La visualisation d'information à l'ère du Big Data : résoudre les problèmes de scalabilité par l'abstraction multi-échelle

Alexandre Perrot

To cite this version:

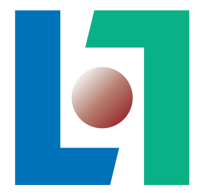
Alexandre Perrot. La visualisation d'information à l'ère du Big Data : résoudre les problèmes de scalabilité par l'abstraction multi-échelle. Autre [cs.OH]. Université de Bordeaux, 2017. Français. NNT : 2017BORD0775 . tel-01664983

HAL Id: tel-01664983
https://theses.hal.science/tel-01664983

Submitted on 15 Dec 2017

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

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



THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

par **Alexandre Perrot**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

La visualisation d'information à l'ère du Big Data : résoudre les problèmes de scalabilité par l'abstraction multi-échelle

---

Date de soutenance : 27 novembre 2017

Devant la commission d'examen composée de :

Clémence MAGNIEN	Directrice de Recherche, CNRS	Rapporteur
Jean-Daniel FEKETE	Directeur de Recherche, INRIA	Rapporteur
Olivier BEAUMONT	Directeur de Recherche, INRIA	Examinateur
Renaud BLANCH	Maître de Conférences, Université Grenoble Alpes	Examinateur
Guy MELANÇON	Professeur, Université de Bordeaux	Examinateur
David AUBER	Maître de Conférences, Université de Bordeaux	Directeur de thèse

– 2017 –

La visualisation d'information à l'ère du Big Data : résoudre les problèmes de scalabilité par l'abstraction multi-échelle

Résumé L'augmentation de la quantité de données à visualiser due au phénomène du Big Data entraîne de nouveaux défis pour le domaine de la visualisation d'information. D'une part, la quantité d'information à représenter dépasse l'espace disponible à l'écran, entraînant de l'occlusion. D'autre part, ces données ne peuvent pas être stockées et traitées sur une machine conventionnelle. Un système de visualisation de données massives doit permettre la scalabilité de perception et de performances.
Dans cette thèse, nous proposons une solution à ces deux problèmes au travers de l'abstraction multi-échelle des données. Plusieurs niveaux de détail sont précalculés sur une infrastructure Big Data pour permettre de visualiser de grands jeux de données jusqu'à plusieurs milliards de points. Pour cela, nous proposons deux approches pour implémenter l'algorithme de *canopy clustering* sur une plateforme de calcul distribué. Nous présentons une application de notre méthode à des données géolocalisées représentées sous forme de carte de chaleur, ainsi qu'à des grands graphes. Ces deux applications sont réalisées à l'aide de la bibliothèque de visualisation dynamique Fatum, également présentée dans cette thèse.

Mots-clés Mégadonnées, Visualisation, Partitionnement
Laboratoire d'accueil LaBRI, UMR 5800, F-33400 Talence, France

Information Visualization in the Big Data era: tackling scalability issues using multiscale abstractions.

Abstract With the advent of the Big Data era come new challenges for Information Visualization. First, the amount of data to be visualized exceeds the available screen space. Second, the data cannot be stored and processed on a conventional computer. To alleviate both of these problems, a Big Data visualization system must provide perceptual and performance scalability.
In this thesis, we propose to use multi-scale abstractions as a solution to both of these issues. Several levels of detail can be precomputed using a Big Data Infrastructure in order to visualize big datasets up to several billion points. For that, we propose two approaches to implementing the canopy clustering algorithm for a distributed computation cluster. We present applications of our method to geolocalized data visualized through a heatmap, and big graphs. Both of these applications use the dynamic visualization library, which is also presented in this thesis.

Keywords Big Data, Visualization, Clustering

À mon grand-père, Michel

Remerciements

Je tiens tout d'abord à remercier mes rapporteurs Clémence Magnien et Jean-Daniel Fekete pour avoir accepté de relire cette thèse. Je remercie également les membres du jury Olivier Beaumont, Renaud Blanch et Guy Melançon. Cette thèse n'aurait évidemment pas été possible sans mon directeur de thèse, David Auber, que je remercie tout particulièrement pour ces quatre ans. J'ai beaucoup appris auprès de lui et j'ose espérer que ce fut réciproque.

Je veux remercier tous mes collègues de l'équipe MaBioVis, du thème EVADOMe et des GT Big Data, en particulier Romain B., Romain G., Bruno, Sofiane et Nicolas, ainsi que mes camarades doctorants et ingénieurs, Joris, Antoine, Arnaud, Fred, Jason, Gaëlle, Aaron, Rémi, Thomas et Tristan pour tous ces moments partagés, petits et grands, serieux ou non.

Je tiens tout particulièrement à remercier mes amis Alexis B., Boris, Germain, Antoine et Alexis C., ainsi que ma belle famille, Christian, Ariane, Cécile et Quentin pour leur présence et leur soutien.

Je veux aussi remercier mes parents, Corinne et Rodolphe, qui ont su me donner la curiosité et le goût de la découverte indispensables pour réaliser une thèse, ma grand-mère Colette et mon grand-père Michel, qui nous a malheureusement quitté peu après le début de cette aventure.

Enfin, je tiens à remercier du fond du coeur ma compagne Claire pour son soutien indéfectible. Sans elle, mener cette thèse à son terme aurait été beaucoup plus difficile.

Table des matières

I Introduction 1

1 Introduction 3
- 1.1 Visualisation . . . 4
  - 1.1.1 Pipeline de visualisation . . . 7
  - 1.1.2 Encodage visuel . . . 8
  - 1.1.3 Idiomes de visualisation . . . 10
- 1.2 Big Data . . . 12
- 1.3 Contributions . . . 15

2 Préliminaires 17
- 2.1 Paradigmes pour le Big Data . . . 17
  - 2.1.1 Architecture distribuée . . . 17
  - 2.1.2 MapReduce . . . 18
  - 2.1.3 Pregel . . . 19
  - 2.1.4 Types de calcul . . . 21
  - 2.1.5 Lambda architecture . . . 21
- 2.2 Outils et implémentations utilisés . . . 22
  - 2.2.1 Hadoop . . . 22
  - 2.2.2 Spark . . . 23

3 État de l'art 25
- 3.1 Scalabilité de perception . . . 25
  - 3.1.1 Visualisation multi-échelle . . . 29
- 3.2 Scalabilité de performance . . . 31

II Visualisation de données massives : Concepts et Techniques 33

4 Méthodologie générale 35
- 4.1 Contraintes et besoins spécifiques . . . 35
- 4.2 Modification du pipeline de visualisation pour les données massives 37
  - 4.2.1 Pipeline orienté pixels . . . 37

4.2.2 Pipeline orienté géométrie 37
4.2.3 Pipeline orienté abstraction de données 38
4.3 Réalisation du pipeline 39
4.3.1 Agrégation de données 39
4.3.2 Indexation & Requêtage 39
4.3.3 Rendu 40
4.4 Conclusion 40

5 Agrégation géométrique pour la visualisation 41
5.1 Canopy Clustering 41
5.1.1 Algorithme pour l'agrégation géométrique 42
5.1.2 Borne sur le nombre de représentants 43
5.1.3 Complexité 43
5.2 Utilisation multi-échelle du canopy clustering 44
5.2.1 Distance d'agrégation par niveau 44
5.2.2 Nombre total de niveaux de détail 45
5.2.3 Complexité 46
5.3 Canopy clustering distribué : approche par partitionnement spatial . 46
5.3.1 Algorithme 46
5.3.2 Taille des partitions 49
5.3.3 Équilibrage de charge 50
5.4 Canopy clustering distribué : approche par ensemble indépendant maximal 51
5.4.1 Principe 51
5.4.2 Calcul du graphe de disques 52
5.4.3 Calcul de l'ensemble indépendant maximal 56
5.5 Conclusion 59

6 Fatum : un middleware pour la visualisation d'information dynamique 61
6.1 *Marks* & *Connections* : un paradigme pour la visualisation d'information 63
6.1.1 Marks 64
6.1.2 Connections 65
6.1.3 Ajout de texte 66
6.1.4 Point de vue dans la visualisation 67
6.1.5 Gestion de la dynamicité sous forme d'états 68
6.1.6 Interface de programmation 71
6.1.7 Exemples de visualisations 72
6.1.8 Utilisation avec D3 76
6.2 Implémentation 78
6.2.1 Choix des frameworks utilisés 78
6.2.2 Gestion des animations par *double buffering* 78
6.2.3 Animation de la forme par *distance field* 80
6.2.4 Occlusion de texte avec garantie de visibilité 82
6.2.5 Performances 84
6.3 Conclusion 85

III Applications 87

7 HeatMapReduce : Visualisation de carte de chaleur à grande échelle 89
- 7.1 Introduction . . . 89
  - 7.1.1 Kernel Density Estimation . . . 89
  - 7.1.2 Approximate Kernel Density Estimate . . . 92
- 7.2 Vue d'ensemble du système . . . 93
- 7.3 Calcul en batch . . . 94
  - 7.3.1 Performances . . . 95
  - 7.3.2 Stockage . . . 96
- 7.4 Calcul en flux . . . 98
  - 7.4.1 Principe . . . 98
  - 7.4.2 Implémentation . . . 100
- 7.5 Visualisation . . . 101
  - 7.5.1 Rendu de carte de chaleur en temps borné . . . 101
  - 7.5.2 Qualité de la visualisation . . . 104
- 7.6 Conclusion . . . 106

8 Cornac : Visualisation de graphes à grande échelle 109
- 8.1 Introduction . . . 109
- 8.2 Vue d'ensemble . . . 110
- 8.3 Agrégation . . . 111
  - 8.3.1 Agrégation de noeuds . . . 111
  - 8.3.2 Agrégation d'arêtes . . . 113
  - 8.3.3 Performances . . . 117
- 8.4 Visualisation . . . 118
  - 8.4.1 Récupération des données . . . 118
  - 8.4.2 Rendu des sommets . . . 120
  - 8.4.3 Rendu des arêtes . . . 121
  - 8.4.4 Faisceautage d'arêtes . . . 122
- 8.5 Étude de cas : Panama Papers . . . 125
  - 8.5.1 Visualisation . . . 125
  - 8.5.2 Performances . . . 127
  - 8.5.3 Discussion . . . 127
- 8.6 Conclusion . . . 129

9 Conclusion 131

Bibliographie 133

Première partie

Introduction

1 Introduction

Jacques Bertin débute son ouvrage « La graphique et le traitement graphique de l'information » [13] par l'anecdote suivante :

> Il était une fois un directeur de grand hôtel soucieux d'améliorer la marche de son entreprise. Il fit établir diverses statistiques. Le tableau de chiffres resta de nombreux jours sur le bureau directorial.
>
> Puis un matin le secrétaire présenta une image construite d'après les chiffres du tableau. Après quelques instants d'attention le directeur fit appeler ses collaborateurs et avec eux :
> il définit une nouvelle politique des prix,
> il modifia les services offerts à la clientèle,
> il ajusta les stocks,
> il modifia le système de promotion-vente.
>
> Enfin il termina sa journée par une visite au maire de la ville, responsable de la date des foires. Les resultats obtenus lui assurèrent une rapide promotion.
> (Jacques Bertin [13])

Cette histoire illustre parfaitement ce qu'apporte la visualisation. Il ne suffit pas de posséder l'information pour en extraire du sens. Le but de la visualisation est d'augmenter la cognition humaine en représentant graphiquement l'information. En utilisant des mécanismes de perception inconscients, il est alors possible de comprendre et faire comprendre le sens contenu dans les données.

A l'heure où la quantité des données disponibles ne cesse d'augmenter, le recours à la visualisation est plus que jamais nécessaire. Cependant, malgré l'augmentation des capacités de calculs de nos ordinateurs, les capacités de perception humaines n'ont pas changé. Cela entraîne de nouveaux défis à surmonter pour espérer visualiser les quantités de données générées à l'heure du "Big Data".

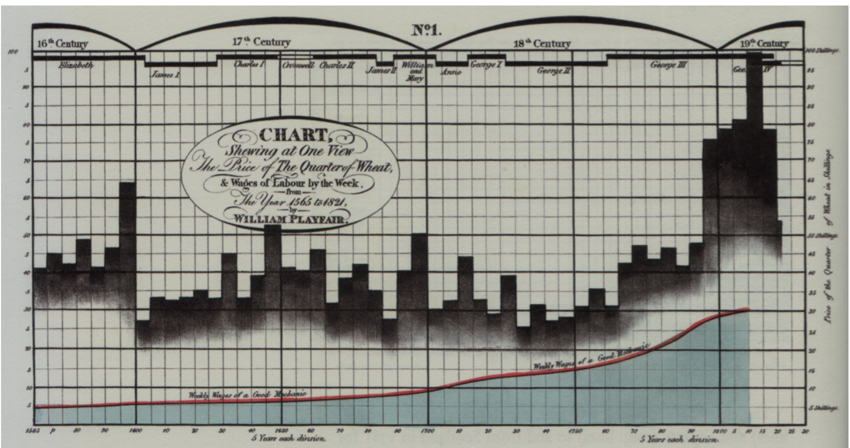


FIGURE 1.1 – Diagramme du prix du blé par William Playfair [89, 106]. Cette visualisation met en perspective l'évolution du prix du blé avec celle d'un salaire hebdomadaire de référence. Playfair souhaite montrer ici que le blé n'a jamais été aussi bon marché si l'on tient compte de l'évolution des salaires. Même si l'information est bien contenue dans le diagramme, cette conclusion n'est pas immédiate car elle requiert une étape d'analyse supplémentaire. En représentant le rapport entre les deux valeurs, cette conclusion serait plus évidente.

1.1 Visualisation

Le domaine de la visualisation s'intéresse à la façon de transformer des données en représentation graphique. Cette représentation graphique s'appelle une visualisation. De nombreux exemples sont disséminés dans la vie de tous les jours. Ainsi, le directeur d'hôtel de l'histoire précédente se rend au travail en métro en vérifiant son itinéraire sur le schéma des lignes. Une fois installé, il peut consulter les cartes qui présentent les résultats d'un sondage dans le journal avant de se plonger dans l'analyse des courbes indicatrices de la bonne santé de son établissement.

Bien que des visualisations, notamment sous forme de cartes, existent depuis l'Antiquité, la représentation graphique d'une information abstraite émerge surtout au XIX$^{\text{ème}}$ siècle. William Playfair (1759-1823) est considéré comme l'un des pionniers de la représentation graphique de l'information. Cet ingénieur et économiste écossais est l'inventeur de plusieurs types de visualisation encore couramment utilisés aujourd'hui, parmi lesquels la courbe temporelle et l'histogramme. Une de ses visualisations les plus connues, visible en Figure 1.1, combine ces deux idiomes [1] pour comparer l'évolution du prix du blé et du salaire hebdomadaire d'un "bon mécanicien". Cette visualisation, ainsi que quelques autres, accompagne la lettre sur l'agriculture que Playfair a adressé au parlement britannique en 1821 [89]. Elle vient appuyer son propos selon lequel le prix du blé n'a jamais été aussi bon marché par rapport au coût du travail. Ici, la visualisation sert donc à exposer la conclu-

1. Le concept d'idiome de visualisation et quelques exemples sont expliqués en section 1.1.3

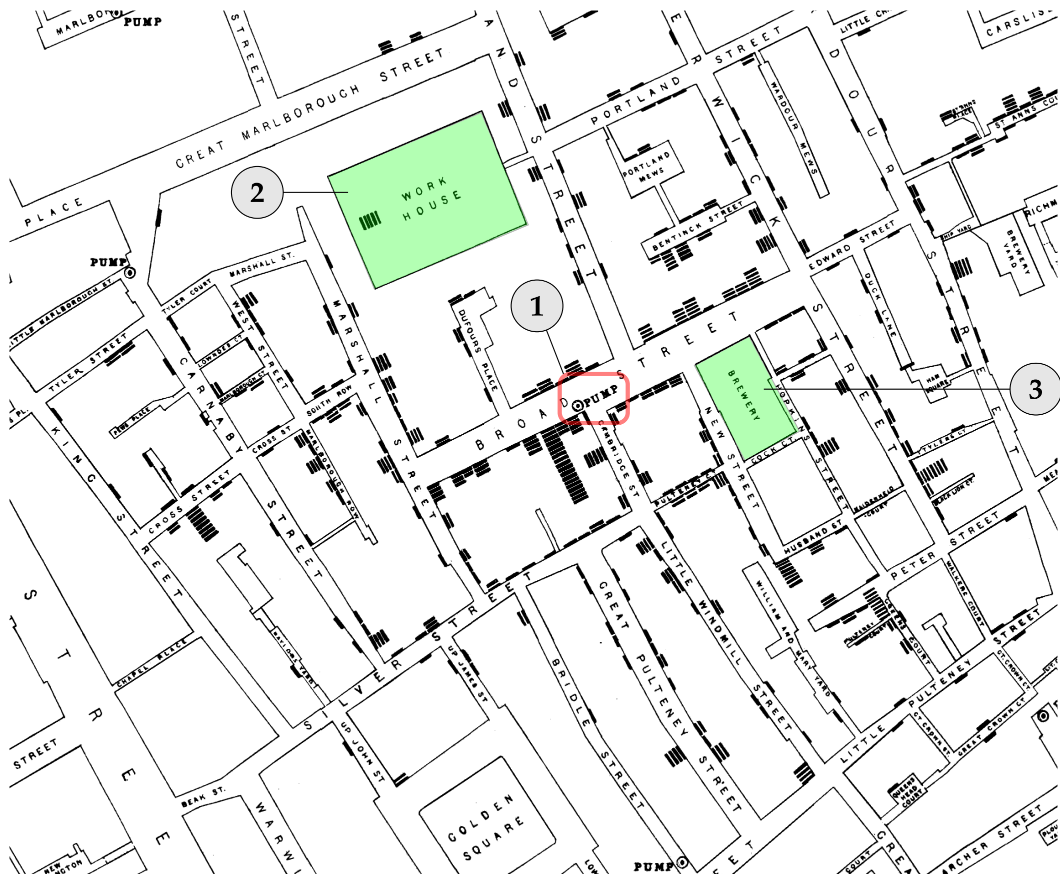


FIGURE 1.2 – Détail de la carte de Londres établie par le Dr. John Snow lors de l'épidémie de choléra de 1854. Chaque point représente un décès attribué à la maladie. La carte indique aussi l'emplacement des pompes utilisées par la population pour puiser de l'eau potable. En repérant une plus grande densité de décès autour de la pompe de Broad Street (①), il en a déduit qu'elle était sans doute infectée. On peut également remarquer sur cette carte deux zones de faible densité malgré leur proximité à la pompe. D'abord, une "workhouse" (②), où seuls 5 des 530 résidents sont tombés malades. Ensuite, une brasserie (③), où aucun cas n'a été recensé. Dans les deux cas, les habitants ne consommaient pas l'eau de la pompe de Broad Street. La "workhouse" possédait son propre puits et la brasserie donnait une ration de bière à ses employés.

sion d'une analyse pour convaincre un interlocuteur. On parle de visualisation descriptive.

La visualisation peut aussi aider à la décision en rendant plus claire les données collectées, comme l'illustre la carte produite par le Dr. John Snow lors de l'épidémie de choléra de Londres en 1854 (voir Figure 1.2). Supposant que l'épidémie se propage par l'eau, il décide de noter sur une carte l'emplacement de chaque victime. Il se dégage alors un plus grand nombre de décès autour d'une pompe du district de

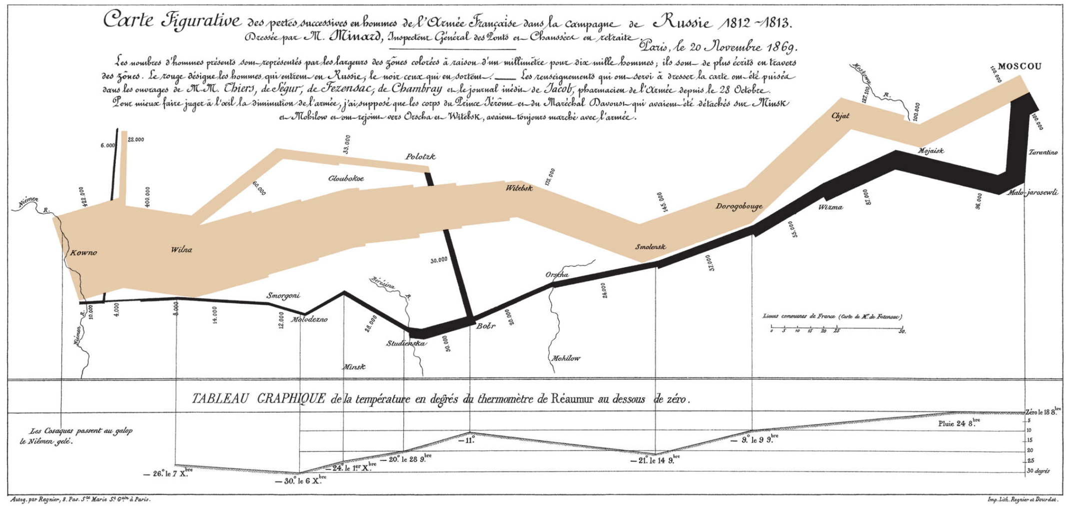


FIGURE 1.3 – Carte de la campagne de Russie par Charles Joseph Minard. La carte montre le trajet aller (en brun) et retour (en noir) des troupes napoléoniennes. A chaque étape, il est également possible de connaître le nombre de soldats. Sur le trajet retour sont également indiquées les températures glaciales qui ont contribué à l'affaiblissement des troupes.

