Dessin de graphe distribué par modèle de force : application au Big Data

Antoine Hinge

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

par **Antoine HINGE**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

Dessin de graphe distribué par modèle de force: application au Big Data

---

Date de soutenance : 28 Juin 2018

Devant la commission d'examen composée de :

Christophe HURTER ...	Professeur, ENAC	Président
Jean-Loup GUILLAUME .	Professeur, L3i	Rapporteur
Pascale KUNTZ-COSPEREC	Professeure, LINA	Rapportrice
David AUBER	Maître de conférence, LaBRI .	Directeur de thèse
Nicolas HANUSSE	Directeur de Recherche, LaBRI	Examinateur
Patricia THEBAULT ...	Maître de conférence, LaBRI .	Examinatrice

– 2018 –

Résumé

Dessin de graphe distribué par modèle de force: application au Big Data

Les graphes, outil mathématique pour modéliser les relations entre des entités, sont en augmentation constante du fait d'internet (par exemple les réseaux sociaux). La visualisation de graphe (aussi appelée dessin) permet d'obtenir immédiatement des informations sur le graphe. Les graphes issus d'internet sont généralement stockés de manière morcelée sur plusieurs machines connectées par un réseau. Cette thèse a pour but de développer des algorithmes de dessin de très grand graphes dans le paradigme MapReduce, utilisé pour le calcul sur cluster.

Parmi les algorithmes de dessin, les algorithmes reposants sur un modèle physique sous-jacent pour réaliser le dessin permettent d'obtenir un bon dessin indépendamment de la nature du graphe. Nous proposons deux algorithmes par modèle de forces conçus dans le paradigme MapReduce. GDAD, le premier algorithme par modèle de force dans le paradigme MapReduce, utilise des pivots pour simplifier le calcul des interactions entre les nœuds du graphes. MuGDAD, le prolongement de GDAD, utilise une simplification récursive du graphe pour effectuer le dessin, toujours à l'aide de pivots. Nous comparons ces deux algorithmes avec les algorithmes de l'état de l'art pour évaluer leurs performances.
Mots clés Visualisation, Big Data, Dessin de graphe, Calcul distribué

Abstract

Distributed force directed graph drawing: a Big Data case study

Graphs, usually used to model relations between entities, are continually growing mainly because of the internet (social networks for example). Graph visualization (also called drawing) is a fast way of collecting data about a graph. Internet graphs are often stored in a distributed manner, split between several machines interconnected. This thesis aims to develop drawing algorithms to draw very large graphs using the MapReduce paradigm, used for cluster computing.

Among graph drawing algorithms, those which rely on a physical model to compute the node placement are generally considered to draw graphs well regardless of the type of graph. We developped two force-directed graph drawing algorithms in the MapReduce paradigm. GDAD, the fist distributed force-directed graph drawing algorithm ever, uses pivots to simplify computations of node interactions. MuGDAD, following GDAD, uses a recursive simplification to draw the original graph, keeping the pivots. We compare these two algorithms with the state of the art to assess their performances.
Keywords Visualization, Big Data, Graph drawing, Distributed algorithm

« She had been listening to friends and relatives ever since she was young, listening while they talked about themselves and each other, carrying them with her in a huge, artfully constructed mental map of crisscrossing lives. [...] Aunt Gerda rolled out a long piece of shelf paper on the dining room table and held it down with drawing pins. She made a big black dot, a round head for each of them, with the name and birth date and title in a pretty little oval. She placed their children alongside, connected to their parents with a red line. She put all romantic relationships in pink – double lines for unconventional or forbidden alliances. Aunt Gerda became engrossed. Some heads were burdened with perfect coronas of pink – like galactic suns, impressive and probably regrettable. »
Tove Jansson, *The Listener*

Remerciements

En premier lieu, je remercie le projet Request qui a financé mes travaux de thèse. Je remercie l'Université de Bordeaux et le LaBRI pour les excellentes conditions d'accueil. Merci aux membres de mon jury, Pascale KUNTZ-COSPEREC, Jean-Loup GUILLAUME, Christophe HURTER, Nicolas HANUSSE et Patricia THEBAULT d'avoir accepté de relire mon manuscrit. Merci à David de m'avoir proposé cette opportunité riche en enseignements et en émotions.

Au sein de l'équipe, j'aimerai remercier en particulier les doctorants qui m'ont épaulés lors de ces quatre années. Merci à Alexandre pour la patience dont il a su faire preuve lors de mes nombreuses interrogations. Merci à Gaëlle pour sa bonne humeur et pour son travail de développement considérable sur MuGDAD. Merci à Frédéric pour le travail précieux d'ingénierie mené souvent dans l'ombre, mais toujours avec brio. Merci à Jason et Joris pour leur curiosité avide qui a contribué sans aucun doute à la bonne ambiance de l'équipe lors de mon séjour au LaBRI. Merci à Aaron, magicien émérite, d'être toujours partant pour se changer les idées lorsque le travail devient trop pesant.

Merci à mes parents pour leur amour et leur soutien. Merci à Marius. Merci à mes grands-parents, oncles et tantes qui suivent mes aventures comme un roman feuilleton. Je sais qu'ils sont tous fiers et je les remercie de continuer à croire en moi. Merci à Marianna d'avoir traqué sans relâche les fautes d'orthographes de mon manuscrit. Celles qui restent sont de mon fait !

Merci à un groupe particulier de personnes, le club escalade de Centrale Lille de 2012-2014, transformé en vidéoclub Youtube par la force des choses. Merci à Driss, Jérémy, Floriane, Clément, Phuong, Pierre et Clara et à Nicolas. J'espère que malgré la distance, l'esprit du club perdurera.

Je garde mes mots les plus doux pour Lélia. Son amour de tous les instants constitue l'aide la plus précieuse que j'ai pu recevoir. Ces années de thèses n'ont été facile pour aucun de nous deux mais je suis très heureux de les avoir passées en ta compagnie. J'espère en passer de très nombreuses encore, moins difficile à naviguer je l'espère.

Table des matières

Table des figures

Chapitre 1

Introduction

1.1 Motivation

La production de données est aujourd'hui plus importante que jamais. L'analyse experte peut s'appuyer sur cette masse de données dans la mesure où celle-ci est analysable par un être humain. Ce postulat de base se confronte aujourd'hui à la croissance rapide du volume de données. Des indicateurs statistiques classiques (moyenne, médiane, écart-type ...) peuvent offrir des informations vitales sur un jeu de données mais cette information n'est pas suffisante pour en comprendre tous les détails. Anscombe [6] propose, en 1973, quatre jeux de données ayant la particularité d'avoir la même droite de régression, voir Figure 1.1. La visualisation permet pourtant de les différencier en un coup d'œil, et de choisir les bons outils statistiques pour les décrire et les comparer.

Plus récemment, Matejka et Fitzmaurice [105] proposent une méthode pour générer des jeux de données à la manière d'Anscombe. Les nuages de points obtenus ont des statistiques proches et peuvent prendre une forme quelconque grâce à une méthode de recuit simulé. Une analyse statistique superficielle ne permet pas de différencier des jeux de données qui sont pourtant clairement de formes différentes. Cette méthode est illustrée en Figure 1.2.

Les deux exemples précédents montrent que l'analyse statistique est possible mais donne une information résumée, insuffisante dans certains cas. Visualiser les données permet de fournir beaucoup d'informations sans restreindre l'analyse à une simple donnée numérique. La visualisation est donc un outil essentiel pour extraire de l'information et guider l'analyse statistique.

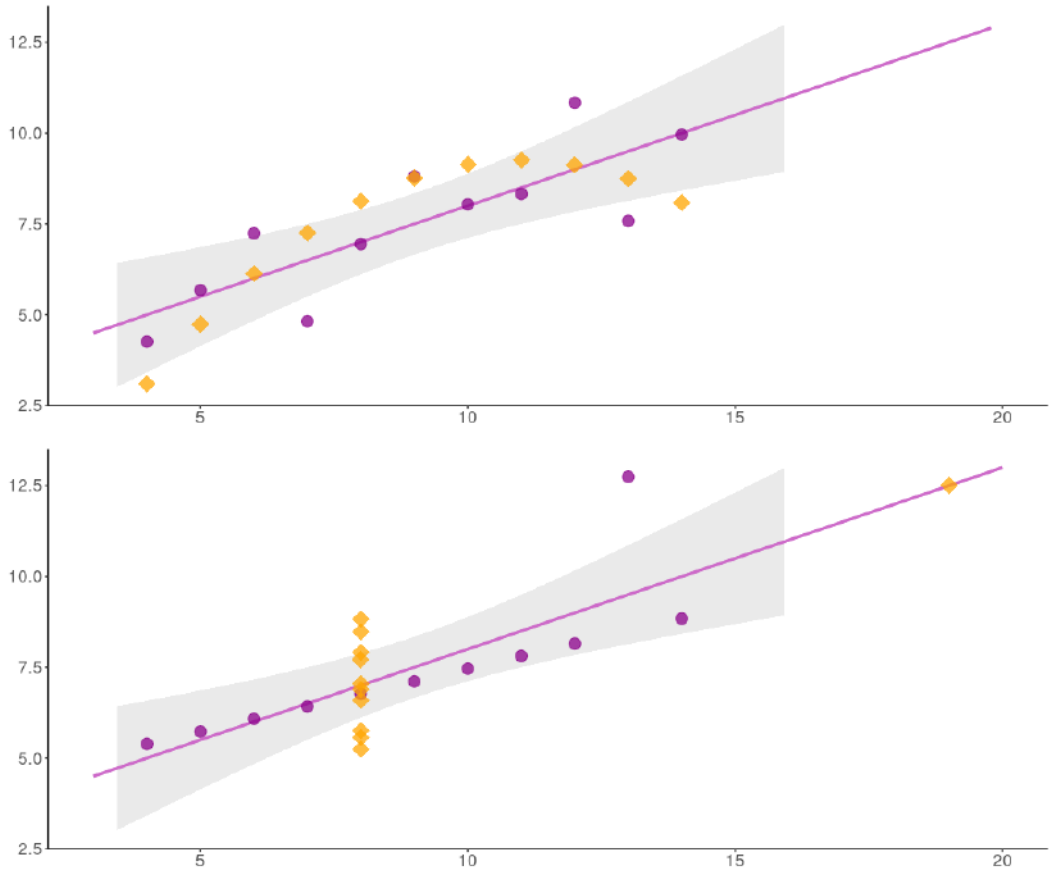


FIGURE 1.1 – Jeux de données d'Anscombe [6] : quatre nuages de 11 points. L'équation de la droite de régression linéaire est la même pour ces jeux de données. La visualisation fait apparaître leur différences.

Les réseaux (transports, réseaux sociaux, etc.) sont un outil indispensable pour représenter les relations entre différents éléments d'un ensemble. Ils sont généralement modélisés mathématiquement par des graphes. La taille des graphes manipulés a drastiquement augmenté lors des dernières années, ce qui en complique l'analyse statistique.

Grisi-Filho et al. [61] comparent, en 2013, les méthodes possibles pour générer des graphes ayant la même distribution de degré des nœuds. Ils démontrent que selon la méthode de génération des graphes utilisée, la structure des graphes diffère de manière significative. Parmi les outils utilisés pour cette démonstration, la visualisation de ces graphes (*cf.* Figure 1.3) est utilisée comme une première approche avant de proposer des outils statistiques adaptés à leur analyse (le degré moyen des nœuds voisins par exemple). La visualisation des graphes est un outil essentiel pour l'exploration de données et vient compléter les méthodes statistiques d'analyse de données.

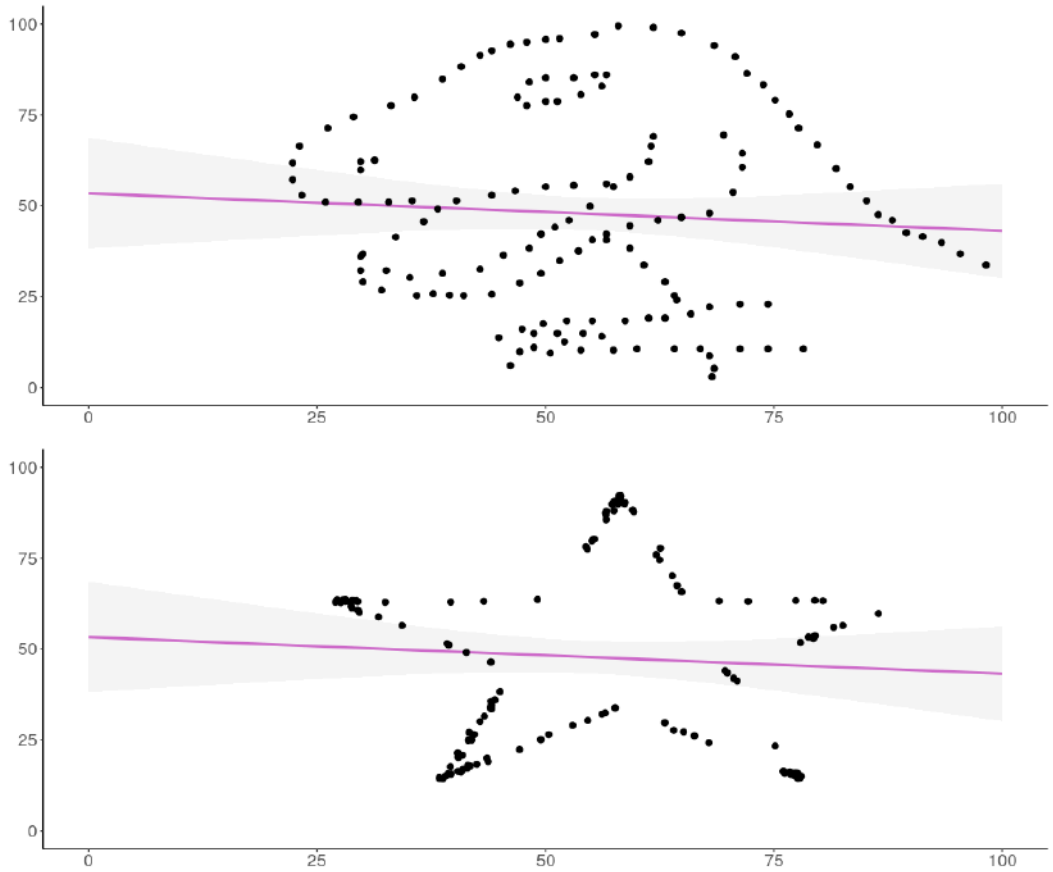


FIGURE 1.2 – Deux nuages de points possédant des statistiques proches représentés avec leur droite de régression linéaire. Ces jeux de données peuvent être générés pour suivre n'importe quelle forme (ici un dinosaure et une étoile) par une méthode de recuit simulée, proposée par Matejka et Fitzmaurice [105].

Pour décrire des données massives, il est nécessaire de trouver les bons outils statistiques. L'exploration numérique de ces données est compliquée par leur volume et la visualisation doit permettre de guider l'analyse numérique une fois le jeu de données appréhendé.

1.2 Objectif de la thèse et périmètre

Cette thèse s'intitule "Dessin de graphe distribué par modèle de force : application au Big Data". Les travaux présentés ont pour objectif de visualiser des données relationnelles massives stockées dans des serveurs. Ces travaux se placent donc à l'intersection de deux champs de recherche : la visualisation de graphe et le calcul distribué.

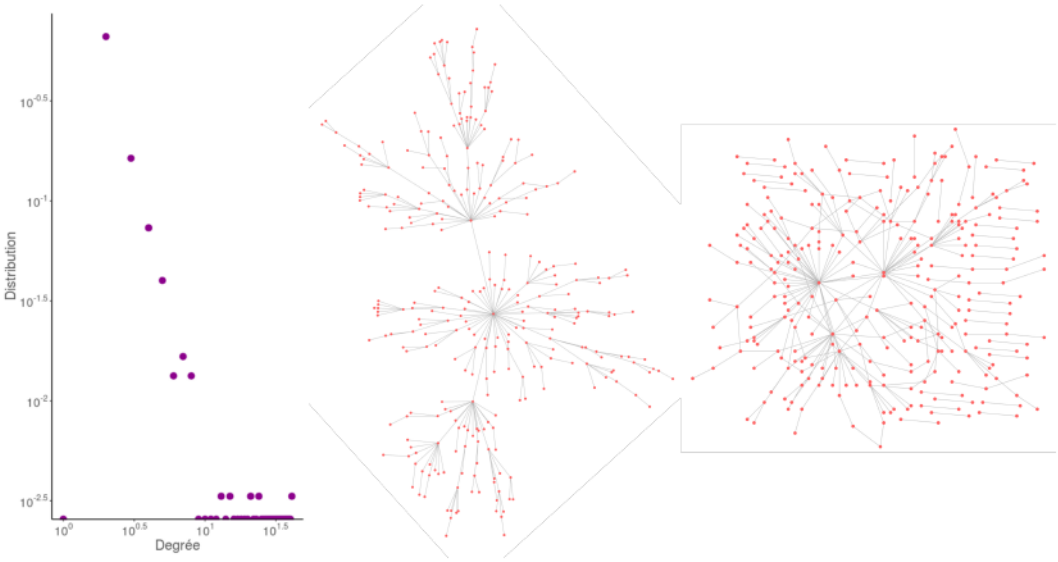


FIGURE 1.3 – Deux graphes ayant la même distribution de degrés *(Gauche)* et des structures différentes, généré par le modèle de Barabasi *(Milieu)* et de Molly-Reed *(Droite)*. Grisi-Filho et al. [61] comparent cinq modèles de génération de graphes ayant la même distribution de degré.

1.2.1 Visualisation de graphe

En Section 1.3, nous présentons les données relationnelles, *i.e.* les données décrivant les associations entre différents éléments d'un ensemble. Ce type de données a de nombreuses applications dans différents champs scientifiques. Ces données sont généralement modélisées par des graphes.

La représentation de graphe est présentée en Section 1.4. Différents types de visualisation existent pour représenter les graphes. Dans nos travaux, nous nous concentrons sur la représentation nœuds-liens.

La représentation de graphe est réalisée grâce à des algorithmes plus ou moins spécialisés. Le choix de l'algorithme est réalisé à partir de la représentation choisie et du type de graphe à représenter. Les algorithmes de représentation des graphes sont présentés en Section 1.5.

Périmètre de la thèse Dans nos travaux, nous avons exploré les algorithmes de dessin par analogie physique pour la représentation nœuds-liens. Pour ces algorithmes, la position des nœuds est obtenue à partir d'un modèle physique sous-jacent. Cette famille d'algorithmes est présentée exhaustivement dans l'état de l'art, *cf.* Section 3.

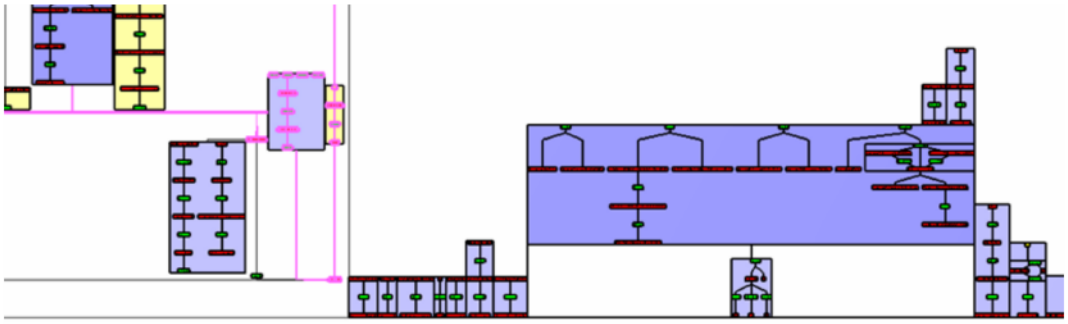


FIGURE 1.4 – Extrait de la visualisation d'un réseau métabolique réalisé avec MetaViz [25]. Les protéines sont représentées par des nœuds. Les liens représentent les interactions entre ces protéines.

1.2.2 Environnement Big Data

Le volume des données produites n'a jamais été aussi important. Ces données sont principalement des données numériques stockées sur des serveurs. Pour stocker ces données, plusieurs ordinateurs sont connectés et partagent leur espace de stockage. Plus récemment, des méthodes de calcul ont été développées pour effectuer des opérations de calcul directement sur ces machines, auparavant utilisées principalement pour le stockage. L'émergence du Big Data est décrite en Section 1.6.

Périmètre de la thèse Parmi les techniques de calcul utilisés en Big Data, nos travaux ont pour enjeu de tirer parti des paradigmes distribués MapReduce et Pregel. Ces deux paradigmes sont décrits dans le Chapitre 2.

1.3 Données relationnelles et graphes

Les données relationnelles sont omniprésentes dans différents champs scientifiques, que ce soit en sciences exactes ou en sciences humaines. Ces données sont généralement bien modélisées grâce à des graphes. Cette section vise à donner quelques exemples d'application de la visualisation de graphes pour des données réelles.

Un des premiers exemples d'utilisation de graphe provient d'un problème mathématique : le problème de Königsberg. Il se pose de la manière suivante : est-il possible de traverser tous les ponts de Königsberg en ne passant qu'une seule fois sur chaque pont. Grâce à une abstraction de ce problème sous la forme d'un graphe, Euler parvient à résoudre le problème, ce qui est considéré aujourd'hui comme le premier résultat de la théorie des graphes.

Les interactions entre différents éléments d'un système peuvent être facilement modélisées par un graphe, ce qui présente un intérêt particulier pour

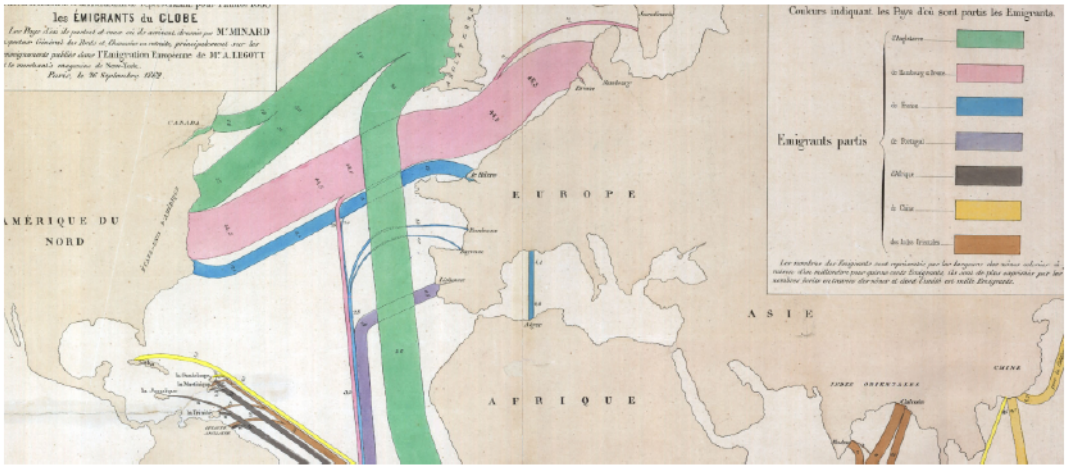


FIGURE 1.5 – Extrait d'une carte de Minard représentant des flux migratoires. L'épaisseur du trait est proportionnelle au nombre de migrants. Cette visualisation peut être modélisée par un graphe orienté où le poids des arêtes représente le nombre de migrants.

les sciences naturelles. Par exemple, les réseaux métaboliques modélisent les interactions et réactions chimiques des différentes molécules au sein d'un organisme ou d'une cellule. Ces interactions peuvent être modélisées sous la forme d'un graphe où les arêtes représentent les interactions entre différentes composantes chimiques. La visualisation de ce type de graphe permet l'exploration ou la validation d'hypothèses, voir par exemple les travaux de Bourqui [25]. Ces réseaux métaboliques sont illustrés en Figure 1.4.

Les flux (humains, de marchandise *etc.*) sont aisément modélisables par un graphe. Les cartes de Minard (voir Figure 1.5), l'un des plus anciens exemples de visualisation, représentent des flux de personnes ou de marchandises dont l'épaisseur d'arête dépend du volume de l'échange. Minard utilise des données géolocalisées et le problème du dessin automatique ne se pose alors pas puisque les flux sont représentés sur une carte. Cependant, si les flux sont abstraits, comme par exemple pour les flux de données transitant sur internet, il n'est plus envisageable de réaliser de telles visualisations à la main. La représentation automatique des flux, comme présenté en Figure 1.5, est un problème difficile.

Un exemple plus récent, qui nous concerne directement, est celui de la création de circuits intégrés dont la densité d'intégration est très élevée (Very-Large Scale Intergration ou VLSI). Chaque composant électronique est constitué de pattes qui sont reliées entre elles de manière à créer un circuit intégré réalisant une fonction donnée. Ce problème est modélisable par un graphe dont les nœuds représentent les composantes à placer et dont les arêtes représentent les connexions électriques. Le placement de ces composants se complexifie avec leur multiplication. En effet, les croisements entre pistes doivent être évités.

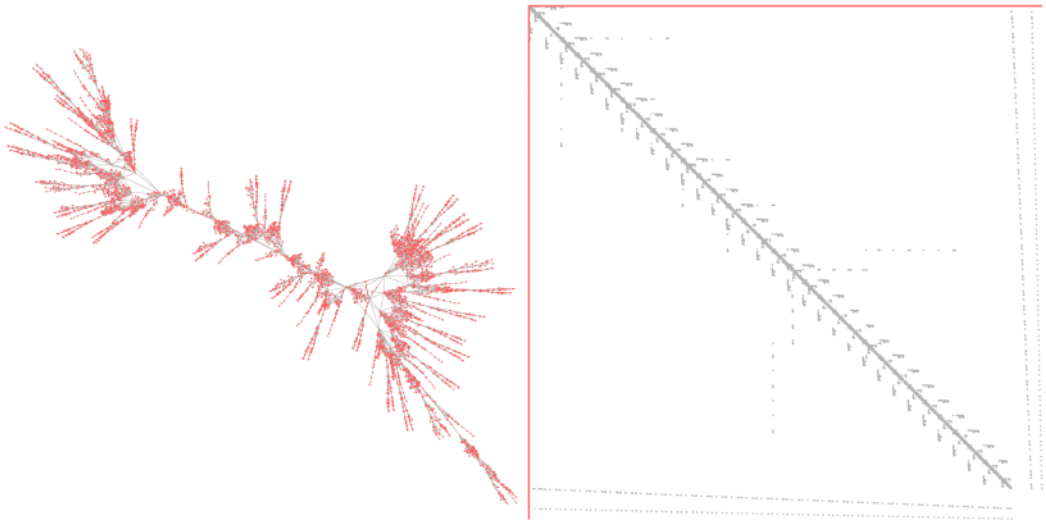


FIGURE 1.6 – Représentation du graphe add32 *(gauche)* Vue noeuds-liens *(droite)* Vue matricielle.

1.4 Représentation des graphes

1.4.1 Représentations usuelles

Représentation nœuds-liens

Dans la représentation nœuds-liens, les nœuds et les arêtes sont représentés par des marques, en reprenant la nomenclature de Bertin [22]. Les nœuds sont représentés par des points et les arêtes par des lignes. Cette représentation est la représentation la plus ancienne et probablement la plus intuitive : à chaque élément du graphe est associé une marque unique. La Figure 1.6 montre un exemple de représentation nœuds-liens.

Le problème du dessin de graphe pour la représentation nœuds-liens revient donc à trouver pour chacun des nœuds une position dans un espace, le plus souvent un plan. Chaque nœud peut être placé à n'importe quel endroit du plan. Pour limiter ces possibilités, une fonction de coût est créée pour définir la position optimale des nœuds et c'est la minimisation de cette fonction qui permet d'obtenir le dessin.

Représentation matricielle

La représentation matricielle est une représentation visuelle de la matrice d'adjacence du graphe. Cette matrice de taille $|V|^2$ représente toutes les arêtes possibles entre les nœuds du graphe. Chaque nœud est donc représenté deux fois dans la visualisation matricielle : une fois horizontalement et une fois verticalement.

FIGURE 1.7 – Représentation du même arbre pour trois visualisations différentes *(gauche)* Vue noeuds-liens *(milieu)* TreeMap circulaire *(droite)* Vue noeuds-liens circulaire.

Chaque valeur de la matrice d'adjacence contient un 1 ou **True** si le graphe contient une arête entre deux nœuds et un 0 ou **False** si le graphe ne contient pas d'arêtes entre deux nœuds. Dans la représentation matricielle, la matrice d'adjacence du graphe est représentée à l'écran en remplaçant les valeurs dans la matrice par un symbole si l'arête existe. La Figure 1.5 montre un exemple de représentation matricielle.

Représenter le graphe revient à choisir un ordre pour les nœuds, c'est à dire à associer à chaque nœud de V un entier distinct dans l'intervalle $[\![1, |V|]\!]$ parmi $V!$ ordres possibles. Ce problème d'arrangement linéaire minimum, aussi appelé MINLA, a été défini en 1964 par Harper [69]. Pour ce problème, une fonction de coût est définie.

Trouver l'ordre qui minimise une fonction de coût choisie au préalable a été prouvé NP-difficile, voir par exemple Garey et Johnson [53]. Trouver un ordre pour les nœuds n'est donc pas réalisable directement via la minimisation d'une fonction de coût si le graphe contient des centaines de nœuds. Des heuristiques sont donc utilisées pour trouver un ordre.

1.4.2 Représentations spécialisées

Représenter les arbres

Les arbres sont des graphes orientés acycliques où les nœuds ont au plus un nœud parent et peuvent avoir plusieurs nœuds enfants. Ces graphes particuliers sont plus faciles à dessiner grâce à la structure hiérarchique contenue dans la définition. Deux types de représentations principales existent pour les arbres : la représentation nœuds-liens et les représentation remplissant l'espace.

La représentation nœuds-liens des arbres est similaire à celle des graphes : les nœuds sont représentés par des formes et reliés entre eux par des arêtes sous la forme de liens entre ces formes. Cependant, les nœuds d'un arbre peuvent

être classés selon leur profondeur (*i.e.* distance à la racine). Wetherell [136] propose ainsi des critères esthétiques pour les algorithmes de dessin d'arbre. Le premier critère est que les nœuds à la même profondeur sont situés sur des droites parallèles. Ceci induit la visualisation par couche de profondeurs. Le deuxième critère esthétique est que le parent de nœuds se situe au barycentre de la position de ses enfants, sur la couche supérieure. Wetherell propose un algorithme de dessin d'arbre respectant ces critères. Reingold et Tifold [117] ajoutent un critère supplémentaire à ceux de Wetherell pour améliorer l'esthétique du dessin. Ils remarquent en effet qu'un arbre n'a pas forcément un dessin symétrique si l'ordre de parcours de ses nœuds est inversé.

Des variations de la représentation nœuds-liens par couches sont possibles [73]. Par exemple, les couches peuvent être placées sur des cercles concentriques (vue radiale) ou encore dans des disques inclus les uns dans les autres (vue ballons, *cf.* par exemple Grivet et al. [62]). Les algorithmes de dessin d'arbres sont généralement de complexité linéaire selon le nombre de nœud puisque grâce à leur structures hiérarchiques, un passage sur les données suffit pour trouver la position des nœuds. Pour la plupart de ces algorithmes, l'espace du dessin n'est pas utilisé optimalement (à l'exception de la visualisation par arbre H [73] qui positionne les nœuds sur une courbe fractale) et les représentations remplissant l'espace proposent une alternative à ce problème.

Les représentations remplissant l'espace sont un autre type de représentation classique pour les arbres. Ce ne sont plus les nœuds qui sont représentés directement, mais leurs relations hiérarchiques. Dans TreeMap [80], les nœuds sont représentés par des boîtes imbriquées où chaque boîte contient les boîtes représentants les nœuds de ses successeurs dans l'arbre. La surface des boîtes est proportionnelle au nombre de feuilles de l'arbre qu'elles englobent et les imbrications récursives permettent de représenter tous les nœuds, de la racine aux feuilles. La racine est alors la boîte englobant toutes les autres.

Représenter les DAG

Les graphes orientés acycliques (directed acyclic graph ou DAG en anglais) ont la propriété de pouvoir être organisés en couches hiérarchiques de nœuds. Pour représenter ces graphes, des algorithmes similaires aux algorithmes de dessin d'arbres sont utilisés.

Gansner et al. [52] proposent en 1988 un algorithme de représentation nœuds-liens pour les DAG. Les nœuds possédant des ancêtres communs sont représentés au même niveau, comme dans la représentation d'arbres.

Kornaropoulos et Tollis [92] proposent plus récemment une représentation matricielle des DAG, basées sur un ordre faible parmi les nœuds [91]. Des liens orthogonaux sont ajoutés entre les nœuds de manière à rendre le parcours dans le graphe plus efficace et à mettre en lumière la hiérarchie au sein des nœuds.

Représentations	Nœuds-liens	Matricielle	TreeMap
Noeuds-liens			
Matricielle			
TreeMap			

FIGURE 1.8 – Matrice des représentations hybrides. *De gauche à droite et de haut en bas (a)* Auber et al. [11] *(b)* NodeTrix [72] *(c)* Elastic Hierarchies [140] *(d)* Quilt [134] *(e)* Holten [77]

1.4.3 Représentations hybrides

Les représentations hybrides de graphes combinent plusieurs représentations pour tirer le meilleur parti de chacune.

Les quilts [134] permettent une visualisation de graphes par couches. La visualisation des graphes est organisée selon des couches de niveau où toutes les arêtes ont la même direction. Les quilts [134] sont une variante de la visualisation par matrice où les matrices sont représentées par bloc plutôt que par une unique matrice carrée. L'organisation par couches permet d'organiser le dessin des matrices selon des blocs chaînés : une première matrice représente les arêtes entre la première et la deuxième couche, une deuxième matrice est utilisée pour représenter les arêtes entre la deuxième et la troisième et ainsi de suite. Il peut arriver que certaines arêtes relient deux couches non consécutives et le nœud destination est alors représenté par sa couleur, hors de la matrice connectant deux couches. Cette visualisation permet de retenir le côté hiérarchique du graphe par couches, qui peut être perdu dans une représentation matricielle.

NodeTrix [72] fait le lien entre la représentation nœuds-liens et la représentation matricielle des graphes. Les limitations de la visualisation nœuds-liens

lorsque les graphes deviennent trop denses poussent Henry et al. à utiliser la représentation matricielle pour représenter les graphes. Cependant, cette visualisation offre des performances moins bonne pour le suivi de chemin [56] que la représentation nœuds-liens. NodeTrix [72] propose un compromis entre les deux approches pour tirer le meilleur parti de chacune. Les sous-graphes denses sont donc représentés par des matrices qui sont connectées les unes aux autres via des liens. Tous les nœuds d'un sous-graphe sont représentés sur chaque coté de la matrice ce qui permet d'éviter l'occlusion visuelle créée par les croisements d'arêtes.

Auber et al. [11] proposent des visualisations nœuds-liens imbriquées pour représenter les graphes petit monde. Ces graphes caractérisés par une faible distance moyenne entre les nœuds sont généralement denses et difficile à représenter. En décomposant un tel graphe en sous-graphes denses, il est possible de créer un graphe quotient reliant les différents sous-graphes. La visualisation est alors composée de méta-nœuds reliés entre eux. Chaque méta-nœud contient une visualisation du sous-graphe qu'il représente, possiblement décomposé lui aussi de cette manière.

Holten [77] propose de visualiser les relations d'adjacences dans les graphes hiérarchiques en superposant des arêtes aux visualisations d'arbres, que ce soient des représentations nœuds-liens ou des représentations remplissant l'espace. Pour éviter l'occlusion visuelle causée par ces arêtes, il propose d'appliquer un faisceautage des arêtes permettant de regrouper les arêtes proches.

Zhao et al. [140] propose Elastic Hierarchies combinant les TreeMap et la représentation nœuds-liens pour la représentation des arbres. Différents niveaux de TreeMap sont chaînés via une représentation nœuds-liens permettant ainsi de visualiser efficacement chaque élément. La visualisation de la hiérarchie est conservée, tâche qui peut être difficile à percevoir dans une représentation TreeMap.

La Figure 1.8 reprend les différentes représentations hybrides présentées sous la forme d'un tableau à deux entrées combinant les représentations nœuds-liens, matricielle et TreeMap.

1.4.4 Qualité du dessin

Comment représenter des données relationnelles de manière efficace ? La qualité d'une représentation est d'abord mesurée par les tâches qu'elle permet de réaliser efficacement. Lee [96] propose une taxonomie des tâches réalisables, décomposée en trois grandes familles de tâches.

Topologie du graphe Cette famille comprend des tâches liées au voisinage des nœuds : trouver les nœuds adjacents à un nœud donné et leur nombre par exemple. Cette catégorie de tâches comprend aussi la recherche du plus court chemin entre deux nœuds et l'identification de cliques ou quasi-cliques.

Vue d'ensemble Elle contient les tâches d'évaluations rapides, du nombre de nœuds et d'arêtes, de la présence de clusters. Elle contient aussi les tâches visuelles de suivi et de lecture, comme la recherche d'un chemin ou d'un nœud déjà visité dans le dessin.

Attributs des nœuds et arêtes Trouver les nœuds ayant un attribut d'une certaine valeur ou les arêtes d'un certain type connectant deux noeuds sont des tâches de cette famille.

Différentes visualisations de graphes permettent d'obtenir de meilleures performances sur différents jeux de tâches. Ghoniem [56] compare la représentation matricielle et la représentation nœuds-liens et conclut que la représentation matricielle offre de meilleures performances pour les tâches d'évaluation du nombre de nœuds, mais pas pour trouver un chemin entre deux nœuds.

Différentes représentations ont aussi différents critères esthétiques qui leur sont propres. Ces contraintes esthétiques sont généralement maximisées par les algorithmes de dessins automatiques et permettent l'amélioration des performances des utilisateurs à la réalisation de différentes tâches. Purchase [113] décrit par exemple le critère de symétrie pour les représentations nœuds-liens qui stipule que les symétries du graphe doivent se retrouver dans le dessin final et elle montre [112] que ce critère permet effectivement d'améliorer les temps de réponses des utilisateurs.

1.5 Algorithmes de visualisation de graphes

1.5.1 Dessin de graphe planaires

Les graphes planaires sont les graphes qui peuvent être dessinés sur un plan (par une représentation nœuds-liens) sans que les arêtes ne se croisent. Les algorithmes de dessin pour les graphes planaires sont généralement basés sur cette propriété particulière. Plusieurs algorithmes de dessin planaire sont présentés dans l'état de l'art de Di Batista et al. [38].

Tutte [128] propose en 1963 un algorithme de dessin pour les graphes planaires triconnexes. Tutte prouve que le dessin obtenu est un dessin sans croisement et que les faces obtenues dans le dessin sont convexes.

Tutte partitionne l'ensemble des nœuds en deux sous-ensembles : l'ensemble des nœuds fixes et l'ensemble des nœuds libres. L'algorithme de Tutte est initialisé en plaçant les nœuds fixes selon un polygone et les nœuds libres à l'origine du dessin. Le polygone formé par les nœuds fixe forme l'enveloppe convexe du dessin.

Une fois le dessin initialisé, les nœuds libre sont déplacés itérativement vers une nouvelle position. Chaque nœud est placé au barycentre de la position de ces voisins. Cette opération est itérée jusqu'à la convergence du dessin. On

note p_u la position du nœud u dans le plan. La nouvelle position du nœud u s'exprime alors comme :

$$p_u^{new} = \frac{1}{deg(u)} \sum_{v \in E} p_v$$

L'algorithme de Tutte est précurseur des algorithmes de dessin par analogie physique. En effet, il est possible de réécrire la nouvelle position du nœud u comme un déplacement lié à un système physique :

$$p_u^{new} = \frac{1}{deg(u)} \sum_{v \in E} (p_v - p_u^{old}) + p_u^{old}$$

Le nœud u subit donc une force d'attraction de la part de ses voisins proportionnelle à leur distance dans le dessin. Ceci peut être modélisée par un système physique où l'ensemble des arêtes est remplacé par un ressort de longueur nominale nulle.

1.5.2 Dessin de graphe par analogie physique

Les tentatives d'automatisation d'intégration des composants électroniques (VLSI, *cf.* Section 1.3) ont donné lieu au premier algorithme de dessin de graphe par modèle de forces. Quinn et al. [116] utilisent un algorithme d'optimisation pour placer des puces interconnectées, modélisées par un graphe. Pour les problèmes de routage des cartes électroniques, la plupart des simplifications possibles sont démontrées NP-complètes [81, 82].

Nous présentons ici brièvement les deux familles d'algorithmes qui fonctionnent sur ce principe : les algorithmes par modèles de forces et les algorithmes par réduction du stress. Un état de l'art plus complet est disponible dans le Chapitre 3.

Algorithme par modèle de force

Les algorithmes de dessin par modèle de forces considèrent les nœuds dans le graphe comme des particules physiques. Les nœuds sont soumis à des forces qui miment les phénomènes physiques comme la répulsion de particules chargées électriquement ou la force de rappel d'un ressort. En relâchant le système physique à partir d'un état initial aléatoire, le système converge vers un état d'équilibre *i.e.* son minimum d'énergie. Fruchterman et Reingold proposent ce modèle en 1991 [48]. Les forces d'attraction sont appliquées entre les nœuds voisins du graphe. Elles assurent l'existence d'un minimum pour le système physique. Les forces de répulsion s'appliquent entre tous les nœuds. Elles assurent la répartition de la position des nœuds dans le plan du dessin.

Soit u un nœud et k un paramètre de valeur fixe. On note p_u la position du nœud u dans le plan du dessin. L'ensemble des forces appliquées au nœud u s'exprime de la manière suivante :

$$\mathbf{F}(u) = \sum_{(u,v)\in E} \frac{\|p_v - p_u\|^2}{k} \hat{\mathbf{uv}} - \sum_{v\in V} \frac{k^2}{\|p_v - p_u\|} \hat{\mathbf{uv}}$$

Pour atteindre l'équilibre, Fruchterman et Reingold [48] proposent de calculer un déplacement des nœuds selon la direction des forces appliquées sur ces nœuds. Pour chaque nœud, les forces d'attraction et de répulsion sont calculées, ajoutées et le déplacement à effectuer en est déduit. Les nœuds sont déplacés simultanément. En pratique, cette méthode permet de réduire l'énergie du système dans la plupart des cas mais il n'y a pas de garantie de convergence de ce processus vers le minimum d'énergie. Pour palier ce problème, Fruchterman et Reingold introduisent un facteur de refroidissement, inspiré des techniques de recuit simulé, qui réduit l'amplitude des forces appliquées à chaque itération et qui permet effectivement de converger vers un minimum d'énergie.

Le calcul des forces, telles que présentés dans [48] a une complexité temporelle de $O(|V|^2+|E|)$. Il est communément admis qu'il est nécessaire d'effectuer $O(|V|)$ itérations pour atteindre l'équilibre du système physique. Avec cette méthode, des graphes d'une centaine de nœud au plus, peuvent être dessinés. Ces algorithmes sont présentés plus en détails dans le Chapitre 3.

Algorithme par réduction du stress

Les algorithmes par réduction du stress peuvent être vu comme une approche plus générique des algorithmes par modèle de force. La fonction de coût exprimée sur le dessin peut s'appliquer non seulement aux nœuds, aux arêtes mais aussi à des sous-ensembles de nœuds et d'arêtes. La fonction de coût (ou d'énergie) ainsi obtenue est celle que l'on cherche à minimiser, généralement grâce à des techniques d'optimisation numérique. Pour la plupart des algorithmes par réduction de stress, il existe une analogie en terme de système physique, mais pour certaines contraintes, cela n'est pas possible.

Kamada et Kawai [84] définissent la distance graphe entre les nœuds comme distance optimale dans le dessin. On note p_i la position du i-ème nœud et l_{ij} la distance entre les i et j-ème nœuds dans le graphe. La fonction de coût à minimiser s'exprime de la manière suivante :

$$E = \sum_{i=1}^{|V|-1} \sum_{j=i+1}^{|V|} (|p_i - p_j| - l_{ij})^2$$

Cet algorithme est généralement modélisé par l'analogie physique suivante : les nœuds sont des anneaux. Entre chaque paire de nœuds est attaché un ressort dont la longueur à vide correspond à la distance des nœuds dans le graphe.

L'état d'équilibre correspond alors à une configuration finale où la distance pour chaque paire de nœud dans le plan est la plus proche possible de la distance graphe de cette paire, *i.e.* où le stress du système physique dû à la tension des ressorts est minimal (d'où le nom de réduction du stress).

Kamada et Kawai [84] utilisent la méthode de Newton pour trouver l'état d'équilibre : le nœud avec la plus haute énergie est choisi et sa nouvelle position est déterminée grâce à la méthode de Newton. Ce processus est itéré jusqu'à atteindre le minimum d'énergie ou à en être suffisamment proche.

1.5.3 Algorithmes pour la représentation matricielle

Mueller [108] propose trois classes d'algorithmes heuristiques pour déterminer un ordre dans la représentation matricielle des graphes : les algorithmes issus de la théorie des graphes, ceux issus de l'algèbre des matrices creuses et ceux issus d'une analyse spectrale des matrices.

Les algorithmes issus de la théorie des graphes sont des algorithmes très souvent appliqués aux graphes qui sont utilisés pour donner un ordre aux noeuds. Par exemple, l'ordre du parcours dans un algorithme de parcours en largeur (ou BFS) ou en profondeur (ou DFS) peut être utilisé dans la représentation matricielle. Dans le pire des cas, ces parcours, et donc l'ordre qui en découle, sont obtenus en temps de $O(|V| + |E|)$. Le degré des nœuds peut aussi être un critère pour déterminer leur ordre. Avec une structure de données adaptée, obtenir le degré d'un nœud est une opération en temps constant. Obtenir l'ordre est alors de la complexité du tri, c'est à dire $O(|V| \cdot \log |V|)$.

Les matrices creuses (ou sparse matrices en anglais) sont des matrices dont la plupart des éléments sont des éléments nuls. Les graphes peuvent être représentés efficacement par ce type de matrices sous réserve qu'ils ne contiennent pas trop d'arête, *i.e.* que le degré moyen des nœuds soit faible. Cette définition est le pendant des matrices denses, où très peu d'éléments sont nuls. Le nombre faible d'éléments non nuls permet une compression aisée de ces matrices : au lieu de stocker tous les éléments de la matrice, il est possible de ne conserver que le numéro de colonne et de ligne des éléments non nuls et leurs valeurs. De très nombreux algorithmes peuvent être développés spécifiquement pour les matrices creuses, voir par exemple [32, 54].

L'algorithme spectral est une méthode utilisée pour trouver un ordre sur les noeuds, basée sur la décomposition en éléments propres de la matrice Laplacienne du graphe. Cette matrice est la différence entre la matrice des degrés des noeuds et la matrice d'adjacence. En trouvant le vecteur propre associé à la valeur propre non-nulle la plus petite, et en triant les nœuds selon les coordonnées de ce vecteur propre, on obtient un ordre qui est indépendant de l'ordre initial des nœuds (à condition que les coordonnées soient toutes différentes). Cet ordre est coûteux à obtenir, puisqu'il nécessite la décomposition en vecteurs propres qui est réalisable en temps $O(|V|^3)$ pour les matrices denses.

Cependant, son indépendance aux conditions initiales lui permet d'être un bon point de comparaison avec d'autres algorithmes.

1.6 Paradigme de calcul Big Data

Bien que la taille des données ait toujours été un facteur limitant en informatique, les données massives représentent un défi relativement nouveau. Avec le développement du Web 2.0, ce n'est plus simplement des pages statiques auxquelles les utilisateurs accèdent mais des pages où le contenu est lui-même enrichi par les utilisateurs. Ces avancées coincident avec l'apparition du terme Big Data vers la fin des années 1990. Diebold [39] trace l'origine du terme à John Mashey, responsable de la recherche scientifique dans une entreprise de calcul intensif. Le volume des données devient incomparable avec celui manipulé jusqu'à présent et l'apparition du terme vient acter le changement de paradigme : le phénomène du Big Data commence déjà à prendre forme. Ce terme est complété par Laney en 2001 (cf Diebold [39]) qui évoque dans une note de recherche non publiée les challenges du Big Data à travers trois V.

Tout d'abord, le Volume des données représente une des composantes du défi posé par le Big Data. Facebook disposait en 2014 d'une capacité de stockage de 300 PetaOctets [129] (c'est à dire $3.00 \cdot 10^{17}$ octets), le triple du volume disponible en 2013. Les géants du Web (Google, Facebook, Twitter, Amazon entre autres) disposent de capacités de stockage de cet ordre. La NSA, agence américaine de renseignement, dispose d'une ferme de stockage de 9000 m^2, sans compter les infrastructures aux alentours, dont la capacité de stockage est évaluée à 12 ExaOctets de données en 2013 [74].

La Variété des données représente un défi pour le calcul et l'automatisation dans le paradigme Big Data. Des données très différentes peuvent être stockées dans un même espace de stockage : des images, des fichiers sons, du texte, des informations sur les utilisateurs d'un produit, les clicks de ces même utilisateurs et bien d'autres types d'informations. La variété de ces supports rend les traitements automatiques plus complexes puisqu'ils doivent prendre en compte le type de fichier à analyser.

Enfin, la Vélocité représente le fait que les données évoluent rapidement. Facebook générait par exemple 600 TeraOctets de données par jour en 2014 [129]. Le fait que les données évoluent rapidement signifie que les calculs réalisées doivent être mis à jour en permanence pour suivre le flux entrant des données.

1.6.1 Technologies pour le Big Data

Pour résoudre des problèmes dans ce nouveau paradigme, des technologies (infrastructures, logiciels etc.) ont été développées pour pouvoir stocker le volume massif de données, les mettre à jour et les traiter.

Stockage des données

Le stockage des données massives et leur mise à jour en temps réel devient un problème quand ces données sont trop massives pour pouvoir tenir sur un seul disque. Malgré les progrès réalisés dans les technologies de stockage, la capacité de stockage maximale pour les disques les plus récents est de quelques Teraoctets, ce qui est donc insuffisant pour contenir toute l'information des problèmes de Big Data.

Pour résoudre ce problème de stockage, une solution privilégiée est de mettre en parallèle plusieurs disques de stockages, contenant chacun une partie de l'information. C'est le parti pris de Ghemawat et al. [55], chercheurs chez Google, qui développent au début des années 2000 un système de fichier distribué appelé Google File System (GFS). Les fichiers sont distribués sur de nombreux disques de stockage de capacité moyenne. Le coût d'une plateforme de stockage massif de ce type est bien moindre par rapport à l'achat de disques de stockages de très grande capacité : le volume de stockage est obtenu par l'addition du volume de chaque disque à bas coût et non par l'achat de technologies de stockage de pointe, souvent très coûteuses. GFS est aussi extensible (de l'anglais "scalable" : capable de passer à l'échelle) : si le besoin en stockage augmente, il est possible d'ajouter de nouveaux disques au système pour l'étendre sans modifier son état courant.

Avec l'ajout de très nombreux disques de stockage fonctionnant en parallèle, le risque de panne est accru. Lorsqu'un disque de stockage tombe en panne, son information est perdue. Pour ne pas troubler le bon fonctionnement du système, GFS a été développé de manière à être tolérant aux pannes. Pour ce faire, les données sont répliquées plusieurs fois sur les différents disques de manière à ce qu'un fichier soit toujours accessible même en cas de panne. Ceci signifie aussi que la lecture d'un fichier ne constitue pas un goulot d'étranglement : la partie du fichier auquel on cherche à accéder est répliquée à plusieurs endroits et ne bloque pas la lecture de plusieurs utilisateurs concurrents.

Calcul sur les données massives

Avec l'apparition des données massives, des nouvelles méthodes de calcul ont été développées pour pouvoir les traiter. L'enjeu est de pouvoir effectuer des calculs rapidement sur un volume de données très important qui évolue vite. Selon Pfister [111], trois approches sont envisageables devant un problème où la performance est un facteur limitant : travailler mieux, travailler plus et demander de l'aide. Travailler mieux consiste à améliorer les algorithmes existants ou à utiliser des techniques pour accélérer les exécutions d'un programme. Cette approche sous-tend une partie de travaux exposés dans ce manuscrit.

Travailler plus, c'est intensifier le calcul, disposer de plus de ressources pour aider à résoudre le problème. C'est l'approche prônée par le calcul à haute performance (HPC). En utilisant des ordinateurs très performants et dédiés

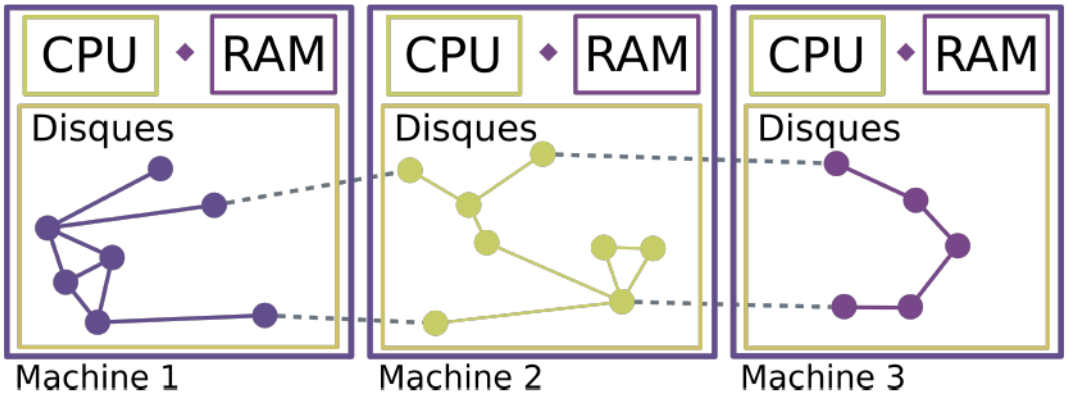


FIGURE 1.9 – Stockage d'un graphe sur un cluster de trois machines. Chaque machine est constituée de ses propres ressources (stockage, mémoire et CPU) et dispose d'une partie des données.

à la résolution de problèmes, la difficulté des calculs peut être surmontée. C'était l'approche employée entre les années 1950 et 1980 pour résoudre les problèmes complexes. Un rapport de recherche américain [31] en offre une brève perspective : les super ordinateurs étaient utilisés principalement pour des applications militaires, biologiques et météorologiques. Des années 1960 jusqu'aux années 1970, CDC est l'entreprise qui démocratise le calcul haute performance, les processeurs restent cependant uniques à chaque machine, ce qui rend leur développement très coûteux. Dans les années 1980, la puissance de calcul de ces super ordinateurs est améliorée grâce au parallélisme : de plus en plus de processeurs disposant de plus de cœurs sont intégrés dans ces machines, ce qui améliore leurs performances.

Concernant demander de l'aide, plusieurs projets de recherche [5, 20, 29] ont été lancés dans cette optique au début des années 1990 et ont formés la base des technologies de calcul distribués accessibles aujourd'hui sous le nom clusters de calcul. L'idée est similaire à celle du stockage distribuée (*cf.* GFS [55]) : coordonner des tâches réparties sur différentes machines, possédant chacune leurs ressources propres. Le calcul du problème est distribué entre plusieurs machines indépendantes, disposant chacune d'espace de stockage, de mémoire et de processeurs ainsi que d'une connexion réseau permettant la synchronisation des étapes du calcul. Comme pour le stockage, la parallélisation de processeurs du commerce permet d'égaler, voire de surpasser, les performances de super-ordinateurs de calcul à moindre coût. Cette approche est aussi extensible puisque si les capacités de calcul sont insuffisantes, un nouveau nœud (c'est à dire une nouvelle machine) peut être ajoutée au cluster sans en perturber les performances. Le calcul sur cluster a été rendu possible par la démocratisation des réseaux locaux rapides, qui permettent aux différentes machines de se coordonner. Cette coordination induit un surcoût qui peut avoir une influence sur les performances de ce type d'approche : la distribution efficace des tâches

améliore les performances à condition qu'il y ait un vrai gain à paralléliser la tâche.

D'autres approches existent, comme le calcul sur grille [119], qui peut être vu comme une variante du calcul sur cluster pour des machines hétérogènes, localisées à des endroits différents. Ces ressources, disponibles via internet et non plus sur un réseau local, sont utilisées lorsqu'elle n'effectuent aucune autre tache.

1.6.2 Technologies actuelles et contexte de recherche

Les technologies de calcul pour les données massives ne sont plus exclusivement réservées aux grandes entreprises et à quelques projets académiques. Des écosystèmes libres, dont le plus connu est Hadoop [122], ont permis de démocratiser ces technologies.

Les alternatives proposées par Hadoop contiennent des logiciels pour l'ensemble de la chaîne de traitement du Big Data : du stockage à l'analyse. HDFS, ou Hadoop File System [122], est le pendant libre de GFS de google. HDFS propose, comme GFS, le stockage distribué de fichiers massifs ainsi que leur réplication pour résister aux pannes et accélérer le requêtage des données. HBase[131] propose des bases de données distribuées sur cluster, basées sur le système de fichiers HDFS.

Différents outils de traitements de données existent dans cet environnement. MapReduce [36] propose un paradigme de calcul basé sur deux opérations simples : Map qui applique une fonction à chaque ligne d'un tableau et Reduce qui fusionne plusieurs lignes pour en tirer un résumé. Avec ces deux opérations, de nombreux algorithmes peuvent être implémentés efficacement de manière distribuée [106]. Ce paradigme de calcul sera développé dans le chapitre 2. Spark [139] est un environnement de calcul fonctionnant sur le même principe que MapReduce. Il utilise des jeux de données distribués appelés RDD (Resilient Distributed Datasets), qui contrairement aux applications MapReduce, peuvent être conservés en mémoire sans avoir à être écrit sur disque entre chaque étape de calcul. Cet environnement permet donc d'implémenter des algorithmes itératifs plus naturellement qu'avec MapReduce. Enfin, pour les applications haut-niveau, Pig [109] propose d'effectuer des traitements sur les jeux de données distribués à l'aide d'un langage de programmation naturel, à la manière de SQL.

Concernant les graphes plus spécifiquement, plusieurs technologies existent dans cet écosystème. Giraph[12] propose une implémentation libre de Pregel [102], un paradigme de calcul distribué basé spécifiquement sur les graphes, qui fonctionne dans l'environnement Hadoop. GraphX [138] est une bibliothèque de Spark conçue pour gérer les graphes. GraphX propose aussi une implémentation de Pregel ainsi que des opérations de type MapReduce spécifiques aux graphes.

