad \frac{\mathbb{P}((N^v = j_v)_{v\in V}, K = \ell)}{\mathbb{P}(K = \ell)} \quad = \quad \frac{\prod_{v\in V} \mathbb{P}(N^v = j_v)}{\mathbb{P}(K = \ell)}$$

By a simple checking, conditionally on $K = l$, the r.v. $(N^v)_{v\in V}$ follows a multinomial distribution. Using Remark 4.2.4, we can conclude that agents are under the stationary regime. To be precise, there is a slight difference with the consideration of Remark 4.2.4, since there the agents were labeled.

Now, it is classical (and easy to show) that K follows a Poisson law distribution with parameter $\sum_{v\in V} \frac{d_v}{2m} \cdot k = k$. Thus, the expected number of agents is $\mathbb{E}(K) = k$. Due to properties of concentration of the Poisson law, K is very close to k, i.e., $\mathbb{P}(|K - k| > k^{1/2+\epsilon}) \longrightarrow 0$ when $k \to +\infty$. Using remark 4.2.9, picking the number of agents at each node according to the Poisson law given above provides w.h.p., the same performances than in Section 4.2. In particular, we have the following:

Proposition 4.4.1 *For any graph G, if m is known, then there exists a distributed procedure for choosing the initial position of agents such that, w.h.p., at any round, the handshake number is up to a constant factor the same as under the stationary regime.*

In the case where neither m nor n are known, we give another distributed solution which is efficient in the case of almost regular graphs. Let $x \in (0, 1)$ be a parameter. Algorithm Dist_Bernoulli depicted in Fig. 4.3 works in rounds. *At each round*, each node creates *only one* agent according to a Bernoulli law with parameter x. If an agent $\mathcal{A}$ is created, then it tries to make a handshake with a neighboring node using the same technique as in algorithm Basic Agent Handshake. Then, the agent $\mathcal{A}$ disappears, and a new round is started.

Notice that the total number K of agents created at each round using Dist_Bernoulli is a r.v. following a binomial distribution with parameter n and x; and its mean is $n \cdot x$ (which matches up to a constant factor the optimal number of agents in Section 4.2 in the case of almost regular graphs). In the next theorem, the handshake number is simply denoted by $H(G)$. Inspired by the analysis of Section 4.2, we can prove the following:

Theorem 4.4.2 *For every graph G, at any round of algorithm* DIST_BERNOULLI*, the expected handshake number verifies:*

$$\mathbb{E}(H(G)) = \sum_{v \in V}(1-x) \cdot \left[1 - \prod_{u \in \mathcal{N}(v)} \left(1 - \frac{x}{d_u}\right)\right]$$

Proof For every $v \in V$, let N^v denotes the number of agents in v which is either 1 or 0. Let us consider a node v with degree d_v. Let ω be the event *"an agent moves from at least a node in $\mathcal{N}(v)$ to v in order to make a handshake in step 2 of algorithm* DIST_BERNOULLI*"*, and $\overline{\omega}$ its complementary event. Thus, we have:

$$\begin{aligned}
\mathbb{P}(\omega, N^v = 0) &= \mathbb{P}(\omega) \cdot \mathbb{P}(N^v = 0) \\
&= (1 - \mathbb{P}(\overline{\omega})) \cdot (1-x) \\
&= \left(1 - \prod_{u \in \mathcal{N}(v)} \left(1 - x \cdot \frac{1}{d_u}\right)\right) \cdot (1-x)
\end{aligned}$$

The theorem follows from the fact that $\mathbb{P}(\omega, N^v = 0)$ is the probability that a handshake occurs in node v. ∎

Corollary 4.4.3 *For every almost d-regular graph G, at any round of algorithm* DIST_BERNOULLI*, the expected handshake number verifies: $\mathbb{E}(H(G)) = \Omega(n)$.*

Proof In the case of an almost regular graph, for every node v, we have $|\mathcal{N}(v)| = d_v = \Theta(\Delta)$. Thus, from Theorem 4.4.2, we have

$$\begin{aligned}
\mathbb{E}(H(G)) &= \sum_{v \in V}(1-x) \cdot \left[1 - \left(1 - \frac{x}{\Theta(\Delta)}\right)^{\Theta(\Delta)}\right] \\
&= n \cdot (1-x) \cdot \left[1 - \left(1 - \tfrac{x}{\Theta(\Delta)}\right)^{\Theta(\Delta)}\right]
\end{aligned}$$

Using the fact that $\left(1 - \frac{x}{\Theta(\Delta)}\right)^{\Theta(\Delta)} = O(e^{-x})$ and because we take x to be a fixed constant, we conclude that $H(G) = \Omega(n)$. ∎

Remark 4.4.4 *For almost regular graphs, Theorem 4.4.2 shows that it is sufficient to make the nodes pick an agent (or a token) at each round according to a Bernoulli law in order to obtain a good handshake number. What happens if agents are created at once at the first round? It is not clear that using the Bernoulli law permits to quickly reach the stationary regime. Following the same reasoning than in Proposition 4.4.1, an alternative solution could be to make* "each node v picks at once a random number of agents N^v according to a Poisson law with a parameter c" *where c is a positive constant. Intuitively, for almost regular graphs and according to the analysis in Section 4.2.3, we think that the stationary regime will be reached very quickly.*

Remark 4.4.5 *An equivalent way to view the* Dist_Bernoulli *algrithm is to initially assign one agent $\mathcal{A}_v$ per node $v \in V$. Then, at each round, each agent $\mathcal{A}_v$ decides according to the Bernoulli law whether or not it tries to make a handshake from v. In this case, there is no more need to create new agents at each round.*

4.5 Open questions

We remark that, with some modifications, algorithm Basic Agent Handshake works in an asynchronous mobile agent model as well . It would be very interesting to give a performance analysis of our algorithm in this case. More precisely, it would be nice to give a theoretical analysis in the case of a weighted graph (where each edge has a weight which models the time needed to be traversed) and weighted agents (where each agent has a weight which models its speed). Hereafter, we discuss some other open problems:

1. It is clear that our algorithm is still correct in the case where we allow an agent in a node v to continue the random walk even if it is not chosen to try a handshake, i.e., if line 2 fails. What can we say about the handshake number in this case? In other words, what happens if an agent moves and tries to make a handshake from another node instead of waiting in v in line 15.

2. For any graph G, how fast the stationary regime is reached, if initially each node creates a random number of agents according to a Poisson law (or even a Bernoulli law) with some parameter possibly depending on its degree? It would be nice to create the agents in such a way the stationary regime is reached in polylogarithmic time for any graph and without any initial knowledge.

3. Using our algorithms, can we improve the results of [DHSZ06]? We conjecture that the answer is yes. More generally, we are optimistic that our technique can help improving related handshake applications. For instance, one can show that our algorithm can be adapted to compute a maximal matching or an edge coloring. We hope that our algorithm can be applied to solve other distributed problems *efficiently* in the mobile agent and message passing models.

Chapter 5

Implementation of Relabeling Systems using Mobile Agents

Abstract.

In this chapter, we give a general framework in order to implement *any* relabeling system in the practical mobile agent model. In particular, we give two new handshake-based algorithms (Agent Full $\mathcal{N}$-Handshake and Agent $\mathcal{N}$-Handshake) allowing to implement the basic CS (Close Star) and OS (Open Star) models. We analyse the performances of our two algorithms and we obtain new improved results compared with previous message passing algorithms.

This work confirms that mobile agents can be very helpful for designing *efficient* and *practical* distributed algorithms.

Résumé.

Dans ce chapitre, nous donnons un cadre général permettant d'implémenter *tout* type de système de réétiquetages dans un modèle distribué pratique utilisant des agents mobiles. En particulier, nous donnons deux algorithmes (Agent Full $\mathcal{N}$-Handshake et Agent $\mathcal{N}$-Handshake) qui étendent la technique *"poignée de main"* et qui permettent d'implémenter les systèmes de réétiquetages de bases dans les étoiles fermés ou ouvertes. Nous analysons la performance de nos deux algorithmes et nous obtenons des nouveaux résultats qui améliorent ceux obtenues précédement dans le modèle avec échanges de messages.

Ce travail confirme le fait que les agents mobiles peuvent être très utiles pour la résolution *efficace* et *pratique* d'algorithmes distribués.

5.1 Introduction

5.1.1 Preliminary results and motivation

One important application of algorithm BASIC AGENT HANDSHAKE described in Chapter 4 is the implementation of the edge local computation model described in Chapter 3. In fact, let us consider a relabeling system $\mathcal{R} = (\mathcal{L}, \mathcal{I}, \mathcal{P})$ in the edge model. We recall that in the edge model, a relabeling rule $R \in \mathcal{P}$ is entirely defined by a precondition and a relabeling on an edge, that is the rule R changes only the labels attached to an edge according to the previous labels of the edge. Now suppose that we have scattered some mobile agents over the network, and suppose that each agent owns a copy of $\mathcal{P}$ as an input. Then, each agent tries to make a handshake on a given edge using algorithm BASIC AGENT HANDSHAKE. From the properties of our handshake algorithm, all the edges where a handshake is assigned are disjoint. Thus, an agent which succeeds a handshake on an edge $e = (u, v)$ can apply a relabeling rule on the edge e in parallel and without interfering with the other agents. In fact, the agent simply collects the labels attached to the edge e, i.e., $\lambda(u)$, $\lambda(v)$ and $\lambda(e)$. Then, it checks whether a rule in $\mathcal{P}$ can be applied or not. If the precondition of a rule $R \in \mathcal{P}$ is satisfied, then the agent assigns the new labels $\lambda'(u)$, $\lambda'(v)$ and $\lambda'(e)$ to u, respectively v and e according to the relabeling specified in rule R. Otherwise, the labels of the edge remain the same. Finally, the agents execute a step of a random walk and start a new round once again. Thus, we obtain a general framework in order to implement the edge local computation model in the practical synchronous distributed mobile agent model.

A natural idea is to extend the BASIC AGENT HANDSHAKE algorithm to other local computation models. In fact, we are interested in designing a *practical framework* for implementing relabeling systems in practical mobile agent models. Such a framework will allow us to abstract the design and the description of a distributed algorithm from its implementation. On one hand, we are able to encode and to prove a distributed algorithm in a formal and unified way using relabeling systems. On the other hand, a practical framework will enable us to implement the obtained relabeling systems and to run them in practical distributed settings.

Many efforts were done in order to implement a distributed algorithm described with relabeling systems in message passing networks. However, no results are known in mobile agent networks. The goal of this chapter is to give an answer to the following question:

Q5:	How can we implement relabeling systems in a mobile agent network in an efficient and practical way?

5.1.2 Related works

In the message passing model, the authors in [MSZ02] gave two frameworks that allow to implement the Open Star (OS) and the Closed Star (CS) local computation models (see

the introduction of Chapter 3 for a formal definition). Let us recall that in the case of the CS model, two relabeling steps can be performed in parallel if they occur on two disjoint stars while, in the case of the OS model two relabeling steps can be performed in parallel if they occur on two stars having disjoint centers. The frameworks of [MSZ02] are based on *randomized local elections* using asynchronous message passing. Hereafter, we recall the algorithms described in [MSZ02]:

- **Implementation of the CS local computation model [MSZ02]:** Each node v selects an integer $rand(v)$ randomly and uniformly from a big set of integers. The node v sends to its neighbors the value $rand(v)$. When it has received from each neighbor an integer, it sends to each neighbor $w \in \mathcal{N}(v)$ the maximum of the set of integers it has received from neighbors different from w. The node v is elected in $\mathcal{N}_2(v)$, i.e., the 2-neighborhood of v, if $rand(v)$ is strictly greater than $rand(w)$ for any node $w \in \mathcal{N}_2(v)$. Following the same notation as in [MSZ02, MSZ00], this local election is called an L_2-*election*. In this case, a computation step may be performed on the star centered in v. During this computation step there is a total exchange of labels by nodes of $\mathcal{N}(v)$, this exchange allows nodes of $\mathcal{N}(v)$ to change their labels. The L_2-*election* algorithm is depicted in Fig 5.1.

Each node** v **repeats forever the following:

1: *the node v selects an integer $rand(v)$ chosen randomly and uniformly from a big set of integers;*
2: *the node v sends $rand(v)$ to its neighbors;*
3: *the node v receives integers from all its neighbors.*
4: *let $m_{v/w}$ the max of the integers that v has received from nodes different from w;*
5: *the node v sends to each neighbor w the integer $m_{v/w}$;*
6: *the node v receives integers from all its neighbors*
7: */* The node v is elected in $\mathcal{N}_2(v)$ if $rand(v)$ is strictly greater than integers received by v; in this case, a relabeling step may be performed in $\mathcal{N}(v)$ */*

Figure 5.1: Randomized L_2-elections [MSZ02]

- **Implementation of the OS local computation model [MSZ02]:** Each node v selects an integer $rand(v)$ randomly and uniformly from a big set of integers. The node v sends to its neighbors the value $rand(v)$. The node v is elected in $\mathcal{N}(v)$, i.e., the 1-neighborhood of v, if for each neighbor $w \in \mathcal{N}(v)$: $rand(v) > rand(w)$. Following the same notation as in [MSZ02], this local election is called an L_1-*election*. In this case, a computation step on the star centered in v is allowed: the center collects the labels of the star and then changes some labels according to a rule. The L_1-*election* algorithm is depicted in Fig 5.2.

Each node v repeats forever the following:

1: *the node v selects an integer $rand(v)$ chosen randomly and uniformly from a big set of integers;*
2: *the node v sends $rand(v)$ to its neighbors;*
3: *the node v receives integers from all its neighbors.*
4: */* The node v is elected in $\mathcal{N}(v)$ if $rand(v)$ is strictly greater than integers received by v; in this case, a relabeling step may be performed in $\mathcal{N}(v)$ */*

Figure 5.2: Randomized L_1-elections [MSZ02]

We remark that the two previous algorithms are based on the consideration of *rounds*. Their performance in terms of number of nodes locally elected in a round was studied in [MSZ02]. In particular, the authors provide the following important proposition, and they interpret it as the degree of parallelism allowed by their algorithms:

Proposition 5.1.1 ([MSZ02]) *Let $G = (V, E)$ be any n-node graph. For any node $v \in V$, let d_v denote the degree of v and $\mathcal{N}_2(v)$ the 2-neighborhood of v. Let Δ denote the maximum degree of G.*

Let $\overline{M_1}(G)$ (resp. $\overline{M_2}(G)$) denote the expected number of nodes locally elected by the L_1 (resp. L_2) algorithm in a round in the graph G. We have:

$$\overline{M_1}(G) \;=\; \sum_{v \in V} \frac{1}{d_v + 1} \;=\; \Omega(\frac{n}{\Delta})$$

and

$$\overline{M_2}(G) \;=\; \sum_{v \in V} \frac{1}{|\mathcal{N}_2(v)| + 1} \;=\; \Omega(\frac{n}{\Delta^2})$$

The algorithms of [MSZ02] can be adapted to the mobile agent model using the technique of [CGMO06]. However, the modified algorithms become very inefficient. In fact, the technique given in [CGMO06] allows to transform any message passing algorithm into a mobile agent one; but it does not care about the performances of the obtained algorithm.

To our knowledge, no frameworks for other local computation models were studied in the past. Furthermore, no work concerning the practical implementation of relabeling systems using mobile agents is known even for the basic models.

5.1.3 Contribution

In this chapter, we investigate for the first time the implementation of relabeling systems using mobile agents.

In the first part of this chapter, that is in Section 5.2, we focus on the CS and OS model. More precisely, we give two algorithms AGENT FULL $\mathcal{N}$-HANDSHAKE and AGENT $\mathcal{N}$-HANDSHAKE in order to implement these two basic local computation models.

We analyze the efficiency of our algorithms, and we obtain some surprising results. For instance, we show that when adapting our agent based algorithms to the message passing model, we obtain new algorithms having a message complexity less than the one of the L_1 and L_2 election algorithms given in [MSZ02].

In the second part of this chapter, that is in Section 5.3, we do not care anymore about the efficiency of implementing relabeling systems by mobile agents. Our main goal is to give a generic framework for implementing *any* local computation model using mobile agents. Our framework is completely independent of the handshake problem and it is simple to understand and to implement in practical networks. In particular, our framework shows that mobile agents can bring a new kind of abstraction in distributed computing.

5.2 Extended handshake algorithms

In the following, we use the same mobile agent model as in Chapter 4. In particular, a network is modeled by an n-node graph $G = (V, E)$ where $|V| = n$ and $|E| = m$. The degree of a node v is denoted by d_v. The maximum (resp. minimum) degree of G is denoted by Δ (resp. δ). We define the 1 (resp. 2) neighborhood of a node v as follows: $\mathcal{N}(v) = \{u \in V \mid d_G(u, v) = 1\}$ (resp. $\mathcal{N}_2(v) = \{u \in V \mid 0 < d_G(u, v) \leqslant 2\}$)

Each node v is equipped with a white-board $\mathcal{WB}(v)$ where agents can read and write information. Each node v aims a local generic function Write(v). In one time unit, there is only one agent for which the Write(v) function returns *true*. Each white-board contains a single boolean variable initialized with $false$. We also assume that there are k agents scattered on the network. It takes one time unit for an agent to cross an edge and negligible time to write or read the white-board of a node.

The algorithms presented in the next two sections work in rounds. We denote by $(\mathcal{A}_i)_{i=\{1,\cdots,k\}}$ the set of all agents. For every integer pulse $t \geqslant 0$ and for every $i \in \{1, \cdots, k\}$, we denote by $\mathcal{A}_i(t) \in V$ the position of agent $\mathcal{A}_i$ at pulse $t \geqslant 0$. Finally, we denote by $\mathbb{P}_G(\mathcal{A}_i(t) = v)$ the probability that agent $\mathcal{A}_i$ is in node v at pulse t.