Soho. Après avoir retiré la poignée de cette pompe vraisemblablement infectée, l'épidémie s'est rapidement arrêtée. Snow s'est servi de la visualisation pour tester ses hypothèses et vérifier sa théorie. Lorsque la visualisation a pour objectif de chercher dans les données, on l'appelle analytique.

Un autre exemple marquant de visualisation de cette époque est la carte réalisée par Charles Joseph Minard en 1869 pour illustrer les pertes de l'armée napoléonienne lors de la campagne de Russie de 1812-1813 (voir Figure 1.3). Cette carte ajoute au trajet suivi par les troupes une représentation graphique du nombre de soldats composant l'armée à chaque instant du parcours. Cela permet de mettre en valeur la diminution drastique au cours du temps des effectifs de l'armée. La présence de repères géographiques permet également de lier cette évolution à des évènements marquants, comme la traversée de la rivière Bérézina sur le trajet retour. Au cours de sa vie, Minard a réalisé une cinquantaine de cartes, représentant majoritairement des flux. Ce type de diagramme où la taille d'un trait est proportionnelle à l'importance d'un flux est aujourd'hui appelé diagramme de Sankey, d'après Matthew Sankey (1853-1926), qui utilisa plus tard le même principe pour représenter l'énergie produite par un moteur à vapeur.

Ces trois visualisations, comme toutes celles de leur époque, ont été spécialement conçues pour un cas en particulier. De plus, elles doivent être entièrement réalisées à la main, aucun processus d'automatisation n'étant disponible. La réalisation d'une visualisation prenait alors beaucoup de temps et ne pouvait se faire que sous forme d'image statique.

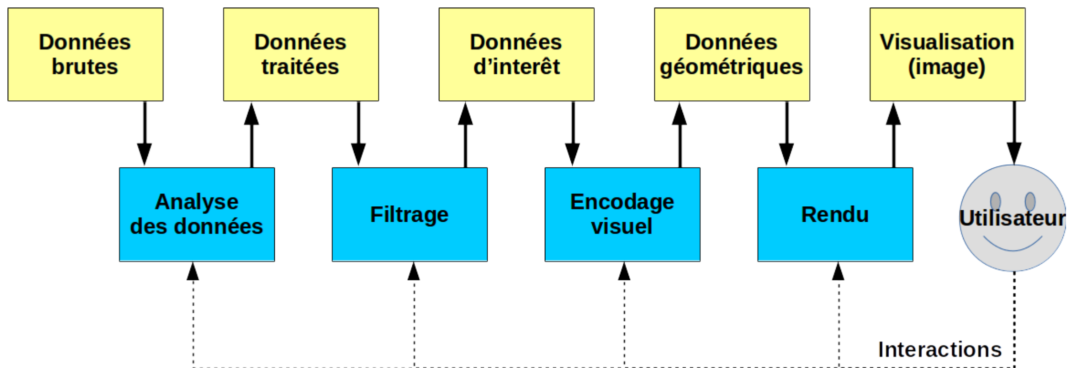


FIGURE 1.4 – Le pipeline de visualisation, adapté de [37]. Dans un système interactif, il peut être exécuté de nombreuses fois par seconde.

1.1.1 Pipeline de visualisation

Avec l'essor de l'informatique, il est possible d'automatiser de nombreux aspects de la visualisation. D'abord, le dessin final (conversion en image) peut être réalisé automatiquement. Ensuite, le traitement de l'information pour produire un type de visualisation peut être confié à des algorithmes, rendant plus facile et rapide la comparaison de plusieurs jeux de données.

Enfin, la visualisation ne se conçoit plus uniquement comme une image statique, mais comme un système interactif, sur lequel l'utilisateur final possède un pouvoir de rétroaction. Les données brutes doivent suivre certaines étapes avant d'être présentées à l'utilisateur, qui peut alors agir sur chaque transformation pour modifier la visualisation finale et en tirer de nouveaux enseignements. Ce processus est résumé dans le *pipeline de visualisation* (voir Figure 1.4), initialement développé par Haber et McNabb [53] pour la visualisation scientifique et plus tard enrichi par Dos Santos et Brodlie [37] pour la visualisation d'information.

Ce pipeline se divise en plusieurs étapes qui permettent de passer de données brutes à une visualisation :

— L'étape de **préparation des données** permet de choisir ce qui va être visualisé. A cette étape les données brutes sont raffinées afin d'en garder les aspects intéressants ou de dériver des valeurs de données existantes (moyenne, différence, projection cartographique ...). Il résulte de cette étape un ensemble de données plus ou moins structurées prêtes à être transformées en visualisation.

— Les données préparées sont ensuite **filtrées** pour ne retenir qu'un sous-ensemble d'intérêt. Ce filtrage peut être par exemple temporel (le résultat d'exploitation d'une année en particulier), sémantique (uniquement les amis d'une personne d'intérêt) ou encore géographique (résultats d'élection sur une commune).

— Une fois les données d'intérêt identifiées vient l'étape d'**encodage visuel** lors de laquelle est décidé sous quelle forme vont être représentée les données. Cette étape produit une représentation géométrique qui décrit la visualisation désirée. C'est à cette étape que les données se transforment en visualisation.

— Enfin, l'étape de **rendu** consiste à dessiner la géométrie produite par l'étape d'encodage visuel. Cette étape est le plus souvent effectuée par un ordinateur et peut faire appel à l'utilisation d'un processeur graphique. L'image ainsi produite est la visualisation finale qui peut être présentée à l'utilisateur.

Toutes les étapes de ce pipeline peuvent être modifiées à tout moment par une interaction de l'utilisateur. Ainsi, lors de l'utilisation d'un système interactif, une partie ou même l'intégralité du pipeline peut être exécutée plusieurs fois par seconde. Dans la plupart des cas, les interactions de l'utilisateur portent principalement sur les étapes de filtrage et d'encodage visuel.

L'objectif d'un système de visualisation d'information est de permettre à l'utilisateur de retirer du sens à partir de ce qui lui est présenté. Les différentes étapes suivies par l'utilisateur dans son exploration des données sont résumées dans le mantra de Shneiderman :

> Overview first, zoom and filter, then details-on-demand.
>
> (Ben Shneiderman [100])

Le système doit donc d'abord présenter une vue d'ensemble des données (*overview*), permettre de filtrer et agrandir interactivement, puis d'afficher les détails sur demande de l'utilisateur.

1.1.2 Encodage visuel

Les différentes possibilités pour représenter de l'information ont commencé à être recensées par Jacques Bertin [12, 13] au milieu du XX$^{\text{ème}}$ siècle. Il décrit des entités graphiques (points, lignes, aires, surfaces et volumes), appelées marques, qui constituent les briques de base de la visualisation. Chacune de ces marques est dotée d'un ensemble de variables visuelles pour décrire son apparence. L'encodage visuel consiste alors à choisir quelles marques et quelles variables visuelles utiliser pour représenter une information. Pour que l'encodage visuel soit le plus efficace possible, c'est-à-dire qu'il permette le passage de l'information sans distraction, il est nécessaire de connaître quelques phénomènes inconscients qui régissent notre perception visuelle.

Il faut d'abord savoir ce que l'on cherche à représenter. Il existe trois types d'attributs à encoder : les attributs *quantitatifs*, qui représentent une quantité et dont l'écart entre les valeurs a un sens, les attributs *ordinaux*, dont les valeurs peuvent être ordonnées sans pour autant représenter une quantité et les attributs *catégoriels* (ou *nominaux*), qui représentent des groupes mais pour lesquels tous les groupes se valent.

Toutes les variables visuelles ne sont pas aptes à représenter tous les types d'attribut. Par exemple, la forme permet de représenter facilement un attribut catégoriel (une forme par groupe), mais ne conduit pas d'information d'ordre et de magnitude nécessaire pour un attribut ordinal ou quantitatif. De même, la longueur

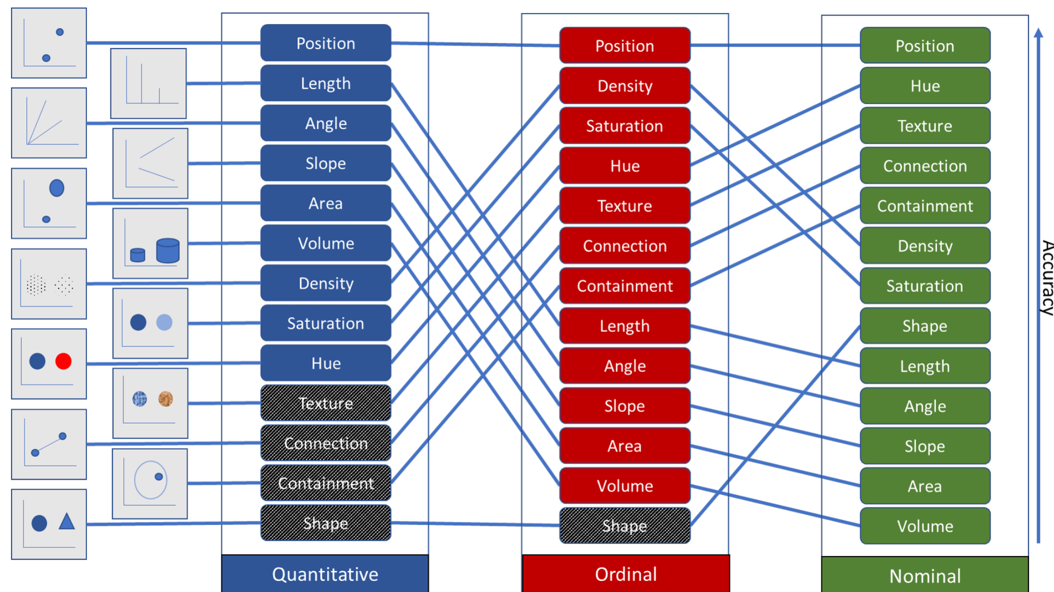


FIGURE 1.5 – Les variables visuelles couramment utilisées. Pour chaque type d'attribut, elles sont classées de la plus efficace (en haut) à la moins efficace (en bas). Le classement est issu des travaux de Cleveland & McGill [31], puis Card & Mackinlay [24]. Les variables en noir sont les variables contre-indiquées pour représenter chaque type d'attribut.

encode très bien les attributs quantitatifs puisqu'elle conduit directement une notion de quantité, mais est totalement inadaptée pour des attributs ordinaux ou catégoriels, l'impression de quantité troublant la perception de ces attributs. Card et al. [24] ont établi un classement des différentes variables en fonction de leur efficacité à représenter les différents attributs, comme le montre la Figure 1.5. Il est intéressant de noter que la position spatiale est la variable la plus efficace pour les trois types d'attributs et est donc toujours la première utilisée.

Il existe également un effet dit de "saillance visuelle" ou "popout", aussi appelé perception pré-attentive, qui permet de remarquer immédiatement un élément différent des autres si une de ses variables visuelles est différente. Cependant, toutes les variables visuelles ne donnent pas cet effet avec la même intensité. Par exemple, comme l'illustre la Figure 1.6, la couleur est plus efficace que la forme.

De plus, certaines variables visuelles interfèrent les unes avec les autres et ne peuvent donc pas être utilisées simultanément pour encoder des attributs différents. On parle alors de variables séparables s'il n'y a pas d'interférence entre les deux, comme par exemple la position et la teinte. Au contraire, on parle de variables intégrales s'il est impossible de séparer l'information encodée sur les deux variables. La largeur et la longueur sont par exemple deux variables intégrales, elle ne peuvent encoder qu'un seul attribut sous forme d'aire.

Enfin, la plupart des variables visuelles permettent de réaliser implicitement un regroupement des marques qui partagent la même valeur pour une variable

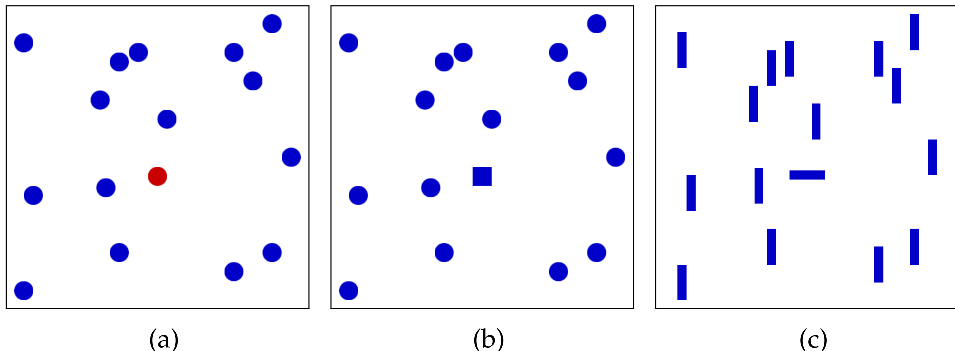


FIGURE 1.6 – Exemple d'effets de saillance visuelle. L'effet est plus ou moins prononcé suivant les variables visuelles mises en jeu. La couleur (a) et l'orientation (c) ont une saillance visuelle bien plus prononcée que la forme (b).

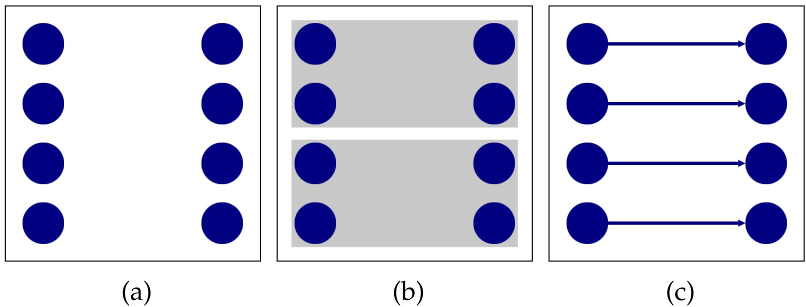


FIGURE 1.7 – Création de groupes visuels : (a) La proximité spatiale créé deux groupes. (b) L'ajout des deux enveloppes créé une séparation plus forte que la simple proximité spatiale. (c) Deux marques reliées sont visuellement associées malgré la distance qui les sépare.

(même couleur, même forme, même orientation, etc...). Il existe cependant trois constructions dont l'impact de regroupement est plus fort que l'association faite par les variables visuelles. Tout d'abord, le regroupement spatial (voir Figure 1.7a). Les marques proches les unes des autres sont perçues comme un groupe, en dépit de leur attributs visuels. Ensuite, l'inclusion (voir Figure 1.7b), c'est-à-dire l'intégration de marques au sein d'une enveloppe fermée. Enfin, la liaison (voir Figure 1.7c). Des marques reliées par un trait par exemple constituent aussi un groupe.

1.1.3 Idiomes de visualisation

Toutes ces variables visuelles et leurs propriétés offrent un arsenal très vaste pour concevoir des visualisations. L'espace des possibilités est donc très grand. Cependant, la plupart des combinaisons ne sont pas efficaces ou même sensées. C'est ce que Bertin appelle une "convention qui détruit la signification des données". Au fil des années, certaines combinaisons se sont imposées comme particulièrement

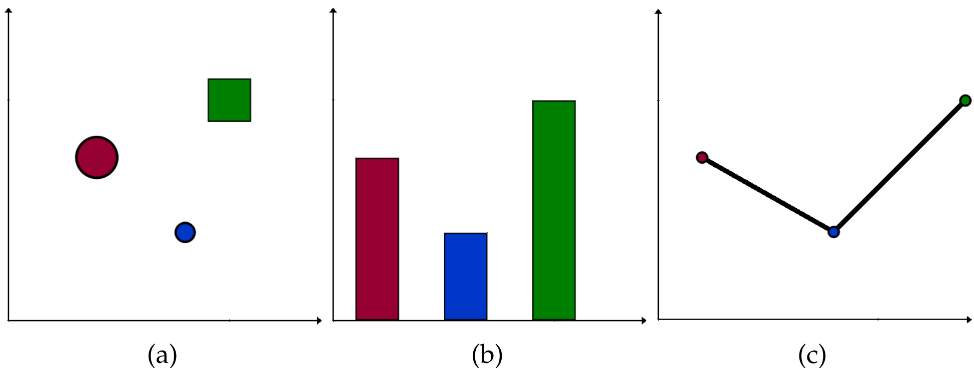


FIGURE 1.8 – Exemples d'idiomes de visualisation. (a) Un scatterplot à 3 éléments. Deux attributs quantitatifs sont encodés en abscisse et ordonnée. La taille, la couleur et la forme peuvent également encoder de attributs suplémentaires. (b) Un histogramme horizontal. (c) Une courbe temporelle.

efficaces ou populaires. C'est ce que l'on appelle des *idiomes de visualisation*.

L'un des idiomes les plus connus est la visualisation dite en "nuage de points", ou "scatterplot" en anglais (voir Figure 1.8a). Dans cet idiome, chaque élément est représenté par un point (marque 0D) et deux attributs quantitatifs sont encodés, l'un avec la position horizontale (abscisse), l'autre avec la position verticale (ordonnée). Il est possible d'encoder plus d'attributs en utilisant les variables restantes, comme la couleur, la taille ou la forme.

L'idiome "histogramme" utilise quant à lui des marques 1D en forme de rectangle pour encoder un attribut quantitatif (la taille du rectangle) et un attribut catégoriel (la position) (voir Figure 1.8b). On trouve cette visualisation le plus souvent avec des barres verticales, mais elle apparaît aussi de manière horizontale. Une variation appelée histogramme empilé permet d'encoder un deuxième attribut catégoriel en empilant plusieurs rectangles différenciés par la couleur.

Un autre idiome bien connu est la courbe temporelle (voir Figure 1.8c). Elle permet d'encoder un attribut quantitatif en ordonnée qui varie au fil du temps (en abscisse). La valeur à chaque pas de temps est représentée par un point. Deux points consécutifs sont reliés par un trait afin de constituer la courbe finale.

Pour représenter des données multidimensionnelles, les coordonnées parallèles représentent les différentes dimensions par des axes verticaux (parallèles, ce qui donne son nom à cet idiome). Chaque élément est alors encodé sous forme d'une ligne qui coupe chaque axe à l'endroit correspondant à sa valeur pour cette dimension. Autrement dit, la valeur de chaque élément pour chaque dimension est encodée par un point sur l'axe correspondant. Les points d'un même élément sont ensuite reliés par une ligne.

On peut remarquer que les mêmes données peuvent être encodées de manière différente. Par exemple, les données hiérarchiques (ou arbres) ont entraînées de nombreuses représentations, dont certaines sont visibles en Figure 1.9. L'arbre peut

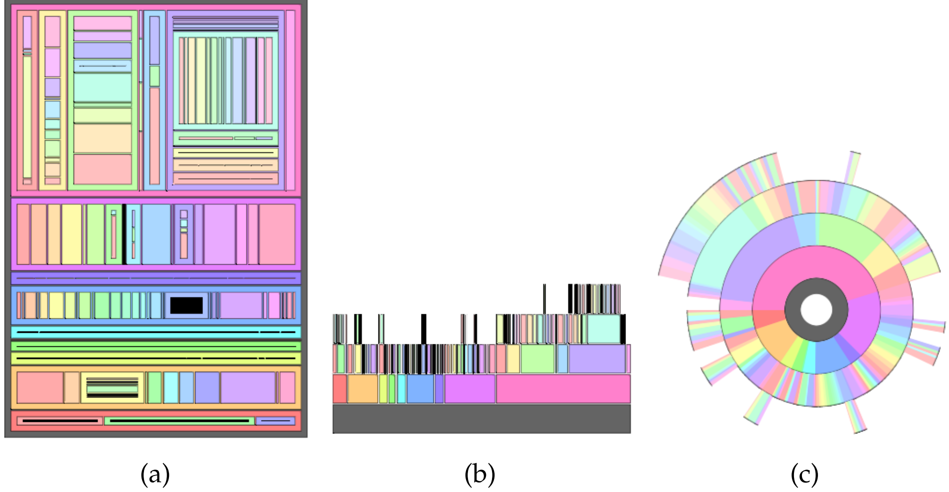


FIGURE 1.9 – Trois visualisations du même arbre : (a) Treemap, qui utilise l'inclusion des éléments pour représenter la hiérarchie, (b) Icicle, qui utilise l'adjacence des rectangles et (c) Sunburst, qui utilise l'adjacence de secteurs angulaires.