1.7 Organisation du mémoire

Dans le Chapitre 2, nous présentons les définitions, notations et prérequis utilisés dans la suite du mémoire. Les définitions et notations concernent principalement les graphes. Les prérequis introduisent des notions de calcul distribué dans le paradigme MapReduce et présentent les technologies utilisées pour réaliser nos travaux de thèses.

Dans le Chapitre 3, nous dressons un état de l'art des algorithmes de dessin de graphe par modèle de force et des algorithmes de dessin de graphe par réduction du stress. Nous présentons d'abord les techniques classiques de dessin de graphe, en passant en revue les différentes forces et optimisations mises au point pour accélérer les algorithmes de dessin par modèle de force. Nous nous concentrons ensuite sur les approches multiéchelles qui ont permis de réduire la complexité temporelle des algorithmes de dessin par modèle de force. Nous présenterons les différents travaux sur le parallélisme des algorithmes de dessin de graphe et les débuts de la distribution de ces algorithmes. Enfin, nous ferons un point sur les techniques de dessin de graphes dynamiques, c'est à dire quand le graphe évolue dans le temps.

Le Chapitre 4 décrit notre première approche pour transposer un algorithme de dessin de graphe dans un environnement MapReduce. A partir des forces décrites par Fruchterman et Reingold [48], nous montrons comment les forces d'attraction sont transposables dans ce modèle. Nous décrivons aussi comment les approximations usuelles des forces de répulsion ne sont pas adaptées à la distribution et au volume des données.

GDAD, le premier algorithme de dessin de graphe dans le paradigme MapReduce, est présenté en Chapitre 5. Nous proposons une force de répulsion adaptée à ce paradigme, qui utilise une structure de donnée distribuable. GDAD est comparé à un autre algorithme de dessin de graphe par modèle de force conçu pour le paradigme MapReduce. Nous comparons aussi les performances GDAD sur un cluster distribué en MapReduce et sur une unique machine dans une implémentation séquentielle classique.

Dans le Chapitre 6, après avoir montré les limites de notre algorithme par modèle de force, nous revenons sur les décompositions multiéchelles de deux algorithmes de dessin de graphe : FM^3 et GRIP. Une méthode permettant de trouver une décomposition multiéchelle est proposée dans le paradigme MapReduce. Cette décomposition étant réalisable efficacement en distribué, nous montrons comment adapter les algorithmes FM^3 et GRIP en MapReduce. Nous comparons notre algorithme, appelé MuGDAD, à MultiGila [9], algorithme de dessin de graphe dans le paradigme Pregel. Nous revenons aussi sur les performances de notre algorithme sur un cluster Big Data et sur une unique machine dans une implémentation séquentielle classique. Ce faisant, nous exhibons les limites d'un algorithme multiéchelle en distribué lorsqu'un niveau n'a pas un volume suffisant pour être distribué, et proposons une interface entre notre

application MapReduce et celle développée en C++.

En conclusion, dans le Chapitre 7, nous ouvrons nos travaux vers des perspectives intéressantes pour le dessin de graphe par modèle de force dans le paradigme MapReduce.

Chapitre 2

Prérequis

2.1 Définition

Nous présentons ici les définitions nécessaires à la compréhension de cette thèse. Les définitions concernant les graphes sont tirées de l'ouvrage de West et al. [135].

2.1.1 Ensembles

Définition 2.1 (Décomposition d'ensemble)**.** Soit E un ensemble. Pour tout i dans l'intervalle $[\![1, p]\!]$, E_i est un sous-ensemble de E. $\{E_i\}_{1 \leq i \leq p}$ est une décomposition de l'ensemble E si et seulement si $\bigcup_{i=1}^{p} E_i = E$.

Définition 2.2 (Partition d'ensemble)**.** Soient E un ensemble et $\{E_i\}_{1 \leq i \leq p}$ une décomposition de l'ensemble E. $\{E_i\}_{1 \leq i \leq p}$ est une partition de E si et seulement si $\forall i, j \in [\![1, p]\!]$, $i \neq j$ on a $E_i \cap E_j = \emptyset$.

Définition 2.3 (Filtration)**.** Une filtration $(E)_n$ est une suite de sous-ensembles de E croissante ou décroissante au sens de l'inclusion.

2.1.2 Graphes

Définition 2.4 (Graphe non-orienté)**.** Soient E et V deux ensembles tels que $E \subseteq \{\{u, v\} \mid u \in V, v \in V\}$. On appelle graphe non-orienté le couple (V, E) et on note $G = (V, E)$.

- Les éléments de l'ensemble V sont appelés les sommets (ou nœuds) du graphe.
- Les éléments de l'ensemble E sont appelés les arêtes du graphe.
- Soit e une arête, les éléments de e sont appelés les extrémités de l'arête.
- Deux nœuds u et v sont adjacents s'il existe une arête dans le graphe dont les extrémités sont u et v.

Définition 2.5 (Graphe orienté)**.** Soient A et V deux ensembles tels que $A \subseteq V^2$. On appelle graphe orienté le couple (V, A) et on note $G = (V, A)$.

- Les éléments de l'ensemble A sont appelés des arcs.
- Soit $a = (u, v)$ un arc, le sommet u (*resp.* v) est appelé la source (*resp.* la destination) de l'arc a.

Définition 2.6 (Sous-graphe)**.** Un sous-graphe $H = (V', E')$ de $G = (V, E)$ est un graphe tel que $V' \subseteq V$ et $E' \subseteq E$. On écrit $H \subseteq G$ et on dit que G contient H.

Définition 2.7 (Graphe induit)**.** Un graphe induit est un sous-graphe obtenu en supprimant un ensemble de nœuds. Soient le graphe $G = (V, E)$ et $T \subseteq V$ On écrit $G[T]$ pour désigner le graphe induit par la suppression des nœuds de $\overline{T} = V \backslash T$ et on dit que $G[T]$ est le graphe induit par T.

Remarque : Supprimer des nœuds d'un graphe implique de supprimer les arêtes ayant pour extrémité l'un des nœuds supprimés.

2.1.3 Degré et chemin

Définition 2.8 (Voisinage d'un sommet)**.** Soient $G = (V, E)$ un graphe et $u \in V$. On appelle voisinage de u dans G, et on note $N_G(u)$ ou $N(u)$, l'ensemble des nœuds adjacents à u.

Remarque : Soient le graphe $G = (V, E)$ et l'ensemble $V' \subseteq V$. Par abus de notation, on écrit $N(V')$ pour désigner l'ensemble $\bigcup_{u \in V'} N(u) \subseteq V$.

Définition 2.9 (Degré d'un sommet)**.** Soient le graphe $G = (V, E)$ et le sommet $u \in V$. On appelle degré de u pour le graphe G, et on note $d_G(u)$ ou $d(u)$, le nombre d'arêtes incidentes au sommet u.

Définition 2.10 (Chemin non orienté)**.** Un chemin non-orienté est un graphe dont les sommets peuvent être ordonnés de manière à ce que les nœuds soient adjacents si et seulement si ils sont consécutifs dans la liste ordonnée.

Définition 2.11 ((u, v)-chemin)**.** Un (u, v)-chemin est un chemin non orienté dont les sommets de degré 1 (les extrémités du chemin) sont les nœuds u et v. Les autres nœuds du chemin sont des nœuds internes.

Définition 2.12 (Longueur d'un chemin)**.** La longueur d'un chemin correspond au nombre d'arêtes dans ce chemin.

Définition 2.13 (Distance dans un graphe)**.** Soient le graphe $G = (V, E)$ et deux nœuds $u, v \in V$. Si G contient un (u, v)-chemin, la distance entre u et v, notée $d_G(u, v)$ ou $d(u, v)$ est la longueur du plus court (u, v)-chemin. Si aucun (u, v)-chemin n'existe entre u et v, $d(u, v) = \infty$.

Définition 2.14 (Graphe pondéré)**.** Un graphe pondéré est un graphe avec des étiquettes numériques sur ses arêtes.

Définition 2.15 (Distance valuée dans un graphe)**.** La distance valuée dans un graphe pondéré dont les poids sont positifs est la somme du poids des arêtes.

2.1.4 Cas particuliers de graphes

Définition 2.16 (Graphe complet)**.** On dit que le graphe G est complet si ses sommets sont adjacents deux à deux.

Définition 2.17 (Graphe connexe)**.** On dit que le graphe G est connexe si chaque paire de sommets est dans un chemin.

Définition 2.18 (Stable)**.** Soit le graphe $G = (V, E)$. Un stable (ou un ensemble indépendant) est un sous-ensemble $V' \subseteq V$ tel que le graphe $G[V']$ ne contient pas d'arêtes.

Définition 2.19 (Graphe Quotient)**.** Soit le graphe $G = (V, E)$. Soit $\{V_i\}_{1 \leq i \leq p}$ une partition de V. On note $\sim$ la relation d'équivalence induite par la partition $\{V_i\}_{1 \leq i \leq p}$ des nœuds de V. On note alors $V/\sim$ l'ensemble quotient. Soit $u \in V$. On note $[u]_\sim$ le représentant canonique de $V/\sim$ associé à u.

Le graphe quotient $Q = (V', E')$ est le graphe tel que $V' = V/\sim$ et $E' = \{([u]_\sim, [v]_\sim) | [u]_\sim \neq [v]_\sim \text{ et } (u, v) \in E(G)\}$.

2.2 Paradigme de calcul Big Data

Nous décrivons dans cette section deux paradigmes de calculs Big Data : le paradigme MapReduce et le paradigme Pregel. MapReduce est un paradigme de calcul général pour les données massives. Pregel est un paradigme de calcul utilisé pour manipuler de grands graphes.

2.2.1 Paradigme MapReduce

Le paradigme MapReduce [36] est présenté en 2008 par Dean et al. Dans un programme MapReduce, deux types d'opérations sont chaînées pour réaliser un calcul : l'opération de Map et l'opération de Reduce. L'ensemble de ces opérations forme un *job* MapReduce.

Dans le paradigme MapReduce, les données sont représentées par des paires clé-valeurs. Les opérations Map et Reduce prennent en entrée un ensemble de paires clé-valeurs et renvoie un ensemble de paires clé-valeurs. Plus précisément, soient K, K' et K'' trois ensembles de clés et V, V' et V'' trois ensembles de valeurs. On utilise l'opérateur $\mathfrak{P}$ pour décrire l'ensemble puissance

(*i.e.* l'ensemble des parties) d'un ensemble. Pour un job, les fonctions Map et Reduce s'écrivent :

$$\begin{aligned} \text{map} : K \times V &\rightarrow \mathfrak{P}(K' \times V') \\ (k, v) &\rightarrow \{(k', v') | (k', v') \in K' \times V'\} \end{aligned}$$

$$\begin{aligned} \text{reduce} : K' \times \mathfrak{P}(V') &\rightarrow \mathfrak{P}(K'' \times V'') \\ (k', \{v' | v' \in V'\}) &\rightarrow \{(k'', v'') | (k'', v'') \in K'' \times V''\} \end{aligned}$$

Ces opérations sont pensées pour permettre une distribution aisée des calculs. Sur un cluster de calcul, chaque machine traite une partie des données seulement, de manière à tirer parti du stockage distribué des fichiers.

Opération Map

L'opération Map permet de transformer les données en entrée vers un état intermédiaire utilisable par l'opération Reduce. Selon l'application, elle peut être utilisée pour filtrer les données, dupliquer les données *etc.* Dans l'implémentation distribuée du paradigme, les machines effectuant l'opération Map sont appelées *mapper*. Chaque *mapper* ne travaille alors que sur une partie des données.

La clé des éléments en sortie de la fonction Map permet de déterminer les valeurs réduites ensemble dans l'opération de Reduce.

Opération Reduce

L'opération de Reduce transforme les valeurs ayant la même clé en une paire clé-valeur. Selon l'application, l'opération de Reduce permet de transformer la sortie du Map en un indicateur statistique, en un jeu de données triées *etc.* Dans l'implémentation distribuée du paradigme, les machines effectuant l'opération Reduce sont appelées *reducer*. Chaque *reducer* travaille sur une partie des données.

Shuffle des données

Entre l'opération Map et l'opération Reduce, le jeu de données subit une modification. L'opération Map renvoie un ensemble de paires clé-valeurs et l'opération Reduce prend en entrée une paire clé-ensemble de valeurs. L'étape de Shuffle permet de faire cette transition.

Lors du Shuffle des données, les données en sortie de l'opération Map sont transmises sur le réseau en direction des *reducers*. Les paires ayant la même clé sont transmises au même *reducer*. Les paires clé-ensemble de valeurs sont formées de cette manière.

Ce transfert peut représenter un point de blocage important de l'implémentation distribuée. Lorsque l'on manipule de grands jeux de données, la quantité d'information échangée sur le réseau peut être importante (jusqu'à plusieurs fois la taille du jeux de données, selon l'application). Il convient donc de faire attention au volume de données transmises lors de cette étape pour améliorer les performances de l'algorithme.

2.2.2 Paradigme Pregel

Calcul de graphe distribué

Le paradigme Pregel [102] est développé pour pallier le manque d'environnements de calcul spécifiques aux graphes sur clusters. Lumsdaine et al. [101] présentent plusieurs défis au développement d'algorithmes de graphes parallèles :

- le calcul dépend des données et ne peut être réalisé sur une structure connue a priori,
- la structure de graphe rend difficile un partitionnement efficace des données,
- l'accès au données en mémoire est compliqué par la mauvaise localité des données de graphe.

Ces problèmes se posent de la même manière lorsque les données sont partitionnées mais le problème de localité est amplifié.

Le paradigme Pregel est une manière naturelle d'exprimer les algorithmes itératifs de graphes. Il est notamment utilisé pour l'algorithme PageRank de Google, pour l'algorithme du chemin le plus court *etc.*, *cf.* Malewicz et al. [102].

Modèle de calcul Pregel

Malewicz et al. [102] proposent un paradigme de calcul inspiré du modèle de parallélisme appelé Bulk Synchronous Parallel (abbrégé BSP) où les algorithmes sont décomposés en superétapes (ou rondes) dans lesquelles les calculs sont réalisés en parallèle puis synchronisés. Une ronde de Pregel se décompose en trois étapes : la création de messages, l'envoi de messages et le changement d'état. L'exécution de ces trois étapes se fait à l'aide des informations dont chaque nœud dispose (son état et les messages qu'il a reçus). Ce paradigme est d'ailleurs aussi connu sous le nom de "Think like a Vertex", souvent abrégé en TLAV.

Lors de la création de messages, Pregel applique une fonction sur chaque nœud en parallèle. Aucun ou plusieurs messages peuvent être créés par nœud lors de cette étape.

Les arêtes entrent en jeu au moment de l'envoi des messages. Les messages sont généralement envoyés aux voisins des nœuds. Cependant, il est possible d'envoyer un message à n'importe quel nœud du graphe.

Enfin, chaque nœud lit les messages reçus et modifie son état en conséquence. Cette étape marque la fin de la ronde. Une nouvelle phase de création de messages débute alors.

Les algorithmes implémentés en Pregel s'arrêtent généralement lorsqu'il n'y a plus de messages transmis entre les nœuds.

Correspondance MapReduce

Le paradigme Pregel peut être décrit à l'aide des opérations utilisées en MapReduce. On distingue ainsi trois étapes qui peuvent être associées à une opération dans le paradigme MapReduce : la création de messages par chaque sommet (correspondant à une opération Map sur les sommets), le transfert des messages (opération de Shuffle) et le changement d'état en fonction des messages reçus (opération de Reduce ayant pour clé un sommet et pour valeurs les messages transmis à ce sommet).

2.3 Spark et le calcul en mémoire

Spark [139] est une bibliothèque de calcul distribuée tirant parti de la mémoire des machines du cluster. Spark utilise des RDD, une structure de données distribuée et stockée en mémoire, pour représenter les grands jeux de données. Spark implémente le paradigme MapReduce et dispose de bibliothèques dédiées au calcul sur les graphes, comme GraphX [138].

Resilient Distributed Datasets

Spark fonctionne à partir de jeux de données robustes appelés Resilient Distributed Datasets (abrégé en RDD). Ces jeux de données distribués constituent le cœur des opérations de Spark. Les RDD utilisent une structure de données immuable.

Contrairement à l'implémentation Hadoop de MapReduce, plusieurs RDD peuvent coexister au sein d'un même job. Les RDD sont évaluées de manière paresseuse de manière à optimiser les opérations effectuées à la suite.

Au sein d'un job, Spark conserve un graphe orienté acyclique des transformations effectuées sur les RDD, de manière à pouvoir recalculer l'état de l'algorithme à tout instant. En cas de panne, l'historique des opérations peut être répété à partir d'une RDD intacte. Pour un algorithme déterministe, les mêmes RDD sont alors générées à nouveau par l'historique des opérations.

Variables partagées et broadcast

Lors de l'exécution d'un algorithme dans un environnement MapReduce ou Pregel, il peut être nécessaire que certaines variables soient accessibles sur

l'ensemble des machines. Spark propose deux solutions pour partager des variables : le broadcast et les accumulateurs.

Le broadcast permet de rendre accessible une variable à toutes les machines. Cette variable est transmise à chaque machine via le réseau de manière efficace et la variable est stockée sur le cache de chaque machine.

Les accumulateurs (ou Accumulators) sont des variables accessibles par l'ensemble des machines qui peuvent uniquement être incrémentées. La variable peut être lue par l'ensemble des machines et modifiée par un accès concurrent.

Chapitre 3

État de l'art

Dans cet état de l'art, nous présentons en détails les algorithmes de dessin de graphe par analogie physique. Trois aspects différents du problème sont mis en lumière :

- Le **modèle physique sous-jacent** permet de définir les interactions entre les nœuds du graphe. L'état d'équilibre de ces modèles définit le dessin optimal.
- La **méthode de résolution** permet de trouver l'état d'équilibre du modèle physique sous-jacent. Par ces méthodes itératives, les nœuds convergent vers leur position optimale.
- L'**approche multiéchelle** permet d'accélérer la convergence du dessin. Cette approche décompose le problème initial en une série de problèmes de difficulté décroissante. Leur résolution permet une convergence rapide du problème initial.

Dans la dernière section, nous présentons les travaux de parallélisation et de distribution des algorithmes de dessin par analogie physique.

3.1 Notations usuelles

Dans la suite de ce chapitre, nous utilisons les notations suivantes. Soient u et v deux nœuds de V. On note p_u le vecteur de coordonnées représentant la position du nœud u. On note d_{uv} la distance graphe entre les nœuds u et v. On note x_u (resp. y_u) pour désigner l'abscisse (resp. l'ordonnée) de la position de u dans le plan.

Les notations $\|\cdot\|$ et $\|\cdot\|_1$ désignent respectivement la norme euclidienne et la norme 1.

Les constantes relatives aux modèles physiques (*cf.* Section 3.2) sont notées k_i, $i \in \mathbb{N}$, ou k s'il n'y a pas d'ambiguïté.

Algorithme	Fonction d'énergie
Quinn	$E = \sum_{(u,v)\in E} \frac{k_1}{2} \cdot \lVert p_v - p_u \rVert_1^2 - \sum_{(u,v)\notin E} k_2 \cdot \lVert p_v - p_u \rVert_1$
Eades	$E = \sum_{(u,v)\in E} k_1 \lVert p_v - p_u \rVert (\log \lVert p_v - p_u \rVert - 1) + \sum_{(u,v)\in V^2} \frac{k_2}{\lVert p_v - p_u \rVert}$
Kamada	$E = \sum_{(u,v)\in V^2} \frac{1}{2} \cdot k_{uv} \cdot (\lVert p_v - p_u \rVert - d_{uv})^2$
Fruchterman	$E = \sum_{(u,v)\in E} \frac{1}{3 \cdot k} \cdot \lVert p_v - p_u \rVert^3 - \sum_{(u,v)\in V^2} k^2 \cdot \log(\lVert p_v - p_u \rVert)$

FIGURE 3.1 – Énergie du système à minimiser selon les forces choisies pour quatre exemples d'algorithmes de dessin classiques : : Quinn [116], Eades [42], Kamada et Kawai [84] et Fruchterman et Reingold [48]

3.2 Modèles physiques et dessin de graphe

Les algorithmes de dessin par modèle de forces se basent sur un modèle physique sous-jacent dont l'état d'équilibre correspond au dessin optimal. Dans cette section nous décrivons les modèles de forces couramment utilisés dans les algorithmes de dessin. Dans ces algorithmes, le modèle de force sous-jacent s'exprime sous la forme d'une fonction d'énergie à optimiser ou sous la forme de forces à appliquer aux nœuds du graphe. Le modèle physique seul détermine la position optimale des nœuds. Les modèles physiques sous-jacents sont présentés en Section 3.2.1.

Le dessin de graphe par modèle de force peut être associé à des contraintes sur le placement des nœuds. Ces contraintes s'expriment sous la forme d'inégalités dont les termes sont fonction de la position des nœuds. Ces contraintes ne définissent pas un modèle physique à proprement parler mais elles influencent le dessin optimal du graphe. Nous présentons plus en détail les algorithmes utilisant de telles contraintes pour dessiner des graphes dans la Section 3.2.3.

3.2.1 Analogies physiques classiques

Quinn [116] développe en 1979 un algorithme permettant de placer automatiquement les composants électroniques d'une carte à puce. Une telle carte peut être modélisée par un graphe dont les nœuds sont les composants à placer et les arêtes les pistes qui les relient. Les objectifs de cet algorithme sont différents de ceux du dessin de graphe classique : il s'agit de placer tous les composants non pas pour maximiser une esthétique mais relativement les uns aux autres de manière à avoir une première ébauche de la carte. Pour trouver leur position, Quinn propose un modèle physique qui préfigure les algorithmes de dessin de graphe. Les composants reliés par une piste subissent une force d'attraction suivant la loi de Hooke (*i.e.* la force de rappel des ressorts). Les composants non-reliés subissent deux à deux une force de répulsion constante. Quinn utilise la norme 1 et pas de racines ni d'inverse dans son analogie physique pour

limiter les opérations mathématiques complexes et réduire les calculs.

En 1984, Eades [42] propose le modèle physique suivant pour dessiner les graphes. Les nœuds des graphes, *i.e.* particules du modèle physique sous-jacent, sont modélisés par des anneaux et les arêtes reliant les nœuds du graphe sont modélisées par des pseudo-ressorts. Une force d'attraction de type logarithmique est appliquée entre les nœuds adjacents. Les forces de répulsion sont appliquées aux nœuds non adjacents et ces forces suivent une loi en carré inverse que l'on retrouve dans les modèles physiques en mécanique ou en électrostatique. Contrairement à Quinn [116] qui cherche la position relative des nœuds, Eades cherche à rendre la visualisation du graphe claire. Les forces sélectionnées permettent d'optimiser la qualité du dessin. Par exemple, une loi logarithmique est préférée par Eades car les nœuds sont trop rapprochés lors du dessin avec la loi de Hooke.

Kamada et Kawai [84] proposent le modèle physique suivant : les forces ne sont pas scindées en forces d'attraction et de répulsion mais interviennent entre tous les nœuds. Dans ce modèle, les nœuds sont connectés deux à deux par un ressort dont la longueur nominale est égale à la distance graphe des nœuds. Cette unique force agit en tant que force d'attraction ou de répulsion selon la valeur du rapport entre la distance dans le dessin et la distance graphe. L'équilibre recherché est alors un dessin où la distance pour chaque paire de nœuds est la plus proche possible de leur distance graphe, *i.e.* une isométrie. Lorsqu'un un poids de 1 est associé à chaque arête, la distance entre chaque paire de nœuds (all-pair shortest path ou APSP) peut être obtenue en utilisant des arbres de parcours en largeur. La création de ces arbres a pour complexité $O(|V| + |E|)$ et doit être réalisée pour chaque nœud, pour une complexité totale de $O(|V|^2)$. Ce calcul reste aujourd'hui difficile à mener pour les grands graphes.

Fruchterman et Reingold [48] proposent en 1991 un modèle similaire à celui d'Eades [42]. Les forces de répulsion de ce modèle sont appliquées entre chaque paire de nœuds. Les forces d'attraction sont proportionnelles au carré de la distance et les forces de répulsion sont inversement proportionnelles à la distance. Fruchterman et Reingold harmonisent ces deux forces grâce au paramètre k :

$$k = C \cdot \sqrt{\frac{a}{|V|}}$$

a représente l'aire de la surface du dessin et C est une constante réglée expérimentalement.

La relation entre la fonction d'énergie et l'expression des forces est la même qu'en mécanique classique. Les dérivées partielles de la fonction d'énergie permettent d'obtenir l'expression des forces. Inversement, l'intégration des forces selon les différentes dimensions permet de retrouver la fonction d'énergie. L'état d'équilibre correspond au minimum d'énergie : la somme des forces appliquées à chaque nœud est nulle. La fonction d'énergie associée à ces analogies

Algorithme	Complexité	Répulsion	Attraction
Quinn	$O(\|V\|^2)$		
Eades	$O(\|V\|^2)$		
Kamada	$O(\|V\|^2)$		
Fruchterman	$O(\|V\|^2 + \|E\|)$		

FIGURE 3.2 – Complexité des modèles physiques et schéma des forces appliquées aux nœuds pour quatre algorithmes classiques de dessin de graphe : Quinn [116], Eades [42], Kamada et Kawai [84] et Fruchterman et Reingold [48]

physiques est résumée dans le tableau Figure 3.1. Pour ces différents modèles physiques, le nombre d'interactions calculées est de $O(|V|^2)$, *cf.* Figure 3.2.

Les algorithmes de dessin de graphe classiques présentés dans cette section sont illustrés en Figure 3.3 pour les jeux de données imdb_small, graphe connectant les acteurs ayant joué dans les mêmes films dans la base de données Imdb, et un tore généré automatiquement.

3.2.2 Autres analogies

Les modèles physiques classiques s'expriment facilement sous la forme de forces ou de fonctions d'énergie pour deux raisons : l'intensité des forces s'exprime par des fonctions régulières au sens de la dérivation et ces fonctions ne s'expriment qu'à l'aide de la distance entre les nœuds dans le dessin. Nous présentons ici deux analogies physiques dont l'expression sous la forme de force ou d'énergie est plus complexe.

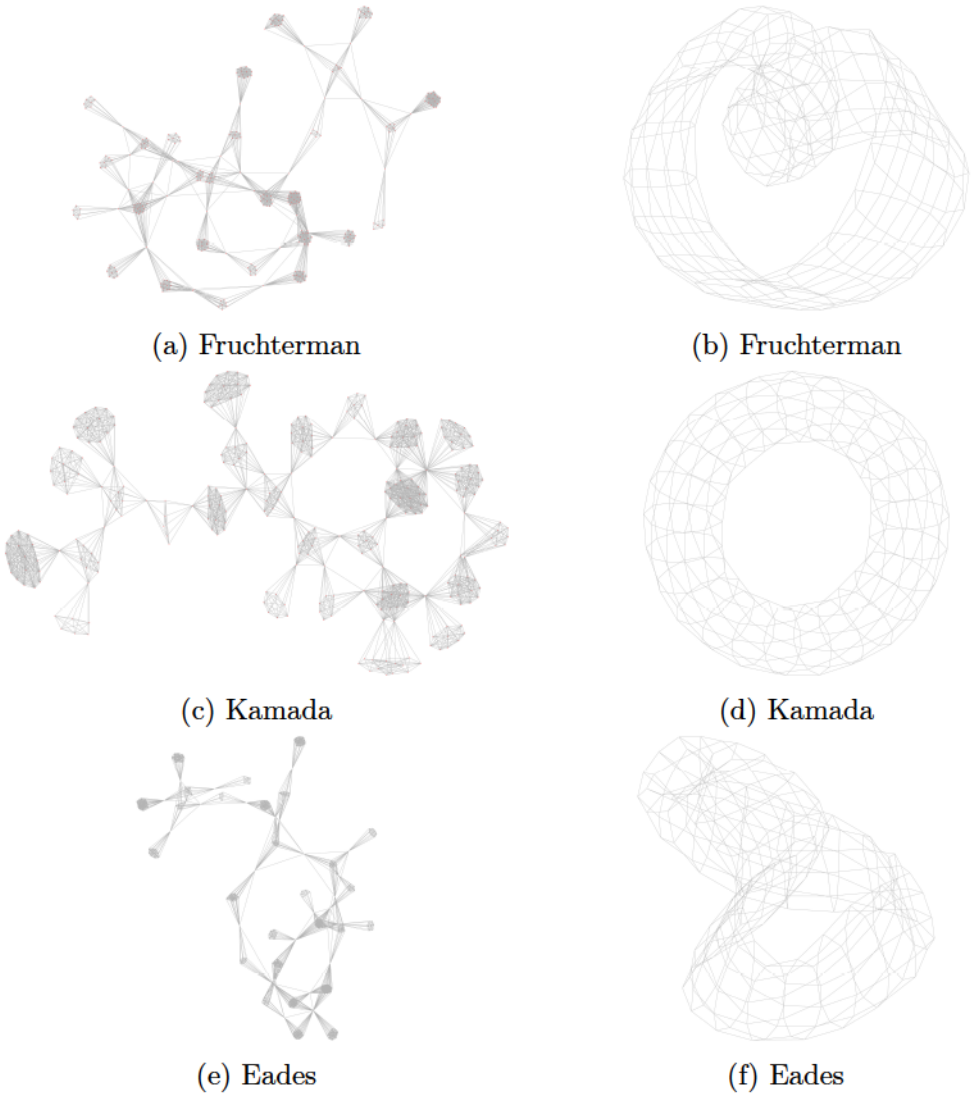


FIGURE 3.3 – Dessin de deux graphes, imdb_small et un tore, à l'aide des algorithmes de et Fruchterman et Reingold [48], Kamada et Kawai [84] et Eades [42]. imdb_small est un graphe de réseau d'acteurs connectés lorsqu'ils jouent dans le même film (298 nœuds et 2058 arêtes). Le tore est un graphe généré automatiquement, contenant 25 anneaux de 10 nœuds (250 nœuds et 500 arêtes).

Force entre un nœud et une arête

Bertault [21], puis Simonetto et al. [123], proposent une approche combinant les approches mathématiques de dessin de graphe planaire et les algorithmes de dessin de graphe par modèle de force. L'algorithme de Bertault assure que le dessin final n'ajoute pas de croisements d'arêtes au dessin initial. Bertault ajoute une force de répulsion entre les nœuds et les arêtes aux forces définies par Fruchterman et Reingold [48]. Les nœuds du graphe sont repoussés par les arêtes de manière orthogonale lorsque les nœuds et les arêtes sont proches. Lors du calcul des forces pour chaque nœud, deux termes opposés sont créés. Le premier terme correspond aux forces de répulsion des arêtes vers les nœuds. Le second terme, la force opposée, s'applique depuis les nœuds vers les arêtes. Puisque les arêtes ne sont pas déplacées directement, ce sont les deux extrémités des arêtes à qui on applique la moitié de la force de répulsion.

La force de répulsion entre un nœud u et une arête (a, b) utilise un nœud virtuel particulier pour être calculé. Ce nœud i est la projection du nœud u sur le segment $[ab]$. L'intensité de la force s'exprime alors à l'aide de la distance entre les nœuds u et i. L'intensité de cette force a pour seul paramètre la distance entre nœuds et arêtes.

Champ magnétique

L'orientation des arêtes peut être un paramètre à contrôler grâce à un modèle physique. Sugiyama et Misue [125] s'inspirent des champs magnétiques pour décrire un nouveau type de forces qui influent directement sur la direction des arêtes. Dans leur modèle, les arêtes sont remplacées par des aimants et plongées dans un champ magnétique. La nouvelle force s'applique aux terminaisons des arêtes et est fonction de l'angle formé entre l'arête et le champ magnétique. Les forces magnétiques permettent de définir une orientation optimale pour les arêtes selon le champ magnétique choisi (parallèle, central ou radial). Les auteurs utilisent ces forces pour implémenter des algorithmes de dessin d'arbres et de graphes.

La force appliquée par le champ magnétique dépend de l'angle formé entre le champ magnétique et l'arête. L'énergie potentielle de ce type de forces s'exprime à l'aide du produit scalaire entre le vecteur formé par l'arête et le champ magnétique : soient $(u, v) \in E$ et B le vecteur représentant le champ magnétique au milieu de cette arête, $E = -\mathbf{uv} \cdot B$.

3.2.3 Contraintes et dessin

L'ajout de contraintes est aussi un champ d'exploration pour le dessin de graphe. Aux forces classiques s'ajoutent des règles à respecter lors du dessin pour délimiter le problème. Les contraintes sont principalement utilisées dans

des algorithmes de dessin où la résolution (*cf.* Section 3.4) est effectuée par optimisation en utilisant le modèle physique proposé par Kamada et Kawai [84]. Dans ces conditions, le dessin sous contrainte se ramène à un problème de programmation quadratique bien connu en mathématiques [41].

La limite entre les contraintes et les modèles physiques n'est pas toujours claire. Pour Dengler et al. [37], les contraintes sont utilisées pour spécifier une forme attendue, *e.g.* des couches successives pour les graphes hiérarchiques, puis intégrées à l'énergie du système sous la forme d'une fonction de coût. Kamps et Kleinz [85] adoptent cette approche pour les contraintes d'orientation des arêtes.

Pour le dessin de graphe, la littérature se concentre sur les contraintes dont les termes sont des fonctions affines de la position des nœuds et de la distance entre les nœuds [85]. Les contraintes ainsi exprimées permettent par exemple :

- L'alignement horizontal ou vertical de nœuds (contraintes de type $x_u = x_v$)
- La délimitation de l'espace du dessin (*e.g.* $y_u \leq 5$)
- La position relative de certains nœuds (*e.g.* $x_u \leq x_v$)

Apport des contraintes au problème du dessin

Plusieurs objectifs peuvent être recherchés en ajoutant des contraintes à la fonction de coût.

He et al. [70] proposent d'utiliser les contraintes pour spécifier le dessin attendu. Ceci peut s'appliquer au dessin de graphe ayant une structure spécifique. Par exemple, Dwyer et Koren [40] utilisent des contraintes pour dessiner des graphes par couches de manière à assurer qu'un maximum d'arcs sont orientés vers le bas du dessin. Ces contraintes peuvent aussi être utilisées pour conserver la carte mentale lors de dessins successifs, *i.e.* limiter les modifications de positions.

Lin et Eades [99] tentent de concilier approche déclarative et algorithme de dessin. L'une de leurs motivations est de pouvoir obtenir un dessin toujours identique en sortie de l'algorithme. Les modèles physiques pour le dessin possèdent en effet de nombreuses symétries (*e.g.* rotations, symétries axiales *etc.*) et plusieurs états d'équilibres locaux (*i.e.* dont la fonction d'énergie est un minimum local). L'état d'équilibre en sortie varie donc d'une exécution à l'autre, même lorsque l'algorithme de dessin par modèle de force est déterministe.

Kamps [85] montre que fixer la position d'un certain nombre de nœuds contribue à diminuer le nombre de dimensions du problème et que de spécifier la position relative des nœuds permet de réduire l'espace d'exploration, accélérant ainsi le calcul. Cependant, Kamps souligne que le gain obtenu par ces contraintes n'est généralement pas suffisant au regard de la complexité du problème.

Compatibilité des contraintes

L'ajout de contraintes complique la résolution puisqu'il faut s'assurer que ces contraintes sont compatibles entre elles. Aucun dessin ne peut répondre à des contraintes contradictoires. Böhringer et Paulisch [23] proposent une recherche binaire des contraintes pour trouver celles à désactiver, ce qui nécessite $O(c^2 \log c)$ opérations avec c le nombre de contraintes. Dengler et al. [37] proposent d'appliquer les contraintes par ordre croissant de restrictivité. Mettre en œuvre une contrainte très restrictive empêche d'autres contraintes d'être appliquées et c'est la réalisation du plus de contraintes possibles qui est donc privilégiée.

3.3 Simplification du modèle physique

Nous présentons les méthodes de simplification ou d'approximation des modèles physiques. Les analogies physiques classiques modélisent $O(|V|^2)$ interactions ce qui ne permet pas d'optimiser efficacement le dessin lorsque le nombre de nœuds du graphe croît. Nous distinguons deux approches distinctes pour ce problème.

La première consiste à limiter les interactions topologiques modélisées par la fonction d'énergie de Kamada et Kawai. Les travaux de cette approche cherchent à limiter le nombre d'interactions par des méthodes basées sur la topologie du graphe. Ces méthodes sont présentées en Section 3.3.

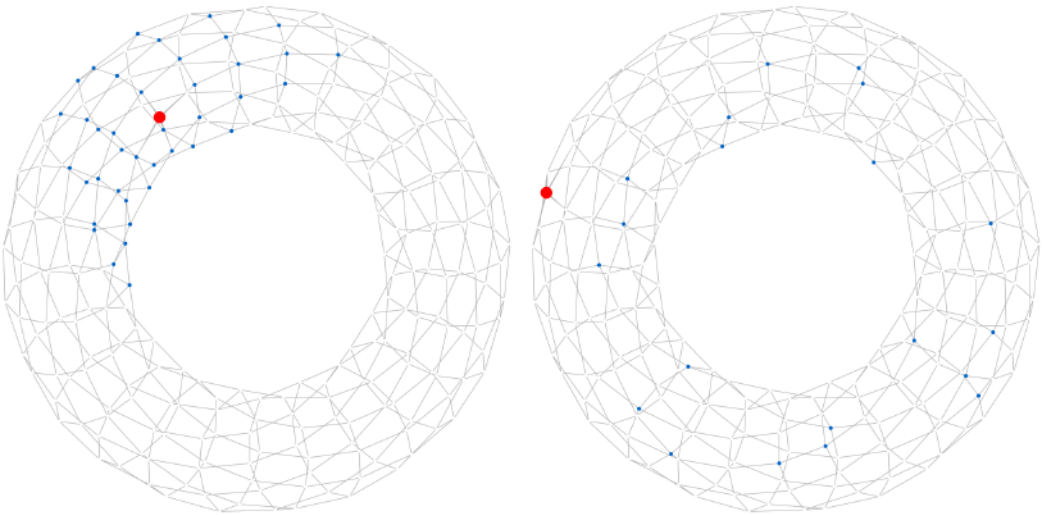
La seconde approche vise à limiter les interactions entre les nœuds considérés comme des particules. Ces approches s'inspirent de la recherche en physique, et notamment des approches à N corps, pour approximer le calcul des forces. Ces différentes méthodes sont présentées en Section 3.3.2.

3.3.1 Topologie et échantillonage des interactions

La sélection, *i.e.* l'échantillonnage, d'une partie des $O(|V|^2)$ interactions permet de simplifier le modèle physique. Le processus de sélection utilise la topologie du graphe pour choisir les interactions à conserver. En basant l'approximation de ces interactions sur la topologie du graphe, il est possible de sélectionner l'échelle des détails à faire apparaître dans le dessin. Conserver en priorité les interactions avec les voisins proches permet de placer correctement les nœuds relativement à leur voisinage. Au contraire, conserver les interactions avec les nœuds éloignés privilégie la structure globale du graphe.

Interactions à différentes échelles

Les deux techniques présentées ici, les k-voisinages et la méthode par pivot, travaillent à deux échelles différentes. Pour les k-voisinages, les interactions



(a) k-voisinage. (b) Pivots.

FIGURE 3.4 – Sélection des interactions sur un graphe torique. Le nœud rouge vif interagit avec les nœuds colorés. *(a)* Interactions avec les nœuds à distance 3 ou moins. *(b)* Interactions avec les 20 pivots sélectionnés.

locales sont privilégiées aux autres interactions. Pour la méthode par pivot, ce sont les interactions avec quelques nœuds situés dans l'ensemble du graphe qui sont retenues. Ces deux cas sont illustrés en Figure 3.4.

k-voisinage Harel et Koren [67] proposent de calculer le dessin localement. Le calcul des interactions est limité aux voisins à distance graphe k ou moins.

Les interactions n'interviennent qu'entre les nœuds du k-voisinage. Pour les graphes de degré borné, le nombre d'interactions est constant. Le nombre total d'interactions calculées est alors de $O(|V|)$.

Méthode par pivot Koren et al. [51, 90] proposent une méthode basée sur des pivots, un sous-ensemble des nœuds. Seules les interactions avec les pivots sont conservées. La structure globale du graphe est obtenue grâce aux interactions avec les pivots. Les pivots sont sélectionnés aléatoirement de manière uniforme dans le graphe.

Koren et al. proposent d'utiliser un nombre de pivots m compris entre 30 et 100 de manière à obtenir une structure globale satisfaisante tout en conservant des performances acceptables. Ceci représente $O(m \cdot |V|)$ interactions avec des pivots.

Modèle physique privilégié

Parmi les modèles physiques présentés en Section 3.2.1, l'échantillonnage des interactions a été appliqué principalement pour le modèle de Kamada et Kawai [84]. Ce choix repose sur deux raisons.

Dans les modèles physiques de Eades [42] et Fruchterman et Reingold [48], la topologie du graphe est utilisée pour le calcul des forces d'attraction entre nœuds voisins. Au contraire, le modèle proposé par Kamada et Kawai repose exclusivement sur la distance graphe entre chaque paire de nœuds. Les interactions entre les nœuds éloignés ont une intensité comparable à celle entre nœuds proches.

Kamada et Kawai cherchent à obtenir un dessin le plus proche possible de l'isométrie, *i.e.* un dessin où la distance entre les nœuds dans le dessin est proche de leur distance graphe. La distance à trois points suffit à placer un nœud de manière unique dans le plan. Le modèle de Kamada et Kawai est donc trop contraint puisqu'il définit la distance optimale à l'ensemble des nœuds du graphe. En pratique, très peu de graphes peuvent être dessinés comme des isométries (les chaînes et les triangles font partie de cette famille). Réduire le nombre d'interactions du modèle de Kamada et Kawai permet de rendre plus simple le placement optimal des nœuds.

Application de ces techniques

Koren et al. [51, 90] utilisent les k-voisinages et les pivots pour définir une fonction d'énergie de Kamada et Kawai [84]. Le modèle ainsi défini permet d'avoir des détails à l'échelle locale grâce aux k-voisinages et à l'échelle globale grâce aux pivots. $O(|V|)$ interactions sont calculées dans cette simplification du modèle.

Harel et Koren [67] et GRIP [50] utilisent les k-voisinages en association avec une approche multiéchelle. La position des nœuds est optimisée dans leur k-voisinage : le dessin est localement optimal. L'approche multiéchelle assure que la structure globale du graphe est correctement représentée, *cf.* Section 3.5.

3.3.2 Approches à N corps et forces approchées

Les forces de répulsion des modèles physiques présentés en Section 3.2.1 interviennent entre chaque paire de nœuds de manière à les répartir sur l'ensemble du plan. Ces forces entraînent le calcul de $O(|V|^2)$ interactions qui dépendent uniquement de la position des nœuds. L'agrégation de sous-ensembles de nœuds permet le calcul d'interactions approchées entre les nœuds et les agrégats. Les algorithmes utilisant ce type d'approximation s'inspirent de plusieurs travaux de recherche en physique. Pour cette raison, dans la suite de cette section, nous parlons de particules pour désigner les nœuds en interaction et de pseudo-particules pour désigner les agrégats. Il n'est donc plus

question de graphe dans cette section.

Dans les modèles physiques, l'intensité des forces de répulsion décroît rapidement avec la distance entre les particules, *cf.* Section 3.2.1. Les particules proches ont une influence forte et les particules éloignées ont une influence faible. Les forces appliquées par des particules éloignées peuvent cependant se cumuler pour créer une influence non négligeable sur une particule. Les méthodes d'approximation des forces de répulsion doivent :

- définir la limite entre particules proches et éloignées,
- agréger la contribution des particules éloignées.

Nous présentons trois méthodes d'approximation des forces de répulsion. La première est une heuristique proposée par Fruchterman et Reingold [48] pour réduire le nombre de forces de répulsion entre particules. Nous évoquons deux méthodes de sommations rapides utilisées en physique pour évaluer les forces : la méthode de Barnes et Hut [15] et la méthode multipôle rapide.

Grille régulière

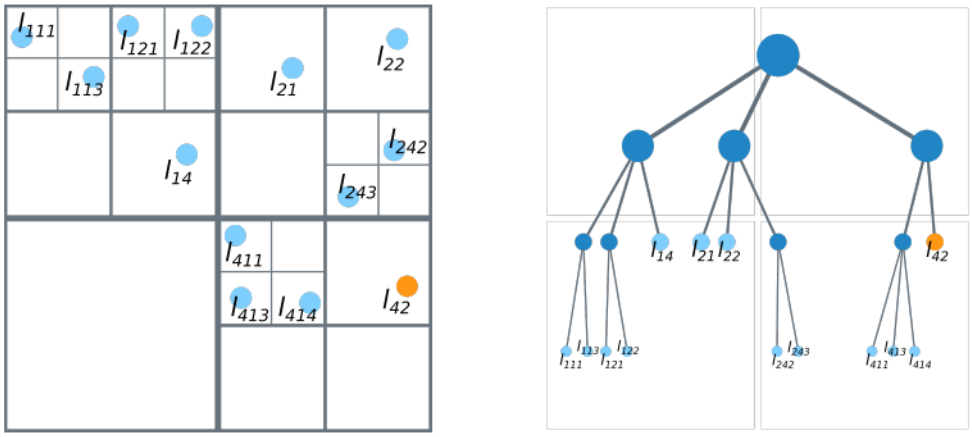
Méthode Fruchterman et Reingold [48] ne prennent plus toutes les particules en compte pour effectuer le calcul des forces de répulsion. Le plan est divisé à l'aide d'une grille régulière. Les forces de répulsion appliquées à une particule sont approximées en considérant uniquement les particules dans les neuf cellules adjacentes de la grille. Les particules éloignées sont celles qui ne se trouvent pas dans les 9 cellules adjacentes de la grille.

A la différence de la sélection topologique par k-voisinages, *cf.* Section 3.3, les particules utilisées pour calculer les forces changent à chaque modification du dessin. Cette sélection est indépendante de la topologie du graphe. Ces deux facteurs font que nous considérons cette heuristique comme une approximation des interactions et non comme un échantillonage.

Erreur et complexité La contribution des particules éloignées est ignorée. De fait, cette approximation n'offre pas de garantie théorique sur la précision du calcul. Cette approximation est donc une heuristique.

La répartition des particules dans le plan est arbitraire. Le nombre d'interactions à calculer n'est pas modifié dans le pire des cas : si la plupart des particules sont dans quelques cases de la grille, le nombre d'interactions à calculer reste de $O(|V|^2)$. En pratique, le temps de calcul est amélioré : seule une fraction des interactions est prise en compte.

Utilisation Pour Fruchterman et Reingold [48], cette technique permet d'accélérer le temps d'exécution de leur algorithme de dessin sans que la qualité du dessin soit trop altérée. Walshaw [132] utilise cette approximation dans son algorithme de dessin de graphe multiéchelle pour réduire le temps de calcul du dessin de chaque niveau.



(a) Décomposition récursive en quadtree. (b) Arbre de décomposition du quadtree.

FIGURE 3.5 – Décomposition récursive en quadtree de manière à ce que chaque nœud soit dans sa propre cellule. Cette décomposition est associée à un arbre de décomposition dont chaque nœud est associé à une pseudo-particule. La particule jaune est celle pour laquelle on cherche à calculer les forces approchées.

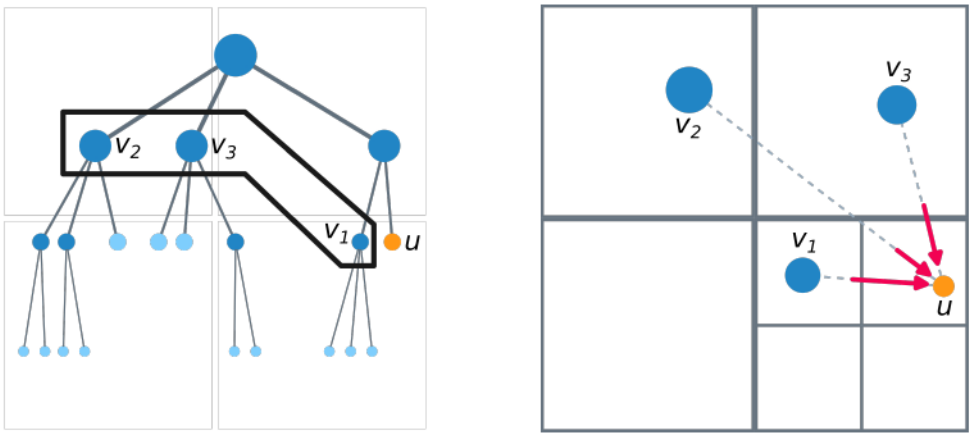
Barnes et Hut

Décomposition spatiale récursive La méthode de Barnes et Hut [15] utilise une décomposition spatiale récursive pour calculer les forces de répulsion. Le quadtree divise le plan du dessin en quatre sous-plans qui sont eux-mêmes subdivisés en quatre sous-plans jusqu'à ce que chaque nœud soit seul dans sa partition. La décomposition récursive par quadtree est illustré en Figure 3.5. Le nonatree est une décomposition du plan en neuf sous-plans. La méthode de Barnes et Hut n'est cependant pas limitée aux décompositions similaires aux quadtrees.

Cette décomposition spatiale récursive forme un arbre de décomposition où chaque feuille représente une particule. Lors du calcul des forces approchées, l'arbre est parcouru de manière à trouver les nœuds de l'arbre (intérieurs ou feuilles) avec lesquels les particules interagissent.

Les nœuds intérieurs de l'arbre sont associés à des pseudo-particules, *i.e.* des particules fictives représentant plusieurs particules réelles. La masse d'une pseudo-particule est le nombre de particules contenues dans le sous-arbre dont elle est racine. La position d'une pseudo-particule correspond au barycentre de la position des particules contenues dans ce même sous-arbre.

Calcul approché des forces de répulsion A partir de l'arbre de décomposition, la méthode de Barnes et Hut [15] calcule une approximation des forces en séparant les interactions avec des nœuds proches et celles avec des nœuds éloignés. Pour une particule u donnée, l'arbre est exploré en partant de la ra-



(a) Selection des pseudo-particules. (b) Calcul des forces approchées.

FIGURE 3.6 – Calcul des forces de répulsion approchées à l'aide de la méthode de Barnes et Hut [15]. Calcul des forces de répulsion approchées entre la particule jaune et les pseudo-particules sélectionnées (délimitées par la zone noire).

cine. La distance entre u et la pseudo-particule v permet de déterminer si v est utilisée pour le calcul approché. Si v n'est pas retenue, les pseudo-particules w_i, enfants de v, sont considérées. Cette décomposition est illustrée en Figure 3.6.

Trois paramètres entrent en jeu pour déterminer si la pseudo-particule v interagit avec u. Le paramètre s_v correspond au diamètre de la cellule contenant la pseudo-particule v. Le paramètre d_v correspond à la distance entre la particule u et la pseudo-particule v. Enfin, la constante θ est le seuil utilisé pour déterminer s'il y a interaction. La pseudo-particule v interagit avec u si $\frac{s_v}{d_v} \leq \theta$. Sinon les pseudo-particules enfants sont considérées comme candidates.

La suite de paramètres s décroît au fur et à mesure que de nouvelles pseudo-particules sont considérées, *i.e.* lorsqu'on descend dans l'arbre de décomposition. L'approximation obtenue par cette méthode :

- calcule exactement les forces pour les nœuds proches (d petit donc s petit)
- donne une approximation pour les nœuds éloignés (d grand donc s grand)

Le choix de θ influence le temps de calcul de cette méthode. Pour θ nul, on retrouve le calcul des forces exactes avec sa complexité de $O(|V|^2)$. Pour $\theta \geq \frac{s_0}{d_{max}}$, le calcul est effectué uniquement avec la pseudo-particule à la racine de l'arbre de décomposition. Pour les valeurs de θ intermédiaires, la complexité du calcul tombe à $O(|V| \cdot \log |V|)$. En effet, la création de l'arbre nécessite l'insertion de $|V|$ éléments dans un arbre binaire (opération en $O(\log |V|)$) pour une complexité totale de $O(V \cdot \log |V|)$. Pour le calcul des forces pour une particule, le temps de calcul est en $O(\log |V|)$.

Erreur et complexité Barnes et Hut [16] proposent en 1989 une analyse de l'erreur commise en utilisant leur algorithme d'approximation des forces par décomposition récursive. Deux évaluations sont menées pour évaluer l'erreur de calcul. La première erreur pour de nombreuses particules éloignées contenues dans un cube et une particule unique et la seconde le total des erreurs commises sur l'ensemble des forces approximées par la méthode de Barnes et Hut.

L'erreur commise entre une particule unique et plusieurs particules éloignées agrégées en une unique meta-particule est exprimée mathématiquement à l'aide du développement multipôle (*cf.* Section suivante pour plus d'informations). Pour le calcul de l'énergie potentielle de gravitation Φ, l'erreur obtenue est de $O(\frac{M \cdot a^4}{r^5})$, avec M la masse de la meta-particule, a la taille de la boîte contenant la meta-particule et r la distance entre la particule et la meta-particule. Ce calcul est valable pour une répartition uniforme des particules dans le plan. Barnes et Hut donnent aussi l'erreur lorsque la densité de particules croît linéairement, en $O(\frac{M \cdot a^2}{r^2})$. Ces développements sont valables pour un nombre de particules $N \to \infty$. Cette erreur est validée expérimentalement en considérant un nombre de particules N croissant pour différentes valeurs des paramètres de l'erreur.

Utilisation en dessin de graphe L'algorithme FADE de Quigley et Eades [115] utilise cette méthode pour calculer les forces de répulsion de son analogie physique. La méthode de Barnes et Hut permet de réduire la complexité du calcul des forces de répulsion de $O(|V|^2)$ à $O(|V| \cdot \log(|V|))$.

Méthode multipôle rapide

Nous présentons dans cette section la méthode multipôle rapide. Cette famille de méthodes repose sur de nombreux résultats théoriques et variantes, *cf.* par exemple les travaux de Greengard [60, 59]. Greengard [19] propose aussi un cours qui constitue une bonne introduction à la structure générale de ces méthodes et sur lequel nous nous appuyons pour quelques-uns des résultats théoriques.

Décomposition multipôle Dans la méthode multipôle rapide, les interactions entre particules éloignées sont calculées grâces à un développement en série entière de l'intensité des forces de répulsion.

Soient u la particule pour laquelle on cherche à calculer les interactions dont la position est fixée et v une particule en interaction avec u. Soit c une particule fictive intermédiaire proche de u. Il est possible d'exprimer le développement en série en fonction du rapport des distances $\frac{d_{uc}}{d_{vc}}$. Ce rapport est petit lorsque u est proche de c et v est éloigné de c.

Si la fonction d'intensité F est développable en série entière sur le disque $]-R, R[$, son développement en série entière s'écrit de la manière suivante à

l'aide des polynômes de Taylor :

$$F(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(0)}{n!} \cdot x^n, \forall x \in]-R, R[$$

Il est possible d'approximer l'évaluation de cette somme en tronquant la série après P termes. Pour la fonctions F, l'erreur commise est $o(x^P)$, x proche de 0. La précision du calcul des forces est donc modulable par le choix du nombre P de termes de la somme à évaluer.

Exemple 1. Pour le calcul des forces de répulsion de Fruchterman et Reingold [48] en une dimension, soient la particule u et la particule éloignée v en interaction avec u. Soit c la particule fictive proche de u. Le développement de la force de répulsion appliquée à v prend la forme suivante :

$$\mathbf{F}_r(v) = \frac{k^2}{p_v - p_u} = \frac{k^2}{p_v - p_c} \cdot \frac{1}{1 - \frac{p_u - p_c}{p_v - p_c}} \approx \frac{k^2}{p_v - p_c} \sum_{i=0}^{P} \left(\frac{p_u - p_c}{p_v - p_c} \right)^i$$

On obtient ainsi une assez bonne approximation de la force de répulsion $\mathbf{F}_r$ appliquée aux particules v éloignées de u. Pour obtenir les P coefficients a_n du développement multipôle, on cherche à réécrire notre développement en série entière sous la forme canonique :

$$\mathbf{F}_r(v) = a_0 \log(|p_v - p_c|) + \sum_{i=1}^{\infty} \frac{a_i}{(p_v - p_c)^i}$$

Cette forme canonique est utilisée dans les Lemmes 4.6 et 4.7 de Beatson et Greengard [19]. En réécrivant le développement obtenu précédemment, on obtient la forme suivante :

$$\mathbf{F}_r(v) \approx 0 \cdot \log(|p_v - p_c|) + k^2 \sum_{i=1}^{P} \frac{(p_u - p_c)^{i-1}}{(p_v - p_c)^i}$$

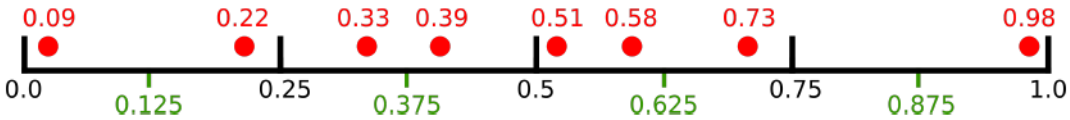
Pour $i \geq 1$, les coefficients a_i s'expriment comme $a_i = k^2 \cdot (p_u - p_c)^{i-1}$. Il est à noter que le coefficient a_0 est nul dans ce développement.

Ensembles bien séparés La décomposition en série est valable uniquement sur un certain intervalle de convergence. La plupart des développements en série entière classiques convergent absolument sur le disque $]-1, 1[$. Nous évaluons ces séries entières pour le rapport $\frac{d_{uc}}{d_{vc}}$ et il est donc nécessaire que $d_{uc} < k \cdot d_{vc}$ avec k une constante dépendante du développement en série. Les ensembles de particules qui respectent cette condition sont dit **bien séparés**. Le calcul des interactions est approximé par un développement en série entière pour les ensembles de particules bien séparés. Pour les autres interactions, l'approche multipôle repose sur le calcul exact des interactions.

La méthode multipôle utilise une grille de décomposition dont le nombre de niveau est fixé. Chaque cellule de cette décomposition contient un ensemble de particules. La particule intermédiaire c est définie comme le centre de la cellule contenant un ensemble de particules.

Deux cellules sont bien séparées si elles ne partagent pas de bord ou de coin. Les particules des cellules biens séparées respectent l'inégalité $d_{uc} < k \cdot d_{vc}$: la distance d_{uc} est petite relativement à la distance d_{vc}. En ne considérant pas les cellules partageant un bord ou un coin comme bien séparées, il est possible de s'affranchir des effets de bords que le découpage en grille introduit. La Figure 3.7a illustre les ensembles bien séparés utilisés pour le calculs des interactions avec une particule donnée.