5.2.1 Full neighborhood handshake

In this subsection, we extend the Basic Agent Handshake algorithm in order to make handshakes between a node and all its neighbors. The new extended algorithm called Agent Full $\mathcal{N}$-Handshake is given in Fig. 5.3.

Algorithm Agent Full $\mathcal{N}$-Handshake works in rounds of $(4\Delta + 1)$ time units each. At the beginning of each round, an agent in a node v tries to make a handshake simultaneously between v and all its neighbors $\mathcal{N}(v)$. In fact, the agent first tries to mark the white-board of v. Then, in case of success, it visits sequentially each node $u \in \mathcal{N}(v)$ and it tries to mark the white-board $\mathcal{WB}(u)$ of node u. If the agent succeeds in marking all the nodes in $\mathcal{N}(v)$, then we say that it succeeds a *full $\mathcal{N}$-handshake*. In other words, the agent succeeds a *full*

```
 1: while true do
 2:    t_0 := t;              /* t: the current pulse */
 3:    M := ∅                 /* set of marked nodes*/
 4:    if WRITE(v) then
 5:       WB(v) := true;
 6:       for each u ∈ N(v) do
 7:          Move from v to u;
 8:          if WB(u) = true then
 9:             Move back from u to v;
10:             go to line 22;
11:          else
12:             if WRITE(u) then
13:                WB(u) := true;
14:                M := M ∪ {u}
15:                Move back from u to v;
16:             else
17:                Move back from u to v;
18:                go to line 22;
19:             end if
20:          end if
21:       end for
22:       for each u ∈ M ∪ {v} do
23:          Move from v to u;
24:          WB(u) := false;
25:          Move back from u to v;
26:       end for
27:    end if
28:    while t ≠ t_0 + 4Δ do
29:       wait;
30:    end while
31:    execute algorithm RANDOM STEP (Fig. 4.1);
32: end while
```

Figure 5.3: Algorithm AGENT FULL $\mathcal{N}$-HANDSHAKE: code for an agent in node v

$\mathcal{N}$-handshake, if it can write first the white-board of v and second the white-board of all nodes $u \in \mathcal{N}(v)$. In this case, we also say that a full $\mathcal{N}$-handshake is assigned to node v.

Notice that the white-boards of marked nodes are re-initialized to $false$ before starting a new round (line 22). Furthermore, at the end of each round, that is after 4Δ time units, every agent performs a random step (defined by algorithm RANDOM STEP given in Chapter 4) of a random walk.

Remark 5.2.1 *In algorithm* AGENT FULL $\mathcal{N}$-HANDSHAKE*, there is a full $\mathcal{N}$-handshake in node v, if there is an agent $\mathcal{A}$ in v such that, the set $\mathcal{M}$ of marked nodes in line* 22 *verifies $\mathcal{M} = \mathcal{N}(v)$.*

Remark 5.2.2 *In algorithm* AGENT FULL $\mathcal{N}$-HANDSHAKE*, we assume that the maximum degree Δ is given in the input of the algorithm, which allows the agent to wait for the start of a new round (line* 28*) and to synchronize their random walks. Nevertheless, this assumption*

is introduced essentially in order to simplify the presentation and the comprehension of the algorithm. In fact, this can be avoided in practical implementations as follows. If an agent $\mathcal{A}$ in node v succeeds the Write(v) *in line* 4*, then it creates d_v temporary "slave-agents". Each slave-agent is responsible for exploring one node in the neighborhood of v. Thus, agent $\mathcal{A}$ does not move from v as specified in lines* 6 *to* 21*. Instead, the slave-agents* (i) *go visit the neighbors of v in parallel and try to write their white-boards. Thenafter,* (ii) *they come back to v and inform $\mathcal{A}$ of the result of their exploration. If all the slave-agents succeed to write the white-boards of the neighbors, then agent $\mathcal{A}$ succeeds the handshake; otherwise it fails. At the end, the slave-agents which succeed their exploration reinitialize (to $false$) the white-boards of the corresponding nodes, and all slave-agents vanish (they die). Thus, the exploration of the neighborhood of node v is only* 3 *time unit consuming instead of 4Δ, and a round last* 4 *time units for all agents.*

One shall also note that using this method, the probability that an agent succeeds a full handshake is no longer the same as the one of the original algorithm. However, the analysis that we will make afterward is still correct.

Using random walk theory, it is easy to show that the distribution of agent positions in algorithm Agent Full $\mathcal{N}$-Handshake converges to a stationary distribution defined by the probability measure π. In our analysis, we use the following definition:

Definition 5.2.3 *We say that the k agents in algorithm* Agent Full $\mathcal{N}$-Handshake *are under the stationary regime, if for every $v \in V$ and for every pulse t, we have:*

$$\mathbb{P}_G(\mathcal{A}_i((4\Delta+1)t) = v) \;=\; \frac{d_v}{2m} \;=\; \pi(v)$$

and the positions of agents are independent: for any $(v_i)_{i\in\{1,\cdots,k\}} \in V^k$, we have:

$$\mathbb{P}_G\Big(\big(\mathcal{A}_i((4\Delta+1)t)\big)_{i\in\{1,\cdots,k\}} = (v_i)_{i\in\{1,\cdots,k\}}\Big) \;=\; \prod_{i=1}^{k}\pi(v_i)$$

Let $p^f(v)$ be the probability that a full $\mathcal{N}$-handshake is assigned to v in a given round of algorithm Agent Full $\mathcal{N}$-Handshake. For every $i \in \{1,\cdots,k\}$, let $p_i^f(v)$ be the probability that at the beginning of a round, *(i)* there are i agents in v, *(ii)* there are no agents in $\mathcal{N}_2(v)$. Then the following lemma holds:

Lemma 5.2.4 *Under the stationary regime, for every node $v \in V$, we have:*

$$p_i^f(v) = \binom{k}{i}\cdot\pi(v)^i\cdot\left(1-\pi(v)-\sum_{u\in\mathcal{N}_2(v)}\pi(u)\right)^{k-i}$$

and

$$p^f(v) \geqslant \left(1-\sum_{u\in\mathcal{N}_2(v)}\pi(u)\right)^{k} - \left(1-\pi(v)-\sum_{u\in\mathcal{N}_2(v)}\pi(u)\right)^{k}$$

Proof The first part holds from a straightforward probabilistic analysis. The second part of the lemma holds from the fact that $p^f(v) \geqslant \sum_{1 \leqslant i \leqslant k} p_i^f(v)$ and by using the Newton's formula. ∎

Let $FH_k(G)$ be the full $\mathcal{N}$-handshake number, i.e., the number of nodes of G to which a full $\mathcal{N}$-handshake is assigned in a round of algorithm AGENT FULL $\mathcal{N}$-HANDSHAKE. It is clear that $\mathbb{E}(FH_k(G)) \geqslant \sum_{v \in V} p^f(v)$. Thus, using Lemma 5.2.4, one can compute a general lower bound of the full $\mathcal{N}$-handshake number. In particular, we have the following:

Theorem 5.2.5 *Let $G(m)$ be a sequence of graphs such that $G(m)$ has m edges and $\Delta^3/m \to 0$ when $m \to +\infty$. Then, there exists $k = \Theta(m/\Delta^3)$ such that, under the stationary regime, the full $\mathcal{N}$-handshake number verifies $\mathbb{E}(FH_k(G)) = \Omega(n\delta/\Delta^3)$.*

Proof (Sketch) Let us assume that $\Delta^3 = o(m)$ and let $\mathbb{E}_1 = 1 - \frac{\Delta^3+\Delta-\delta}{2m}$ and $\mathbb{E}_2 = 1 - \frac{\Delta^3+\Delta}{2m}$. Using Lemma 5.2.4 (and similarly to the proof of Theorem 4.2.8), one can easily check that

$$p^f(v) \geqslant \mathbb{E}_1^k - \mathbb{E}_2^k$$

The best number of agents that maximizes $\mathbb{E}_1^k - \mathbb{E}_2^k$ is obtained for:

$$k \;=\; \frac{\log(\frac{\log(\mathbb{E}_2)}{\log(\mathbb{E}_1)})}{\log(\frac{\mathbb{E}_1}{\mathbb{E}_2})} \;\sim\; \log\left(\frac{\Delta^3+\Delta}{\Delta^3+\Delta-\delta}\right) \cdot \frac{2m}{\delta} \;=\; \Theta(\frac{m}{\Delta^3})$$

The full $\mathcal{N}$-handshake number verifies:

$$\mathbb{E}(FH_k(G)) \;\geqslant\; n \cdot (\mathbb{E}_1^k - \mathbb{E}_2^k) \;=\; n \cdot \exp\left(k \cdot \log(\mathbb{E}_2)\right) \cdot \left(\frac{\log(\mathbb{E}_2)}{\log(\mathbb{E}_1)} - 1\right)$$

By replacing k with its optimal value and by a simple cheking, we find

$$\mathbb{E}(FH_k(G)) \;=\; \Omega(\frac{n\delta}{\Delta^3})$$

∎

Corollary 5.2.6 *Let $G(m)$ be a sequence of almost Δ-regular graphs such that $G(m)$ has m edges and $\Delta/\sqrt{n} \to 0$ when $m \to +\infty$. Then, there exists $k = \Theta(n/\Delta^2)$, such that the full $\mathcal{N}$-handshake number verifies $\mathbb{E}(FH_k(G)) = \Omega(n/\Delta^2)$.*

5.2.2 Simple neighborhood handshake

In this subsection, we are interested in another kind of handshake which can be viewed as an intermediate case between the basic handshake of Section 4.2 and the full handshake of Subsection 5.2.1. The new extended algorithm called AGENT $\mathcal{N}$-HANDSHAKE is given in Fig. 5.4.

Algorithm AGENT $\mathcal{N}$-HANDSHAKE works in rounds of $(2\Delta + 1)$ time units each. At the beginning of each round, an agent in a node v tries to mark the white-board of v. Then, in

```
 1: while true do
 2:    t_0 := t;              /* t: the current pulse */
 3:    if WRITE(v) then
 4:       WB(v) := true;
 5:       for each u ∈ N(v) do
 6:          Move from v to u;
 7:          if WRITE(u) then
 8:             if WB(u) = false then
 9:                Move back from u to v;
10:             else
11:                Move back from u to v;
12:                WB(v) := false;
13:                go to line 18;
14:             end if
15:          end if
16:       end for
17:    end if
18:    while t ≠ t_0 + 2Δ do
19:       wait;
20:    end while
21:    execute algorithm RANDOM STEP (Fig. 4.1);
22: end while
```

Figure 5.4: Algorithm AGENT $\mathcal{N}$-HANDSHAKE: code for an agent in node v

case of success, it visits sequentially the nodes of the neighborhood $\mathcal{N}(v)$ of v and it simply verifies whether the white-boards of nodes in $\mathcal{N}(v)$ have been marked by any other agent. If none of the white-boards of the nodes in $\mathcal{N}(v)$ is marked, then the agent succeeds a *simple $\mathcal{N}$-handshake*. For instance, if an agent succeeds in writing the white-board of v and if there are no agents in the neighborhood $\mathcal{N}(v)$ of v, then the agent succeeds a *simple $\mathcal{N}$-handshake* at v.

Remark 5.2.7 *In algorithm* AGENT $\mathcal{N}$-HANDSHAKE*, an agent re-initialize the white-board of its departure node v to $false$ (line* 12*) immediately without waiting until time $t_0 + 2\Delta$ in line* 18*. Therefore, it may happen that an agent in node v succeeds a simple handshake even if there are other agents in $\mathcal{N}(v)$ at time t_0. Nevertheless, this is not taken into account in our probabilistic analysis.*

Remark 5.2.8 *Remark 5.2.2 can also be applied in the case of algorithm* AGENT $\mathcal{N}$-HANDSHAKE *in order to avoid using Δ in the description of the algorithm.*

Similarly to the previous analysis, the distribution of agent positions converges to a stationary distribution defined by the probability measure π, and thus we use the following definition in our analysis:

Definition 5.2.9 *We say that the k agents in algorithm* AGENT $\mathcal{N}$-HANDSHAKE *are under*

the stationary regime, if for every $v \in V$ and for every pulse t, we have:

$$\mathbb{P}_G(\mathcal{A}_i((2\Delta+1)t) = v) \;=\; \frac{d_v}{2m} \;=\; \pi(v)$$

and the positions of agents are independent: for any $(v_i)_{i\in\{1,k\}} \in V^k$, we have:

$$\mathbb{P}_G\Big(\big(\mathcal{A}_i((2\Delta+1)t)\big)_{i\in\{1,\cdots,k\}} = (v_i)_{i\in\{1,\cdots,k\}}\Big) \;=\; \prod_{i=1}^{k}\pi(v_i)$$

Let $p^s(v)$ be the probability that a simple $\mathcal{N}$-handshake is assigned to node v at some round. For every $i \in \{1,\cdots,k\}$, let $p_i^s(v)$ the probability that at the beginning of a round, *(i)* there are i agents in v, and *(ii)* there are no agents in $\mathcal{N}(v)$. Then, we have the following:

Lemma 5.2.10 *Under the stationary regime, for every node $v \in V$, we have:*

$$p_i^s(v) = \binom{k}{i}\cdot\pi(v)^i\cdot\left(1-\pi(v)-\sum_{u\in\mathcal{N}(v)}\pi(u)\right)^{k-i}$$

and

$$p^s(v) \geqslant \left(1-\sum_{u\in\mathcal{N}(v)}\pi(u)\right)^k - \left(1-\pi(v)-\sum_{u\in\mathcal{N}(v)}\pi(u)\right)^k$$

Let $SH_k(G)$ be the simple $\mathcal{N}$-handshake number, i.e., the total number of nodes of G to which a simple $\mathcal{N}$-handshake is assigned at a round of algorithm AGENT $\mathcal{N}$-HANDSHAKE. Thus, using Lemma 5.2.10, we have the following theorem:

Theorem 5.2.11 *Let $G(m)$ be a sequence of graphs such that $G(m)$ has m edges and $\Delta^2/m \to 0$ when $m \to +\infty$. Then, there exists $k = \Theta(m/\Delta^2)$ such that, under the stationary regime, the simple $\mathcal{N}$-handshake number verifies $\mathbb{E}(SH_k(G)) = \Omega(n\delta/\Delta^2)$.*

Proof (Sketch) Let us assume that $\Delta^2 = o(m)$ and let $\mathbb{E}_1 = 1-\frac{\Delta^2+\Delta-\delta}{2m}$ and $\mathbb{E}_2 = 1-\frac{\Delta^2+\Delta}{2m}$. Using Lemma 5.2.10, we obtain

$$p^s(v) = \mathbb{E}_1^k - \mathbb{E}_2^k$$

The best number of agents that maximizes the previous formula is obtained for:

$$k \;=\; \frac{\log(\frac{\log(\mathbb{E}_2)}{\log(\mathbb{E}_1)})}{\log(\frac{\mathbb{E}_1}{\mathbb{E}_2})} \;\sim\; \log\left(\frac{\Delta^2+\Delta}{\Delta^2+\Delta-\delta}\right)\cdot\frac{2m}{\delta} \;=\; \Theta(\frac{m}{\Delta^2})$$

The simple $\mathcal{N}$-handshake number verifies $\mathbb{E}(SH_k(G)) \geqslant n\cdot(\mathbb{E}_1^k - \mathbb{E}_2^k)$

Thus by a simple checking and for the optimal value of k, we obtain

$$\mathbb{E}(SH_k(G)) = \Omega(\frac{n\delta}{\Delta^2})$$

∎

Corollary 5.2.12 *Let $G(m)$ be a sequence of almost Δ-regular graphs such that $G(m)$ has m edges and $\Delta/n \to 0$ when $m \to +\infty$. Then, there exists $k = \Theta(n/\Delta)$ such that, the simple $\mathcal{N}$-handshake number verifies $\mathbb{E}(SH_k(G)) = \Omega(n/\Delta)$.*

5.2.3 Application to the CS and the OS local computation models

It is straightforward that algorithm Agent Full $\mathcal{N}$-Handshake and algorithm Agent $\mathcal{N}$-Handshake define a generic framework for implementing relabeling systems in the CS and the OS local computation models using mobile agents. In fact, it is sufficient to first let the agents make a handshake using our algorithms. Then, an agent which succeeds a handshake *(i)* collects the labels of its neighborhood, *(ii)* checks if some rules can be applied, *(iii)* applies a rule and then *(iv)* continues the random walk.

However, collecting the neighborhood of a node v is time consuming. Thus, we have to be careful because all our analysis is based on the fact that the agents make a step of a random walk simultaneously (at the same pulse). In order to synchronize the random steps when applying our handshake algorithms to the CS and OS model, the agents have to wait more time before moving to a new node. More precisely, the agents which do not succeed a handshake have to wait until the other agents finish applying a relabeling rule. It is easy to see that the duration of a round increases by only a multiplicative constant 4 in the case of the CS model and 2 in the case of the OS model.

Moreover, based on remark 5.2.2, and using "slave-agents" in order to apply a relabeling rule, the number of time units needed to execute a round can be reduced to 7 for the CS model and to 6 for the OS model.

The previous discussion shows that the handshake number of our algorithms corresponds to the number of stars that can be relabeled simultaneously in parallel. Our handshake numbers have to be compared with the results of [MSZ02] resumed in Proposition 5.1.1.