être représenté sous forme de graphe, en utilisant des points pour les sommets et des liens pour relier les fils à leur père. Il est aussi possible d'utiliser la notion d'inclusion pour encoder la hiérarchie. Avec des rectangles, on a alors un "treemap", avec des cercles, un "circular treemap". L'adjacence peut aussi être utilisée avec des rectangles (icicle) ou des cercles (sunburst).

Pour des données géographiques, la position et la taille sont souvent contraintes par le fond de carte. La couleur est alors utilisée pour encoder l'information. Si les couleurs remplissent des régions discrètes de la carte (qui correspondent souvent à un découpage administratif), on parle de carte choroplèthe, du grec χώρος : "zone / région" et πληϑαίν : "multiple". Si au contraire, l'information est un champ continu de valeurs, la couleur agit comme un calque supplémentaire de la carte. On parle alors de "carte de chaleur" (ou *heatmap*).

Un idiome vise à exploiter la position et la taille sur des données géographiques : le cartogramme. Il s'agit de déformer la carte pour conserver la position relative des pays, mais modifier leur aire pour encoder l'information. Si au contraire, la position et la forme des pays sont conservées, mais que leur taille est diminuée (ce qui entraîne la perte de la connexité), on parle de cartogramme non-contigu.

1.2 Big Data

Ces dernières années ont vu la convergence de trois révolutions : une révolution technologique, une révolution des données et une révolution des usages.

La **révolution technologique** est constituée de plusieurs éléments. L'augmenta-

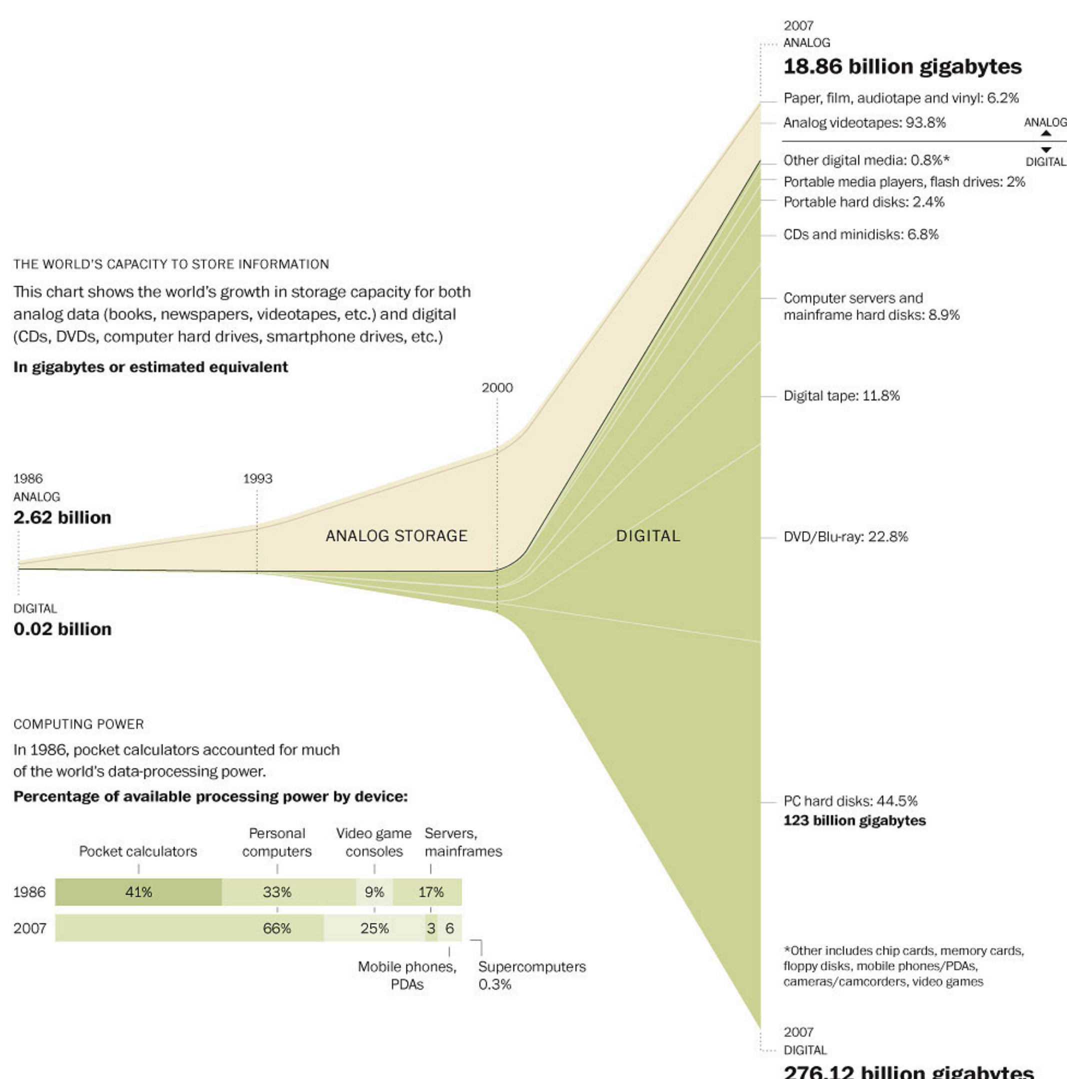


FIGURE 1.10 – Diagramme montrant l'augmentation de la capacité de stockage de l'information. Source : http://www.washingtonpost.com/wp-dyn/content/graphic/2011/02/11/GR2011021100614.html

tion de la puissance des processeurs entraîne une explosion des capacités de calcul exploitables. Les progrès dans les technologies de stockage permettent aujourd'hui de conserver d'énormes quantités de données à moindre coût. Les vitesses de transfert réseau ont aussi suivi ce mouvement. En conséquence, l'accès à Internet s'est grandement répandu, permettant l'interconnexion de milliards de personnes et entreprises. Les téléphones portables sont devenus omniprésents, permettant un accès rapide et à tout moment au réseau.

La deuxième révolution est une **révolution des données**. La quantité de données générées est en constante augmentation, comme l'illustre le diagramme de la Figure 1.10. Grâce à la révolution technologique, elles peuvent maintenant être

systématiquement conservées puis traitées. Les données les plus récentes peuvent aussi être comparées aux données anciennes, puisque tout l'historique peut être conservé.

Enfin, la troisième révolution est une **révolution des usages**. Le nombre d'usagers des nouvelles technologies augmente lui aussi continuellement, ce qui accroît les sources potentielles de données. De plus, la quantité de données stockées rend possible de nouveaux usages qui ont besoin d'une masse critique de données, comme des algorithmes d'apprentissage automatique.

Ces trois révolutions s'entretiennent dans une boucle de rétroaction. La technologie permet de nouveaux usages qui génèrent de nouvelles données pour améliorer la technologie, et ainsi de suite. Ensemble, ces trois révolutions constituent ce que l'on nomme « Big Data ». Le phénomène est souvent caractérisé par une liste de termes appelée "les 3 V" : volume, vélocité, variété. Ces trois caractéristiques ont été introduites en 2001 par Laney [71], puis popularisées par l'agence Gartner [15].

Le volume désigne l'énorme quantité de données à traiter. Même si ce problème a toujours été présent dans de nombreux domaines, il est fortement exacerbé dans le Big Data.

La vélocité désigne la vitesse de génération et donc de collecte des données. Il peut aussi en résulter une rapide obsolescence des données collectées, ayant été remplacées par de nouvelles données plus pertinentes. Il faut alors être capable de traiter les données sans accumuler de retard et tant qu'elles sont pertinentes.

Enfin, la variété désigne le caractère hétérogène et non structuré des données collectées. Les nombreuses sources de données disponibles n'utilisent pas les mêmes formats et fournissent des données de type différent. Il faut alors les recouper pour obtenir une information plus riche.

On peut aussi trouver des caractérisations du Big Data qui utilisent jusqu'à 8, voire 12 "V" [2]. Nous retiendrons ici surtout un des plus récurrents et celui qui est le plus en rapport avec cette thèse : Visualisation. La visualisation des données est un enjeu majeur du Big Data, à la fois pour trouver les informations pertinentes mais aussi pour les présenter par la suite. Il faut donc faire appel à la fois à la visualisation descriptive et analytique.

Avec le développement de l'analyse de données massives, de plus en plus d'entreprises désirent mettre à profit les données qu'elles ont pu collecter. Au vu des quantités de données en jeu, il est nécessaire de s'orienter vers des solutions capables de gérer de grandes quantités de données. Mais les solutions classiques utilisant des plateformes de calcul haute performance ou des supercalculateurs sont trop complexes à mettre en place et à maintenir pour des petites structures. Il est donc nécessaire d'avoir des solutions plus simples d'utilisation. Ces solutions doivent constituer un compromis entre la complexité de mise en œuvre et les performances finales.

2. Et même 42 : `http://www.kdnuggets.com/2017/04/42-vs-big-data-data-science.html`.

Pour satisfaire ce besoin, de nouvelles solutions ont vu le jour, la plus populaire étant Hadoop. Cet ensemble de logiciels répond à deux besoins majeurs pour le traitement de données massives : le stockage distribué avec le système de fichiers HDFS et le calcul distribué avec le paradigme MapReduce. Ce système est conçu pour être facilement déployable et utilisable sans avoir de connaissances particulière en calcul distribué, ce qui le rend accessible à un plus grand public.

Aujourd'hui, il est possible de disposer facilement et rapidement d'un cluster Hadoop en quelques minutes. Avec l'apparition du "cloud", de nombreuses entreprises proposent de mettre à disposition un cluster virtuel, disponible en quelques clics. Dans le même temps, des distributions Linux intégrant Hadoop et d'autres outils pour le traitement de données massives sont proposées, ce qui facilite le déploiement des machines du cluster.

1.3 Contributions

Au sein du domaine de la visualisation d'information, la révolution du Big Data apporte à la fois de nouveaux problèmes, comme la quantité de données à visualiser et leur vitesse de collecte, mais aussi de nouvelles possibilités, notamment à travers les solutions technologiques émergentes. Dans cette thèse, nous proposons des solutions pour permettre la visualisation interactive de grandes masses de données au sein de l'écosystème Big Data.

La première partie expose le contexte général. Le chapitre 2 explique quelques connaissances préalables nécessaires à la compréhension de la thèse. Le chapitre 3 est consacré à l'état de l'art concernant les moyens de visualiser des données massives en assurant d'une part la scalabilité de la perception humaine et d'autre part celle des performances de calcul.

La deuxième partie concerne les concepts et techniques mis en œuvre dans le cadre de cette thèse. Tout d'abord, le chapitre 4 propose une méthode générale pour visualiser les données massives en scindant le pipeline de visualisation en deux parties. Ensuite, le chapitre 5 est consacré aux algorithmes d'agrégation utilisés dans cette thèse. Un algorithme séquentiel est présenté, ainsi que son utilisation pour créer une abstraction multi-échelle. Nous proposons aussi deux approches pour rendre cet algorithme distribué. Enfin, le chapitre 6 présente la réalisation d'une bibliothèque de visualisation sur client léger. Nous nous appuyons sur les travaux de Bertin [12, 13] pour définir un système de *marks* et de *connections* qui permet de décrire simplement des visualisations.

Enfin, la dernière partie présente les applications de ces techniques pour la visualisation de données massives. Le chapitre 7 est consacré à la visualisation de cartes de chaleur à grande échelle. Nous présentons une approche de traitement par lots et une autre en flux. Le dessin final est assuré par un algorithme qui permet de borner le temps de rendu en baissant éventuellement la résolution de l'image pour conserver l'interactivité du système. Dans ce chapitre, nous adressons le volume et la

vélocité. Le chapitre 8 présente une application à la visualisation de grands graphes. Une technique d'agrégation d'arêtes est ajoutée à celles présentées précédemment, ainsi qu'un algorithme de faisceautage en temps réel. Ce système permet de gérer le volume de données associé aux grands graphes. Dans ces applications, nous gérons donc 2 des 3 "V" du Big Data : le volume et/ou la vélocité. Le 3$^{\text{ème}}$ "V", la variété, est géré en amont lors de la collecte des jeux de données.

Les contributions présentées dans cette thèse ont fait l'objet de trois publications en conférence internationale. La bibliothèque Fatum a été présentée à la conférence *IV* 2015 [86]. Notre approche pour la visualisation de carte de chaleur à grande échelle a été présentée au symposium *LDAV* 2015 [88]. De plus, l'adaptation de cette approche pour le calcul en flux a été présentée à la conférence *IV* 2017 [87]. Enfin, le système de visualisation de graphes *Cornac* a fait l'objet d'un article soumis au journal *IEEE Transactions on Big Data*.

2
Préliminaires

2.1 Paradigmes pour le Big Data

Dans cette section nous présentons les fondamentaux liés au Big Data utilisés dans cette thèse.

2.1.1 Architecture distribuée

Pour permettre de gérer les quantités de données induites par le Big Data, il est peu commode et très onéreux d'utiliser une unique machine extrêmement puissante, dont la panne signifierait l'arrêt du service. La solution privilégiée est d'utiliser plusieurs machines plus modestes qui collaborent au travers du réseau. Cet ensemble de machines est appelé un "cluster" ou grappe.

Ce type d'architecture utilisée en conjonction avec les logiciels adéquats permet de bénéficier de plusieurs avantages. Tout d'abord, il est plus facile d'assurer une meilleure tolérance aux pannes grâce à la réplication des données entre plusieurs machines. On définit alors un facteur de réplication R, qui indique le nombre total de copies des données. La panne de moins de R machines n'entraîne aucune perte de données. Après une panne, les données répliquées moins de R fois peuvent être recopiées sur d'autres machines pour retrouver le facteur de réplication initial. De plus, cette tolérance s'applique également à la perte de puissance de calcul, puisque l'impact de la perte d'une machine est inversement proportionnel au nombre de machines du cluster. Les machines restantes devront se partager la charge de la machine en panne. La perte d'une seule machine est donc plus dommageable dans un cluster de 10 machines que dans un cluster de 100.

Ensuite, ce type d'architecture permet d'ajouter facilement des ressources supplémentaires au besoin en ajoutant de nouvelles machines au cluster lorsque la charge devient trop importante. Au contraire, si les machines sont sous utilisées, il est possible de diminuer la taille du cluster. Cette capacité à adapter les ressources suivant la charge est ce que l'on appelle l'élasticité. Cette caractéristique est d'autant plus intéressante qu'aujourd'hui, de nombreux systèmes sont gérés directement grâce à

l'informatique en nuage, en mutualisant et virtualisant les ressources utilisées. Dans ce contexte, l'élasticité n'induit pas d'achat ou de vente de matériel pour l'utilisateur et lui permet de ne payer que pour des ressources utilisées.

2.1.2 MapReduce

Le paradigme MapReduce a été inventé par Dean et al. [36] en 2008. Ce paradigme permet de distribuer facilement n'importe quel calcul en l'exprimant sous la forme d'une suite d'opérations `map` et `reduce` fournies par l'utilisateur. Les données à traiter sont structurées en un ensemble de paires comprenant une clé et une valeur, notés $(k, v), k \in K, v \in V$, sur lesquelles vont agir les deux opérations.

L'opération `map` prend en entrée une paire (k, v) et la transforme en une liste de paires (k', v'). L'opération `reduce` quant à elle prend en entrée une clé k' et la liste des valeurs v' associées. On peut formaliser ces opérations ainsi :

$$\begin{aligned} map(k, v) &\rightarrow R_m = [(k', v')] \subset K' \times V' \\ reduce(k', [v' \mid (k', v') \in R_m]) &\rightarrow R_r = [(k'', v'')] \subset K'' \times V'' \end{aligned}$$

Les listes retournées par les deux opérations peuvent tout à fait être vides. Par contre, la fonction `reduce` n'est appelée que pour les clés qui correspondent à au moins une valeur. Par souci de commodité, au lieu de manipuler une liste en y ajoutant les paires à retourner, on utilisera la fonction `emit(k,v)` qui permet d'ajouter une paire (k, v) à l'ensemble des paires à retourner. On remarquera que les types des clés et des valeurs en entrée et sortie des fonctions peuvent être différents. Cependant, il est nécessaire que les types de sortie de la fonction `map` et d'entrée de la fonction `reduce` soient identiques.

Le programme le plus simple que l'on peut écrire en MapReduce est le programme identité, qui retourne les paires d'entrée telles quelles :

Algorithme 2.1 : Programme MapReduce Identité

```
1 Map(K k, V v)
2 |  emit(k,v)
3 Reduce(K k, V[] l)
4 |  for V v in l do
5 |  |  emit(k,v)
```

Le fonctionnement de MapReduce est décomposé en trois phases : *Map*, *Shuffle* et *Reduce*. La phase de *Map* consiste à appliquer la fonction `map` fournie par l'utilisateur à chacune des paires du jeu de données à traiter. La phase de *Shuffle* groupe les valeurs pour chaque clé afin de créer les listes pour la phase suivante. Lors de cette phase, les données émises par la phase de *Map* doivent transiter sur le réseau. Enfin, lors de la phase de *Reduce*, la fonction `reduce` fournie par l'utilisateur est appliquée à chacune des clés et aux valeurs correspondantes. L'ensemble de ces trois étapes

constitue ce que l'on appelle un *job* MapReduce. Pour des traitements complexes, il peut être nécessaire d'enchaîner plusieurs *jobs*.

Les fonctions `map` et `reduce` ne nécessitant qu'une paire en entrée, elles peuvent être appliquées en parallèle à tout le jeu de données. L'utilisateur n'a donc qu'à exprimer son calcul au moyen de ces fonctions pour permettre de le paralléliser.

2.1.2.1 Exemple : comptage d'occurrences

L'exemple le plus répandu de l'usage de MapReduce est le comptage d'occurrences de mots dans un texte. Dans ce cas, les paires d'entrée sont constituées du numéro et du contenu de chaque ligne d'un texte. La fonction `map` sépare chaque ligne en mots et émet pour chaque mot m une paire $(m, 1)$ qui dénote que le mot a été observé une fois. Lors de la phase de *Shuffle*, tous les 1 correspondants au même mot sont groupés dans une liste. Le nombre d'occurrences calculé par la fonction `reduce` est donc la longueur de cette liste.

Algorithme 2.2 : Comptage d'occurrences en MapReduce.

Input : Un texte
Output : Le nombre d'occurrences de chaque mot

```
1 Map(Int n, String line)
2   for word in split(line) do
3     emit(word,1)
4 Reduce(String word, Int[] occurrences)
5   emit(word,length(occurrences))
```

2.1.3 Pregel

Le paradigme de programmation Pregel [76] permet d'exprimer des calculs distribués sur des graphes [1]. Basé sur le paradigme *Bulk Synchronous Parallel* (BSP), introduit par Valiant [107], Pregel est organisé en phases de calcul séparées par des barrières de synchronisation, durant lesquelles des messages sont échangés.

Pregel s'exécute sur un graphe dont les sommets et les arêtes possèdent un identifiant unique et un attribut modifiable défini par l'utilisateur. Lors de chaque phase, les sommets du graphe peuvent recevoir les messages qui leur ont été envoyés lors de la phase précédente, modifier leur attribut et générer des messages à envoyer aux autres sommets. Suivant les besoins, les conditions d'arrêt du calcul peuvent être définies de trois manières différentes :

— Soit par un nombre maximum d'itérations,

1. Le paradigme Pregel est nommé d'après la rivière Pregel qui coule à Kaliningrad en Russie (anciennement Königsberg en Prusse-Orientale), dont les ponts sont à l'origine de la théorie des graphes à travers le fameux problème des sept ponts de Königsberg.