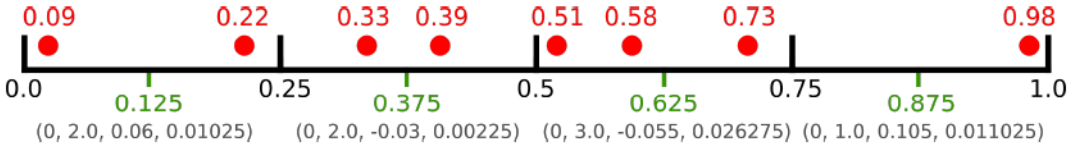
Exemple 2. Pour la décomposition en deux niveaux suivante, nous cherchons à calculer la décomposition multipôle pour chaque cellule. Nous fixons la constante $k = 1$ et le paramètre de précision $P = 3$.



Pour la première cellule, représentant les particules dans l'intervalle $[0, 0.25[$, nous calculons les coefficients a_i, $1 \leq i \leq 3$, pour les deux particules par rapport au centre de cette cellule. Le paramètre p_u est égal à 0.125.

$$a_1 = (0.09 - 0.125)^0 + (0.22 - 0.125)^0 = 2.0$$
$$a_2 = (0.09 - 0.125)^1 + (0.22 - 0.125)^1 = 0.06$$
$$a_3 = (0.09 - 0.125)^2 + (0.22 - 0.125)^2 = 0.01025$$

En faisant de même pour les trois autres cellules, nous obtenons les coefficients suivants pour le développement en série positionné au centre de chaque cellule.



Méthode multipôle sur un niveau Pour une particule donnée, le calcul est décomposé en deux composantes : les interactions directes et les interactions avec les particules éloignées, utilisant les décompositions multipôles. Pour les interactions directe, les interactions sont calculées entre les particules et celles dans des cellules voisines, *i.e.* qui ne sont pas bien séparées.

Pour les interactions avec la décomposition multipôles, l'idée est de combiner les développements multipôles des cellules bien séparées. Lorsque les

développements multipôles sont exprimés à la même position, les coefficients peuvent être sommés pour obtenir la contribution globale de ces développements multipôles. Grâce à la combinaison des développements, l'évaluation des interactions éloignées s'effectue en un calcul au lieu d'effectuer un calcul pour chaque cellule bien séparée. L'évaluation des interactions éloignées est réalisée pour chaque particule.

Pour une cellule donnée, nous cherchons donc à déplacer les développements multipôles bien séparés vers le centre de la cellule pour les combiner. Ce processus est réalisé lors de la conversion du développement multipôle en développement local sous la forme d'une série entière $\sum_i b_i p^i$. Le Lemme 4.7 de Beatson et Greengard [19] permet de transformer les développements multipôles en développements locaux centrés en zéro. Pour le développement multipôle exprimé sous la forme :

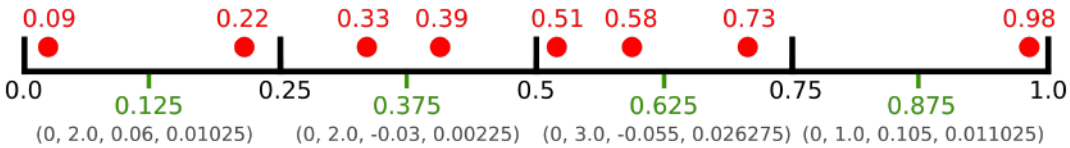
$$\mathbf{F}_r(p_v) = a_0 \log(|p_v - p_c|) + \sum_{i=1}^{\infty} \frac{a_i}{(p_v - p_c)^i}$$

il est possible de réécrire ce développement comme une série entière pour p_v proche de 0 dont les coefficients s'expriment comme :

$$\begin{aligned} b_0 =& a_0 \cdot \log(p_c) + \sum_{i=1}^{\infty} (-1)^i \cdot \frac{a_i}{p_c^i} \\ b_l =& -\frac{a_0}{l \cdot p_c^l} + \frac{1}{p_c^l} \sum_{i=1}^{\infty} (-1)^i \binom{l+i-1}{i-1} \cdot \frac{a_i}{p_c^i} \end{aligned}$$

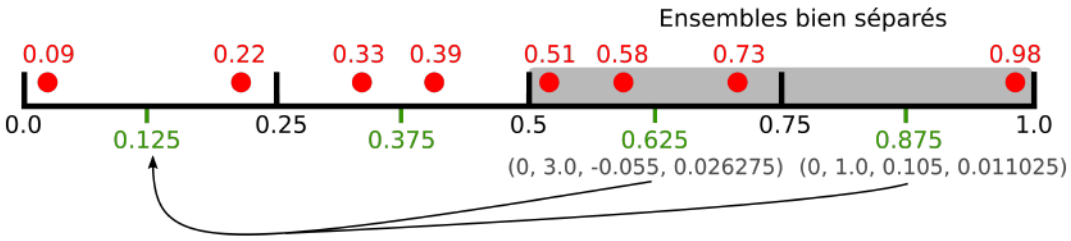
En pratique, les coefficients b_l sont obtenus en tronquant la somme à P termes. Ce lemme est exprimé pour un développement local en 0. Cependant, il est possible de l'exprimer au centre d'une cellule directement en effectuant un changement de variable.

Exemple 3. Dans cet exemple, en utilisant les développements multipôles obtenus dans l'Exemple 2, nous calculons ici les forces de répulsion approchées pour les particules de la première cellule.



Les cellules bien séparées de la première cellule sont les cellules 3 et 4. Pour ces deux cellules, nous calculons le développement local en série entière exprimé au centre de la cellule 1 :

Le calcul des coefficients pour la cellule 3 et la cellule 4 prennent la forme suivante respectivement :



$$b_0 = -6.4302 \quad b_1 = -14.1412 \quad b_2 = -31.6848 \quad b_3 = -71.8560$$
$$b_0 = -1.1728 \quad b_1 = -1.38453 \quad b_2 = -1.65357 \quad b_3 = -2.01007$$

Nous évaluons les forces de répulsion éloignées appliquées sur la particule à la position 0.09 :

$$\begin{aligned} F_{r,far}(0.09) &= \sum_{i=0}^{3} b_i \cdot (0.09 - 0.125)^i \\ &\approx -7.097272 \end{aligned}$$

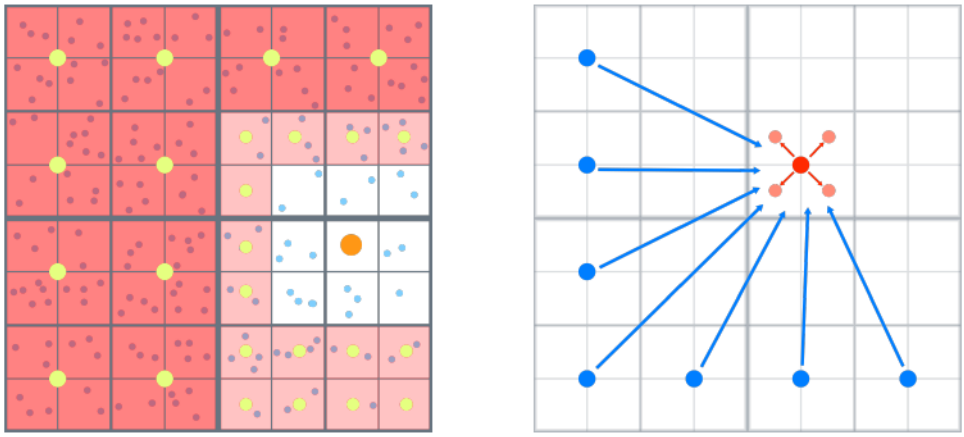
Puis les forces de répulsion proches sont calculées exactement :

$$\begin{aligned} F_{r,close}(0.09) &= \frac{1}{0.09 - 0.22} + \frac{1}{0.09 - 0.33} + \frac{1}{0.09 - 0.39} \\ &\approx -15.19231 \end{aligned}$$

Les forces de répulsion totales appliquées sur la particule à la position 0.09 sont alors $F_r(0.09) \approx -22.28958$.

Méthode multipôle à plusieurs niveaux La méthode multipôle à un niveau peut être adaptée pour fonctionner avec une décomposition à plusieurs niveaux. Pour des décompositions à trois niveaux ou plus, l'approche à un niveau est appliquée à chaque niveau et adaptée de manière à accélérer les calculs. L'idée est ici de trouver les cellules bien séparées les plus larges possibles pour les utiliser dans le développement local au lieu d'utiliser uniquement les cellules bien séparées de plus bas niveau. Cette approche est similaire à l'arbre de décomposition utilisé dans Barnes et Hut : nous cherchons les cellules de plus haut niveau possible interagissant avec une cellule donnée. La Figure 3.7a illustre les cellules bien séparées (en rouge) utilisées pour le développement local de la cellule jaune.

En pratique, l'approche multipôle est d'abord calculée pour chaque cellule de chaque niveau en commençant par la grille la plus fine. Puis les développement locaux sont calculés à chaque niveau entre cellules bien séparées. Comme dans la section précédente, deux résultats permettent de modifier la position des développements multipôles et locaux. Ces deux résultats permettent de



(a) Ensembles bien séparés pour le calcul des forces approchées.

(b) Calcul des décompositions multipôles par translation.

FIGURE 3.7 – Calcul des forces de répulsion approchées à l'aide de la méthode multipôle rapide. *(a)* Calcul des forces approchées pour le nœud jaune et visualisation des ensembles bien séparés. *(b)* Calcul des décompositions multipôles par translation. En rouge, translation pour la combinaison d'une décomposition interne. En bleu, translation de décompositions externes pour le calcul d'interactions.

capitaliser les calculs déjà menés pour obtenir les développements multipôles au niveau le plus bas de la décomposition (respectivement les développements locaux obtenus aux niveaux plus hauts).

Pour les niveaux intermédiaires, la décomposition multipôle d'une cellule est déduite des quatre décompositions multipôles des cellules du niveau inférieur qu'elle représente. Le Lemme 4.6 de Beatson et Greengard [19] permet de réécrire une décomposition multipôle à une position différente, et notamment au centre d'une cellule d'un niveau supérieur. Ce lemme permet de sommer les coefficients des quatre décompositions multipôles enfants pour former une décomposition multipôle parent. Grâce à ce lemme, il est possible d'obtenir le développement multipôle de chaque cellule de la décomposition en ne parcourant les particules qu'une seule fois. L'opération de déplacement coûte P^2 opérations et il faut donc $4 \cdot P^2$ opérations pour former un développement multipôle intermédiaire à partir des développements multipôles des cellules enfants.

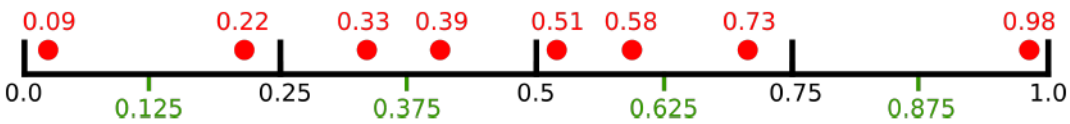
Les développements locaux sont obtenus à chaque niveau entre les cellules bien séparées. De cette manière, ce sont toujours les cellules de plus haut niveau qui sont utilisées pour effectuer le calcul (*cf.* arbre de décomposition de Barnes et Hut). Cependant, les développements locaux obtenus lors des niveaux intermédiaires sont placés au centre des cellules intermédiaires. Entre chaque niveau, il faut donc déplacer ces développements locaux au centre des

cellules enfants pour obtenir leur développement local. Le Lemme 4.8 de Beatson et Greengard [19] donne l'expression des coefficients de la série entière après une translation. La Figure 3.7b *(flèches bleues)* illustre ce processus. Les développements locaux sont exprimés au centre d'une cellule (*flèches bleues*) puis translatés pour les cellules enfants *(flèches rouges)*. L'opération de translation coûte elle aussi P^2 opérations et doit être réalisée pour les quatre cellules enfants pour un total de $4 \cdot P$ opérations.

Évaluation de l'erreur Contrairement à la méthode de Barnes et Hut [15], l'erreur dans la méthode multipôle rapide est maîtrisée. Les approximations interviennent à trois endroits dans cette méthode : lors de la troncature du développement multipôle à P termes, lors de la translation d'un développement multipôle et lors de l'expression du développement local en un point. Dans les trois cas, il s'agit de considérer les P premiers termes d'une somme infinie. Les bornes pour les séries tronquées après P termes sont données par Beatson et Greengard [19]. Dans ce papier, le lecteur pourra notamment consulter les Lemme 4.6 et 4.7 ainsi que la Section 2.1 (équation 2.2).

Le paramètre P permet donc de régler la précision : choisir un nombre de termes P à évaluer plus important réduit l'erreur mais augmente le nombre de calculs intermédiaires (*e.g.* les $4 \cdot P^2$ opérations nécessaires pour les translations, *cf.* section précédente). Le choix de P ne modifie pas la complexité globale du problème qui reste en $O(|V|)$: au regard du nombre de particules généralement traitées à l'aide de cette méthode (des millions, voire des milliards de particules), P est considéré comme une constante.

Exemple 4. Dans l'Exemple 3, nous présentons le calcul approché des forces de répulsion pour la particule à la position 0.09.



Nous proposons ici d'évaluer l'erreur commise entre cette approximation et le calcul exact pour différentes valeurs de P. La valeur exacte du calcul est $F_r(0.09) = -22.30017$. Les valeurs approchées données dans le tableau sont arrondies à deux décimales.

Valeurs de P	0	1	2	3	4
Calcul approché	-15.19	-22.04	-22.10	-22.29	-22.29
Erreur	7.11	$2.57 \cdot 10^{-1}$	$2.02 \cdot 10^{-1}$	$1.06 \cdot 10^{-2}$	$7.84 \cdot 10^{-3}$

Utilisation en dessin de graphe La méthode multipôle permet de réduire la complexité du calcul de l'ensemble des interactions à $O(|V|)$. C'est la méthode de calcul des forces de répulsion privilégiée par exemple par FM^3 [63] et Godiyal [57].

Dans le cas du dessin, la méthode multipôle recommence à zéro à chaque itération. Il faut alors générer à nouveau les développements multipôles en tenant compte de la nouvelle position des nœuds. Cependant, ce calcul nécessite seulement $O(|V|)$ opérations pour calculer l'ensemble des interactions entre toutes les paires.

3.4 Méthodes de résolution

Pour atteindre l'équilibre de ces analogies physiques, deux familles de méthodes de résolutions sont utilisées. Les méthodes de minimisation de l'énergie cherchent à trouver le minimum des fonctions d'énergie par différentes techniques d'optimisation, qu'elles soient exactes ou issues d'heuristiques. La deuxième famille simule le comportement du système itérativement sur des pas de temps court en appliquant sur chaque point une force issue du modèle physique sous-jacent. Dans les deux cas, le but est d'atteindre un état d'équilibre qui correspond au minimum d'énergie du système physique. Nous présentons dans cette section les deux familles en détail et les variations au sein de ces familles. Nous présentons aussi les heuristiques d'optimisation (recuit simulé et algorithme génétique) qui ont été appliquées au dessin de graphe.

3.4.1 Simulation physique

Approche classique

L'application itérative de forces est une méthode de minimisation de l'énergie introduite par Eades [42] en 1984. Pour un nœud donné, l'ensemble des forces exercées (d'attraction et de répulsion) permettent de calculer le déplacement du nœud dans le dessin. Tous les nœuds sont déplacés simultanément. Cette méthode de résolution est présentée en Algorithme 1. Les nœuds sont au départ placés aléatoirement dans le plan du dessin. En pratique, les applications successives de forces permettent de converger vers un état d'équilibre mais cette convergence n'est pas garantie. Un cas de figure courant est l'oscillation de nœuds qui dépassent leur position optimale et oscillent autour sans y converger.

Fruchterman et Reingold [48] reprennent les travaux d'Eades et y introduisent un refroidissement de la position des particules dans le dessin inspiré par les méthodes de recuit simulé [86]. Au fur et à mesure du déroulement de l'algorithme, seule une fraction de la force totale est réellement appliquée à

Algorithm 1 Dessin par application itérative de forces

```
Initialisation du dessin
Initialisation de la température T
for |V| itérations do
    Calcul des forces
    Calcul du déplacement pour la température T
    T ← γ · T
end for
```

chaque particule à la manière des frottements que peut subir un système physique. La température T diminue d'un facteur de refroidissement constant γ à chaque itération. C'est ce refroidissement qui assure la convergence du dessin.

Températures locales

Frick et al. [46] poussent la gestion des températures plus loin afin d'améliorer le dessin et la rapidité de l'algorithme. La structure de GEM (Graph EMbedder) est basée sur celle de l'algorithme de Fruchterman et Reingold [48]. Contrairement à l'approche de Fruchterman et Reingold, GEM calcule le déplacement de chaque nœud individuellement au lieu de déplacer l'ensemble des nœuds simultanément. L'ordre dans lequel les nœuds sont visités est aléatoire mais l'algorithme est découpé en rondes lors desquelles chaque nœud est déplacé.

GEM utilise des températures locales (une par nœud) pour mieux évaluer les déplacements autorisés à chaque nœud. Cette température est basée sur les optimisations qui peuvent encore être réalisées sur la position d'un nœud donné. Elles prennent aussi en compte deux artefacts qui apparaissent souvent dans les méthodes de dessin : les oscillations et les rotations.

Rotations Le minimum atteint par les algorithmes de dessin de graphe est dans la plupart des cas invariant aux rotations (à quelques exceptions près, *cf.* Sugiyama et Misue [125]) : faire tourner l'ensemble des nœuds autour d'un centre choisi au hasard ne modifie pas l'énergie du dessin puisque celui-ci est basé sur les distances entre les nœuds. Lorsque GEM détecte une rotation, il diminue la température du nœud en conséquence afin d'empêcher un déplacement qui n'est pas utile dans l'optimisation du dessin.

Oscillations Les oscillations apparaissent lorsque plusieurs nœuds oscillent autour de leur position optimale, la dépassant à chaque fois. Ces comportements entraînent des calculs supplémentaires qui ne permettent pas de faire émerger une structure présente dans le graphe. Pour les éviter, la température locale des nœuds oscillants est diminuée de manière importante afin de limiter leur déplacements.

Modèle par Barycentre de Tutte

En 1963, Tutte [128] propose un algorithme itératif permettant de dessiner les graphes triconnexes planaires sans croisements. Le sous-ensemble V_0 est constitué des nœuds qui représentent le bord du dessin. Ces nœuds sont arrangés selon un polygone convexe. Lors de chaque itération, les nœuds restants, dans l'ensemble $V_1 = V \backslash V_0$ sont placés au barycentre de la position de leur voisin. En itérant jusqu'à convergence, Tutte prouve que le dessin obtenu n'a pas de croisement et que les faces formées sont convexes.

Cet algorithme est ré-exprimé par di Battista et al. [18, 87] sous la forme d'un algorithme par simulation physique. Tous les nœuds libres, *i.e.* les nœuds dans V_1, sont placés au barycentre de leur voisins simultanément. Ce processus est itéré jusqu'à un état d'équilibre.

Cet état d'équilibre est atteint lorsque la somme des forces appliquée à chaque nœud est nulle, ce qui s'exprime sous la forme :

$$\forall v \in V_1 \quad F(v) = \sum_{(u,v) \in E} (p_u - p_v) = 0$$

C'est l'utilisation du polygone extérieur qui assure que les nœuds ne sont pas tous à la même position.

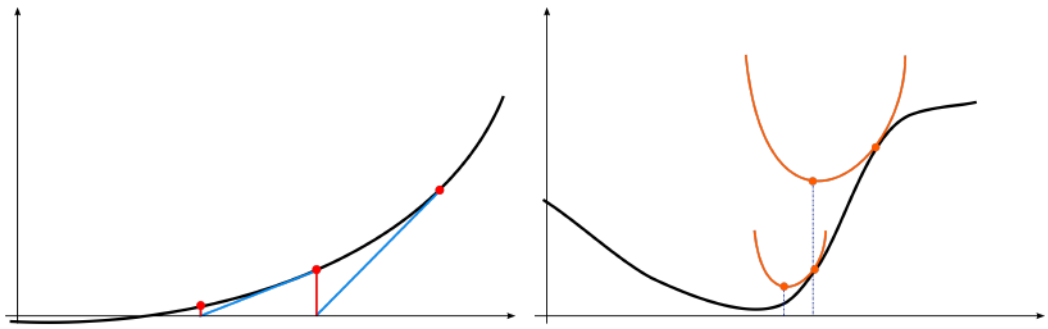
3.4.2 Outils mathématiques d'optimisation

L'autre approche pour obtenir le dessin du graphe est de minimiser la fonction d'énergie du modèle physique pour trouver l'état d'équilibre. Dans ce problème d'optimisation, il y a $d \cdot |V|$ paramètres à optimiser, d étant la dimension de l'espace du dessin. L'optimisation multivariée classique propose de nombreux outils qui peuvent être appliqués au dessin de graphe.

Méthode de Newton-Raphson

Les algorithmes de Quinn [116] et de Kamada et Kawai [84] appliquent la méthode de Newton-Raphson [124] pour trouver les coordonnées du point annulant les dérivées partielles de la fonction d'énergie. L'énergie de chaque nœud est calculée et celui ayant la plus haute énergie est sélectionné. Sa position optimale, minimisant l'énergie globale, est obtenue par la méthode de Newton-Raphson. Ces deux étapes sont itérées jusqu'à converger vers un minimum local d'énergie.

Pour l'algorithme de Kamada et Kawai, $d \cdot |V|$ dérivées partielles sont calculées, *i.e.* une par variable à optimiser. Les minima locaux sont les solutions du système $\frac{\partial E}{\partial x_i} = \frac{\partial E}{\partial y_i} = 0$ pour tout nœud i de V. La résolution directe n'est pas possible car ces équations ne sont pas indépendantes. Pour les rendre indépendantes, Kamada et Kawai fixent la position de tous les nœuds à l'exception



(a) Itérations de Newton Raphson pour résoudre $f(x) = 0$

(b) Majoration itérative de la fonction de stress

FIGURE 3.8 – Schéma des algorithmes d'optimisation détaillés dans cette section. *(a)* Méthode de Newton-Raphson en une dimension pour résoudre l'équation $f(x) = 0$. Obtention du minimum d'énergie après dérivation, *cf.* Kamada et Kawai [84]. *(b)* Méthode de majoration de la fonction de stress pour trouver le minimum d'énergie de manière itérative, *cf.* Gansner [51].

du nœud de plus haute énergie m. Le système se réécrit alors sous la forme des deux équations $\frac{\partial E}{\partial x_m} = \frac{\partial E}{\partial y_m} = 0$.

C'est ces deux dernières équations que l'on résout grâce à la méthode itérative de Newton-Raphson [124]. La position du nœud m à l'itération t est notée $(x_m^{(t)}, y_m^{(t)})$. L'équation des plans tangents aux fonctions $\frac{\partial E}{\partial x_m}$ et $\frac{\partial E}{\partial y_m}$ en $(x_m^{(t)}, y_m^{(t)})$ sont calculées à l'aide des dérivées partielles :

$$\begin{cases} z = \frac{\partial^2 E}{\partial x_m^2}(x_m^{(t)}, y_m^{(t)}) \cdot (x - x_m^{(t)}) + \frac{\partial^2 E}{\partial x_m \partial y_m} \cdot (y - y_m^{(t)}) + \frac{\partial E}{\partial x_m}(x_m^{(t)}, y_m^{(t)}) \\ z = \frac{\partial^2 E}{\partial x_m \partial y_m}(x_m^{(t)}, y_m^{(t)}) \cdot (x - x_m^{(t)}) + \frac{\partial^2 E}{\partial y_m^2} \cdot (y - y_m^{(t)}) + \frac{\partial E}{\partial y_m}(x_m^{(t)}, y_m^{(t)}) \end{cases} \tag{3.1}$$

Pour obtenir la position du nœud m pour l'itération suivante, c'est à dire $(x_m^{(t+1)}, y_m^{(t+1)})$, on résout le système d'équations linéaires 3.1 pour $z = 0$. Après quelques itérations, la position obtenue est proche du minimum global et donne la position finale du nœud. Un schéma de la méthode de Newton-Raphson en une dimension est proposé en Figure 3.8a.

Descente de gradient

La descente de gradient, aussi connu comme méthode de la plus forte pente, est une méthode d'optimisation mathématique utilisée pour minimiser des fonctions convexes. Cette méthode permet de trouver un minimum local indépendamment de la fonction à optimiser. L'énergie est minimisée de façon itérative dans la direction de la plus forte pente, obtenue grâce à l'hyperplan tangent. La configuration courante est alors déplacée dans cette direction de manière à minimiser l'énergie.

Hadany et Harel [65] l'utilisent pour optimiser des fonctions d'énergies locales dans le modèle de Kamada et Kawai [84]. Un nœud est choisi et sa position est optimisée selon la méthode de gradient : ce nœud est déplacé d'un vecteur unitaire selon la direction de plus forte pente. Pour obtenir l'hyperplan tangent, les dérivées partielles sont calculées selon les coordonnées de ce nœud.

Cette méthode est une variante de la simulation physique présentée dans la section précédente pour laquelle un seul nœud est déplacé à chaque itération. En calculant les dérivées partielles premières de la fonction d'énergie selon les coordonnées du nœud u, on retrouve les forces que l'on applique ensuite au nœud u pour le déplacer. Cette méthode est rarement présentée comme telle dans la littérature, où on lui préfère la formulation par simulation physique.

3.4.3 Outils d'optimisation pour un modèle physique particulier

Modèle de Kamada et Kawai

Le positionnement multidimensionnel [24] (aussi appelé MultiDimensional Scaling ou MDS) est une technique de visualisation exploratoire partageant certaines similarités avec le dessin de graphe. Le principe du positionnement multidimensionnel est de placer des nœuds dans un espace à n dimensions (généralement $n = 2$ ou $n = 3$) de manière à ce que la distance entre chaque nœud dans le dessin soit le plus proche possible de la matrice des distances entre ces nœuds. Les données d'entrées des MDS sont des matrices de distances symétriques. Cette technique est utilisée dans divers champs de recherche, comme par exemple en psychologie, pour révéler des corrélations entre les données et trouver des axes d'ordonnancement à des caractéristiques qui n'en ont pas au premier abord.

Ce champ de recherche propose des techniques pour le dessin de graphe dans un cas particulier : la fonction d'énergie de Kamada et Kawai [84]. Dans Kamada et Kawai, le distance graphe entre chaque paire de nœuds est utilisée pour créer un dessin se rapprochant le plus possible de l'isométrie (*i.e.* la distance dans le dessin est égale à la distance dans le graphe). La minimisation de cette fonction d'énergie particulière, appelée la fonction de stress en MDS, permet d'obtenir cette pseudo-isométrie.

Majoration du stress Gansner [51] propose en 2005 une méthode d'optimisation basée sur la majoration du stress, une technique utilisée depuis les années 1980 en MDS (*cf.* De Leuuw [35] par exemple). Cet algorithme repose sur l'optimisation d'une fonction majorant la fonction de stress.

Soit $X \in \mathbb{R}^{n \times d}$ la matrice des positions des n nœuds dans l'espace de dimension d. La fonction majorante F_X est générée pour la configuration courante du dessin X. F_X est strictement supérieure à la fonction d'énergie E pour

toute configuration $Y \in \mathbb{R}^{n\times d}$ non colinéaire à X. En utilisant l'argument du minimum [1], on définit $\tilde{X}$ comme $\tilde{X} = \underset{Y}{\operatorname{argmin}}\, F_X(Y)$. F_X majore la fonction d'énergie E en tout point et on a donc les inégalités suivantes :

$$E(\tilde{X}) \leq F_X(\tilde{X}) \leq F_X(X) = E(X) \tag{3.2}$$

Le cas d'égalité de l'Equation 3.2 intervient lorsque $X = \tilde{X} = \underset{Y}{\operatorname{argmin}}\, E(Y)$, *i.e.* lorsqu'on a trouvé un minimum d'énergie. De ce résultat, on déduit un algorithme permettant de minimiser la fonction d'énergie, *cf.* Algorithme 2. Cet algorithme est illustré en une dimension dans la Figure 3.8b.

Algorithm 2 Optimisation par majoration de la fonction de stress

```
X ∈ ℝ^{n×d}
while E(X) n'est pas minimal do
    Trouver F_X majorant E
    X̃ = argmin_Y F_X(Y)
    X ← X̃
end while
```

Fonction majorante Pour obtenir la fonction majorante F_X, on développe l'expression de la fonction de stress :

$$\begin{aligned} E(X) &= \sum_{i<j} w_{ij} \left(\|X_i - X_j\| - d_{ij}\right)^2 \\ &= \sum_{i<j} w_{ij} d_{ij} + \sum_{i<j} w_{ij} \|X_i - X_j\|^2 - 2 \sum_{i<j} w_{ij} d_{ij} \|X_i - X_j\| \end{aligned}$$

La matrice laplacienne L d'un graphe est définie comme la différence entre la matrice de degré D d'un graphe et sa matrice d'adjacence A. Grâce à cette matrice, il est possible de réécrire la distance entre nœuds dans le dessin sous la forme matricielle $Tr(X^T L X)$ pour $X \in \mathbb{R}^{n\times d}$ la matrice de coordonnées des nœuds. Le deuxième terme de la forme développée du stress se réécrit ainsi $\sum_{i<j} w_{ij} \|X_i - X_j\|^2 = Tr(X^T L^w X)$, avec L^w la matrice laplacienne des poids w_{ij}.

L'expression sous forme matricielle est intéressante car elle permet d'exprimer facilement les systèmes d'équations obtenus par dérivations partielles, *cf.* par exemple le livre de recette des matrices de Petersen [110].

Pour majorer la fonction de stress, on minore le terme $\sum_{i<j} w_{ij} d_{ij} \|X_i - X_j\|$. A l'aide de l'inégalité de Cauchy-Schwarz, on a pour tout Z de $\mathbb{R}^{n\times d}$:

1. L'argument du minimum, généralement noté argmin, est l'ensemble des valeurs pour lesquelles une fonction atteint son minimum.

$$0 \leq \frac{w_{ij}}{\|Z_i - Z_j\|}(X_i - X_j)^T(Z_i - Z_j) \leq w_{ij}\|X_i - X_j\|$$

Le terme minorant peut être réécrit sous la forme $Tr(X^T L^Z Z)$. On a ainsi trouvé une fonction majorant la fonction de stress pour tout X, Z de $\mathbb{R}^{n\times d}$:

$$\begin{aligned} E(X) &= \sum_{i<j} w_{ij}d_{ij} + Tr(X^T L^w X) - 2\sum_{i<j} w_{ij}d_{ij}\|X_i - X_j\| \\ &\leq \sum_{i<j} w_{ij}d_{ij} + Tr(X^T L^w X) - 2Tr(X^T L^Z Z) \qquad = F_Z(X) \end{aligned}$$

Minimisation de la fonction majorante On connaît désormais l'expression de la fonction majorante F_Z. Après n itérations de l'Algorithme 2, nous avons obtenu le dessin Z. Nous cherchons l'argument du minimum $\tilde{X}$ de la fonction F_Z, *i.e.* $\tilde{X}$ tel que $E(\tilde{X}) \leq E(Z)$. Pour cela, nous dérivons la fonction F_Z selon la variable X (attention, ce sont des dérivées partielles).

F_Z s'exprime sous une forme quadratique de la variable X. Petersen [110] donne des formules pour les dérivées partielles des traces, *cf.* Equation 103 et 108. On obtient :

$$\frac{\partial F_Z}{\partial X} = 2L^w X - 2L^Z Z$$

On cherche à résoudre $\frac{\partial F_Z}{\partial X} = 0$, ce qui nous donne le système linéaire suivant :

$$L^w X = L^Z Z$$

La solution $\tilde{X}$ de ce système donne le dessin minimisant la fonction majorante, et donc a fortiori faisant diminuer l'énergie globale.

Stress en une dimension Koren et Harel [90] isolent une dimension du dessin pour effectuer l'optimisation de la position de l'ensemble des nœuds dans cette dimension. Cette approche repose sur une fonction majorante pour effectuer l'optimisation. L'Algorithme 2 donne les étapes de minimisation de l'énergie globale.

Dans le cas d'un dessin en une dimension, la fonction de stress est majorée par la fonction $E^{\tilde{x}} = \sum_{i<j} k_{ij}\left(\delta_{ij}^{\tilde{x}}(x) - d_{ij}\right)^2$ où :

$$\delta_{ij}^{\tilde{x}}(x) = \begin{cases} x_i - x_j & \text{si } \tilde{x}_i \geq \tilde{x}_j \\ x_j - x_i & \text{sinon} \end{cases} \quad \text{pour} 1 \leq i < j \leq |V|$$

La fonction $E^{\tilde{x}}$ peut être réécrite sous la forme $E^{\tilde{x}}(x) = x^T Lx - 2x^T b^{\tilde{x}} + C$ que l'on dérive pour obtenir le système d'équations $Lx = b^{\tilde{x}}$. Pour optimiser la position des nœuds dans une dimension donnée, on résout itérativement

ce système avec comme paramètre $\tilde{x}$ la position des nœuds lors de l'itération courante, *cf.* section précédente.

Pour effectuer le dessin en deux dimensions, l'algorithme calcule la position optimale des nœuds selon l'axe x puis la position optimale des nœuds selon l'axe y. Lors du dessin selon l'axe y, la position des nœuds selon l'axe x est prise en compte. Koren et Harel utilisent une distance résiduelle au lieu de la distance graphe pour refléter le fait qu'une partie de la distance entre les nœuds est déjà représentée selon l'axe x.

Modèle de Hall

Hall [66] utilise un modèle d'énergie plus simple que ceux proposés par les algorithmes de dessin de graphe par modèle de force et l'applique aux MDS et au dessin de graphe. Son modèle s'exprime comme la différence pondérée des coordonnées des nœuds selon chaque dimension. En dimension 1, la fonction d'énergie s'exprime ainsi :

$$E(x) = \sum_{i,j} w_{ij}(x_i - x_j)^2$$

Pour éviter le minimum trivial de cette fonction ($x_1 = x_2 = \cdots = x_{|V|} = 0$), Hall ajoute la contrainte arbitraire $x^T x = 1$.

Vecteurs propres Hall [66] et l'algorithme ACE [88] reposent sur le modèle de Hall pour trouver la position des nœuds. Dans ce modèle quadratique, les positions qui optimisent la fonction d'énergie sont données directement par les vecteurs propres.

La fonction d'énergie E est réécrite sous la forme matricielle $X^T L^w X$. Pour en trouver le minimum, on soustrait la contrainte à la fonction d'énergie ce qui donne l'équation $L = X^T L^w X - \lambda(X^T X - 1)$ que l'on peut dériver pour en chercher le minimum. On cherche alors à résoudre :

$$(L^w - \lambda I)X = 0 \tag{3.3}$$

Les solutions non triviales de cette équation sont les solutions pour lesquelles λ est une valeur propre et X le vecteur propre correspondant. L'énergie du système est alors $E = X^T L^w X = \lambda X^T X = \lambda$ grâce à la contrainte $X^T X = 1$. Les vecteurs propres associés aux valeurs propres les plus petites donnent les positions minimisant l'énergie.

On trie les valeurs propres par ordre croissant. La matrice laplacienne pondérée L^w représentant les $|V|$ nœuds d'un graphe est de rang $|V| - 1$. La première valeur propre est nulle. Elle est associée à un vecteur proportionnel à $(1, 1, \ldots, 1)$. Le vecteur propre associé à la plus petite valeur propre non nulle permet d'obtenir les coordonnées des nœuds pour le dessin en une dimension.

Pour généraliser cette approche à deux dimensions, Hall étend la fonction d'énergie en une dimension sous la forme $E(X,Y) = X^T L^w X + Y^T L^w Y$ avec les contraintes $X^T X = Y^T Y = 1$ ce qui conduit à résoudre un système de une équations identique à l'Equation 3.3 pour chacune des coordonnée. Pour trouver une solution non triviale du problème où les nœuds ont des coordonnées différentes selon les deux axes, on choisit alors le deuxième et troisième vecteurs propres. Cette méthode peut être généralisée à d dimensions.

3.4.4 Heuristiques

D'autres approches d'optimisation heuristiques ont été appliquées au dessin de graphe. Nous évoquons ici les approches de recuit simulé et les algorithmes génétiques.

Recuit simulé

Davidson et Harel [33] proposent d'utiliser la méthode de recuit simulé pour le dessin de graphe. La méthode de recuit simulé est efficace pour résoudre des problèmes d'optimisation discrète dont la fonction de coût ne peut pas être optimisée en épuisant les combinaisons possibles à cause de leur nombre. Inspirée par la mécanique statistique, le recuit simulé cherche à reproduire la cristallisation d'un liquide lors de son refroidissement.

A partir d'une configuration et d'une température initiales, le voisinage d'une configuration est explorée. Si une des configuration voisine permet de réduire la fonction de coût, elle est adoptée. Si elle ne réduit par la fonction de coût, cette configuration peut être adoptée avec une probabilité suivant la distribution de Boltzman :

$$e^{-\frac{C_2 - C_1}{k \cdot T}}$$

où C_i représente la fonction de coût dans la configuration i, T la température du système et k une constante. La température décroît régulièrement, réduisant ainsi la possibilité de changements ne minimisant pas l'énergie. La méthode de recuit simulé permet au système physique d'éviter un minimum local en augmentant temporairement l'énergie du système dans l'espoir de trouver un minimum global.

Pour adapter ce processus au dessin de graphe, plusieurs modifications ont été mises en places. Tout d'abord, les nœuds du graphes ne sont plus placés dans $\mathbb{R}^2$ mais sur une grille, limitant ainsi les positions possibles à un sous-ensemble discret. Les configurations voisines d'une configuration initiale sont définies comme les configurations où un seul nœuds est déplacé. L'algorithme est initialisé aléatoirement.

La fonction de coût proposée par Davidson et Harel [33] est proche du modèle physique de Fruchterman et Reingold. En ne bloquant pas les configurations qui ne font pas diminuer l'énergie globale du système, la méthode par

Algorithm 3 Dessin par recuit simulé

σ et T initiales
while σ n'est pas minimal **do**
  Nouvelle configuration σ'
  r tiré selon $U(0,1)$
  if $C(\sigma') < C(\sigma)$ **then**
    $\sigma \leftarrow \sigma'$
  else if $r < e^{-\frac{C(\sigma')-C(\sigma)}{k \cdot T}}$ **then**
    $\sigma \leftarrow \sigma'$
  end if
  $T \leftarrow \gamma \cdot T$
end while

recuit simulé cherche à éviter les minima locaux pour converger directement vers le minimum global.

Algorithme génétique

Branke et al. [27] utilisent une variation de l'algorithme génétique pour dessiner les graphes. L'algorithme génétique est un algorithme d'optimisation stochastique inspiré de la reproduction biologique et du brassage des gènes. Les solutions potentielles du problème d'optimisation sont appelées individus et forment une population. A partir de cette population, deux individus sont sélectionnés aléatoirement et se combinent pour former un nouvel individu qui est ajouté à la population. L'individu donnant la moins bonne solution est alors supprimé de la population. Ce cycle est répété jusqu'à convergence vers l'optimum du problème. Un nouvel individu est formé en suivant trois étapes : sélection des parents, recombinaison des parents et mutation. La sélection des parents est réalisée en attribuant une probabilité à chaque individu proportionnelle à la qualité de la solution proposée au problème d'optimisation. Une fois deux individus sélectionnés, ils sont recombinés ensemble pour former un nouvel individu : une partie des éléments de solution d'un individu provient de l'un des parents et l'autre partie de l'autre. La recombinaison effectuée, une incertitude est ajoutée sur le résultat via une mutation, qui permet l'exploration de solutions hors de la population.

Concernant le dessin de graphe, chaque individu correspond au dessin d'un graphe et est constitué d'un vecteur de positions réelles correspondant aux positions des nœuds. La sélection des parents se fait aléatoirement, proportionnellement à la qualité du dessin. Une fois les parents sélectionnés, un nouveau dessin est obtenu à partir des deux dessins initiaux : pour l'un des parents, on vient sélectionner un sous-graphe d'une taille fixée à l'avance, et les nœuds non-sélectionnés par ce parent sont positionnés à l'aide des coordonnées de l'autre parent. Pour sélectionner ce sous-graphe, un nœud est choisi aléatoire-

FIGURE 3.9 – Exemple de décomposition multiéchelle sur trois niveau. Les nœuds sélectionnés (en bleu) forment un ensemble indépendant maximal. Un graphe est recréé à partir de ces nœuds uniquement jusqu'à trouver un graphe suffisamment petit (ici trois nœuds).

ment et des nœuds voisins sont ajoutés jusqu'à obtenir un sous-graphe de la taille souhaitée. Enfin, les mutations sont appliquées à chaque nœud sous la forme d'une perturbation suivant une loi normale. Les principaux critères de la fonction de coût dans l'approche de Branke [27] sont le nombre de croisement d'arêtes, la force moyenne appliquée sur chaque nœud par l'algorithme de Eades [42], la longueur d'arête et la distance minimale entre un nœud et une arête.

Plusieurs problèmes spécifiques au dessin de graphe se posent avec cette approche. Le premier est lié à la recombinaison : le dessin d'un graphe a des solutions dont la qualité reste équivalente par symétrie (translation, rotation et symétrie axiale). Recombiner deux dessins peut poser problème s'ils ne sont pas correctement alignés. Branke et al. [27] proposent un réalignement des deux dessins avant d'effectuer la recombinaison. Pour le second problème, le dessin obtenu par recombinaison n'est pas optimal. Pour y remédier, Branke et al. effectuent une itération de l'algorithme par modèle de force d'Eades [42] et conservent le dessin dans la population s'il est meilleur que le dessin original.

3.5 Approches multiéchelles pour le dessin

Les algorithmes par modèle de force classiques, tels que ceux de Fruchterman et Reingold [48] ou de Kamada et Kawai [84], itèrent jusqu'à obtenir atteindre un état d'équilibre correspondant à minimum d'énergie. Il est généralement admis que la résolution pour ces algorithmes requiert $O(|V|)$ itérations à partir d'un état initial aléatoire. La taille du graphe à dessiner est donc un facteur limitant : plus le graphe est grand et plus sa convergence vers un état d'équilibre est difficile.

3.5.1 Structure des algorithmes multiéchelles

Approche classique

L'approche multiéchelle pour le dessin de graphe repose sur différentes échelles d'optimisations. Le graphe à dessiner est résumé à plusieurs échelles

et le dessin est effectué pour chaque échelle. La taille des graphes dans cette suite d'échelles décroît et il est facile de dessiner les graphes à l'échelle la plus petite. A l'inverse, quelques itérations suffisent pour placer correctement les nœuds aux échelles plus hautes puisqu'elles bénéficient d'une bonne initialisation et c'est pour ces échelles que le dessin est le plus coûteux. La position initiale des nœuds à une échelle donnée est obtenue à partir du dessin du graphe de l'échelle précédente, accélérant ainsi la convergence de la méthode de résolution.

L'approche multiéchelle repose sur deux composants principaux, la décomposition récursive et la méthode de placement initial comme illustré dans l'Algorithme 4. La méthode de résolution choisie pour effectuer le dessin peut être un paramètre de l'approche mais c'est généralement l'application de forces qui est utilisé dans ces approches. Le nombre d'itérations nécessaires pour arriver à la convergence par application de forces est en effet grandement réduit, *e.g.* constant pour FM^3 [63]. Nous détaillons ces deux composants dans la suite de cette section.

Algorithm 4 Approche multiéchelle générique pour le dessin

$G_0 = (V_0, E_0) \leftarrow$ Graphe initial

Décomp. réc.

$i \leftarrow 0$
while $|V_i| > 3$ **do**
  $V_{i+1} \leftarrow$ Filtration(V_i)
  $i \leftarrow i + 1$
  Création du graphe $G_i = (V_i, E_i)$
end while

Placement init.

$N \leftarrow i$
for j **de** N **à** 0 **do**
  if j < N **then**
    Placement des nœuds de G_{j-1} selon le dessin de G_j
  end if
  Dessin de G_j (quelques itérations)
end for

Décomposition récursive La décomposition récursive permet de diminuer la taille du graphe initial pour en fournir une séquence de résumés facile à dessiner, *cf.* l'opération de décomposition récursive Figure 3.9. La taille de ces graphes décroît : comme présenté dans la décomposition récursive de l'Algorithme 4, les relations d'inclusions $V_N \subset V_{N-1} \subset \cdots \subset V_1 \subset V_0$ sont vérifiées.

Une décomposition idéale donne un bon résumé du graphe initial en conservant au maximum les propriétés topologiques saillantes du graphe parent. En pratique, la simplicité des décompositions est privilégiée. En effet, il est plus aisée d'obtenir des résultats théoriques sur la complexité temporelle pour des décompositions simples.

La décomposition multiéchelle s'effectue nécessairement sur la structure de graphe et non sur la position des nœuds dans le dessin puisqu'elle intervient avant même que le graphe soit dessiné.

Placement initial Le placement initial propose un dessin initial à chaque niveau de la décomposition pour accélérer sa convergence vers un état d'équilibre. Pour certains algorithmes, seuls les nœuds déjà placés sont propagés au dessin d'une échelle supérieure.

Dans la plupart des cas, une position est attribuée aux nœuds qui n'ont pas encore été placé lors du dessin du niveau précédent. Ce processus, appelé placement intelligent en référence à l'algorithme GRIP [50], est réalisé par plusieurs heuristiques qui fonctionnent plutôt bien en pratique. L'utilité du placement intelligent des nœuds n'est pas avérée, comme le montre le benchmark réalisé sur les algorithmes multiéchelles par Barlet et al. [17].

Approche multiéchelle inverse : du local au global

L'approche multiéchelle classique, présentée dans la section précédente, se focalise sur la structure globale du graphe. Les détails du dessin, *i.e.* les voisinages topologiques, sont ajoutés en remontant la décomposition récursive. L'approche multiéchelle peut aussi être considérée en se focalisant sur le dessin local au lieu de la structure globale. Le graphe est décomposé récursivement pour trouver des voisinages locaux qui sont ensuite combinés ensemble de manière à former le dessin global.

L'étape de décomposition permet de trouver des sous-graphes pour lesquels le dessin de graphe est facile à dessiner (quelques dizaines de sommets). L'étape de placement initial cherche à combiner ces dessin localement optimaux de manière à obtenir un dessin global proche du minimum d'énergie. Cette approche est utilisée par exemple dans TopoLayout [7] ou encore par Hadany et Harel [65].

Cette approche est privilégiée par les algorithmes utilisant une méthode de résolution par optimisation associée au modèle de Kamada et Kawai [84]. En effet, la fonction de stress de Kamada et Kawai permet d'obtenir des dessins proche de l'isométrie au prix d'une complexité de $O(|V|^3)$. Cependant, lors du dessin de sous-graphes obtenus par décomposition récursive, $|V|$ est petit. De plus, seules quelques itérations de la méthodes de résolution sont nécessaires pour optimiser la combinaison du dessin des sous-graphes après l'étape de placement initial.

3.5.2 Méthodes de décomposition récursives

Décomposition par ensemble indépendant maximal

Une méthode de décomposition utilisée par plusieurs algorithmes est basée sur les ensembles maximaux indépendants (aussi appelés Maximal Independent Set ou MIS). Ce problème consiste à trouver un ensemble maximal de sommets qui ne soient pas voisins. Ceci signifie que les sommets du graphe sont dans le MIS ou voisin d'un sommet dans le MIS. On notera que cet ensemble est maximal et non maximum puisque trouver l'ensemble maximal avec le moins de sommets est un problème NP-difficile (*cf.* Robson [118] par exemple).

Gajer et al. [49] proposent d'utiliser le MIS pour sélectionner des sous-ensembles de nœuds sans modifier la structure du graphe d'origine. Un MIS est calculé pour le graphe original. Une fois ce premier ensemble obtenu, le niveau suivant consiste en un MIS en ne considérant plus que les sommets sélectionnés à distance graphe de 2 ou moins. Ce processus est répété en multipliant à chaque fois la distance graphe minimale entre les nœuds par 2. Cette approche garantit qu'il y a $O(\log |V|)$ niveaux dans la filtration. Cette décomposition est aussi utilisée dans l'algorithme GRIP de Gajer et Kobourov [50].

Walshaw [132] propose d'effectuer les décompositions successives en utilisant une filtration basée sur le MIS, mais il recrée un graphe à chaque étape à partir de la structure du graphe parent et des sommets sélectionnés.

Hachul et Jünger [63] utilisent une approche qui est similaire au calcul d'un ensemble maximum indépendant dont la distance graphe minimale est de 3. L'algorithme de décomposition proposé dans FM^3 correspond à l'algorithme séquentiel utilisé pour déterminer un MIS : un nœud est sélectionné et ses voisins à distance 1 et 2 sont désactivés. Ce processus est itéré jusqu'à ce que chaque nœud soit sélectionné, voisin d'un nœud sélectionné ou voisin d'un voisin. A la différence de Walshaw [132] et Gajer et al. [49], le voisinage des nœuds sélectionnés est conservé pour créer le graphe suivant dans la décomposition. Des systèmes solaires sont ainsi créés où les nœuds sélectionnés sont les soleils et les nœuds gravitant autour sont appelés planètes (voisins des soleils) et lunes (voisins des planètes). Les systèmes solaires sont alors agrégés en meta-nœuds et sont reliés par des arêtes s'ils sont voisins dans le graphe parent.

Décomposition par k-centres

Harel et Koren [67] proposent une décomposition des graphes basée sur des k-centres, k sommets minimisant la distance inter-sommets dans le graphe. Ce problème étant NP-difficile, l'algorithme utilise un algorithme approché efficace permettant d'obtenir la décomposition en un temps raisonnable. L'approche de Kamada et Kawai [84] est utilisée pour effectuer les dessins de graphe.

Décomposition par couverture d'arête

L'approche multi-échelle présentée par Hadany et Harel [65] résout les problème intermédiaire à l'aide d'une technique d'optimisation (descente de gradient). Le clustering est effectué non pas en sélectionnant des nœuds mais en sélectionnant un ensemble d'arêtes qui sont compressées de manière à créer le graphe suivant. Ces arêtes sont sélectionnés de manière à minimiser une fonction de coût comprenant trois objectifs : créer de petit clusters bien répartis sur l'ensemble du graphe, conserver des nœuds de haut degrés et éviter de trop simplifier le graphe. Le dessin en lui même est effectué en trois phases : une première phase de dessin du graphe, un appel récursif à la fonction de dessin pour le niveau suivant, puis une nouvelle phase de dessin pour optimiser le résultat final. Pour les deux phases de dessins, une fonction d'énergie similaire à celle de Kamada et Kawai [84] (à paramètre d'échelle près) est optimisée. Cette fonction est cependant limitée au interactions avec les nœuds dans un voisinage de distance fixe. Avec chaque itération de l'algorithme, de dessin, ce rayon augmente de manière à pouvoir obtenir la structure globale du graphe.

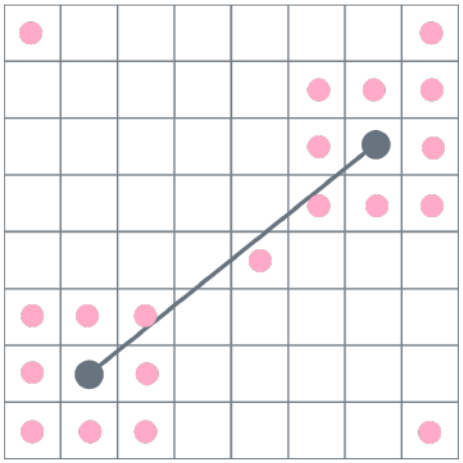
3.5.3 Placement initial

L'initialisation joue un rôle important dans les algorithmes de dessin par analogie physique : une bonne position des nœuds initiales contribue à accélérer le dessin. Lorsque le dessin est proche du minimum d'énergie, l'optimisation de ce dessin est facilitée. Un bon placement initial permet de réduire le nombre d'itérations des méthodes de résolutions, *i.e.* réduire les $O(|V|)$ itérations des méthodes par applications de forces. Par exemple, FM^3 [63] effectue un nombre constant d'itérations à chaque niveau.

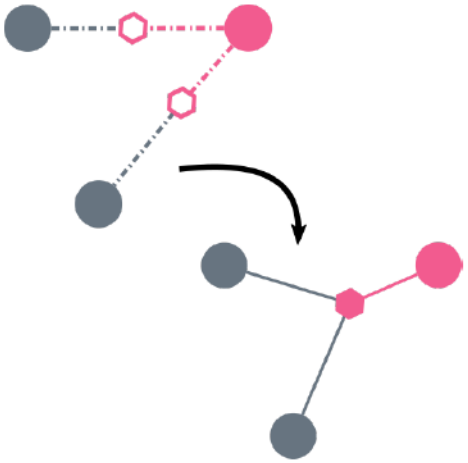
Un placement initial intelligent est utilisé dans la plupart des algorithmes multiéchelle afin de placer les nœuds qui ne sont pas encore placés avant d'effectuer le dessin d'un niveau. Ce placement repose généralement sur la position de voisins déjà placés qui servent à trouver la position initiale d'un nœud à placer.

Le premier exemple de placement initial provient de Tunkenlang [127] qui propose une méthode pour placer les nœuds au fur et à mesure. Ce placement sert à placer le nœud proche de sa position finale et un raffinement est utilisé (ici l'optimisation d'une fonction de coût) pour trouver sa position exacte. Pour trouver le positionnement optimal, l'espace du dessin est décomposé en grille et les cellules adjacentes aux voisins du nœuds sont évaluées pour trouver la cellule minimisant la fonction de coût. D'autres cellules sont aussi considérées, comme les coins du dessin ainsi que les cellules placées autour du barycentre des nœuds voisins, , *cf.* Figure 3.10a.

Gajer et al. [49] soulignent par exemple qu'en sélectionnant deux voisins, il est possible de placer un nœud en respectant les distances graphes entre



(a) Grille des positions à évaluer pour optimiser la position du nœud rose, *cf.* Tunkelang [127].



(b) Placement d'un nœud du cluster rose ayant deux voisins selon la méthode de FM^3 [63].

FIGURE 3.10 – Exemples de placement initial des nœuds avant dessin. *(a)* Placement par la méthode de FM^3. Le nœud rose, dans le cluster rose, est placé sur le segment connectant son soleil à d'autres soleils. La position finale du nœud est le barycentre de ces position. *(b)* Placement initial par la méthode de Tunkelang [127]. Sur la grille, un ensemble de positions raisonnables sont évaluées et celle minimisant la fonction de coût est choisie comme position initiale.

les nœuds (deux positions possibles). Pour trouver la position optimale, ils proposent alors d'utiliser non pas deux nœuds mais trois, ce qui conduit à un système surdéterminé, ce qui peut être contourné en cherchant la position optimale pour chaque paire de voisins et en plaçant le nœud au barycentre de ces positions.

Pour Hachul et Jünger [63] les planètes et les lunes (*i.e.* les nœuds pas encore placés dans les systèmes solaires) sont placés sur les segments connectant les soleils (déjà placés). Ces nœuds sont placés à une distance proportionnelle à leur distance pondérée à leur soleil. Si des nœuds se trouvent sur plusieurs segments par ce processus, leur position initiale correspond au barycentre de ces positions, *cf.* Figure 3.10b.

La méthode multigrille algébrique (aussi connue sous le nom de AMG ou Algebraic Multigrid) est basée sur une réduction itérative de dimension. Koren et al. [88, 89] la transposent au dessin de graphe. Cette méthode est utilisée pour minimiser la fonction d'énergie proposée par Hall [66], *cf.* Section 3.4.3. Dans ce modèle, la position des nœuds est obtenue grâce aux vecteurs propres associés aux valeurs propres les plus petites. La décomposition récursive est obtenue par interpolation (exprimant de manière algébrique création de méta-nœud). A chaque niveau, les vecteurs propres sont calculés de manière à trouver

le dessin. Après interpolation inverse, les vecteurs obtenus (*i.e.* les positions initiales) sont presque les vecteurs propres du systèmes. Quelques itérations d'un algorithme de puissance itérée [93] permet de trouver les vecteurs propres.

D'autres variantes de ce placement existent comme par exemple à la position médiane de la position des voisins pour chaque coordonnée, ce qui permet d'éviter les effets des nœuds dessinés loin des autres. Ces variantes sont recensées dans l'évaluation de Bartel et al. [17].

3.6 Parallélisme et distribution

3.6.1 Dessin de graphe et GPU

L'accélération des algorithmes de dessin de graphe passe aussi par l'amélioration des capacités de calculs et notamment les améliorations des cartes graphiques (Graphical Processing Unit ou GPU). Les GPU sont développées notamment pour la génération, l'affichage et l'animation de simulations physiques ou multi-particules. Des cartes graphiques peuvent être utilisées pour simuler des mouvements de fluides ou de fumées. Ces animations peuvent être générées en faisant interagir des particules simulées. Ces interactions permettent de déterminer le mouvement des particules et donc l'animation qui en découle. Ces simulations multi-particules sont possibles grâce au haut niveau de parallélisme permis par le GPU.

Le dessin de graphe par modèle de force fonctionne à la manière des systèmes physique multi-particules et des adaptations des algorithmes de dessin de graphe ont donc été réalisées en tirant parti du haut parallélisme des GPU. A l'aide de ces nouvelles capacités de calcul et de parallélisation, les algorithmes, nécessitant beaucoup de capacités de calcul, se trouvent accélérés. Par exemple, en 2007, Auber et Chiricota [10] présentent une adaptation des algorithmes par force de rappel utilisant le GPU. Dans le même temps, Frishman et Tal [47] proposent une implémentation d'un algorithme de dessin de graphe multi-échelle tirent parti des capacités de calcul offertes par le GPU. Enfin, Godiyal *et al.* [57] proposent une adaptation des algorithmes de dessin tirant parti des forces multipôles fonctionnant à l'aide du GPU.