In particular, we note that in the case of almost Δ-regular graphs, our extended handshake algorithms allow to obtain the same performance as the randomized algorithms of [MSZ02] in the message passing model. In addition, the number of computation entities in the network is drastically reduced by a factor $\Theta(\Delta)$. For instance, for typical values of $\Delta = \Theta(\sqrt{n})$, we can use only $\Theta(\sqrt{n})$ mobile agents against n computation entities for message passing networks, but we achieve the same performance (we compute the same number of stars).

5.2.4 Application to the message passing model

Similarly to algorithm Basic Agent Handshake, our extended algorithms can be adapted to run on a message passing distributed system. The main idea is to use k tokens in order to simulate the k agents. We also use 5.2.2 in order the remove Δ from the description of our new message passing algorithms.

Message passing algorithm for the CS model

Algorithm 6 (SYNCHRONOUS DISTRIBUTED $\mathcal{N}$-HANDSHAKE) given below is a synchronous message passing implementation of algorithm AGENT FULL $\mathcal{N}$-HANDSHAKE.

```
1  while true do
2      if #tokens > 0 then
3          Full-N-Hs-trial := true;
4          sendAll(1);                                   /* send 1 to all neighbors */
5          for j ∈ [1 · ·d_v] do
6              Msg := receiveFrom(j);                    /* receive message from port j */
7              if Msg = 1 then Full-N-Hs-trial := false;
8          if Full-N-Hs-trial then
9              for j ∈ [1 · ·d_v] do
10                 Msg := receiveFrom(j);
11                 if Msg = null then Full-N-Hs-trial := false;
12         else
13             receive all incoming messages and wait for the next pulse;
14         if Full-N-Hs-trial then
15             /* full handshake success*/;
16         MOVE TOKENS;
17     else
18         wait for the next pulse;
19         request := boolean [1 · ·d_v];
20         for j ∈ [1 · ·d_v] do
21             Msg := receiveFrom(j);
22             if Msg = 1 then request.[j] := true;
23         if ∃j such that request.[j] = true then
24             choose at random i ∈ [1, d_v] such that request.[i] = true;
25             sendTo(i, 1);                             /* send 1 to neighbor i */
26     RECEIVE INCOMING TOKENS;
```

Algorithm 6 : SYNCHRONOUS DISTRIBUTED FULL $\mathcal{N}$-HANDSHAKE

In algorithm SYNCHRONOUS DISTRIBUTED FULL $\mathcal{N}$-HANDSHAKE, we have omitted the details of the MOVE TOKENS procedure in line 22. This procedure simulates a random step of agents. For each token, the MOVE TOKENS procedure allows a node to decide with probability 1/2 whether the token is moved or not. Then, an outgoing edge is chosen equally likely and the token is sent through the chosen edge. Symmetrically, the RECEIVE INCOMING TOKENS procedure allows a node to receive the tokens sent by its neighbors. It just verifies (for all edges) whether a token arrives or not using the *receiveFrom* function.

Remark 5.2.13 *In the synchronous message passing model, a message which is sent at a given pulse t arrives at pulse $t+1$. The receiveFrom function allows a node to receive a*

message from a given port in the current pulse. If there are no messages, then it returns null, i.e., the node knows that no messages have been sent by the corresponding neighbor in the previous pulse.

It is not difficult to see that our message passing implementation is correct, that is our algorithm allows to compute a set of disjoint stars at each round. In addition, the analysis made for the mobile agent algorithm can be adapted for our new message passing algorithm and we obtain the same full $\mathcal{N}$-handshake number.

When analyzing the message complexity of our algorithm, we obtain the following interesting lemma:

Lemma 5.2.14 *The message complexity of the* Synchronous Distributed Full $\mathcal{N}$-Handshake *algorithm is* $O(k \cdot \Delta)$ *messages per round.*

On one hand, one can easily see that a synchronous implementation of the L_2-election algorithm described in [MSZ02] and depicted in Fig. 5.1 leads to a message complexity of $O(m)$ messages per round. Hence, using Proposition 5.1.1, the L_2-election algorithm given in [MSZ02] ensures that the expected number of stars locally elected is $\Omega(n/\Delta^2)$ per round using $O(m)$ messages per round.

On the other hand, using Corollary 5.2.6 and Lemma 5.2.14, algorithm Synchronous Distributed Full $\mathcal{N}$-Handshake allows to obtain $\Omega(n/\Delta^2)$ simultaneous relabeling per round while using only $O(n)$ messages per round, for all almost Δ-regular graphs.

Remark 5.2.15 *Notice that we have assumed the stationary regime in our performance analysis, while, the randomized local election algorithms of [MSZ02] do not make any similar assumptions, i.e., their bounds hold at any round. However, the analysis given in Section 4.4 of Chapter 5 allows us to start the agents (tokens) in such a way they are quickly under the stationary regime. In particular, the technique of Section 4.4 is very efficient for almost* Δ *regular graphs.*

Message passing algorithms for the OS model

Algorithm 7 (Synchronous Distributed $\mathcal{N}$-Handshake) given below is a synchronous message passing implementation of algorithm Agent $\mathcal{N}$-Handshake. The discussion and remarks made in previous paragraphs holds also for algorithm Synchronous Distributed $\mathcal{N}$-Handshake. In particular, the algorithm is correct, that is it allows to compute a set of stars having no common centers. The performance of the algorithm is the same as the one with mobile agents. Moreover, we have the following lemma:

Lemma 5.2.16 *The message complexity of the* Synchronous Distributed $\mathcal{N}$-Handshake *algorithm is* $O(k \cdot \Delta)$ *messages per round.*

As a consequence, algorithm SYNCHRONOUS DISTRIBUTED $\mathcal{N}$-HANDSHAKE improves the message complexity of the L_1-election algorithm of [MSZ02] from $O(m)$ message per round to $O(n)$ messages while maintaining the same expected number of elected stars for all almost regular graphs.

```
1  while true do
2      if #tokens > 0 then
3          N-Hs-trial := true;
4          sendAll(1);
5          for j ∈ [1 · ·d_v] do
6              Msg := receiveFrom(j);
7              if Msg = 1 then N-Hs-trial := false;
8          if N-Hs-trial then
9              /* simple handshake success */
10         MOVE TOKENS;
11     else
12         wait for the next pulse;
13         for j ∈ [1 · ·d_v] do
14             Msg := receiveFrom(j);
15     RECEIVE INCOMING TOKENS;
```

Algorithm 7 : SYNCHRONOUS DISTRIBUTED $\mathcal{N}$-HANDSHAKE

5.3 A general framework for implementing local computations with mobile agents

5.3.1 Preliminaries

The previous handshake algorithms can be extended to the more general case of *any* local computation model, that is they can be exetended for implementing relabeling rules on balls of any fixed radius. However, the resulting algorithms become hard to understand and to run in a real setting. In the following, we do not care anymore about the efficiency of implementing a relabeling system. Instead, we want to provide *a very high level and comprehensible framework in order to implement* any *relabeling system in practice* by using mobile agents.

Let us consider a ℓ-local relabeling system $\mathcal{R} = (\mathcal{L}, \mathcal{I}, \mathcal{P})$. We recall from Chapter 3 that $\mathcal{R}$ is ℓ-local if any rule $R \in \mathcal{P}$ is ℓ-local, that is the rule R is entirely defined by the precondition and the relabeling of a generic ball of radius at most ℓ. (Intuitively, only the labels attached to nodes and edges in a ball of radius ℓ are changed).

Assume that we have k agents which have been scattered over the network. Our goal is to make the agents apply the relabelling rules in a distributed way. The main difficulty is to make the agents execute the rules in an independent and concurrent way, that is, if an agent

is executing a rule in some region, then no other agent should execute a rule simultaneously on the same region otherwise the relabeling may be wrong. We first give an algorithm in the case where there is a unique agent in the network ($k = 1$), thenafter we extend the algorithm in the more general case of many agents ($k > 1$).

In the following, we assume that the white-board $\mathcal{WB}(v)$ of a node v contains *a couple* of variables denoted by $\mathcal{WB}(v).c = (X, i)$ with $X \in \{M, L\}$ and $i \in \{1, \cdots, k\}$. The couple (X, i) allows agents to exchange information and to communicate with each other. Typically, it is used by the agents in order to decide whether a node can be relabeled or not. In addition, we assume that the label of the node v is stored in the white-board $\mathcal{WB}(v)$ of v. For simplicity, we use the classical notation $\lambda(v)$, $\lambda'(v)$, $\lambda(u, v)$ and $\lambda'(u, v)$ to denote the label of a node v, repectively the relabeling of v, the label of an edge (u, v) and the relabeling of (u, v).

5.3.2 Single agent implementation

For now we assume that we have only one agent ($k = 1$) in the netwotk to *implement* a local relabeling system. The general idea is to make the agent travel from a node to another and execute the rules at each new visited node. Two main problems must be solved in this scenario:

(P_1): How shall the agent travel the whole network without ommitting any node?

(P_2): How does the agent recognize the ℓ-neighborhood of a node in order to apply a relabeling rule on this node in that neighborhood?

Herafter we give a solution to the two previous problems:

(S_1): The travelling problem can be solved using a *spanning tree*. Thus, first we make the agent construct a rooted spanning tree $\mathcal{T}_G$ of the whole network. Many algorithms are known in the literature and any kind of spanning tree will do. Now, the agent can use $\mathcal{T}_G$ as a *map* in order to travel across the network. For instance, the agent could perform a DFS-traversal of $\mathcal{T}_G$. When the agent visits a new node, it stops the DFS-traversal momentarily in order to execute a rule. Once it has finished the execution of a rule, the agent continues the DFS-traversal to visit another node. Once the entire network is traversed, the agent starts a new DFS-traversal and so on until no further relabeling rules are applicable. This method ensures that all the nodes of the graph will be visited at some time by the agent, such that node starvation is impossible.

(S_2): Now, we describe how the agent can execute a rule R after arrival at some node v. The idea is to make the agent learn the ℓ-neighborhood of v and then check if some rule can be applied. In order to learn the node's ℓ-neighborhood, the agent first constructs a BFS-tree $\mathcal{T}_{B(v,\ell)}$ of $B(v, \ell)$ rooted at v. Then, the agent 'collect' the entire topology of $B(v, \ell)$ by traversing the neighborhood tree $\mathcal{T}_{B(v,\ell)}$. In case the network nodes have

unique identifiers, the learing of a node's ℓ-neighborhood is rather straightforward. In case no such unique identifiers are available, it is also not too difficult to let the agent himself create such identifiers for the visited nodes (e.g., when constructing the tree $\mathcal{T}_G$). Having learned the topology of $B(v,k)$, and having noticed that some relabeling rule r is applicable in the context of $B(v,k)$, the agent visits $B(v,k)$ again (using the neighborhood tree $\mathcal{T}_{B(v,k)}$) and attaches new labels according to rule r.

Remark 5.3.1 *Since the network topology never changes, the BFS-tree $\mathcal{T}_{B(v,\ell)}$ constrcuted by the agent at each node $v \in V$ and allowing to learn the topology of $B(v,\ell)$ can be constructed at once. In fact, using a unique identifier given to node v by the agent, the neighborhood tree $\mathcal{T}_{B(v,\ell)}$ can be assigned specifically to node v, and then it can be used next times.*

5.3.3 Multiple agent implementation

Preliminary.

In the previous section we have argued that a single agent can assign identifiers to network nodes in order to learn the topology of a neighborhood sussrounding that node. In the case of multiple agents $k > 1$ and in order to avoid the problems of breaking the symmetry in *anonymous networks*, we assume that either the graph or the agents are not anonymous (the case of anonymous networks leads to impossibility results that do not interest us in this work).

For the clarity of our algorithms, we assume that each agent $\mathcal{A}_i$ have a unique identifier i ($i \in \{1, \cdots, k\}$). In the rest of this section, we describe our generic framework for implementing a ℓ-local relabelling system for any ℓ and for any $k \geqslant 1$.

The general framework.

Our main idea is to partition the graph G into k regions $(G_i)_{i\in\{1,\cdots,k\}}$ and to assign a region G_i to every agent $\mathcal{A}_i$. Then, each agent applies the relabeling rules in its own region independently of the other agents. Inside a region, an agent applies the relabeling rules like described in the single agent implementation. Thus, it remains to solve two problems:

1. How the application of rules is managed at the borderline between two regions?
2. How the regions are assigned to agents?

Let us first we show how the regions are assigned in a distributed way. The main idea is to make every agent compete against the other in order to compute spanning tree corresponding to its regions.

At the beginning, each agent executes algorithm AGENT_INIT_REGION. A high level description is given in Algorithm 8. Algorithm AGENT_INIT_REGION is a variant of the classical DFS-tree algorithm. For simplicity, we have ommited the details showing how an agent marks a node or an edge (this is straightforward using the white-boards of nodes). After

```
1  if current node v is marked then
2  |  move back to the last visited node;
3  else
4  |  mark the current node v;
5  |  mark the incoming edge as part of T_{G_i}
6  while no unexplored edge is found do
7  |  move back to the last visited node;
8  |  if current node is the root (departure position) and all edges were explored then
9  |  |  the construction of T_{G_i} is finished;
10 move to a new direction (continue the DFS traversal);
11 go to line 1;
```

Algorithm 8 : algorithm AGENT_INIT_REGION: code for agent $\mathcal{A}_i$ in a node v

termination, every agent has computed a spanning tree denoted by $\mathcal{T}_{G_i}$. In other words, the region G_i is defined by the subgraph of G induced by the tree constructed by agent i.

Remark 5.3.2 *Notice that it may happen that an agent fails to compute a tree. In this case, the agent should vanishes and the actual number of agents is decreased. Notice also that the case of a unique agent corresponds to the case where there is only one region (the whole graph).*

Now that the regions are constrcuted, each agent is responsible for executing the rules in its own region. However, some conflicts may occur at the borderline between two adjacent regions. In order to manage these conflicts, each agent $\mathcal{A}_i$ first constructs a BFS-spanning tree $\mathcal{T}_{B(v,\ell)}$ of $B(v,\ell)$ for each node $v \in G_i$ ($\mathcal{T}_{B(v,\ell)}$ may contain nodes in another region $G_j \neq G_i$). Then, each agent $\mathcal{A}_i$ travrerses G_i in a DFS fashion using $\mathcal{T}_{G_i}$. When agent $\mathcal{A}_i$ is at some node $v \in G_i$, it tries to apply a rule using the following four step strategy:

Step 1: At the first step, agent $\mathcal{A}_i$ traverses $\mathcal{T}_{B(v,\ell)}$ and collects the labels of $B(v,\ell)$ in order to check if a rule can be applied. If no rule can be applied, then $\mathcal{A}_i$ continues the traversal of $\mathcal{T}_{B(v,\ell)}$. Otherwise, $\mathcal{A}_i$ goes to the second step.

Step 2: At the second step, agent $\mathcal{A}_i$ traverses $\mathcal{T}_{B(v,\ell)}$ and tries to mark the $\mathcal{WB}(w).c$ field of all nodes $w \in B(v,\ell)$ using the extra label **(M,i)**. If a node $w \in B(v,\ell)$ is already marked **(M,j)** by another agent $\mathcal{A}_j \neq \mathcal{A}_i$, then there are two cases.

1. If $i < j$ then $\mathcal{A}_i$ unmark all the nodes he has already marked and continues the traversal of $\mathcal{T}_{G_i}$ (visit a new node).
2. Otherwise, $\mathcal{A}_i$ marks w with label **(M,i)** and continues the traversal of $\mathcal{T}_{B(v,\ell)}$ (exploration of $B(v,\ell)$).

Step 3: If $\mathcal{A}_i$ succeeds in marking all the nodes of $B(v,\ell)$ with **(M,i)**, then it traverses $\mathcal{T}_{B(v,\ell)}$ once again in order to lock all the nodes in $B(v,\ell)$ by marking them with the extra label **(locked,i)**, i.e., the ball is ready to be relabelled according to a rule. If the label of at least one node $w \in B(v,\ell)$ is not **(M,i)** then $\mathcal{A}_i$ goes back and reinitializes all nodes locked with label **(locked,i)** or marked **(M,i)** and then continues the DFS-traversal of $\mathcal{T}_{G_i}$. When an agent $\mathcal{A}_i$ traverses $\mathcal{T}_{B(v,\ell)}$ in order to lock the nodes, it also collects the topology of $B(v,\ell)$ in order to prepare executing a rule (which avoids to make another traversal).

Step 4: This step is executed if and only if the agent $\mathcal{A}_i$ has succedeed locking all the nodes of $B(v,\ell)$. Hence, the agent traverses $B(v,\ell)$ for the fourth time in order to apply a rule. At the same time, it unlocks the nodes in $B(v,\ell)$. Finally, the agent continues the DFS-traversal of $\mathcal{T}_{G_i}$.

Remark 5.3.3 *Note that an agent executes Step* 2 *if and only if it finds a rule to execute after the first traversal in Step* 1*. Nevertheless, it may happen that in Step* 4*, no rule can be applied since the label of some nodes in* $B(v,\ell)$ *may change.*

Remark 5.3.4 *Notice that several traversals of* $B(v,\ell)$ *are needed only in the case where* v *belongs to the frontier of some other regions, i.e., there exists some* $j \neq i$ *such that* $B(v,\ell) \cap G_j \neq \varnothing$*. Based on this observation, the agents can make some further precomputations in order to mark the nodes at their frontier, and thus they can avoid traversing the ball* $B(v,\ell)$ *several times if node* v *does not belong to the frontier.*

Proposition 5.3.5 *There are no deadlocks in our generic framework, that is:*

- *there are* no agent deadlocks*: An agent can not be blocked an infinite time in any node.*
- *there are* no rule starvations*: If any rule* R *has to be executed in any node* v *in order to continue the relabelling of the graph, then there exists an agent* $\mathcal{A}_i$ *that succeeds to relabel* $B(v,\ell)$ *within a finite time according to* R*.*

Remark 5.3.6 *Suppose that we want to execute an* ℓ*-local relabelling system in a message passing distributed model. Then, it is sufficient to adapt our general algorithm using tokens. Nevertheless, we think that a practical implementation or just a detailed description would be very complicated compared to an agent implementation since:*

1. *We have to write a non uniform code due to the fact that nodes will not play the same role.*
2. *The notion of a region controlled by an agent will completely disappear.*
3. *The notion of only one computation entity (which is the agent) trying to relabel a ball* $B(v,\ell)$ *will also completely disappear.*

For these reasons, we think that mobile agents can help making some distributed solutions more comprehensible and easy to implement in a real distributed system.