— Soit par un vote des sommets à chaque itération,

— Soit par l'absence de messages à la fin d'une itération.

Lors de chaque phase, le calcul pour chaque sommet est indépendant, il peut donc s'exécuter en parallèle. Tous les calculs se font selon la perspective d'un sommet. Pour cette raison, le paradigme Pregel est souvent décrit comme *"Think like a vertex"*. Pregel peut aisément être implémenté en utilisant les primitives de MapReduce. En effet, une itération de Pregel peut être vue comme la succession de trois opérations : d'abord la génération de nouveaux messages en fonction de l'état de chaque sommet (*Map*) puis l'échange des messages (*Shuffle*) et enfin la modification de l'état de chaque sommet suivant les messages reçus (*Reduce*).

2.1.3.1 Exemple : calcul de composantes connexes

Dans un graphe, une composante connexe désigne un ensemble de sommets tel qu'il est possible d'atteindre n'importe quel sommet de l'ensemble depuis n'importe quel autre. Le calcul de ces composantes en Pregel repose sur le principe de la propagation : chaque sommet propage à ses voisins l'identifiant de sa composante connexe. Chaque sommet rejoint alors la composante avec l'identifiant le plus petit. Au début de l'algorithme, chaque sommet constitue sa propre composante connexe, qui utilise comme identifiant l'identifiant du sommet. A la fin de l'algorithme, chaque composante connexe a pour identifiant celui du plus petit sommet qu'elle contient.

Pour implémenter un algorithme en Pregel, nous définissons une fonction `compute` qui contient le programme exécuté par chaque sommet et `sendMessage` qui permet d'envoyer un message à un autre sommet. Dans le cas du calcul de composantes connexes, l'état de chaque sommet est constitué de l'identifiant de sa composante connexe cc, initialisé avec l'identifiant du sommet.

Algorithme 2.3 : Calcul de composantes connexes avec Pregel.

```
1 compute (messages)
2 |  min_cc = min(messages)
3 |  if min_cc < cc then
4 |  |  cc = min_cc
5 |  |  for each neighbour n do
6 |  |  |  sendMessage(n,cc)
```

Les messages entre sommets ne sont générés qu'après un changement de composante connexe. Lorsque tous les sommets ont rejoint leur composante connexe, plus aucun message n'est envoyé et le calcul prend fin.

2.1.4 Types de calcul

On peut différencier deux types de calcul : batch (par lots) ou en streaming (en flux).

Le calcul en batch traite l'ensemble des données à la fois. Il requiert donc potentiellement des ressources proportionnelles au volume total de données à traiter. Ce type de calcul permet de traiter un grand volume de données, mais aucun résultat n'est disponible avant la fin du calcul. Les calculs en batch ont donc une grande latence, souvent plusieurs heures. Cette latence rend impossible de répondre rapidement à une requête d'un utilisateur.

A l'inverse, les calculs en flux sont optimisés pour avoir la latence la plus faible possible, jusqu'à quelques millisecondes. Pour cela, chaque donnée est traitée au fur et à mesure de la collecte. Les systèmes qui utilisent du calcul en streaming sont conçus pour fonctionner en permanence, afin de traiter la donnée dès son arrivée.

2.1.4.1 Modèles pour le calcul en flux : one-at-a-time et micro batch

Il existe de plus deux modèles d'exécution pour le calcul en flux. Le modèle dit *one-at-a-time* traite chaque nouvel enregistrement individuellement. Ce modèle est capable de traiter très rapidement les données reçues, ce qui permet d'obtenir des résultats avec très peu de latence. Le modèle *micro-batch* groupe les nouveaux enregistrements pour les traiter par paquet. Tous les enregistrements reçus pendant un intervalle de temps appelé intervalle de micro-batch sont traités ensemble de la même manière que serait traité un jeu de données complet. L'intervalle de temps utilisé peut varier de quelques millisecondes à plusieurs minutes selon les besoins. C'est cet intervalle qui détermine la latence totale d'un système utilisant le modèle *micro-batch*. Cette latence est généralement plus importante qu'avec le modèle *one-at-a-time*, mais le modèle *micro-batch* permet de traiter plus de données à la fois.

2.1.5 Lambda architecture

Le calcul en batch et le calcul en streaming présentent des avantages et inconvénients opposés. D'un côté, le calcul en batch permet de traiter de larges volumes de données mais entraîne une grande latence. De l'autre, le calcul en streaming a une faible latence mais traite des volumes de données plus limités.

Nathan Marz propose donc de combiner ces deux types de calcul pour bénéficier de leur avantages et contrebalancer leurs faiblesses au sein d'une même architecture qu'il appelle *Lambda Architecture* [77]. L'objectif de cette architecture est de permettre de traiter de grands volumes de données, tout en assurant de pouvoir répondre en temps réel à un ensemble de requêtes choisies à l'avance. Pour cela, le système maintient des données précalculées appelées "vues".

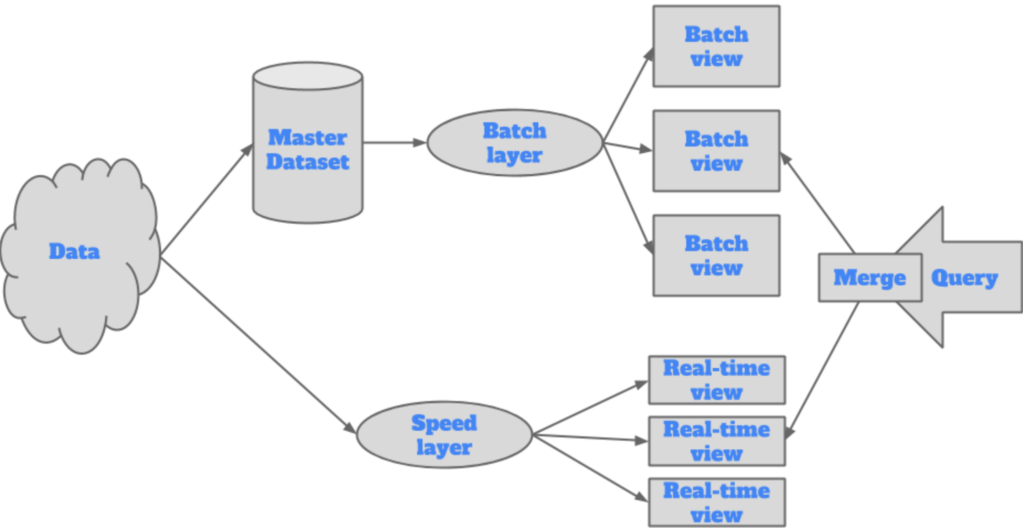


FIGURE 2.1 – Schéma de fonctionnement de la *Lambda Architecture*. Lors de la réception de nouvelles données, elles sont ajoutées au *master dataset* pour être traitée par le *batch layer* et envoyées au *speed layer* pour être ajoutées aux vues temps réel.

La *Lambda Architecture* est composée d'un système de traitement en batch et d'un système en streaming. Chacun alimente son ensemble de vues. Lorsque les données sont collectées par la *Lambda Architecture*, elles suivent deux chemins parallèles. D'une part, elles sont stockées pour être traitées en batch et d'autre part elles sont injectées dans le système de traitement en streaming. Le schéma général du fonctionnement de la *Lambda Architecture* est visible sur la Figure 2.1.

2.2 Outils et implémentations utilisés

2.2.1 Hadoop

Le framework Hadoop est devenu au fil des années le standard *de facto* dans le domaine du Big Data. Au départ simple implémentation open source du paradigme MapReduce, il s'est depuis enrichi et a inspiré de nombreux autres outils. On parle même aujourd'hui d'*écosystème* Hadoop pour désigner l'ensemble composé d'Hadoop et des multiples outils complémentaires.

2.2.1.1 HDFS

HDFS est un système de fichiers distribué. Il permet le stockage de fichiers très volumineux en les répartissant sur les machines du cluster. Chaque fichier est

découpé en blocs de taille fixe (par défaut 128Mo) puis chaque bloc est assigné à une machine. Un noeud maître est responsable de conserver l'emplacement de tous les blocs de chaque fichier ainsi stocké dans HDFS.

De plus, HDFS permet de répliquer les blocs pour permettre une redondance de l'information et éviter les pertes en cas de panne d'une des machines du cluster. Chaque bloc à répliquer est donc stocké plusieurs fois. Avec un facteur de réplication R, aucune donnée ne sera perdue tant que moins de R machines sont en panne.

HDFS permet la plupart des opérations classiquement disponibles au sein d'un système de fichiers : création, suppression, déplacement d'un fichier / dossier. En revanche, il ne permet pas de lire et d'écrire à n'importe quel emplacement dans un fichier. En effet, les opérations supportées par HDFS doivent pouvoir être réalisées de manière distribuée et asynchrone. Il est donc uniquement possible d'ajouter des données en fin de fichier.

2.2.1.2 HBase

HBase est une base de données construite en tant que surcouche pour HDFS. Conçue également pour pouvoir gérer de grandes quantités de données, ce n'est pas une base de données relationnelle (ou SQL), comme la plupart des bases de données utilisées aujourd'hui. Elle fait donc partie des bases de données dites "NoSQL", pour Not Only SQL.

Elle implémente un modèle de données orienté colonnes, développé pour Google BigTable [27]. Ce modèle proche du modèle relationnel utilise un format $(Clé, Valeur)$ qui permet de ne stocker de l'information que lorsque celle-ci existe bel et bien (contrairement au modèle relationnel, où toutes les colonnes d'une table doivent être remplies).

HBase est basé sur un modèle de cluster maître esclave, comme HDFS. Le maître a la responsabilité de savoir quelles sont les données stockées sur chaque noeud esclave et où elles sont répliquées.

Les données stockées dans HBase sont indexées par leur clé. Il n'existe pas de fonction de recherche, il est uniquement possible de récupérer les données correspondant à une clé (opération `Get`) ou un ensemble de clés contiguës (opération `Scan`). Ceci rend l'utilisation d'HBase plus complexe que celle d'une simple base relationnelle. Cependant, plusieurs logiciels comme *Apache Pheonix*, *Impala* ou *Drill* permettent d'utiliser SQL comme langage de requête pour HBase.

2.2.2 Spark

Spark [117] est un framework de calcul distribué qui vise à accélérer les traitements en utilisant au maximum la mémoire vive. Contrairement à Hadoop, où les données sont stockées sur disque, Spark les conserve en mémoire vive. Son architecture est basée sur le principe des RDDs (Resilient Distributed Datasets) [116]. Un

RDD consiste en un ensemble de partitions qui contiennent des données, ainsi que la chaîne d'opérations nécessaire pour les calculer. Les RDD sont des structures de données paresseuses, c'est-à-dire que les calculs ne sont effectués que si le contenu de la structure est effectivement utilisé. De plus ce calcul peut n'être effectué que pour certaines partitions. Cela permet de rendre la structure de données tolérante aux pannes, puisqu'en cas de perte de données, il est facile de recalculer les partitions perdues en rejouant l'historique des opérations à partir de données antérieures qui ont été conservées.

Contrairement à MapReduce qui ne permet de manipuler que des données au format $(Clé, Valeur)$, les RDD peuvent contenir n'importe quel type de donnée, ce qui leur donne une plus grande flexibilité.

Plusieurs opérations sont définies sur les RDD, dont les opérations map et reduce, ce qui donne à Spark la possibilité d'exprimer et d'effectuer les mêmes calculs que MapReduce. Les autres opérations sont des cas particuliers de map et reduce définis pour faciliter le développement et la lecture de programmes Spark, ou des implémentations plus efficaces d'opérations courantes.

De plus Spark intègre plusieurs autres frameworks spécialisés dont certains sont utilisés au sein de cette thèse. D'une part, GraphX est orienté pour les calculs sur les grands graphes. Il modélise les graphes sous forme de plusieurs RDD. Il contient une implémentation du paradigme Pregel. D'autre part, Spark Streaming permet le calcul en flux avec le modèle micro-batch. Il définit un flux de données où chaque pas de temps prend la forme d'un RDD. Il est ainsi possible de traiter chaque pas de temps de la même manière qu'avec un programme Spark en batch.

3
État de l'art

Lorsque la taille des données à visualiser augmente, les techniques de visualisation classiques se heurtent à deux problèmes fondamentaux, soulignés par Elmqvist et Fekete dans [44] :

— **Perception** : Le nombre d'entités visuelles rend difficile, voire impossible d'avoir une vue d'ensemble des données. Il devient fréquent que plusieurs points de donnée soient représentés sur un même pixel. C'est ce que l'on appelle le problème d'occlusion, ou overplot. L'aspect final de la visualisation dépend alors de l'ordre dans lequel les données sont dessinées. Comme le montre [68], cet effet peut donner des visualisations et interprétations très différentes à partir de données pourtant identiques.

— **Performance** : La quantité de données à afficher entraîne une baisse significative du nombre d'images affichées par seconde, ce qui nuit à l'interactivité de la visualisation. Or cette interactivité est essentielle pour accéder au détail des données.

Combinés, ces deux problèmes rendent difficile de respecter le mantra de Shneiderman [100]. Pour visualiser des données massives, il faut donc trouver des techniques pour pallier ces deux problèmes à la fois.

Dans cette section, nous présentons les différentes techniques existantes pour résoudre d'un côté le problème de la scalabilité de perception et de l'autre celui de la scalabilité de performance.

3.1 Scalabilité de perception

L'occlusion se produit lorsque plusieurs points de donnée se chevauchent dans la visualisation. Dans le cas le plus extrême, ils peuvent occuper le même pixel, mais le fait d'être proches est suffisant pour gêner la visualisation. Une partie des données est alors cachée, ce qui empêche de les prendre en compte lors de l'interprétation. De plus, en changeant l'ordre dans lequel les données sont placées, les données visibles

et cachées changent, ce qui change les interprétations possibles de la visualisation. La Figure 3.1 montre le phénomène sur le jeu de données de John Snow (voir le chapitre 1) et montre l'effet d'occlusion sur une visualisation en nuage de points.

L'occlusion étant un problème connu depuis longtemps, de nombreuses méthodes existent pour y remédier. Ellis et Dix [42] recensent ces méthodes et les évaluent suivant des critères comme la capacité à représenter des grands jeux de données ou à encoder la densité.

La méthode la plus simple consiste à changer l'apparence des points de donnée. Il est par exemple possible de diminuer la taille des points pour éviter le recouvrement. Cette solution est assez efficace s'il y a peu de points, mais comme il n'est pas possible de diminuer la taille des points à l'infini, elle ne fonctionne plus lorsque le nombre de points augmente. De plus, cela ne change rien au problème si plusieurs points partagent exactement les mêmes coordonnées. Il faut aussi savoir que changer la taille des points perturbe la perception des autres variables visuelles, notamment la couleur. Une autre méthode populaire consiste à changer l'opacité des points pour révéler les points recouverts. Mais il suffit d'un petit nombre de chevauchements pour que l'opacité ne soit plus efficace [42, 57]. Une autre technique appelée *jitter* [1] consiste à introduire un déplacement des points pour retirer l'occlusion [105]. Encore une fois, l'efficacité de cette méthode diminue fortement avec des grands nuages de points.

Changer le type de visualisation peut aussi permettre d'éviter l'occlusion. Par exemple, la Figure 3.1 montre une carte de chaleur affichant la densité des points. Ce type de visualisation permet de résoudre le problème d'occlusion, puisqu'elle ne représente pas de points, mais un champ continu. L'information est encodée par une échelle de couleurs, dont le choix influe sur ce qu'il est possible de percevoir dans la visualisation. Cette technique peut par exemple être utilisée pour visualiser la densité de noeuds [109, 121] ou d'arêtes [23] d'un graphe.

D'une manière générale, les visualisations dites space-filling, comme les treemaps ou sunburst, sont conçues pour utiliser entièrement l'espace disponible sans chevauchement et donc ne génèrent pas d'occlusion [46]. Parmi ce type de visualisation, on trouve les visualisations orientées pixel, qui réduisent chaque entité à un unique pixel [66]. La disposition des pixels peut ensuite s'appuyer sur une courbe fractale pour optimiser l'utilisation de l'espace. Si la position spatiale a une importance, comme dans le cas de données géographiques, il est possible de déformer l'espace pour donner à chaque zone suffisamment de pixels pour représenter l'ensemble des données [67].

Des techniques de déformation spatiale peuvent aussi être utilisées pour résoudre l'occlusion localement. La technique la plus connue est le *fisheye* [25, 49, 97], qui déforme l'espace pour agrandir une zone d'interêt. Cette technique a donné le principe de l'interaction par *lentilles*, qui permettent d'appliquer une déformation ou un filtre à une zone de la visualisation [103, 104, 115].

1. Ce terme est couramment traduit par *gigue* dans d'autres domaines, mais cette traduction ne semble pas être usitée en visualisation.

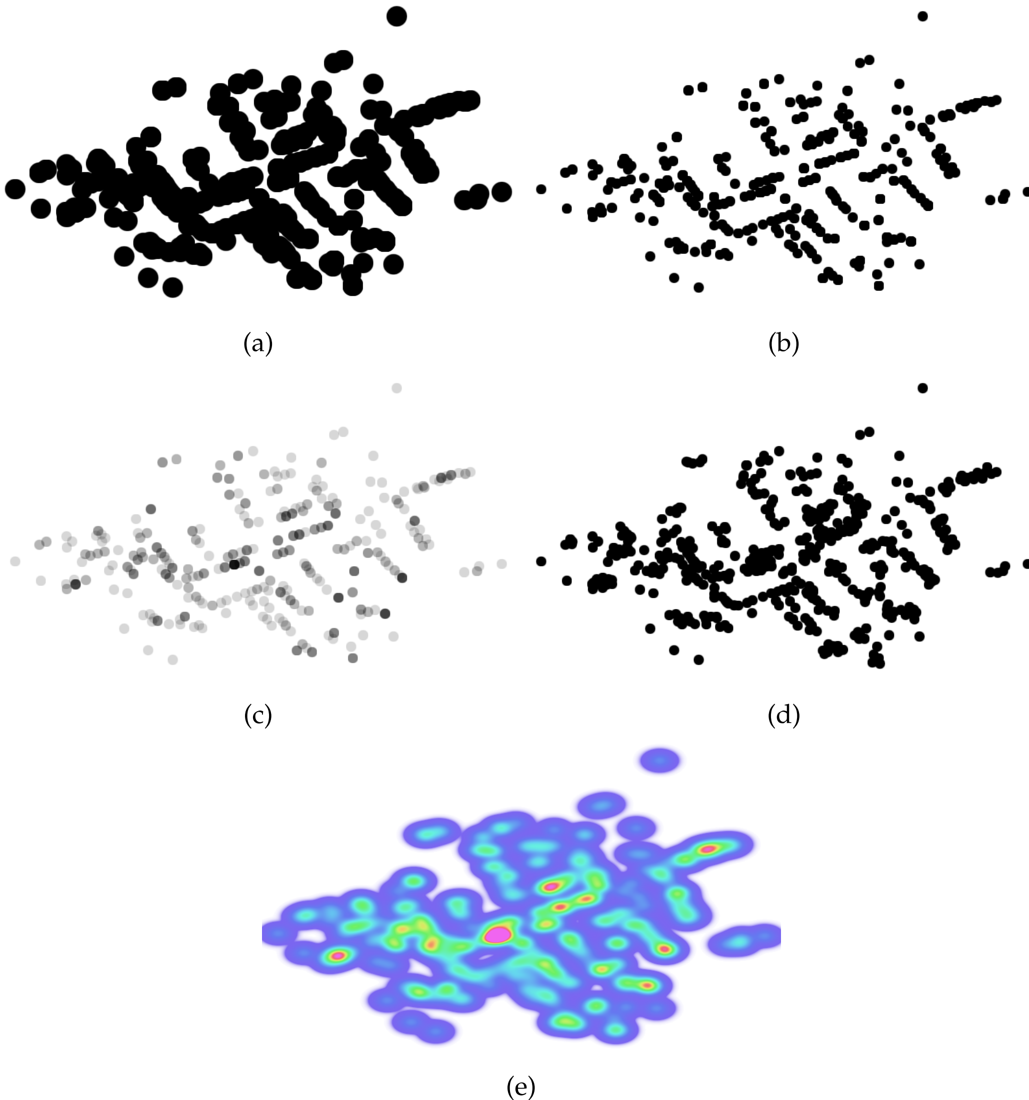


FIGURE 3.1 – Effet de l'occlusion sur la visualisation de l'épidémie de choléra de 1854. (a) Visualisation de départ. La taille des points provoque de l'occlusion. (b) Diminuer la taille des points permet de les distinguer, mais cache toujours les points superposés. (c) Avec de la transparence, on peut distinguer les endroits où des points sont empilés. (d) Le déplacement des points ne résout que partiellement le problème d'occlusion et modifie l'information spatiale. Dans la visualisation originale, le Dr. Snow a opté pour un déplacement plus contrôlé. (e) En changeant l'encodage visuel pour représenter la densité des points, l'occlusion a disparu. Il est alors aisé de repérer que la source de l'épidémie doit se trouver proche de la zone la plus dense, au milieu.