3.6.2 Dessin de graphe déporté

Abello et al. proposent l'algorithme Ask-GraphView [1] en 2006. Le principe est de déporter le calcul du dessin de graphe sur un serveur et d'offrir à l'utilisateur un client léger permettant l'interaction avec le graphe à dessiner. Cet algorithme se base sur les travaux existants du dessin de graphe hiérarchique, qui vise à dessiner les graphes pour lesquels une structure hiérarchique est connue [43, 44]. Dans leur approche, Abello et al. utilisent le dessin hiérarchique pour le transposer à un paradigme client-serveur. Ainsi, lorsque

l'utilisateur souhaite ouvrir un cluster de la hiérarchie pour en visualiser les éléments, il demande au serveur d'effectuer le dessin des nœuds contenus dans ce méta-nœud et de lui envoyer leurs positions. Pour que les échanges entre le client et le serveur fonctionnent, Abello et al. dimensionnent les requêtes possibles de manière à ce que le dessin de graphe à calculer tienne dans la RAM du serveur. Pour ce faire, ils présentent des structures de données adaptées au calcul sur le serveur, qui sont équilibrées de manière à obtenir une charge équivalente quelles que soient les requêtes du client. Par cette première approche, Abello et al. proposent une première version du dessin de graphe Out of core en tirant parti des capacités de calculs disponibles sur des serveurs au lieu de celles moindres des ordinateurs de bureau.

Plus récemment, des algorithmes de dessin par modèle de force ont été développés pour des murs d'images [28, 107, 126]. Ces architectures sont composées de plusieurs ordinateurs stockant uniquement les nœuds à afficher sur l'écran leur étant associé. Une telle architecture permet de dessiner des graphes avec des millions d'arêtes en évitant l'encombrement visuel grâce à la taille des écrans combinés : sur un seul écran, l'encombrement visuel est atteint rapidement. Cette famille d'algorithmes se concentre sur les méthodes pour distribuer efficacement les nœuds sur les machines et utilise un algorithme similaire à celui de Fruchterman et Reingold [48] pour les sous-graphes contenus sur chaque machine. Certains nœuds sont autorisés à migrer d'une machine à l'autre, lorsque le dessin les place à la frontière de plusieurs écrans. Ces propositions sont plus adaptées à des graphes dont le partitionnement est facile avec peu de chevauchements. Dans le cas de graphes comme des graphes petits mondes, ces techniques ne sont pas adaptées.

Langevin et al. [94] proposent une architecture de plateforme pour dessin de graphe stockés sur cluster comprenant un algorithme de dessin par modèle de force. Langevin propose de décomposer le graphe de manière récursive en communautés imbriquées les unes aux autres. Les communautés sont obtenues à l'aide de la méthode de Louvain de manière récursive et ce sont les communautés obtenues de cette manière qui sont dessinées à l'aide de la méthode de Fruchterman et Reingold [48] et non les nœuds directement. Les dessins sont regroupés en tuiles qui peuvent facilement être envoyées à un client pour effectuer le rendu en cas de zoom au sein de la visualisation du graphe.

Chapitre 4

Adapter les techniques de dessin de graphe au paradigme MapReduce

4.1 Introduction

Dans l'état de l'art (Chapitre 3), nous avons présenté différents algorithmes de dessin de graphe par modèle de force que nous avons décomposé en trois types de composants : l'analogie physique, la méthode de résolution et les techniques algorithmiques pour la convergence. Ces différents éléments peuvent être combinés ensemble de plusieurs manières et certains se prêtent plus à la parallélisation et à la distribution que d'autres.

Dans ce chapitre, nous présentons les analogies physiques et les méthodes de résolution qui sont le plus adaptées pour le dessin de graphe distribué sur cluster. Les méthodes présentées dans l'état de l'art peuvent être implémentées dans le paradigme MapReduce mais elles ne permettent pas toutes de tirer parti de la scalabilité horizontale offerte par le calcul sur cluster.

4.2 Méthode de résolution pour le dessin distribué

Les méthodes de résolutions décrites dans la Section 3.4.2 se décomposent en trois grandes familles : les méthodes d'optimisation mathématiques, l'application itérative de forces et les heuristiques. Dans cette section, nous explorons les possibilités de transposer ces méthodes de résolution dans le paradigme MapReduce.

4.2.1 Méthode de Newton-Raphson

Les méthodes d'optimisation telles que présentées dans la Section 3.4.2 sont séquentielles. Pour la méthode de Newton-Raphson par exemple, il s'agit de résoudre un système d'équations linéaires de manière à obtenir la position minimisant l'énergie globale pour le point sélectionné.

Optimisation nœud à nœud

Dans l'algorithme de dessin de Kamada et Kawai [84], la méthode d'optimisation de Newton-Raphson est utilisée pour minimiser l'énergie globale un nœud après l'autre. Dans ce cas de figure, aucun bénéfice n'est tiré du parallélisme offert par le cluster. En effet, l'étape d'optimisation bloque la poursuite de l'algorithme.

Optimisations simultanées

A l'inverse, si l'on tente d'optimiser plusieurs nœuds simultanément, il n'y a pas de garantie que l'énergie globale du système décroisse. L'optimisation de plusieurs nœuds de manière simultanée fait décroître l'énergie à condition que les nœuds sélectionnés ne soient pas en interaction. Cependant, dans la plupart des analogies physiques pour le dessin, tous les nœuds sont en interaction (*cf.* Section 3.2). Optimiser plusieurs nœuds simultanément avec la méthode de Newton-Raphson n'assure donc pas la convergence de la méthode.

4.2.2 Descente de gradient stochastique

Dans le domaine de l'apprentissage automatique, l'optimisation est un passage nécessaire pour adapter le modèle mathématique aux données observées. Pour les problèmes d'apprentissage supervisé, il s'agit de trouver le vecteur de poids optimisant la fonction de coût dans le cas des données observées. C'est par exemple le cas pour les machines à vecteurs de support, pour la recherche du maximum de vraisemblance, pour la régression logistique *etc.* Ces problèmes traitent aussi des données massives : les données contenues dans la base à partir desquelles on cherche à régler le modèle mathématique.

Approche standard

Soit f_i la fonction relative à l'observation i et $\mathbf{w}$ le vecteur de poids à optimiser. On note C la fonction de coût définie par $C(\mathbf{w}) = \sum_i f_i(\mathbf{w})$.

Plutôt que d'optimiser le vecteur de poids w pour la fonction C directement, ce qui constitue un problème difficile, la descente de gradient stochastique propose d'optimiser le vecteur de poids selon un sous-ensemble des fonctions f_i. Le vecteur de poids $\mathbf{w}$ est déplacé dans le sens de la plus grande pente

relative au sous-ensemble d'observations $\{i\}$. Les observations sont visitées dans un ordre aléatoire.

En pratique, le vecteur de poids $\mathbf{w}$ est optimisé pour une permutation de l'ensemble des observations. Une fois le vecteur $\mathbf{w}$ optimisé selon chaque observation, une nouvelle permutation est tirée. Ce processus itère jusqu'à convergence. En un passage sur les données, $\mathbf{w}$ est optimisé une fois par fonction. Il est donc nécessaire d'effectuer plusieurs boucles pour obtenir un minimum global.

Bousquet et Bouttou [26] proposent une comparaison de cette méthode avec la descente de gradient classique. Dans leur comparaison, Bousquet et Bouttou soulignent que la descente de gradient stochastique est indépendante du nombre d'observations. Cet algorithme est donc adapté pour l'optimisation de grands jeux de données. Cependant, l'algorithme de descente de gradient stochastique reste un algorithme essentiellement séquentiel difficile à paralléliser.

Descente stochastique avec retard

En introduisant un retard dans l'application du gradient, différentes optimisations locales peuvent se synchroniser de manière à trouver un optimum global. Soit $\tau \in \mathbb{N}$ le paramètre de retard. A l'itération t, le poids w_t est optimisé selon le gradient $g_{t-\tau}$. En choisissant un paramètre de retard $\tau = n - 1$, avec n le nombre de machines, chaque machine effectue indépendamment l'optimisation du problème en visitant un état optimisé par une machine différente pour une observation différente. Ces différentes optimisations se combinent pour fournir l'optimum global du problème.

La descente de gradient stochastique avec retard [95, 141] est utilisée dans le paradigme MapReduce. Si n machines optimisent chacune le vecteur $\mathbf{w}$ pour une fonction f_i différente, elles ne peuvent pas modifier $\mathbf{w}$ de manière concurrente sans perdre aucun calcul.

Langford et al. [95] évaluent les performances de cette méthode de résolution dans un environnement séquentiel. Un retard de 10 cycles n'a pas d'effet majeur sur l'erreur commise. A partir de 100 cycles de retard, l'erreur commise est encore acceptable (erreur relative de 10% pour les données simulés et de 20% pour les données réelles). Les performances de l'algorithme sont grandement améliorées avec la parallélisation. Utiliser 4 threads divise le temps d'exécution par 2. Avec 7 threads, le temps d'exécution est divisé par 4.

Adaptation pour le dessin de graphe

Le problèmes d'apprentissage automatique supervisé et celui du dessin de graphe s'expriment de manière différente, ce qui complique l'adaptation de la descente de gradient stochastique pour le dessin de graphe. Dans le problème

d'apprentissage, le vecteur de paramètre à optimiser contient un nombre restreint de variables à optimiser (généralement de l'ordre de la dizaine) et un nombre important d'observations pour optimiser ces variables. Le problème du dessin de graphe contient de nombreuses variables à optimiser, *i.e.* $d \cdot |V|$ avec d le nombre de dimensions du dessin, et l'optimisation repose sur la topologie du graphe, ce qui peut représenter jusqu'à $|V|^2$ interactions.

La solution généralement adoptée par les algorithmes de dessin utilisant des méthodes d'optimisation mathématique est de restreindre l'espace des variables en fixant la position de tous les nœuds à l'exception d'un sous-ensemble. En sélectionnant plusieurs variables indépendantes les unes des autres du point de vue de la fonction de coût, *cf.* section précédente sur la méthode de Newton-Raphson, il pourrait être envisagé d'utiliser ce type d'approche pour calculer la position optimale des nœuds.

En poussant l'analogie plus loin, sélectionner les interactions est équivalent à restreindre l'espace des observations utilisées dans les approches de descentes de gradient pour l'apprentissage automatique. Les sélections d'interactions ont déjà été utilisées en dessin de graphe pour l'approximation du calcul des forces à des sous-ensembles de nœuds proposée par Harel et Koren [67] (*cf.* section 3.3.1). Une conjugaison de la descente de gradient stochastique avec retard et des sélections des interactions pourrait être envisagée pour obtenir la position optimale des nœuds dans le paradigme MapReduce.

4.2.3 Application itérative de forces

L'application itérative de forces se prête à la distribution. Le calcul des forces est parallélisable puisque les forces appliquées à chaque nœud peuvent être calculées indépendamment les unes des autres. Cette approche tire parti du parallélisme offert par le cluster à condition que tous les nœuds soient déplacés simultanément. Si les nœuds sont déplacés un par un, la parallélisation de l'algorithme n'est pas possible, *cf.* Section 4.2.2. L'approche par application itérative de forces est la méthode retenue dans notre contribution [75] et par Arleo et al. [8].

Une fois les forces obtenues, leur application à tous les nœuds s'effectue en MapReduce à l'aide d'une opération Map. Pour chaque nœud, sa position et la valeur des forces sont ajoutées grâce à l'opération Map. En une passe, la position de l'ensemble des nœuds est mise à jour, sans nécessiter de déplacer les données (étape de Shuffle avant l'opération de Reduce).

Cette approche est adaptée à l'environnement MapReduce mais elle n'assure par la convergence de l'algorithme à elle seule. Pour assurer la convergence de cette approche, il est nécessaire d'utiliser une technique de refroidissement, comme présenté dans Fruchterman et Reingold [48] (*cf.* Section 3.4.1). Cette technique est aussi applicable dans le paradigme MapReduce en conservant une variable globale, partagée par toutes les machines via Broadcast, conte-

nant le facteur de refroidissement à appliquer. L'opération Map d'application des forces utilise alors cette variable pour calculer les forces réelles. Cette variable est mise à jour après chaque application des forces puis retransmise à chaque machine du cluster.

Le calcul des forces constitue le point bloquant de l'approche. Chaque interaction peut être calculée indépendamment. Le calcul des forces est donc parallélisable. Cependant, le nombre d'interactions à calculer peut rendre difficile la distribution du calcul. Dans la suite du chapitre, nous parlerons de forces pour décrire les interactions de l'analogie physique.

4.3 Analogies physiques et dessin distribué

Dans cette section, nous revenons sur les analogies physiques et leur transposition dans le paradigme MapReduce.

4.3.1 Forces d'attraction

Les forces d'attraction proposées dans les analogies physiques sont généralement très similaires : elles s'appliquent entre chaque paire de nœuds connectés et suivent la loi de Hooke ou une fonction logarithmique selon les forces de répulsion proposées par l'algorithme. Leur longueur nominale, si elle existe, permet de déterminer la longueur optimale d'une arête. Ce modèle d'interactions est utilisé dans les algorithmes de Quinn [116], d'Eades [42], de Fruchterman et Reingold [48] et pour les algorithmes multiéchelles s'en inspirant [50, 63]. Pour cette analogie, $2 \cdot |E|$ forces d'attraction sont calculées.

Pour ces forces, la transposition dans le paradigme MapReduce est possible. Cependant, à partir des arêtes stockées de manière distribuée, la méthode la plus simple pour calculer les forces est d'utiliser le paradigme Pregel [102], aussi connu sous le nom de *Think Like A Vertex* (généralement abrégé en TLAV). Ce paradigme de calcul distribué est spécifiquement conçu pour les algorithmes de graphes utilisant les arêtes pour passer des messages le long des arêtes simultanément (*cf.* section 2.2.2).

Pour effectuer le calcul dans ce paradigme, chaque nœud transmet sa position à ses voisins ainsi que les informations sur l'arête (la taille de l'arête par exemple). Une fois ces messages obtenus, la force est calculée entre chaque paire de nœuds voisins et agrégée par une opération Reduce sur chaque nœud. En une ronde de Pregel, les forces d'attraction sont calculées pour l'ensemble des nœuds du graphe. Le code Spark (Scala) permettant de calculer les forces de répulsion est donné en Figure 4.1.

```
def attractiveForce(ctx: EdgeContext) = {
    //Definition des variables
    val coordSrc:(Double,Double) = ctx.srcAttr;
    val coordDst:(Double,Double) = ctx.dstAttr;
    val edgeLength:Int = ctx.attr;

    //Calcul des parametres
    val distX = coordSrc._1 - coordDst._1;
    val distY = coordSrc._2 - coordDst._2;
    val dist  = math.hypot(distX, distY);

    //Calcul de la force
    val d     = (dist - edgeLength);
    val forceX = d^2 / K * distX / dist;
    val forceY = d^2 / K * distY / dist;

    return (forceX, forceY, 1L);
}

def mergeMsg(a, b) = {
    //Reduce effectue la somme des valeurs des messages
    return (a._1 + b._1, a._2 + b._2, a._3 + b._3);
}

val vertexAttForces =
    graph.aggregateMessages(
        ctx => {
            val attForces = attractiveForce(ctx);
            ctx.sendToDst(attForces);},
        mergeMsg);
```

FIGURE 4.1 – Code Spark (Scala) pour calculer les forces d'attraction

Partitionnement

Les messages étant principalement transmis entre nœuds voisins, les implémentations de Pregel comme GraphX [138] ou Giraph [12] optimisent le partitionnement des nœuds de manière à minimiser le nombre de messages transmis entre les partitions. Le partitionnement initial détermine quels nœuds se trouvent sur quelle partition. Cette étape a des conséquences importantes sur les performances de l'algorithme puisque c'est en partie elle qui détermine la quantité de messages échangés sur le réseau. Si deux nœuds se trouvent sur une même partition, les messages qu'ils s'envoient ne passent pas par le réseau mais restent au sein de la machine ce qui a pour effet d'améliorer les performances.

Dans l'algorithme d'Arleo et al. [8], la topologie du graphe joue un rôle important puisqu'elle est utilisée pour les forces d'attraction et les forces de répulsion approchées. Avec un partitionnement efficace, les voisinages peuvent se retrouver sur des machines proches et réduire les messages transmis sur le réseau. Par défaut, le partitionnement dans Giraph est effectué à l'aide d'une fonction de hashage dans l'objectif d'obtenir des partitions équilibrées en nombre de nœuds.

Dans Clint, l'algorithme d'Arleo et al. , le partitionnement est effectué à l'aide de Spinner [104, 130], un algorithme créant des partitions équilibrées en se basant sur la topologie du graphe. L'enjeu de cet algorithme, développé pour fonctionner avec Giraph [12] est de trouver la balance entre la localité des partitionnements (*i.e.* principalement des nœuds voisins) et l'équilibre entre les partitions. En utilisant plusieurs rondes de Pregel, les nœuds sont amenés à migrer d'une partition à une autre selon les partitions auxquelles appartiennent leur voisinage. Spinner utilise deux critères pour atteindre cette balance : un score mesurant la qualité du partitionnement en cas de migration vers une autre partition et une pénalité à ce score en cas de déséquilibre des partitions. En quelques itérations, cet équilibre est atteint à partir d'un partitionnement initial indépendant de la topologie.

GraphX [138] propose quatre partitionnements des arêtes, dont deux sont identiques en utilisant l'ordre canonique pour les arêtes. Ces partitionnements sont décrits et comparés dans la Section 5.4.2.

4.3.2 Forces de répulsion exactes

Modèle de Fruchterman et Reingold

Les forces de répulsion exactes proposées dans différentes analogies physiques sont similaires : elles s'appliquent entre chaque paire de nœuds et leur intensité décroît rapidement avec la distance (loi du carré inverse, loi inverse *etc.*). C'est ce que propose Fruchterman et Reingold [48] et Eades [42] par exemple. Il y a $O(|V|^2)$ forces de répulsion calculées de cette manière.

Pour calculer les forces exactes, chaque machine doit transmettre sur le réseau la position des nœuds de sa partition à l'ensemble des autres machines du cluster. Soit k le nombre de machine du cluster. Si chaque machine stocke la position de $\frac{|V|}{k}$ nœuds, le nombre de messages envoyés de cette manière est de $(k-1) \cdot |V|$. La transmission du jeu de données entier à plusieurs machines ne peut être réalisée dans un environnement distribué. Après la transmission des messages, chaque machine stocke en mémoire la position de l'ensemble des nœuds. Cette approche ne présente donc pas la scalabilité horizontale nécessaire aux applications dans un environnement distribué.

Expérimentalement, nous avons implémenté ce calcul à l'aide du produit cartésien de la position des nœuds. L'agrégation du calcul des forces en une force globale est obtenue par une opération de Reduce. Cette implémentation mène au même constat : l'application n'arrive pas à sa conclusion par manque de mémoire.

Modèle de Kamada et Kawai

Kamada et Kawai [84] proposent d'ajouter des ressorts entre chaque paire de nœuds dont la longueur nominale correspond à la distance graphe, ce qui revient aussi à calculer $O(|V|^2)$ forces. Cette analogie physique requiert de calculer la distance graphe entre chaque paire de nœuds. Cette opération est difficile à mener dans un environnement séquentiel.

Dans un environnement distribué, les nœuds et arêtes du graphe sont répartis sur plusieurs machines. Calculer la distance graphe entre chaque paire de nœuds est réalisable en distribué (voir par exemple Holzer [79]) mais nécessite $O(|V|)$ rondes de Pregel pour converger vers la solution. De plus, le stockage de la distance graphe pour toutes les paires de nœuds a une complexité spatiale de $O(|V|^2)$, ce qui n'est pas envisageable pour de grands graphes. Cette étape représente une phase de précalcul pour l'algorithme de Kamada et Kawai, qui intervient avant le calcul des interactions.

L'analogie physique proposée par Kamada et Kawai [84] ne peut pas être approximée à l'aide des approches à N corps. En effet, les forces de rappel proposées dans cette analogie ne peuvent pas être négligées pour les nœuds éloignés. Il est donc difficile de proposer une approximation de ces forces de rappel, comme il est possible de le faire pour les forces de répulsion.

4.3.3 Approximation par grille

Fruchterman et Reingold [48] et Walshaw [132] utilisent une grille pour réduire le nombre de nœuds utilisés pour calculer les forces de répulsion. Au lieu de calculer chaque interaction, les interactions ne sont prises en compte qu'avec les nœuds se trouvant dans une case adjacente de la grille. Cette technique est implémentable dans un environnement MapReduce. Implémenter cette tech-

```
type Vertex = (Long, (Double, Double));

def repulsiveForce(vertexPair: (Vertex,Vertex) ) = {
    //Definition des variables
    val coordSrc:(Double, Double) = v._1._2;
    val idDst: Long = v._2._1;
    val coordDst:(Double, Double) = v._2._2;

    //Calcul des parametres
    val distX = coordSrc._1 - coordDst._1;
    val distY = coordSrc._2 - coordDst._2;
    val dist  = math.hypot(distX, distY);

    //Calcul de la force
    val d     = (dist - edgeLength);
    val forceX = - K^2 / d * distX / dist;
    val forceY = - K^2 / d * distY / dist;

    return (idDst,(forceX, forceY, 1L));
}

def totalForce(a, b) = {
    //Reduce effectue la somme des valeurs des messages
    return (a._1 + b._1, a._2 + b._2, a._3 + b._3);
}

val cartPos = vPos.cartesian(vPos);
val vertexRepForces =
    cartPos.map(
        vertexPair => repulsiveForce(vertexPair))
    .reduceByKey(
        (v1,v2) => totalForce(v1,v2));
```

FIGURE 4.2 – Code Spark (Scala) pour calculer les forces entre chaque paire de nœuds (Kamada et Kawai ou forces de répulsion exactes)

nique revient à faire le calcul des forces de répulsion exactes avec les cases adjacentes. Cependant, cette technique accélère le temps de calcul mais ne réduit pas la complexité dans le pire des cas. En effet, une mauvaise répartition des nœuds (tous très proches) implique le calcul de toutes les interactions, c'est-à-dire la force de répulsion exacte. Dans le pire des cas, ce calcul n'est pas réalisable dans le paradigme MapReduce.

4.3.4 Méthode de Barnes et Hut

Approche classique

Comme vu en Section 3.3.2, l'approche de Barnes et Hut [15] permet de réduire la complexité à $O(|V| \log |V|)$ pour les forces de répulsion.

Barnes et Hut utilisent une décomposition spatiale récursive permettant de définir les $O(\log |V|)$ interactions. Cet arbre contient $O(|V| \log |V|)$ nœuds et doit être partagé avec chacune des k machines du cluster. Il y a alors $O(k \cdot |V| \log |V|)$ messages transmis ce qui ne permet pas d'assurer la scalabilité de l'algorithme. Des adaptations sont donc nécessaires dans le paradigme MapReduce.

Arbre à profondeur fixe

Adelman [2] (note non publiée) propose d'effectuer une décomposition spatiale grâce à un arbre dont la profondeur est fixée. A partir de cette décomposition, les forces de répulsion entre nœuds éloignés sont calculées par la méthode de Barnes et Hut. Pour les nœuds proches, *i.e.* contenus dans la même cellule, les forces de répulsion sont calculées directement. La complexité ce cet algorithme pour une répartition uniforme des points est de $O(|V|^{3/2})$. Adelman propose une implémentation de cet algorithme dans le paradigme MapReduce.

kd-tree à profondeur fixe

Warren et Salmon [133] travaillent sur des simulations d'astrophysique et utilisent l'approximation de Barnes et Hut pour calculer les forces gravitationnelles. Leurs simulations sont réalisées sur des infrastructures HPC contenant 512 cœurs. Deux idées sont utilisées pour paralléliser l'approche de Barnes et Hut à très grande échelle.

Premièrement, les particules sont assignées à un processeur en utilisant une décomposition orthogonale récursive, c'est à dire une décomposition récursive de l'espace en deux qui équilibre le nombre de particules dans chaque partition. Cette décomposition est similaire à celle d'un kd-tree mais la récursion ne descend pas jusqu'à obtenir l'ensemble des nœuds en tant que feuille. De cette manière, les processeurs travaillent sur le même nombre de particules et effectuent donc un nombre de calculs comparables.

Deuxièmement, une fois cette décomposition effectuée, des arbres localement essentiels (locally essential trees) sont calculés pour chaque processeur. Ces arbres sont ceux de Barnes et Hut tronqués de manière à contenir uniquement l'information nécessaire pour effectuer le calcul des interactions pour les nœuds d'un processeur donné. Cette opération est réalisée après $log(P)$ échanges entres processeurs, P étant le nombre de processeurs. Cette propriété est obtenue grâce à la décomposition orthogonale récursive : chaque processeur transmet les informations nécessaires à la création de l'arbre à la moitié des processeurs décrivant l'espace adjacent au sien pour les différents niveaux.

Cette approche est intéressante du point de vue du parallélisme, mais il n'est pas évident qu'elle puisse être transposée dans un environnement MapReduce. Elle souligne tout de même la difficulté pour les algorithmes parallèles, et à fortiori distribués, de traiter les différentes échelles de calculs (ici les nœuds proches et éloignés) dans une même approche.

4.3.5 Méthodes multipôles rapides

L'adaptation des méthodes multipôles rapides est réalisée dans les environnements HPC.

Les méthodes multipôles rapides, *cf.* Section 3.3.2, évaluent différemment le calcul des interactions proches et éloignées. Pour les interactions entre particules proches, le calcul exact est effectué. Pour les interactions éloignées, une approximation du calcul est réalisée sur la base d'un vecteur de poids représentant le développement limité des interactions au centre de la cellule.

Notations

On note k le nombre de cellules dans la grille régulière. On note p le nombre de coefficients du développement limité, *i.e.* le facteur de précision. Enfin, on note M le nombre de mappers.

On suppose aussi que les nœuds sont répartis uniformément dans l'espace du dessin. On suppose aussi que les nœuds n'ont pas été partitionnés. A chaque itération de l'algorithme de dessin, l'ensemble des nœuds est déplacé, ce qui rend difficile la partition.

Évaluation des interactions éloignées

L'évaluation des interactions éloignées nécessite plusieurs étapes :

- le découpage de l'espace en une grille régulière,
- l'évaluation pour chaque cellule du vecteur des poids correspondant au développement limité,
- la recombinaison de ces poids pour trouver les interactions éloignées pour chaque cellule,

- l'application des interactions éloignées à chaque nœud.

Initialisation des vecteurs de poids. A partir d'une grille régulière, le découpage de l'espace en une grille régulière est réalisable en une étape de Map. Pour chaque nœud, l'identifiant de sa cellule est déterminé en utilisant le quotient et le reste de la division euclidienne pour déterminer ses coordonnées dans la grille.

Dans la même opération de Map, chaque nœud calcule sa contribution au développement limité au sein de sa cellule. Il y a ainsi $|V|$ vecteurs de coefficients générés de cette manière. Au sein de chaque mapper, les vecteurs sont combinés en k vecteurs totaux correspondant à la contribution des nœuds du mapper à chaque cellule.

Enfin, l'opération de Reduce est effectuée sur un unique Reducer. Elle combine les $k \cdot M$ vecteurs qui lui sont transmis en k vecteurs correspondant aux coefficients du développement limité associé à chaque cellule.

Lors de cette étape, le shuffle des données représente la transmission de $k \cdot M$ vecteurs de poids. Le vecteur de coefficients est encodé à l'aide de p Double, soit donc $4 \cdot p$ octets. Pour 50 mappers, une grille de 100 de coté et une précision $p = 10$ coefficients, il y a $50 \cdot 100^2 \cdot 10 \cdot 4 = 20$ Mo de données transmises lors de l'étape de Shuffle.

Combinaison et interactions éloignées La combinaison des poids pour les cellules bien séparées permet d'obtenir l'approximation des interactions éloignées de chaque cellule.

Dans la méthode non récursive, les poids sont déplacés vers le centre de chaque cellule bien séparée puis sommés en ce point. De cette manière, chaque cellule dispose de l'approximation des interactions éloignées en son centre.

Dans la méthode récursive, les poids sont d'abord combinés en suivant un arbre de décomposition (à la manière de Barnes et Hut [15], *cf.* Section 3.3.2). Les cellules bien séparées sont évaluées séparément à chaque niveau de la décomposition, avec une précision croissante au fur et à mesure de la descente dans l'arbre de décomposition.

Pour ces deux méthodes, la combinaison intervient pour un petit nombre de variables, les k vecteurs de poids de la méthode récursive. La combinaison peut donc être réalisée directement sur une unique machine. Cette opération prend en entrée les k vecteurs de poids correspondant à chaque cellule et retourne la combinaison de poids permettant de calculer les interactions éloignées au centre de chaque cellule.

Application des forces Pour appliquer les forces, la combinaison de poids pour le calcul des interactions éloignées doit être disponible sur chaque machine. En une étape de Map, chaque nœud trouve la cellule qui lui correspond

et calcule les interactions éloignées à l'aide du vecteur de poids associé à la cellule.

Chaque mapper stocke le vecteur de poids. Pour une grille de 100 de coté et une précision $p = 10$ coefficients, il y a $100^2 \cdot 10 \cdot 4 = 400$ ko de données stockées sur chaque mapper.

Évaluation des interactions proches

Les interactions proches sont évaluées nœud à nœud pour les cellules de la grille, adjacentes par un bord ou un coin. Pour ces interactions, le calcul exact est réalisé.

Une opération de Map permet de trouver la position de chaque nœud dans la grille. Cependant, le calcul des interactions exactes entraîne alors le Shuffle de l'ensemble des nœuds. Le recours à une étape de combinaison ne permet pas de diminuer le volume de données transmises.

Pour contourner l'évaluation des interactions proches, il faut s'assurer que les ensembles bien séparés contiennent peu de nœuds. Pour cela, il faut une grille très fine.

Précision de la grille régulière Pour une répartition uniforme des nœuds, si l'on veut au plus un nœud par cellule de la grille, il faut utiliser une grille régulière de coté $\sqrt{|V|}$ ou plus. Pour un million de nœuds, cela représente donc une grille de coté 1000.

En reprenant l'exemple de l'évaluation des interactions éloignées, *cf.* Section 4.3.5, cela représente alors un shuffle de $50 \cdot 1000^2 \cdot 10 \cdot 4 = 2$ Go de données. Le calcul des interactions éloignées est, de plus, complexe sur une seule machine.

Dimensionnement de la grille On peut trouver un compromis entre le nombre d'interactions éloignées oubliées et le poids du transfert lors de l'étape de Reduce en exprimant sune fonction de coût. Le nombre d'interactions proches, pour une distribution uniforme des nœuds, s'exprime en fonction du nombre de nœuds par cellule $\frac{|V|}{k}$. La constante c_1 représente la pénalité appliquée par interaction proche non prise en compte.

$$C_1 = 9 \cdot c_1 \cdot \frac{|V|}{k}$$

Le poids des données à transférer est fonction linéaire du nombre de cellules de la grille. On peut de plus exprimer le nombre de mappers comme une fraction c_3 de l'ensemble des nœuds. La constante c_2 correspond à la pénalité appliquée par octet. C_2 s'exprime alors comme :

$$\begin{aligned} C_2 &= 4 \cdot c_2 \cdot M \cdot p \cdot k \\ &= 4 \cdot c_2 \cdot \frac{|V|}{c_3} \cdot p \cdot k \end{aligned}$$

Pour une fonction de coût totale de :

$$\begin{aligned} C(k) &= C_1(k) + C_2(k) \\ &= 9 \cdot c_1 \cdot \frac{|V|}{k} + 4 \cdot c_2 \cdot p \cdot \frac{|V|}{c_3} \cdot k \end{aligned}$$

Par dérivation, on obtient l'expression de k au minimum de la fonction de coût.

$$k = \sqrt{\frac{9 \cdot c_1 \cdot c_3}{4 \cdot p \cdot c_2}}$$

Application numérique Soit G un graphe d'un million de nœuds. Le coefficient de précision p de l'approximation des interactions éloignées est fixé à 10. La pénalité c_1 appliquée par interaction proche non prise en compte est fixée à 3. la pénalité par octet c_2 est fixée à 1 par 10 Mo soit donc 10^{-7}. Enfin, chaque mapper traite $c_3 = 20\,000$ nœuds.

Avec ces valeurs, on trouve une grille constituée du nombre de cellules suivant :

$$\begin{aligned} k &= \sqrt{\frac{9 \cdot 3 \cdot 20000}{4 \cdot 10 \cdot 10^{-7}}} \\ &\approx 367\,424.5 \end{aligned}$$

Pour une grille régulière, le côté doit être de 607 ou plus. Il y a alors $\frac{9 \cdot 10^6}{367\,424} \approx 25$ nœuds non pris en compte dans le calcul des interactions proches et $4 \cdot 10^{-7} \cdot 20\,000 \cdot 10 \cdot 367\,424 \approx 735$ Mo transmis sur le réseau.

Conclusions

Compromis Le calcul des interactions éloignées utilisé dans la méthode multipôle est transposable dans le paradigme MapReduce. Cependant, les interactions entre particules proches sont difficiles à évaluer. En utilisant une grille plus fine, il est possible de contourner ce problème au prix de la génération de très nombreux développements limités locaux, et donc d'une étape de shuffle conséquente.

Implémentation La méthode multipôle rapide est connue pour sa complexité linéaire pour le calcul des interactions entre toutes les paires de nœuds. Du point de vue de l'implémentation, les constantes liés à la combinaison des

	Description	Statut
Méthode de résolution		
Newton-Raphson	Optimisation simultanée de plusieurs nœuds.	Difficile
Descente stochastique avec retard	Optimisation simultanée sur plusieurs machines des positions d'un sous-ensemble de nœuds.	Non exploré
Application de forces	Application des forces à tous les nœuds simultanément.	Implémenté
Interactions		
Forces d'attraction	Calcul de $\|E\|$ interactions.	Implémenté
Force de répulsion exactes	Calcul de $\|V\|^2$ interactions.	Difficile
Approximation par grille	Calcul des interactions en utilisant une grille : $O(\|V\|^2)$ interactions.	Difficile
Barnes et Hut	Création d'un arbre contenant $O(\|V\| \log \|V\|)$ nœuds transmis à chaque machine du cluster.	Difficile
Arbre à profondeur fixe	Création d'un arbre contenant un nombre fixe de nœud transmis à chaque machine du cluster.	Non exploré
Méthode multipôle rapide	Aucune implémentation d'une méthode multipôle dans un environnement MapReduce à notre connaissance.	Non exploré

FIGURE 4.3 – Tableau récapitulatif des techniques de dessin dans le paradigme MapReduce. Lorsque la transposition dans le paradigme MapReduce ne permet pas d'assurer les performances, la ligne apparaît en gris foncé. Lorsque la transposition est possible mais n'a pas été explorée, la ligne apparaît en gris.

développements limités locaux constituent une grande partie des performances de l'algorithme. Le nombre de développements limités à traiter avec une grille fine est très important et peut considérablement ralentir l'approximation des forces.

Distribution des nœuds Notre présentation rapide de l'algorithme n'évoque pas non plus le problème de répartition des nœuds dans le plan. Lors des étapes du dessin, les nœuds ne se trouvent pas uniformément répartis dans le plan mais plutôt groupés par clusters représentant la topologie du graphe. Il est alors nécessaire d'utiliser une grille plus fine ou d'ignorer plus d'interactions proches.

Aluru et al.[3, 4] et Sevilgen et al. [121] proposent de rendre la méthode indépendante de la distribution des nœuds. Cependant, ces méthodes sont basées sur une subdivision de l'espace jusqu'à obtenir une cellule par nœud du graphe. L'étape de Shuffle du calcul des interactions éloignées est alors de taille $|V| \cdot p$, avec p la précision du développement.

Interactions proches Les interactions proches que nous avons ignorées sont à priori celles qui contribuent le plus à la force totale appliquée sur chaque nœud. Dans l'exemple des forces de répulsion de Fruchterman et Reingold [48], l'intensité des forces est inversement proportionnelle à la distance entre les nœuds. Ce sont alors les nœuds proches qui contribuent en grande partie à la force de répulsion totale.

4.4 Conclusion

Dans l'état de l'art, *cf.* Section 3, nous avons fait la liste des méthodes de résolution et d'approximation des forces de répulsion. Toutes ces techniques ne peuvent pas être utilisées dans le paradigme MapReduce, où le partitionnement des données joue un rôle crucial et où chaque communication est coûteuse. Les Sections 4.2 et 4.3 évaluent si chacune de ces techniques peut être transposée dans le paradigme MapReduce.

Chapitre 5

GDAD : Algorithme de dessin de graphe distribué

5.1 Introduction

Nous présentons dans ce chapitre notre première contribution : le premier algorithme de dessin par modèle de force dans le paradigme MapReduce appelé GDAD [75]. Nous décrivons les analogies physiques retenues pour le dessin de graphe dans le paradigme MapReduce. La complexité de l'algorithme ainsi que la preuve de sa scalabilité horizontale sont présentées. Arleo et al [8] présentent un algorithme de dessin de graphe distribué. Cet algorithme utilise une analogie physique différente de celle présentée dans GDAD [75].

5.2 Structures de données pour les forces de répulsion approchées

Les forces de répulsion bloquent la transposition des algorithmes de dessin par modèle de forces dans le paradigme MapReduce. Pour rendre scalable un algorithme implémenté dans ce paradigme, il faut réduire le nombre d'interactions à calculer.

Deux algorithmes sont proposés en ce sens en 2015, notre contribution appelée GDAD [75] puis l'algorithme Clint d'Arleo et al. [8]. Les deux approches utilisent des méthodes différentes pour calculer les forces de répulsion.

Dans GDAD, les forces de répulsion sont calculées entre les nœuds et des pivots qui représentent une partition des nœuds, *cf.* Section 5.2.1.

Les pivots obtenus sont dynamiques : leur nombre et leur position sont régulièrement mises à jour de manière à suivre l'évolution de la structure du dessin. GDAD utilise deux heuristiques pour mettre à jour ces pivots, *cf.* Section 5.2.2.

Nous évoquons finalement les travaux d'Arleo et al. [8] dans la Section 5.2.3.

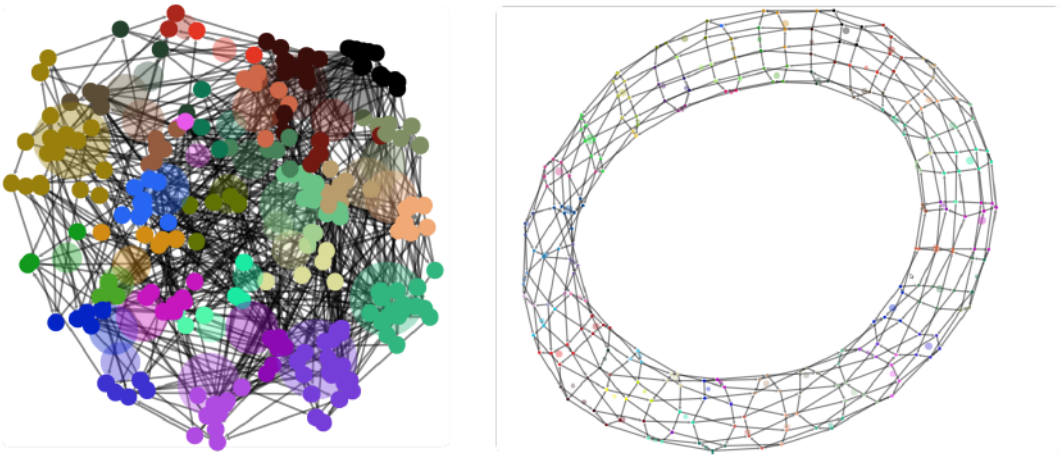


FIGURE 5.1 – Dessin d'un graphe torique (25 anneaux de 10 nœuds, soit 250 nœuds). *(Gauche)* Initialisation du dessin (quelques itérations). Les centroïdes représentent plusieurs nœuds qui ne sont pas nécessairement voisins dans le graphe. *(Droite)* Dessin final. Les centroïdes représentent des clusters de nœuds voisins. Au cœur du tore, il n'y a ni nœud ni pivot.

5.2.1 Forces approchées par pivots

Description des pivots

Pour obtenir les pivots, une méthode de clustering est utilisée. Les clusters sont formés en utilisant uniquement la position des nœuds. En plus de leur position, on associe deux métriques à chaque pivot. La masse d'un pivot représente le nombre de nœuds dans le cluster lui étant associé. Le rayon représente la distance maximale entre un pivot et un nœud de son cluster.

La structure du graphe doit être reflétée par la disposition des nœuds dans le dessin. Utiliser des pivots pour calculer des forces de répulsion approchées est plus pertinent que d'utiliser une grille, dont la décomposition ne tient pas compte de la topologie du graphe. Les zones du dessin contenant de nombreux nœuds sont représentées par plusieurs pivots juxtaposés. Au contraire, si le dessin ne contient aucun nœud dans une zone, il n'y a aucun pivot dans cette zone. L'utilisation de l'approche par pivots assure donc que les forces approchées suivent au plus près la forme du dessin, là où une décomposition spatiale récursive n'en tient pas compte, *cf.* le dessin d'un graphe torique en Figure 5.1.

Calcul des forces de répulsion

Les forces sont calculées entre les nœuds et les pivots. La scalabilité horizontale est ainsi assurée quelle que soit la répartition des nœuds dans le plan. Chaque nœud nécessite uniquement la position de n pivots pour effectuer le calcul des forces de répulsion. Sur un cluster, les n pivots peuvent être trans-

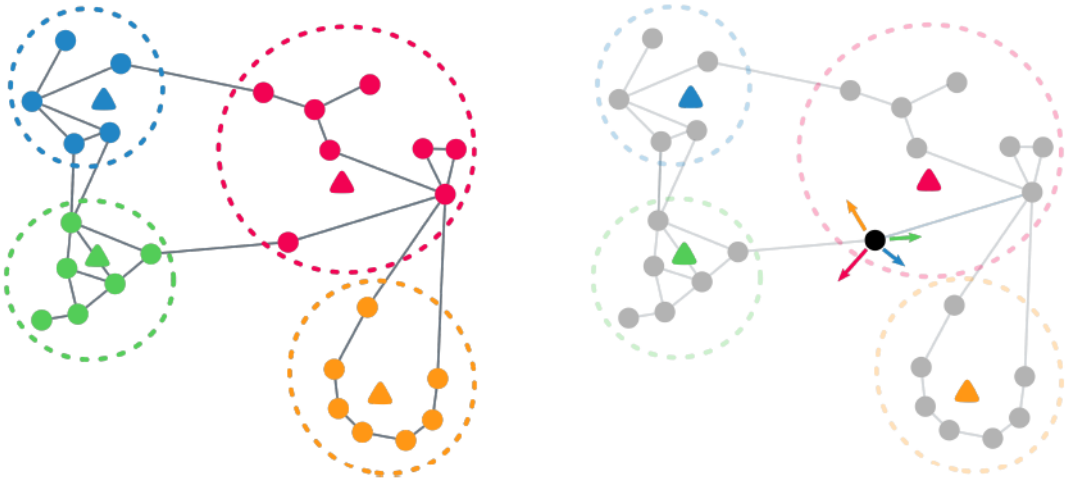


FIGURE 5.2 – *(Gauche)* Clustering des nœuds du graphe à partir de leur position dans le dessin. Les clusters formés sont représentés par un pivot. *(Droite)* Calcul des forces de répulsion entre un nœud et les pivots. Les forces de répulsion s'appliquent à chaque nœud et l'ensemble des pivots.

mis à toutes les machines. Le nombre d'interactions calculées est alors réduit à $O(n \cdot |V|)$.

Le calcul des forces de répulsion pour un nœud est schématisé en Figure 5.2.

5.2.2 Heuristiques pour des pivots dynamiques

Les forces de répulsion approchées permettent d'assurer une bonne répartition des clusters de nœuds dans le plan. Pour cela, il est nécessaire que les pivots soient un bon résumé de la structure du graphe. Afin d'éviter les problèmes de sous ou surapprentissage, nous avons mis en place deux heuristiques nous permettant de réguler ces comportements.

La **duplication** de pivots permet de mieux coller aux données et d'éviter le sous-apprentissage. La **suppression** d'un pivot permet d'éviter d'être trop spécifique et évite ainsi le surapprentissage.

Les heuristiques utilisent des critères pour déclencher la duplication ou la suppression d'un pivot. Dans notre approche, ces critères reposent uniquement sur la masse des pivots.

Duplication des pivots

Lors de la duplication, un nouveau pivot est créé. Ce pivot est placé près du pivot original dans une direction aléatoire. En plaçant ce pivot à une distance inférieure à la longueur d'arête dans le dessin, le dédoublement du cluster est assuré. L'étape d'expansion est illustrée en Figure 5.3.

La duplication est déclenchée si un pivot représente trop de nœuds.

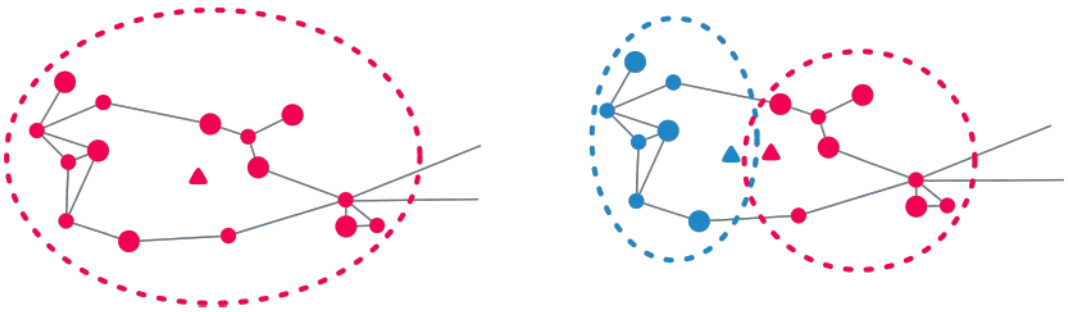


FIGURE 5.3 – Expansion d'un pivot. *(Gauche)* Le pivot est dupliqué *(Droite)* pour former deux pivots proches. Le cluster qu'il représentait contenait trop de nœuds.

Définition 5.1 (Masse critique maximale d'un pivot)**.** Soient $k_{max} \in]0,1]$ et $p = (p_x, p_y) \in \mathbb{R}^2$ de masse m. Soit $M = |V|$ la masse totale des nœuds du graphe. La duplication de p est déclenchée lorsque $m > k_{max} \cdot M$.

Propriété 5.1 (Nombre maximum de pivots)**.** Lorsque tous les pivots respectent le critère de masse critique (*cf.* Définition 5.1) il y a au minimum $\left\lceil \frac{1}{k_{max}} \right\rceil$ pivots dans le dessin.

Démonstration. Soit $k_{max} \in]0,1]$ le paramètre de masse critique. Soit $n_p \in \mathbb{N}$ le nombre de pivots dans le graphe. Soit $\{m_i | 1 \leq i \leq n_p\}$ l'ensemble des masses des pivots du dessin. On note $M = |V|$ la masse totale des nœuds du graphe.

Par principe des tiroirs, si chaque nœud respecte la contrainte 5.1, la masse maximale des pivots permet d'obtenir le nombre minimum de pivots. La somme des masses des pivots s'écrit :

$$\begin{aligned} \sum_{1 \leq i \leq n_p} m_i = M &\leq k_{max} \cdot \sum_{1 \leq i \leq n_p} M \\ &\leq k_{max} \cdot M \cdot n_p \end{aligned}$$

On a donc $\frac{1}{k_{max}} \leq n_p$ et n_p est entier. Le nombre minimal de pivots est donc $\left\lceil \frac{1}{k_{max}} \right\rceil$.

□

Suppression des pivots

Lorsqu'un pivot représente une zone trop restreinte du dessin, il est supprimé. De cette manière, les nœuds qui lui étaient associés sont redistribués dans d'autres clusters. Grâce à cette suppression, nous cherchons à éviter le surapprentissage et à limiter le nombre d'interactions calculées. L'étape de suppression d'un pivot est illustrée en Figure 5.4.

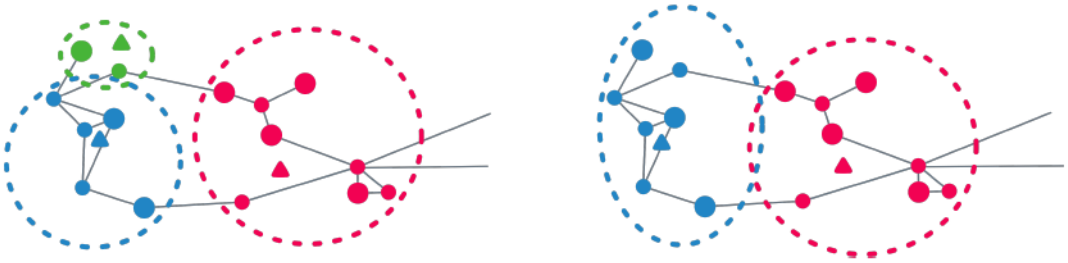


FIGURE 5.4 – Suppression d'un pivot. *(Gauche)* Le pivot vert (en haut du dessin de graphe) forme un cluster de seulement deux nœuds. Il est supprimé de manière à créer deux pivots représentatifs.

La suppression des pivots est déclenchée par l'existence d'une masse critique minimale.

Définition 5.2 (Masse critique minimale d'un pivot)**.** Soient $k_{min} \in]0, 1]$ et $p = (p_x, p_y) \in \mathbb{R}^2$ de masse m. Soit $M = |V|$ la masse totale des nœuds du graphe. La suppression de p est déclenchée lorsque $m < k_{min} \cdot M$.

Masse critique minimale On peut déduire du premier critère un nombre maximum de pivots, *cf.* Propriété 5.1.

Propriété 5.2 (Nombre maximum de pivots)**.** Lorsque tous les pivots respectent le critère de masse critique minimale (*cf.* Définition 5.2) il y a au maximum $\left\lfloor \frac{1}{k_{min}} \right\rfloor$ pivots dans le dessin.

Démonstration. Voir démonstration de la Propriété 5.1. □

Nombre d'interactions et complexité

Les critères de masses critiques (minimale et maximale) sur les pivots impliquent que le nombre de pivots est borné. Avec ces critères, le nombre de forces de répulsion calculées à chaque itération est réduite à $O(|V|)$. De plus, il y a $O(|E|)$ forces d'attraction calculées à chaque itération, *cf.* Section 4.3.1. L'application des forces est effectuée en une opération par nœud, pour une complexité de $O(|V|)$.

5.2.3 Forces de répulsion des voisins topologiques

Dans Clint [8], le calcul des forces de répulsion est effectué par une approximation basée sur le voisinage topologique de chaque nœud. Clint utilise Pregel pour calculer des forces et les appliquer à chaque nœud du graphe.

L'analyse proposée par Arleo et al souligne trois contraintes majeures pour le dessin de graphe dans un environnement distribué : chaque nœud ne connaît

FIGURE 5.5 – Deux étapes de flooding contrôlé. *(Gauche)* Initialisation de la technique pour deux nœuds sélectionnés. *(Milieu)* Première ronde de flooding. La position des nœuds sélectionnés est transmise à leurs voisins. *(Droite)* Deuxième ronde de flooding. Les nœuds ayant reçu des messages les transmettent à leur voisins.

que la position de ses voisins, l'information stockée par chaque sommet est limitée, et le nombre total de messages transmis doit être peu élevé (*i.e.* proportionnel au nombre d'arêtes). La troisième contrainte rend impossible de connaître la position de chacun des autres nœuds du graphe.

Flooding contrôlé

La solution proposée tient compte de deux facteurs souvent présents dans les algorithmes de dessin de graphe par modèle de force : la distance dans le graphe est proche de la distance dans le plan du dessin, les forces de répulsion entre nœuds éloignés comptent peu par rapport aux forces de répulsion entre nœuds proches. De ces deux contraintes, la localité des forces émerge comme un point saillant : au lieu de calculer l'ensemble des forces de répulsion, celles appliquées par les sommets dans un k-voisinage (*i.e.* les sommets dont la distance est au plus k) est une solution pour réduire la complexité.

Pour pouvoir effectuer le calcul des forces de répulsion avec son k-voisinage, chaque nœud nécessite la position des nœuds dans ce voisinage. Ceci est réalisable en utilisant Pregel pour transmettre la position des nœuds en suivant les arêtes. En k rondes, chaque nœud obtient la position de ses voisins à distance graphe k ou moins par une technique de routage appelée le flooding (*cf.* Lim [98] par exemple). Cette technique consiste à transmettre un message à chaque voisin, qui le transmet à son tour à ses voisins. Ainsi, à l'étape i, chaque nœud connaît l'ensemble de ses voisins à distance i. A l'aide de cette technique de routage, les nœuds du graphe connaissent la position des nœuds dans leur voisinage proche et peuvent calculer les forces de répulsion approchées.

La technique de flooding contrôlée est illustrée pour deux nœuds en Figure 5.5.

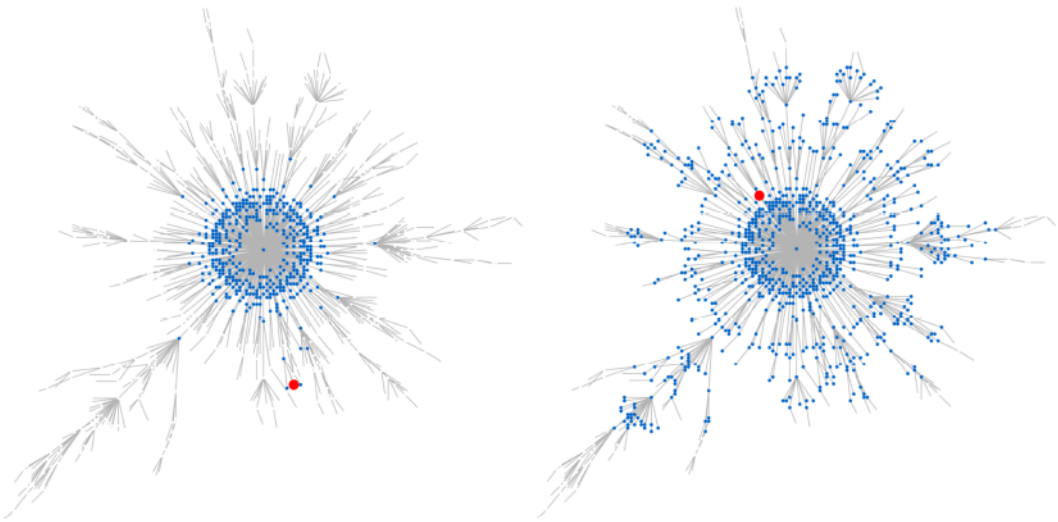


FIGURE 5.6 – Graphe de 1000 nœuds suivant une loi de puissance généré par attachement préférentiel [14]. Pour deux nœuds sélectionnés au hasard (en rouge), le voisinage à distance graphe 3 est représenté (en bleu). *(Gauche)* Le voisinage contient 407 nœuds. *(Droite)* Le voisinage contient 870 nœuds.

Influence de la distance de flooding

La distance de flooding k est le paramètre permettant de régler la précision de l'approximation de l'algorithme. Pour $k \geq d_G$, avec d_G le diamètre du graphe, le calcul des forces de répulsion est effectué entre tous les nœuds du graphe et correspond donc au calcul exact des forces approchées. Pour $k = 1$, le calcul des forces est effectué uniquement avec les voisins et revient à appliquer une fonction différente pour les forces d'attraction tenant en compte un élément de répulsion. Dans l'étude d'Arleo et al. [8], le paramètre k est donc principalement considéré dans la plage $2 \leq k \leq d_G$. Selon la topologie du graphe, différents paramètres peuvent être plus ou moins adaptés à l'étude : pour des graphes denses, $k = 2$ donne une bonne approximation des forces de répulsion alors que pour des graphes creux, $k \geq 3$ permet d'améliorer l'approximation.

La complexité de l'algorithme dépend aussi de ce paramètre k. Pour un graphe de degré maximum Δ, le nombre d'opérations maximum est de $O(\Delta^k)$. En effet, pour un nœud donné, il possède au plus Δ voisins qui possèdent au plus Δ voisins. Dans le pire des cas, si tous les nœuds sont de degré maximum et qu'aucun voisin n'est commun à plusieurs nœuds, il y a Δ^k voisins à distance k ou moins.

Limites de l'algorithme

Dessin de graphe petit monde Le nombre d'interactions retenues par sélection à l'aide d'un voisinage topologique repose sur deux facteurs : la distance graphe k à laquelle les interactions sont retenues et le degré maximum Δ déterminant la complexité dans le pire des cas.

Lorsque le degré du graphe Δ n'est pas borné, par exemple lorsque le degré suit une loi de puissance, la complexité du nombre d'interactions dans le pire des cas n'est plus valable. En effet, le nœud de degré le plus élevé peut être de l'ordre de $|V|$ nœuds, ce qui signifie qu'il y a de l'ordre de $|V|^2$ interactions sélectionnées. Cela revient à effectuer le calcul exact, ce qui n'est pas possible (*cf.* Section 4.3.2).

Ce modèle peut donc vite atteindre ses limites dans le cas du dessin de graphes réels, et notamment des graphes petits mondes. La définition de ces graphes n'est pas standardisée mais ils partagent quelques caractéristiques, dont notamment un petit diamètre et une distance graphe moyenne entre les nœuds faible. L'exemple couramment utilisé est celui des cinq degrés de séparation : en moyenne, il est possible d'atteindre n'importe quel autre nœud à l'aide de cinq transitions. En sélectionnant un paramètre $k > 5$ pour ce type de graphe, on sélectionne presque l'ensemble du graphe et il y a donc $|V|^2$ interactions à calculer.

Ce phénomène est illustré en Figure 5.6. Dans cet exemple, le graphe est généré selon l'algorithme aléatoire d'attachement préférentiel. Le graphe généré contient 1000 nœuds. Le facteur de la loi de puissance est fixé à $\frac{6}{5}$. Deux nœuds sont sélectionnés au hasard et leur voisinage pour $k = 3$ est déterminé. Dans le premier cas, ce voisinage comprend 407 nœuds. Dans le deuxième cas, ce voisinage comprend 870 nœuds. Calculer les forces dans le voisinage de ces nœuds revient donc presque à calculer l'ensemble des interactions avec les nœuds du graphe.

Structure globale du graphe Dans le chapitre d'état de l'art, Section 3.3.1, nous évoquons les techniques de sélection des interactions développées par Koren et al. [51, 67, 90]. La technique utilisée par Arleo et al. [8] repose sur les k-voisinages associés au modèle de Fruchterman et Reingold. Harel et Koren [67] proposent la même sélection des interactions dans le modèle de Kamada et Kawai.