5.4 Open questions

The algorithms given in this chapter raise at least two important remarks. First, from an efficiency point of view, mobile agents can lead to improved results compared with message passing for some distributed problems. Second, mobile agents can bring a new level of abstraction in distributed computing. In fact, in the message passing model, the nodes (and the way they are connected, i.e., the graph) define both the topology of the network and the autonomous entities which are computing permanently in the network. In opposite, in the mobile agent model, the nodes define only the topology of the network, while the agents define the computation entities of the network. This observation could be crucial for the case of practical dynamic distributed systems where the network topology can change. For some applications, it seems to be more practical to sperate the topology aspects of the network and the computation aspects by using mobile agents.

Within this context, the general goal of our mobile agent was to bring some new results in order to comprehend the following general problem: Given a distributed problem, which is more accurate: a mobile agent solution or a message passing solution ?

It is obvious that the answer of the previous question will vary according to the distributed problem, to the network topology, and to the network synchrony. We also believe that there are many other parameters which must be taken into account in order to comprehend the strength of the two models and many further investigations and comparative studies have to be done.

Part IV

An Experimental and Educational Approach in Distributed Computing: the *ViSiDiA* Platform

Chapter 6

ViSiDiA: *Vi*sualization and *Si*mulation of *Di*stributed *A*lgorithms

Abstract.

In this chapter, we present a powerful software tool called *ViSiDiA* allowing to visualize and to simulate distributed algorithms. Our tool is intended to researchers and to students as well.

We give a general view of *ViSiDiA* by describing its main features without going into technical or implementation details. More precisely, we show *how ViSiDiA* can be *used* in order to implement, test and experiment distributed algorithms in the asynchronous/synchrnous message-passing/mobile-agent model.

Résumé.

Dans ce chapitre, nous présentons la plateforme *ViSiDiA*: un outil logiciel permettant de visualiser et de simuler des algorithmes distribués. Notre outil est destiné aussi bien aux chercheurs qu'aux étudiants.

Nous décrivons de façon générale et concise les principales fonctionalités de *ViSiDiA* sans rentrer dans les détails techniques d'implémentation. Plus précisement, nous montrons comment *ViSiDiA* peut être utilisée pour implémenter, tester et expérimenter rapidement un algorithme distribué dans un modèle synchrone/asynchrone avec messages/agents-mobiles.

6.1 Introduction

6.1.1 Motivation

The development of increasingly complex and massively distributed information systems on wide-area networks, e.g. Internet, is very fast. The number of servers and machines connected to the net is growing rapidly. Therefore, the implementation of distributed applications is a real challenge. Technological progress in workstations and networks, in particular with high data flow, and the standardization efforts of middle-ware have made it possible to overcome the traditional low-level problems due to heterogeneous networks or the variety of operating systems and programming languages. Thus, distributed systems and co-operative solutions have become an accessible and even current option for many applications in industry, banking, multi-media, or e-commerce. The companies must fit themselves into this ongoing evolution in order to guarantee their productivity and competitiveness in phase with the technological developments. However, the development of distributed applications is a difficult task and a complicated process: In addition to the complexity of classical centralized applications, a new kind of complexity arises due to the distribution and communication between the involved agents, and due to their competitions and resource conflicts.

We conjecture that the availability of toolkits for simulation, visualization and evidence support becomes essential in the design and validation of programs in distributed environments. The underlying models represent the networks by graphs wherein the nodes correspond to machines or processors, and edges to the inter-processor connections. For the designers and researchers of distributed algorithms, such toolkits will make it possible to carry out tests and simulations easily in order to understand the operations of a distributed algorithm and to gain insight into its properties. The availability of such tools seems particularly essential in order to avoid those investigations to be carried out tediously by hand. Moreover, such tools could be useful for educational purposes when the execution of a distributed algorithm has to be explained in the classroom. For the developers, the visualization of distributed program execution allows for an abstraction of the particular applications, which is especially beneficial in designing, installing and testing these programs. For several distributed applications, the development carried out by analyzing text files, traces or logs, turns often out as ineffective because of the complexity and the dynamic changes of data and network. It is assumed that in these cases visualization can be helpful in error diagnosis.

In this context, the main goal of the *ViSiDiA* project is to bring together researchers having various and synergetic competences in distributed algorithms in order to develop a simulation and visualization environment for distributed algorithms. In particular, it aims at providing a real-world experimentation environment. Since many years, many efforts have been done by the *LaBRI*'s distributed computing team in order to achieve these goals. In fact, a software platform also called *ViSiDiA* was developed and many researchers and students are using *ViSiDiA* at the aim of implementing, experimenting and understanding their own

distributed algorithms.

6.1.2 Contribution and outline

In this chapter, we introduce the *ViSiDiA* platform and we give our main contributions. We will not give implementation details. We just outline some important conceptual features of the software and give a kind of a tutorial (or a manual) in order to illustrate the strength of the *ViSiDiA* platform. In fact, our goal is not to give technical explanation which could be hard to follow for non programming experts, but we hope that at the end of this chapter, the reader learns enough about *ViSiDiA* so he can write and run his own distributed algorithms. The software source code and documentation (GPL license) are available for free download on the *ViSiDiA* web page [ViS06]. *ViSiDiA* is written in Java and can be installed and executed very easily. Many examples including those of this chapter are also provided with the software so that the reader can test them.

The rest of this chapter is organized as follows. First, we recall the basic features of *ViSiDiA* by giving a practical example (Section 6.2). Second, we outline *our main contributions and the new features that we add to the* ViSiDiA *platform, namely:*

1. A complete and friendly new application programming interface (api) allowing to simulate and to visualize on the fly a distributed algorithm in *the synchronous message passing distributed model* (Section 6.4). This part improves the existing asynchronous message passing api.

2. A complete and friendly new api allowing to simulate and to visualize on the fly a distributed algorithm in the *asynchronous/synchronous mobile agent distributed model* (Sections 6.3 and 6.4).

3. A *distributed version* of *ViSiDiA* which runs on N ($N \geqslant 1$) real physical machines (Section 6.5). This version is a first step towards a powerful platform allowing an experimentation of distributed algorithms on very huge graphs.

We conclude the chapter by giving some remarks and some future improvements of the *ViSiDiA* platform (Section 6.6).

6.1.3 Related works

ViSiDiA is among the first platforms providing a complete api for implementing, simulating, visualizing and experimenting distributed algorithms in the two classical and widely used message and mobile agents models. However, many other platforms providing related libraries already exist in the literature [KPT03, BA01, SK93, MPTU98, CFJ+03]. Hereafter, we give a general description of the most relevant ones:

- VADE (visualization of algorithms in distributed environment [MPTU98]): This tool allows the end-user to visualize an asynchronous message passing distributed algorithm in a web page while the real execution of the algorithm is done on a remote server. A particular feature of the VADE tool is to make it possible for the programmer to write animations quickly. To the best of our knowledges, there are no recent developments of this tool.

- PARADE (parallel program animation development environment [SK93]): The general goal of this tool is to enable the visualization of many different types of programs, from different architectures, different programming models and languages and different applications. Hence, it is not specifically devoted to distributed algorithms. The general organization of PARADE can be divided into three components. The first component allows to gather the information needed to define a program execution: this is done using trace files of the program actions. The third component allows to construct a graphical view of the program execution. The second or middle component allows to construct a mapping from program execution data to appropriate visualization actions.

- LYDIAN (library of distributed algorithms and animations [KPT03]): This tool is a simulation and visualization environment for distributed algorithms that provides to the students an experimental environment to test and visualize the behavior of distributed algorithms. LYDIAN supports the simulation and the animation of asynchronous message passing algorithms. The simulation is event driven, i.e., when an event takes place, a process performs a computation depending on its state. New protocols can be added and tested easily by end-users using the LYDIAN library. In this case, the user have to implement the actions to be executed by a process depending on the set of possible couples of state/event that could occur. The visualization in LYDIAN takes as input any possible execution trace corresponding to a given algorithm. The user may have different graphical views of an execution. In particular, he can visualize the *Causality* view which illustrate the causal relation between events in the system execution. To the best of our knowledges, LYDIAN covers closest the aspects addressed by *ViSiDiA*.

- DAJ (distributed algorithms in java [BA01]): this tool allows students to write a distributed algorithm (in java) and to interact with the state of a process. The main goal of the DAJ tool is that students achieve a basic understanding of distributed algorithms and be able to create and to analyze their own scenarios. In particular, the students can construct a scenario step-by-step, and they are responsible for selecting the state changes at each step.

6.2 Introduction to *ViSiDiA*: a concrete example in the asynchronous message passing model

6.2.1 Preliminary

Let us consider the asynchronous message passing model and suppose we want to simulate a distributed algorithm on some network. *ViSiDiA* allows us to:

1. write the code to be executed by each node of the network.
2. draw the graph which models the network using the graphical user interface (GUI for short).
3. load the distributed algorithm and start the simulation using the GUI.
4. visualize on the fly the execution of the algorithm on the GUI, that is the messages and the states of nodes and edges are visualized on the GUI in a real-time manner while the algorithm is being executed.

The three last points are automatically handled by *ViSiDiA*. The only effort a user has to do is to write his algorithm and to compile it. Knowing only few procedures provided by the *ViSiDiA* api, testing an algorithm becomes an easy task. In fact, the simulation and the visualization are completely transparent to the user and no specific knowledges of how *ViSiDiA* is practically encoded are required.

Let us explain how a distributed algorithm can be implemented in practice by *ViSiDiA*. First, the user must keep in mind the following three important features:

1. In *ViSiDiA*, each node v is an autonomous entity of computation modeled using a *Java thread*. Each node can communicate only with its neighbors using its outgoing edges (or links). For each node v, the outgoing edges of v are ordered automatically by *ViSiDiA* from 0 to $d_v - 1$ where d_v is the degree of v (see Fig. 6.1).

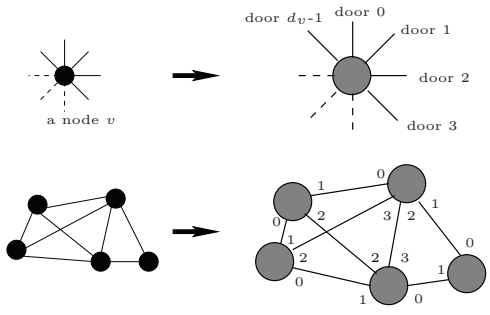


Figure 6.1: The model of a node in *ViSiDiA*

2. A node (thread) can send (resp. receive) messages to (resp. from) its outgoing edges using the **sendTo** (resp. **receiveFrom**) procedures provided by the *ViSiDiA* api.

3. A node (thread) may handle any set of local variables. However, each node can store some local variables using a particular local data structure provided by *ViSiDiA*. In order to write (resp. read) some variables in its local data structure, each node should use the **putProperty** (resp. **getProperty**) procedure of the *ViSiDiA* api. The local data structure enables *ViSiDiA* to visualize the variables stored by the corresponding node.

6.2.2 Example of the Flood algorithm

Now, we are ready to write our first algorithm using *ViSiDiA*. For instance, let us consider the classical FLOOD algorithm for constructing a rooted spanning tree. In Fig 6.2, we give a simplified high level description of this algorithm as it appears in Peleg's book [Pel00] (Chapter 3, page 33). The algorithm begins by broadcasting a message from a root node r_0 to all nodes in the network by simply forwarding the message over all links. The FLOOD algorithm constructs a directed tree T_{FLOOD} rooted at r_0 and spanning all the nodes in the network, with the parent of each non-root node v in T_{FLOOD} being the node from which v has received a message for the first time.

```
1: if the node v = r_0 then
2:    Send a message on all outgoing links.
3: else
4:    Upon receiving a message for the first time over an edge e do
5:          Mark the edge e in the tree T_FLOOD
6:          Forward the message on every other edge
7:    Upon receiving a message again (over other edges) do
8:          Discard it and do nothing
9: end if
```

Figure 6.2: Algorithm FLOOD [Pel00]: high level code for a node v

In Fig. 6.3 we give a concrete and complete implementation of the flood algorithm with the communication details in the message passing model using *ViSiDiA*. In Fig. 6.4, we give a snapshot of the GUI of *ViSiDiA* while the flood algorithm is being executed.

6.2.3 Overview of the *ViSiDiA* api: explanation of the $FloodTree$ algorithm

The *ViSiDiA* *FloodTree* class is organized according to two important rules which must always be satisfied by any algorithm. In the following, we explain these rules:

```
public class FloodTree extends Algorithm {
  /* the code to be executed by each node */
  public void init(){
    /* the degree of the node */
    int nodeDegree = getArity() ;
    /* the door leading to the father in the tree */
    int parentDoor;
    /* the node label */
    String label = (String) getProperty("label");
    /* the current node is the source */
    if(label.equals("R")) {
      /* the node brodcasts a message to neighbors */
      sendAll(new StringMessage("WAVE"));
    } else {
      /* the node waits until a message arrives. The message is
       * returned in the msg variable. The incoming door is
       * returned in the door variable */
      Door door = new Door();
      Message msg = receive(door);
      parentDoor = door.getNum();
      /* the node becomes in the tree */
      putProperty("label",new String("T"));
      /* the edge is marked in bold in the GUI */
      setDoorState(new MarkedState(true),parentDoor);
      /* the node broadcast a message to its neighbors except
       * his parent in the tree */
      for(int i=0; i < nodeDegree; i++){
        if(i != parentDoor) {
          sendTo(i, new StringMessage("WAVE"));
        }
      }
      /* other messages are ignored and the node locally
       * terminates: no more actions to do */
    }
  }
}
```

Figure 6.3: Algorithm *FloodTree*: code for a node with *ViSiDiA*

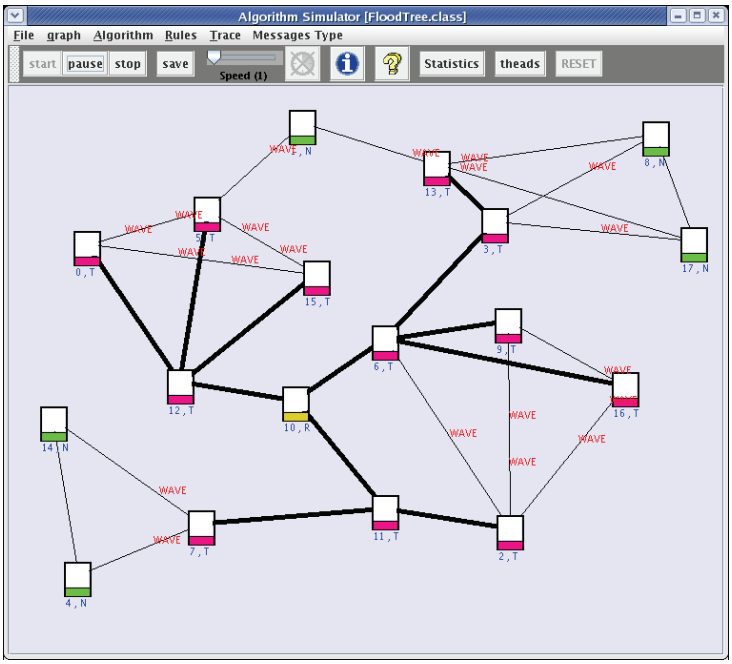


Figure 6.4: An execution of the *FloodTree* algorithm with *ViSiDiA*

- The concrete java algorithm must always inherit from the *Algorithm* class of the *ViSiDiA* api. In fact, all the basic procedures provided to the user and which allow to send and receive messages are accessible only from the *Algorithm* class.

- The code to be executed by each node must be specified in the **init()** method. In fact, the thread created by *ViSiDiA* for each node will execute the instructions specified in the **init()** method automatically.

The abstract class *Algorithm* of *ViSiDiA* provides many methods to help the user writing his algorithm. Let us review some of them based on *FloodTree* example:

- The **sendTo()** method allows a node to send a message on a specified door. The **sendAll()** method allows a node to send a message to all its neighbors. The **receiveFrom()** method allows a node to receive a message. There are two types of the **receiveFrom()** method. The first one allows a node to wait until the reception of a message and it puts the value of the door from which the message arrives in the **Door** variable given in parameter (this is the case in our example). The second one allows a node to wait for a message on a specified door given in parameter. In this case, the **receiveFrom()** method returns if and only if a message arrives from the specified door. Notice that the **receiveFrom()** method is very sensitive, since it locks the node until receiving a message. Here we remark that *ViSiDiA* also provides some boolean methods that enable a node to test whether or not a message is arrived on any door. This allows

a different approach when writing a distributed algorithm. For instance, the user can easily encode the following instruction: *"if a node has received a message (on some door) then it executes some task, otherwise its executes some other task"*.