Les techniques suivantes visent à modifier les données à afficher plutôt que l'aspect visuel. On peut recenser quatre catégories de techniques de ce type : le filtrage, l'échantillonnage, le binning et le clustering.

Le filtrage et l'échantillonnage sont deux techniques voisines qui consistent à retirer des données de l'ensemble à visualiser. Le filtrage applique un critère de sélection et ne retient que les données qui satisfont ce critère. Il est pertinent d'appliquer un filtrage si l'utilisateur contrôle ce critère, comme ce que proposent HomeFinder [114] et FilmFinder [2]. Dans Splatterplots [78], le scatterplot est filtré aux endroits où la densité est la plus élevée. Ces zones sont remplacées par des aplats de couleur, comme sur une carte de chaleur. Cette technique permet de drastiquement diminuer la quantité de données à visualiser aux endroits où l'occlusion est la plus forte. L'échantillonnage sélectionne lui aussi une partie des données, mais utilise un tirage aléatoire. Il n'y a donc plus de critère objectif. L'échantillonnage permet ainsi de réduire la quantité de données lorsqu'on ne peut pas appliquer de critère de sélection. Cette technique peut être utilisée pour visualiser des scatterplots [14, 28], des données géographiques [35] ou des graphes [92].

Le *binning* consiste à placer les points de donnée dans les cases d'une grille, puis à représenter une statistique (comptage, somme, moyenne...) pour chaque case. Le principe est souvent utilisé pour représenter une variable continue dans un histogramme en 1D en groupant les données par plage de valeur. Le concept est étendu en 2D pour réaliser une carte de chaleur sous forme de matrice [113] ou de carte géographique [48, 74]. La grille utilisée peut alors être rectangulaire ou hexagonale [26]. L'avantage du binning est que l'assignation d'un point à une case de la grille est très facile à calculer, ce qui rend la technique rapide à mettre en oeuvre. Mais comme toute technique de discrétisation, le résultat dépend fortement du choix de la taille et de l'emplacement de la grille. Le principe du binning est étendu par Wickham dans [112] par une étape de résumé et de lissage. En remarquant que le processus de dessin d'une image constituée de pixels est une forme de binning où chaque pixel constitue une case, Cottam et al. [34] proposent d'appliquer un post-traitement après le binning pour permettre de nouvelles visualisations. Ce principe rejoint les arguments avancés par Liu et al. dans [74] pour dire que la scalabilité de l'encodage doit être limitée par la résolution choisie et non la taille des données.

Enfin, les différentes techniques de clustering visent à créer des groupes de données. La visualisation peut alors représenter les groupes au lieu des données brutes, ce qui réduit le nombre de primitives visuelles et limite l'occlusion. Le clustering peut être sémantique ou géométrique. Le clustering sémantique crée les groupes en fonction de critères de similitude issus des données. Il existe de nombreux algorithmes pour choisir ces groupes, comme par exemple l'algorithme de Louvain [19]. Ce type de clustering est souvent utilisé en visualisation de graphe. Après avoir appliqué le clustering, on obtient un nouveau graphe où chaque cluster est représenté par un sommet. On appelle ce graphe *méta-graphe* et ses sommets *méta-noeuds* [9].

Le clustering géométrique, lui, opère un regroupement basé sur une notion de distance entre les éléments graphiques. Il s'applique sur la représentation géomé-

trique plutôt que dans l'espace des données. Le clustering géométrique permet de simplifier la représentation sans la changer drastiquement. L'objectif est que le résultat visuel soit très proche de ce que la visualisation aurait été sans appliquer de clustering.

3.1.1 Visualisation multi-échelle

Toutes les techniques présentées précédemment permettent de réduire l'occlusion pour obtenir une vue d'ensemble des données. Mais cette simplification enlève la possibilité d'accéder au détail. Pour retrouver la possibilité d'explorer les données, la visualisation multi-échelle agrège récursivement les données. Les données agrégées constituent alors plusieurs niveaux de détail. Les niveaux avec peu de détail permettent de disposer d'une vue d'ensemble alors que ceux avec plus de détail peuvent aller jusqu'à représenter les données brutes.

Les exemples les plus répandus de visualisation multi-échelle sont les systèmes de cartographie en ligne (Google Maps, Bing Maps, OpenStreetMap...). Ces systèmes permettent à l'utilisateur de démarrer sur une vue d'ensemble, puis de zoomer pour voir du détail sur une zone d'intérêt. Les données affichées changent alors suivant le niveau de détail correspondant. Le principe a d'ailleurs été repris par Nachmanson et al. [84] pour visualiser de grands graphes.

Certaines techniques considèrent que la décomposition hiérarchique est donnée en entrée. C'est le cas de la visualisation multi-niveau de Eades et Feng [38], où chaque niveau de détail est disposé sur un plan 2D pour une visualisation en 3D. C'est aussi le cas lorsque les données sont déjà organisées sous forme d'arbre. Les idiomes de visualisation d'arbres se prêtent alors très bien à la visualisation multi-échelle [17, 44, 99]. Cette décomposition hiérarchique peut aussi venir du caractère space-filling de la visualisation, comme c'est le cas pour les matrices d'adjacence [43].

Lorsque les niveaux de détail doivent être générés, toutes les techniques précédentes peuvent être utilisées. Dans le cadre de la visualisation multi-échelle, le clustering géométrique permet de donner l'impression que l'ensemble des données est représenté tout en préservant l'aspect de la visualisation à tous les niveaux de détail. Quigley et Eades [90] l'utilisent à la fois pour dessiner un graphe, puis en faire la visualisation à l'aide d'une décomposition spatiale en quad-tree. Van Ham et Van Wijk [108] représentent les clusters obtenus par des sphères. Le clustering géométrique est alors naturel lorsque les sphères s'intersectent.

La décomposition hiérarchique des données, qu'elle soit préexistante ou calculée pour la visualisation, peut être représentée sous forme d'arbre. Les noeuds de l'arbre situées à la même profondeur constituent un niveau de détail. La visualisation multi-échelle consiste à choisir les noeuds de l'arbre qui seront représentés. Une classification des différents parcours possibles est donnée par Elmqvist et Fekete dans [44]. On peut distinguer quatre types de parcours, représentés sur la Figure 3.2 :

— Le parcours par **niveau** consiste à sélectionner tous les noeuds d'un même niveau de détail.

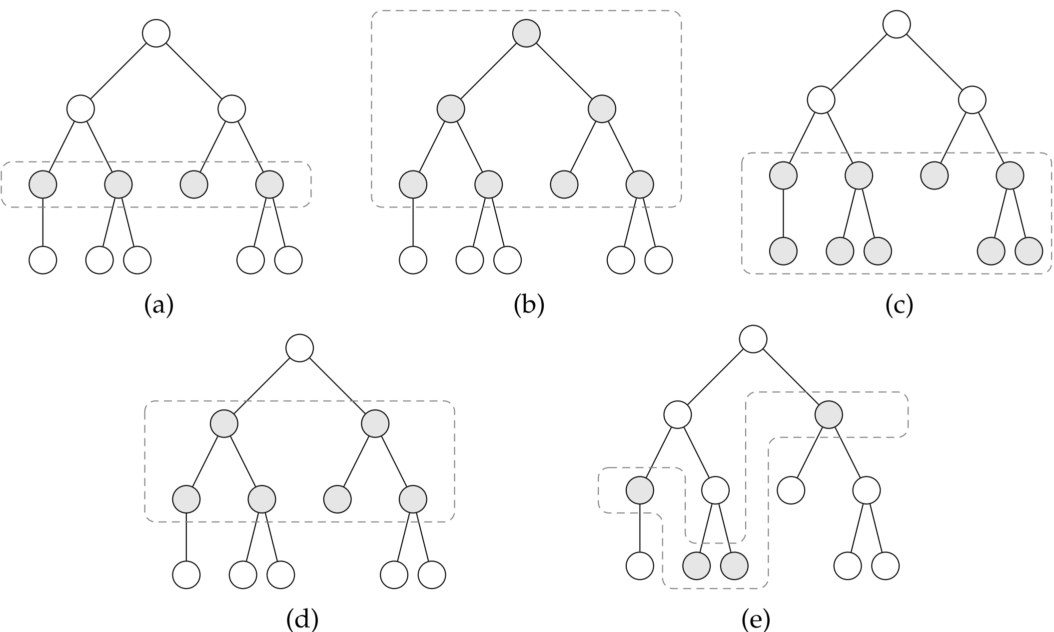


FIGURE 3.2 – Les différents parcours d'arbre de clustering possibles. Les noeuds grisés sont ceux sélectionnés pour être affichés dans la visualisation. (a) Parcours par niveau. (b) Parcours supérieur. (c) Parcours inférieur. (d) Parcours par intervalle. (e) Selection d'une antichaîne.

— Le parcours **supérieur** affiche tous les noeuds du niveau de détail sélectionné jusqu'à la racine de l'arbre.

— Le parcours **inférieur** affiche tous les noeuds du niveau sélectionné jusqu'aux feuilles de l'arbre.

— Le parcours par **intervalle** affiche tous les noeuds compris entre deux niveaux de détail sélectionnés.

Le parcours par niveau est la technique utilisée dans les systèmes de cartographie en ligne et de nombreux articles de la littérature [84, 90]. Le parcours supérieur est aussi utilisé dans un certain nombre de travaux, notamment les travaux de Archambault, Munzner et Auber [3, 4, 5] et de Sansen et al. [96]. Le parcours par intervalle permet de montrer un niveau de détail ainsi que des informations sur le(s) niveau(x) suivant(s) et/ou précédent(s). Ce parcours est parfaitement illustré par [9], où la visualisation montre à l'intérieur des méta-noeuds, le dessin du sous-graphe qu'ils représentent. Enfin, la visualisation de graphe sous forme de surfaces implicites proposée par Balzer et Deussen [10] permet de sélectionner à la fois le niveau maximum et le niveau minimum à visualiser, ce qui permet de reproduire tous les types de parcours.

La sélection des données à visualiser peut aussi être plus flexible en utilisant une antichaîne choisie dans l'arbre de clustering au lieu d'un niveau, comme illustré sur la Figure 3.2e. Une antichaîne est un ensemble de noeuds de l'arbre tel que lorsqu'un noeud fait partie de l'antichaîne, ni ses descendants ni ses ancêtres n'en font partie.

Autrement dit, l'antichaîne ne contient que des noeuds incomparables par la relation d'ordre ancêtre/descendant. Cela permet de contrôler plus localement le niveau de détail voulu. Le contrôle donné à l'utilisateur peut être direct sur l'antichaîne, comme dans [1] ou alors s'opérer à l'aide d'une interaction à la souris [51, 108].

La navigation dans les interfaces multi-échelles demande de supporter, en plus des opérations classiques de zoom & pan, les opérations qui permettent de changer de niveau de détail. Ces opérations sont appelées *drill-down* pour afficher un niveau plus détaillé et *roll-up* pour afficher un niveau moins détaillé. Ces opérations sont en principe indépendantes des opérations de zoom & pan. En effet, lorsque l'utilisateur a un contrôle direct sur ces opérations, comme dans [1], il est possible d'interagir sans changer de niveau de détail. En revanche, dans de nombreux cas, notamment après utilisation d'un clustering géométrique, il est plus pertinent de coupler l'interaction de zoom géométrique (agrandissement/rétrécissement visuel) et de *drill-down*/*roll-up* afin de garantir que le niveau de détail affiché est toujours celui correspondant au niveau de zoom actuel. C'est par exemple la technique employée dans Google Maps et [121]. On peut alors parler de zoom *sémantique*, puisque l'action du zoom change les données affichées.

3.2 Scalabilité de performance

L'autre problème de la visualisation de données massives est le problème de performance. La quantité de données influe sur les temps de calcul en amont de la visualisation et sur les temps de rendu.

Là encore, la visualisation multi-échelle permet de pallier le problème, notamment à l'étape du rendu. Le concept de budget d'entité [44] est fondamental à l'élaboration d'une visualisation multi-échelle dont les temps de rendu sont interactifs. Le système ASK-graphview [1] va même plus loin en proposant de traiter l'écran, la mémoire et le disque comme des ressources quantifiées. Le nombre d'éléments affichés à l'écran et gardés en mémoire tiennent alors compte de ces limites.

Le calcul des niveaux de détail peut parfois demander des ressources importantes. Par exemple, le précalcul des niveaux de détail de GraphMaps [84] prend 6 heures pour un graphe de 38K noeuds et 85K arêtes. Il est alors intéressant de chercher à accélérer ce précalcul.

L'utilisation d'un supercalculateur n'est accessible qu'à une petite minorité de chercheurs. En cela, le cloud offre une solution bien plus adaptée pour accélérer des calculs coûteux. Vo et al. montrent même que des algorithmes répandus en visualisation sont implémentables sur Hadoop en utilisant le paradigme MapReduce [110].

Le domaine de la visualisation géographique est fortement impacté par l'augmentation des quantités de donner à visualiser. Plusieurs frameworks ont été développés pour faciliter l'utilisation d'Hadoop dans ce domaine [39, 41], à la fois pour le traitement de données et pour la visualisation. Ces frameworks permettent la

mise en place de systèmes de visualisation basés sur Hadoop sans en maîtriser le fonctionnement [40].

Mais Hadoop peut aussi être utilisé directement pour permettre l'agrégation de données. GraphVizDB [16] d'une part et Jonker et al. [65] d'autre part utilisent tous deux Hadoop pour dessiner et calculer les niveaux de détail d'une visualisation de graphe grâçe à un algorithme de partitionnement. Un sytème similaire est utilisé par Sansen et al. [96] pour visualiser des diagrammes de Sankey multi-échelle.

Enfin, Hadoop est aussi utilisé pour calculer des abstractions multi-échelle dans le cadre de dessin de grands graphes [6, 7, 59].

Deuxième partie

Visualisation de données massives : Concepts et Techniques

4
Méthodologie générale

Dans ce chapitre, nous détaillons une méthodologie générale applicable à la visualisation de données massives. Cette méthode nous conduit à modifier le pipeline de visualisation pour le scinder en deux parties. La première comprend les étapes de précalcul sur le cloud et la seconde regroupe les étapes effectuées sur le client de visualisation.

4.1 Contraintes et besoins spécifiques d'un système de visualisation de données massives

La quantité d'information qui peut être représentée sur un écran d'ordinateur est intrinsèquement limitée par sa résolution (son nombre de pixels). Lorsque plusieurs informations occupent le même pixel, on parle d'occlusion ou "overplot". Il existe plusieurs solutions pour pallier ce problème. La première est d'augmenter le nombre de pixels de l'écran ainsi que sa taille. Cette solution a l'avantage de permettre une grande immersion dans la visualisation mais empêche de fait de prendre connaissance d'un seul coup de l'entièreté de l'information, puisque l'écran dépasse le champ visuel de l'utilisateur. Il est donc compliqué d'obtenir une vue d'ensemble des données. De plus, elle nécessite du matériel spécifique, qui ne peut généralement pas être déplacé. Comme le souligne Munzner dans "Visualization Analysis & Design" [83], une telle installation nécessite un local adapté, ce qui l'empêche d'être intégré à une méthode de travail quotidienne et demande un changement de contexte mental.

L'autre solution consiste à présenter à l'utilisateur une abstraction des données, de façon à réduire la quantité d'information à représenter. Pour permettre l'exploration interactive de données, il est alors nécessaire de prévoir plusieurs niveaux d'abstraction ou niveaux de détail. Chaque niveau supplémentaire donnant plus d'information, mais devant être plus fortement filtré pour être présenté à l'utilisateur. On parle alors de visualisation multi-échelle, comme présenté précédemment. Ce concept est fondamental pour permettre de visualiser des données massives au delà de ce que permet une résolution classique.

Nous avons vu en introduction que les besoins liés à la collecte, le stockage et le traitement de données massives avaient fait émerger de nouvelles infrastructures distribuées. Les données à visualiser ne peuvent que difficilement quitter ces infrastructures. D'une part, elles sont trop volumineuses pour être copiées : même avec l'augmentation des vitesses de transfert réseau, le temps de copie d'une telle quantité de données reste prohibitif sans pour autant apporter d'avantage décisif lors du traitement ultérieur. D'autre part, les données brutes sont devenues une ressource précieuse. Les détenteurs de cette nouvelle ressource préfèrent donc la conserver au sein d'une infrastructure contrôlée et sécurisée.

L'autre problème que pose la visualisation de données massives est celui des performances de calcul. Aucun calcul sur les données brutes ne peut être effectué en temps réel. Il est donc impératif de procéder à un précalcul et une indexation des données pour accélérer les étapes ultérieures. En tenant compte des remarques précédentes, il faut aussi calculer une abstraction multi-échelle, ce qui implique plusieurs étapes de calcul sur l'ensemble des données. Dans ces conditions, l'utilisation des capacités de calcul et de stockage distribué offertes par l'infrastructure Big Data s'avère indispensable, indépendamment de la localisation initiale des données.

La visualisation au sein d'une infrastructure Big Data implique une asymétrie des moyens. En effet, les données sont stockées au sein d'une grappe de machines dont les ressources en puissance de calcul et mémoire vive ne sont limitées que par l'investissement qui est fait. Au contraire, la visualisation s'effectue sur une machine comparativement très limitée en ressources. Il faut alors trouver le moyen de diminuer la quantité d'information à transférer et représenter. Là encore, l'utilisation d'une abstraction multi-échelle permet de ne transférer qu'une partie des données. Dans l'idéal, la quantité de données à transférer doit être très limitée, voire bornée. Cette contrainte rejoint la problématique d'occlusion. Cependant, elle est moins forte, puisqu'il est possible de commencer par transférer très peu de données et de rajouter du détail ensuite sans bloquer le processus de visualisation.

Toutes ces conditions à la mise en place d'un système de visualisation de données massives peuvent sembler contraignantes, puisqu'elle sortent du cadre classique de visualisation de données sur un client unique. Il n'est pas non plus direct d'adapter des méthodes connues pour les intégrer dans ce contexte. Cependant, un tel système basé sur une visualisation déportée représente aussi une chance et offre de nombreuses opportunités. On peut par exemple retenir celles décrites par Holliman & Watson [60] : le partage des ressources de calcul, la simplification du code du client, ce qui facilite son portage sur de nouvelles plateformes et bien sûr la scalabilité offerte par le cloud. A ces avantages, nous ajouterons le partage des précalculs et résultats. Tous les calculs effectués sur le cloud peuvent être conservés et partagés entre les utilisateurs. Ainsi, même si le précalcul d'une abstraction multi-échelle d'un grand jeu de données requiert plusieurs heures de calcul, une fois celui-ci effectué, plusieurs utilisateurs peuvent en bénéficier de manière transparente, ce qui permet d'amortir le coût en temps de calcul par utilisateur.

4.2 Modification du pipeline de visualisation pour les données massives

Le déséquilibre de ressources entre l'infrastructure Big Data et le client de visualisation entraîne des répercussions sur le pipeline de visualisation. Les étapes les plus coûteuses en temps et/ou en mémoire doivent s'effectuer dans le cloud, tandis que les autres peuvent être exécutées sur le client. Le pipeline se retrouve ainsi séparé en deux : d'abord la partie cloud et ensuite la partie client. Suivant où se fait la séparation, plusieurs types de pipeline pour les données massives émergent. Tout d'abord, les approches orientées pixel et géométrie, proposées par Holliman et Watson [60]. Nous proposons aussi notre approche basée sur l'utilisation d'une abstraction multi-échelle, qui permet de résoudre certains des problèmes liés aux approches précédentes.