Cependant, Koren et al. utilisent des pivots [51, 90] et une approche multiéchelle [67] de manière à obtenir la structure globale du graphe. Arleo et al. ne calculent pas d'autres interactions que celles avec les nœuds du k-voisinages, ce qui permet d'obtenir un dessin localement optimal sans garantie sur la structure globale.

Algorithme	Pivot	Voisinage
Paramètre	Nombre de centroïdes c	Distance du voisinage k
Cas d'égalité	$c = \|V\|$	$k \geq d_G$
Complexité	$O(c \cdot \|V\|)$	$O(\Delta^k \cdot \|V\|)$

FIGURE 5.7 – Calcul de forces approchées : Comparaison des approches par centroïde et par voisinage. On note d_G le diamètre du graphe et Δ le degré maximum du graphe.

5.3 GDAD : algorithme de dessin par centroïdes

Dans cette section, nous présentons GDAD, notre algorithme de dessin dans le paradigme MapReduce [75]. L'Algorithme 5 présente chaque étape en vis-à-vis avec le nombre d'opérations Map et Reduce (ou ronde de Pregel) requises.

Algorithm 5 Algorithme de dessin par centroïdes

```
Init Initialisation des centroïdes
while Condition d'arrêt do
    Mise à jour des centroïdes
    Expansion et suppression des centroïdes          ▷ Map : 0 Reduce : 0
    Calcul de la position des centroïdes             ▷ Map : 1 Reduce : 1
    Partage des centroïdes                           ▷ Broadcast : 1
    Calcul des forces
    Calcul des forces d'attraction                   ▷ Pregel : 1
    Calcul des forces de répulsion                   ▷ Map : 1
    Déplacement des nœuds                            ▷ Map : 1
end while
```

Les centroïdes sont le type de pivot que nous utilisons dans GDAD. Ces centroïdes sont le barycentre de clusters de nœuds, *cf.* Section 5.3.2. Cette approximation des forces est inspiré par Frick et al. [46] qui propose une force de répulsion centrale dans l'algorithme GEM.

Nous présentons ensuite un algorithme pour obtenir les k-means dans le paradigme MapReduce, *cf.* Section 5.3.3.

Enfin, nous détaillons le modèle de forces utilisé dans l'algorithme GDAD, *cf.* Section 5.3.4.

5.3.1 Bornes du nombre de pivots

Notre algorithme utilise les forces de répulsion approchées par pivots, présentées en Section 5.2. Pour que le nombre de pivots colle au plus près à la

réalité du graphe, nous utilisons des pivots dynamiques, qui sont dupliqués ou supprimés lorsqu'ils représentent trop ou trop peu de nœuds. Cette technique nous permet aussi d'assurer que le nombre de pivots est borné, réduisant alors le nombre d'interactions de répulsion calculées à $O(|V|)$ itérations.

Nous détaillons ici les bornes que nous avons sélectionnées pour le nombre de pivots dans GDAD.

Nombre maximum de pivots

La limite supérieure du nombre de pivots a été déterminée de manière à réduire l'encombrement visuel à l'écran. Chaque pivot est associé à un cluster de nœuds du graphe. Visuellement, la multiplication du nombre d'éléments à l'écran entraîne une difficulté à dissocier ces éléments. Dans GDAD, on souhaite que l'ensemble des clusters puisse être parcouru visuellement en un temps court (inférieur à 10 secondes).

Wolfe [137] combine plusieurs des études menées par son équipe de recherche ophtalmique pour évaluer le temps de réponse d'utilisateurs à la recherche d'un élément ayant des caractéristiques spécifiques parmi un ensemble encombré visuellement. Lorsque l'utilisateur recherche des caractéristiques particulières, le temps de réponse est linéaire et varie entre 20 et 30 ms par élément représenté. Lorsque l'utilisateur n'est pas certain de trouver ces caractéristiques, le temps de recherche est deux fois plus long.

Dans l'objectif de dimensionner le nombre de pivots, nous reprenons ces temps de réponse pour avoir un ordre d'idée du temps de parcours de l'ensemble des pivots. Dans le cadre du dessin de graphe, le parcours de 400 clusters à la recherche d'un cluster déjà identifié par l'utilisateur prendrait en moyenne 10s. Le parcours exploratoire des clusters prendrait, lui, environ 20s.

Dans GDAD, nous avons donc fixé la limite supérieure à 400 pivots. Chaque pivot représente alors au moins 0.25% des nœuds.

Nombre minimum de pivots

Le nombre minimum de pivots permet d'assurer une bonne répartition des clusters dans le plan du dessin. Pour que le dessin converge vers un état stable, une technique parfois utilisée dans les algorithmes de dessin consiste à diviser l'espace autour d'un nœud en trois secteurs angulaires que l'on essaye d'équilibrer. De cette manière, les forces de répulsion forment un tripode qui permet d'assurer la stabilité du nœud.

En transposant cette technique aux clusters de nœuds (et non plus aux nœuds directement), on détermine un nombre minimum de pivots. Un cluster est équilibré s'il reçoit des forces de répulsion provenant de trois secteurs équiangles. On souhaite ainsi s'assurer que chaque cluster reçoit des forces de répulsion de trois directions différentes. On souhaite que 75% des pivots aient au moins un pivot voisin dans chacun des secteurs angulaires.

Pour trouver le nombre de pivots requis, nous effectuons la simulation suivante. n pivots sont répartis aléatoirement dans le sous-plan $[0,1] \times [0,1]$ selon une loi uniforme. Pour chaque pivot, on détermine s'il y a au moins un pivot différent dans chacun des secteurs angulaires $]-\pi, \frac{-\pi}{3}]$, $]\frac{-\pi}{3}, \frac{\pi}{3}]$ et $]\frac{\pi}{3}, \pi]$. Le nombre de pivots respectant cette contrainte nous fournit un ratio.

A partir de 40 pivots, il y a en moyenne 75% des pivots ayant au moins un pivot adjacent dans chaque secteur angulaire. Dans GDAD, nous avons donc fixé la limite inférieure à 40 pivots. Chaque pivot représente alors au plus 2.5% des nœuds.

5.3.2 Pivots et algorithme du k-means

Algorithme du k-means

Pour trouver ces clusters, nous utilisons une variante de l'algorithme des k-means. Cet algorithme de clustering itératif forme k clusters constitués des nœuds les plus proches de chacun des k pivots. Le pivot, associé à un cluster, est placé au barycentre de la position des nœuds du cluster. Nous appelons donc centroïdes les pivots obtenus par l'algorithme des k-means.

Initialisation k éléments sont sélectionnés au hasard pour initialiser l'algorithme. Ils correspondent aux k pivots initiaux.

Itération Chaque itération se décompose en deux étapes : l'association des nœuds à un cluster puis la mise à jour de la position du pivot :

- **Association** Les nœuds les plus proches du pivot sont assignés à son cluster.
- **Mise à jour** La position de chaque pivot est mise à jour en utilisant la position des nœuds appartenant au cluster.

Convergence L'algorithme est itéré jusqu'à converger vers un état stable.

Centroïdes et convergence du dessin

Lors de chaque itération de GDAD, une seule itération de l'algorithme des k-means est effectuée. Pour chaque itération de GDAD, la position des centroïdes est mise à jour. Puis, les forces sont calculées et appliquées. Cette simplification de l'algorithme est motivée par deux raisons :

Performances La convergence de l'algorithme k-means peut être longue. Cependant, les centroïdes sont conservés d'une itération à l'autre. En appliquant une itération des k-means à chaque itération de GDAD, les centroïdes

convergent vers un état d'équilibre au fur et à mesure de l'exécution de l'algorithme. Cette simplification permet d'économiser des étapes coûteuses dans le paradigme MapReduce.

Convergence simultanée Lors des premières itérations, il n'est pas possible d'obtenir un clustering acceptable. En effet, pour initialiser l'algorithme de dessin, les nœuds sont placés à une position aléatoire. Le clustering obtenu par l'algorithme k-means à partir du dessin initial ne renseigne pas sur la structure du graphe. Les forces de répulsion générées lors de ces itérations n'ont donc pas grand sens.

Cependant, au fur et à mesure que le dessin converge, l'algorithme du k-means converge de plus en plus vite. En effet, les centroïdes ont participé à former des clusters de nœuds dans le dessin qui reflètent la structure topologique du graphe. Les forces de répulsion calculées plus tard dans le dessin permettent d'atteindre un état d'équilibre reposant sur la topologie du graphe.

Duplication et suppression des centroïdes

Pour effectuer la duplication ou la suppression de centroïdes, il n'y a pas besoin de connaître la position des nœuds ou d'accéder aux arêtes du graphe. Les critères définis en Section 5.2.2 sont basés sur trois paramètres : la masse du centroïde, le nombre de nœuds du graphe (constant) et la longueur souhaitée pour les arêtes dans le dessin (constante).

Dans le paradigme séquentiel, chaque centroïde est parcouru pour vérifier chaque critère. De plus, le nombre de centroïdes est borné. Cette étape est rapide puisqu'elle ne nécessite pas de lecture des données volumineuses.

Dans le paradigme MapReduce, les centroïdes (au plus 400) sont stockés sur une seule machine. Le nombre de nœuds du graphe et la longueur des arêtes sont deux paramètres constants. L'étape de duplication et suppression des centroïdes est donc réalisée sur une seule machine.

5.3.3 k-means dans le paradigme MapReduce

L'algorithme k-means fonctionne en itérant deux étapes : l'association nœuds-centroïdes et la mise à jour des centroïdes. L'opération Map permet de trouver le centroïde le plus proche de chaque nœud. L'opération de Reduce permet de calculer la nouvelle position de chaque centroïde.

Les étapes de cet algorithme MapReduce sont schématisées en Figure 5.8.

Opération Map

Préalablement à l'opération de Map, les centroïdes sont transmis à l'ensemble des machines par une opération de Broadcast, *cf.* Section 2.3.

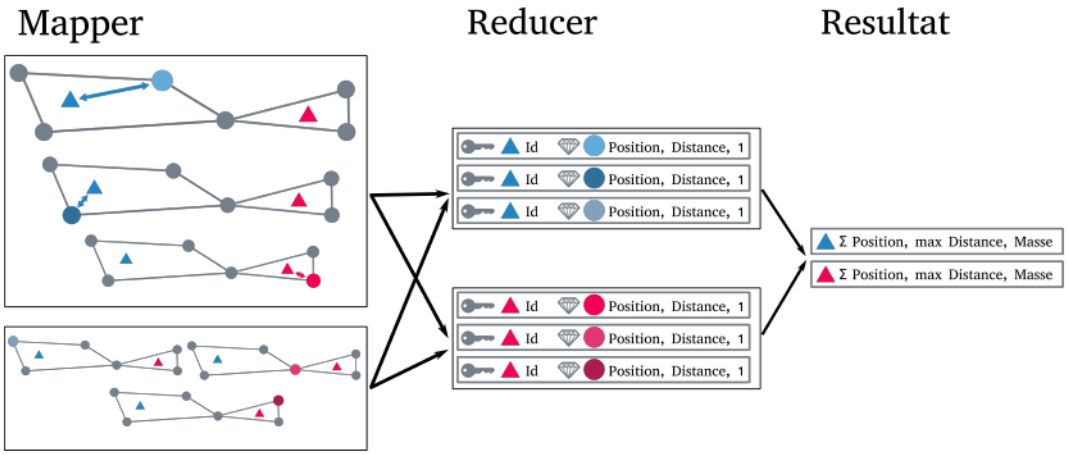


FIGURE 5.8 – Schéma du calcul de la position des centroïdes. *(gauche)* Opération de Map. Recherche du centroïde le plus proche de chaque nœud. *(milieu)* Opération de Reduce. Les informations des nœuds associés à un même centroïde sont envoyés au même Reducer. *(droite)* Résultat obtenu après l'opération de Reduce. Les positions et la masses sont ajoutées pour obtenir la position moyenne.

L'opération de Map s'effectue sur les nœuds du graphe. La distance entre chaque nœud et l'ensemble des centroïdes est calculée de manière à trouver le centroïde le plus proche. Une fois le centroïde le plus proche trouvé, son identifiant est utilisé comme clef d'une paire clef-valeurs. Les valeurs transmises sont les coordonnées du nœud, la distance à son centroïde le plus proche et une variable de masse fixée à 1.

Opération Reduce

Dans l'étape de Reduce, les coordonnées des nœuds associés au même cluster (même clef) sont agrégées pour obtenir les coordonnées totales et la masse.

Les coordonnées et le poids sont ajoutés, suivant ainsi le design pattern du calcul de la moyenne en MapReduce (*cf.* Miner et Shook [106] par exemple). Les coordonnées réelles du centroïde peuvent alors être obtenues en divisant les coordonnées totales par la masse du centroïde.

Une étape de Broadcast, *cf.* Section 2.3, permet de partager le vecteur des positions avec l'ensemble des machines du cluster. La position des centroïdes est notamment utilisée pour le calcul des forces de répulsion.

Seul un petit nombre de messages sont transmis lors du calcul de la position des centroïdes. En utilisant une opération de Combine, les positions des nœuds sont combinées lors de l'opération Map. Il y a alors au plus c messages qui sortent de chaque machine du cluster. Soit M le nombre de machines du cluster. L'unique reducer ne transmet pas la position combinée des centroïdes qu'il a calculée. Il y a alors $c \cdot (M - 1)$ messages transmis vers le reducer.

5.3.4 Modèle de forces

Deux types de forces sont calculées dans notre algorithme : les forces d'attraction selon les arêtes et les forces de répulsion entre chaque nœud et l'ensemble des centroïdes. Nous présentons ici les forces utilisées dans GDAD.

Forces d'attraction

Les forces d'attraction sont appliquées entre chaque paire de nœuds connectés.

Soient u et v deux nœuds voisins du graphe. Soient k_1 une constante, d_{uv} la distance entre u et v dans le dessin et e_{uv} la longueur de l'arête uv. Soit $\mathbf{u}$ le vecteur unitaire directeur de la droite (uv) allant de u vers v.

La force d'attraction que v applique sur u notée $\mathbf{F}_{u \leftarrow v}$ est colinéaire au vecteur u. Son sens est déterminé par la distance d'arête optimale : si la distance entre u et v est plus grande que la longueur d'arête entre u et v, cette force est attractive, sinon elle est répulsive. La force d'attraction a pour intensité $k_1 \cdot (d_{uv} - e_{uv})$. Au final, cette force s'écrit :

$$\mathbf{F}_{u \leftarrow v} = k_1 \cdot (d_{uv} - e_{uv}) \cdot \mathbf{u}$$

Forces de répulsion

Les forces de répulsion sont appliquées entre chaque nœud et l'ensemble des centroïdes calculés dans le graphe.

Soient u un nœud et c un centroïde. Soient k_2 une constante, d_{uc} la distance entre u et c dans le dessin. Soit $\mathbf{u}$ le vecteur unitaire directeur de la droite (uv) allant de u vers c.

La force de répulsion que c applique sur u notée $\mathbf{F}_{u \leftarrow c}$ est colinéaire au vecteur u. La force de répulsion a le sens opposé de u. La force de répulsion a pour intensité $\frac{k_2}{d_{uv}^2}$. Au final cette force s'écrit :

$$\mathbf{F}_{u \leftarrow c} = -\frac{k_2}{d_{uv}^2} \cdot \mathbf{u}$$

Déplacement des nœuds

Une fois l'ensemble des forces calculées pour chaque nœud, les nœuds sont déplacés. Pour chaque nœud, une nouvelle position est calculée en appliquant (*i.e.* en sommant) les forces à la position courante. Les forces d'attraction et de répulsion sont agrégées séparément. Les forces d'attraction et de répulsion sont ajoutées à part égales (la moyenne des deux types de forces) pour calculer la force totale.

Dans le paradigme MapReduce, le déplacement des nœuds correspond à une opération Map sur les nœuds. Dans cette opération, la position de chaque nœud

est sommée avec la force totale (force de répulsion totale et force d'attraction totale).

5.3.5 Complexité de GDAD

Une itération de GDAD (calcul et application des forces) a donc pour complexité $O(|V|+|E|)$. Il est généralement accepté que pour atteindre la convergence, $O(|V|)$ itérations de l'algorithme par modèle de force sont nécessaires. La complexité totale est alors de $O(|V| \cdot (|V|+|E|))$.

En général, le nombre d'arêtes $|E|$ est un multiple du nombre de nœuds. On écrit alors $|E| = k \cdot |V|$ où le facteur k est négligeable par rapport au nombre de nœuds $|V|$. La complexité totale de notre algorithme est alors de $O(|V|^2)$.

5.4 Implémentation du dessin par centroïdes

L'implémentation de notre algorithme de dessin par centroïdes a été réalisée avec Spark dans un environnement distribué et en C++ dans un environnement séquentiel. Nous donnons ici les détails de ces deux implémentations.

5.4.1 Généralités sur Spark

L'environnement distribué Spark [139] dispose de nombreuses fonctionnalités qui facilitent le développement de l'algorithme par centroïdes.

Jeux de données en mémoire

En premier lieu, cet environnement permet d'enchaîner les opérations MapReduce. Dans l'implémentation Hadoop de MapReduce, le jeu de données est écrit sur disque après chaque opération Reduce. Cette limitation rend compliqué les algorithmes itératifs, comme le sont par exemple les algorithmes de dessin par modèle de force. Le jeu de donnés est donc lu et écrit plusieurs fois sur disque pour effectuer une itération et plusieurs itérations sont requises pour arriver à un état de convergence.

Spark permet de contourner cette contrainte en conservant les résultats des opérations sous la forme de RDD (Resilient Distributed Datasets). Ces RDD sont des tables de données immuables en lecture seule. Elles sont conservées en mémoire et peuvent être reconstruites en cas de panne grâce à un suivi des opérations.

Approche graphe et approche géométrique

Notre algorithme mêle les algorithmes orientés graphes (calcul des forces d'attraction selon les arêtes par exemple) avec des algorithmes d'analyse de

données ligne par ligne (calcul des k-means par exemple). Spark permet de passer du paradigme MapReduce au paradigme Pregel, grâce à sa bibliothèque de graphe nommée GraphX [138].

5.4.2 Calcul et application des forces

Nous décrivons dans cette section l'implémentation du calcul des forces et le déplacement des nœuds, dont l'algorithme a été présenté en Section 5.3.4. Une fois la liste des centroïdes obtenus, le calcul et l'application des forces sont réalisés à l'aide de Pregel.

Structure du graphe

GraphX [138], la bibliothèque Spark orientée graphe, propose une implémentation de Pregel. Dans GraphX, un graphe est décrit à l'aide de deux RDD (Resilient Distributed Dataset) l'un pour les arêtes et l'autre pour les nœuds. La RDD contenant les arêtes contient le couple de nœuds source et destination, identifiés par des *Long*. La RDD contenant les nœuds contient pour chaque nœud un identifiant (*Long*) et une position (paire de *Double*).

Les RDD sont des structures distribuées accessibles en lecture seule : une nouvelle RDD est recréée après chaque transformation du jeu de données. Après chaque mise à jour de la position des nœuds, une nouvelle RDD contenant la position mise à jour des nœuds est créée et le graphe est mis à jour.

Calcul des forces

L'implémentation de Pregel proposée par GraphX prend trois fonctions en entrée et retourne un graphe dont les sommets ont été mis à jour. La première modifie l'état des nœuds en fonction des messages reçus. La deuxième permet d'envoyer les messages. La troisième fonction combine les messages à destination d'un même nœud.

Implémentation Pregel Lors de l'envoi des messages, chaque arête est visitée de manière à calculer la force d'attraction associée. Les positions des extrémités de l'arête sont utilisées pour réaliser le calcul des forces d'attraction. Ces forces sont ensuite envoyées aux deux extrémités (l'une opposée à l'autre). Les messages à destination d'un même nœud sont combinés de manière à calculer la force totale (*i.e.* la somme des forces).

Une fois que les messages ont été transmis et combinés, l'état des nœuds (ici leur position) est mis à jour. Les messages reçus lors de cette étape, *i.e.* les forces d'attraction, sont utilisés pour calculer le déplacement des nœuds. Le calcul des forces de répulsion et de déplacement des nœuds est aussi réalisé dans cette étape que l'on peut considérer similiaire à une opération Map. En effet, pour ces deux étapes, seule une opération Map est nécessaire, *cf.* Section 5.3.4.

Arêtes doubles GraphX stocke les arêtes sous la forme d'arcs (arêtes orientées). Le format de stockage des graphes peut conduire à construire un graphe dont les arêtes entre deux sommets sont dupliquées. Dans le cas du dessin de graphes, les arcs connectant la même source et la même destination sont généralement fusionnés de manière à éviter de renforcer la force d'attraction (appliquée plusieurs fois).

Il est possible de représenter les graphes non-orientés de deux manières dans GraphX : en dédoublant les arêtes sous la forme de deux arcs de sens opposés ou sous la forme d'un arc simple (généralement suivant un ordre croissant des identifiants). Le calcul des forces d'attraction se réalise alors en utilisant le sens de l'arête (respectivement en calculant les interactions pour les deux extrémités).

Dans le cadre de notre algorithme de dessin, ces deux approches ont été implémentées. Nous n'avons cependant pas relevé de différence de performance notable entre nos deux implémentations. Par souci de limiter l'espace requis par le stockage des arêtes, nous avons donc opté pour la représentation à l'aide des arcs simples.

Partitionnement du graphe

Le partitionnement du graphe joue un rôle important sur les performances de l'algorithme : il détermine la quantité de données transmise entre les machines du cluster et peut créer des latences dans l'exécution de l'algorithme.

Fonctionnement du partitionnement L'implémentation Pregel de GraphX [138] repose sur la duplication des nœuds (à l'opposé de la duplication des arêtes). Les arêtes et les nœuds (et leurs attributs) sont partitionnés. GraphX gère aussi une RDD associant chaque nœud avec les partitions d'arêtes où le nœud est présent.

GraphX repose sur la création rapide de triplets pour effectuer les calculs dans le graphe. Ces triplets associent à chaque arête les attributs des extrémités. Grâce aux trois RDD représentant le graphe, les triplets sont reconstruits rapidement par une opération de jointure.

Le partitionnement est conservé lors de la modification du graphe. Lorsqu'une application modifie uniquement les attributs des nœuds ou des arêtes, le partitionnement est conservé, ainsi que la table de routage.

Minimiser les données transitant sur le réseaux revient à un problème d'optimisation : maximiser le nombre d'arêtes partageant une extrémité présente sur une même partition en minimisant les arêtes transmettant des messages vers d'autres partitions.

Notre algorithme de dessin se place dans ce contexte :

- seule la position des nœuds (attributs des nœuds) est modifiée,

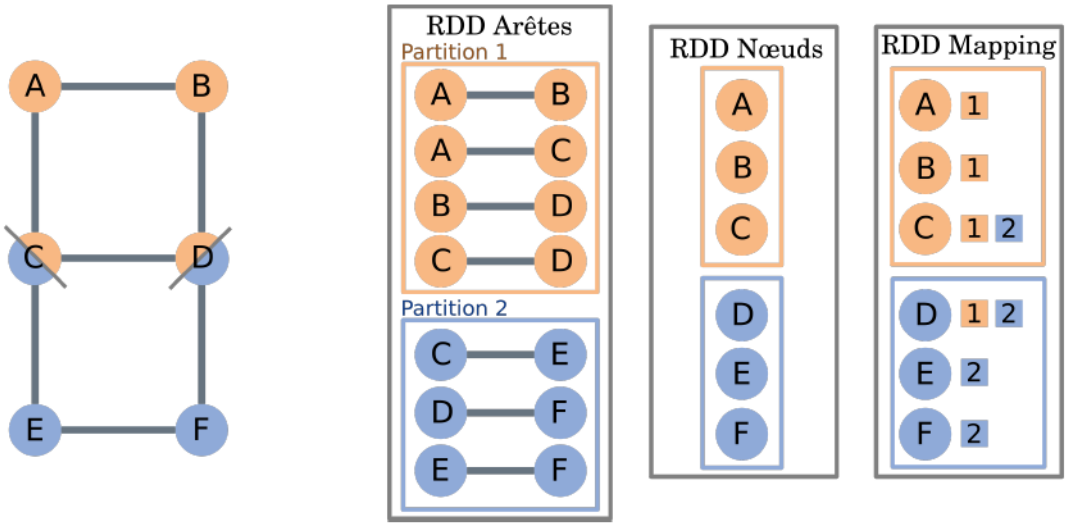


FIGURE 5.9 – Partitionnement des graphes sous GraphX. *(Gauche)* Partitionnement des arêtes par la méthode Vertex-Cut, dupliquant les sommets. *(Droite)* Trois RDD représentant le graphe dans GraphX. Les arêtes, les nœuds et leurs attributs sont partitionnés selon une stratégie, *cf.* Section 5.4.2. La table de Mapping duplique les nœuds. Elle indique les partitions dont le nœud est l'une des extrémités.

- le partitionnement cherche à minimiser le nombre de forces d'attraction transmises sur le réseau.

Stratégie de partitionnement GraphX [138] propose quatre stratégies de partitionnement, accessible via la classe **PartitionStrategy**. Ces stratégies sont appliquées au graphe via la méthode *partitionBy* de la classe **Graph**.

RandomVertexCut partitionne les arêtes via une fonction de hashage prenant en compte les identifiants de la source et de la destination. Les arêtes ayant la même source et la même destination sont localisées sur la même partition. Dans notre implémentation, il n'existe pas de telle arête. Cette stratégie revient donc à un partitionnement aléatoire des arêtes.

CanonicalRandomVertexCut partitionne les arêtes via la même fonction de hashage que la stratégie *RandomVertexCut* mais les identifiants des extrémités sont ordonnés par ordre croissant. Les arêtes ayant les mêmes extrémités sont alors localisées sur la même partition. Dans notre implémentation, les arêtes sont déjà rangées dans cet ordre, 5.4.2, et cette stratégie est donc identique à la précédente.

EdgePartition1D utilise uniquement l'identifiant de la source pour déterminer le partitionnement. Les arêtes ayant la même source sont donc localisées sur la même partition.

EdgePartition2D partitionne les arêtes en utilisant un partitionnement 2D

de la matrice d'adjacence qui garantit une borne de $2 \cdot \sqrt{p}$ réplication pour les nœuds, où p est le nombre de partitions.

Comparaison des stratégies Nous avons comparé les performances de ces trois stratégies pour deux grands graphes, crack et finan512. Les temps présentés en Figure 5.10 correspondent au temps médian d'une itération de GDAD [75] pour chacune des trois stratégies proposée par GraphX en fonction du nombre de machines lancées lors de l'exécution.

Les temps des trois méthodes sont comparables pour les deux graphes sélectionnés. Dans les deux cas, il semble que la stratégie de partitionnement *EdgePartition2D* soit la plus efficace (temps minimum obtenu). Le partitionnement aléatoire (*RandomVertexCut* et *CanonicalRandomVertexCut*) offre des performances proches mais globalement moins bonnes que les méthodes de partitionnement *EdgePartition1D* et *EdgePartition2D*. Le bénéfice d'une stratégie par rapport aux autres ne semble pas être énorme pour notre algorithme.

Dans leur comparaison de GraphX [138] et Giraph [12], Kabiljo et al. [83] privilégient la stratégie *EdgePartition2D* avec laquelle leur performance est améliorée par un facteur de $2,5$. Nos expérimentations semblent confirmer que cette stratégie offre de meilleures performances mais le gain obtenu pour notre algorithme est moindre.

5.4.3 Mise à jour des centroïdes

Partage des centroïdes

Le partage des centroïde en Broadcast, *cf.* Section 2.3, utilise le **SparkContext** pour transmettre les centroïdes à chaque machine du cluster. Ce partage est réalisé comme étape préparatoire au calcul des forces de répulsion (en une opération Map).

Un centroïde est encodé en utilisant 5 variables : un identifiant (*Long*), deux coordonnées (*Double*), la masse du centroïde (*Double*) et son rayon (*Double*). En Scala, les types *Long* et *Double* sont encodés sur 64 bits. Un centroïde a donc un poids de 40 octets. Il y a au plus 400 centroïdes dans le dessin. Le poids maximum des données à transmettre est donc de 16 ko lors d'une itération donnée.

Le partage de données proposé par Spark est effectué de manière efficace. Une fois le jeu de données transmis à une machine, cette machine est à même de le transmettre à d'autres machines du cluster. Pour n machines, le jeu de données est alors transmis en $\log(n)$ rondes.

Exemple Pour un débit de 1 Go par seconde, sur un cluster constitué de 32 machines, le vecteur de centroïdes est transmis à une autre machine en 16 μs.

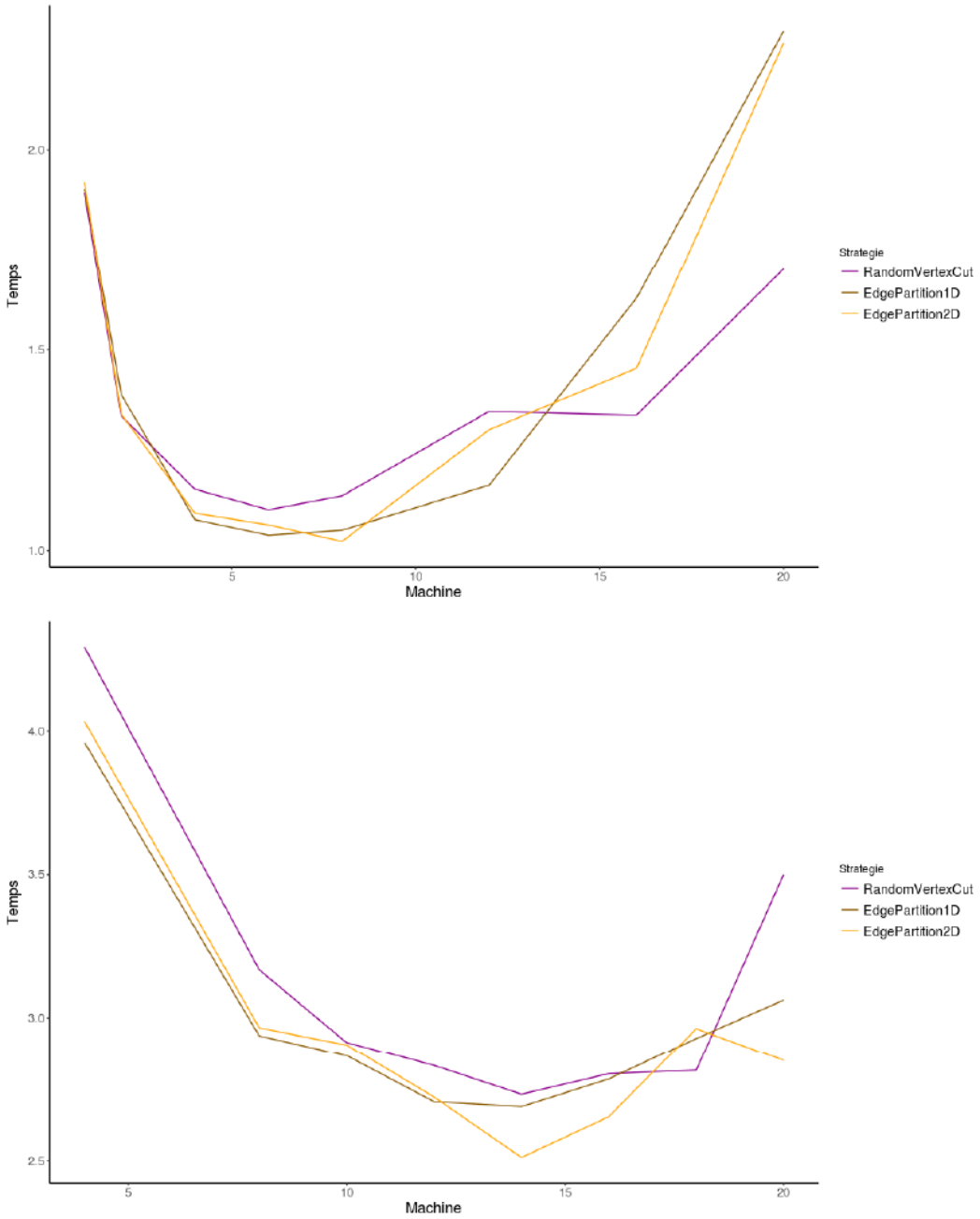


FIGURE 5.10 – Temps d'exécution médian par itération de GDAD en fonction du nombre de machines utilisées pour trois stratégies de partitionnement implémentées par GraphX [138]. *(Haut)* Temps d'exécution pour le graphe crack. Crack contient 10 240 nœuds et 30 380 arêtes. *(Bas)* Temps d'exécution pour le graphe finan512. Finan512 contient 74 752 nœuds et 261 120 arêtes. Hachul et Jünger [64] classent ces deux graphes dans la famille des graphes réels faciles à dessiner.

Transmettre les centroïdes à 32 machines est $\log_2(32) = 5$ fois plus lent, soit donc 80 μs.

Mise à jour de la position

Combinaison des positions intermédiaires La mise à jour de la position des centroïdes est réalisée grâce à une opération Map et une opération Reduce, cf Section 5.3.2. Dans l'opération Map, la distance à l'ensemble des centroïdes existants est calculée de manière à trouver le centroïde le plus proche. La position moyenne des nœuds associée à un même centroïde est alors calculée de manière à mettre à jour la position du centroïde (opération Reduce).

Cet algorithme bénéficie d'une étape de combinaison après l'opération Map. En effet, il y a plus de nœuds que de centroïdes. Ainsi, sur chaque partition, par principe des tiroirs, plusieurs nœuds sont proches du même centroïde. La combinaison de ces valeurs permet de transmettre moins de données lors du shuffle effectué avant l'opération de Reduce.

Partitionnement et combinaison Pour limiter le shuffle (*i.e.* la transmission des données) avant l'étape de Reduce, l'idéal est de placer les nœuds associés au même centroïde sur la même partition. De cette manière, aucun shuffle n'est nécessaire : les nœuds associés à un centroïde se trouvent sur la même partition.

Effectuer un nouveau partitionnement des nœuds à chaque itération n'est pas envisageable. En effet, cela revient à déplacer l'ensemble des données vers de nouvelles partitions. Cependant, au fur et à mesure de la convergence du dessin, la position des nœuds se fige et les clusters sont bien définis. Renouveler le partitionnement à intervalles réguliers peut être un compromis pour accélérer l'étape de Reduce.

Nous avons tenté d'effectuer un nouveau partitionnement à un nombre d'itérations fixe. Le temps de calcul médian pour une itération est multiplié par un facteur 4 en effectuant un nouveau partitionnement. Ce phénomène est probablement lié au coût du partitionnement des nœuds, qui implique un shuffle de l'ensemble des données. Le gain obtenu par le partitionnement des nœuds ne compense pas le temps de partitionnement de ces nœuds. Nous n'effectuons donc pas de partitionnement des nœuds dans notre implémentation de l'algorithme de dessin.

Intervalle de mise à jour

Inspiré par les travaux d'Hegeman et al. [71] sur l'intervalle de mise à jour d'un quadtree pour la simulation physique de particules au GPU, nous avons essayé de ne mettre à jour les centroïdes qu'à un intervalle donné.

Nous avons expérimenté avec une mise à jour de la position des centroïdes toutes les 5 itérations. Les dessin obtenus par cette optimisation semblent,

visuellement au moins, équivalents à ceux obtenus sans. Cependant, cette optimisation n'a pas amélioré le temps de calcul de manière significative.

5.4.4 Implémentation séquentielle

Un algorithme de dessin nécessite de nombreux réglages (*e.g.* les constantes du modèle physique, des techniques d'accélération de la convergence *etc.*) pour parvenir à des dessins acceptables. Il est plus simple de prototyper l'algorithme dans un environnement séquentiel. En effet, l'exécution des jobs Spark est plus lente qu'une implémentation séquentielle pour des petits graphes, notamment à cause du surcoût de communication. L'implémentation dans le paradigme MapReduce est aussi moins intuitive que dans un environnement séquentiel.

Une fois le prototype stabilisé, nous traduisons cet algorithme en Spark. L'algorithme a été implémenté de manière séquentielle en `C++`. Ces deux implémentations donneront lieu à des comparaisons sur les temps d'exécution de l'algorithme, *cf.* Section 5.5.1.

5.5 Résultats expérimentaux et performances

Dans cette section, nous revenons sur des résultats expérimentaux de notre algorithme de dessin par centroïde. Dans l'ensemble de cette section, les temps d'exécution de l'implémentation distribuée ont été obtenus à partir d'une plateforme de calcul distribué. La plateforme est un cluster composé de 16 machines. Chaque machine du cluster dispose de 64 Go de RAM et de 2x6 cœurs hyperthreadés à une fréquence de 2.1 GHz ainsi que de deux disques de 1 To. Les machines du cluster sont connectées par un réseau à 1Go par seconde.

Les temps d'exécution de l'implémentation séquentielle ont été obtenus à partir d'un ordinateur portable. L'ordinateur dispose d'un processeur Core i7-4710HQ, soit 2x4 cœurs hyperthreadés, et de 16 Go de RAM.

Nous comparons ensuite les temps d'exécution de l'algorithme dans une implémentation distribuée et séquentielle. Dans un dernier temps, nous comparons les performances de notre algorithme distribué avec Clint, l'algorithme d'Arleo et al. [8]. Nous effectuons une comparaison visuelle de quelques grands graphes ainsi qu'une évaluation du nombre d'itérations requises pour atteindre l'état d'équilibre.

5.5.1 Comparaison des performances : approche séquentielle et approche MapReduce

Le tableau 5.11 compare les temps de l'implémentation séquentielle et MapReduce de GDAD. L'implémentation séquentielle est réalisée en `C++` (*cf.* Section 5.4.4) et l'implémentation MapReduce en Spark (*cf.* Section 5.4). Les

Jeu de données	|V|	|E|	Séquentiel	MapReduce
crack	10 240	30 380	0,0153	1,14
fe_pwt	36 463	144 794	0,0538	1,87
finan512	74 752	261 120	0,114	2,03
fe_ocean	143 437	409 593	0,219	3,24
Amazon0302	262 111	899 792	0,503	5,60
Com_DBLP	317 080	1 049 866	0,625	5,75
Com_Amazon	334 863	925 872	0,643	6,01
Web-google	875 713	4 322 051	0,873	9,68
Roadnet_PA	1 087 562	1 541 514	1,17	9,46
delaunay_n22	4 100 000	12 200 000	4,09	18,3
hugetric-00010	6 600 000	10 000 000	7,15	19,9

FIGURE 5.11 – Tableau comparatif des temps d'exécution des implémentations séquentielles et distribuées de GDAD. Les temps obtenus sont les temps d'exécution médians pour une itération de l'algorithme GDAD.

temps obtenus sont les temps d'exécution médians pour une itération de l'algorithme GDAD.

Pour l'implémentation MapReduce, nous avons choisi le temps médian pour le nombre de machines optimisant l'exécution sur le cluster.

Pour les deux implémentations, le temps de lecture des données n'est pas pris en compte. Dans l'implémentation MapReduce, le temps de partitionnement du graphe n'est pas non plus pris en compte.

Performances séquentielles

Nombre de nœuds Le temps de calcul séquentiel croît linéairement avec le nombre de nœuds dans le graphe. Lorsque le temps est exprimé comme une fonction affine du nombre de nœuds, la régression linéaire a un coefficient de détermination $r = 0.995$. Ce coefficient compris entre 0 et 1 évalue la part de variation du temps qui peut être imputée aux variations du nombre de nœuds par le modèle.

Nombre d'arêtes Le temps de calcul séquentiel croît aussi linéairement avec le nombre d'arête. Cependant, la régression linéaire colle moins bien aux données. Ce résultat est cependant moins significatif, pour deux raisons.

Tout d'abord, la croissance des tailles du graphe entraîne l'augmentation du nombre de nœuds et d'arêtes simultanément. Il est donc normal de trouver une corrélation entre l'augmentation du nombre d'arêtes et le temps de calcul puisque le nombre de nœuds augmente simultanément.

De plus, dans l'implémentation séquentielle, les arêtes ne sont utilisées que

lors du calcul des forces d'attraction, qui est réalisé rapidement dans notre implémentation. Les calculs effectués lors de l'exécution d'une itération sont principalement le calcul des forces de répulsion et la mise à jour des centroïdes. Il n'y a pas, dans le paradigme séquentiel, de problème lié à la distribution des arêtes, comme cela peut être le cas dans le paradigme MapReduce.

Complexité de l'algorithme séquentiel Les performances séquentielles permettent de valider expérimentalement qu'une itération de GDAD est de complexité $O(|V|)$, *cf.* Section 5.2.2.

Performances MapReduce

Les performances de notre implémentation Spark de GDAD sont aussi présentées en Figure 5.11. Les temps présentés dans ce tableau correspondent au temps médian d'une itération de GDAD pour le nombre de machines offrant les performances optimales sur notre cluster de calcul. Ces essais ont été lancés sur des exécuteurs disposants de 7Go de mémoire.

Scalabilité horizontale Les temps optimaux obtenus avec notre implémentation n'assurent pas la scalabilité horizontale. En ajoutant des machines, nous n'arrivons pas à obtenir des performances constantes de GDAD sur le cluster.

Le temps de calcul augmente par exemple avec le nombre de nœuds du graphe. Cependant, dans nos relevés temporels, nous n'avons pas observé de corrélation entre la performances et le type de graphes à dessiner.

Complexité de l'implémentation La comparaison avec la version séquentielle et la croissance des temps de calcul médian d'une itération indiquent que notre implémentation Spark de l'algorithme GDAD n'est pas de complexité linéaire. Cette remarque est validée par les temps expérimentaux qui sont bien modélisés par une fonction racine du nombre de nœuds dans le graphe.

Les temps exprimés sont obtenus pour le nombre de machines maximisant les performances de l'algorithme. En prenant en compte la scalabilité horizontale, notre implémentation a donc une complexité totale de $O(|V|^{\frac{3}{2}})$.

Nous n'avons pas déterminé d'où provient cette différence de performances pour notre implémentation Spark de l'algorithme. Il est possible que le problème provienne de l'une des fonctions Spark [139] ou GraphX [138] sur laquelle nous nous appuyons dans notre implémentation.

Comparaison avec la version séquentielle Par rapport à l'implémentation séquentielle de GDAD, pour laquelle les temps d'exécution croissent linéairement avec le nombre de nœuds, l'implémentation distribuée offre de meilleures performances avec des graphes contenant plus de nœuds.

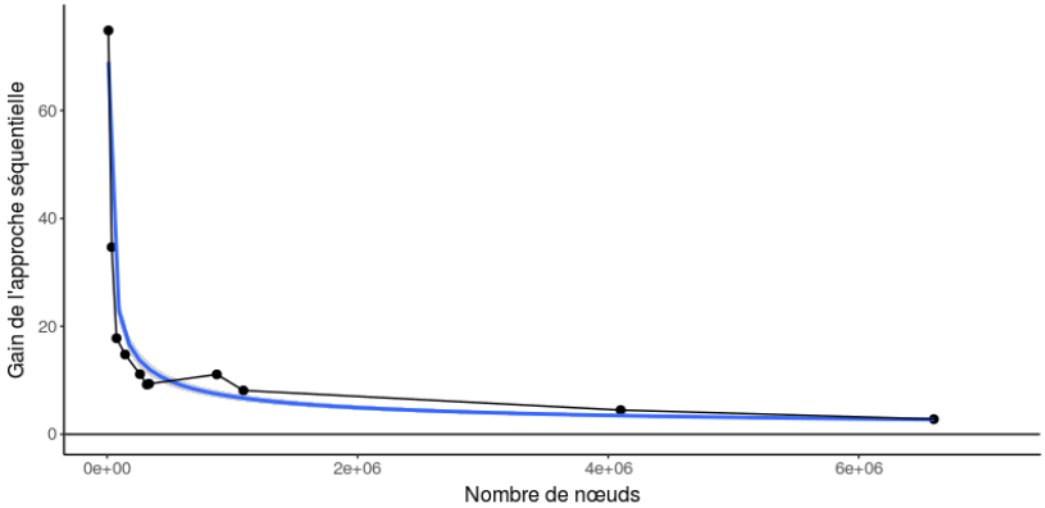


FIGURE 5.12 – Gain de l'implémentation séquentielle par rapport à l'implémentation MapReduce en fonction du nombre de nœuds du graphe. La courbe de gain est représentée avec une courbe de régression d'équation $y = \frac{c}{\sqrt{x}}$, avec c le paramètre de régression.

La Figure 5.12 présente le gain d'une itération effectuée avec l'implémentation séquentielle par rapport à l'implémentation MapReduce. Le gain est bien approché par une courbe d'équation $y = \frac{c}{\sqrt{x}}$. La régression a un coefficient de détermination $r = 0.9807$. Pour cette régression, le paramètre c a pour valeur $c = 6985.1$.

En supposant que le modèle de régression reste valable pour un nombre de nœuds plus important, on peut déduire le nombre de nœuds à partir duquel il est intéressant d'utiliser notre implémentation Spark pour mener le calcul d'une itération :

$$\frac{6985.1}{\sqrt{v}} < 1$$

En résolvant cette inéquation, on trouve alors qu'il est avantageux d'utiliser l'implémentation MapReduce lorsque $v > 6985.1 = 48\,791\,622$ nœuds.

5.5.2 Comparaison avec Clint

Dans le tableau de la Figure 5.13 nous présentons les performances de l'algorithme Clint d'Arleo et al. [8] ainsi que les performances de GDAD (implémentation séquentielle et MapReduce) pour différents grands graphes.

Pour Clint et l'implémentation distribuée de GDAD, les temps d'exécution correspondent au temps médian d'une itération pour le nombre de machines qui optimise ce temps (*workers* dans le cas de Giraph et *executors* dans le cas

			GDAD		Clint		GDAD Séq.
Jeu de données	|V|	|E|	Temps	# Mach.	Temps	# Mach.	Temps
Amazon0302	262 111	899 792	5,60	40	3,45	40	0,503
Com_DBLP	317 080	1 049 866	5,75	50	21,4	160	0,625
Com_Amazon	334 863	925 872	6,01	48	2,56	40	0,643
Web-google	875 713	4 322 051	9,70	100	-	-	0,873
Roadnet_PA	1 087 562	1 541 514	9,46	100	1,37	40	1,17
delaunay_n22	4 100 000	12 200 000	18,3	160	10,2	60	4,09
hugetric-00010	6 600 000	10 000 000	19,9	250	5,79	80	7,15

FIGURE 5.13 – Comparatif des temps d'exécution de GDAD [75] et Clint [8]. Le temps de calcul correspond au temps médian d'une itération de l'algorithme de dessin. Le temps médian est obtenu à partir du nombre optimal de machines pour l'exécution sur le cluster de calcul. Le nombre de machines est indiqué pour chacun des jeux de données. Les temps de calcul de GDAD dans son implémentation séquentielle sont présentés pour référence.

de Spark).

Clint utilise une distance topologique pour effectuer le calcul des forces de répulsion , *cf.* Section 5.2.3. La distance retenue pour cette comparaison est de 3, la valeur par défaut du paramètre dans l'implémentation fournie par Arleo.

Comparaison des performances de Clint et GDAD

Clint a de meilleures performances par itération que l'implémentation Spark de GDAD pour la plupart des jeux de données présentés en Figure 5.13. Les performances obtenues par Clint ne sont, de plus, pas directement corrélées au nombre de nœuds.

En comparaison de l'implémentation séquentielle de GDAD, l'implémentation d'Arleo a de meilleures performances uniquement pour le graphe hugetric-00010, contenant 6 millions de nœuds et 10 millions d'arêtes. En ce sens, l'implémentation d'Arleo tire parti de la distribution des données avant notre implémentation Spark de GDAD (meilleures performances à partir de 48 millions de nœuds, *cf.* Section 5.5.1).

Performances des environnements de calcul L'écart entre les performances distribuées de Clint et GDAD peut être en partie expliquée par l'environnement de calcul utilisée, *i.e.* respectivement Giraph [12] et GraphX [138]. Dans un comparatif de 2016, Kabiljo et al. [83] comparent les performances de GraphX et Giraph.

Dans leur conclusion, Kabiljo et al. relèvent que Giraph est capable de gérer des graphes 50 fois plus grands que ceux traités par GraphX et qu'il présente de meilleures performances pour les petits graphes. Pour les performances optimales des deux systèmes, Giraph utilise moins de machines et a de meilleures

Jeu de données	|V|	|E|	Clint Temps	Clint # Mach.	Distribution des arêtes Degré Max.	Distribution des arêtes Diamètre
Amazon0302	262 111	899 792	3,45	40	425	32
Com_DBLP	317 080	1 049 866	21,4	160	343	21
Com_Amazon	334 863	925 872	2,56	40	549	44
Web-google	875 713	4 322 051	-	-	6353	21
Roadnet_PA	1 087 562	1 541 514	1,37	40	18	786
delaunay_n22	4 100 000	12 200 000	10,2	60	23	-
hugetric-00010	6 600 000	10 000 000	5,79	80	3	-

FIGURE 5.14 – Temps d'exécution de Clint [8] en fonction de métriques de distribution des arêtes. Pour les graphes ayant un grand degré max et un petit diamètre, une itération de Clint prend un temps considérable avec un nombre important de machines (DBLP) voire n'aboutit pas (web-google). Cela confirme notre analyse de l'algorithme, *cf.* Section 5.2.3.

performances que GraphX.

Pour le graphe de Twitter, les temps de calculs obtenus pour 16 machines avec Giraph sont 4.5 fois inférieurs à ceux obtenus avec GraphX. Les écarts de temps que nous avons relevés sont de cet ordre. Le gain de temps s'échelonne entre un facteur 1.5 et 7 avec un gain moyen de 3.22 pour les graphes où Clint a de meilleures performances.

Les deux algorithmes sont cependant différents, ce qui rend une comparaison des deux implémentations plus difficile.

Limites de Clint

Pour deux jeux de données, com_dblp et Web-google, les performances de Clint sont en deçà de celles de l'implémentation Spark de GDAD. Pour le graphe dblp, représentant le graphe de personnes collaborant à des publications scientifiques en informatique, Clint est 4 fois plus lent que l'implémentation Spark de GDAD. Pour le graphe Web-google, représentant les hyperliens entre différents sites internet en 2002, le calcul n'a pas été mené à bout avec jusqu'à 300 *workers* en Giraph.

En Figure 5.14, nous présentons quelques métriques sur la distribution des arêtes (lorsqu'elles sont connues) en vis-à-vis des temps de calcul de Clint.

Degré maximum Le degré maximum est une métrique utile pour évaluer le nombre de voisins topologiques. La complexité de l'algorithme de Clint repose sur ce degré maximum, *cf.* Section 5.2.3. Soit Δ, le degré maximum du graphe. Lorsque tous les nœuds sont de degré Δ, il y a au plus Δ^k nœuds à distance k ou moins.

Les performances de Clint sont donc en partie liées à ce degré maximum.

Les deux graphes pour lesquels Clint a de moins bonnes performances ont un degré maximum élevé. Ainsi, pour dblp, le degré maximum est de 343 et pour web-google le degré maximum est de 6353, plus que tout autre jeu de données par un facteur 10.

Diamètre Le diamètre du graphe nous renseigne aussi sur la manière dont les arêtes sont réparties. Le diamètre correspond à la distance maximum entre deux nœuds du graphes. Cette métrique renseigne, indirectement, sur le nombre de voisins topologiques d'un nœud donné. Soient d le diamètre du graphe et u un nœud quelconque du graphe. Tout nœud v est à distance graphe d ou moins de u.

Les deux graphes qui ont des performances moindres en utilisant Clint ont un petit diamètre ($d = 21$) par rapport aux autres graphes du jeu de données. A l'inverse, Clint a de très bonnes performances pour le jeu de données RoadNet-PA qui est de diamètre 786.

Conclusion

Les résultats expérimentaux montrent que Clint ne présente pas de scalabilité avec la taille des graphes. Les performances de Clint sont liées à la topologie du graphe.

Les grands graphes stockés dans des infrastructures distribuées proviennent principalement de réseaux sociaux et sont généralement qualifiés de graphes petit monde. Pour ces graphes, possédant un petit diamètre et un degré maximum important, les temps de calculs expérimentaux de Clint sont au moins quatre fois supérieurs à ceux de GDAD. Dans le pire des cas, l'exécution de l'algorithme n'aboutit pas.

Ces résultats expérimentaux confirment les résultats théoriques sur le nombre de voisins topologiques. Lorsque le graphe contient des nœuds centraux de degré important, le nombre de voisins topologiques explose. Dans le pire des cas, il est de l'ordre de $|V|$, ce qui revient à calculer des forces de répulsion entre tous les nœuds du graphe. Ce n'est pas envisageable dans le paradigme MapReduce, *cf.* Section 4.3.2.

5.5.3 Comparaison visuelle des dessins et nombre d'itérations

Dans cette section, nous présentons quelques dessins obtenus avec GDAD et les comparons avec les dessins obtenus par FM^3 [63], un algorithme de dessin multiéchelle par modèle de force. FM^3 est un algorithme rapide donnant de très bons dessins indépendamment de la topologie du graphe.

Il est généralement admis que $O(|V|)$ itérations d'un algorithme de dessin par modèle de force sont nécessaires pour atteindre un état d'équilibre. Nous

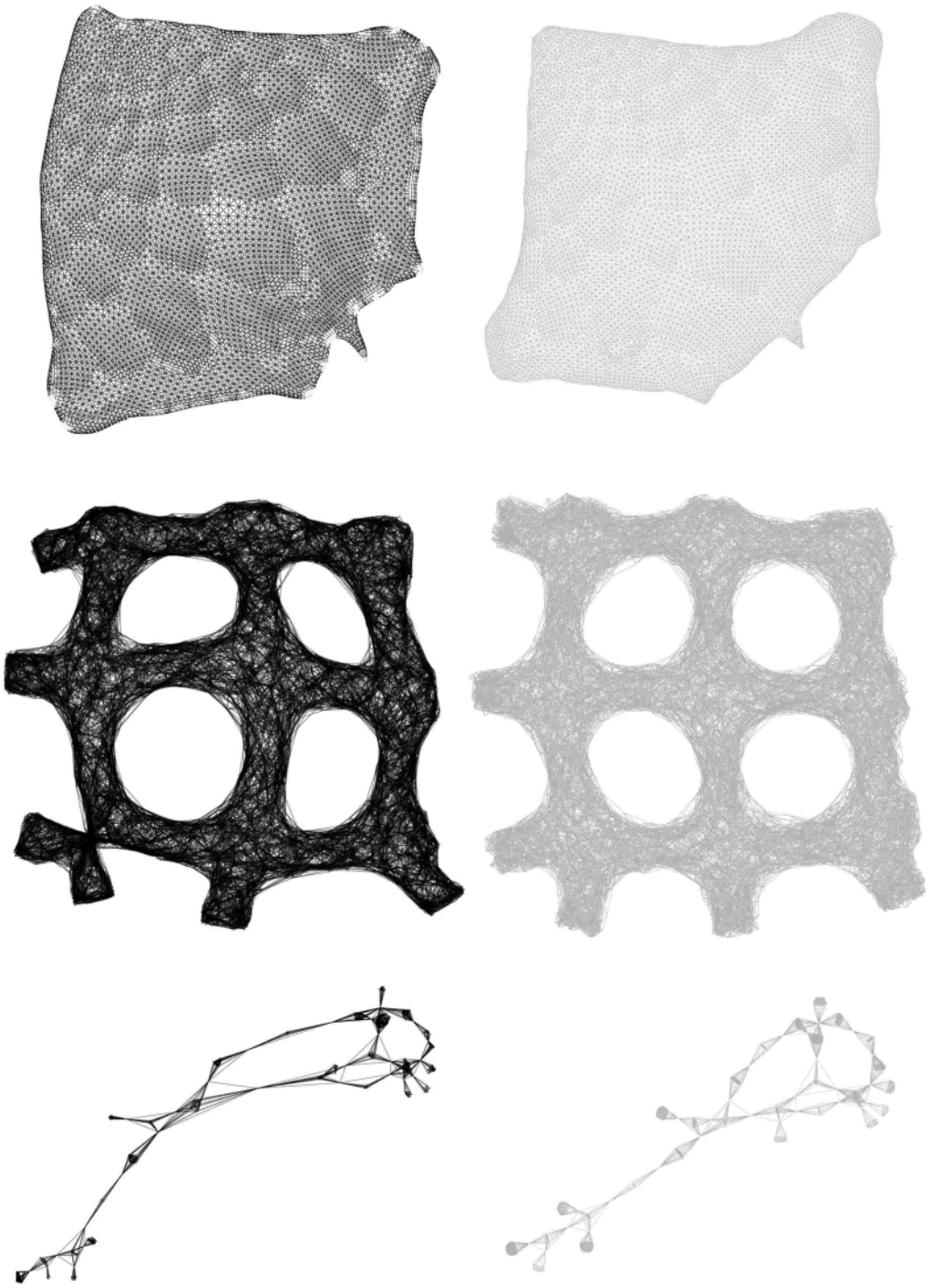


FIGURE 5.15 – Comparaison visuelle de dessins obtenus par GDAD *(Gauche)* et FM^3 *(Droite)*. *(Haut)* Dessin de crack. *(Milieu)* Dessin d'une grille aléatoire. *(Bas)* Dessin de imdb_small.

donnons donc un ordre d'idées du nombre d'itérations nécessaire pour atteindre un dessin optimal avec GDAD.

Comparaison visuelle de plusieurs graphes

La Figure 5.15 et la Figure 5.16 présentent les dessins de trois graphes obtenus avec GDAD et FM^3.

Présentation des graphes

Crack est un graphe de 10 240 nœuds et 30 380 arêtes. Crack est une triangulation. Finan_512 est un graphe de 74 752 nœuds et 261 120 arêtes. Fe_ocean. est un graphe de 143 437 nœuds et 409 593 arêtes. Hachul et Jünger [64] classifient ces trois graphes comme graphe réel simple à dessiner.

La grille aléatoire est composé de 8000 nœuds et 35 000 arêtes. Ce graphe est obtenu en plaçant les nœuds selon une grille régulière, en supprimant des nœuds régulièrement pour former les trous et en ajoutant un nombre constant d'arêtes à chaque nœuds. Le triangle de Sierpinski est une surface fractale générée récursivement en évidant un triangle plein pour créer trois sous-triangles. Notre graphe, obtenu après 8 itérations, est constitué de 3282 nœuds et 6561 arêtes.