- A message is implemented in *ViSiDiA* using the $Message$ abstract class. There exist many kinds of messages (string, boolean, integer, vector). In our example we use only string messages which are implemented by the $StringMessage$ class. We also remark that the user can assign many properties to a messages in *ViSiDiA*. For instance, the user can use the $StringMessage$ class to create string messages but with as many types as he wants. This allows to handle the messages according to their types. In other words, the user can easily encode the following instruction: *"if a node receives some message with type X then it executes some task, otherwise it executes some other task"*. The messages can also be easily customized for the visualization. For instance, the user can choose the color of messages according to their types or choose to not visualize some messages.

- As mentioned before, each node has a local data structure which is used to store some particular variables. By default, each node in *ViSiDiA* is given a local data structure containing only one variable: a label initialized to N. Using the GUI of *ViSiDiA*, the user can for instance select a node and assign another label to it. This is very helpful if some nodes have to be initialized with some particular value. This is the case in our example where a root node has to be selected. We choose the special label R to distinguish this node which implies that the user has to assign the label R to a particular node in the GUI before starting the simulation (otherwise, all nodes will have label N). The **getProperty()** method is then used in the concrete algorithm in order to read the value of the label of a node and to decide if the node was chosen by the user as being the root. The **putProperty()** method allows the user to change the label of a node (which is visualized automatically in the GUI). For instance, when a node becomes in the tree, it changes its label from N to T.

- We also use some other methods in our code. For instance, the **getArity()** method is a standard *ViSiDiA* method which returns the degree of the node. The **setDoorState()** allows to change the shape of an edge in the GUI. For instance, in our example, the edges of the tree are drawn in bold.

Let us remark that the instructions of the practical *Floodtree* algorithm (Fig. 6.3) are *essentially the same* as the instructions given by the high level Flood algorithm Fig. 6.2. In addition, the visualization of the execution of the algorithm is completely handled by *ViSiDiA* without any intervention of the user. These two observations together with the library provided by the api of *ViSiDiA* forms the real strength of the *ViSiDiA* platform.

6.2.4 How does it work ? The general architecture of *ViSiDiA*

In the following paragraphs, we outline the general architecture used in *ViSiDiA* and allowing the simulation and the visualization of a distributed algorithm.

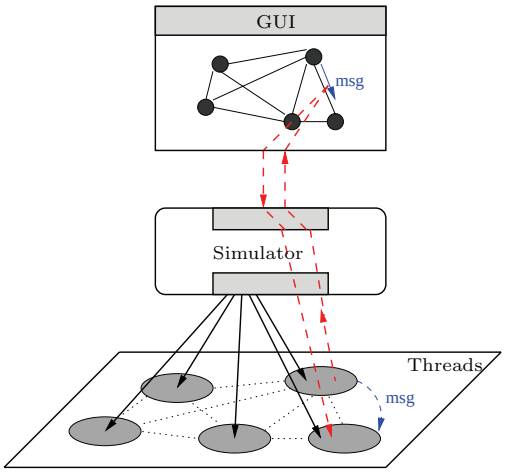


Figure 6.5: General architecture of the centralized version of *ViSiDiA*

In Fig. 6.5, we give the three main components of the *ViSiDiA* platform:

- First, we have the *graphical user interface* which allows to draw a graph. The GUI also allows the visualization of the algorithm.
- Second, we have the *simulator* which is an intermediate entity having a global view of the simulation.
- Third, we have the *threads* which model the nodes of the network and which execute the algorithm.

Remark 6.2.1 *Notice that this three level architecture of* ViSiDiA *allows a programmer to add easily new functionalities to any level independently of the other levels.*

The general idea of the simulation is based on the notion of *events*. For instance, each time a node wants to send a message to a neighbor, a corresponding event $\mathcal{EVT}_1$ is generated. Then, the event $\mathcal{EVT}_1$ is caught by the simulator. The simulator may perform some actions corresponding to the event $\mathcal{EVT}_1$. Then, the simulator generates a new event $\mathcal{EVT}_2$. This event contains all the information needed for the visualization. The event $\mathcal{EVT}_2$ is then caught by the GUI and some visualization is performed. For instance, in the case of a message sending, the GUI draws the message from the sender to the receiver. Once, the visualization corresponding to event $\mathcal{EVT}_2$ is finished, the GUI generates a new event $\mathcal{EVT}_3$ which is caught by the simulator. The simulator uses the information given by $\mathcal{EVT}_3$ in order to perform some actions. For instance, in the case of a message sending, the simulator puts the message in the

queue of the receiver. All these events are handled in a transparent way and the simulation is entirely handled internally by *ViSiDiA*. In fact, the user should only write the code to be executed by each node using the high level procedures provided by *ViSiDiA* as shown in our *FloodTree* example.

6.3 The mobile agent model in *ViSiDiA*

Recently, we have extended the *ViSiDiA* api in order to support the simulation and the visualization of distributed algorithms in the mobile agent model. Similarly to the message passing simulation, *ViSiDiA* allows a user to:

1. write a mobile agent algorithm using the high level procedures provided by the *ViSiDiA* api. This is the only step where the user has to write a concrete Java code.

2. draw the graph which models the network using the GUI.

3. choose the initial position of one or more agents either by hand by picking sequentially a set of nodes using the GUI, or by writing a sequential algorithm which takes in input the whole graph and decides whether an agent is initially created on a given node or not.

4. initialize the white-boards of some nodes (or all off them) by hand by adding some simple (boolean, string, integer ...) variables using the GUI or by adding more sophisticated variables using an already written algorithm.

5. load the mobile agent algorithm and start the simulation using the GUI.

6. visualize on the fly the execution of the algorithm, i.e., the agent movements and node (edge) states in the GUI.

6.3.1 The basic features of the mobile agent api

The agents are modeled using Java threads. A thread is created automatically by *ViSiDiA* for each agent. The thread executes the instructions in the java code written by the user. In order to encode a mobile agent algorithm with *ViSiDiA*, the user must simply inherit from the *Agent* class of *ViSiDiA* and implement the **init()** method. The *Agent* class provides many useful methods that enable the user to write his algorithm. Let us review the most important ones:

- **moveToDoor(int door)**: allows an agent in a node v to move to a neighboring node by using the door number given in parameter.

- **setVertexProperty(Object key, Object var)**: allows an agent in node v to write a variable **var** identified by the **key** parameter on the white-board of v. By default, the white-board of each node contains one variable: its label. The key object allowing to access the label of a node is the string object "label".

- **getVertexProperty(Object key)**: returns the value of the local variable of the white-board of a node corresponding to the **key** parameter. The **getVeretxPropertyKeys()** allows an agent in a node v to learn all the local variables stored in the white-board of node v.

- **lockVertexProperties()**: allows an agent in node v to have an exclusive access to the white-board of node v. If the node is already locked, then the agent waits until the node is unlocked. The **vertexPropertiesLocked()** method allows to test whether node v is already locked by another agent or not. We remark that an agent has an exclusive access to a single variable of the white-board of a node even if the **lockVertexProperties()** is not used.

- **unlockVertexProperties()**: allows the agent in node v to unlock the white-board of node v so that an other agent can have an exclusive access.

- **cloneAgent()**: allows an agent in node v to create a clone of itself in the same node v.

- **createAgent(Class agClass)**: allows an agent in node v to create a new agent in the same node v. The new agent will execute the code specified in the agClass parameter.

6.3.2 A concrete example: Searching for a dangerous thief

Let us consider the following scenario:

- *A dangerous thief arrives to the small and rich town of* Irbal. *The thief goes from house to house and tries to steal the strong-box in each house. It takes few seconds to the thief to break out the alarm device and to steal the strong-box.*

- *The police is aware that a thief is being operating in their town and many policemen are trying to capture him. The only moment a policeman can capture the thief is when the thief is stealing.*

- *The thief is under pressure and once he steals a house, he randomly moves and hides out in a new house for some time, then he tries to steal the new house.*

The policemen are aware of the technique of the thief but they do not know what strategy they must adopt in order to capture the thief quickly before he steals all the town. *ViSiDiA* can help them! In fact, the houses of the town can be modeled using a graph. The thief and the policemen can be simulated using agents. The algorithm executed by the thief can be

easily encoded using the api of *ViSiDiA*. It remains to define the strategy to be adopted by the policemen and to test its performance. Suppose for example that we decide to scatter k policemen in the town, and suppose that each policeman adopts the following strategy: *"move randomly from house to house wondering to creep up on the thief"*.

In Fig. 6.6 (resp. Fig. 6.7), we give the algorithm to be executed by the thief (resp. a policeman). In Fig. 6.8, we give a snapshot of the GUI of *ViSiDiA* while a simulation of the thief and the policemen is running. One can see three policemen moving and the thief stealing the strong-box of node number 0. Notice also that the label of the nodes are used in order to model the state of the thief:

- if a node v has label S, then the thief is in v and is stealing the strong-box of v.
- if a node v has label H, then the thief is in v and is hiding off himself in v.
- if a node v has label N (default), then the strong-box of node v is not yet stolen by the thief.
- if a node v has label D, then the thief has already stolen the strong-box of v.
- if a node v has label C, then the thief was captured by a policeman in node v. Note that we use a static variable **thiefCaptured** in the code of the policemen. This variable is shared by all the policemen and allows a policeman to inform the others that he captured the thief. This can model the following scenario: *"if a policeman captures the thief then he informs all other policemen by phone"*.

6.4 The synchronous model in *ViSiDiA*

Let us first recall that: *" In a synchronous network, there exists a global clock which generates pulses"*. One can then refine this definition by giving precisely the time needed to make some computation.

For instance, we can have the following three different communication assumptions:

- in the message passing model, it takes one time unit for a node to send a message to *one* neighbor.
- in the message passing model it takes one time unit for a node to send a message to *all* its neighbors.
- in the mobile agent model, it takes one time unit to an agent to cross an edge.

In addition, one can make some other kind of assumptions. For instance, we can have the following:

```
public class Thief extends Agent {
  public void init() {
    Random randHide = new Random();
    Random randMove = new Random();

    boolean captured = false;

    while ( ! captured ) {
      int degree = getArity();
      int randomDirection = Math.abs(randMove.nextInt(degree));
      moveToDoor(randomDirection);

      String oldLabel = (String)getVertexProperty("label");

      /* The thief is in the house */
      setVertexProperty("label", new String("H"));
      try {
        /* The thief waits a random time before stealing */
        Thread.sleep((Math.abs(randHide.nextInt(5))+1)*500);
      } catch (Exception e) { }

      /* if the house was not stolen */
      if(oldLabel.equals("N") ) {
        /* The thief is stealing for 3 seconds */
        setVertexProperty("label", new String("S"));
        try {
          Thread.sleep(3000);
        } catch (Exception e) { }

        lockVertexProperties();
        /* if the thief is captured */
        if(((String)getVertexProperty("label")).equals("C"))
          captured = true;
        else
          /* the house is stolen */
          setVertexProperty("label", new String("D"));
        unlockVertexProperties();
      } else {
        /* the house was already stolen */
        setVertexProperty("label", new String("D"));
      }
    }
  }
}
```

Figure 6.6: Thief algorithm: code with *ViSiDiA*

```
public class Policeman extends Agent {
  static boolean thiefCaptured = false;
  public void init() {
    Random randMove = new Random();
    while ( ! thiefCaptured ) {
      int degree = getArity();
      int randomDirection = Math.abs(randMove.nextInt(degree));
      moveToDoor(randomDirection);
      lockVertexProperties();
      /* if the thief is stealing then capture him */
      if(((String)getVertexProperty("label")).equals("S")) {
        thiefCaptured = true;
        setVertexProperty("label", new String("C"));
      }
      unlockVertexProperties();
  }
}
```

Figure 6.7: Policeman algorithm: code with *ViSiDiA*

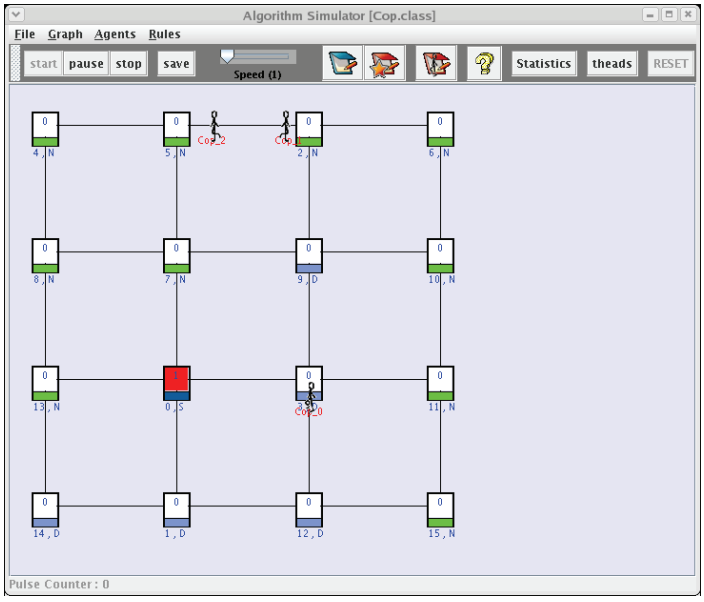


Figure 6.8: An execution using one thief and three policemen with *ViSiDiA*

- in the message passing model, it takes no time for a node to do any local computation.
- in the mobile agent model, it takes one time unit to write/read the white-board of a node.
- in the mobile agent model, each time unit, only one agent can write/read the white-board of a node.

ViSiDiA allows the user to encode and to simulate a synchronous algorithm under any combination of the previous assumptions. The general idea is to write an algorithm in many blocks, each of them takes one time unit. No matter what kind of instructions the user writes in a block, the only restriction is that the instructions of a block take one time unit in the theoretical model.

The most important feature of the synchronous api is the **nextPulse()** method. In fact, when the user invokes this method in his algorithm, the corresponding computation entity (node or agent) is blocked until all the other entities invoke the **nextPulse()** method. In practice, the user must simply insert the **nextPulse()** method in his algorithm each time he writes a sequence of instructions which takes one time unit in his theoretical synchronous model. In other words, any block of instructions encapsulated by two calls to the **nextPulse()** method takes one time unit.

Remark 6.4.1 *The methods provided by ViSiDiA for sending and receiving messages in the synchronous model are essentially the same as in the asynchronous case from a syntactical point of view. Nevertheless, their actions are fundamentally different. In fact, assume a synchronous model and suppose that it takes one time unit to send a message. Suppose that at some pulse t, a node v sends a message to a node u and then v invokes the* **nextPulse()** *method, i.e., the node declares that it is safe for the current pulse t. In this case, the synchronous api of ViSiDiA guarantees that*

- *the message is visualized before pulse t is finished, that is before node u enters pulse $t+1$.*
- *the message is delivered to u only at the beginning of pulse $t+1$. In other words, if node u executes the* **receiveFrom()** *method in pulse t, then it can not read the message sent by v because it has not yet arrived to destination.*

This remark is crucial for the correctness of the simulation and the visualization architecture of the synchronous version of ViSiDiA.

In Fig. 6.9, we give an implementation of the classical synchronous message passing BFS tree algorithm using the api of *ViSiDiA*. The algorithm is a direct consequence of the FLOOD algorithm (Fig. 6.2) in a synchronous model where each time unit a node can send a message to all its neighbors.

```
public class SynchBFS extends SyncAlgorithm {
  public void init() {
    boolean run = true;
    int degree = getArity();
    String label = (String)getProperty("label");
    while(run) {
      if(label.equals("R")) { // the node is the root
        sendAll(new StringMessage("WAVE",wave));
        run = false;
        nextPulse();
      } else {
        if(anyMsg()) { // check if any message is arrived
          Door door = new Door();
          StringMessage msg = (StringMessage)receive(door);
          // mark my father in the tree
          setDoorState(new MarkedState(true),door.getNum());
          putProperty("label", new String("T"));
          // broadcast the message to neighbors
          sendAll(new StringMessage("WAVE",wave));
          run = false;
        }
        nextPulse();
      }
    }
  }
}
```

Figure 6.9: Synchronous message passing BFS tree: code for a node with *ViSiDiA*

6.5 The distributed version of *ViSiDiA*

6.5.1 General idea

In order to allow experimentations on very huge graphs, we have developed a distributed version of *ViSiDiA* . This new version is based on distributing the threads which execute an algorithm on a real network of machines. More precisely, the visualization of an algorithm takes place on the local host of the user whereas the simulation is distributed over many machines. In the following paragraphs, we outline the architecture of the distributed version without going into technical details.

Let us consider a *simulation graph* on which we want to simulate a distributed algorithm. Typically, the simulation graph is the graph drawn by the user on the GUI. Suppose that we have a real network of machines that we model using a *network graph*. The general idea of the distributed version is to decompose the nodes of the simulation graph into clusters and to make the nodes of each cluster run on one machine of the real network. In Fig. 6.10, we give a concrete example of a network graph, a simulation graph and the corresponding decomposition.

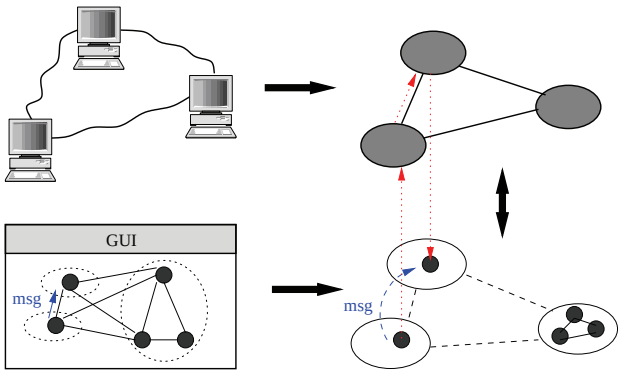


Figure 6.10: The general idea of the simulation in the distributed version of *ViSiDiA*

6.5.2 General architecture

We recall that, in the centralized version of *ViSiDiA*, the simulation is controlled by a *simulator* and the threads corresponding to the simulation graph run on the local host of the user. In particular, the simulator allows the threads to send messages to their neighbors.