4.2.1 Pipeline orienté pixels

L'approche orientée pixels est la plus simple possible. Dans ce découpage, tout le pipeline est exécuté dans le cloud. L'image finale est ensuite envoyée au client pour l'affichage. Avec cette approche, le rendu est effectué dans le cloud, il n'y a donc aucune contrainte de performance pour le client. Mais cela implique aussi que toutes les interactions doivent être transmises au serveur pour être traitées et générer une nouvelle image qui sera transmise au client. Toutes les interactions nécessitent donc un aller-retour à travers le réseau, ce qui augmente leur latence. De même, la bande passante utilisée est proportionnelle à la résolution de la visualisation. Les images envoyées peuvent être compressées, mais cela augmente encore la latence.

La simplicité de cette approche permet de la rendre compatible avec n'importe quelle visualisation, mais n'apporte au final rien de plus par rapport à un logiciel de partage de bureau à distance, puisque le rôle du client est réduit à un écran sur lequel afficher l'image finale qui lui est transmise.

4.2.2 Pipeline orienté géométrie

L'approche orientée géométrie opère la séparation du pipeline plus en amont. La partie cloud est responsable de la génération de la géométrie, mais plus du rendu. Cette étape est assurée par le client. Le client doit donc posséder une puissance suffisante pour dessiner la géométrie qui lui est transmise. Cette répartition des rôles rend le client un peu moins dépendant de la partie cloud. Certaines interactions peuvent être réalisées sans l'aide du serveur, ce qui améliore la latence par rapport à l'approche précédente. Cependant, les interactions les plus complexes, qui requièrent une manipulation des données, ont toujours besoin d'avoir recours au serveur.

La bande passante qu'utilise cette approche est difficile à estimer, puisque la quantité de données géométriques à envoyer dépend entièrement de l'application.

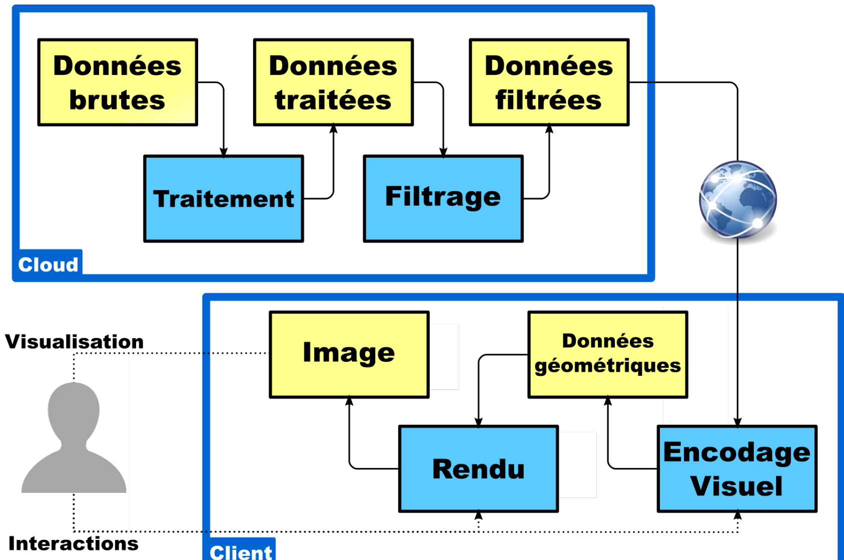


FIGURE 4.1 – Le pipeline de visualisation, modifié pour prendre en compte les contraintes à visualiser des données massives. Les étapes du pipeline sont maintenant séparées en deux phases : la phase de précalcul s'effectue dans le cloud alors que la phase de visualisation s'effectue sur une machine locale et requiert le transfert d'une partie des données précalculées.

Cette approche peut être comparée à des bibliothèques de rendu graphique orientées client-serveur, comme X11 et OpenGL.

4.2.3 Pipeline orienté abstraction de données

Les deux approches précédentes adaptent le modèle existant pour déporter certaines tâches coûteuses dans le cloud, mais elles ne prennent pas en compte un des problèmes fondamentaux liés à la visualisation de données massives : l'occlusion.

Notre approche, visible en Figure 4.1, sépare le pipeline au niveau de l'encodage visuel, donc plus en amont que l'approche orientée géométrie. Les données envoyées au client sont représentées sous forme d'abstraction multi-échelle, ce qui permet à la fois de résoudre le problème d'occlusion et de mieux prévoir, voire de borner la bande passante utilisée. Avec cette approche, le client devient beaucoup plus indépendant du serveur. La plupart des interactions peuvent être implémentées sans transfert réseau. Puisque le client dispose d'une abstraction des données, il est possible d'appliquer des algorithmes de visualisation classiques sur le client.

Grâce à l'utilisation de l'abstraction multi-échelle, le client n'a pas besoin de disposer d'une grande capacité de calcul. Pour s'adapter aux appareils les moins puissants, le niveau d'abstraction à utiliser peut être augmenté, c'est-à-dire que moins de données seront transférées.

Cette approche est la plus complète et la plus flexible. C'est donc celle ci que

nous utiliserons dans le cadre de cette thèse.

4.3 Réalisation du pipeline

Dans cette section, nous détaillons comment mettre en place les différentes étapes du pipeline orienté abstraction de données.

4.3.1 Agrégation de données

La première étape est le calcul d'une abstraction multi-échelle des données à visualiser. Ce calcul va permettre de réduire la quantité de données que chacune des étapes suivantes devra gérer. Il règle à la fois les problèmes d'occlusion et de performance dans la suite du processus.

L'abstraction multi-échelle à utiliser pourra varier suivant la visualisation voulue, mais il est possible de déterminer certaines caractéristiques désirables :

— **Scalabilité visuelle** : L'abstraction utilisée doit pouvoir être appliquée à une quantité de données arbitrairement grande.
— **Scalabilité computationnelle** : Les calculs à effectuer pour produire l'abstraction doivent satisfaire le principe de scalabilité horizontale, de façon à permettre de générer l'abstraction pour de grands jeux de données. La distribution et parallélisation des calculs est impérative pour atteindre cet objectif.
— **Scalabilité de stockage** : La taille du résultat de l'abstraction ne doit pas grandir démesurément plus vite que la taille du jeu de donnée. La quantité de données à stocker devrait grandir au même rythme que les calculs pour la générer.

Comme les données sont beaucoup trop volumineuses pour que l'abstraction soit entièrement calculée en temps réel, elle devra donc être précalculée. Il est possible d'envisager de ne précalculer qu'une partie de l'abstraction puis de calculer le reste à la demande. Les parties précalculées doivent alors être choisies pour permettre de calculer le reste en temps réel.

4.3.2 Indexation & Requêtage

Pour permettre au client de récupérer seulement la partie visible des données, il est nécessaire d'indexer le résultat de l'agrégation. Cette indexation doit permettre de requêter n'importe quelle partie des données en temps constant et le plus court possible. De cette façon, le délai entre la requête du client et l'envoi des données est réduit au minimum.

La méthode d'indexation la plus utilisée est la pyramide de tiles. Elle est utilisée notamment par les services de cartographie en ligne, tels Google Maps et par

le système GraphMaps [84]. La pyramide comporte plusieurs niveaux, chacun composé de tiles. Les niveaux sont numérotés de haut en bas, du plus agrégé au moins agrégé. Le niveau 0 se trouve en haut de la pyramide et est constitué d'un unique tile. Pour constituer le niveau suivant, chaque tile est coupé en 4. Le nombre de tiles du niveau i est donc 4^i. Un tile peut être identifié de manière unique par un triplet composé de son niveau et de ses coordonnées en abscisse et en ordonnée au sein du niveau. Les coordonnées au sein du niveau i varient de 0 à $2^i - 1$. L'origine peut indifféremment être placée dans n'importe quel coin de la pyramide. Il est alors possible d'accéder aux données contenues dans le tile en utilisant son identifiant.

Un tile représente une surface constante à l'écran, généralement 256×256 pixels. La navigation dans la pyramide correspond donc à la fois à un zoom sémantique (changement de niveau de détail) et géométrique (changement de taille des entités).

Pour être récupérés rapidement, les tiles sont stockés dans une base de données distribuée. Elle doit permettre d'équilibrer à la fois le stockage et la charge de requêtes entre plusieurs machines et d'obtenir un tile en temps constant.

4.3.3 Rendu

La dernière étape consiste à requêter la partie visible de l'abstraction multi-échelle sous forme de tile et rendre cette partie de la visualisation. Ici, seule une partie des données est disponible. Les données à visualiser changent à chaque déplacement de l'utilisateur. Un cache local permet de ne pas requêter sans arrêt les mêmes tiles. De plus, la bibliothèque utilisée pour réaliser la visualisation doit être capable de gérer tous les changements entraînés par le chargement des données.

4.4 Conclusion

Le pipeline de visualisation par abstraction de données présenté dans ce chapitre sert de base à la méthode de visualisation de données massives que nous développons dans cette thèse. Elle repose sur le calcul d'une agrégation multi-échelle, détaillé au chapitre 5. La navigation dans cette abstraction multi-échelle des données s'effectue sur un client léger. Nous détaillerons au chapitre 6 comment nous avons traité cette étape. Enfin, la méthode complète sera mise en pratique au chapitre 7 pour visualiser des cartes de chaleur et au chapitre 8 pour visualiser des grands graphes.

5

Agrégation géométrique pour la visualisation

Dans ce chapitre nous présentons notre approche de clustering géométrique pour la visualisation de données massives. Cette approche se base sur l'algorithme de canopy clustering, publié par McCallum, Nigam et Ungar en 2000 [79].

Nous présentons ici son application comme un algorithme de clustering géométrique, c'est-à-dire qui utilise le plongement 2D des données pour décider des regroupements. Nous montrons ensuite comment l'utiliser pour réaliser une abstraction multi-échelle des données. L'algorithme original étant purement séquentiel, nous présentons aussi deux approches qui permettent de le rendre parallèle et distribué.

5.1 Canopy Clustering

L'algorithme de canopy clustering permet de choisir des représentants, appelés canopies, dans un ensemble de points. Deux distances d_1 et d_2 sont utilisées pour les définir, avec $d_1 \leq d_2$. Les règles suivantes sont appliquées :

- Si un point n'a pas de représentant plus proche que d_1, il peut devenir son propre représentant.
- Si un point est à une distance inférieure à d_2 d'un représentant, il fait partie de son groupe.

Chaque représentant définit un groupe, appelé canopy, pour les points qui sont à une distance inférieure à d_2 de lui. La forme du groupe dépend de la définition de distance utilisée. Elle sera par exemple circulaire pour une distance euclidienne. On dit que ces points sont couverts par leur représentant, d'où le nom de canopy en référence à la canopée qui couvre le reste de la forêt. On peut remarquer que comme $d_1 \leq d_2$, une canopy peut contenir le représentant d'une autre canopy. De plus, un point peut appartenir à plusieurs groupes.

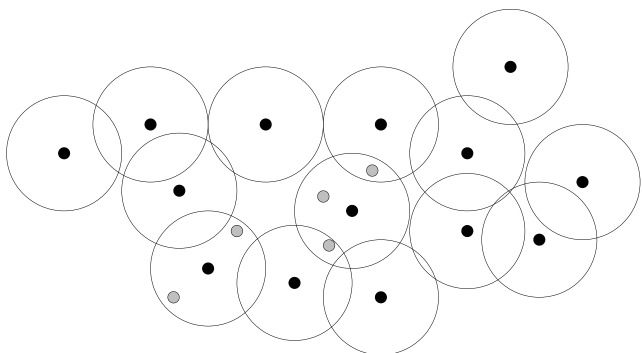


FIGURE 5.1 – Exemple de canopy clustering où $d_1 = d_2$. Les points noirs ont été choisis comme représentants. Les cercles représentent l'étendue de la zone couverte par chaque représentant.

Cet algorithme a d'abord été pensé comme une étape simple de prétraitement permettant de réduire le coût computationel d'algorithmes de clustering plus complexes. Cette amélioration s'opère en n'effectuant des calculs de distance qu'entre les points faisant partie des mêmes canopies et en considérant que les autres points seront de toute façon trop éloignés pour qu'il soit intéressant de les prendre en compte. Ainsi, il est souvent utilisé en amont de l'algorithme des k-moyennes [56], d'une part pour réduire le nombre de calculs de distance à effectuer et d'autre part pour permettre d'estimer le paramètre k en utilisant comme valeur le nombre de représentants choisis par l'algorithme de canopy clustering.

5.1.1 Algorithme pour l'agrégation géométrique

Pour utiliser l'algorithme de canopy clustering pour de l'agrégation géométrique, nous allons utiliser le cas particulier $d_1 = d_2$. Dans ce cas, l'algorithme n'utilise plus qu'une seule distance d'agrégation que nous nommerons d.

Algorithme 5.1 : Canopy clustering séquentiel.

Data : Un ensemble de points S, une distance d
Result : The set of canopies

```
1 canopies = ∅;
2 for every point p in S do
3 |   for every canopy c in canopies do
4 |   |   if distance(p,c) < d then
5 |   |   |   go to next point
6 |   add p to canopies
7 return canopies
```

On a alors les garanties suivantes :

— Deux représentants sont séparés d'au moins d.

— Un point qui n'est pas représentant est au plus à distance d d'un représentant.

Là encore, un point peut appartenir à plusieurs groupes, mais ce nombre est borné par le "*kissing number*" [32], c'est-à-dire le nombre maximal de cercles identiques qui peuvent être tangents à un même cercle sans intersection. En deux dimensions, un point appartiendra donc au maximum à 6 groupes.

5.1.2 Borne sur le nombre de représentants

Une première intuition laisse penser que dans le pire des cas, tous les points peuvent devenir représentant. Mais il est possible de borner ce nombre par une constante. L'aire totale (A) de la boite englobante du jeu de données est manifestement finie et facile à calculer. L'algorithme nous garantit que les représentants sont tous séparés par au moins d. Le nombre maximal de points que l'on peut répartir au sein de la boite englobante en respectant cette contrainte ne dépend pas du nombre total de points, seulement de d et de l'aire totale.

En utilisant le problème de packing de cercles, on peut trouver une relation entre la distance d'agrégation choisie et le nombre maximal de représentants possible. En effet, des points tous séparés d'au moins d sont équivalents à des cercles de diamètre d sans intersection. Le pavage avec la plus grande densité de cercles est le pavage hexagonal [45, 102], dont la densité est de $\frac{\pi}{2\sqrt{3}}$. Ce nombre mesure le ratio entre l'aire totale et l'aire occupée par les cercles. On peut donc poser la relation suivante, qui correspond à l'aire occupée par c cercles :

$$A\frac{\pi}{2\sqrt{3}} = c\pi\left(\frac{d}{2}\right)^2$$

On peut en déduire les deux relations suivantes entre la distance d'agrégation d et le nombre maximal de cercles c, donc de représentants, produits :

$$c = \frac{2A}{d^2\sqrt{3}} \qquad d = \sqrt{\frac{2A}{c\sqrt{3}}}$$

En utilisant ces relations, il est possible de décider d'un nombre maximal de représentants pour paramétrer l'algorithme au lieu d'une distance d'agrégation. En effet, la force de l'agrégation pour une valeur de d dépend de l'aire totale occupée par le jeu de données. Une même valeur de d donnera donc des données plus ou moins abstraites en fonction du jeu de données. Au contraire, pour une même valeur de c, il est possible de calculer la valeur de d correspondante pour chaque jeu de données. L'utilisation d'un nombre maximal de représentants c permet donc de contrôler la force de l'agrégation produite indépendamment du jeu de données.

5.1.3 Complexité

La complexité de cet algorithme dépend beaucoup de la structure de données utilisée pour stocker les représentants et les retrouver pour faire la comparaison avec

le point courant et déterminer s'il peut être ou non choisi comme représentant. Si on appelle n le nombre total de points et $\mathscr{C}$ la complexité de la phase de comparaison, la complexité totale est de $O(n\mathscr{C})$. La valeur de $\mathscr{C}$ est directement déterminée par les opérations à effectuer pour la comparaison. Dans le pire des cas, il faudra comparer le point courant à tous les représentants. Ce nombre est borné comme nous venons de le démontrer. Il est donc possible de considérer $\mathscr{C}$ comme constant, ce qui donne une complexité dans le pire des cas en $O(n)$.

Cependant, l'analyse précédente ne peut donner une idée du temps de calcul de l'algorithme que si $c \ll n$. Dans ce cas, il y a de fortes chances que le nombre réel de représentants approche fortement le nombre maximal. Dans le cas où $c \approx n$ ou $c > n$, le nombre maximal de représentants ne sera probablement pas atteint. $\mathscr{C}$ ne peut alors plus être considéré comme constant et devient dépendant de l'implémentation utilisée. Sans autre hypothèse, la meilleure possibilité est d'utiliser une structure d'indexation spatiale, comme un QuadTree [47], pour obtenir en temps logarithmique les représentants suffisamment proches du point courant. La complexité finale est alors $O(n * \log(n))$.

5.2 Utilisation multi-échelle du canopy clustering

5.2.1 Distance d'agrégation par niveau

L'algorithme de canopy clustering peut être paramétré en utilisant soit une distance d'agrégation, soit un nombre maximal de représentants voulu. Quelle que soit la solution retenue, cette valeur contrôle directement l'intensité de l'agrégation. Pour réaliser plusieurs niveaux de détail, il suffit donc de changer la distance d'agrégation utilisée.

Une abstraction multi-échelle avec le canopy clustering peut être définie par une distance d'agrégation de base d_0, puis un ratio r à appliquer à chaque niveau. Ainsi, le premier niveau utilisera la distance d_0, le niveau inférieur $\frac{d_0}{r}$ et ainsi de suite. La distance utilisée par le niveau i est donc définie par $\frac{d_0}{r^i}$.

Le calcul de l'agrégation de chaque niveau peut être fait indépendamment, mais cette méthode présente deux inconvénients :

- Chaque niveau devra traiter la totalité des données
- Aucun lien ne pourra être fait entre les représentants des différents niveaux

Il est plus intéressant de calculer l'agrégation en commençant par le niveau de détail le plus fort (celui avec l'agrégation la moins forte) et d'utiliser le résultat du niveau $i + 1$ pour calculer celui du niveau i. De cette manière, chaque niveau reçoit en entrée non pas la totalité du jeu de données, mais seulement les représentants du niveau inférieur, dont le nombre est borné. Ainsi, chaque niveau supplémentaire demandera de moins en moins de calcul. De plus, les représentants du niveau i seront choisis parmi ceux du niveau $i + 1$. Il existe donc une relation hiérarchique

entre les représentants. Un représentant au niveau i est aussi forcément présent dans tous les niveaux inférieurs. Passer d'un niveau à l'autre revient à supprimer ou ajouter des représentants.

Après une application du canopy clustering, un point du jeu de données initial correspond à un représentant qui se trouve au plus à une distance d. Lorsque l'algorithme est utilisé récursivement, un point peut "sauter" de représentant en représentant et son représentant au dernier niveau d'agrégation peut être plus éloigné que d. La distance totale maximale entre un point et son représentant final est la somme des distances maximale entre les représentants successifs :

$$\sum_{i=0}^{max} \frac{d_0}{r^i} = d_0 \sum_{i=0}^{max} \frac{1}{r^i}$$

Lorsque le nombre de niveaux augmente, cette somme converge vers $\frac{r}{r-1}$ en tant que série géométrique de raison $\frac{1}{r}$. La distance entre un point du jeu de données de départ et son représentant final est donc d à un facteur constant près.

5.2.2 Nombre total de niveaux de détail

Il existe une distance suffisamment petite pour qu'aucune agrégation n'ait lieu. Cette distance correspond à la distance entre les deux points les plus proches dans le jeu de données. Il n'est donc pas nécessaire de calculer les niveaux de détail qui utilisent une distance inférieure ou égale.

Pour pouvoir calculer l'agrégation en partant du bas vers le haut, il faut alors connaître directement le nombre de niveaux à calculer et la distance la plus petite. Si on considère que l'on fixe la distance d'agrégation maximale d_0 et que d_{min} correspond à la distance entre les deux points les plus proches, le numéro du niveau le plus bas à calculer est donné par :

$$i_{max} = \left\lfloor \log_r \left(\frac{d_0}{d_{min}} \right) \right\rfloor$$

Cette formule implique qu'il n'est pas nécessaire de connaître la distance exacte pour calculer le bon nombre de niveaux de détail. Il suffit d'approximer cette distance à un facteur r prêt.