Imdb_small est un graphe de 298 nœuds et 2058 arêtes. Dans ce réseau, deux acteurs ayant joué ensemble dans au moins un film sont connectés par une arrête. Imdb_small représente un sous-ensemble des acteurs recensés sur le site IMDB. Des communautés apparaissent clairement.

Comparaison des dessins

Pour les dessins obtenus en Figure 5.15, les dessins obtenus avec GDAD sont proches de ceux obtenus avec FM^3. Pour crack, la structure globale est parfaitement discernable et la structure locale est détaillée. Pour la grille aléatoire, la structure globale et locale sont aussi parfaitement discernables avec toutefois une torsion du dessin dans le coin inférieur gauche. Pour imdb_small, le dessin obtenu par GDAD fait clairement apparaître les mêmes structures que le dessin obtenu avec FM^3 selon une disposition similaire. GDAD retourne un dessin légèrement plus distendu que celui obtenu avec FM^3.

Pour les dessins de finan512 et Sierpinski, *cf.* Figure 5.16, la structure locale est discernable dans les dessins générés par GDAD. Cependant, la structure globale n'est pas aussi nette que sur les dessins obtenus avec FM^3. Pour ces deux graphes, les dessins obtenus avec GDAD sont dans un état intermédiaire correspondant à un minimum local. Le déplacement des forces permet d'échapper à cet état pour converger vers un minimum global, mais ce processus nécessite un grand nombre d'itérations.

Pour le dessin de fe_ocean, *cf.* Figure 5.16, le dessin obtenu par GDAD est plus éloigné de l'état d'équilibre obtenu avec FM^3. Ce graphe est le plus

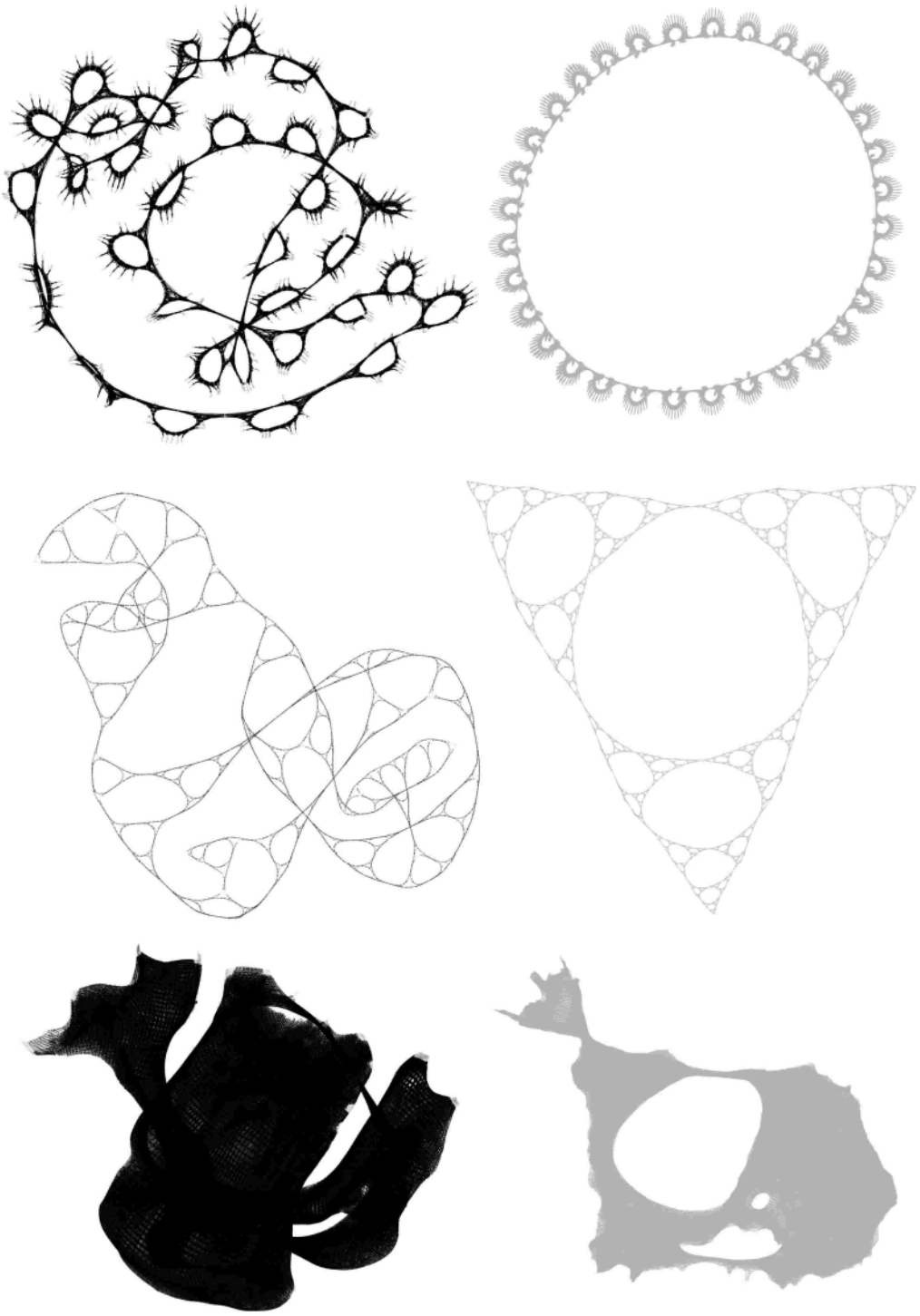


FIGURE 5.16 – Comparaison visuelle de dessins obtenus par GDAD *(Gauche)* et FM^3 *(Droite)*. *(Haut)* Dessin de finan512. *(Milieu)* Dessin d'un triangle de Sierpinski (8 niveaux). *(Bas)* Dessin de fe_ocean.

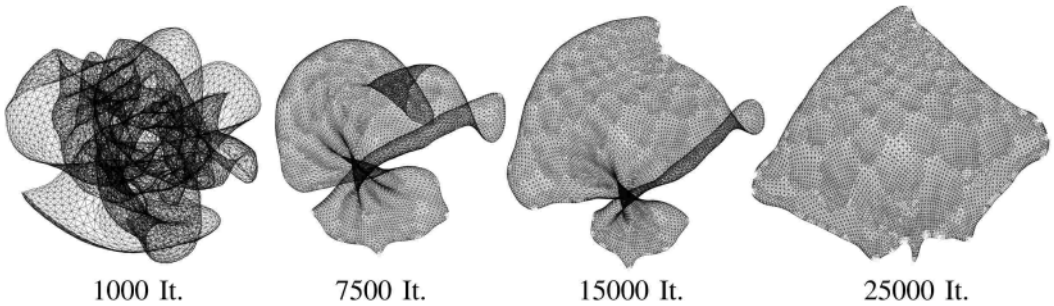


FIGURE 5.17 – Dessins de crack à quatre étapes du dessin avec GDAD. Le nombre d'itérations du modèle de forces est indiqué sous chaque dessin.

gros des graphes pour lesquels nous présentons une comparaison visuelle (avec 143 437 nœuds et 409 593 arêtes). Nous avons arrêté le dessin après 20 000 itérations. Le dessin n'a donc probablement pas totalement convergé. On peut cependant voir quelques éléments de la structure globale, comme l'épi dans le coin supérieur gauche ou encore les deux trous en bas à droite du dessin.

Nombre d'itérations et équilibre

Vitesse initiale et finale de convergence En Figure 5.17, nous représentons quatre étapes du dessin de crack avec GDAD, pour 1000, 7500, 15 000 et 20 000 itérations. On peut noter que la convergence initiale (passage de 1000 à 7500 itérations) entraîne une modification plus rapide du graphe que les paliers suivants (de 7500 à 15 000 et de 15 000 à 25 000). Une fois proche de l'équilibre, la convergence est en effet plus lente et nécessite donc plus d'itérations.

Ce phénomène est aussi observé pour les autres graphes dont nous avons présenté le dessin en Figure 5.15 et en Figure 5.16. Le phénomène est particulièrement visible pour la grille trouée et le triangle de Sierpinski.

La différence de vitesse de convergence est liée aux minimums locaux de la fonction d'énergie. Une fois le dessin proche d'un minimum local, seuls un petit nombre de nœuds sont déplacés de manière significative. L'algorithme de dessin requiert alors de nombreuses itérations pour échapper à l'orbite de ce minimum local pour converger vers un minimum global.

Pour éviter ce phénomène, les algorithmes de dessin par modèle de forces sont généralement associés à une décomposition multiéchelle, permettant d'obtenir des dessins proches du minimum global à chaque niveau de la décomposition. Il n'est alors pas nécessaire d'appliquer de nombreuses itérations du modèle de forces pour passer d'un minimum local à un minimum global. Cette technique est décrite en détail en Section 3.5.

Nombre d'itérations et équilibre Crack contient 10 240 nœuds. Pour crack, le dessin final est obtenu après $2.5 \cdot |V|$ itérations. Pour les dessins présentés en Figure 5.15, le dessin final est obtenu après $1.88 \cdot |V|$ itérations pour la grille aléatoire et après $3.3 \cdot |V|$ itérations pour imdb_small.

Pour les dessins présentés en Figure 5.16, les dessins obtenus par GDAD correspondent à $0.2 \cdot |V|$ itérations pour finan512 et à $3.05 \cdot |V|$ itérations pour le graphe de Sierpinski. Pour le dessin de fe_ocean, il représente $0.12 \cdot |V|$ itérations. Le dessin n'a cependant pas totalement convergé.

D'après ces résultats expérimentaux, nous affirmons que GDAD converge en $O(|V|)$ itérations. Le nombre d'itérations réellement utiles dépend cependant du nombre de minima locaux à éviter (par exemple Sierpinski et finan512). Pour éviter ces minima locaux, les algorithmes de dessin par modèle de forces utilisent une approche multiéchelle, *cf.* Section 3.5.

5.6 Conclusion

Dans ce chapitre, nous avons présenté en détail notre cheminement pour obtenir GDAD [75], le premier algorithme de dessin de graphe dans un environnement MapReduce.

Dans la Section 5.2.1, nous présentons l'approximation des forces de répulsion à l'aide de pivots. Ces pivots permettent de réduire le nombre de forces de répulsion à $O(|V|)$ interactions, en bornant le nombre de pivots. GDAD utilise ces pivots pour effectuer le calcul des forces de répulsion.

Les résultats expérimentaux sont encourageants pour GDAD. Les dessins obtenus par GDAD sont proches de ceux obtenus par FM^3 [63]. GDAD converge vers un dessin optimal en $O(|V|)$ itérations.

GDAD a une complexité temporelle de $O(|V|+|E|)$ par itération. Pour l'implémentation séquentielle de GDAD, nous avons retrouvé expérimentalement ces performances. Pour l'implémentation Spark de GDAD, les temps obtenus par itération indiquent que la complexité est de $O(|V| \cdot \sqrt{|V|})$.

Le nombre d'itérations reste un facteur limitant pour le dessin dans le paradigme MapReduce. Chaque itération de l'implémentation Spark de GDAD est coûteuse. Le nombre d'itérations pour atteindre la convergence du dessin nécessite cependant $O(|V|)$ itérations.

Les approches multiéchelles ont été créées pour réduire le nombre d'itérations des algorithmes de dessin et améliorer la stabilité du dessin final. Cependant, les approches présentées dans la section 3.5 sont généralement difficiles à paralléliser, et de surcroît à distribuer dans un environnement MapReduce. Cela constitue l'enjeu de notre prochain chapitre.

Chapitre 6

Dessin de graphe multiéchelle dans un environnement distribué

6.1 Introduction

Les résultats présentés dans le chapitre précédent ont mis en lumière les limites des algorithmes de dessin distribué (notre algorithme et celui d'Arleo et al. [8]) : le nombre d'itérations nécessaires pour que le dessin converge est le facteur limitant principal. La solution adoptée par les algorithmes séquentiels de l'état de l'art est une technique appelée la décomposition multiéchelle, *cf.* Section 3.5.

Chaque itération de notre algorithme de dessin, présenté dans le Chapitre 4, s'exécute en temps $O(|V|)$. Dans un environnement distribué, cette complexité est divisée par le nombre de machines à chaque opération, que ce soit une opération de Map ou de Reduce. Cependant, pour arriver à convergence, $|V|$ itérations sont nécessaires, portant la complexité de l'algorithme à $O(|V|^2)$.

La décomposition multiéchelle permet de réduire le nombre d'itérations nécessaires pour atteindre l'état d'équilibre du dessin. En décomposant le graphe initial en une suite de graphes quotients décroissante pour la taille, il est possible d'obtenir des conditions initiales proches de l'état d'équilibre. Par exemple, l'algorithme FM^3 utilise une décomposition de ce type et effectue un nombre constant d'itérations de l'algorithme de dessin à chaque niveau de la décomposition. Les décompositions multiéchelles permettent donc d'accélérer la convergence du dessin, ce que nous avons identifié comme un point bloquant pour notre algorithme de dessin seul.

Dans la première section, nous présentons en détail deux algorithmes multiéchelles de dessin de l'état de l'art : GRIP et FM^3. Pour ces deux algorithmes, la transposition de la décomposition multiéchelle récursive est étudiée dans un environnement distribué. Pour notre première contribution, nous détaillons

chaque étape de ces deux décompositions et décrivons les difficultés à les transposer dans un environnement distribué (paradigme MapReduce ou Pregel).

Les décompositions récursives utilisées dans GRIP et FM³ reposent sur la sélection d'un ensemble indépendant maximal de nœuds du graphe. Nous présentons une famille d'algorithmes développés par Luby en 1985 [100] permettant d'effectuer cette sélection de manière parallèle et montrons comment elle peut être transposée directement dans le paradigme Pregel. En 2016, Arleo et al. [9] redécouvrent indépendamment une variante de ces algorithmes sans en donner la complexité. Nous proposons d'établir la complexité de la décomposition récursive utilisée par Arleo et al.

Nous avons développé un algorithme de dessin multiéchelle distribué, appelé MuGDAD (Multilevel Graph Drawing Algorithm in Distributed) utilisant les forces approchées par centroïdes décrites dans le chapitre précédent. La décomposition récursive et le placement initial utilisés dans MuGDAD sont présentés en détail, dans le paradigme séquentiel et dans un environnement distribué. Nous présentons aussi des techniques nous permettant d'optimiser l'algorithme de dessin par centroïdes en combinaison avec l'approche multiéchelle.

Dans nos deux dernières sections, nous présentons l'implémentation de MuGDAD dans un environnement séquentiel (en C++) et dans un environnement distribué (en Spark). Nous présentons aussi une série de résultats expérimentaux.

6.2 Décomposition multiéchelle : l'exemple de FM³ et GRIP

Dans cette section, nous détaillons la structure d'un algorithme multiéchelle par modèle de force à travers deux exemples d'algorithmes : GRIP de Gajer et Kobourov [50] ainsi que FM³ de Hachul et Jünger [63]. Les deux algorithmes utilisent une simplification récursive du graphe d'origine combiné à un algorithme de dessin de graphe par modèle de force pour dessiner chaque niveau.

Nous étudions la possibilité de transposer ces deux algorithmes dans un environnement distribué. Ces résultats sont notre première contribution au dessin de graphe multiéchelle dans un environnement distribué.

6.2.1 Structure générique d'un algorithme multiéchelle

Dans la phase de filtration, *cf.* Section 3.5.1, une suite de graphes est créée dont le nombre de nœuds est strictement décroissant. Un sous-ensemble de nœuds est sélectionné à chaque niveau. Les nœuds sélectionnés sont utilisés

pour créer un nouveau graphe conservant les propriétés topologiques du graphe parent.

Les graphes de cette suite sont dessinés avec un algorithme de dessin par modèle de force classique, en commençant par le graphe le plus petit. La position des nœuds dans le dessin est utilisée comme point de départ du dessin pour le graphe suivant. Les nœuds manquants sont ajoutés par un placement intelligent.

Les conditions initiales du dessin sont proches du minimum d'énergie et quelques itérations d'un algorithme de dessin par modèle de force suffisent pour converger. Ce processus est détaillé dans l'Algorithme 6.

Algorithm 6 Structure générique des algorithmes de dessin multiéchelles

```
G_0 = (V_0, E_0) ← Graphe initial                    ▷ Initialisation
i ← 0
while |V_i| > 3 do                                   ▷ Phase de sélection
    V_{i+1} ← Filtration(V_i)
    i ← i + 1
    Création du graphe G_i = (V_i, E_i)
end while
N ← i
for j de N à 0 do                                    ▷ Phase de dessin
    Calcul du dessin de G_j (quelques itérations)
    Placement des nœuds de G_{j-1} selon le dessin de G_j
end for
```

6.2.2 L'approche multiéchelle dans GRIP

Phase de filtration

A partir du graphe $G = (V, E)$, une filtration $\mathcal{V} : V = V_0 \supset V_1 \supset ... \supset V_N \supset \emptyset$ est calculée sur les nœuds jusqu'à ce qu'il ne reste que trois nœuds dans V_N au niveau N. Pour calculer cette filtration, GRIP utilise un ensemble indépendant maximal des nœuds de G_i pour obtenir l'ensemble de nœuds V_{i+1}. Cet ensemble maximal est calculé pour les nœuds de V_i à distance $2^i + 1$ dans le graphe initial $G = G_0$. Les niveaux V_{N-1} et V_N sont recalculés de manière à obtenir exactement trois nœuds dans l'ensemble V_N.

Pour construire V_{i+1} à partir de V_i, il faut trouver les nœuds à distance 2^i ou moins dans le graphe initial G. GRIP [50] construit un arbre de parcours en largeur (Breadth First Search ou BFS) de profondeur 2^i pour chaque nœud de V_i de manière à déterminer les nœuds voisins pour calculer l'ensemble maximal indépendant.

Adaptation dans un environnement distribué

GRIP [50] conserve le graphe original G comme référence. Celui-ci est utilisé pour le calcul de la filtration, où les nœuds sont considérés comme voisins ou non selon leur distance réelle dans le graphe. Il est aussi utilisé pour effectuer le dessin puisqu'à chaque étape, c'est la distance réelle entre les nœuds qui est utilisée comme poids sur les arêtes pour effectuer le dessin.

Pour i variant de 0 à N, de moins en moins de nœuds sont sélectionnés mais les arbres de recherche doivent aller chercher les nœuds à distance 2^i. Pour les sous-ensembles de V contenant un nombre important de nœuds, il faut pouvoir maintenir autant d'arbres. Au contraire, pour les sous-ensembles de V ne contenant que quelques nœuds, les arbres de parcours en largeur correspondants sont très profonds. Dans un environnement de calcul distribué, il n'est pas possible que la profondeur de ces arbres représente tout le graphe ou presque. Un tel arbre contiendrait trop de nœuds pour pouvoir être stocké sur une seule machine du cluster. Pour cette raison, nous n'avons pas utilisé cette approche de filtration dans MuGDAD. La gestion des arbres de parcours en largeur de manière efficace dans un environnement distribué reste une question ouverte.

6.2.3 L'approche multiéchelle dans FM^3

Phase de filtration

Dans FM^3 [63], les niveaux sont calculés à l'aide d'un ensemble indépendant maximal à distance 2. Les nœuds sélectionnés dans cet ensemble sont séparés d'une distance graphe de 3 ou plus. L'algorithme séquentiel pour obtenir un MIS est répété jusqu'à ce que tous les nœuds soient dans un cluster. Cet algorithme est donc similaire à l'algorithme séquentiel pour obtenir un ensemble indépendant maximal à deux exceptions : les voisins à distance 2 sont supprimés du buffer et les clusters autour des nœuds sont conservés.

En plus de la sélection des nœuds, FM^3 crée des clusters de nœuds associés à chaque nœud sélectionné. Hachul et Jünger nomment ces clusters des systèmes solaires. Au sein de ces clusters, le nœud sélectionné est appelé soleil, les nœuds à distance 1 sont des planètes et les nœuds à distance 2 sont des lunes.

Une fois chaque nœud sélectionné ou dans un cluster un nouveau graphe est créé, résumant la structure du précédent graphe. Les clusters sont transformés en meta-nœuds, *i.e.* les nœuds du nouveau graphe, et des meta-arêtes, *i.e.* les arêtes du nouveau graphe, sont créées entre les meta-nœuds. Les meta-arêtes ont un poids qui correspond à la moyenne de la longueur des chemins connectant deux systèmes solaires (en tenant compte du poids associé à chaque arête).

Le graphe créé de cette manière est filtré de manière récursive jusqu'à obtenir un graphe suffisamment petit pour être dessiné rapidement.

Adaptation dans un environnement distribué

Dans la prochaine section, nous présentons deux algorithmes pour obtenir le stable maximal dans un environnement distribué [100]. L'adaptation de la décomposition proposée par FM^3 repose sur le fonctionnement de l'algorithme de décomposition. Nous revenons plus en détail sur l'adaptation de FM^3 dans un environnement distribué dans la Section 6.3.4.

6.3 Stable maximal et algorithmes de Luby

Les algorithmes de dessin multiéchelle FM^3 [63] et GRIP [50], que nous avons présentés en détail dans la section précédente, reposent tout les deux sur une décomposition récursive basée sur un ensemble indépendant maximal, aussi appelé stable maximal ou MIS. Dans le cas de FM^3, le MIS est réalisé à chaque niveau de la décomposition à distance 2. Pour GRIP, le MIS est réalisé à partir du graphe original pour des distances graphe de 2^i avec i le niveau de décomposition.

Un algorithme séquentiel classique pour obtenir un MIS est composé de deux étapes : un nœud du graphe est sélectionné au hasard, puis ce nœud et ses voisins sont supprimés du buffer contenant l'ensemble des nœuds du graphe. En répétant ces deux étapes jusqu'à ce que le buffer soit vide, on obtient un sous-ensemble maximal de nœuds non-voisins. Les étapes de cet algorithme sont décrites dans l'Algorithme 7.

Algorithm 7 Algorithme séquentiel pour un ensemble indépendant maximal

$V_{Buffer} \leftarrow V$
$V_{Selected} \leftarrow \emptyset$
while $|V_{Buffer}| > 0$ **do**
  $n \leftarrow$ Tirage aléatoire d'un nœud de V_{Buffer}
  $V_{Selected} \leftarrow V_{Selected} \cup \{n\}$
  $V_{Buffer} \leftarrow V_{Buffer} \backslash (\{n\} \cup \text{Neighbours}(n))$
end while

La parallélisation de l'algorithme séquentiel classique n'est pas immédiate. Les nœuds de l'ensemble maximal indépendant sont choisis les uns après les autres. Pour paralléliser cet algorithme, il faut pouvoir sélectionner plusieurs nœuds simultanément avec l'assurance que ces nœuds ne sont pas voisins.

6.3.1 Algorithmes parallèles pour le stable maximal

Luby [100] décrit en 1985 plusieurs algorithmes parallèles pour la recherche d'un ensemble indépendant maximal. Il propose une structure de sélection pa-

rallélisable décomposée en deux fonctions appliquées en parallèle : l'une sur les nœuds (appelée ALGVERTEX) et l'autre sur les arêtes (appelée ALGEDGE).

La fonction ALGVERTEX présélectionne les nœuds du graphe. Une fois cette étape réalisée, cette présélection est comparée selon les arêtes de manière à ne conserver qu'un sous-ensemble de nœuds indépendants.

La fonction ALGEDGE compare les nœuds présélectionnés. Pour chaque arête, les nœuds sont comparés de manière à déterminer quel nœud conserver dans l'ensemble indépendant intermédiaire. L'ensemble maximal obtenu de cette manière est bien indépendant : lors de ALGEDGE, l'algorithme vérifie que deux nœuds sélectionnés dans l'étape ALGVERTEX ne sont pas voisins.

La structure d'algorithme parallèle proposée par Luby est décrite dans l'Algorithme 8.

Algorithm 8 Structure d'un algorithme parallèle : étape de sélection

$V_{Buffer} \leftarrow V$
$V_{Selected} \leftarrow \emptyset$
while $|V_{Buffer}| > 0$ **do**
  Soit $G = (V_{Buffer}, E_{Buffer})$ le sous-graphe induit par V_{Buffer}
  $V_{Temp} \leftarrow \emptyset$

  for $i \in V_{Buffer}$, *en parallèle* **do** ▷ ALGVERTEX
    $V_{Temp} \leftarrow V_{Temp} \cup$ ALGVERTEX(i)
  end for

  for $e = (v_1, v_2) \in E_{Buffer}$, *en parallèle* **do** ▷ ALGEDGE
    if $v_1 \in V_{Temp}$ et $v_2 \in V_{Temp}$ **then**
      $v \leftarrow$ ALGEDGE(e)
      $V_{Temp} \leftarrow V_{Temp} \backslash \{v\}$
    end if
  end for

  $V_{Selected} \leftarrow V_{Selected} \cup V_{Temp}$
end while

6.3.2 ALGVERTEX et ALGEDGE

Luby [100] propose quatre algorithmes parallèles pour déterminer un ensemble indépendant maximal. Nous décrivons ici les deux premiers qui ne sont pas déterministes. L'algorithme A, présenté en Section 6.3.2, a une complexité moyenne en $O(\log |V|)$. L'algorithme B, présenté en Section 6.3.2, a une complexité moyenne en $O((\log |V|)^2)$. Les deux autres algorithmes présentés par

Luby sont une version déterministe des algorithmes présentés ici. Leur complexité est de $O((\log |V|)^2)$. Le fait que l'algorithme de sélection des nœuds soit aléatoire n'est pas un problème pour l'application au dessin de graphe multiéchelle, comme démontré dans FM^3 [63] et GRIP [50]. Nous préférons donc nous concentrer sur les algorithmes ayant la complexité minimale.

Nous faisons ici un point sur quelques notations. I désigne un sous-ensemble de nœuds indépendants. $N(I)$ désigne l'ensemble des nœuds voisins des nœuds de l'ensemble I. I' désigne un sous-ensemble de nœuds sélectionnés lors de l'itération courante. $G' = (V', E')$ désigne le sous-graphe de $G = (V, E)$ induit par $V \backslash (I \cup N(I))$.

Algorithme A

Le premier algorithme proposé par Luby est basé sur un ordre global des nœuds. Schématiquement, une fois le rang des nœuds déterminé, l'algorithme sélectionne un ensemble indépendant de nœuds dont le rang est minimal. De cette manière, plusieurs nœuds sont sélectionnés simultanément. Pour décrire l'algorithme, nous reprenons la structure établie en Algorithme 8 et détaillons les fonctions ALGVERTEX et ALGEDGE.

La fonction ALGVERTEX permet d'attribuer un rang à chaque nœud du graphe, de manière parallèle. Choisir une permutation des nœuds (*i.e.* un tirage sans remise dans $[\![1, |V'|]\!]$) est une opération essentiellement séquentielle : il faut s'assurer que deux nœuds n'ont pas le même rang. Pour paralléliser cette étape, les rangs sont tirés aléatoirement dans l'intervalle $[\![1, |V'|^4]\!]$, ce qui permet de tirer une permutation avec une bonne probabilité. Plus précisément, la probabilité qu'une paire de nœuds ait le même rang est $\frac{|V'|^2}{|V'|^4}$ puisqu'il y a $|V'|^2$ paires de nœuds et $|V'|^4$ rangs possibles. La probabilité que le tirage soit une permutation est alors de $1 - \frac{1}{|V'|^2}$. Des détails d'implémentation de cette méthode dans le cas de grand graphes sont donnés en Section 6.4.1.

L'algorithme parcourt ensuite les arêtes pour déterminer les nœuds ayant un rang plus petit que tous leurs voisins. Cette étape correspond à l'application de la fonction ALGEDGE. Si un nœud répond à ces conditions, il est sélectionné et lui et son voisinage sont supprimés du buffer I'. Ce processus est décrit par l'Algorithme 9. On peut noter que par cette fonction, seul les nœuds ayant un rang strictement inférieur à l'ensemble des rangs de leurs voisins peuvent être sélectionnés. Deux voisins ne peuvent donc pas être sélectionnés simultanément ce qui assure que l'ensemble maximal est bien indépendant.

Après plusieurs itérations de ces deux fonctions, lorsque $|V'| = 0$, l'ensemble indépendant maximum est obtenu. La complexité temporelle moyenne de cet algorithme est détaillées en section 6.3.3.

Algorithm 9 Algorithme A : ALGEDGE

```
π permutation des nœuds de V'
I' ← V'
for (i, j) ∈ E' do
    if π(i) < π(j) then
        I' ← I'\{j}
    else
        I' ← I'\{i}
    end if
end for
```

Algorithme B

L'algorithme B n'utilise pas un rang pour classer les nœuds mais présélectionne un sous-ensemble de nœuds aléatoirement selon leur degré. Seuls les nœuds ayant un degré supérieur à l'ensemble de leurs voisins sont conservés. Nous décrivons maintenant les fonctions ALGVERTEX et ALGEDGE.

La fonction ALGVERTEX nécessite que chaque nœud connaisse son degré lors de l'itération en cours. Pour chaque nœud, une variable aléatoire est définie à valeurs dans l'ensemble $[0, 1]$. Un tirage aléatoire est effectué pour chaque nœud et les nœuds ayant tiré la valeur 1 sont présélectionnés. La variable aléatoire prend la valeur 1 avec probabilité $\frac{1}{2 \cdot d(i)}$. Pour les nœuds de degré 0, la variable aléatoire prend la valeur 1 avec probabilité 1. Les nœuds de degré important ont donc moins de chances d'être présélectionnés de cette manière.

La fonction ALGEDGE parcourt ensuite les arêtes pour déterminer les nœuds présélectionnés de plus haut rang. Pour chaque arête, si ses deux extrémités sont présélectionnées, seul le nœud de plus haut degré est retenu et l'autre est supprimé. Ce processus est décrit dans l'Algorithme 10. Seuls les nœuds ayant un degré plus élevé que leurs voisins sont finalement sélectionnés, ce qui implique que l'ensemble maximal ainsi obtenu est bien indépendant. Les nœuds de haut degré ont une probabilité moins élevée que les autres nœuds d'être présélectionnés, mais s'ils le sont ils ont une forte probabilité d'être finalement sélectionné. Il est aussi à noter qu'en conservant les nœuds de plus haut degré le sous-graphe induit diminue rapidement de taille puisque les nœuds sélectionnés ont de nombreux voisins.

L'algorithme B nécessite de recalculer à chaque itération les degrés des nœuds pour le sous-graphe induit G'. Cette étape supplémentaire est réalisée en $O(\log |V|)$ pour $O(|E|)$ processeurs en parallèles, contre $O(1)$ pour l'algorithme A avec le même nombre de processeurs. Pour obtenir la complexité totale de ces algorithmes, il reste à évaluer le nombre d'itérations moyennes avant d'obtenir un MIS. Dans la prochaine section, nous présentons la preuve du nombre d'itérations moyen avant d'obtenir un MIS.

Algorithm 10 Algorithme B : ALGEDGE

```
I' ← X l'ensemble des nœuds présélectionnés
for (i, j) ∈ E' do
    if i ∈ X et j ∈ X then
        if d(i) < d(j) then
            I' ← I'\{i}
        else
            I' ← I'\{j}
        end if
    end if
end for
```

6.3.3 Nombre d'itérations des algorithmes de Luby

Dans cette section, nous donnons quelques éléments de preuve justifiant le nombre d'itérations nécessaires en moyenne pour obtenir un MIS des deux algorithmes de Luby présentés dans la section précédente. Ces éléments de preuve nous donnent des arguments pour choisir comment implémenter la sélection d'un sous-ensemble indépendant de nœuds dans un environnement distribué. Notamment, ce résultat nous permet d'obtenir la complexité de l'algorithme de sélection proposé par Arleo et al. [9], ce qui constitue l'une des contributions de notre thèse. L'ensemble des preuves peut être retrouvé dans le papier de Luby [100].

Théorème 6.1. *Soient Y_k^A (respectivement Y_k^B) les variables aléatoires représentant le nombre d'arêtes dans E' avant la k-ième itération de l'algorithme A (respectivement B).*

$$\mathbb{E}[Y_k^A - Y_{k+1}^A] \geq \frac{1}{8} \cdot Y_k^A - \frac{1}{16}$$

$$\mathbb{E}[Y_k^B - Y_{k+1}^B] \geq \frac{1}{8} \cdot Y_k^B$$

Le Théorème 6.1 permet de déduire la complexité temporelle de l'algorithme. En moyenne, à chaque itération, plus d'un huitième des arêtes du sous-graphe restant sont supprimées du buffer. Ainsi, le nombre d'arêtes décroît exponentiellement vite, tant pour l'algorithme A que l'algorithme B. En moyenne, la complexité temporelle de ces algorithmes est donc de $O(\log(|V|))$. Nous donnons ensuite des éléments de preuves pour l'algorithme B.

Éléments de preuves du Théorème 6.1. Les arêtes qui sont supprimées du buffer entre deux niveaux sont celles dont l'une des extrémités au moins se trouve dans l'ensemble des nœuds sélectionnés et de leur voisins. En sommant sur les nœuds, on compte ces arêtes deux fois. L'espérance peut alors être minorée de la manière suivante :

$$\begin{aligned}\mathbb{E}[Y_k^B - Y_{k+1}^B] &\geq \frac{1}{2}\sum_{i\in V'} d(i)\cdot \Pr[i \in I' \cup N(I')] \\ &\geq \frac{1}{2}\sum_{i\in V'} d(i)\cdot \Pr[i \in N(I')]\end{aligned}$$

Un lemme entre en jeu dans la suite de la démonstration. Nous n'en proposons pas de preuve mais elle peut être retrouvée en intégralité dans le papier de Luby [100].

Lemme 6.2. *$\forall i \in V'$ tel que $d(i) \geq 1$:*

$$Pr[i \in N(I')] \geq \frac{1}{4} min\left(\frac{sum(i)}{2}, 1\right)$$

Avec $sum(i) = \sum_{j\in N(i)} \frac{1}{d(j)}$.

Grâce à ce lemme, nous pouvons développer l'inégalité en effectuant une disjonction de cas sur la valeur de $sum(i)$.

$$\begin{aligned}\mathbb{E}[Y_k^B - Y_{k+1}^B] &\geq \frac{1}{8}\left(\sum_{\substack{i\in V' \\ sum(i)\leq 2}} d(i)\cdot\frac{sum(i)}{2} + \sum_{\substack{i\in V' \\ sum(i)>2}} d(i)\right) \\ &\geq \frac{1}{8}\left(\frac{1}{2}\sum_{\substack{i\in V' \\ sum(i)\leq 2}}\sum_{j\in N(i)}\frac{d(i)}{d(j)} + \sum_{\substack{i\in V' \\ sum(i)>2}}\sum_{j\in N(i)} 1\right)\end{aligned}$$

En effectuant la même disjonction de cas pour les nœuds voisins j, il est possible de transformer ces sommes sur les nœuds en des sommes sur les arêtes du graphe $G' = (V', E')$. En effectuant cette transformation, on ne compte plus qu'une fois la contribution de chaque nœud, ce qui entraîne la multiplication de certains termes par un facteur 2. En remarquant que toutes les arêtes du graphe sont décrites par la disjonction de cas et que le terme de chaque somme est supérieur à 1, on exprime finalement cette inégalité en terme du nombre d'arêtes.

$$\begin{aligned}\mathbb{E}[\Delta Y^B] &\geq \frac{1}{8}\left(\sum_{\substack{(i,j)\in E' \\ sum(i)\leq 2 \\ sum(j)\leq 2}} \frac{1}{2}\cdot\left(\frac{d(i)}{d(j)} + \frac{d(j)}{d(i)}\right) + \sum_{\substack{(i,j)\in E' \\ sum(i)\leq 2 \\ sum(j)>2}}\left(\frac{d(i)}{2\cdot d(j)} + 1\right) + \sum_{\substack{(i,j)\in E' \\ sum(i)>2 \\ sum(j)>2}} 2\right) \\ &\geq \frac{1}{8}|E'| = \frac{1}{8}Y_k^B\end{aligned}$$

□

6.3.4 Distribution des algorithmes de Luby

La transposition des algorithmes de Luby est réalisable au sein du paradigme Pregel. En effet, comme présenté en Section 6.3.1, les algorithmes parallèles de Luby se décomposent en un deux fonctions à appliquer en parallèle, une sur les nœuds et une sur les arêtes. Transposer ces fonctions dans le paradigme Pregel est aisé puisque Pregel suit déjà ces étapes lors de l'exécution des algorithmes de graphes.

La fonction ALGVERTEX, appliquée sur les nœuds en parallèle, revient à appliquer la fonction Map sur l'ensemble des nœuds. Cette étape est systématiquement réalisé après l'arrivée de message vers un nœud.

La fonction ALGEDGE n'existe pas directement puisqu'aucune opération n'est réalisée directement sur les arêtes. Cependant, sa transposition correspond à l'envoi de message le long des arêtes. Pour appliquer la fonction ALGEDGE, il est nécessaire de connaître les données de chaque nœud ainsi que les données des arêtes. Toutes ces informations sont transmises aux nœuds voisins par envoi de messages et la fonction ALGEDGE peut être directement appliquée à chaque nœud lors de la réception des messages.

Il n'y a pas de perte de scalabilité en utilisant cette transposition. L'opération de Map est effectuée en parallèle sur chaque nœud. Après l'envoi de messages, ceux-ci sont combinés en un seul message ce qui assure qu'il n'y ait pas de pertes de performances lors du Map à cause d'un nombre trop important de voisins. Cette combinaison des messages arrivant sur un nœud s'apparente à une opération de Reduce et ne modifie en rien le déroulement de l'algorithme.

Nous avons implémenté l'algorithme A de Luby [100] en Spark [139]. Le code de cet algorithme est présenté en Figure 6.1. Les performances cette implémentation sont présentées en Section 6.5.1.

Implémentation

Nous apportons une modification mineure au tirage du rang pour chacun des nœuds lors de la décomposition récursive. Le rang de chaque nœud est un entier **long** de 64-bits au lieu d'être tiré de manière uniforme dans l'intervalle $[\![1, |V|^4]\!]$.

En appliquant strictement l'algorithme de Luby, il ne serait pas possible de décomposer des graphes de $2^{\frac{64}{4}} = 2^{16}$ ou moins. Dans la pratique, cette approximation n'altère pas les performances de l'algorithme de MIS, d'autant plus qu'au bout de quelques itérations le nombre de nœuds restant dans le graphe a fortement diminué.

```
case object NOT_SELECTED extends Selection {}
case object SELECTED extends Selection {}
case object NEIGHBOUR_SELECTED extends Selection {}

def sendMessage(t: Triplet) = {
  (t.srcAttr._2, t.dstAttr._2) match {
    case (NOT_SELECTED, NOT_SELECTED) =>
      t.sendToSrc((t.dstAttr._1, false))
      t.sendToDst((t.srcAttr._1, false))
    case (NOT_SELECTED, SELECTED) =>
      t.sendToSrc((1.0, true))
    case (SELECTED, NOT_SELECTED) =>
      t.sendToDst((1.0, true))
    case (NOT_SELECTED, NEIGHBOUR_SELECTED) =>
      t.sendToSrc((1.0, false))
    case (NEIGHBOUR_SELECTED, NOT_SELECTED) =>
      t.sendToDst((1.0, false))
    case _ =>
  }
}

def mergeMessage( (seed1, has1), (seed2, has2) ) = {
  (Math.min(seed1,seed2) , has1||has2)
}

def  algVertex((myseed, mysel), (minSeed, selNei)) = {
  if((mysel == SELECTED)||(mysel == NEIGHBOUR_SELECTED))
    (myseed, mysel);
  else if (selNei)
    (myseed, NEIGHBOUR_SELECTED);
  else if (myseed < minSeed)
    (myseed, SELECTED);
  else
    (scala.util.Random.nextFloat(), mysel);
}

val mis = Pregel(
    graph, (0.0, false),maxIterations = Int.MaxValue)
  (algVertex(_,_), sendMessage(_), mergeMessage(_,_));
```

FIGURE 6.1 – Code Spark (Scala) pour l'algorithme A de Luby [100]

6.3.5 MultiGILA : Complexité de la décomposition récursive

Dans leur papier de 2016, Arleo et al. [9] présentent un algorithme de dessin multiéchelle distribué. L'algorithme de décomposition correspond à l'algorithme A de Luby [100] pour un MIS à distance 2. Arleo et al. ne proposent pas de complexité pour leur algorithme de décomposition. Nous proposons une analyse de leur algorithme au regard des résultats présentés par Luby.

Génération d'une permutation Arleo et al. utilisent les identifiants des nœuds comme rang dans la permutation au lieu d'en générer de nouveaux à chaque itération. Il n'y a donc pas de génération de nombre aléatoire dans leur version de l'algorithme A de Luby. Cela n'a pas d'incidence sur la complexité théorique de l'algorithme.

Dans la pratique, nous avons testé cette méthode et la méthode générant des rangs aléatoires aux nœuds à chaque itération converge significativement plus rapidement qu'en conservant les identifiants des nœuds comme rang tout au long de l'exécution.

Nous avançons l'hypothèse que lorsque les rangs sont conservés, il est plus difficile de trouver des nœuds ayant un rang minimal après la première ronde de sélection. Les nœuds non sélectionnés lors de cette ronde ne le sont pas à cause du rang de leur voisin, qui ne sont pas modifiés entre les rondes. Un nouveau tirage entre chaque itération permettrait d'éviter ces situations de blocage.

Nombre d'itérations La filtration proposée par FM^3 est séquentielle, mais elle peut être distribuée en utilisant l'algorithme de Luby, voir Section 6.3. Pour un clustering à distance 2, les nœuds sélectionnés doivent être à distance 3 ou plus les uns des autres. Il n'est donc pas suffisant qu'un nœud ait un rang inférieur à ses voisins, il faut aussi qu'il ait un rang inférieur aux voisins de ses voisins.

En utilisant l'algorithme A de Luby pour trouver un ensemble indépendant à distance 3, le nombre de rondes nécessaires en moyenne pour obtenir un MIS n'est pas modifiée. L'algorithme sélectionne moins de nœuds que pour un ensemble indépendant maximal à distance 2 mais supprime en retour plus de nœuds et d'arêtes. Il est possible de réécrire le début de la démonstration du Théorème 6.1 de la manière suivante :

$$\begin{aligned}\mathbb{E}[Y_k^B - Y_{k+1}^B] &\geq \frac{1}{2}\sum_{i\in V'} d(i)\cdot \Pr[i \in I' \cup N(I') \cup N(N(I'))] \\ &\geq \frac{1}{2}\sum_{i\in V'} d(i)\cdot \Pr[i \in N(I')]\end{aligned}$$

La borne inférieure obtenue ainsi n'est pas serrée : il y a intuitivement plus de nœuds dans l'ensemble des voisins à distance 2 que dans l'ensemble des voisins. Cependant, le reste de la démonstration est inchangé.

Nombre de messages Dans un environnement distribué où les données sont réparties sur plusieurs machines, chaque nœud doit connaître l'information de ses voisins à distances 2 pour pouvoir prendre une décision. Ceci est valable pour la sélection d'un sous-ensemble de nœuds indépendants, mais aussi pour créer le sous-graphe avec les poids d'arêtes correspondants et pour placer les nœuds après avoir dessiné une échelle. Du point de vue du nombre de messages transmis dans le paradigme Pregel, deux rondes sont nécessaires pour transmettre des données aux nœuds à distance 2. Lors de la première ronde, chaque nœud envoie son rang à chacun de ses voisins, soient $2 \cdot |E|$ messages envoyés au total. Dans la seconde ronde, chaque nœud retransmet les messages qu'il a reçus à chacun de ses voisins, soient $\sum_{v \in V_i} \deg(v)^2$ messages. Le nœud v retransmet $deg(v)$ messages à $deg(v)$ nœuds. Un papier de de Caen [34] propose des bornes inférieures et supérieures pour la somme du carré des degrés d'un graphe :

$$\frac{4|E|^2}{|V|} \leq \sum_{v \in V_i} \deg(v)^2 \leq |E|(\frac{2|E|}{|V|-1} + |V| - 2)$$

Le nombre total de messages est donc $O(\frac{|E|^2}{|V|} + |V||E|)$, ce qui ne passe pas bien à l'échelle. Dans MuGDAD, nous avons choisi de nous limiter à un clustering à distance 1 de manière à éviter les cas extrêmes et à simplifier les étapes de sélection d'un sous-ensemble de nœuds, de création du sous-graphe et de placement des nœuds.

6.4 MuGDAD : Algorithme de dessin multiéchelle

Dans cette section, nous revenons sur MuGDAD [76], notre algorithme de dessin multiéchelle adapté au paradigme MapReduce. La structure de MuGDAD, présentée dans l'Algorithme 11, suit la structure de GRIP et FM^3 présentée dans l'Algorithme 6. Cette structure est la même pour l'algorithme séquentiel et l'algorithme dans le paradigme MapReduce. L'algorithme 11 présente en commentaire les sections où sont détaillées chaque étape de l'algorithme.

Trois étapes sont utilisées pour créer une hiérarchie de graphes : la décomposition récursive(*cf.* Section 6.4.1), le calcul du poids des arêtes (*cf.* Section 6.4.2) et le placement initial des nœuds (*cf.* Section 6.4.3). Pour chacune de ces étapes, l'algorithme séquentiel et l'algorithme dans le paradigme MapReduce sont détaillés.

Algorithm 11 MuGDAD

```
G_0 = (V_0, E_0) ← Graphe initial                          ▷ Initialisation
i ← 0
while |V_i| > 3 do                                         ▷ Phase de sélection
    V_{i+1} ← Luby_A+Clustering(V_i)                       ▷ Section 6.4.1
    i ← i + 1
    Calcul du poids des arêtes E_i                         ▷ Section 6.4.2
    Création du graphe quotient G_i = (V_i, E_i)           ▷ Section 6.4.2
end while
N ← i
for j de N à 0 do                                          ▷ Phase de dessin
    if j < N then
        Placement des nœuds de G_{j-1}                     ▷ Section 6.4.3
    end if
    Calcul du dessin de G_j (quelques itérations)          ▷ Section 6.4.4
end for
```

Dans un dernier temps, nous détaillons les modifications réalisées sur notre algorithme de dessin distribué classique (*cf.* Chapitre 4) pour le rendre plus efficace dans en combinaison avec l'approche multiéchelle, *cf.* Section 6.4.4.

6.4.1 Sélection des nœuds et clustering

Détails de l'algorithme

La décomposition récursive utilisée dans MuGDAD [76] est similaire à celle proposée par FM^3 [63]. Cette décomposition associe un algorithme de sélection de nœuds indépendant avec un algorithme de clustering. Les clusters sont un sous-ensemble de nœuds du graphe. Le résultat de cette décomposition est un ensemble de clusters formant une partition des nœuds du graphe.

La décomposition en cluster permet de créer un graphe quotient qui correspond à l'échelle suivante dans la décomposition multiéchelle. Dans le graphe quotient, les clusters sont agrégés en meta-nœuds et les arêtes connectant des clusters différents sont agrégés en meta-arêtes.

Ensemble maximal indépendant Pour trouver un ensemble maximal indépendant (MIS), MuGDAD utilise l'algorithme parallèle A de Luby [100]. Nous avons choisi cet algorithme parmi ceux proposés par Luby car sa complexité moyenne de $O(\log |V|)$ est plus basse que celle proposée pour les autres algorithmes de Luby. Cette version génère plus de nombres aléatoires, ce qui n'est pas un problème dans notre cas. Cette version est aussi la plus simple à implémenter parmi toutes les versions proposées.

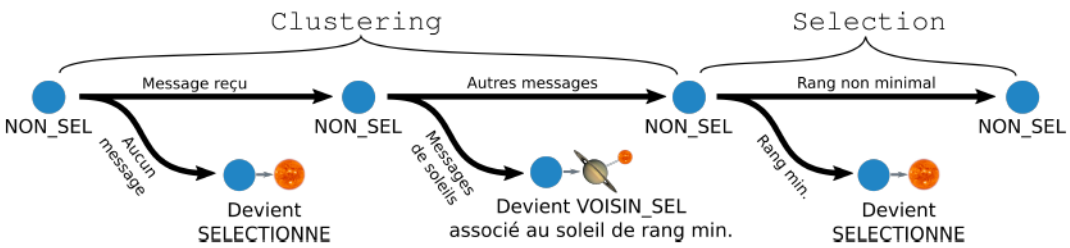


FIGURE 6.2 – Arbre de décision appliquée à chaque nœud non sélectionné à chaque ronde de Pregel.

A chaque itération de l'algorithme, un sous-ensemble indépendant de nœuds est sélectionné et le clustering est effectué à partir de ces nœuds.

MIS et clustering L'étape de formation des clusters est réalisée après chaque sélection d'un sous-ensemble de nœuds par l'algorithme de MIS. Le voisinage des nœuds sélectionnés est notifié de manière à créer les clusters. Cette étape remplace l'étape de suppression des nœuds du buffer dans les algorithmes de MIS, *cf.* Algorithme 7 : les nœuds restant dans le buffer sont les nœuds qui n'appartiennent à aucun cluster.

Lors de l'algorithme de sélection, les nœuds peuvent être dans trois états : non sélectionné (NON_SEL), sélectionné (SELECTIONNE) et voisin d'un nœud sélectionné (VOISIN_SEL). NON_SEL correspond à l'état des nœuds qui n'appartiennent à aucun cluster. SELECTIONNE correspond aux nœuds sélectionnés avec l'algorithme de MIS. VOISIN_SEL correspond aux nœuds voisins des nœuds sélectionnés. Pour les nœuds dans l'état VOIS_SEL, l'information de leur voisin de référence est conservée de manière à déterminer les clusters.

Dans un environnement séquentiel, lors de l'étape de clustering, les nœuds sélectionnés notifient leurs voisins de manière à créer les clusters. De nouveaux nœuds sont ensuite sélectionnés grâce à l'algorithme de Luby jusqu'à ce que tous les nœuds du graphe soient dans un cluster. Le changement d'état des nœuds dans le paradigme Pregel peut être réalisé sans distinguer l'étape de sélection des nœuds et l'étape de clustering, *cf.* Section suivante.

Algorithme dans le paradigme Pregel

Pour transposer la décomposition récursive dans un environnement distribué, le paradigme Pregel est le paradigme le plus adapté à la sélection d'un sous-ensemble de nœuds, *cf.* Section 6.3.4. Pour décrire les algorithmes de ce paradigme, il suffit de décrire l'algorithme que chaque nœud applique selon les messages qu'il reçoit et les messages qu'il transmet aux nœuds du graphe (généralement à ses voisins), *cf.* Section 2.2.2.

L'envoi de messages est réalisé uniquement à destination des nœuds non

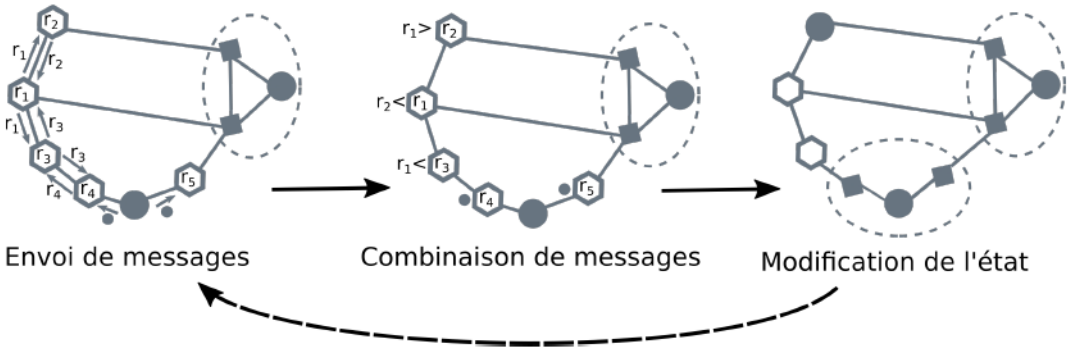


FIGURE 6.3 – Approche multiéchelle en Pregel. *(Gauche)* Envoi de messages aux nœuds non sélectionnés. Le type de message dépend de l'émetteur (soleil ou nœud non sélectionné). *(Milieu)* Combinaison des messages reçus. Les messages reçus sont réduits en un unique message. *(Droite)* Les nœuds modifient leur état en fonction du message reçu.

sélectionnés, les autres nœuds étant déjà dans un cluster. Deux types de messages sont transmis à chaque ronde de Pregel :

- entre nœuds non sélectionnés (NON_SEL) pour déterminer les nœuds du MIS selon l'algorithme de Luby. Les nœuds non sélectionnés transmettent à leur voisins leur identifiant ainsi que leur rang de manière à trouver les nœuds de rang minimal,
- entre nœuds sélectionnés (SELECTIONNE) et nœuds non sélectionnés (NON_-SEL) pour former des clusters. Ces messages contiennent l'identifiant du nœud sélectionné.

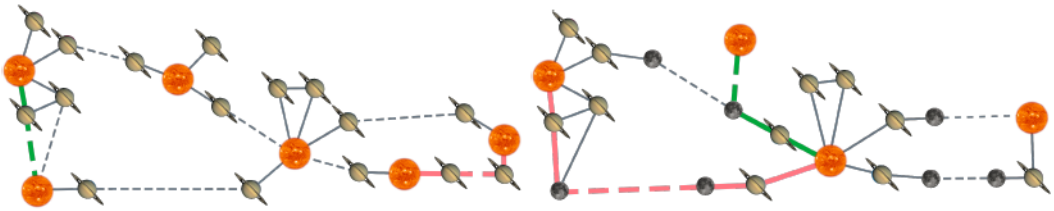
Parmi les différents messages qu'un nœud reçoit, l'étape de Reduce permet de sélectionner un unique message pour modifier l'état du nœud. La priorité est donnée aux messages provenant de nœuds sélectionnés, pour former un nouveau cluster. Si le nœud n'est pas déjà dans un cluster, il est agrégé avec son voisin sélectionné. Si un nœud non sélectionné ne reçoit aucun message de ce type, il compare son rang au rang minimal parmi ses voisins non-sélectionnés de manière à déterminer s'il change d'état, *cf.* Section 6.3.2. L'étape de Reduce est illustrée par l'arbre de décision en Figure 6.2.

Dans le paradigme Pregel et pour des clusters de nœuds voisins, le clustering et la sélection des nœuds sont réalisés lors de la même étape. Cette propriété ne peut être conservée pour un clustering à distance graphe de 2 ou plus, *cf.* Section suivante.

L'algorithme de sélection dans le paradigme Pregel est illustré en Figure 6.3.

Influence de la distance de MIS

Dans MuGDAD, à la différence de FM^3, l'ensemble maximal indépendant est formé à distance 2 (c'est-à-dire entre nœuds non voisins) et non pas 3.



(a) Distance 2 (b) Distance 3

FIGURE 6.4 – Décomposition en cluster basée sur un ensemble maximal indépendant. En vert, distance minimale pour le chemin le plus court entre deux soleils. En rouge, distance maximale pour le chemin le plus court entre deux soleils. Les nœuds sont représentés par des astres selon la métaphore proposée par FM^3 *(Gauche)* Ensemble maximal à distance 2. *(Droite)* Ensemble maximal à distance 3.

En effet, la décomposition récursive pour la version MapReduce de notre algorithme est en effet plus facile à mettre en œuvre à distance 2 et offre une complexité moindre, *cf.* Section 6.3.5. Nous détaillons ici les principales différences liées à la distance choisie pour le MIS en distinguant les MIS à distance 2 et les MIS à distance 3 ou plus.

Forme des clusters Les clusters obtenus en utilisant une décomposition à distance 2 ou 3 sont différents. La distance entre nœuds sélectionnés (soleils dans la métaphore de FM^3) pour des clusters voisins, *i.e.* pour lesquels il existe une arête connectant les deux clusters, permet, par exemple, de différencier les deux approches.

Soit u et v deux nœuds sélectionnés dont les clusters sont voisins. En utilisant un MIS classique, *i.e.* à distance 2, la distance graphe d_{uv} entre u et v est comprise entre 2 et 3. Dans le cas d'un MIS à distance 3, la distance graphe d_{uv} est comprise entre 3 et 5. Le clustering à partir d'un MIS à distance 3 permet donc d'obtenir moins de clusters que le clustering à partir d'un MIS classique et ces clusters contiennent plus de nœuds. Ces distances sont illustrées en Figure 6.4.

Graphe en étoile Le clustering à partir d'un MIS à distance 3 permet d'éviter un cas pathologique de la décomposition récursive : les graphes en étoile. Cette famille de graphes est constituée des arbres de profondeur 1 avec k feuilles. Pour ces graphes, lorsque $k > 1$, il y a un seul nœud dont le degré est strictement supérieur à 1.

Les algorithmes classiques pour trouver un MIS favorisent la sélection des feuilles de l'arbre. L'algorithme séquentiel classique a une probabilité de $\frac{|V|-1}{|V|}$ de sélectionner les feuilles de l'arbre pour créer le MIS. Les algorithmes de Luby favorisent aussi la sélection des feuilles (comparaison avec le rang des

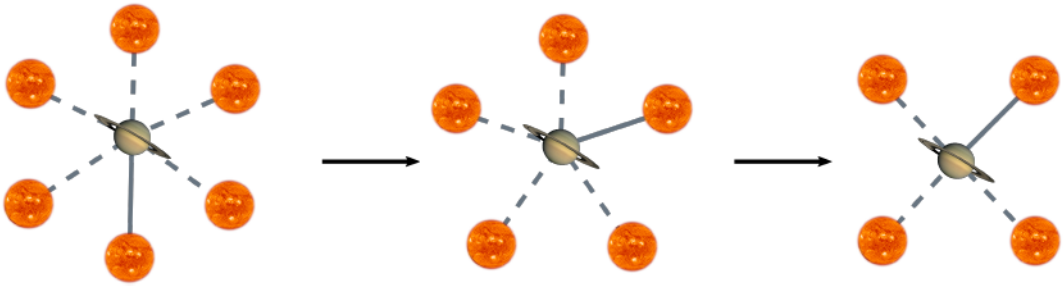


FIGURE 6.5 – Décomposition récursive d'un graphe en étoile par l'algorithme de décomposition de MuGDAD : cas pathologique.

voisins pour l'algorithme A et utilisation du degré du nœud pour l'algorithme B).

Le graphe créé à partir de ce MIS est alors lui aussi un graphe en étoile contenant $|V| - 1$ nœuds, comme illustré en Figure 6.5. La décomposition récursive est donc bloquée dans une boucle jusqu'à ce que le nœud central soit sélectionné. Le graphe est alors résumé à un unique nœud.