In the distributed version, there is no simulator and the threads corresponding to the simulation graph are completely independent of each others and do not share any centralized entity. In fact, we use a *remote object* called *remote node* in order to implement a node of the simulation graph. Each remote node is hosted by a network machine and run the distributed algorithm using a java thread in a real autonomous way. In particular, each node aims a set of references (computed at the beginning of the simulation) and allowing it to contact its neighbors. The general data structure of a remote object corresponding to a node is depicted in Fig. 6.11. In practice, a node can be viewed as an object that acts some times like a "server" and some times like a "client". The communication between the nodes is ensured using java remote method invocations RMI (see, e.g., [RD00, Sun]).

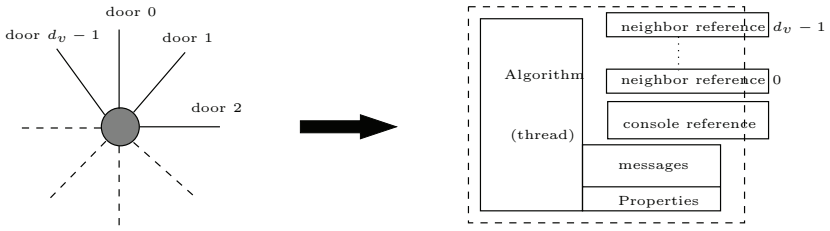


Figure 6.11: Remote node structure

For the visualization, we use a remote object called the *console*. The console (and the GUI) run on the local-host of the user. Each time a remote node makes an action which needs to be visualized, it generates an event and transmits it to the console. Then, the console can perform the actions needed for the visualization.

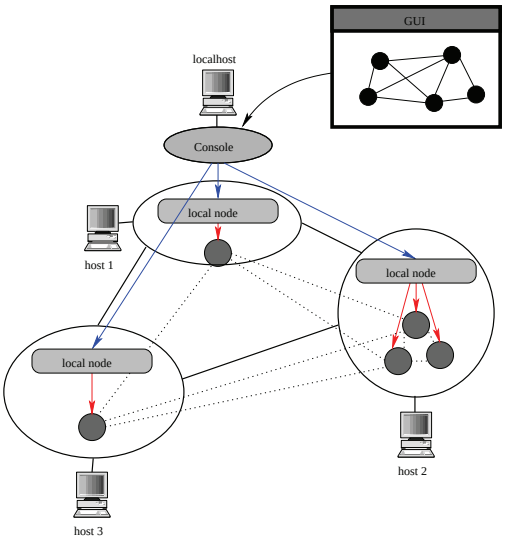


Figure 6.12: A general view of the remote objects in the distributed version of *ViSiDiA*

In order to dynamically create the remote nodes corresponding to the simulation graph, we use another remote object called *remote local node*. The user must create at least one remote local node per machine of the network graph. Each remote local node is responsible for creating a group of remote nodes. Before starting a simulation, the user must *configure ViSiDiA* by providing the name of the remote local nodes participating in the simulation and the way the remote nodes are distributed over the network.

Generally speaking, the local nodes and the console allow the user to supervise the simulation. When the user starts a simulation using the distribued version of *ViSiDiA*, the console contacts sequentially each remote local node and asks it to create as many remote nodes as needed. Once the remote nodes are created on each machine, they are intialized in order to run autonomously. In particular, we initialize the references (see Fig. 6.11) enabling them to communicate with their neighbors. At any moment of the simulation, the console and the remote local nodes can collect informations about the simulation. But, they do not interfere with the computations and the communications done by the remote nodes.

In Fig. 6.12, we give a general view of the distributed architecture of *ViSiDiA* (a simulation graph, a three-machine network graph, the console running on the local-host, three remote local nodes, seven remote nodes distributed according to the decomposition of Fig 6.10).

Remark 6.5.1 *The distributed version of* ViSiDiA *can be run using one machine. In this case, all the remote objects are created on the same machine.*

Remark 6.5.2 *In practice, the api provided by the distributed version is essentially the same as the centralized one and any algorithm can be immediately transformed to run on the distributed version by simply inheriting from the* ViSiDiA *DistAlgorithm class. Nevertheless,*

the distributed version allows to implement distributed algorithms only in the asynchronous message passing model.

6.5.3 Performance

The goal of the distributed version of *ViSiDiA* is to make experimentations of distributed algorithms using very huge graphs. Typically, the user should use the centralized version in order to test the correctness of an algorithm. Then, he should use the distributed version in order to get relevant experimental results using huge simulation graphs.

The performance of the distributed version depends heavily on the properties of the real network. In fact, if we use a larger number of machines or more powerful machines for the real network, then we are able to create more threads corresponding to the simulation graph, and thus to make experimentation on larger graphs. In practice, we think that it is always possible to find the resources required to create as many threads as necessary for a simulation. Hence, the distributed version of *ViSiDiA* is tailored for experimentation of distributed algorithms on very huge graphs.

The current implementation of the distributed version of *ViSiDiA* improves significantly the performance of the centralized one. Typically, it allows a simulation on a graph (grid) of 20000 nodes, using a network of five machines (Pentium III Bi-PRO 32 Mo), whereas the centralized version allows a simulation on a grid of at most 5000 nodes (on a localhost Pentium III Bi-PRO 256 Mo). Our goal in the future is to run a simulation on a graph having some millions of nodes. Since, the *ViSiDiA* architecture is perfectly adapted to such a simulation, the main remaining challenge is to improve the GUI of *ViSiDiA* in order to support such huge graphs. We are currently working on it.

Remark 6.5.3 *The decomposition (of the simulation graph) implied by the distributed version can also be of a theoretical interest. In the current implementation, we simply allow the user to decompose the simulation graph by hand using the GUI, or to randomly distribute the nodes of the simulation graph over the real network. It would be very interesting to design and implement more sophisticated decomposition algorithms.*

6.6 Future works

The ultimate ambition behind the *ViSiDiA* project is to provide a reference platform allowing researchers and students to implement and test distributed algorithms. We are continuing our efforts in order to attain this goal. In our future works, we plan to add the following features:

1. Add new functionalities to *ViSiDiA* in order to simulate distributed algorithm using weighted graphs. In fact, at present, the data structure corresponding to a graph in *ViSiDiA* implements only unweighted graphs. The user has to write an extra code in

his algorithm in order to assign weights to edges in a consistent manner. In addition, the visualization of the weights of edges is not taken into account yet.

2. Allow the user to control the execution of a distributed algorithm by implementing his own process manager. Typically, the user should be able to speed up the execution of a thread modeling a given node or even to give priority to the threads corresponding to each node of the graph. This allows the user to observe the behavior of a distributed algorithm when assuming different processor speeds. One can also think of a graph structure which allows the user to control the speed of messages for each edge. This can be very precious when studying some highly asynchronous distributed algorithms.

3. Find a friendly way to simulate and analyze distributed protocols using precomputed data structures. For instance, many rooting algorithms use a precomputed data structure in order to describe a rooting scheme. In general, the rooting scheme is local, i.e., that is it relies only on local information stored in the message and/or in the corresponding node. It would be very interesting to provide an api for generating the precomputed data structure in an easy way and to focus on the rooting scheme itself.

4. Find new ideas in order to handle the simulation on very huge graphs, i.e., about 10^6 nodes. There are at least two possible solutions to achieve our goal. The first one is to improve our GUI. The second one is to create on the fly the data structure needed for the distributed simulation by efficiently parsing a text file modeling the graph, instead of using the GUI for launching the simulation. The goal here is no more to visualize but to get relevant experimental results.

5. Improve the experimentation module of *ViSiDiA*. For instance, *ViSiDiA* allows to run automatically an algorithm many times and to collect the total number of messages exchanged by nodes at each execution using a java window. Thus, the user can for example compute the average message complexity of an algorithm. The user can also collect other experimental results by writing extra-instructions in the java code of the distributed algorithm to be tested. It would be interesting to enable the user to configure *ViSiDiA* in order to collect any kind of experimental information *using the GUI*. The goal here is to allow researchers to get experimental results in a *friendly* way.

Conclusion

Les travaux présentés dans cette thèse sont de natures différentes. Ils partagent cependant un objectifs commun: celui d'une meilleure compréhension des aspects locaux de l'algorithmique distribuée. Ces aspects locaux se sont exprimés en premier lieu lors de la conception d'algorithmes efficaces en temps. Ainsi, nous avons proposé de nouveaux algorithmes efficaces qui permettent de construire des *structures locales de graphes* couramment utilisées en calculs distribués. D'abord, la construction de *partitions* a permis de mettre en lumière quelques problèmes typiques en algorithmique distribuée, et de mettre en place les techniques adéquates pour les résoudre de façon efficace. Ensuite, nos travaux sur *la localité des sous graphes couvrants (spanners)* ont permis d'exposer de nouveaux types de spanners ayant de nouvelles propriétés. Nous avons proposé une technique efficace, en terme de temps, pour *casser la symétrie* qui apparaît inévitablement lors de la résolution de problèmes distribués. Ces différents travaux ont aussi l'intérêt d'ouvrir de nouvelles perspectives et de poser de nouvelles questions notamment sur l'amélioration de la performance des algorithmes et des structures présentés. Il serait par exemple très intéressant de pouvoir construire des spanners optimaux de façon déterministe et en temps poly-logarithmique.

Les aspects locaux se sont ensuite exprimés dans un cadre plus *formel*, celui de *la conception d'algorithmes distribués* de façon *abstraite et unifiée* avec *les systèmes de réétiquetages*. L'utilisation même des systèmes de réétiquetages et leur adéquation pour encoder des algorithmes distribués découlent naturellement de cet aspect local des calculs distribués. Les travaux que nous avons menés dans ce cadre montrent la puissance des systèmes de réétiquetages dans l'encodage et la modélisation d'algorithmes distribués. Nos observations montrent aussi qu'il reste encore un long travail de formalisation rigoureuse pour arriver à proposer une méthode générale et automatique pour la description d'algorithmes distribués avec des systèmes de réétiquetages. Nous pensons que nos travaux futurs dans ce cadre peuvent donner lieu à de jolis résultats aussi puissants qu'inattendus.

Les *agents mobiles* n'ont pas été moins révélateurs des aspects locaux propres à certains algorithmes distribués. En effet, les algorithmes que nous avons proposés pour le problème de *Handshake* permettent d'obtenir des résultats surprenants par rapport à des solutions qui se basent sur l'échange de messages. Dans ce sens, les agents mobiles peuvent changer la vision que nous avons de certains problèmes et nous aider à trouver de nouvelles idées performantes

et compréhensibles. Les liens que possède le problème du Handshake avec d'autres problèmes tel que le calcul d'ensemble d'arêtes indépendantes et la coloration laisse le champ libre pour de nouveaux travaux. Nous sommes confiants que ces travaux mènerons à de nouveaux résultats.

Finalement, *la plateforme ViSiDiA* a été d'une aide précieuse dans beaucoup de travaux présentés dans cette thèse. À titre d'exemple, les tests que nous avons effectués dans le cadre du problème du Handshake nous ont confortés dans notre intuition théorique et nous ont encouragés à pousser le raisonnement théorique encore plus loin. Nous espérons que cette plateforme sera utilisée de plus en plus massivement par les étudiants en algorithmique distribuée mais aussi par les chercheurs. Pour cela, nous poursuivons nos développements, nos améliorations et nos collaborations avec d'autres chercheurs dans le but de proposer un outil de référence.

Appendix A

Neighborhood Covers and Network Synchronizers

Neighborhood covers can be though as a generalization of the basic partition of Chapter 1. In fact, for any positive integer ρ, a ρ-neighborhood cover of a graph G can be defined as a collection of clusters $\mathcal{C} = \cup C$ such that for every $v \in V$, there exists a cluster $C \in \mathcal{C}$ such that $\mathcal{N}_\rho(v) \subseteq C$ where $\mathcal{N}_\rho(v) = \{u \in V \mid d(u,v) \leqslant \rho\}$ denotes the ρ-neighborhood of node v in the graph G. For instance, the basic partition described before is a 0-neighborhood cover of G. It is not too difficult to see that it is possible to extend our basic partion algorithms in order to construct ρ-neighborhood covers with good properties.

In addition, network covers can be used as a data structure for network synchronizers. In order to design the efficient network synchronizers given in [MS00], we just need a 1-neighborhood cover. Therefore, in next paragraphs, we only show how to extend the algorithms of Chapter 1 for $\rho = 1$ and we briefly outline the main modifications to be done

A.1 Distributed construction of 1-neighborhood covers

In this section, we extend algorithm Dist_Part in order to cover the 1-neighborhood of each node. We use the same distributed techniques as in Chapter 1 to manage cluster growth. However, we make a cluster explore two layers at the same time instead of only one. At each new exploration, each cluster fights in order to maintain two layers l_i and l_{i+1} with i the radius of the cluster. The first layer l_i allows the cluster to compute the sparsity condition (the same one than in algorithm Dist_Part). The second layer l_{i+1} (which is the last explored one) guarantees that the neighborhoods of all nodes in layer l_i are in the current cluster. There are mainly five important modifications to do:

1. At the beginning of the algorithm, all nodes are orphans. An orphan node first explores *two consecutive* layers before starting computing the sparsity condition.

2. If the sparsity condition is satisfied for layer l_i, then a cluster begins a new exploration, i.e., the leaves in layer l_{i+1} try to invade new nodes. If the new exploration succeeds, then layer l_{i+1} becomes layer $l_{i'=i+1}$ and the new explored layer becomes the new $l_{i'+1}$ layer.

3. If the sparsity condition for layer l_i is not satisfied, then the construction of the cluster is finished. The finished cluster contains not only all layers $l_{j<i}$ but also the two layers l_i and l_{i+1}. Nevertheless, only nodes in layers $l_{j<i}$ are in a final state. The 1-neighborhoods of all nodes in layer l_i are covered by the finished cluster but they do not stop computing yet. In fact, the 1-neighborhoods of nodes in layer l_{i+1} may not be covered by a cluster. Hence, nodes in layer l_i become orphan clusters with identity $-\infty$ in order to allow other clusters to grow and cover the neighborhoods of nodes in layer l_{i+1}. On the other side, nodes in layer l_{i+1} become orphan clusters with their initial identifiers and continue competing in order to grow new clusters.

4. If a new exploration fails, i.e., there is a cluster at distance 1 or 2 (from layer l_{i+1}) with a bigger identifier, then:

   - either the winner lost against another neighboring cluster and the current cluster is not invaded. Hence, the cluster simply retries a new exploration.
   - or the current cluster is invaded and the cluster looses its last layer l_{i+1}. Hence, invaded nodes in layer l_{i+1} become part of the last layer $l_{i_{win}+1}$ of the winner cluster. Nodes in layer l_{i+1} which have not been invaded become orphan nodes and begin a new exploration using their own identifiers. Layer l_i becomes the last layer $l_{i'+1=i}$ and layer l_{i-1} becomes layer $l_{i'=i-1}$. Then the cluster begins a new exploration once again.

5. When the construction of a cluster is finished, nodes at distance at least 2 from the border of the cluster, i.e., layers $l_{j\leqslant i-1}$, switch to final states. In fact, layer l_{i-1} of a finished cluster acts as a barrier that protects the finished cluster from future invasions. Layers $l_{j\leqslant i-1}$ are usually called the *Kernel* of the cluster.

It is easy to see that the time and message complexity of the extended algorithm increases by only a constant factor due to the computation of the extra layer l_{i+1}. Notice also that it is not difficult to adapt the techniques of algorithms FAST_PART and ELECT_PART in order to obtain sublinear time complexity.

An example of cluster growth

In Fig. A.1, we give an example of how the cover is constructed. In our example, there are four active clusters: 1, 2, 3 and 4 with identities $Id_1 > Id_2 > Id_3 > Id_4$. We suppose that there is a finished cluster in the neighborhood of cluster 1.

The nodes in layer l_i of the finished cluster (first part of Fig. A.1) still participate in the computation with identity $-\infty$, all the nodes in the Kernel of the finished cluster are in a final state. There is also a node in layer l_{i+1} of the finished cluster which belongs to layer l_{i+1} of cluster 1.

Suppose that the layers l_i of the active clusters satisfy the sparsity condition, then these clusters will try to grow. Cluster 2 can not grow because cluster 1 is at distance two of it and has a bigger identity. Cluster 1 will invade both clusters 3 and 4. Cluster 4 is orphan and it simply joins the last layer of cluster 1. Cluster 3 will lose its last layer l_{i+1}. The invaded nodes of cluster 3 join cluster 1 and the other nodes which have not been invaded become orphan clusters (second part of Fig. A.1). Note also that the node with identity $-\infty$ in the finished cluster is invaded by cluster 1. This guarantees that the neighborhood of the children of the $(-\infty)$-node in the finished cluster is covered by cluster 1.

Once the new exploration is finished, cluster 1 verifies the sparsity condition. If it is satisfied, a new exploration will begin and clusters 2 and 3 will be invaded. Note that nodes in the Kernel of the finished cluster will not be invaded by cluster 1. If the sparsity condition is not satisfied which is the case in the third part of Fig. A.1, the construction of cluster 1 is finished. The nodes in layer $l_{i'}$ become orphans with identity $-\infty$. The nodes in layer $l_{i'+1}$ become orphan nodes except those which are already in layer l_i of another finished cluster (those whose neighborhoods are covered). Note that the two finished clusters we have constructed overlap (they have a common edge).