Le problème du calcul exact de cette distance, appelé "problème de la paire la plus proche", a fait l'objet de nombreuses recherches. Le premier algorithme optimal en $O(n \log(n))$ est donné par Shamos et Bentley [11] et repris par Cormen, Leiserson, Rivest et Stein [33] dans leur "Introduction à l'Algorithmique". Smid [101] a réalisé une revue des algorithmes existants, pour les versions statique, en ligne et dynamique du problème. Plus récemment, Har-Peld [55] montre qu'il est possible de résoudre le problème avec une complexité en moyenne de $O(n)$. Mais tous ces algorithmes sont séquentiels et aucun n'est facilement distribuable.

Une solution très efficace reste l'échantillonnage du jeu de données initial. L'idée est à la base de l'algorithme randomisé de Rabin [91], qui calcule à partir d'un échantillon la taille d'une grille permettant avec une grande probabilité de résoudre le problème en temps linéaire. En pratique, calculer la distance minimale d'un échantillon est suffisant pour notre utilisation. Cette méthode est d'autant plus efficace qu'il est possible de répéter l'opération avec plusieurs échantillons pour ne garder que le minimum des distances minimales trouvées. Cette méthode n'offre bien sûr aucune garantie, mais produit un résultat satisfaisant pour les besoins de cette thèse et l'échantillonnage est une opération facilement distribuable.

5.2.3 Complexité

L'utilisation récursive du canopy clustering apporte une hypothèse supplémentaire pour le calcul de complexité d'un niveau i : tous les points en entrée sont séparés d'au moins $d_{i+1} = \frac{d_i}{r}$. D'après le lemme 2.1 présenté par Smid [101], chaque case d'une grille de taille d_i contient au plus $(2r + r)^2 = 9r^2$ points. En utilisant une telle grille, la phase de comparaison requiert de vérifier la distance entre le point courant et les points des 9 cases adjacentes (en incluant la case du point courant). Comme ces cases contiennent toutes un nombre borné de points, $\mathcal{C}$ est une constante et la complexité du calcul d'un niveau est $O(n)$.

Cette propriété est vérifiée à tous les niveaux d'agrégation, y compris le plus bas si on a correctement calculé la distance minimale entre deux points du jeu de données, à un facteur constant près.

5.3 Canopy clustering distribué : approche par partitionnement spatial

Dans cette section, nous présentons une première approche pour distribuer le calcul du canopy clustering. Cette approche utilise un principe très répandu en calcul parallèle et distribué : le partitionnement de l'espace. Un algorithme est exécuté indépendamment dans chaque partition, puis les résultats sont harmonisés à l'interface de chaque partition. Nous utiliserons le paradigme MapReduce pour distribuer le calcul. Notre approche permet de réaliser le clustering en deux *jobs*.

5.3.1 Algorithme

Le principe général de l'algorithme est de séparer l'espace en bandes. Un premier calcul est effectué dans chaque bande. Le calcul au sein de chaque bande est séquentiel, mais la séparation permet de les effectuer en parallèle. Ensuite, le partitionnement est modifié de manière à pouvoir corriger les incohérences produites par le premier calcul. Cette étape est aussi réalisée en parallèle. A la fin de ces deux étapes, on obtient un canopy clustering sur l'ensemble des données.

5.3.1.1 Partitionnement de l'espace

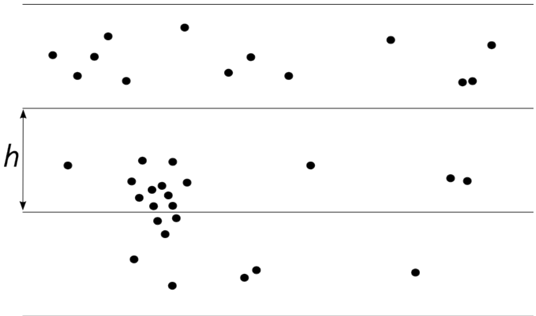


FIGURE 5.2 – Illustration du partitionnement de l'espace en bandes de hauteur égale.

Le partitionnement de l'espace se fait en plusieurs bandes parallèles. L'approche est valable aussi bien pour des bandes horizontales que verticales, mais nous considérerons ici le cas de bandes horizontales. Chaque bande a une hauteur h. La bande à laquelle appartient un point $p = (x, y)$ peut être trouvée en temps constant par $\left\lfloor \frac{y}{h} \right\rfloor$.

5.3.1.2 Premier *job* : clustering local

Algorithme 5.2 : Premier job Map Reduce pour le canopy clustering distribué par partitionnement de l'espace.

Input : Un ensemble de points
Output : Chaque point marqué comme COVERED ou CANOPY

```
1 Map(_, point)
2 |  emit(⌊point.y/h⌋,point)
3 Reduce(strip, points)
4 |  for point p in points do
5 |  |  if canopies.isCovered(p) then
6 |  |  |  emit(COVERED, p)
7 |  |  else
8 |  |  |  canopies.add(p)
9 |  |  |  emit(CANOPY, p)
```

Le premier *job* effectue un canopy clustering localement au sein de chaque bande. La phase de map est utilisée pour assigner chaque point à la bande correspondante. Les données émises sont des couples $(cle, valeur)$ où la valeur est un point et la clé est le numéro de la bande correspondante. Cette bande est calculée en divisant l'ordonnée du point par la hauteur de la bande.

La phase de *shuffle* de MapReduce permet ensuite de regrouper tous les points appartenant à la même bande. Le reduce n'a alors plus qu'à appliquer localement un canopy clustering. On obtient alors des points qui ont étés élus représentants et d'autres non. Mais le résultat obtenu ne correspond pas encore à un choix définitif. On ne peut donc pas éliminer de point à ce stade de l'algorithme. Les données émises par la phase de reduce sont les points assortis de leur état (représentants ou non).

5.3.1.3 Second *job* : résolution des conflits

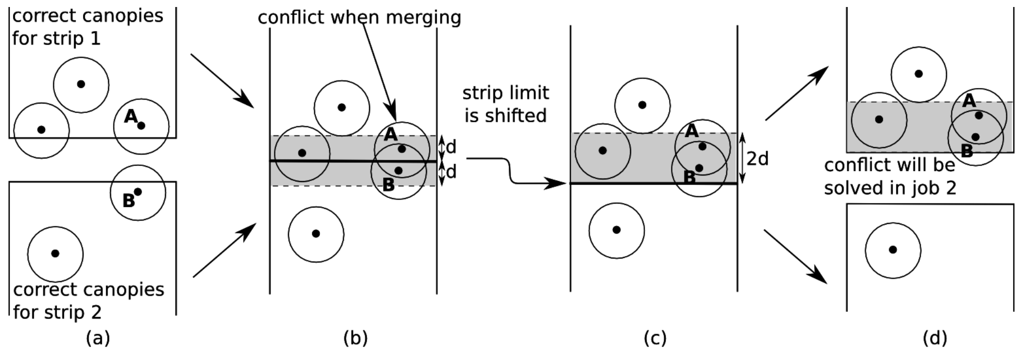


FIGURE 5.3 – Principe de la résolution de conflit. Après le premier job, il peut exister des paires de représentants plus proches que la distance d'agrégation. La limite des bandes est alors déplacée vers le bas pour englober toute la zone de conflit. Durant le second job Map Reduce, le conflit peut être détecté et résolu en supprimant le représentant de la partie supérieure.

A la fin du premier *job*, des représentants ont été choisis indépendamment au sein de chaque bande. Tous les points sont couverts par au moins un représentant, mais il peut exister des représentants plus proches que la distance d'agrégation voulue. On appelle ce cas un conflit. L'objectif du second *job* est de résoudre ces conflits pour respecter les deux contraintes du canopy clustering.

Un conflit implique deux représentants séparés par une distance inférieure à d. Un représentant peut être impliqué dans plusieurs conflits à la fois. Après le premier job, ils ne peuvent pas être situés dans la même bande. Ils sont donc situés de part et d'autre de la limite d'une bande, à distance inférieure à d du bord, comme le montre la Figure 5.3. Nous appelons cet espace de $2d$ de hauteur "zone de conflit".

Pour que le conflit puisse être résolu, les deux représentants doivent être regroupés au sein de la même partition. Ce second job conserve le partitionnement par bandes, mais déplace la limite de chaque bande de d vers le bas. Les deux parties de la zone de conflit sont donc maintenant contenues dans la même bande. Nous les appellerons "zone supérieure" et "zone inférieure". Un conflit concerne un représentant dans chaque zone. Le calcul effectué dans le premier job nous assure que :

— Les points de la zone supérieure sont tous couverts par au moins un représentant de la même zone.
— Les points de la zone inférieure sont couverts par au moins un représentant situé soit dans la zone inférieure, soit dans la bande du dessous.
— Les représentants de la zone inférieure peuvent couvrir des points situés dans la zone inférieure ou dans la bande du dessous.

La résolution du conflit passera inévitablement par la rétrogradation d'un des deux représentants. Ce faisant, certains points qu'il couvrait peuvent se retrouver sans représentant suffisamment proche s'ils n'étaient pas couverts par un autre représentant. Il faut alors procéder à une nouvelle élection de représentant pour les couvrir. Il y a alors deux possibilités :

— Supprimer le représentant de la zone supérieure. Les points potentiellement sans représentant sont alors aussi situés dans cette zone et l'élection peut se faire localement.
— Supprimer le représentant de la zone inférieure. Des points de la bande du dessous peuvent se retrouver sans représentant. Mais cette information ne peut être connue localement, la vérification devrait donc passer par un nouveau redécoupage.

C'est donc la première solution qui doit être appliquée dans tous les cas, puisqu'elle permet de terminer directement le clustering. Pour chaque conflit, le représentant situé zone inférieure sera conservé et celui de la zone supérieure supprimé.

5.3.2 Taille des partitions

La taille de chaque bande détermine le nombre total de bandes nécessaire pour partitionner l'espace des données. Ce nombre permet ensuite de paralléliser les calculs. Il semble donc judicieux de rendre chaque bande aussi petite que possible pour en maximiser le nombre total. Mais si les bandes deviennent trop petites, les calculs et décisions d'une bande peuvent avoir une influence sur ceux d'une bande voisine. Cet effet n'est pas présent dans le premier job, puisqu'on ne cherche alors pas à avoir un calcul global. En revanche, il est primordial de l'éviter lors du second job, dont l'objectif est de conclure le calcul. Nous déterminons ici la taille minimale que peuvent avoir les bandes en évitant les effets indésirables.

L'effet que l'on cherche à éviter est l'apparition d'un nouveau conflit après l'élimination du représentant de la partie supérieure de la zone de conflit au cours du job 2. Autrement dit, la taille de la bande doit être suffisante pour pouvoir déterminer si un point laissé découvert peut être choisi comme nouveau représentant.

La configuration de points qui nécessite la taille de bande la plus grande est un alignement vertical (perpendiculairement au sens de la bande), nous ne nous occuperons donc pas de la position horizontale des points. Deux points en particulier nous intéressent ici : le représentant supprimé, que nous appellerons r, un point qu'il couvrait et qui se retrouve non couvert du fait de la suppression, que nous

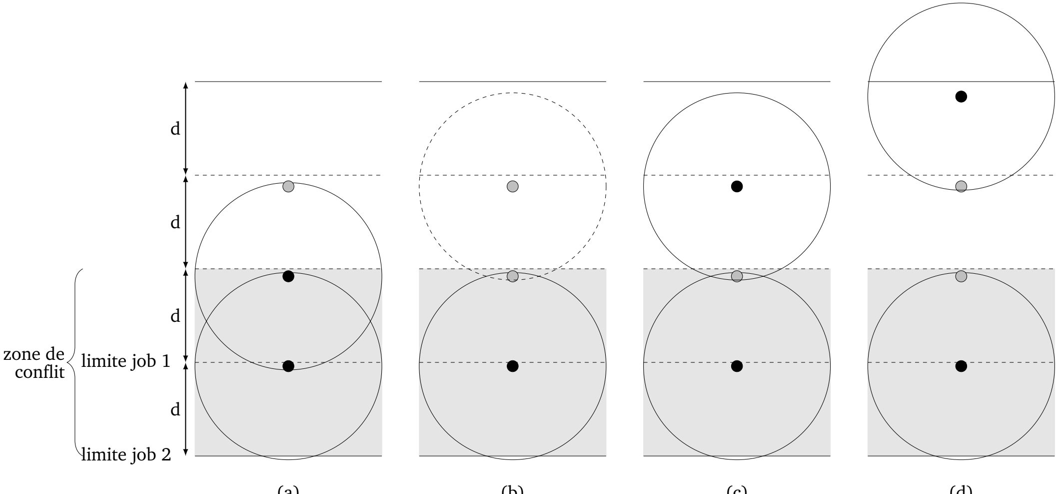


FIGURE 5.4 – Exemple de taille de bande minimale. (a) Un conflit entre deux représentants. Celui de la partie supérieure perd son statut. (b) Pour chaque point que ce représentant couvrait, on doit vérifier dans la zone en pointillés l'existence d'un autre représentant. Deux cas de figure sont possibles. (c) Il n'y en existe pas, le point peut donc devenir représentant. (d) Un autre représentant le couvre. Ce représentant est au maximum à distance $4d$ du bord de la bande, ce qui nous donne sa taille minimale.

appellerons p. Pour exprimer la position verticale de ces deux points, nous prendrons comme origine le bord inférieur de la bande. On peut donc poser que $d < y_r < 2d$, puisque r se trouve dans la partie supérieure de la zone de conflit. Le point p se trouve au plus à une distance d de r, mais ne peut se trouver dans la partie inférieure de la zone de conflit, comme nous l'avons déjà vu. La position verticale de p est donc $d < y_p < 3d$. Si un autre représentant c couvre p, sa position verticale est $0 < y_c < 4d$. La bande doit englober tous ces points pour pouvoir effectuer le calcul, la hauteur minimale d'une bande est donc $4d$. La situation est représentée sur la Figure 5.4.

5.3.3 Équilibrage de charge

Un point primordial pour assurer les performances d'un calcul distribué est de réussir à répartir le plus uniformément possible les calculs entre les différents noeuds. Un mauvais équilibrage de charge ralentit l'ensemble du calcul puisque les noeuds les moins chargés doivent attendre ceux qui sont les plus chargés.

Ici, l'équilibrage de charge consiste à décider de la répartition des bandes sur les k noeuds de calcul. Nous détaillons deux techniques : le round-robin et le partitionnement par intervalle.

Round-robin Le round-robin consiste à assigner alternativement une bande à chaque noeud. Le noeud de calcul par lequel une bande i doit être traitée est donc

donné par $i \bmod k$. La répartition de la charge sera équilibrée si le nombre de bande est suffisamment grand pour que les disparités de densité se compensent. Néanmoins, cette technique peut entraîner un déséquilibre dans des cas particuliers où les points se concentrent dans un petit nombre de bandes.

Partitionnement par intervalle Pour assurer une bonne répartition dans tous les cas, le partitionnement par intervalle découpe l'ensemble des bandes en k intervalles. Chaque intervalle comprend un nombre différent de bandes, mais un nombre similaire de points. Ainsi, la quantité de travail à réaliser dans chaque intervalle est comparable et la charge est équilibrée. Pour décider des intervalles, on peut estimer la répartition des points dans chaque bande par un échantillonage aléatoire. Chaque intervalle s'obtient en ajoutant des bandes à l'intervalle courant tant que le nombre de points de l'intervalle ne dépasse pas $\frac{n}{k}$.

En pratique, nous utilisons l'équilibrage en *round-robin* pour sa simplicité de mise en œuvre.

5.4 Canopy clustering distribué : approche par ensemble indépendant maximal

Nous avons présenté une première approche pour distribuer le calcul du canopy clustering. Cette approche a le désavantage de contraindre le choix des représentants en cas de conflit, ce qui ne permet pas de définir une priorité entre les points par exemple. Dans cette section, nous présentons une seconde approche qui ne présente pas cet inconvénient.

5.4.1 Principe

Notre approche par ensemble indépendant maximal se déroule en deux temps. Dans un premier temps, les paires de points plus proches que la distance d'agrégation voulue sont détectées. Elles correspondent à des couples de points qui ne devront pas être tous deux représentants. Ces paires constituent les arêtes d'un graphe de disque. Dans un second temps, les représentants sont choisis sans créer de conflit en calculant un ensemble indépendant maximal dans le graphe de disques.

5.4.1.1 Graphe de disques

Un graphe de disques (*unit disk graph*) est la modélisation sous forme de graphe de la relation de proximité dans un ensemble de points. Un graphe de disque est construit à partir d'un ensemble de points dans l'espace. Chaque point ayant une position (x, y), une arête du graphe relie deux points si la distance qui les sépare est inférieure à une distance d choisie. Ce graphe peut aussi être modélisé de deux

manières différentes au moyen de disques. Tout d'abord, chaque point est assorti d'un disque de rayon d et une arête est créée si un point se trouve à l'intérieur d'un disque. Alternativement, les disques peuvent avoir pour rayon $d/2$ et chaque arête correspond alors à l'intersection de deux disques. Ces deux définitions sont équivalentes. Pour les représentations de graphes de disques de ce chapitre, nous prendrons la seconde, puisqu'elle permet de minimiser la surface totale des disques et de leurs intersections et donc l'occlusion qui en résulte.

5.4.1.2 Ensemble indépendant maximal

Dans un graphe, un ensemble indépendant est un ensemble de sommets qui ne comprend pas deux sommets voisins. Cet ensemble est dit maximal s'il n'est pas possible de rajouter un sommet sans violer la propriété d'indépendance. Un ensemble indépendant maximal a alors deux propriétés :

- Si un sommet est dans l'ensemble, aucun de ses voisins n'est dans l'ensemble (propriété d'indépendance).
- Si un sommet n'est pas dans l'ensemble, il a au moins un voisin dans l'ensemble (propriété de maximalité).

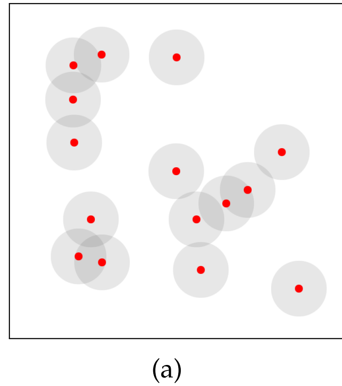
5.4.1.3 Utilisation pour le canopy clustering

Le problème de canopy clustering peut être réduit au calcul d'un graphe d'intersection de disques avec la distance d'agrégation d, puis d'un ensemble indépendant maximal au sein de ce graphe. Le graphe de disques permet de s'affranchir des contraintes géométriques pour se ramener à un problème de graphe. Après la construction du graphe, l'ensemble des représentants du canopy clustering correspond à un ensemble indépendant maximal :

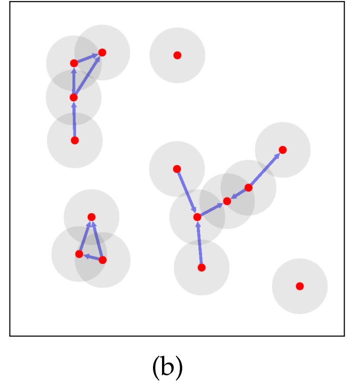
- Deux points plus proche que d seront voisins dans le graphe et ne pourront donc pas tous les deux appartenir à un ensemble indépendant. La propriété d'indépendance garantie donc la séparation des représentants.
- Un point doit être couvert par un représentant au plus à distance d. Il existe donc une arête entre ces deux points dans le graphe. Comme ce point n'est pas un représentant, au moins un de ses voisins l'est. La propriété de maximalité permet de garantir la couverture des points par les représentants.

5.4.2 Calcul du graphe de disques

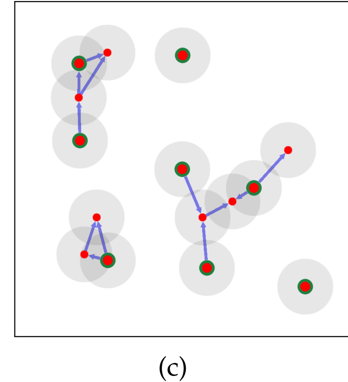
Le principe du calcul distribué du graphe de disques repose sur l'utilisation d'une grille pour partitionner les données et distribuer les calculs. En utilisant une grille infinie dont les cases ont une largeur d, il faut comparer chaque point d'une case à ceux des 8 cases adjacentes pour trouver les paires de points qui constituent les arêtes du graphe. Nous désignerons chacun de ces voisins au moyen des huit points cardinaux (voir Figure 5.6a). En plus de la comparaison des points au sein



(a)



(b)



(c)

FIGURE 5.5 – (a) : Représentation des disques de rayon d sur un ensemble de points. (b) : Graphe de disques pour cet ensemble de point. (c) : Les représentants choisis dans l'ensemble indépendant sont entourés en vert.

de la même case, il y a 8 tests à effectuer. Si ces 8 tests sont effectués pour chaque case, chaque test aura été effectué deux fois. Pour éviter de générer chaque arête du graphe de disques en double, il faudra donc trouver un moyen de n'effectuer chaque test qu'une seule fois. Les tests étant faits de manière symétrique pour une paire de cases, la comparaison doit pouvoir être faite de manière optimale en seulement quatre tests par case.