Clustering et distance Lorsque le clustering est effectué avec les nœuds à distance graphe 2 ou plus, il est nécessaire d'ajouter une étape de synchronisation entre la sélection. En effet, les nœuds ne disposent pas de l'état de leurs voisins à distance 2 ou plus lorsqu'ils prennent la décision de modifier son état. Il faut alors segmenter l'étape de sélection des nœuds et celle de création des clusters de manière à ce que les nœuds qui devraient être attribués à un cluster donné ne puissent devenir sélectionnés le temps que l'information leur parvienne.

Dans le cas particulier du clustering entre nœuds voisins, l'étape de sélection des nœuds et l'étape de création des clusters peuvent être réalisées sans nécessiter d'étape de synchronisation. Les nœuds sélectionnés et les nœuds non-sélectionnés transmettent leur état sous la forme de message. Pour les nœuds sélectionnés, ces messages visent à former des clusters. Pour les nœuds non-sélectionnés, ces messages cherchent à déterminer si ce nœud devient sélectionné ou non. Lors de l'étape de Reduce des messages, un arbre de décision permet de modifier l'état du nœud en conséquence.

6.4.2 Graphe quotient et poids des arêtes

Une fois la décomposition en clusters obtenue, le graphe suivant de l'approche multiéchelle est calculé. Dans le cas de MuGDAD, ce graphe est un graphe quotient obtenu grâce aux clusters. Les clusters sont agrégés en nœuds et les arêtes connectant deux clusters différents sont agrégés en arêtes. Les arêtes du graphes quotient sont pondérées de manière à ce que le dessin des

graphes quotients donnent de bonnes conditions initiales pour le dessin des graphes à l'échelle supérieure.

Détails de l'algorithme

Le calcul du graphe quotient suit le schéma proposé par FM^3 [63]. Le graphe quotient est calculé et le poids de ses arêtes est déterminé. Le poids des arêtes correspond au poids moyen des arêtes

Graphe quotient Une fois la décomposition en cluster obtenue pour un niveau, le graphe de l'échelle suivante est calculé. Pour MuGDAD, ce graphe correspond au graphe quotient induit par les clusters de nœuds. Les arêtes du graphe quotient sont obtenues en agrégeant les arêtes du graphe connectant des clusters différents.

Les clusters obtenus dans l'étape de filtration sont compressés en meta-nœuds, les nœuds du graphe correspondant à la prochaine échelle. De nouvelles meta-arêtes sont créées entre les clusters adjacents.

Poids des arêtes Lors de la simplification récursive du graphe original par des graphes quotients, la distance graphe entre les arêtes du graphe original est perdue. Le graphe quotient seul ne permet pas d'obtenir de bonnes conditions initiales pour le dessin suivant dans la décomposition multiéchelle. Les algorithmes de dessin multiéchelle associent donc un poids à chaque arête qui est une approximation de la distance graphe des nœuds dans le graphe original.

Pour le calcul du poids des arêtes, FM^3 utilise le chemin le plus court entre deux soleils voisins comme référence. Les soleils voisins sont les nœuds sélectionnés dont les clusters sont connectés. S'il existe un unique chemin le plus court (pour la distance graphe) entre deux soleils voisins, l'arête du graphe quotient a pour poids la somme des poids des arêtes du chemin.

Dans la plupart des cas, il existe plusieurs chemins de ce type. Deux cas de figure sont possibles :

- au sein d'un cluster, plusieurs chemins les plus courts existent vers un soleil,
- il existe plusieurs arêtes connectant les mêmes clusters.

Si plusieurs chemins existent, la moyenne du poids total de ces chemins est calculée pour déterminer le poids de l'arête du graphe quotient. Ces deux types de chemins sont représentés en Figure 6.6.

Pour calculer le poids d'une arête du graphe quotient, FM^3 part des arêtes du graphe quotient. Pour chaque arête du graphe quotient, les arêtes du graphe parent connectant les deux clusters (nœuds dans le graphe quotient) sont considérées. Pour chacune de ces arêtes, la longueur des chemins les plus courts entre les deux soleils empruntant cette arête sont évalués. Finalement, la moyenne

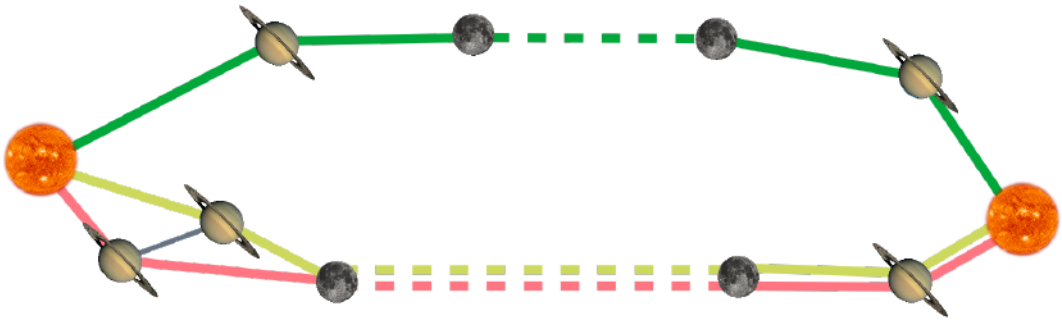


FIGURE 6.6 – Trois chemins les plus courts entre deux soleils. *(En bas)* Les chemins jaune et rouge illustrent deux chemins les plus courts passant par la même arête connectant les deux clusters. Au sein du cluster, il existe deux chemins possibles vers le soleil. *(En haut)* Autre chemin possible passant par une autre arête connectant les deux clusters.

du total de ces chemins est calculée, ce qui donne le poids de l'arête du graphe quotient.

Le calcul effectué par FM^3 prend en compte les chemins plus courts transitant par les arêtes du graphe parent connectant deux clusters et non pas seulement les chemins plus courts du point de vue de la distance graphe.

Influence de la distance

Dans MuGDAD, le MIS est calculé entre nœuds à distance graphe 2, contrairement à la distance 3 utilisée par FM^3. Pour la création du graphe quotient et le calcul du poids des arêtes, les calculs sont simplifiés. Nous détaillons ici les changements liés à la distance graphe utilisée pour le clustering.

Simplification des calculs Le choix d'effectuer le MIS à distance 2 permet de simplifier le calcul du poids des arêtes. Pour un MIS à distance 2 associé à un clustering avec les nœuds voisins, les arêtes connectant deux clusters différents sont associées à un unique chemin le plus court entre deux soleils. En effet, ces arêtes connectent deux planètes (VOIS_SEL) ou une planète (VOIS_SEL) et un soleil (SELECTIONNE). Il n'y a pas d'embranchements possibles au plus court chemin au sein du cluster en partant d'une planète.

La distance de clustering réduite permet d'accélérer le calcul du poids des arêtes, que ce soit dans un environnement séquentiel ou dans le paradigme MapReduce. Le calcul du poids des arêtes nécessite donc moins d'intermédiaires : les données nécessaires au calcul sont disponibles à une distance maximale de 1.

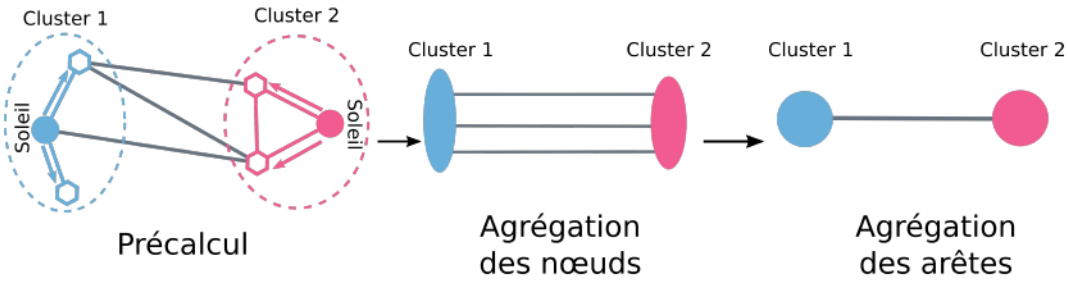


FIGURE 6.7 – Dans le paradigme MapReduce, le graphe quotient est calculé en trois étapes. *(Gauche)* Étape de préparation du calcul. *(Milieu)* Étape d'agrégation des nœuds. *(Droite)* Etape d'agrégation des arêtes.

Croissance exponentielle du poids Le calcul du poids des arêtes proposé par FM^3 ne peut pas être transposé directement à MuGDAD. Le calcul proposé par FM^3 crée des arêtes dont le poids est exponentiel avec le nombre de niveaux. Lorsque la décomposition récursive crée de nombreux niveaux, comme cela peut être le cas dans MuGDAD à cause de la décomposition à distance 1, le poids des arêtes n'a plus aucun rapport avec la distance graphe entre les nœuds.

A chaque niveau de la décomposition de FM^3, le poids de l'arête est multiplié par la distance graphe moyenne entre des systèmes solaires voisins. Plus précisément, en considérant le poids moyen des arêtes p_i au niveau i, une arête du graphe quotient a pour poids $k \cdot p_i$ avec k un coefficient correspondant à la distance entre les deux soleils, compris entre 3 et 5 dans le cas de FM^3.

La manière par laquelle FM^3 gère ce problème n'est pas claire : soit l'algorithme ne rencontre pas le problème grâce à son clustering à distance graphe 2, soit le poids des arêtes est mis à l'échelle entre chaque niveau pour assurer que le dessin est convenable.

Pour éviter cette croissance exponentielle, nous proposons une autre heuristique. Parmi les arêtes le long du chemin le plus court, le poids maximum est conservé. Les autres arêtes contribuent sous la forme d'un tirage aléatoire selon une loi uniforme dans l'intervalle $[\![1, i]\!]$, avec i le nombre d'échelles dans la décomposition multiéchelle. Cette heuristique permet d'assurer une continuité du poids des arêtes (qui est strictement croissant) et évite la croissance exponentielle.

Algorithme dans le paradigme MapReduce

Comme relevé dans la section précédente, le graphe quotient et le calcul du poids des arêtes peut être effectué en considérant les arêtes connectant deux clusters parmi l'ensemble des arêtes. Cette opération se prête au paradigme MapReduce : une opération de Map sur les arêtes du graphe parent permet d'effectuer le calcul du poids des arêtes connectant deux clusters et une opé-

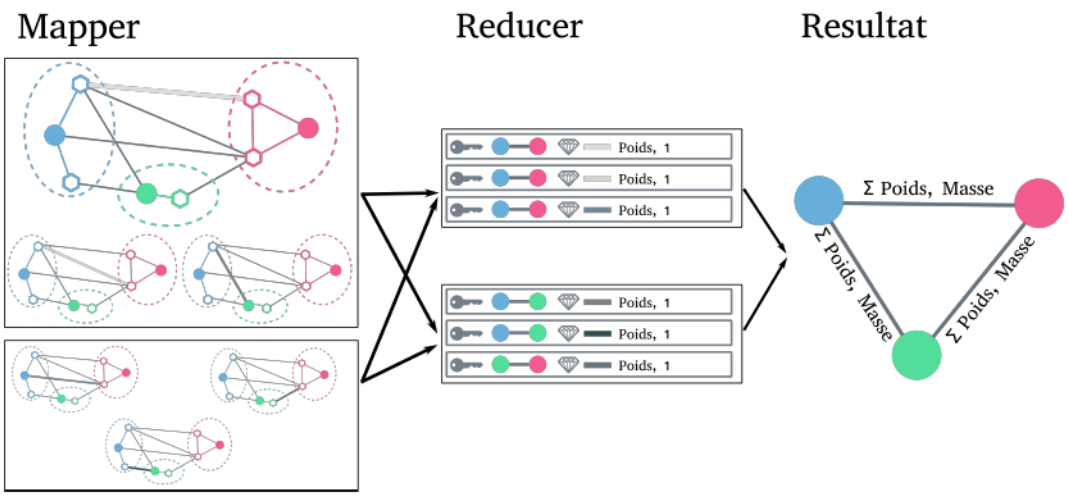


FIGURE 6.8 – Création d'une nouvelle échelle. *(Gauche)* Opération Map sur les arêtes. Chaque arête connectant deux clusters est transformée en une meta-arête dont la masse est calculée. *(Milieu)* Opération de Reduce sur les meta-arêtes. Les meta-arêtes connectant les même clusters sont fusionnées en une seule arête. *(Droite)* Résultat. Graphe final dont le poids des arêtes correspond à la moyenne du poids des meta-arêtes.

ration de Reduce permet de trouver le poids moyen des arêtes connectant les même clusters.

Le calcul du graphe quotient s'effectue en deux temps : une étape préparatoire au calcul et les opérations de MapReduce. Ces étapes sont illustrées en Figure 6.7.

Etape préparatoire au calcul Lors de l'étape de précalcul, le calcul du poids des arêtes est mis en place. Les données nécessaires à ce calcul sont envoyées aux planètes (VOIS_SEL) du cluster par le soleil (SELECTIONNE) du cluster. Ces données sont transmises par des messages envoyés le long des arêtes connectant les planètes au soleil d'un même cluster. Les messages contiennent le poids de l'arête connectant la planète à son soleil.

Lors de l'opération de Map, *cf.* Section suivante, le poids total d'une arête connectant deux clusters peut être calculé directement : ce poids total est la somme du poids de l'arête considérée et des arêtes constituant le chemin le plus court dont le poids a été transmis lors de cette étape de précalcul.

Algorithme MapReduce Le calcul du graphe quotient est réalisé grâce à une opération de MapReduce, dont la sortie consiste en un ensemble d'arêtes pondérées constituant le nouveau graphe. L'identifiant des nœuds de ces arêtes correspond à l'identifiant du soleil du cluster, de manière à initialiser la position du dessin facilement, *cf.* Section 6.4.3.

Une opération Map est appliquée sur les arêtes du graphe. Lors de cette opération, les nœuds du graphe quotient sont créés. Leur identifiant est celui du soleil du cluster correspondant. Cette opération renvoie l'ensemble des arêtes dont les extrémités se trouvent dans deux clusters différents. Le calcul du poids des arêtes intermédiaires est aussi effectué lors de l'étape de Map. L'ensemble des informations sont disponibles grâce à la phase préparatoire.

Enfin, l'opération de Reduce permet d'agréger les arêtes connectant les mêmes systèmes solaires. Par une opération de moyenne, les meta-arêtes ainsi dupliquées sont réduites en une seule arête. Les arêtes pondérées du graphe quotient sont obtenues en sortie de cette opération de Reduce. Pour calculer la moyenne, un patron de conception est utilisé (*cf.* Miner et Shook [106] par exemple).

Complexité Soit $G_i = (V_i, E_i)$ le graphe généré par l'approche multiéchelle au niveau i. Nous proposons d'observer la complexité de la création du graphe quotient lors des différentes opérations de l'algorithme MapReduce.

Lors de la phase préparatoire, jusqu'à $|E_i|$ messages sont transmis. En effet, les messages sont transmis au sein des clusters, des soleils vers les planètes. De plus, cette inégalité est une égalité lorsqu'il n'y a aucune arête entre les différents clusters, *i.e.* lorsqu'il n'y a qu'un cluster dans le graphe.

Lors de l'étape de Map, il y a au plus $|E_i|-1$ arêtes entre différents clusters. Cette inégalité est une égalité pour les graphes en étoile, *cf.* Section 6.4.1. Cette opération Map est divisée entre plusieurs mappers, chacun travaillant sur une partie des arêtes. La complexité totale de cette opération de Map est donc divisée par le nombre de mappers m, pour une complexité totale de $O(\frac{|E_i|}{m})$.

L'opération de Reduce est appliquée sur les $O(|E_i|)$ arêtes obtenues dans l'opération de Map. L'opération de Reduce est distribuée parmi les r reducers, pour une complexité totale de $O(\frac{|E_i|}{r})$.

6.4.3 Initialisation et placement intelligent

Une fois le dessin d'un niveau terminé, à l'aide de l'algorithme de dessin présenté en Chapitre 4, le dessin du graphe est utilisé comme initialisation du dessin du graphe précédent dans la décomposition multiéchelle. La position des nœuds présents dans les deux graphes, *i.e.* les nœuds sélectionnés (soleils), est propagée. Ensuite, l'algorithme procède à un placement intelligent des nœuds non placés (la plupart des planètes).

L'étape d'initialisation de la position des nœuds et celle de création du graphe quotient ont de nombreuses similarités. Du point de vue algorithmique, les mêmes étapes sont appliquées, en miroir, pour calculer le graphe quotient et pour placer les nœuds de manière intelligente.

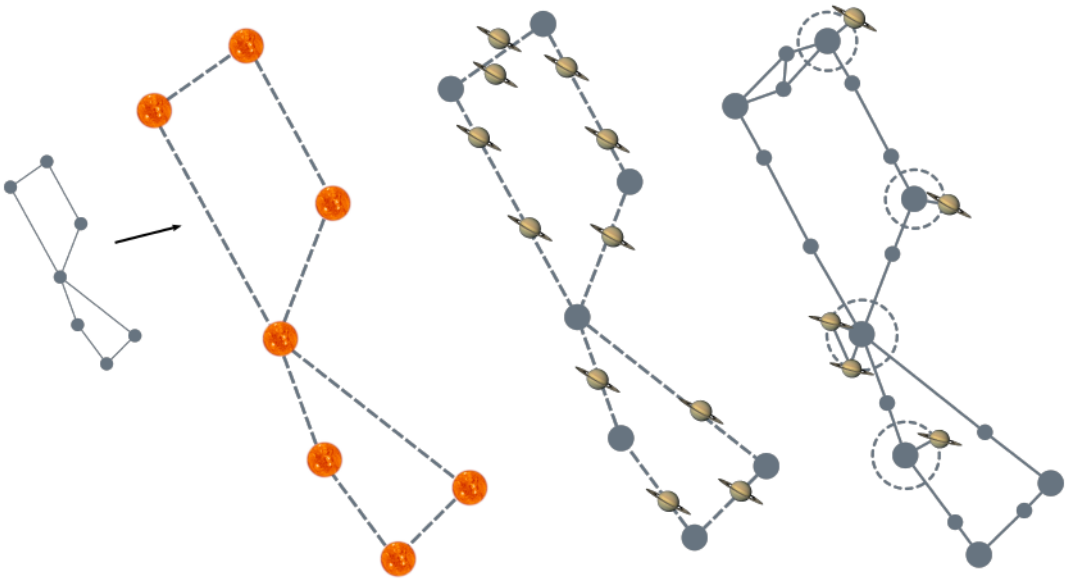


FIGURE 6.9 – Placement initial des nœuds d'un niveau. *(Gauche)* Propagation de la position des nœuds sélectionnés. La position des nœuds du graphe enfant, associés aux soleils du graphe parent, est conservée. *(Milieu)* Placement des planètes connectées. Les planètes étant sur le chemin connectant deux clusters sont placées de manière intelligente sur le segment connectant les deux soleils. *(Droite)* Placement des planètes isolées. Les planètes isolées sont placées en cercle autour de leur soleil.

Détails de l'algorithme

L'algorithme de placement est réalisé en trois étapes :

- la propagation de la position des soleils,
- le placement intelligent des planètes connectées,
- le placement en étoile des planètes isolées.

Chacune de ces étapes procède à un calcul différent pour trouver la position des nœuds non placés. Pour placer les soleils, le dessin du graphe précédent sert d'initialisation. Les planètes connectées sont interpolées grâce aux poids attribués aux arêtes, de manière à ce que ces nœuds se trouvent à une position proche de leur position finale. Enfin, pour les planètes isolées, il n'est pas possible d'effectuer une triangulation de la position optimale, comme réalisée lors du placement intelligent. Ces planètes isolées sont placées dans l'orbite de leur soleil. Ces trois étapes sont illustrées en Figure 6.9 et détaillées dans cette section.

Placement des soleils Les positions des nœuds du dessin de G_{i+1} sont propagées au graphe G_i. Les nœuds du graphe G_{i+1} correspondent aux nœuds

sélectionnés par l'algorithme de décomposition pour le nœud G_i. Leur position, optimisée par l'algorithme de dessin, est conservée. Ces positions servent de référence pour le placement des planètes, qu'elles soient connectées ou isolées.

En conservant les identifiants des nœuds lors de la création du graphe quotient, la position des soleils peut être propagée sans utiliser de structure de données intermédiaire. En effet, lorsque des identifiants différents sont utilisés pour les nœuds sélectionnés de G_i et les nœuds de G_{i+1}, il faut conserver chaque couple d'identifiants, par exemple en utilisant un tableau associant les nœuds sélectionnés de G_i aux nœuds de G_{i+1}.

Placement intelligent des planètes connectées Le processus de placement intelligent rappelle l'étape de calcul du poids des arêtes présentée dans la section précédente. Ces deux étapes sont inspirées par l'algorithme FM^3.

A partir de la position des soleils de G_i, la position des planètes connectées à d'autres systèmes solaires est interpolée par un processus de placement intelligent. Les arêtes connectant des clusters différents sont parcourues pour trouver les nœuds à placer. Leurs extrémités sont placées sur le segment connectant les deux soleils. Le calcul de la position de ces nœuds sur le segment est réalisé en utilisant le poids des arêtes.

On parcourt les arêtes connectant deux clusters différents. Pour le clustering à distance 1, ces arêtes définissent un unique chemin le plus court entre les deux soleils. Elles permettent donc de placer une ou deux planètes de manière intelligente.

Les nœuds sont placés sur le segment connectant les deux systèmes solaires. La position exacte sur ce segment est calculée grâce aux longueurs d'arêtes relatives : le nœud est placé à distance $\frac{l_0}{l_{tot}}$, où l_0 correspond au poids de l'arête connectant le nœud à son soleil et l_{tot} correspond à la somme des poids selon le chemin le plus court reliant les deux soleils.

Si un nœud est placé plusieurs fois, sa position finale correspond à la moyenne des positions calculées par le processus de placement intelligent.

Placement en étoile des planètes isolées Après le placement des soleils et le placement intelligent d'une partie des planètes, certains nœuds ne sont pas placés. Ces nœuds sont les planètes qui ne sont connectées qu'à un unique cluster. Ces planètes peuvent tout de même être connectées aux autres planètes de leur cluster, ce qui rend complexe l'interpolation.

L'interpolation de la position optimale de ces planètes à partir de la position d'un unique soleil n'est pas possible. L'interpolation utilisant la position interpolée des planètes connectées est complexe et repose sur un calcul lui-même approché. Nous privilégions une solution simple pour placer ces planètes isolées.

Les planètes isolées sont placées à proximité de leur soleil, en cercle. De cette manière, ces planètes sont proches de leur position finale dans le dessin.

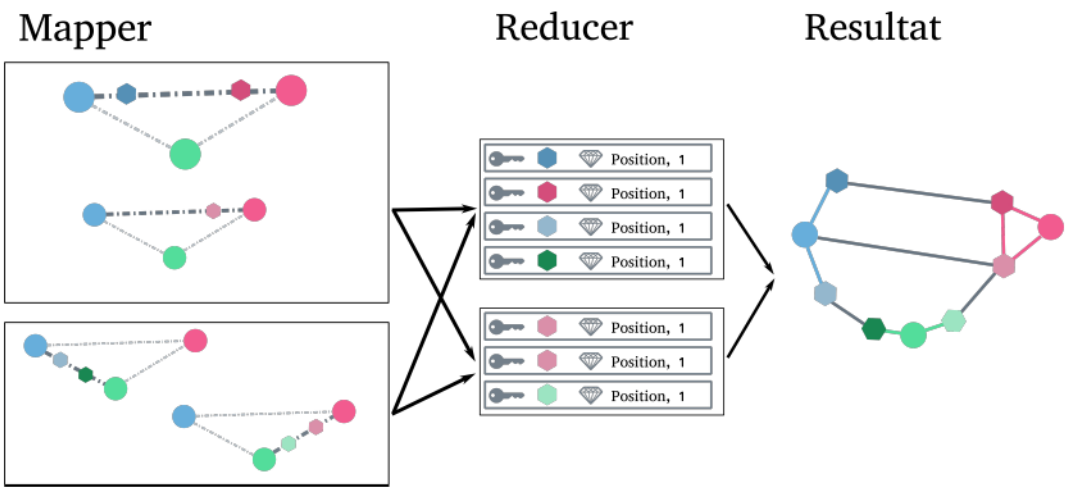


FIGURE 6.10 – Propagation des positions initiales pour trois clusters. *(Gauche)* Opération de Map sur les arêtes connectant deux clusters. Les nœuds sont positionnés sur le segment connectant les clusters *(Milieu)* Opération de Reduce. Les différentes positions générées pour un même nœud sont agrégées. *(Droite)* Résultat. Graphe final dont les planètes sont positionnées proches de leur position optimale.

C'est l'algorithme de dessin qui permettra d'optimiser leur position.

Algorithme MapReduce

Placement des soleils Pour transmettre la position des nœuds du graphe G_{i+1} au graphe G_i, nous utilisons une opération de jointure externe dans l'environnement distribué.

Plusieurs algorithmes existent pour effectuer la jointure dans un environnement MapReduce, *cf.* Chapitre 5 de Miner et Shook [106]. Nous utilisons une jointure du coté Reduce. Dans cet algorithme, l'ensemble des positions est réécrit sous la forme d'une paire clé-valeur et envoyé au reducers. Les valeurs ayant la même clef sont jointes par les reducers.

Cette algorithme de jointure est le plus simple à implémenter. L'étape de placement des soleils ne constitue pas un goulot d'étranglement de l'algorithme. Cette étape n'est appliquée que lors des changements de niveaux. De plus, grâce à la décomposition multiéchelle, seuls les premiers niveaux contiennent un volume important de données.

Etape préparatoire au placement des planètes Lors de l'étape de précalcul, des messages sont envoyés le long des arêtes connectant les soleils aux planètes d'un même cluster. Ces messages contiennent le poids de l'arête connectant les deux nœuds ainsi que la position du soleil.

Cette étape est réalisée grâce à une opération Map ou via une ronde de Pregel, déjà implémentée dans GraphX. Jusqu'à $|E_i|$ messages sont transmis de cette manière au niveau i.

Lors de l'étape préparatoire au placement des planètes connectés, l'ensemble des nœuds est placé en orbite de leur soleil. De cette manière, les planètes isolées sont déjà placées. Pour les planètes connectées, cette position sera mise à jour par le placement intelligent.

Placement des planètes connectées Pour placer les planètes connectant plusieurs systèmes solaires, une opération MapReduce est requise. Cette opération est illustrée en Figure 6.10.

Lors de l'opération Map, les arêtes connectant des clusters différents sont considérées. Cette opération renvoie les positions possibles des nœuds placés entre les clusters. L'opération Map renvoie une ou deux positions selon le nombre de planètes à ses extrémités. La position des soleils est utilisée pour déterminer les extrémités du segment sur lequel les nœuds sont placés. Le poids relatif des arêtes est calculé de manière à trouver la position exacte des nœuds sur ce segment.

Lors de l'opération Reduce, la position moyenne des nœuds est calculée pour les nœuds placés plusieurs fois lors de l'opération Map. Les positions multiples de chaque nœud sont réduites en une seule position. Pour calculer la moyenne, un patron de conception est utilisé (*cf.* Miner et Shook [106] par exemple) : lors de l'étape de Map, un compte, assigné à 1, est transmise en plus des coordonnées. Lors du Reduce, les coordonnées et le compte sont sommés séparément, de manière à obtenir la somme des coordonnées et le compte total. Lors d'une opération Map ultérieure, il sera possible de retrouver la position en divisant la somme des coordonnées par le compte (nombre de positions différentes du nœud).

Complexité Soit $G_i = (V_i, E_i)$ le graphe au niveau i. Le graphe G_{i+1} a été dessiné lors de l'étape de dessin et la position initiale des nœuds du graphe G_i est calculée. Nous proposons d'étudier la complexité du placement initial pour les différentes étapes de l'algorithme. La complexité du placement intelligent des planètes est similaire à celui du calcul du poids d'arêtes, comme nous allons le montrer.

Pour placer les soleils, nous utilisons un algorithme de jointure externe par Reduce. Dans cet algorithme, les lignes contenant la même clé sont envoyées au même reducer qui effectue l'opération de jointure. Le jeu de données en entrée est constitué de la position des nœuds dans le graphe G_{i+1} et des identifiants des nœuds G_i, soit donc $2|V_i| - 1$ nœuds dans le pire des cas (graphe en étoile). Il y a donc $O(|V_i|)$ positions transmises aux reducers utilisées pour appliquer la jointure.

Lors de la phase préparatoire, jusqu'à $|E_i|$ messages sont transmis. En effet, les messages sont transmis au sein des clusters, des soleils vers les planètes. Par rapport au calcul du poids des arêtes, les messages envoyés selon les arêtes contiennent une donnée additionnelle : la position des soleils. Lors de cette étape, la position des planètes isolées est calculée.

Lors de l'étape de Map, au plus $|E_i| - 1$ arêtes sont parcourues pour placer les nœuds. Cette inégalité est une égalité pour les graphe en étoile, *cf.* Section 6.4.1. Cette opération Map est divisée entre plusieurs mappers, chacun travaillant sur une partie des arêtes. La complexité totale de cette opération de Map est donc divisée par le nombre de mappers m, pour une complexité totale de $O(\frac{|E_i|}{m})$.

L'opération de Reduce est appliquée aux positions des nœuds générées par le mapper. L'opération Map génère une ou deux positions pour chaque arête connectant deux clusters. Il y a donc au plus $2|E_i|$ positions générées par l'opération Map. L'opération de Reduce est distribuée parmi les r reducers, pour une complexité totale de $O(\frac{|E_i|}{r})$.

6.4.4 Adaptations à l'algorithme de dessin

Le dessin initial est obtenu en combinant le dessin du niveau précédent avec un placement intelligent des nœuds. A chaque niveau de notre filtration, un algorithme de dessin par modèle de force est appliqué pour optimiser le dessin. Pour effectuer le dessin à chaque étape, nous choisissons notre algorithme de dessin en distribué (*cf.* Hinge et Auber [75]).

Dans cet algorithme, le calcul des forces de répulsion est effectué à l'aide de centroïdes représentant des clusters de nœuds dans le dessin. Ces centroïdes permettent de trouver un compromis entre une optimisation locale et globale de la position des nœuds. Cet algorithme permet de réduire la complexité des forces de répulsion classiques de $O(|V|^2)$ à $O(|V|)$. L'algorithme est décrit plus en détail dans le Chapitre 4.

Le choix de cet algorithme découle du fait qu'il peut se combiner de manière intéressante avec une approche multiéchelle.

Modifications de l'algorithme par centroïdes

Des modifications mineures ont été réalisées à l'algorithme de dessin de Hinge et Auber [75] pour prendre en compte le poids des arêtes. Le poids des arêtes représente la distance théorique entre les nœuds dans les algorithmes de dessin multiéchelles. Pour cela, les forces d'attraction sont proportionnelles à $|d_{att} - d_0|$, avec d_{att} la distance réelle entre les nœuds et d_0 le poids de l'arête. Pour contrebalancer cet effet, les forces de répulsion sont multipliées par le poids moyen des arêtes de manière à équilibrer les forces d'attraction et de répulsion.

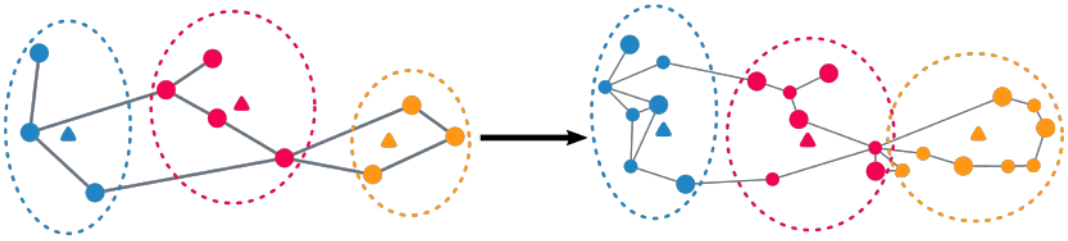


FIGURE 6.11 – Conservation des centroïdes après le placement initial des nœuds. La position des centroïdes est conservée. Leurs autres caractéristiques, et notamment les clusters, sont recalculés à partir du placement initial des nœuds du graphe.

Dessin par centroïdes et approche multiéchelle

Dans notre algorithme de dessin de graphe [75], *cf.* Chapitre 4, des clusters de nœuds sont formés à l'aide d'un processus similaire à l'algorithme des k-means. k clusters sont représentés par des centroïdes dont la position est au barycentre des nœuds du cluster. Ce sont ces centroïdes qui sont utilisés pour calculer les forces de répulsion approchées.

La position des centroïdes est conservée d'un niveau à l'autre dans l'approche multiéchelle. L'algorithme de clustering, une variante des k-means, converge vers une meilleure solution lorsqu'il est initialisé avec de bons centroïdes. Des algorithmes comme scalable k-means++ [13] ont montré qu'avec une bonne initialisation, l'algorithme des k-means converge vers une meilleure solution. Conserver les centroïdes entre chaque niveau assure une bonne initialisation au problème des k-means, ce qui améliore les forces de répulsion approchées.

Le dessin par centroïde est plus efficace lorsque les clusters sont bien définis, *cf.* Section 5.2.2. Dans MuGDAD, l'approche multiéchelle utilise un placement intelligent. Les dessins à chaque échelle sont donc déjà proches du dessin final ce qui veut dire qu'il est plus facile de former des clusters qui ont un sens du point de vue du dessin. Ce processus est illustré en Figure 6.11.

6.4.5 Algorithme de dessin multiparadigme

Le calcul distribué donne de moins bonnes performances qu'une implémentation séquentielle lorsque les données sont trop petites. Le surcoût de communication et de synchronisation est bien plus important que le gain obtenu par la distribution et le parallélisme. En utilisant un algorithme multiéchelle, le problème du dessin est simplifié jusqu'à pouvoir être résolu sur une seule machine. Il y a donc un seuil au-delà duquel l'approche multiéchelle devient inefficace dans un environnement distribué. Lorsque ce seuil est atteint, le calcul bascule sur une unique machine utilisant une version séquentielle de l'algorithme.

MuGDAD est un algorithme intéressant pour cette approche puisqu'il fonctionne de manière similaire dans les environnements séquentiel et distribué. En utilisant les forces par centroïdes dans les deux paradigmes, il est possible de conserver les centroïdes entre les deux approches, avec les mêmes avantages que la conservation des centroïdes présentée en Section 6.4.4. Lorsque l'implémentation séquentielle est terminée, le dessin et les clusters sont transmis à l'implémentation distribuée qui reprend le calcul là où l'approche séquentielle s'était arrêtée.

MuGDAD est implémentée dans deux versions différentes : une version séquentielle (parallélisée) développée en C++ et l'autre développée en utilisant Spark [139]. L'implémentation Spark de MuGDAD utilise l'implémentation C++ pour dessiner les graphes lorsqu'ils deviennent trop petits pour bénéficier de la distribution. Pour cela, nous avons utilisé l'interface Java native (Java Native Interface ou JNI) permettant d'exécuter un programme natif à travers une interface Java.

6.5 Résultats expérimentaux

6.5.1 Algorithme distribué de Luby pour le MIS

Dans cette section, nous présentons nos résultats expérimentaux concernant la décomposition en MIS d'un graphe dans un environnement distribué, *cf.* Section 6.3. Notre implémentation, *cf.* Section 6.3.4, est à notre connaissance la première implémentation Pregel de l'algorithme de Luby [100].

Dans un premier temps, nous confirmons la croissance logarithmique du nombre d'itérations nécessaires pour obtenir la décomposition. Puis, nous présentons les temps d'exécution de notre implémentation Spark de l'algorithme. Les résultats expérimentaux obtenus pour valider notre approche sont présentés en Figure 6.12.

Nombre d'itérations

Croissance logarithmique Avec l'algorithme de Luby, le nombre d'itérations moyen croît en $O(\log |V|)$. Notre implémentation distribuée de l'algorithme présente cette croissance. Le nombre d'itérations nécessaires pour obtenir le MIS sont présentés pour différents graphes en Figure 6.12.

Pour les graphes de Sierpinski, le nombre de nœuds est multiplié par 3 entre chaque niveau (décomposition des triangles en trois sous-triangles). Le nombre d'itérations nécessaires pour obtenir le MIS croît lentement. Pour les graphes ayant entre 300 et 3000 nœuds, 5 itérations sont nécessaires pour obtenir le MIS. Pour les graphes ayant entre 9000 et 270 000 nœuds, le MIS est obtenu en 6 itérations.

Graphe	Info. Graphes V	E	Temps Itération 1	Total	Nb. it.
Grands graphes					
amazon	334 863	1 851 744	2.14	20.11	7
Amazon0302	262 111	1 799 584	2.78	21.31	7
dblp	317 080	2 099 732	2.02	19.83	7
delaunay_n22	4 194 304	25 165 738	7.90	37.69	8
hugetric-00010	6 592 765	19 771 708	7.12	32.29	7
hugetric-00020	7 122 792	21 361 554	7.88	33.58	7
roadNet-PA	1 087 562	3 083 028	2.95	19.79	6
web-google	875 713	8 644 102	3.44	23.37	8
Graphes générés					
Sierpinski_3	15	54	0.38	6.68	3
Sierpinski_6	366	1458	0.46	10.53	5
Sierpinski_7	1095	4374	0.45	16.17	5
Sierpinski_8	3282	13 122	0.52	10.45	5
Sierpinski_9	9843	39 366	0.52	11.02	6
Sierpinski_10	29 526	118 098	0.69	9.96	6
Sierpinski_11	88 575	354 294	1.06	18.37	6
Sierpinski_12	265 722	1 062 882	1.80	21.53	6
Sierpinski_13	797 163	3 188 646	3.89	18.07	7
Sierpinski_14	2 391 486	9 565 938	5.91	25.46	7
Sierpinski_15	7 174 455	28 697 814	8.61	35.10	7

FIGURE 6.12 – Tableau récapitulatif des temps d'exécution de l'implémentation distribuée de l'algorithme de MIS, *cf.* Section 6.3.4. Les temps médians d'exécution sont donnés en seconde. Les temps relevés dans ce tableau sont le temps d'exécution de la première itération de l'algorithme et le temps total d'exécution de l'algorithme de MIS.

Lorsque le nombre de nœuds maximum est multiplié par 100, une itération supplémentaire est nécessaire pour obtenir le MIS. Ce phénomène se vérifie pour le reste des graphes présentés dans le tableau récapitulatif.

Limites de l'implémentation Dans notre implémentation, le tirage aléatoire d'une permutation des nœuds n'est pas garanti lorsqu'il y a un nombre trop important de nœuds. Au lieu de tirer une permutation (difficile en distribué), un rang aléatoire est attribué à chaque nœud utilisant un **Long**, encodé sur 64 bits. Les graphes décomposables de cette manière ont alors 66 000 nœuds au maximum, *cf.* Section 6.3.4.

Dans le tableau récapitulatif, en Figure 6.12, nous obtenons le MIS pour des graphes ayant bien plus de 66 000 nœuds. Malgré le fait que les rangs attribués aux nœuds ne forment pas une permutation, le nombre d'itérations reste raisonnable (7 itérations pour les graphes hugetric, 8 itérations pour le graphe delaunay *etc.*).

Temps d'exécution

Pour les temps d'exécution de l'implémentation, nous avons relevé le temps de calcul de la première itération, *i.e.* sur l'ensemble du graphe, et le temps de calcul total de la décomposition. Les temps relevés en Figure 6.12 correspondent au temps optimal et sont donc dépendants du nombre de machines.

Le temps de calcul de la première itération permet d'évaluer la scalabilité de notre implémentation. Lors de cet itération, l'ensemble du graphe est considéré pour calculer le MIS.

Le temps de calcul total permet d'évaluer le temps nécessaire au calcul du MIS en entier. Le temps de calcul total est dépendant du nombre d'itérations de l'algorithme de Luby. Plus un graphe contient de nœuds, plus il y a d'itérations pour obtenir le MIS.

Première itération et scalabilité Les temps d'exécution relevés lors de la première itération ne présentent pas de scalabilité horizontale. Lorsque les graphes sont plus grands, les temps d'exécution s'allongent. Cependant, cette croissance est sous-linéaire : lorsque le nombre de nœuds d'un graphe est multiplié par deux, le temps d'exécution n'est pas multiplié par deux. L'évolution des temps de calcul en fonction du nombre de nœuds est illustrée en Figure 6.13.

Comme nous l'avions observé lors de l'analyse de GDAD, *cf.* Section 5.5.1, le temps d'exécution semble suivre une croissance en $\sqrt{|V|}$. Cette corrélation est illustrée par la courbe de prévision en Figure 6.13. La corrélation entre les deux courbes a ainsi un coefficient de détermination $R^2 = 0.9773$.

Ces tests ont été effectués en utilisant principalement l'implémentation GraphX [138] de Pregel. Il est donc probable que ce phénomène provienne de l'implémentation de Pregel proposée par GraphX.

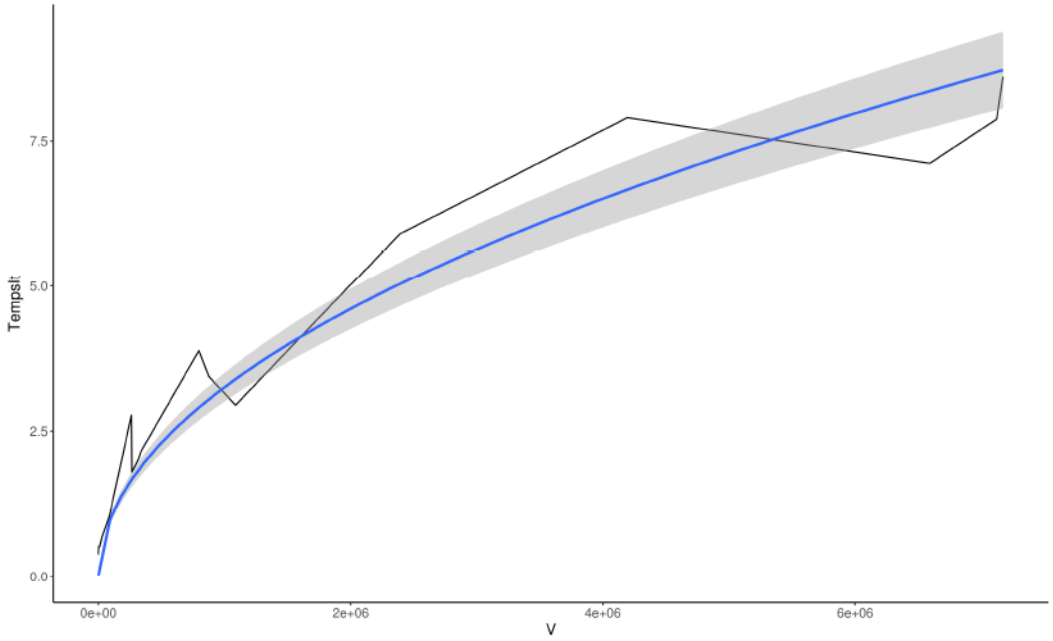


FIGURE 6.13 – Temps médian optimal de la première itération de l'algorithme de Luby en fonction du nombre de nœuds. Les temps sont exprimés en seconde.

Des papiers récents [58, 114] évaluent la scalabilité de GraphX en calculant le gain obtenu par le doublement du nombre de machines. Pour les très grands graphes, il est en effet difficile d'évaluer la performance optimale sur un cluster qui requiert un très grand nombre de machines pour être atteinte. Parmi ces analyses, Gonzalez [58] note que le passage de 8 à 32 machines entraîne un gain en temps d'un facteur 3 lors du calcul de PageRank sur le graphe de Twitter. Cependant, le passage de 8 à 64 machines entraîne un gain d'un facteur 3.5 seulement. Gonzalez note donc que la scalabilité offerte par GraphX n'est pas linéaire. Les analyses récentes des performances de GraphX n'ont cependant pas relevé la corrélation que nous avons observée lors de nos calculs.

Temps total Le temps total du calcul du MIS s'échelonne entre 10 et 30 secondes pour les graphes testés. Même si la scalabilité horizontale n'est pas atteinte, le calcul du MIS pour un niveau du graphe est réalisé en un temps raisonnable, quelle que soit la taille du graphe.

Les performances du temps total présentent des variations causées par plusieurs facteurs. Tout d'abord, le nombre d'itérations croît avec la taille du graphe, ce qui entraîne alors de plus nombreux calculs. De surcroît, après quelques itérations de l'algorithme de Luby [100], le nombre de nœuds actifs lors d'une itération de Pregel (*i.e.* transmettant des messages) a décru exponentiellement. Le coût de communication entraîne un ralentissement de l'algorithme. Le sous-graphe actif est suffisamment petit pour pouvoir être

	Info. Graphes		MuGDAD				MultiGila
Graphe	V	E	Filtration	Jni	Dessin	Total	Temps
amazon	334 863	1 851 744	67.29	123.47	113.37	304.13	7066.47
Amazon0302	262 111	1 799 584	30.57	59.02	41.64	131.22	-
dblp	317 080	2 099 732	36.79	261.51	37.45	335.75	-
delaunay_n22	4 194 304	25 165 738	198.26	338.29	1030.76	1567.31	10 720.32
hugetric-00010	6 592 765	19 771 708	190.14	511.73	1766.01	2467.88	12 590.93
hugetric-00020	7 122 792	21 361 554	222.54	413.16	2076.08	2711.78	13 599.51
roadNet-PA	1 087 562	3 083 028	66.84	504.59	296.35	867.77	11 869.80

FIGURE 6.14 – Tableau comparatif des performances des algorithmes MuGDAD [76] et MultiGila [9]. Les temps sont donnés en seconde. Pour MuGDAD, le temps de calcul est donné pour les différentes étapes de l'algorithme.

traité par une seule machine. Un nouveau partitionnement du graphe est cependant trop coûteux pour pouvoir être envisagé au cours de l'exécution de l'algorithme.

Ces optimisations de l'algorithme ne sont donc pas nécessaire pour notre application. L'algorithme de MIS est appelé une dizaine de fois au plus, pour chaque niveau de la décomposition multiéchelle, et prend au plus 30 secondes par niveau. Au regard du coût d'une itération, *cf.* Section 5.5.1, l'optimisation de ce calcul ne représente pas un gain important.

6.5.2 Comparaison avec MultiGILA

Dans cette section, nous présentons les temps de calcul obtenus pour MuGDAD [76] et MultiGila [9]. Pour MuGDAD, nous présentons de plus les temps de calcul de chaque étape de l'algorithme. Ces performances sont récapitulées en Figure 6.14.

Performances de MuGDAD

En Figure 6.14, nous présentons les performances optimales de l'implémentation Spark de MuGDAD. Les temps de calculs présentés sont ceux pour l'algorithme avec l'implémentation JNI. Lorsque la filtration génère un graphe de moins de 100 000 nœuds, l'implémentation distribuée de MuGDAD passe le relais à l'implémentation séquentielle grâce à JNI. Cette exécution est alors réalisée sur une seule machine.

Globalement, les performances distribuées obtenues avec MuGDAD+JNI sont bonnes. Pour les plus grands graphes (hugetric et delaunay), le dessin est obtenu en 35 minutes au plus. Cependant, *cf.* Figure 6.15 en section suivante, les performances distribuées de MuGDAD+JNI sont moins bonnes que les performances parallèles obtenues lors de l'exécution sur une seule machine. Le

gain apporté par MuGDAD+JNI reste à évaluer avec des graphes encore plus grands.

La scalabilité horizontale n'est probablement pas atteinte dans MuGDAD. Malgré le petit nombre d'itérations de GDAD appliquées sur le graphe complet, il est à noter que les performances distribuées de GDAD [75] ne présentaient pas de scalabilité horizontale, *cf.* Section 5.5.1.

Cependant, la scalabilité horizontale ne peut pas être envisagée pour les algorithmes multiéchelles. En effet, plus un graphe est grand et plus sa filtration contient de niveaux. Le nombre d'itérations réalisés à chaque niveau dépend, de plus, de la taille du graphe à dessiner et de sa profondeur dans la filtration. Ainsi, la filtration de hugetric-00010 contient environ 15 niveaux dont 4 sont de taille équivalente au graphe Amazon0302.

Performances détaillées Pour tous les graphes présentés, le calcul de la filtration est une étape rapide de l'exécution de MuGDAD. Pour les plus grands graphes (hugetric par exemple), cette étape prend moins de 5 minutes. Il est à noter que la filtration est réalisée sur les premiers niveaux puisqu'au delà, la filtration est gérée par JNI.

La phase de dessin est la phase la plus coûteuse de l'algorithme. Cette phase représente plus du tiers du temps d'exécution de l'algorithme. Pour les très grand graphes, l'algorithme passe beaucoup de temps à dessiner le graphe entier pour quelques itérations.

JNI sert à prolonger la filtration et le dessin pour des graphes trop petits à dessiner à l'aide d'une plateforme de calcul distribuée. Les temps que nous avons relevés ne dépassent pas les 10 minutes d'exécution.

Performances et JNI Le passage d'un algorithme distribué à un algorithme séquentiel lancé sur une seule machine constitue un réel gain de temps par rapport à une exécution uniquement distribuée. Ainsi, nous avons tenté de lancer notre algorithme distribué sans cette interconnexion. Il requiert alors plus de deux heures pour dessiner amazon, l'un des plus petits graphes présentés dans le tableau récapitulatif.

MuGDAD passe de nombreuses itérations à dessiner les plus petits graphes de la filtration. Ces itérations sont très coûteuses dans un environnement distribué disposant de plusieurs machines. Pour conserver des performances égales quel que soit le niveau de la filtration à dessiner, il faut adapter le nombre de machines à la taille du graphe.

Dans l'environnement distribué Spark [139], il est difficile de réduire le nombre de machines en cours d'exécution. Diminuer le nombre de partitions des petits graphes n'améliore pas non plus le nombre de performances, selon notre expérience. Le recours à JNI est donc une solution adaptée à notre approche.

Comparaison avec les performances de MultiGila

Nous avons mesuré les performances de MultiGila [9] sur notre cluster de calcul. Les temps de calculs observés, *cf.* Figure 6.14, sont supérieurs à ceux mesurés pour l'algorithme MuGDAD d'un facteur compris entre 5 et 10. Dans deux cas de figure, pour les graphes dblp et Amazon0302, le calcul n'aboutit pas. Le gain de performances de MuGDAD est en partie expliqué par l'utilisation de JNI, utilisation qui n'a pas d'équivalent dans l'algorithme d'Arleo et al.

L'écart de mesure entre les temps annoncés dans Arleo et al. [9] et ceux que nous avons relevé sur notre plateforme de calcul distribué est important. Nous avançons deux explications.

Débit du réseau dans le cluster Arleo et al. ont mené leurs tests de performances sur la plateforme de calcul distribuée de Amazon. Sur cette plateforme, le débit du réseau est généralement plus élevé (jusqu'à 25Go/s selon les machines choisies) que sur notre cluster de calcul local (1Go/s).

Une partie de cet écart peut donc venir du fait que notre plateforme ne dispose pas d'un débit permettant d'atteindre les performances annoncées par Arleo et al. Cette hypothèse est renforcée par le fait que Multi-Gila est un algorithme du paradigme TLAV (Pregel) et que le nombre de messages transmis peut être très important.

Voisinage et distance graphe Notre deuxième hypothèse est similaire à la remarque réalisée en Section 5.5.2. Dans Multi-Gila, la plupart des calculs sont réalisés à une distance-graphe égale à 3. Ainsi, les forces sont calculées à distance 3, et la filtration est réalisé à distance 2 (à la manière de FM^3 [63]). L'augmentation de la distance de recherche dans le graphe entraîne une explosion du nombre de voisins à considérer, *cf.* par exemple Section 5.2.3.

Pour les graphes réels roadNet-PA et amazon, Multi-Gila nécessite une centaine de *workers* pour terminer le calcul. Pour Amazon0302 et dblp, 200 *workers* ne suffisent pas à terminer le calcul. Or, comme nous l'avons relevé en Section 5.5.2, ces graphes réels ont une distribution de degré différente de celle des graphes générés. Il est donc probable que le dessin de ces graphes soit difficile pour des forces de répulsion à distance 2 ou plus.

6.5.3 Performances séquentielles et comparaison avec l'état de l'art

Dans cette section, nous présentons les performances de l'implémentation séquentielle de MuGDAD [76], notre algorithme de dessin multiéchelle. Ces performances sont comparées avec d'autres algorithmes de dessin rapides (mul-

tiéchelle ou algébrique) de l'état de l'art. Un récapitulatif des temps de calculs pour différents graphes est présentée en Figure 6.15.

Présentation des algorithmes

En Figure 6.15, nous présentons un tableau récapitulant les temps d'exécution de quatre algorithmes et de MuGDAD. Parmi les algorithmes de l'état de l'art, deux sont des algorithmes multiéchelle par modèle de force et deux sont des algorithmes algébriques. Nous présentons ici les algorithmes utilisés pour le comparatif.

FM^3 [63] est un algorithme de dessin multiéchelle présenté au début de ce Chapitre, *cf.* Section 6.2.3. FM^3 utilise une approximation des forces de répulsion basée sur la méthode multipôle rapide, *cf.* Section 3.3.2. Pour notre comparatif, l'implémentation OGDF [30] de FM^3 est utilisée.

SFDP est un algorithme de dessin multiéchelle basé sur le modèle de force de Fruchterman et Reingold [48]. Cet algorithme est disponible au sein du logiciel Graphviz [45].

ACE [88] est une méthode de dessin basée sur le calcul des vecteurs propres de la matrice d'adjacence. Cette méthode est présentée dans l'état de l'art, en Section 3.4.3.

HDE [68] (High Dimensionnal Embedding, *i.e.* dessin en haute dimension) est un algorithme de dessin de Harel et Koren. Dans cette méthode, le dessin de graphe est réalisé en haute dimension puis projeté en deux ou trois dimensions par l'algorithme de PCA (Principal Component Analysis) qui maximise la variance dans un nombre réduit de dimension. Le dessin du graphe est plus facile à réaliser dans un espace de grande dimension (typiquement 50) puisqu'il y a plus de degrés de libertés disponibles pour placer les nœuds de manière à respecter la distance graphe.

Comparaison des temps de calculs

Pour établir le comparatif présenté en Figure 6.15, les temps ont été obtenus à partir d'un ordinateur portable disposant d'un processeur Core i7-4710HQ, soit 2x4 cœurs hyperthreadés, et de 16 Go de RAM.

Algorithmes multiéchelles Par rapport aux deux algorithmes multiéchelle utilisés dans ce comparatif, MuGDAD [76] offre de meilleures performances d'un facteur 2 dans le pire des cas. Pour les très grands graphes, cet écart se creuse jusqu'à atteindre un gain de performance d'un facteur 10 (Sierpinsky_-14 ou delaunay_n22).

MuGDAD permet de dessiner des grands graphes que les autres algorithmes ne peuvent pas gérer. Par exemple, le dessin de Sierpinski_15 avec FM^3 renvoie une erreur *Mémoire insuffisante*. Pour les dessins de delaunay_n22, hugetric

Graphes	Info. Graphes V	E	Multiéchelle FM^3	SFDP	Algébrique ACE	HDE	MuGDAD
Petits graphes							
add32	4960	9462	1.70	1.92	0.04	0.01	0.49
bcsstk30	28 924	1 007 284	13.04	28.88	0.22	0.22	1.12
bcsstk31	35 586	572 913	13.76	28.42	0.30	0.21	1.00
bcsstk32	44 609	985 046	18.20	38.36	0.06	0.30	1.44
bcsstk33	8738	291 583	3.63	7.40	0.03	0.06	0.41
crack	10 240	30 380	2.93	4.32	0.10	0.03	0.41
fe_ocean	143 437	409 593	55.88	128.07	0.03	0.65	7.45
fe_pwt	36 463	144 794	11.19	20.77	0.11	0.11	0.94
finan512	74 752	261 120	26.51	55.54	0.26	0.27	3.06
Grands graphes							
Amazon0302	262 111	899 792	136.15	318.82	1.66	1.79	16.08
Com_DBLP	317 080	1 049 866	193.60	447.26	1.95	2.10	97.23
Com_Amazon	334 863	925 872	138.49	490.06	2.47	2.38	86.25
Roadnet_PA	1 087 562	1 541 514	642.84	1276.30	4.48	5.53	187.67
delaunay_n22	4 194 304	12 582 869	2213.34	>3600	29.97	28.27	229.21
hugetric-00010	6 592 765	9 885 854	3322.16	>3600	38.80	36.79	451.74
hugetric-00020	7 122 792	10 680 777	3752.59	>3600	39.47	36.92	394.98
Graphes générés							
sierpinski_8	3282	6561	0.74	0.82	0.01	0.01	0.12
sierpinski_9	9843	19 683	2.33	3.10	0.08	0.02	0.45
sierpinski_10	29 526	59 049	8.26	12.57	0.08	0.08	0.66
sierpinski_11	88 575	177 147	28.61	53.14	0.30	0.32	2.36
sierpinski_12	265 722	531 441	93.23	212.56	1.02	1.08	6.84
sierpinski_13	797 163	1 594 323	294.33	803.37	3.32	3.48	37.07
sierpinski_14	2 391 486	4 782 969	1136.43	3238.70	17.92	12.63	98.64
sierpinski_15	7 174 455	14 348 907	Mémoire	>3600	30.20	35.46	545.13

FIGURE 6.15 – Tableau comparatif des temps d'exécution de différents algorithmes de dessin exprimés en seconde. Les graphes sont classés en trois catégories de dessin : petits graphes, grands graphes et graphes générés. SFDP est un algorithme multiéchelle basé sur le modèle de force de Fruchterman et Reingold [48]. Il est implémenté dans GraphViz [45]. Pour FM^3, ACE et HDE, l'implémentation utilisée est celle d'OGDF [30].

et Sierpinski_15 avec SFDP, le temps de calcul total est supérieur à 1h (plus de 3h pour delaunay_n22).

Algorithmes algébriques En comparaison des algorithmes algébriques, MuGDAD prend dix fois plus de temps pour dessiner les graphes. Ce phénomène est constant pour les différentes tailles de graphes utilisées dans le comparatif.

Malgré leur efficacité, les algorithmes algébriques sont généralement considérés comme donnant des dessins déformés. En dehors des maillages, c'est à dire la plupart des graphes réels, les dessins obtenus par ces algorithmes sont éloignés de la structure du graphe. Ce phénomène est aussi observé pour les très grands graphes, qui sont pourtant généralement issus de maillages. Pour s'en convaincre, un comparatif visuel de dessins obtenus avec les différents algorithmes est disponible en Figure 6.16.