A.2 Application to network synchronizers

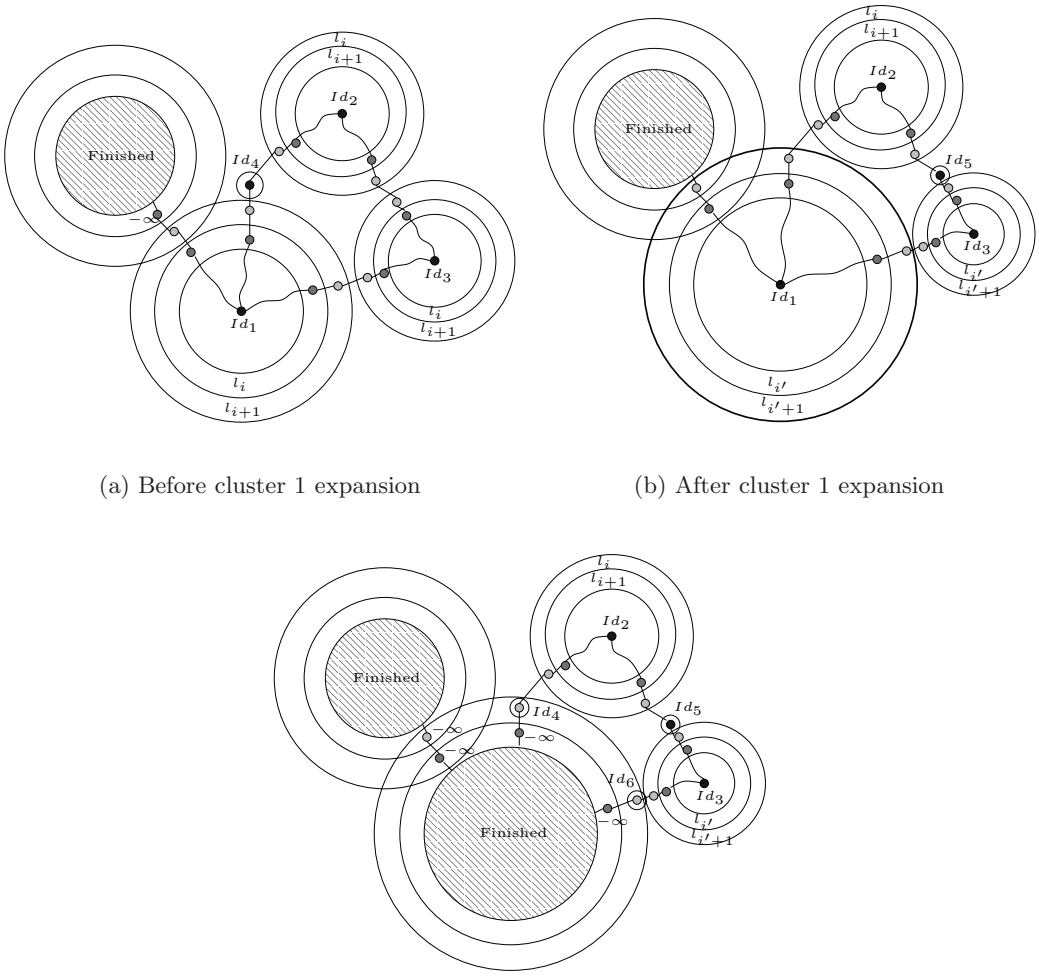
The basic partition of Chapter 1 and the 1-neighborhood covers constructed in previous sections are of special interest for designing network synchronizers γ, γ_1 and γ_2 [MS00]. In the following, we review the basic properties of these synchronizers.

Background

Network synchronizers allow to transform a synchronous algorithm into an asynchronous one. In general, one prefers to design a distributed algorithm in a synchronous model rather than an asynchronous model which is typically harder to grasp and to analyze ([Pel00], Chapter 6). From a practical point of view, network synchronizers provide a uniform methodology for transforming synchronous distributed algorithms in asynchronous ones.

Generally speaking, the basic idea of network synchronizers is to simulate a global clock by using *local pulse generators*. If the local clock pulse of some node v is equal to p, then node v knows that the messages that it has sent at pulse $p-1$ have reached their destinations. Many simulation techniques were developed in order to guarantee this property:

1. The first basic technique is known as synchronizer α. The general idea of synchronizer



(a) Before cluster 1 expansion

(b) After cluster 1 expansion

(c) After construction of cluster 1 is finished

Figure A.1: An example of a cluster expansion for cover needed for γ_2

α is to send an acknowledgment corresponding to each received message of the original synchronous algorithm. Once, a node receives an acknowledgment of all the original messages corresponding to one pulse, the node informs its neighbors and then it generates the next pulse. This technique leads to an overhead of $O(|E|)$ messages in order to simulate a pulse. Assuming that a message delay is at most $O(1)$ (this is only for performance analysis), it also leads to the theoretical $O(1)$ time overhead per pulse.

2. The second basic technique called synchronizer β assumes a precomputed rooted BFS spanning tree T of G. Only the root of T have a pulse generator which controls all other nodes. In fact, once a node u receives the acknowledgments of the messages it has sent, the node u is ready for the next pulse and it informs its parent in the tree T. The

parents forward this information until it reaches the root of T. Once the root learns that all the nodes are ready for the next pulse, it broadcasts a message saying *"it is time for the next pulse"*! Thus, synchronizer β implies an overhead of $O(|V|)$ messages and $O(D)$ time per pulse, where D is the diameter of G.

3. The third technique is an intermediate technique which provides a good time-message trade-offs. This technique implies three synchronizers γ, γ_1 and γ_2 ([MS00]). All of these three synchronizers use sparse covers. More precisely, synchronizer γ uses the basic partition as a data structure. Synchronizer γ_1 uses a cover based on the basic partition where each edge belongs to at least one cluster. This modified partition can be easily constructed by our algorithms by simply marking the last rejected layer as part of the cluster. Finally, synchronizer γ_2 uses the the 1-neighborhood cover described in the previous section.

A detailed description of synchronizers γ, γ_1 and γ_2 can be found in [MS00]. In the following, we just outline the basic ideas used in synchronizer γ (the two other synchronizers are based on the same general ideas). First, we assume the following:

1. A partition $\mathcal{C}$ of G is constructed.

2. A rooted BFS spanning tree T_C for each cluster $C \in \mathcal{C}$ is constructed.

3. A set $\mathcal{I}$ of intercluster edges is selected.

In order to simulate a pulse, we combine the techniques of synchronizers α and β. Roughly speaking, the root of each tree T_C first waits to learn that the nodes in C are ready for the next pulse (which costs $O(|C|)$ messages and $O(Rad(C))$ time for each cluster). Then, the cluster tries to synchronize with its neighbors using the intercluster edges. More precisely, the root of C broadcasts a notification message all along the tree T_C saying that all the nodes in its cluster are ready. When the leaves of T_C receive the notification message, they forward it to neighboring clusters using the selected intercluster edges (which costs $O(|\mathcal{I}|)$ message and $O(1)$ time). Symmetrically, the leaves receive a notification from their neighboring clusters. When receiving such a notification, they send it back to their root. Once the root receives the notification messages of its neighbors, it sends a message to the nodes in its cluster saying "it is time for the next pulse". Thus, the global overhead is $O(n + |\mathcal{I}|)$ messages and $O(Rad(\mathcal{C}))$ time per pulse.

Thus, if we take the basic partition as a data structure, then the global overhead is $O(n^{1+1/k})$ messages and $O(k)$ time per pulse which gives a good compromise comparing with synchronizer α and β.

Contribution

The previous overhead is essentially optimal according to Lemma 25.1.7 in Peleg's book [Pel00]. Synchronizers γ, γ_1 and γ_2 have the same performances up to a constant factor. Hence, the remaining challenge was to improve the pre-processing step of constructing the required graph data structure. Using our algorithms, the time complexity of this pre-processing step is reduced from $O(n)$ in previous implementations to $O(n^{1-1/k})$.

Appendix B

Case Study: Circulant Graphs

In this chapter, we study the efficiency of our sublinear partition algorithms (FAST_PART and ELECT_PART) described and analyzed in Chapter 1 in the case of *Circulant Graphs*. In fact, Circulant Graphs are *dense* enough to be interesting for the algorithm we are studying. They have enough large diameter in order to let the analysis non trivial and constructive. In addition, the analysis given here is interesting from a theoretical point of view. In particular, the proofs of theorem B.0.6 illustrate the improvements discussed in Section 1.6.4.

Definition B.0.1 *A circulant graph* $\mathcal{C}ir_n(\mathcal{L})$ *is a graph of* n *nodes* $\{1, 2, ..., 3\}$ *in which a vertex* i *is adjacent to nodes* $(i-j)$ *and* $(i+j)$ *for each* i *and* j *in the list* $\mathcal{L}$ *(see Fig. B.1 for an example).*

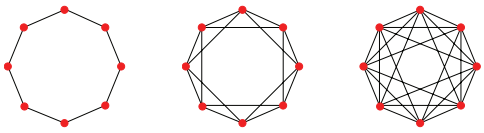


Figure B.1: An Example of $\mathcal{C}ir_n(\mathcal{L})$ graphs with n=8 and $\mathcal{L} \in \{\{1\}, \{1, 2\}, \{1, 2, 3\}\}$

Definition B.0.2 *For every parameter* ϵ *such that* $0 < \epsilon \leqslant 1$*, we define the graph* $\mathcal{C}ir_n^\epsilon$ *to be the circulant graph* $\mathcal{C}ir_n(1, 2, ..., \lfloor \frac{n^\epsilon}{2} \rfloor)$.

In the sequel, we suppose that $\epsilon > \frac{1}{k}$. In fact, if $\epsilon \leqslant \frac{1}{k}$ then the graph is already sparse and all our algorithms terminate in $O(1)$ rounds.

Theorem B.0.3 *For* $k < \log(n)$ *and for every graph* $\mathcal{C}ir_n^\epsilon$*, the time complexity of algorithm* FAST_PART *is bounded by* $O(n^{1-\epsilon})$.

Proof For $k < \log(n)$, any constructed cluster has radius at most 1. It can also be shown that $\Lambda \leqslant 2\,\frac{n}{n^\epsilon} = O(n^{1-\epsilon})$. Thus, the theorem follows as a consequence of Theorem 1.5.8. ∎

Theorem B.0.4 *Let T be the time complexity of algorithm* ELECT_PART. *Then, for every graph $\mathcal{C}ir_n^\epsilon$, the expected value of T satisfies:*

$$\mathbb{E}(T) = O(k^3 \log(n)\, n^\epsilon)$$

Proof For any graph $\mathcal{C}ir_n^\epsilon$, it is easy to show that $K = 2k\, n^\epsilon$ (we recall that K is an upper bound of the $2k$-neighborhood of any node). Thus, $\log(1 - \frac{1}{K}) \leqslant -\frac{1}{K} = -\frac{1}{2kn^\epsilon}$ and the result follows immediately from Theorem 1.6.3. ∎

The two theorems B.0.3 and B.0.4 are immediate consequences of the analysis we have already made for algorithms SYNC_PART and ELECT_PART. In particular, we obtain a time complexity which is better than $O(n^{1-\frac{1}{k}})$. Nevertheless, using a more careful analysis, we obtain the following bounds:

Theorem B.0.5 *For every graph $\mathcal{C}ir_n^\epsilon$, the expected time complexity T of algorithm* ELECT_PART *satisfies:*

$$\mathbb{E}(T) = O\left(k^3 log(n) + kn^{\frac{1}{k}}\right)$$

Theorem B.0.6 *Using the improved version of algorithm* ELECT_PART *described in Section 1.6.4, the expected time complexity T of algorithm* ELECT_PART *satisfies:*

$$\mathbb{E}(T) = O(k^3 \log(n))$$

Proof We prove the previous two theorems in two parts. The first part is common to the two theorems. The technical arguments are similar to those in the analysis made in Theorem 1.6.3 but the reasoning is different and gives an idea about the proof of the improvements of Section 1.6.4.

First Part of the proof Let $i \geqslant 0$ be a phase of algorithm ELECT_PART and $(G_i)_{i \geqslant 0}$ be the sequence of graphs such that $G_0 = G$ and for all $i \geqslant 1$, G_i is the graph obtained by removing the nodes belonging to a finished cluster from G_{i-1}.
Let V_i be the set of nodes having a degree higher than $\lfloor \frac{n^\epsilon}{2} \rfloor$ in phase i. Let X_i be the random variable which denotes the number of nodes in V_i, and let Y_i be the number of nodes from V_i which are locally k-elected in the i^{th} step. The following inequality holds:

$$\mathbb{E}\left(Y_i \mid G_i\right) \geqslant \frac{X_i}{K},$$

One can show that if the node v belongs to V_i, then every active neighbor w of v is also in V_i. Hence, we can state the following:

$$\begin{aligned}\mathbb{E}\left(X_{i+1} \mid G_i\right) &\leqslant X_i - \mathbb{E}\left(Y_i \mid G_i\right)\frac{n^\epsilon}{2} \\ &\leqslant X_i(1-\tfrac{n^\epsilon}{2K}) \\ &\leqslant X_i(1-\tfrac{1}{2\cdot 2k}).\end{aligned}$$

By induction and using the same arguments as in Theorem 1.6.3, the expected time such that $X_i = 1$ is bounded by:

$$O(k^2\frac{\log(n)}{\log(\frac{4k}{4k-1})})$$

Let us consider the time after which all nodes in the graph have a degree less than $\lfloor\frac{n^\epsilon}{2}\rfloor$, i.e., the time such that $V_i = 0$) . One can show that the remaining nodes are grouped in many connected fragments that can be divided in two types: dense components with more than $n^{\frac{1}{k}}$ nodes and sparse components with no more than $n^{\frac{1}{k}}$ nodes. All theses components are disjoint and do not share any node. Thus, the algorithm runs independently on each component.

Let us consider a dense component C_d, i.e., $n^{\frac{1}{k}} < |C_d| < \lfloor\frac{n^\epsilon}{2}\rfloor$). In one phase, there will be exactly one elected node in C_d and the finished cluster constructed around this node will contain the whole component C_d. Thus, in $O(k)$ time, all nodes in C_d become finished.
Second part of the proof of *Theorem B.0.5*: Let us consider a sparse component C_s, i.e., $|C_s| \leqslant n^{\frac{1}{k}}$). The nodes of such a component have a degree less than $n^{\frac{1}{k}}$. At each phase of algorithm Elect_Part, there will be exactly one elected node in C_s which forms a finished single node cluster. Thus, we need at most $O(n^{\frac{1}{k}})$ phases of $O(k)$ time units each before all nodes in C_s become finished.

To conclude, if V_i becomes empty then we need at most $O(kn^{\frac{1}{k}})$ time units before the algorithm terminates and Theorem B.0.5 holds.

Second Part of the proof of *Theorem B.0.6*: Let us consider a sparse component C_s, i.e., $|C_s| \leqslant n^{\frac{1}{k}}$). The nodes of such a component have a degree less then $n^{\frac{1}{k}}$. Thus, using from the improvements of algorithm Elect_Part in Section 1.6.4, these nodes are allowed to be finished. Hence, in $O(1)$ time, they all become finished single node clusters.

To conclude, if V_i becomes empty then we need at most $O(k)$ time units before the algorithm terminates and Theorem B.0.6 holds. ∎

Bibliography

[AAER05] D. Angluin, J. Aspnes, D. Eisenstat, and E. Ruppert. On the power of anonymous one-way communication. In 9^{th} *conf. on Principles of Distributed Computing (PODC05)*, pages 307–318, 2005.

[ABCP93] Baruch Awerbuch, Bonnie Berger, Lenore J. Cowen, and David Peleg. Near-linear cost sequential and distributed constructions of sparse neighborhood coverss. In 34^{th} *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 638–647. IEEE Computer Society Press, November 1993.

[ABCP96] Baruch Awerbuch, Bonnie Berger, Lenore J. Cowen, and David Peleg. Fast distributed network decompositions and covers. *Journal of Parallel and Distributed Computing*, 39:105–114, 1996.

[ABCP98] Baruch Awerbuch, Bonnie Berger, Lenore J. Cowen, and David Peleg. Near-linear time construction of sparse neighborhood covers. *SIAM Journal on Computing*, 28(1):263–277, February 1998.

[ABNLP89] Baruch Awerbuch, Amotz Bar-Noy, Nathan Linial, and David Peleg. Compact distributed data structures for adaptive routing. *CWI Quarterly*, 2(4):277–305, 1989.

[ABNLP90] Baruch Awerbuch, Amotz Bar-Noy, Nathan Linial, and David Peleg. Improved routing strategies with succinct tables. *Journal of Algorithms*, 11(3):307–341, 1990.

[AGLP89] B. Awerbuch, A. V. Goldberg, M. Luby, and S. A. Poltkin. Network decomposition and locality in distributed computation. 30^{th} *IEEE Symposium on Foundation of Computer Science (FOCS89)*, pages 364–369, 1989.

[Ang80] D. Angluin. Local and global properties in networks of processors. In 12^{th} *Symposium on Theory Of Computing (STOC80)*, pages 82–93, 1980.

[AP90a] B. Awerbuch and D. Peleg. Network synchronization with polylogarithmic overhead. 31^{st} *IEEE Symposium on Foundations of Computer Science (FOCS90)*, 2:503–513, October 1990.

[AP90b] Baruch Awerbuch and David Peleg. Sparse partitions. In 31^{th} *Symposium on Foundations of Computer Science (FOCS90)*, pages 503–513. IEEE Computer Society Press, October 1990.

[AP92] B. Awerbuch and D. Peleg. Routing with polynomial communication-space trade-off. *SIAM Journal on Discrete Mathematics*, 5:151–162, 1992.