6.5.4 Comparaison visuelle des algorithmes de dessin

Présentation des graphes

finan_512 est un graphe de 74 752 nœuds et 261 120 arêtes. fe_ocean. est un graphe de 143 437 nœuds et 409 593 arêtes. Hachul et Jünger [64] classifient ces deux graphes comme graphe réel simple à dessiner.

Les deux graphes amazon et Amazon0302 proviennent des produits conseillés automatiquement par le vendeur en ligne Amazon. Lorsqu'un produit est conseillé sur la page d'un autre produit, une arête est créée dans le graphe. Ces deux graphes sont disponibles sur la plateforme d'analyse de graphe SNAP [97] de l'université de Stanford.

Le graphe roadNet-PA représente le réseau routier de Pennsylvanie. Les intersections et extrémités sont représentées par des nœuds. Les routes sont modélisées par des arêtes entre les nœuds. Ce graphe est disponible sur la plateforme d'analyse de graphe SNAP [97].

Le graphe delaunay_n22 est générée par une triangulation de Delaunay à partir de points tirés aléatoirement dans $[0, 1]^2$. Cette méthode est présentée en 2010 par Holtgrewe et al. [78].

Les graphes hugetric (10 et 20) sont générés selon la méthode présentée par Marquardt et Schamberger [103]. Les graphes présentés sont les maillages dynamiques. Les deux graphes représentent l'évolution après 10 et 20 itérations.

Dessins et algorithmes algébriques

En Figure 6.16, nous présentons une comparaison visuelle du dessin du graphe roadNet-PA par les algorithmes HDE, ACE (algébriques), MuGDAD et FM^3 (multiéchelles par modèle de forces). Malgré leur rapidité, les algorithmes algébriques ne permettent pas de distinguer clairement la structure du graphe

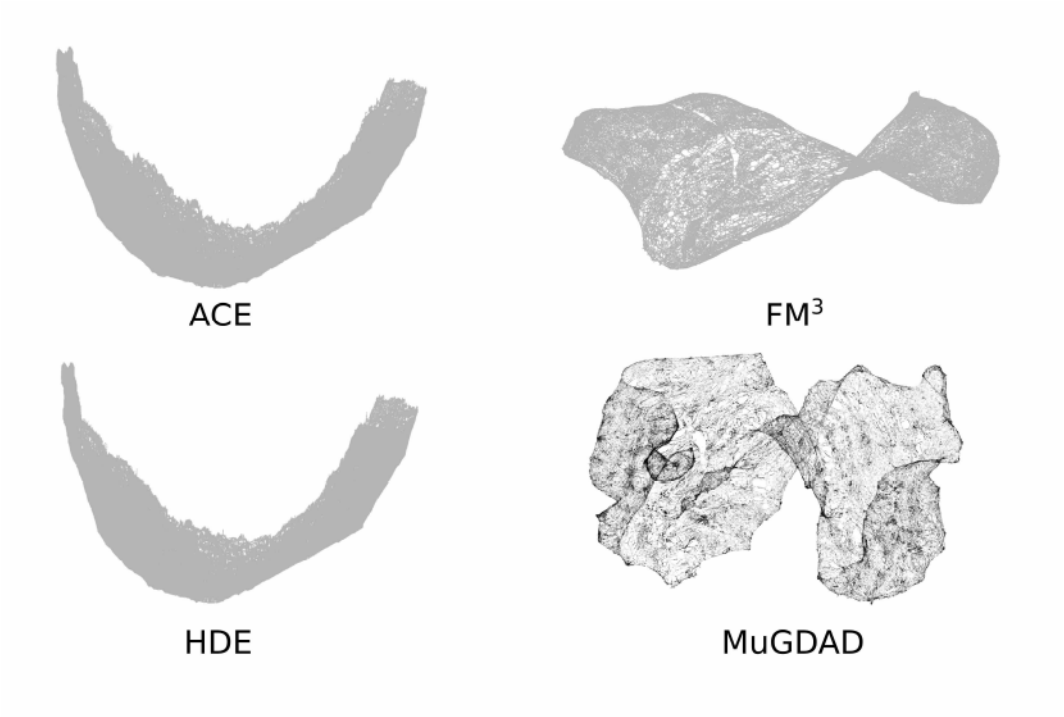


FIGURE 6.16 – Comparaison visuelle du dessin de roadNet-PA obtenu par ACE, HDE *(Gauche)*, MuGDAD et FM^3 *(Droite)*. Les dessins obtenus avec les méthodes algébriques (ACE et HDE) ne permettent pas de distinguer correctement la structure du graphe, au contraire des dessins obtenus avec MuGDAD ou FM^3.

roadNet-PA. Le dessin du graphe Amazon0302 permet d'arriver à la même conclusion.

Pour dessiner les graphes réels, les algorithmes HDE et ACE ne permettent pas d'obtenir un dessin convenable. Pour les maillages, ces algorithmes sont très efficaces. Par exemple, le dessin de hugetric par ACE est très similaire au dessin obtenu avec FM^3 en un centième du temps d'exécution. Ces algorithmes peuvent donc être utilisés pour dessiner de très grands graphes dans quelques cas particuliers mais ne permettent pas d'obtenir des résultats satisfaisants en général.

Approche multiéchelle et graphes structurés

En Figure 6.17 nous présentons le dessin de trois graphes structurés pour lesquels le dessin avec GDAD [75] est difficile, *cf.* Figure 5.16. Pour ces graphes, le dessin avec GDAD ne parvient pas à un état d'équilibre après de très nombreuses itérations.

Pour le graphe finan512, GDAD permettait de trouver les corolles mais pas la structure globale, en cercle, du dessin. L'approche multiéchelle utilisée dans MuGDAD permet de faire apparaître cette structure globale. Cette structure est clairement visible avec l'algorithme multiéchelle FM^3. Par rapport au dessin obtenu avec FM^3, certaines corolles sont alternées (intérieur ou extérieur). Cela n'a cependant pas d'incidence sur la lisibilité du dessin.

Le dessin du triangle de Sierpinski amène à une analyse similaire. Le dessin obtenu avec MuGDAD représente correctement la structure globale du dessin, ce qui n'avait pas été obtenu avec GDAD. Dans le dessin obtenu avec MuGDAD, certains triangles sont déformés, mais bien placés relativement à leurs voisins. La lisibilité du dessin n'est donc pas affectée.

Pour le dessin de fe_ocean, la structure du graphe est parfaitement visible avec l'algorithme MuGDAD. Les dessins obtenus avec FM^3 et MuGDAD sont très similaires.

Dessin de très grands graphes

Graphes générés et maillages En Figure 6.18, nous présentons des comparaisons visuelles pour trois jeux de données simulés. delaunay_n22 est généré à partir d'une triangulation de delaunay et les graphes hugetric représentent l'évolution temporelle d'un maillage. Les dessins obtenus avec FM^3 et MuGDAD sont proches.

Pour le dessin de delaunay_n22, la forme globale du dessin est similaire pour les algorithmes FM^3 et MuGDAD. Dans le dessin de MuGDAD, il est possible de distinguer un cercle à l'intérieur du dessin qui est présent dans la structure du graphe. Cette structure est représentée dans le dessin de FM^3 mais le rendu représenté ici ne permet pas de distinguer cette structure.

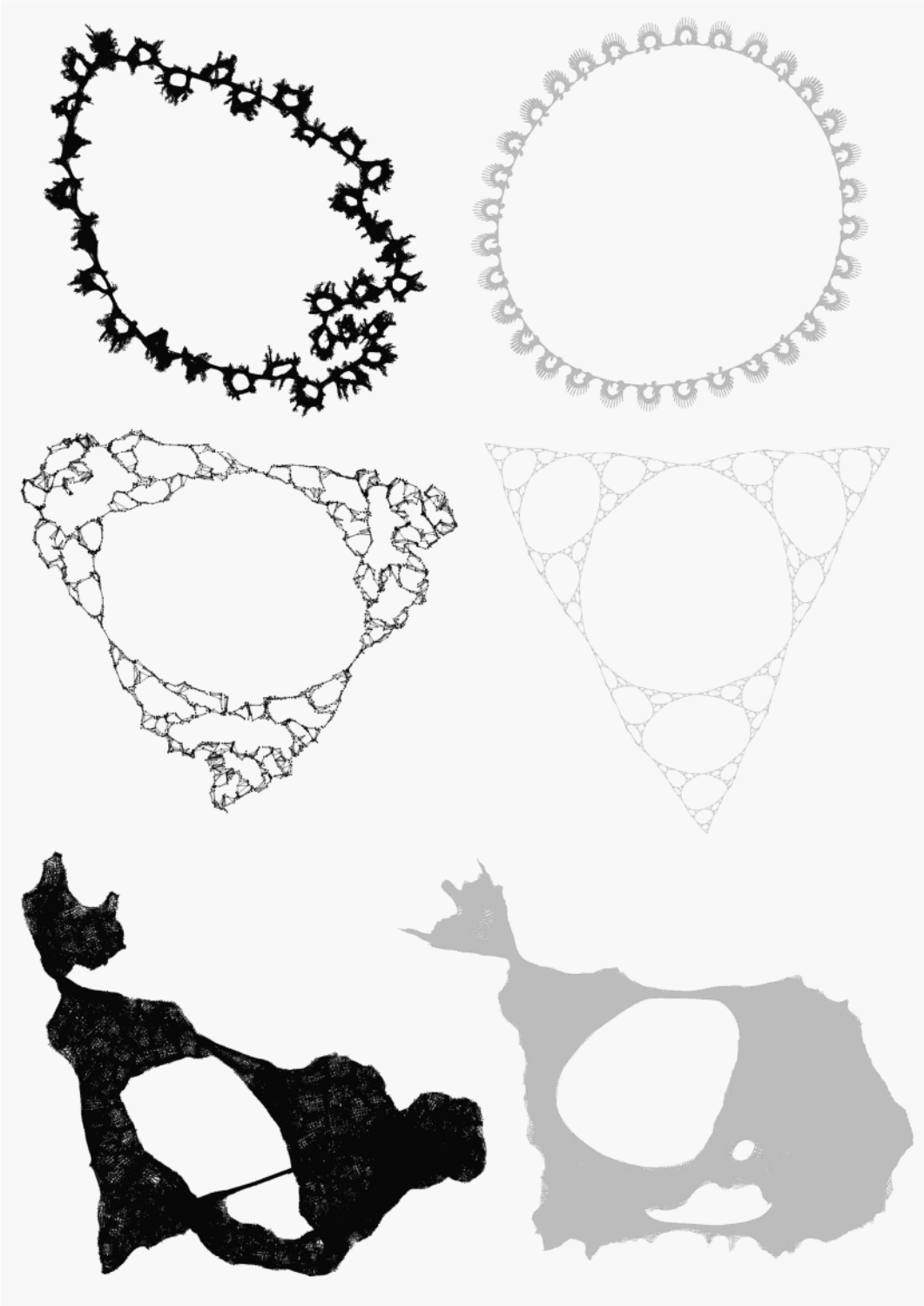


FIGURE 6.17 – Comparaison visuelle de dessins obtenus par MuGDAD *(Gauche)* et FM^3 *(Droite)*. *(Haut)* Dessin de finan512. *(Milieu)* Dessin d'un triangle de Sierpinski (8 niveaux). *(Bas)* Dessin de fe_ocean.

Les dessins des graphes hugetric avec MuGDAD et FM^3 présentent plus de différences. Le pourtour des dessins est plus différent entre les deux approches. Cependant, il est possible de distinguer la boucle formée par le graphe dans les deux cas de dessin. Pour le dessin de hugetric-00020 par MuGDAD, la boucle (représentée en bas du dessin) n'est pas clairement formée. Cependant, elle est clairement visible dans le dessin de hugetric-00010.

Graphes Réels En Figure 6.19, nous présentons des comparaisons visuelles pour deux jeux de données provenant d'Amazon et pour le réseau routier de Pennsylvanie. Pour ces graphes réels, les dessins obtenus avec FM^3 et MuGDAD sont proches.

Pour les jeux de données provenant d'Amazon, les graphes sont constitués de 262 111 nœuds (respectivement 334 863 nœuds) et de trois fois plus d'arêtes. Il est difficile de distinguer la structure intérieure du graphe. Cependant, au bord des dessins, nous retrouvons des structures périphériques similaires dans les deux dessins, qui représentent probablement des communautés bien séparées du cœur du graphe.

Pour le dessin du réseau routier de Pennsylvanie avec FM^3 et MuGDAD, il n'est pas possible de superposer exactement la carte de Pennsylvanie avec le dessin des routes obtenu. Il est possible d'observer une torsion dans les deux dessins ainsi qu'un repliement au bord du dessin de MuGDAD. Contrairement aux graphes simulés, la densité des routes en Pennsylvanie n'est pas partout la même, ce qui induit des déformations dans le dessin du graphe correspondant. Notamment, au niveau des villes les plus importantes, Pittsburgh à l'est et Philadelphie à l'ouest, le réseau routier est plus important, ce qui pourrait expliquer les déformations du dessin.

Artefacts de visualisation Pour les dessins générés par MuGDAD, il est possible d'identifier les clusters de nœuds appartenant à un centroïde directement dans le dessin. Des artefacts de visualisation apparaissent ainsi avec des zones plus denses (représentées en noir) dans le dessin et des zones plus claires (représentées en gris). Ces zones correspondent respectivement au centre et au bord des clusters. Ces artefacts sont particulièrement visibles dans les dessins de maillages, *cf.* Figure 6.18.

Ces artefacts n'apparaissent pas dans les dessins obtenus avec GDAD. Dans GDAD, de très nombreuses itérations sont requises pour atteindre un état de convergence. Dans MuGDAD, lors du dernier niveau de la décomposition, nous effectuons 5 itérations de dessin. Le coût d'une itération du dessin est en effet coûteux pour le dessin du graphe entier. Il est donc possible que ces itérations ne soient pas suffisantes pour faire disparaître ces artefacts.

La présence de ces artefacts n'a pas d'influence notable sur la visualisation de la structure globale du graphe. Localement, le dessin est déformé (plus

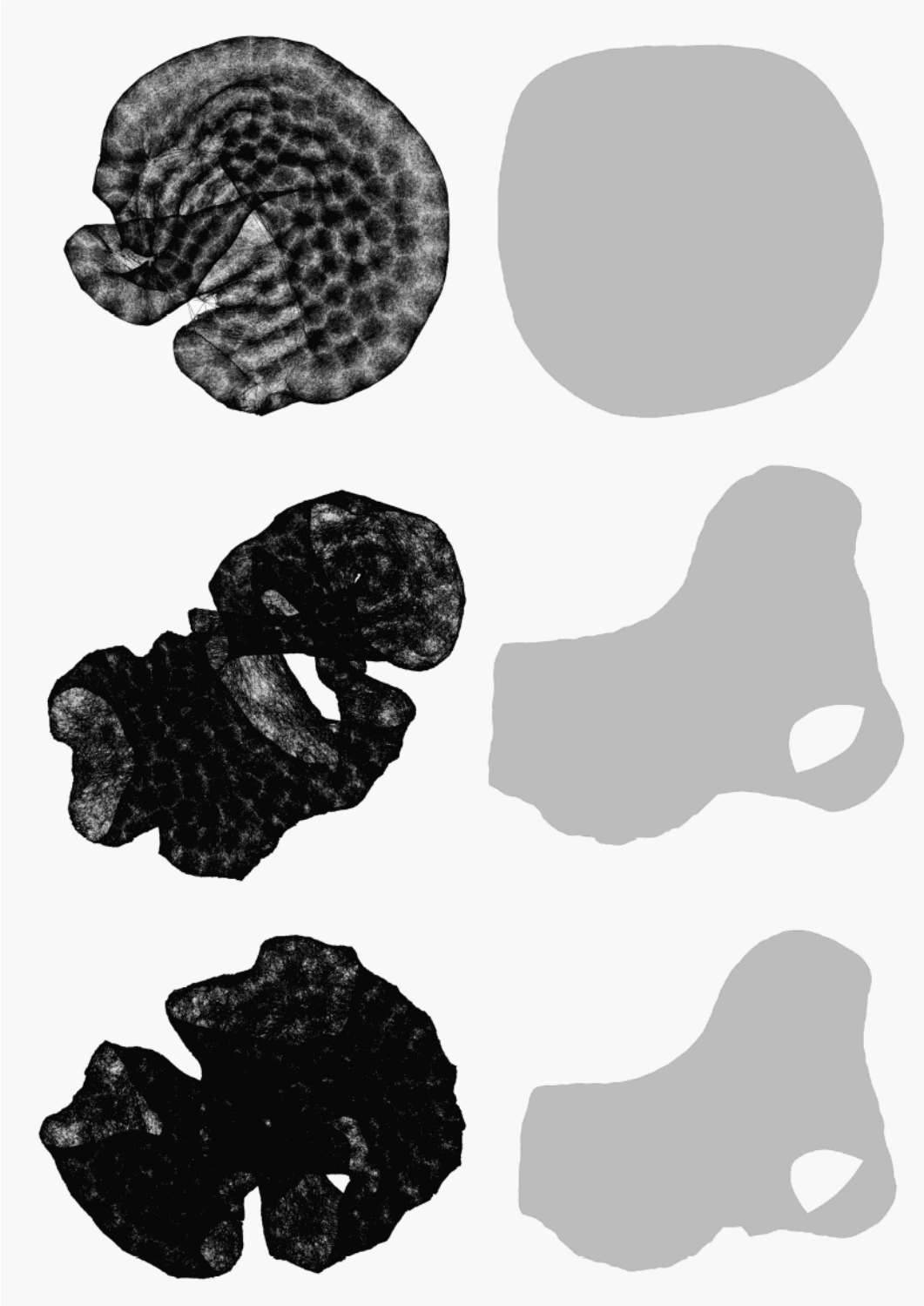


FIGURE 6.18 – Comparaison visuelle de dessins obtenus par MuGDAD *(Gauche)* et FM^3 *(Droite)*. *(Haut)* Dessin de delaunay_n22. *(Milieu)* Dessin de hugetric-00010. *(Bas)* Dessin de hugetric-00020.

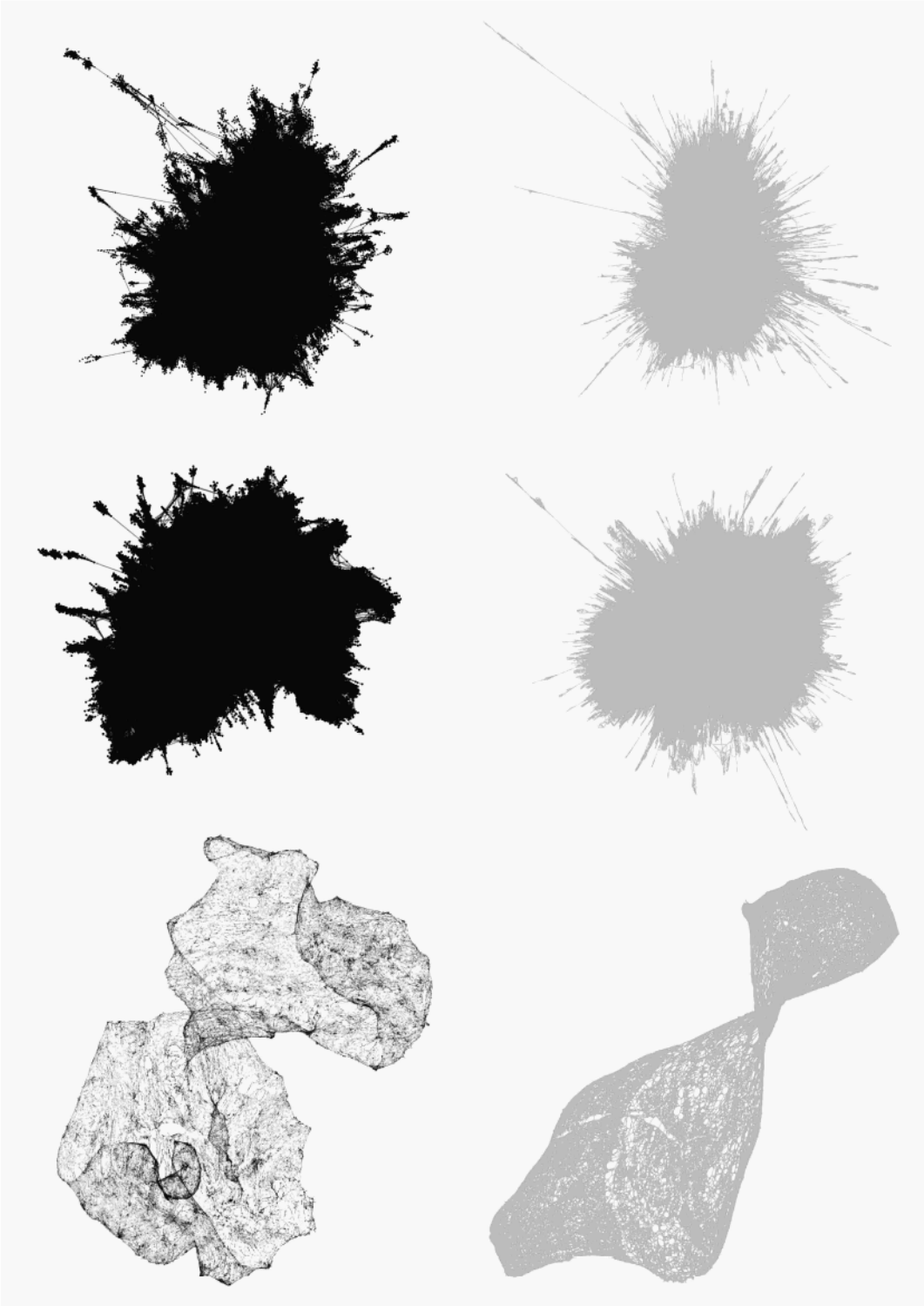


FIGURE 6.19 – Comparaison visuelle de dessins obtenus par MuGDAD *(Gauche)* et FM^3 *(Droite)*. *(Haut)* Dessin de amazon. *(Milieu)* Dessin d'Amazon0302. *(Bas)* Dessin de RoadNet-PA.

concentrée autour des centroïdes) mais la position relative des nœuds semble ne pas être affectée par cette déformation.

Il est envisageable de corriger localement la position des nœuds lors d'une interaction avec le dessin permettant d'accéder aux détails du dessin. Une correction réalisée par le client de visualisation pourrait permettre de faire disparaître ces artefacts. L'effet fisheye décrit par Sarkar et Brown [120], permet par exemple de déformer l'espace afin d'accéder aux détails du dessin. Une déformation inverse pour effacer la compression des centroïdes pourrait ainsi être envisagée. Herman et al. [73] décrivent plus en détail ces algorithmes de déformations dans leur état de l'art sur la visualisation de graphes.

6.5.5 Conclusion

Implémentation distribuée

MuGDAD+JNI est un algorithme de dessin efficace dans un environnement distribué. Les performances temporelles obtenues avec MuGDAD+JNI sont meilleures que celles de Multi-Gila [9] sur notre cluster de calcul.

Le gain de performances est en partie lié au fait que nous utilisons JNI pour effectuer le dessin des petits graphes. L'utilisation de JNI permet ainsi de passer peu de temps à dessiner les petits graphes, une opération très coûteuse dans un environnement distribué.

MuGDAD+JNI présente des temps de calcul supérieurs à ceux obtenus avec la version séquentielle de MuGDAD. MuGDAD+JNI est donc utile au sein d'une application de visualisation réalisée uniquement sur cluster. Le gain pour des graphes plus grands que ceux étudiés ici reste à prouver pour MuGDAD+JNI.

Implémentation séquentielle

MuGDAD, au vue des performances séquentielles de l'algorithme, peut être vu comme un entre-deux entre les algorithmes algébriques et les algorithmes de dessin par modèle de forces. MuGDAD permet d'obtenir rapidement l'aperçu d'un graphe en un temps inférieur à FM^3 pour une qualité de dessin supérieure à HDE.

Les temps d'exécution de MuGDAD sont compris entre ceux des algorithmes algébriques (très rapides) et des algorithmes de dessin multiéchelles (rapides). MuGDAD est 10 fois plus lent que ACE ou HDE et 2 à 10 fois plus rapide que FM^3. MuGDAD permet aussi de dessiner des graphes que FM^3 ou SFDP ne permettent pas de dessiner dans un temps raisonnable (ici une heure). Ces résultats sont détaillées en Section 6.5.3.

Du point de vue du dessin, la visualisation obtenue avec MuGDAD est moins précise que celle obtenue par FM^3. L'approximation des forces de répulsion utilisée dans MuGDAD ne garantit pas de borne d'erreur comme peut le

faire la méthode multipôle rapide utilisée dans FM^3, ce qui explique la qualité des dessins générés par FM^3. Cependant, la structure générale et les détails sont discernables. Le dessin obtenu ne déforme pas le graphe, comme cela peut être le cas pour les algorithmes algébriques comme ACE ou HDE. Une comparaison visuelle plus approfondie est effectuée en Section 6.5.4.

6.6 Conclusion

MuGDAD permet de résoudre les problèmes de convergence de l'algorithme de dessin distribué grâce à son approche multiéchelle. La distribution des algorithmes multiéchelle nécessite d'adapter les différentes étapes dans un nouveau paradigme, ce qui n'est pas toujours évident comme l'atteste la distribution de l'algorithme de sélection d'un ensemble indépendant maximal, *cf.* Section 6.3.

MuGDAD propose aussi deux apports intéressants du point de vue des algorithmes de dessin de graphe. Tout d'abord, il propose de combiner les forces de répulsion approchées avec une décomposition multiéchelle de manière à conserver les informations apprises lors du dessin (les centroïdes et leurs positions dans le dessin). Il propose aussi une approche compatible à la fois avec une implémentation parallèle sur une unique machine ou distribuée pour le calcul sur cluster. Cette approche multiparadigme permet de surcroît de tirer le meilleur parti de la décomposition multiéchelle puisqu'il permet d'adapter les ressources en fonction de la taille du problème.

Les performances de MuGDAD ont aussi été grandement améliorées par rapport à l'algorithme de dessin distribué classique. Les temps de calcul de l'implémentation parallèle permettent de rivaliser avec FM^3, en partie grâce aux calculs des forces de répulsion plus simples, mais moins précis, que l'approche multipôle. Les performances du dessin distribué ont été réellement améliorées grâce à l'approche multiéchelle. Enfin, les dessins obtenus, bien que moins nets que les algorithmes de l'état de l'art, permettent de discerner à la fois une structure globale et des détails plus fins.

Chapitre 7

Conclusion

7.1 Résultats

Les travaux présentés dans cette thèse portent sur la question de l'adaptation des algorithmes de dessin pour le calcul sur cluster dans un environnement distribué. Notre cheminement suit de près l'évolution des algorithmes de dessin de graphe des trente dernières années, comme présenté dans le Chapitre 3, à savoir un premier algorithme de dessin de graphe par modèle de force classique puis une variante de cet algorithme combinée à une approche multiéchelle.

Dans le Chapitre 4, nous présentons un algorithme de dessin de graphe compatible avec le calcul sur cluster. Nous présentons quelles méthodes de résolutions peuvent être efficacement transposées dans le paradigme MapReduce, ce qui nous a conduit à privilégier l'approche par application itérative de forces dans notre premier travail publié [75]. Cette méthode est aussi privilégiée par Arleo et al. [8] qui présentent leur approche en 2015, peu de temps après la nôtre. Concernant le modèle physique sous-jacent, l'observation de départ est la même : les forces entre chaque paire de nœuds existant dans la plupart des modèles physiques pour le dessin ne sont pas viables pour de très grands graphes et a fortiori lorsque l'ensemble du graphe n'est pas accessible en mémoire mais est distribué sur plusieurs machines. Les deux travaux diffèrent sur le choix du modèle physique sous-jacent : des sous-ensembles de nœuds formant des clusters sont utilisés dans les deux approches pour limiter le nombre d'interactions à prendre en compte mais nous avons opté pour un clustering collant au dessin [75] là où Arleo et al. optent [8] pour le calcul des interactions dans un voisinage topologique.

L'approche du Chapitre 6 provient des limitations des algorithmes de dessin par modèle de force classiques. Le nombre d'itérations était déjà identifié comme un problème pour le dessin des grands graphes au début des années 2000 [49, 65, 132]. Le même problème se pose pour le dessin de graphe dans le paradigme MapReduce mais chaque itération représente un surcoût potentiel encore plus important que dans une version séquentielle ou parallèle. Un

des enjeux de ce chapitre est de présenter comment effectuer la décomposition multiéchelle dans le paradigme distribué à partir d'un algorithme de sélection séquentiel difficile à paralléliser, *cf.* Luby [100]. Arleo et al. [9] proposent une telle méthode en 2016 mais sans parler des algorithmes de Luby qui offrent pourtant une borne pour la complexité moyenne. Notre contribution [76] revient sur ce problème et montre comment il est possible de le combiner avec l'approche par pivot développée en Chapitre 4.

7.2 Ouvertures

Nous envisageons ici des pistes pour la poursuite des travaux présentés.

7.2.1 Dessin de graphe en ligne

Problème du dessin en ligne

L'approche en ligne est généralement considérée comme une solution puissante pour les problèmes impliquant des données massives. Au lieu de traiter les données en bloc, ce qui est compliqué par le volume des jeux de données, une approche en ligne dispose d'une solution qu'elle met à jour au fur et à mesure que de nouvelles informations arrivent. De cette manière, seule une solution de taille modérée est conservée (et pas les calculs intermédiaires) et la mise à jour de cette solution ne repose que sur les données entrantes. Cette approche s'est prouvée efficace pour des algorithmes d'apprentissage statistique.

Pour les algorithmes de graphes, et le dessin de graphe en particulier, cette approche est compliquée par plusieurs aspects. Les changements qui peuvent être appliqués à un graphe dans une approche en ligne sont principalement l'ajout et la suppression de nœuds et d'arêtes. Ceci rend l'approche en ligne compliquée puisque la topologie même du graphe est modifiée, ce qui peut entraîner des modifications brutales (*e.g.* supprimer une arête peut conduire à la création d'une nouvelle composante connexe). De plus, dans le cas particulier du dessin, la solution elle-même ne peut pas être résumée par une structure de données simple et légère : à chaque nœud correspond une solution. Les ramifications de la modification du graphe peuvent donc affecter de très nombreuses données.

Dessin de graphe en ligne additif

Une première approche pourrait être de ne considérer que les ajouts de nœuds et d'arêtes dans le graphe. L'ajout d'éléments dans le graphe est moins disruptif que la suppression d'éléments. Il pourrait par exemple être envisagé d'adapter l'algorithme de placement des nœuds de Tunkelang [127] pour venir

placer les nouveaux nœuds directement dans le graphe et d'utiliser quelques rondes d'un algorithme par modèle de force pour affiner le dessin.

De la même manière, dans un modèle purement additif, la plupart des décompositions multiéchelles peuvent être conservées sans modification (par exemple, une décomposition par ensemble maximal indépendant en ignorant les arêtes connectant des nœuds sélectionnés) et le calcul multiéchelle pourrait reprendre directement d'un niveau intermédiaire plutôt que d'exécuter à nouveau l'ensemble de l'algorithme.

Architecture lambda

Cette approche du dessin de graphe en ligne peut aussi s'inscrire dans un paradigme de calcul distribué appelé architecture lambda. Cette architecture propose deux processus de traitement des données simultanés : un rapide mettant à jour une solution préexistante par un algorithme en ligne et un processus lent de calcul sur l'ensemble des données permettant un recalage de l'algorithme en ligne à intervalles réguliers. Cette approche permettrait de pallier les temps de calcul du dessin distribué qui reste trop lent pour une visualisation en temps réel.

Par ce type d'approche, la carte mentale de l'utilisateur ne peut pas être conservée. Cela peut être problématique lorsque l'utilisateur de la visualisation s'en sert pour contrôler certaines communautés.

7.2.2 Descente de gradient stochastique

Nos travaux se sont concentrés sur les algorithmes de dessin par application de forces. L'application des forces est naturellement distribuable, ce qui nous a permis de nous concentrer sur les difficultés de la distribution (à savoir la distribution des forces de répulsion et du calcul d'une décomposition multiéchelle).

Les algorithmes optimisant une fonction de coût représentent une partie importante de la littérature autour du dessin de graphe. Ces algorithmes permettent d'exploiter des résultats mathématiques pour assurer la convergence du dessin. Notamment, il est possible de s'assurer que la fonction de coût est strictement décroissante, et donc que le dessin converge bien vers un état d'équilibre. Il est aussi possible d'évaluer le gain d'un pas d'optimisation, ce qui est difficile avec un algorithme par application de forces.

En Section 4.2.2, nous avons présenté la descente de gradient stochastique, une méthode d'optimisation mathématique qui peut être distribuée. Cette méthode ouvre la voie pour transposer des algorithmes de dessin par optimisation dans un environnement distribué. De nombreuses questions restent cependant en suspens (quelles interactions choisir dans le dessin ?).

7.2.3 Méthode multipôle distribuée

Le modèle de force présenté par GDAD, *cf.* Section 5.2, peut être vu comme une variante de la méthode de Barnes et Hut. Les pivots obtenus à partir de clusters de nœuds sont utilisés pour le calcul des forces de répulsion, à la manière de Barnes et Hut [15]. A la différence de Barnes et Hut, ces pivots suivent la structure du dessin et non pas une décomposition hiérarchique arbitraire. Cela a pour conséquence de passer la complexité du calcul des interactions de $O(|V| \cdot \log |V|)$ à $O(|V|)$. Cependant, l'existence d'une borne d'erreur n'est pas garantie avec cette approximation.

Dans la Section 4.3.5, nous présentons le cheminement naïf pour adapter la méthode multipôle rapide au paradigme MapReduce. Cette méthode permet de calculer les interactions éloignées, mais pas les interactions proches. La borne d'erreur est conservée. Il se pose donc la question de savoir s'il est possible de créer une méthode de calcul des interactions approchées dans un environnement distribué qui soit scalable et pour laquelle une borne d'erreur est prouvée.

Bibliographie

[1] James Abello, Frank Van Ham, and Neeraj Krishnan. Ask-graphview : A large scale graph visualization system. *Visualization and Computer Graphics, IEEE Transactions on*, 12(5) :669–676, 2006.

[2] Ross Adelman. The barnes-hut algorithm in mapreduce.

[3] Srinivas Aluru, John Gustafson, Gurpur M Prabhu, and Fatih E Sevilgen. Distribution-independent hierarchical algorithms for the n-body problem. *The Journal of Supercomputing*, 12(4) :303–323, 1998.

[4] Srinivas Aluru, Gurpur M Prabhu, and John Gustafson. Truly distribution-independent algorithms for the n-body problem. In *Proceedings of the 1994 ACM/IEEE conference on Supercomputing*, pages 420–428. IEEE Computer Society Press, 1994.

[5] Thomas E Anderson, David E Culler, and David Patterson. A case for now (networks of workstations). *IEEE micro*, 15(1) :54–64, 1995.

[6] Francis J Anscombe. Graphs in statistical analysis. *The American Statistician*, 27(1) :17–21, 1973.

[7] Daniel Archambault, Tamara Munzner, and David Auber. Topolayout : Multilevel graph layout by topological features. *Visualization and Computer Graphics, IEEE Transactions on*, 13(2) :305–317, 2007.

[8] Alessio Arleo, Walter Didimo, Giuseppe Liotta, and Fabrizio Montecchiani. A million edge drawing for a fistful of dollars. In *Graph Drawing and Network Visualization*, pages 44–51. Springer, 2015.

[9] Alessio Arleo, Walter Didimo, Giuseppe Liotta, and Fabrizio Montecchiani. A distributed multilevel force-directed algorithm. In *International Symposium on Graph Drawing and Network Visualization*, pages 3–17. Springer, 2016.

[10] David Auber and Yves Chiricota. Improved efficiency of spring embedders : Taking advantage of gpu programming. In *Visualization, Imaging, and Image Processing*, volume 2007, pages 169–175, 2007.

[11] David Auber, Yves Chiricota, Fabien Jourdan, Guy Melançon, et al. Multiscale visualization of small world networks. In *InfoVis*, volume 3, pages 75–81, 2003.

[12] Ching Avery. Giraph : Large-scale graph processing infrastructure on hadoop. *Proceedings of the Hadoop Summit. Santa Clara*, 2011.

[13] Bahman Bahmani, Benjamin Moseley, Andrea Vattani, Ravi Kumar, and Sergei Vassilvitskii. Scalable k-means++. *Proceedings of the VLDB Endowment*, 5(7) :622–633, 2012.

[14] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *science*, 286(5439) :509–512, 1999.

[15] Josh Barnes and Piet Hut. A hierarchical o (n log n) force-calculation algorithm. *Nature*, 324 :446–449, 1986.

[16] Joshua E Barnes and Piet Hut. Error analysis of a tree code. *The astrophysical journal supplement series*, 70 :389–417, 1989.

[17] Gereon Bartel, Carsten Gutwenger, Karsten Klein, and Petra Mutzel. An experimental evaluation of multilevel layout methods. In *International Symposium on Graph Drawing*, pages 80–91. Springer, 2010.

[18] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G Tollis. *Graph drawing : algorithms for the visualization of graphs*. Prentice Hall PTR, 1998.

[19] Rick Beatson and Leslie Greengard. A short course on fast multipole methods. *Wavelets, multilevel methods and elliptic PDEs*, 1 :1–37, 1997.

[20] Donald J Becker, Thomas Sterling, Daniel Savarese, John E Dorband, Udaya A Ranawak, and Charles V Packer. Beowulf : A parallel workstation for scientific computation. In *Proceedings, International Conference on Parallel Processing*, volume 95, pages 11–14, 1995.

[21] François Bertault. A force-directed algorithm that preserves edge crossing properties. In *Graph Drawing*, pages 351–358. Springer, 1999.

[22] Jacques Bertin. *Semiologie graphique*. Mouton, 1970.

[23] Karl-Friedrich Böhringer and Frances Newbery Paulisch. Using constraints to achieve stability in automatic graph layout algorithms. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 43–51. ACM, 1990.

[24] Ingwer Borg and Patrick JF Groenen. *Modern multidimensional scaling : Theory and applications*. Springer Science & Business Media, 2005.

[25] Romain Bourqui, Ludovic Cottret, Vincent Lacroix, David Auber, Patrick Mary, Marie-France Sagot, and Fabien Jourdan. Metabolic network visualization eliminating node redundance and preserving metabolic pathways. *BMC systems biology*, 1(1) :29, 2007.

[26] Olivier Bousquet and Léon Bottou. The tradeoffs of large scale learning. In *Advances in neural information processing systems*, pages 161–168, 2008.

[27] Jürgen Branke, Frank Bucher, and Hartmut Schmeck. Using genetic algorithms for drawing undirected graphs. In *The Third Nordic Workshop on Genetic Algorithms and their Applications*. Citeseer, 1996.

[28] Sangwon Chae, Aditi Majumder, and M Gopi. Hd-graphviz : highly distributed graph visualization on tiled displays. In *Proceedings of the Eighth Indian Conference on Computer Vision, Graphics and Image Processing*, page 43. ACM, 2012.

[29] Andrew A Chien, Scott Pakin, Mario Lauria, Matt Buchanan, Kay Hane, Louis A Giannini, and Jane Prusakova. High performance virtual machines (hpvm's) : Clusters with supercomputing api's and performance. In *PPSC*, 1997.

[30] Markus Chimani, Carsten Gutwenger, Michael Jünger, Gunnar W Klau, Karsten Klein, and Petra Mutzel. The open graph drawing framework (ogdf). *Handbook of Graph Drawing and Visualization*, 2011 :543–569, 2013.

[31] National Research Council et al. *Getting up to speed : The future of supercomputing*. National Academies Press, 2005.

[32] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th National Conference*, ACM '69, pages 157–172, New York, NY, USA, 1969. ACM.

[33] Ron Davidson and David Harel. Drawing graphs nicely using simulated annealing. *ACM Transactions on Graphics (TOG)*, 15(4) :301–331, 1996.

[34] Dominique de Caen. An upper bound on the sum of squares of degrees in a graph. *Discrete Mathematics*, 185(1) :245–248, 1998.

[35] Jan De Leeuw. Convergence of the majorization method for multidimensional scaling. *Journal of classification*, 5(2) :163–180, 1988.

[36] Jeffrey Dean and Sanjay Ghemawat. Mapreduce : simplified data processing on large clusters. *Communications of the ACM*, 51(1) :107–113, 2008.

[37] Edmund Dengler, Mark Friedell, and Joe Marks. Constraint-driven diagram layout. In *Visual Languages, 1993., Proceedings 1993 IEEE Symposium on*, pages 330–335. IEEE, 1993.

[38] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G Tollis. Algorithms for drawing graphs : an annotated bibliography. *Computational Geometry*, 4(5) :235–282, 1994.

[39] Francis X Diebold. A personal perspective on the origin (s) and development of'big data' : The phenomenon, the term, and the discipline, second version. 2012.

[40] Tim Dwyer and Yehuda Koren. Dig-cola : directed graph layout through constrained energy minimization. In *Information Visualization, 2005. INFOVIS 2005. IEEE Symposium on*, pages 65–72. IEEE, 2005.

[41] Tim Dwyer, Yehuda Koren, and Kim Marriott. Drawing directed graphs using quadratic programming. *IEEE Transactions on Visualization and Computer Graphics*, 12(4) :536–548, 2006.

[42] Peter Eades. A heuristic for graph drawing. *Congressus numerantium*, 42 :146–160, 1984.

[43] Peter Eades and Qing-Wen Feng. Multilevel visualization of clustered graphs. In *Graph drawing*, pages 101–112. Springer, 1997.

[44] Peter Eades and Mao Lin Huang. Navigating clustered graphs using force-directed methods. *J. Graph Algorithms Appl.*, 4(3) :157–181, 2000.

[45] John Ellson, Emden R Gansner, Eleftherios Koutsofios, Stephen C North, and Gordon Woodhull. Graphviz and dynagraph : static and dynamic graph drawing tools. *Graph drawing software*, pages 127–148, 2004.

[46] Arne Frick, Andreas Ludwig, and Heiko Mehldau. A fast adaptive layout algorithm for undirected graphs (extended abstract and system demonstration). In *Graph Drawing*, pages 388–403. Springer, 1995.

[47] Yaniv Frishman and Ayellet Tal. Multi-level graph layout on the gpu. *Visualization and Computer Graphics, IEEE Transactions on*, 13(6) :1310–1319, 2007.

[48] Thomas MJ Fruchterman and Edward M Reingold. Graph drawing by force-directed placement. *Softw., Pract. Exper.*, 21(11) :1129–1164, 1991.

[49] Pawel Gajer, Michael T Goodrich, and Stephen G Kobourov. A multi-dimensional approach to force-directed layouts of large graphs. In *Graph Drawing*, pages 211–221. Springer, 2001.

[50] Pawel Gajer and Stephen G Kobourov. Grip : Graph drawing with intelligent placement. In *Graph Drawing*, pages 222–228. Springer, 2001.

[51] Emden R Gansner, Yehuda Koren, and Stephen North. Graph drawing by stress majorization. In *Graph Drawing*, pages 239–250. Springer, 2005.

[52] Emden R Gansner, Stephen C North, and Kiem-Phong Vo. Dag-a program that draws directed graphs. *Software : Practice and Experience*, 18(11) :1047–1062, 1988.

[53] Michael R Garey and David S Johnson. *Computers and intractability*, volume 29. wh freeman New York, 2002.

[54] Alan George and Joseph W Liu. Computer solution of large sparse positive definite. 1981.

[55] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM SIGOPS operating systems review*, volume 37, pages 29–43. ACM, 2003.

[56] Mohammad Ghoniem, J-D Fekete, and Philippe Castagliola. A comparison of the readability of graphs using node-link and matrix-based representations. In *Information Visualization, 2004. INFOVIS 2004. IEEE Symposium on*, pages 17–24. Ieee, 2004.

[57] Apeksha Godiyal, Jared Hoberock, Michael Garland, and John C Hart. Rapid multipole graph drawing on the gpu. In *Graph Drawing*, pages 90–101. Springer, 2009.

[58] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. Graphx : Graph processing in a distributed dataflow framework. In *OSDI*, volume 14, pages 599–613, 2014.

[59] Leslie Greengard. *The rapid evaluation of potential fields in particle systems*. MIT press, 1988.

[60] Leslie Greengard and Vladimir Rokhlin. A fast algorithm for particle simulations. *Journal of computational physics*, 73(2) :325–348, 1987.

[61] José HH Grisi-Filho, Raul Ossada, Fernando Ferreira, and Marcos Amaku. Scale-free networks with the same degree distribution : Different structural properties. *Physics Research International*, 2013, 2013.

[62] Sébastien Grivet, David Auber, Jean-Philippe Domenger, and Guy Melancon. Bubble tree drawing algorithm. In *Computer vision and graphics*, pages 633–641. Springer, 2006.

[63] Stefan Hachul and Michael Jünger. Drawing large graphs with a potential-field-based multilevel algorithm. In *Graph drawing*, pages 285–295. Springer, 2005.

[64] Stefan Hachul and Michael Jünger. An experimental comparison of fast algorithms for drawing general large graphs. In *Graph Drawing*, pages 235–250. Springer, 2006.

[65] Ronny Hadany and David Harel. A multi-scale algorithm for drawing graphs nicely. *Discrete Applied Mathematics*, 113(1) :3–21, 2001.

[66] Kenneth M Hall. An r-dimensional quadratic placement algorithm. *Management science*, 17(3) :219–229, 1970.

[67] David Harel and Yehuda Koren. A fast multi-scale method for drawing large graphs. *Journal of graph algorithms and applications*, 6(3) :179–202, 2002.

[68] David Harel and Yehuda Koren. Graph drawing by high-dimensional embedding. In *Graph Drawing*, pages 207–219. Springer, 2002.

[69] Lawrence H Harper. Optimal assignments of numbers to vertices. *Journal of the Society for Industrial and Applied Mathematics*, 12(1) :131–135, 1964.

[70] Weiqing He and Kim Marriott. Constrained graph layout. In *Graph Drawing*, pages 217–232. Springer, 1997.

[71] Kyle Hegeman, Nathan A Carr, and Gavin SP Miller. Particle-based fluid simulation on the gpu. In *Computational Science–ICCS 2006*, pages 228–235. Springer, 2006.

[72] Nathalie Henry, Jean-Daniel Fekete, and Michael J McGuffin. Nodetrix : a hybrid visualization of social networks. *IEEE transactions on visualization and computer graphics*, 13(6) :1302–1309, 2007.

[73] Ivan Herman, Guy Melançon, and M Scott Marshall. Graph visualization and navigation in information visualization : A survey. *Visualization and Computer Graphics, IEEE Transactions on*, 6(1) :24–43, 2000.

[74] Kashmir Hill. Blueprints of nsa's ridiculously expensive data center in utah suggest it holds less info than thought. https://www.forbes.com/sites/kashmirhill/2013/07/24/blueprints-of-nsa-data-center-in-utah-suggest-its-storage-capacity-is-less-impressive-than-thought, 2013. Accessed : 2017-05-29.

[75] Antoine Hinge and David Auber. Distributed graph layout with Spark. In *Information Visualisation (iV), 2015 19th International Conference on*, pages 271–276. IEEE, 2015.

[76] Antoine Hinge, David Auber, and Gaelle Richer. Mugdad : Multilevel graph drawing algorithm in a distributed architecture. In *Computer Graphics, Visualization, Computer Vision and Image Processing, 11th International Conference on*, July 2017.

[77] Danny Holten. Hierarchical edge bundles : Visualization of adjacency relations in hierarchical data. *IEEE Transactions on visualization and computer graphics*, 12(5) :741–748, 2006.

[78] Manuel Holtgrewe, Peter Sanders, and Christian Schulz. Engineering a scalable high quality graph partitioner. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.

[79] Stephan Holzer and Roger Wattenhofer. Optimal distributed all pairs shortest paths and applications. In *Proceedings of the 2012 ACM symposium on Principles of distributed computing*, pages 355–364. ACM, 2012.

[80] Brian Johnson and Ben Shneiderman. Tree-maps : A space-filling approach to the visualization of hierarchical information structures. In *Proceedings of the 2nd conference on Visualization'91*, pages 284–291. IEEE Computer Society Press, 1991.

[81] David S Johnson. The np-completeness column : An ongoing guide. *Journal of Algorithms*, 3(2) :182–195, 1982.

[82] David S Johnson. The np-completeness column : An ongoing guide. *Journal of Algorithms*, 5(1) :147–160, 1984.

[83] Maja Kabiljo, Dionysis Logothetis, Sergey Edunov, and Avery Ching. A comparison of state-of-the-art graph processing systems, Octobre 2016.

[84] Tomihisa Kamada and Satoru Kawai. An algorithm for drawing general undirected graphs. *Information processing letters*, 31(1) :7–15, 1989.

[85] Thomas Kamps, Joerg Kleinz, and John Read. Constraint-based spring-model algorithm for graph layout. In *Graph drawing*, pages 349–360. Springer, 1996.

[86] Scott Kirkpatrick, C Daniel Gelatt, Mario P Vecchi, et al. Optimization by simulated annealing. *Science*, 220(4598) :671–680, 1983.

[87] Stephen G Kobourov. Spring embedders and force directed graph drawing algorithms. *arXiv preprint arXiv :1201.3011*, 2012.

[88] Yehuda Koren, Liran Carmel, and David Harel. Ace : A fast multiscale eigenvectors computation for drawing huge graphs. In *Information Visualization, 2002. INFOVIS 2002. IEEE Symposium on*, pages 137–144. IEEE, 2002.

[89] Yehuda Koren, Liran Carmel, and David Harel. Drawing huge graphs by algebraic multigrid optimization. *Multiscale Modeling & Simulation*, 1(4) :645–673, 2003.

[90] Yehuda Koren and David Harel. Axis-by-axis stress minimization. In *Graph Drawing*, pages 450–459. Springer, 2003.

[91] Evgenios M Kornaropoulos and Ioannis G Tollis. Overloaded orthogonal drawings. In *International Symposium on Graph Drawing*, pages 242–253. Springer, 2011.

[92] Evgenios M Kornaropoulos and Ioannis G Tollis. Dagview : an approach for visualizing large graphs. In *International Symposium on Graph Drawing*, pages 499–510. Springer, 2012.

[93] Cornelius Lanczos. *An iteration method for the solution of the eigenvalue problem of linear differential and integral operators*. United States Governm. Press Office Los Angeles, CA, 1950.

[94] Scott Langevin, David Jonker, David Giesbrecht, and Michael Crouch. Multi-scale community visualization of massive graph data. *Proceedings of the exploring graphs at scale (EGAS)*, 2015.

[95] John Langford, Alex J Smola, and Martin Zinkevich. Slow learners are fast. *Advances in Neural Information Processing Systems*, 22 :2331–2339, 2009.

[96] Bongshin Lee, Catherine Plaisant, Cynthia Sims Parr, Jean-Daniel Fekete, and Nathalie Henry. Task taxonomy for graph visualization. In *Proceedings of the 2006 AVI workshop on BEyond time and errors : novel evaluation methods for information visualization*, pages 1–5. ACM, 2006.

[97] Jure Leskovec and Rok Sosič. Snap : A general-purpose network analysis and graph-mining library. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 8(1) :1, 2016.

[98] Hyojun Lim and Chongkwon Kim. Flooding in wireless ad hoc networks. *Computer Communications*, 24(3) :353–363, 2001.

[99] Tao Lin and Peter Eades. Integration of declarative and algorithmic approaches for layout creation. In *Graph Drawing*, pages 376–387. Springer, 1995.

[100] M Luby. A simple parallel algorithm for the maximal independent set problem. In *Proceedings of the seventeenth annual ACM symposium on Theory of computing*, pages 1–10. ACM, 1985.

[101] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(01) :5–20, 2007.

[102] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel : a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[103] Oliver Marquardt and Stefan Schamberger. Open benchmarks for load balancing heuristics in parallel adaptive finite element computations. In *PDPTA*, pages 685–691, 2005.

[104] Claudio Martella, Dionysios Logothetis, Andreas Loukas, and Georgos Siganos. Spinner : Scalable graph partitioning in the cloud. In *Data Engineering (ICDE), 2017 IEEE 33rd International Conference on*, pages 1083–1094. IEEE, 2017.

[105] Justin Matejka and George Fitzmaurice. Same stats, different graphs : Generating datasets with varied appearance and identical statistics through simulated annealing. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, pages 1290–1294. ACM, 2017.

[106] Donald Miner and Adam Shook. *MapReduce Design Patterns : Building Effective Algorithms and Analytics for Hadoop and Other Systems.* " O'Reilly Media, Inc.", 2012.

[107] Christopher Mueller, Douglas Gregor, and Andrew Lumsdaine. Distributed force-directed graph layout and visualization. In *EGPGV*, pages 83–90, 2006.

[108] Christopher Mueller, Benjamin Martin, and Andrew Lumsdaine. A comparison of vertex ordering algorithms for large graph visualization. In *Visualization, 2007. APVIS'07. 2007 6th International Asia-Pacific Symposium on*, pages 141–148. IEEE, 2007.

[109] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin : a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM, 2008.

[110] Kaare Brandt Petersen, Michael Syskind Pedersen, et al. The matrix cookbook. *Technical University of Denmark*, 7 :15, 2008.

[111] Gregory F Pfister. *In search of clusters*. Prentice-Hall, Inc., 1998.

[112] Helen Purchase. Which aesthetic has the greatest effect on human understanding ? In *International Symposium on Graph Drawing*, pages 248–261. Springer, 1997.

[113] Helen C Purchase. Metrics for graph drawing aesthetics. *Journal of Visual Languages & Computing*, 13(5) :501–516, 2002.

[114] Xinyu Que, Lars Schneidenbach, Fabio Checconi, Carlos HÃ Costa, and Daniele Buono. Performance analysis of spark/graphx on power8 cluster. In *International Conference on High Performance Computing*, pages 268–285. Springer, 2016.

[115] Aaron Quigley and Peter Eades. Fade : Graph drawing, clustering, and visual abstraction. In *Graph Drawing*, pages 197–210. Springer, 2001.

[116] Neil R Quinn Jr, Melvin Breuer, et al. A forced directed component placement procedure for printed circuit boards. *Circuits and Systems, IEEE Transactions on*, 26(6) :377–388, 1979.

[117] Edward M. Reingold and John S. Tilford. Tidier drawings of trees. *IEEE Transactions on software Engineering*, (2) :223–228, 1981.

[118] John Michaels Robson. Algorithms for maximum independent sets. *Journal of Algorithms*, 7(3) :425–440, 1986.

[119] Naidila Sadashiv and SM Dilip Kumar. Cluster, grid and cloud computing : A detailed comparison. In *Computer Science & Education (ICCSE), 2011 6th International Conference on*, pages 477–482. IEEE, 2011.

[120] Manojit Sarkar and Marc H Brown. Graphical fisheye views of graphs. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 83–91. ACM, 1992.

[121] Fatih Erdogan Sevilgen, Srinivas Aluru, and Natsuhiko Futamura. A provably optimal, distribution-independent parallel fast multipole method. In *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*, pages 77–84. IEEE, 2000.

[122] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

[123] Paolo Simonetto, Daniel Archambault, David Auber, and Romain Bourqui. Impred : An improved force-directed algorithm that prevents nodes from crossing edges. In *Computer Graphics Forum*, volume 30, pages 1071–1080. Wiley Online Library, 2011.

[124] Peter A Stark. *Introduction to numerical methods*. Macmillan New York, 1970.

[125] Kozo Sugiyama and Kazuo Misue. A simple and unified method for drawing graphs : Magnetic-spring algorithm. In *International Symposium on Graph Drawing*, pages 364–375. Springer, 1994.

[126] Anna Tikhonova and Kwan-Liu Ma. A scalable parallel force-directed graph layout algorithm. In *Proceedings of the 8th Eurographics conference on Parallel Graphics and Visualization*, pages 25–32. Eurographics Association, 2008.

[127] Daniel Tunkelang. A practical approach to drawing undirected graphs. Technical report, DTIC Document, 1994.

[128] William T Tutte. How to draw a graph. *Proc. London Math. Soc*, 13(3) :743–768, 1963.

[129] Pamela Vagata and Kevin Wilfong. Scaling the facebook data warehouse to 300 pb. https://code.facebook.com/posts/229861827208629/scaling-the-facebook-data-warehouse-to-300-pb/, 2014. Accessed : 2017-05-29.

[130] Luis Vaquero, Felix Cuadrado, Dionysios Logothetis, and Claudio Martella. Adaptive partitioning for large-scale dynamic graphs. In *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on*, pages 144–153. IEEE, 2014.

[131] Mehul Nalin Vora. Hadoop-hbase for large-scale data. In *Computer science and network technology (ICCSNT), 2011 international conference on*, volume 1, pages 601–605. IEEE, 2011.

[132] Chris Walshaw. A multilevel algorithm for force-directed graph drawing. In *Graph Drawing*, pages 171–182. Springer, 2001.

[133] Michael S Warren and John K Salmon. Astrophysical n-body simulations using hierarchical tree data structures. In *Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*, pages 570–576. IEEE Computer Society Press, 1992.

[134] Benjamin Watson, David Brink, Thomas A Lograsso, D Devajaran, Theresa-Marie Rhyne, and Himesh Patel. Visualizing very large layered graphs with quilts. Technical report, North Carolina State University. Dept. of Computer Science, 2008.

[135] Douglas Brent West et al. *Introduction to graph theory*, volume 2. Prentice hall Upper Saddle River, 2001.

[136] Charles Wetherell and Alfred Shannon. Tidy drawings of trees. *IEEE Transactions on software Engineering*, (5) :514–520, 1979.

[137] Jeremy M Wolfe. What can 1 million trials tell us about visual search ? *Psychological Science*, 9(1) :33–39, 1998.

[138] Reynold S Xin, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. Graphx : A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, page 2. ACM, 2013.

[139] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark : cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, volume 10, page 10, 2010.

[140] Shengdong Zhao, Michael J McGuffin, and Mark H Chignell. Elastic hierarchies : Combining treemaps and node-link diagrams. In *Information Visualization, 2005. INFOVIS 2005. IEEE Symposium on*, pages 57–64. IEEE, 2005.

[141] Martin Zinkevich, Markus Weimer, Lihong Li, and Alex J Smola. Parallelized stochastic gradient descent. In *Advances in neural information processing systems*, pages 2595–2603, 2010.