[AP95] B. Awerbuch and D. Peleg. Online tracking of mobile users. *Journal of the ACM*, 42:1021–1058, 1995.

[AR93] Y. Afek and M. Ricklin. Sparser : a paradigm for running distributed algorithms. *Journal of Algorithms*, 14:316–328, 1993.

[AS92] Noga Alon and Joel H. Spencer. *The Probabilistic Method*. John Wiley & Sons, 1992.

[Awe85] B. Awerbuch. Complexity of network synchronization. *Journal of the ACM*, 32:804–823, 1985.

[Awe87] Baruch Awerbuch. Optimal distributed algorithms for minimum weight spanning tree, counting, leader election and related problems. In 19^{th} *ACM Symposium on Theory of Computing (STOC87)*, pages 230–240. ACM Press, May 1987.

[BA01] Mordechai Ben-Ari. Interactive execution of distributed algorithms. *Journal on Educational Resources in Computing (JERIC)*, 1(2):Article No. 2, 2001. http://stwww.weizmann.ac.il/g-cs/benari/daj/index.html.

[BBCD02] F. Belkouch, M. Bui, L. Chen, and A. K. Datta. Self-stabilizing deterministic network decomposition. *Journal of Parralel and Distributed Computing*, 62:696–714, 2002.

[BFFS03] L. Barrière, P. Flocchini, P. Fraigniaud, and N. Santoro. Can we elect if we cannot compare? In 15^{th} *ACM Symposium on Parallel Algorithms and Architectures (SPAA03)*, pages 324–332, 2003.

[BGS05] Surender Baswana, Vishrut Goyal, and Sandeep Sen. All-pairs nearly 2-approximate shortest-paths in $O(n^2\text{polylog}n)$ time. In 21^{st} *Symposium on Theoretical Aspects of Computer Science (STACS05)*, volume 3404 of Lecture Notes in Computer Science, pages 666–679. Springer, 2005.

[BKMP05] Surender Baswana, Telikepalli Kavitha, Kurt Mehlhorn, and Seth Pettie. New constructions of (α, β)-spanners and purely additive spanners. In 16^{th} *Symposium on Discrete Algorithms (SODA05)*, pages 672–681. ACM-SIAM, January 2005.

[BS03] Surender Baswana and Sandeep Sen. A simple linear time algorithm for computing a $(2k-1)$-spanner of $O(n^{1+1/k})$ size in weighted graphs. In 30^{th} *International Colloquium on Automata, Languages and Programming (ICALP03)*, volume 2719 of Lecture Notes in Computer Science, pages 384–396. Springer, July 2003.

[BS04] Surender Baswana and Sandeep Sen. Approximate distance oracles for unweighted graphs in $\tilde{O}(n^2)$ time. In 15^{th} *Symposium on Discrete Algorithms (SODA04)*, pages 271–280. ACM-SIAM, January 2004.

[CFJ+03] Steve Carr, Changpeng Fang, Timothy R. Jozwowski, Jean Mayo, and Ching-Kuang Shene. Concurrentmentor: A visualization system for distributed programming education. In *The international Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA06)*, pages 1676–1682. CSREA Press, 2003. http://www.cs.mtu.edu/ shene/NSF-3/index.html.

[CGMO06] J. Chalopin, E. Godard, Y. Métivier, and R. Ossamy. Mobile agent algorithms versus message passing algorithms. In 10^{th} *International Conference On Principles Of Distributed Systems (OPODIS06)*, page to appear, 2006.

[Cha82] E. J. H. Chang. Echo algorithms: Depth parallel operations on general graphs. *IEEE Trans. Software Eng.*, 8(4):391–401, 1982.

[Cha05] J. Chalopin. Local computations on closed unlabelled edges : the election problem and the naming problem. In 31^{st} *Annual Conference on Current Trends in Theory and Practice of Informatics (SOFSEM05)*, volume 3381 of *Lecture Notes in Computer Science*, pages 82–91. Springer-Verlag, jan 2005.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill, Cambridge, MA, 1990.

[CM04] J. Chalopin and Y. Métivier. Election and local computations on edges. In *Foundations of System Specification and Computation Structures (FoSSaCS04)*, volume 2987 of *Lecture Notes in Computer Science*, pages 90–104. Springer-Verlag, mar 2004.

[CM05] J. Chalopin and Y. Métivier. A bridge between the asynchronous message passing model and local computations in graphs. In *Mathematical Foundations of Computer Science (MFCS05)*, volume 3618 of *Lecture Notes in Computer Science*, pages 212–223. Springer-Verlag, aug 2005.

[CMZ06] J. Chalopin, Y. Métivier, and W. Zielonka. Local computations in graphs: the case of cellular edge local computations. *Fundamenta informaticae*, to appear, 2006.

[Coh98] Edith Cohen. Fast algorithms for constructing t-spanners and paths with stretch t. *SIAM Journal on Computing*, 28(1):210–236, 1998.

[Cow93] Lenore J. Cowen. *On Local Representations of Graphs and Networks.* Ph. D Thesis, MIT, 1993.

[Cow01] Lenore J. Cowen. Compact routing with minimum stretch. *Journal of Algorithms*, 38(1):170–183, 2001.

[CV86] Richard Cole and Uzi Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control*, 70(1):32–53, 1986.

[CZ01] Edith Cohen and Uri Zwick. All-pairs small-stretch paths. *Journal of Algorithms*, 38:335–353, 2001.

[DHSZ04] P. Duchon, N. Hanusse, N. Saheb, and A. Zemmari. Broadcast in the rendezvous model. In 21^{st} *Symposium on Theoretical Aspects of Computer Science (STACS04)*, volume 2996 of *Lecture Notes in Computer Science*, pages 559–570, 2004.

[DHSZ06] P. Duchon, N. Hanusse, N. Saheb, and A. Zemmari. Broadcast in the rendezvous model. *Information and Computation*, 204(5):697 – 712, 2006. extended abstract published in the proceedings of 21^{st} Symposium on Theoretical Aspects of Computer Science (STACS04) (see [DHSZ04]).

[DHZ00] Dorit Dor, Shay Halperin, and Uri Zwick. All-pairs almost shortest paths. *SIAM Journal on Computing*, 29(5):1740–1759, 2000.

[EGP98] Tamar Eilam, Cyril Gavoille, and David Peleg. Compact routing schemes with low stretch factor. Research Report RR-1195-98, LaBRI, University of Bordeaux 1, 351, cours de la Libération, 33405 Talence Cedex, France, January 1998.

[EGP03] Tamar Eilam, Cyril Gavoille, and David Peleg. Compact routing schemes with low stretch factor. *Journal of Algorithms*, 46:97–114, 2003.

[Elk01] Michael Elkin. Computing almost shortest paths. In 20^{th} *ACM Symposium on Principles of Distributed Computing (PODC01)*, pages 53–62. ACM Press, 2001.

[Elk04a] Michael Elkin. A faster distributed protocol for constructing a minimum spanning tree. In 15^{th} *Symposium on Discrete Algorithms (SODA04)*, pages 359–368. ACM-SIAM, January 2004.

[Elk04b] Michael Elkin. Unconditional lower bounds on the time-approximation tradeoffs for the distributed minimum spanning tree problems. In 36^{th} *ACM Symposium on Theory of Computing (STOC04)*, pages 331–340. ACM Press, May 2004.

[EZ04] Michael Elkin and Jian Zhang. Efficient algorithms for constructing $(1+\epsilon,\beta)$-spanners in the distributed and streaming models. In 23^{rd} *ACM Symposium on Principles of Distributed Computing (PODC04)*, pages 160–168. ACM Press, July 2004.

[GKP98] J.A. Garay, S. Kutten, and D. Peleg. A sublinear time distributed algorithm for minimum-weight spanning trees. *SIAM Journal on Computing*, 27:302–316, February 1998.

[GM02] E. Godard and Y. Métivier. A characterization of families of graphs in which election is possible. In *Foundations of System Specification and Computation Structures (FoSSaCS02) (EATCS best paper award)*, pages 159–171. Lecture Notes in Computer Science 2303, Springer-Verlag, 2002.

[GM03] I. Gaber and Y. Mansour. Centralized broadcast in multihop radio networks. *Journal of Algorithms*, 46:1–20, 2003.

[GMM04] E. Godard, Y. Métivier, and A. Muscholl. Characterizations of classes of graphs recognizable by local computations. *Theory of Computing Systems*, 37:2:249–293, 2004.

[GPPR04] Cyril Gavoille, David Peleg, Stéphane Pérennès, and Ran Raz. Distance labeling in graphs. *Journal of Algorithms*, 53(1):85–112, 2004.

[GPS88] Andrew V. Goldberg, Serge A. Plotkin, and Gregory E. Shannon. Parallel symmetry-breaking in sparse graphs. *SIAM Journal on Discrete Mathematics*, 1(4):434–446, 1988.

[HKP98] M. Hanckowiak, M. Karonski, and A. Panconesi. On the distributed complexity of computing maximal matchings. In 9^{th} *Symposium on Discrete Algorithms (SODA98)*, pages 219–225, 1998.

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

[KMNW05] Fabian Kuhn, Thomas Moscibroda, Tim Nieberg, and Roger Wattenhofer. Fast deterministic distributed maximal independent set computation on growth-bounded graphs. In 19^{th} *International Symposium on Distributed Computing*

(DISC05), volume Lecture Notes in Computer Science. Springer, September 2005.

[KMW04] Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. What cannot be computed locally! In 23^{rd} *ACM Symposium on Principles of Distributed Computing (PODC04)*, pages 300–309. ACM Press, July 2004.

[KMW06] Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. The price of being near-sighted. In 17^{th} *Symposium on Discrete Algorithms (SODA06)*, pages 980–989. ACM-SIAM, January 2006.

[KP98] Shay Kutten and David Peleg. Fast distributed construction of small k-dominating sets and applications. *Journal of Algorithms*, 28(1):40–66, 1998.

[KPT03] Boris Koldehofe, Marina Papatriantafilou, and Philippas Tsigas. Integrating a simulation-visualisation environment in a basic distributed systems course: A case study using lydian. In 8^{th} *SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education (ITiCSE03)*, pages 35–39. ACM press, 2003. http://www.cs.chalmers.se/ lydian.

[KY96] T. Kameda and M. Yamashita. Computing on anonymous networks: Part i - characterizing the solvable cases. *IEEE Transactions on Parallel and Distributed Systems*, 7(1):69–89, 1996.

[LB96] Weifa Liang and Richard P. Brent. Constructing the spanners of graphs in parallel. In 10^{th} *International Parallel Processing Symposium (IPPS96)*, pages 206–210, April 1996.

[Lin87] Nathan Linial. Distributive graph algorithms - Global solutions from local data. In 28^{th} *IEEE Symposium on Foundations of Computer Science (FOCS87)*, pages 331–335. IEEE Computer Society Press, October 1987.

[Lin92] Nathan Linial. Locality in distributed graphs algorithms. *SIAM Journal on Computing*, 21(1):193–201, 1992.

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

[LMS99] I. Litovsky, Y. Métivier, and E. Sopena. Graph relabelling systems and distributed algorithms. In H. Ehrig, H.J. Kreowski, U. Montanari, and G. Rozenberg, editors, *Handbook of graph grammars and computing by graph transformation*, volume 3, pages 1–56. World Scientific, 1999.

[Lov75] Laszlo Lovász. On the ratio of optimal integral and fractional covers. *Discrete Mathematics*, 13:383–390, 1975.

[Lov96] L. LovÃ¡sz. Random walks on graphs: a survey. *Combinatorics, Paul erdos is eighty*, 2:353–397, 1996.

[LPSP01] Zvi Lotker, Boaz Patt-Shamir, and David Peleg. Distributed MST for constant diameter graphs. In 20^{th} *ACM Symposium on Principles of Distributed Computing (PODC01)*, pages 63–71. ACM Press, 2001.

[LPSPP05] Zvi Lotker, Boaz Patt-Shamir, Elan Pavlov, and David Peleg. Minimum-weight spanning tree construction in $O(\log \log n)$ communication rounds. *SIAM Journal on Discrete Mathematics*, 35(1):120–131, 2005.

[Lub86] Michael Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM Journal on Computing*, 15(4):1036–1053, November 1986.

[Lyn96] N. A. Lynch. *Distributed algorithms*. Morgan KaufmanPublishers, Inc., 1996.

[Maz97] A. Mazurkiewicz. Distributed enumeration. *Information Processing Letters*, 61:233–239, 1997.

[MMS02] Y. Métivier, M. Mosbah, and A. Sellami. Proving distributed algorithms by graph relabeling systems: Examples of trees in networks with processor identities. In *Applied Graph Transformations*, pages 45–57, Grenoble, 2002.

[MPTU98] Yoram Moses, Zvi Polunsky, Ayellet Tal, and Leonid Ulitsky. Algorithm visualization for distributed environments. In *IEEE Symposium on Information Visualization (InfoVis98)*, pages 71–78, 1998.

[MS00] Shlomo Moran and Sagi Snir. Simple and efficient network decomposition and synchronization. *Theoretical Computer Science*, 243(1-2):217–241, 2000.

[MSZ00] Y. Métivier, N. Saheb, and A. Zemmari. Randomized rendez vous. In *Mathematics and computer science: Algorithms, trees, combinatorics and probabilities*, Trends in mathematics, pages 183–194. Birkhäuser, 2000.

[MSZ02] Y. Métivier, N. Saheb, and A. Zemmari. Randomized local elections. *Information Processing Letters*, 82:313–120, 2002.

[MSZ03] Y. Métivier, N. Saheb, and A. Zemmari. Analysis of a randomized rendez vous algorithm. *Information and Computation*, 184:109–128, 2003.

[Pel00] David Peleg. *Distributed Computing: A Locality-Sensitive Approach*. SIAM Monographs on Discrete Mathematics and Applications, 2000.

[PR00] David Peleg and Vitaly Rubinovich. A near-tight lower bound on the time complexity of distributed minimum-weight spanning tree construction. *SIAM Journal on Computing*, 30(5):1427–1442, 2000.

[Pri57] Robert. C. Prim. Shortest connection networks and some generalisations. *Bell System Technical Journal*, 36:1389–1401, 1957.

[PS92] A. Panconesi and A. Srinivasan. Improved distributed algorithms for coloring and network decomposition. 24^{th} *ACM Symposium on Theory of Computing (STOC92)*, pages 581–592, 1992.

[PS96] Alessandro Panconesi and Aravind Srinivasan. On the complexity of distributed network decomposition. *Journal of Algorithms*, 20(2):356–374, 1996.

[PU88] David Peleg and Eli Upfal. A tradeoff between space and efficiency for routing tables. In 20^{th} *ACM Symposium on Theory of Computing (STOC88)*, pages 43–52. ACM Press, May 1988.

[PU89a] David Peleg and Jeffrey D. Ullman. An optimal synchornizer for the hypercube. *SIAM Journal on Computing*, 18(4):740–747, 1989.

[PU89b] David Peleg and Eli Upfal. A trade-off between space and efficiency for routing tables. *Journal of the ACM*, 36(3):510–530, July 1989.

[PV04] Lucia Draque Penso and C. Barbosa Valmir. A distributed algorithm to find k-dominating sets. *Discrete Applied Mathematics*, 141(1-3):243–253, May 2004.

[RD00] G. Roussel and E. Dusris. *Java et Internet, Concepts et programmation.* Vuibert, 2000.

[RS82] J. Reif and P. Spirakis. Real time resource allocation in distributed systems. In 1^{st} *ACM Symposium on Principles of Distributed Computing (PODC82)*, pages 84–94. ACM Press, 1982.

[RTZ02] Liam Roditty, Mikkel Thorup, and Uri Zwick. Roundtrip spanners and roundtrip routing in directed graphs. In 13^{th} *Symposium on Discrete Algorithms (SODA02)*, pages 844–851. ACM-SIAM, January 2002.

[RTZ05] Liam Roditty, Mikkel Thorup, and Uri Zwick. Deterministic constructions of approximate distance oracles and spanners. In 32^{nd} *International Colloquium on Automata, Languages and Programming (ICALP)*, volume Lecture Notes in Computer Science, 2005.

[Seg83] A. Segall. Distributed network protocols. *IEEE Transactions on Information Theory*, 29(1):23–34, 1983.

[SK93] John T. Stasko and Eileen Kraemer. A methodology for building application-specific visualizations of parallel programs". *Journal of*

Parallel and Distributed Computing, 18(2):258–264, 1993. http://www-static.cc.gatech.edu/gvu/softviz/parviz/parviz.html.

[SS94] L. Shabtay and A. Segall. Low complexity network synchronization. 8^{th} *Internatinal Workshop on Distributed Algorithms*, pages 223–237, 1994.

[Sun] Java Sun. *Java Remote Method Invocation (RMI). http://www.sun.com/products/jdk/rmi/.*

[Tel00] G. Tel. *Introduction to distributed algorithms.* Cambridge University Press, 2000.

[TZ01] Mikkel Thorup and Uri Zwick. Compact routing schemes. In 13^{th} *ACM Symposium on Parallel Algorithms and Architectures (SPAA01)*, pages 1–10. ACM Press, July 2001.

[TZ05] Mikkel Thorup and Uri Zwick. Approximate distance oracles. *Journal of the ACM*, 52(1):1–24, January 2005.

[ViS06] ViSiDiA. http://www.labri.fr/visidia. LaBRI Distributed Algorithms Group, 2006.

[Wen91] Rephael Wenger. Extremal graphs with no C^4's, C^6's, or C^{10}'s. *Journal of Combinatorial Theory, Series B*, 52(1):113–116, 1991.

[Wil93] D. Williams. *Probability with Martingals.* Cambridge University Press, 1993.