Ce type de calcul s'implémente en MapReduce en assignant comme clé à chaque point le numéro de sa case, puis en groupant les points par clé. Les points associés à la même case sont alors regroupés et peuvent être comparés. Cependant, dans un environnement distribué comme MapReduce, il n'est pas possible d'accéder aux données groupées sous d'autres clés, c'est-à-dire les points associés aux cases adjacentes. Seules les données qui ont la même clé peuvent être traitées conjointement lors de la phase de Reduce. Pour pallier cette limitation, nous proposons deux solutions.

5.4.2.1 Approche par duplication de données

La première solution que nous proposons repose sur la duplication des données. Lorsqu'un point doit être assigné à une case de la grille, il est dupliqué en 8 exemplaires supplémentaires qui sont assignés aux cases adjacentes. De cette manière, chaque case a connaissance des points situés dans les cases voisines, ce qui permet de faire toutes les comparaisons nécessaires. L'inconvénient de cette solution est que la taille des données initiales est multipliée par 9, ce qui peut être prohibitif lors du traitement de données massives. Cette solution est illustrée par la Figure 5.6a.

Cette solution implique que la comparaison de deux cases c_1 et c_2 est effectuée deux fois : une fois en comparant c_1 à c_2 et une fois en comparant c_2 à c_1. Soient deux points $p_1 \in c_1$ et $p_2 \in c_2$. En plus de ses coordonnées chaque point possède un identifiant noté $id(p_1)$ et $id(p_2)$. Si $id(p_1) < id(p_2)$, les deux points sont comparés lors

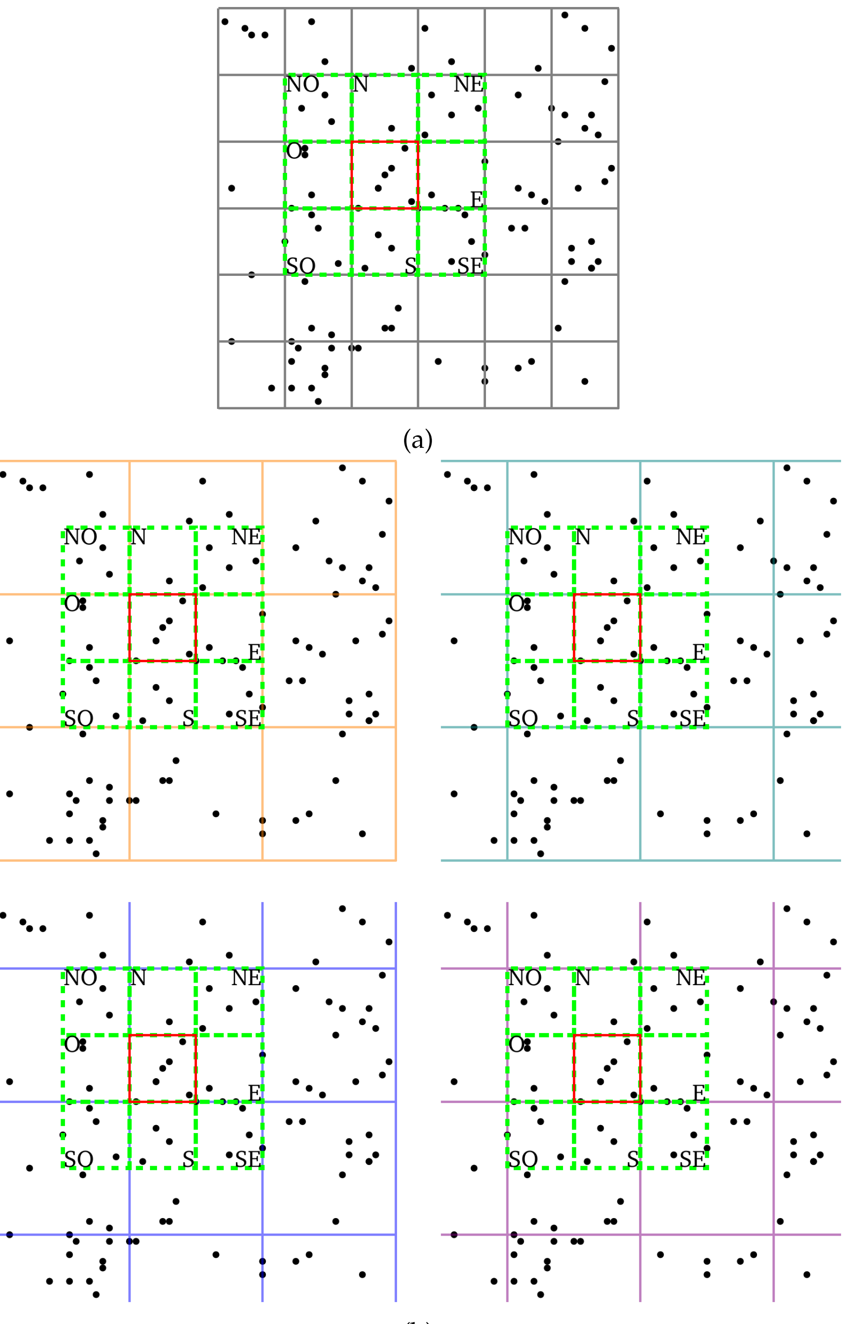
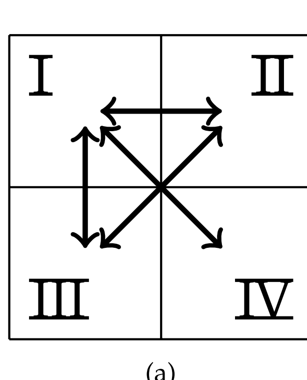
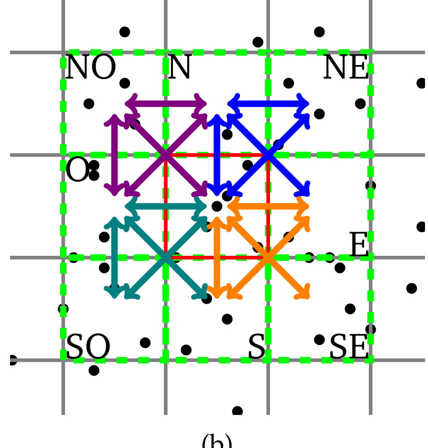


FIGURE 5.6 – Partitionnement par grilles. (a) Grille avec des cases de taille d. Les points des cases vertes sont dupliqués vers la case rouge et inversement, les points de la case rouge sont dupliqués pour chacune des cases vertes, ce qui donne une multiplication des données par 9. (b) Chacune des 4 grilles a des cases de taille $2d$, mais possède une origine différente : de haut en bas et de gauche à droite $(0, 0)$, $(d, 0)$, $(0, d)$ et (d, d). En combinant les 4 grilles, le même calcul que précédemment peut être effectué sans duplication des données, mais avec 4 étapes.



(a)



(b)

FIGURE 5.7 – Quadrants d'une case de la grille $\mathcal{G}'$. Les flèches représentent les comparaisons effectuées entre les cases. (a) Numérotation des quadrants. (b) Comparaisons dans les cases de $\mathcal{G}'$ qui contiennent la case rouge. La couleur des flèches correspond à celle de la grille utilisée sur la Figure 5.6b.

de la comparaison de c_1 à c_2. Dans le cas contraire, ils le seront lors de la comparaison de c_2 à c_1. De cette manière, chaque paire de points n'est comparée qu'une fois.

La taille des données utilisées par cette approche est $9n$, avec n le nombre de points total. Même si chaque paire n'est comparée qu'une seule fois, il est tout de même nécessaire d'itérer sur les $9n$ points. De plus, les $9n$ points doivent être présents en mémoire en même temps, ce qui sera souvent impossible pour de grandes quantités de données.

5.4.2.2 Approche multi-grille

La seconde solution que nous proposons scinde le problème au moyen de 4 grilles de taille $2d$. Cette fois, les données ne sont pas dupliquées, mais chaque grille est déplacée de d vers le bas et/ou la droite, ce qui donne les quatre décalages suivants : $(0, 0)$, $(d, 0)$, $(0, d)$ et (d, d). Ensemble, ces quatre grilles couvrent l'aire des 9 cases de la grille précédente, ce qui permet de faire le même calcul. Ces grilles sont illustrées sur la Figure 5.6b. Dans cette approche, il n'y a pas de duplication de données, les points sont successivement groupés selon une des quatre grilles.

Nous appellerons $\mathcal{G}$ la grille avec des cases de taille d et $\mathcal{G}'$ les quatre grilles avec des cases de taille $2d$. Chaque case de $\mathcal{G}'$ contient 4 cases de $\mathcal{G}$, que nous appellerons alors *quadrants* et désignerons par des chiffres romains : I en haut à gauche, II en haut à droite, III en bas à gauche et IV en bas à droite, comme l'illustre la Figure 5.7a. Chaque case de $\mathcal{G}$ occupera dans chaque grille $\mathcal{G}'$ un quadrant différent. De même, pour chaque case de $\mathcal{G}$ il existe une et une seule des quatre grilles $\mathcal{G}'$ dans laquelle cette case occupe le quadrant I et elle sera tour à tour groupée avec chacun de ses

voisins dans $\mathcal{G}$. En considérant une case $c \in \mathcal{G}$, on peut nommer les quatre grilles $\mathcal{G}'_{\mathrm{I}}$, $\mathcal{G}'_{\mathrm{II}}$, $\mathcal{G}'_{\mathrm{III}}$ et $\mathcal{G}'_{\mathrm{IV}}$ en fonction du quadrant qu'elle y occupe. Cette désignation est bien sûr relative à une case en particulier. Par exemple, la grille $\mathcal{G}'_{\mathrm{I}}$ pour c correspond pour son voisin Sud-Est à $\mathcal{G}'_{\mathrm{IV}}$.

Au sein de chaque case de chaque grille $\mathcal{G}'$, les comparaisons, illustrées sur la Figure 5.7, ont lieu entre les quadrants suivants : I et II, I et III, I et IV, II et III. Ainsi, pour une case de $\mathcal{G}$, chaque grille $\mathcal{G}'$ permet de la comparer aux voisins suivants :

- $\mathcal{G}'_{\mathrm{I}}$: Est, Sud et Sud-Est
- $\mathcal{G}'_{\mathrm{II}}$: Sud-Ouest et Ouest
- $\mathcal{G}'_{\mathrm{III}}$: Nord et Nord-Est
- $\mathcal{G}'_{\mathrm{IV}}$: Nord-Ouest

Chaque grille $\mathcal{G}'$ étant décalée par rapport aux trois autres, elle permet de comparer un ensemble de voisins différents. Ensemble, les quatre grilles permettent de comparer chaque case à chacun de ses voisins, comme le montre la Figure 5.7b. Ces quatre comparaisons sont donc suffisantes pour effectuer toutes les comparaisons nécessaires à la création du graphe de disques, ce qui donne un calcul équivalent à la solution précédente.

Cette approche permet de résoudre le problème avec un plus petit nombre de comparaisons de cases et sans duplication de données. Les données doivent être parcourues 4 fois (une fois par grille $\mathcal{G}'$), mais seuls n points doivent être gardés en mémoire, ce qui permet d'utiliser cette approche sur des jeux données plus volumineux. Si les ressources en mémoire vive le permettent, il est possible de faire 4 copies des données et de traiter chaque grille en parallèle. Cette approche est donc plus flexible que la précédente.

Une fois les comparaisons effectuées, on obtient la liste des arêtes du graphe de disques. La position des points n'est alors plus utile. Le graphe de disque a permis de transformer un problème géométrique en problème sur un graphe.

5.4.3 Calcul de l'ensemble indépendant maximal

Le calcul d'un ensemble indépendant maximal de manière séquentielle peut se faire à l'aide d'un algorithme simple : pour chaque sommet, si il n'a pas de voisin dans l'ensemble, on peut l'ajouter à l'ensemble. L'ordre dans lequel cet algorithme considère les sommets détermine l'ensemble qui en résulte. Un seul sommet est considéré à la fois, ce qui ne laisse pas la possibilité de choisir deux voisins en même temps. Le principe général de cet algorithme est illustré sur l'algorithme 5.3.

5.4.3.1 Algorithmes parallèles et distribués

Pour calculer un ensemble indépendant maximal à l'aide d'un algorithme parallèle ou distribué, la difficulté se trouve dans la phase de sélection des sommets à

Algorithme 5.3 : Principe des algorithmes de MIS.

Data : Un graphe $G = (V, E)$
Result : Un ensemble indépendant maximal.
1 $I = \emptyset$;
2 **while** $V \neq \emptyset$ **do**
3 $v =$ select a vertex from V;
4 $I = I \cup v$;
5 $V = V \setminus (v \cup neighbours(v))$;
6 **return** I

considérer lors de l'itération courante. Contrairement à l'algorithme séquentiel, il faut s'assurer de ne pas sélectionner deux voisins.

La technique la plus simple et la plus utilisée pour les algorithmes parallèles et distribués est de considérer un ordre sur les sommets. L'idée est alors de sélectionner les sommets qui sont des minimums locaux pour cet ordre. On est alors assuré de ne pas sélectionner deux voisins. Si l'ordre n'est que partiel, il faut être capable de casser les égalités par un autre ordre ou aléatoirement.

Tous les algorithmes qui utilisent cette technique de sélection se distinguent donc par le choix de l'ordre. Le choix le plus simple est de prendre l'ordre dans lequel les sommets sont considérés par l'algorithme séquentiel, comme décrit par Blelloch et al. [18]. Cet ordre a l'avantage d'être total et de donner un résultat déterministe et identique à l'algorithme séquentiel. Lors de chaque itération, un seul sommet par composante connexe sera sélectionné dans le pire des cas, ce qui donne un nombre d'itérations en $O(n)$.

L'ordre des sommets peut aussi changer aléatoirement à chaque itération. Dans ce cas, le résultat de l'algorithme n'est plus déterministe. Mais ce type d'algorithme permet généralement de sélectionner un ensemble de sommets plus grand à chaque itération, ce qui réduit le nombre total d'itérations. Luby [75] présente plusieurs algorithmes parallèles, dont un utilise une permutation aléatoire des sommets, déterminée lors de chaque itération. Il est donc nécessaire pour cet algorithme de connaître la taille du graphe. Métivier et al. [80] présentent aussi un algorithme où à chaque itération chaque sommet tire un nombre aléatoire compris entre 0 et 1. L'ordre qui en résulte a de très fortes chances d'être total et ne nécessite pas de connaître la taille du graphe. En pratique, c'est ce dernier algorithme que nous avons trouvé le plus simple à implémenter et efficace, même si la différence n'est que de quelques itérations en moyenne.

5.4.3.2 Implémentation dans le paradigme Pregel

Pour adapter un algorithme dans le paradigme Pregel, il faut prendre le point de vue d'un sommet. L'algorithme est alors composé d'une phase d'envoi de message où les sommets échangent les informations nécessaires à la sélection, puis vient

une phase où chaque sommet modifie indépendamment son état de sélection en se basant sur les informations qu'il a reçu. L'algorithme original inclus aussi la suppression des voisins des sommets sélectionnés de l'ensemble de départ. Comme Pregel ne permet pas de distinguer plusieurs phases à exécuter, tel une phase de sélection suivie d'une phase de propagation de l'information aux voisins, ces deux phases devront être combinées.

Nous considérons ici l'implémentation de l'algorithme de Métivier et al. [80], mais seul le critère de sélection change d'un algorithme à l'autre.

L'état de chaque sommet comprend deux valeurs : $\alpha \in [0, 1]$ est le nombre aléatoire tiré par chaque sommet à chaque itération et σ représente l'état de sélection du sommet. σ peut prendre trois valeurs qui correspondent à l'appartenance du sommet v aux ensembles V et I :

— NOT_SELECTED, qui est l'état par défaut, lorsque $v \in V$ et $v \notin I$.

— SELECTED lorsque le sommet a été sélectionné, donc $v \notin V$ et $v \in I$.

— NEIGHBOUR_SELECTED lorsqu'un des voisins de v a été sélectionné, ce qui correspond à $v \notin V$ et $v \notin I$.

Algorithme 5.4 : Programme Pregel pour l'ensemble indépendant maximal (à exécuter en parallèle par chaque sommet).

Data : α : le nombre aléatoire courant, σ : l'état de sélection courant, alphas : les nombres aléatoires envoyés par les voisins, sigmas : les états de sélection envoyés par les voisins

```
1  if σ == NOT_SELECTED then
2  |   minα = min(alphas);
3  |   neighbourSelected = sigmas.any(SELECTED);
4  |   if neighbourSelected then
5  |   |   σ = NEIGHBOUR_SELECTED;
6  |   |   α = 1;
7  |   else if α < minα then
8  |   |   σ = SELECTED;
9  |   else
10 |   |   α = random();
11 |   for each neighbour n NOT_SELECTED do
12 |   |   sendMessage(n, (α, σ));
```

L'algorithme 5.4 montre le programme qu'exécute chaque sommet. Seuls les sommets NOT_SELECTED participent à l'algorithme (l.1). Les messages envoyés par les voisins du sommet courant sont agrégés pour obtenir le minimum des α envoyés (l.2) et un booléen indiquant si au moins l'un d'eux est SELECTED (l.3). Ensuite, si le sommet possède un voisin sélectionné, il passe dans l'état NEIGHBOUR_SELECTED (l.5) et son nombre aléatoire est ramené à 1 (l.6), ce qui retire ce sommet de la compétition pour la sélection. Si aucun des voisins n'est sélectionné, on compare le α du sommet courant avec le minimum envoyé. C'est l'étape de sélection. Si le sommet est un minimum local, il passe à l'état SELECTED, sinon un nouvel α est tiré

aléatoirement. Enfin, le sommet envoie à tous ses voisins dans l'état NOT_SELECTED un message comprenant son α et son σ.

Cet algorithme garantit qu'un sommet passe à l'état SELECTED s'il est un minimum local et NEIGHBOUR_SELECTED si un de ses voisins est sélectionné. En revanche, la situation d'un sommet v NOT_SELECTED dont tous les voisins sont NEIGHBOUR_SELECTED est moins évidente. Chaque voisin enverra un dernier message à ce sommet. Il doit alors être sélectionné. Sinon, il ne recevra plus de message et restera NOT_SELECTED jusqu'à la fin de l'algorithme. C'est pour cette raison que le α des sommets NEIGHBOUR_SELECTED est mis à 1 (l.6). Cela permet au processus de sélection d'avoir lieu et de faire que v devienne un minimum local.

De même, les sommets de degré 0 ne sont pas pris en charge. Ils ne reçoivent pas de message puisqu'ils n'ont pas de voisins. Mais cela implique qu'ils peuvent de toute façon être sélectionnés. Une phase de précalcul permet de les identifier et de modifier leur état en conséquence.

L'algorithme termine lorsque plus aucun message n'est généré. Comme seuls les sommets NOT_SELECTED émettent des messages, tous les sommets sont donc dans l'état SELECTED ou NEIGHBOUR_SELECTED. Pour obtenir l'ensemble indépendant maximal, il suffit de filtrer les sommets SELECTED.

5.5 Conclusion

Dans ce chapitre, nous avons montré comment utiliser l'algorithme de canopy clustering pour réaliser une abstraction multi-échelle d'un jeu de données. Nous avons prouvé que le nombre maximal de représentants généré par cet algorithme est borné et comment choisir ce nombre pour contrôler le niveau de détail du résultat.

Nous avons aussi détaillé deux méthodes pour adapter le calcul de canopy clustering sur une plateforme de calcul distribué. La première méthode utilise un partitionnement de l'espace pour découper les calculs à effectuer. Cette méthode sera utilisée dans le chapitre 7. La seconde méthode utilise un graphe de disque pour détecter les conflits et un ensemble indépendant maximal pour les résoudre. Cette technique sera utilisée dans le chapitre 8.

6 Fatum : un middleware pour la visualisation d'information dynamique

La bibliothèque de visualisation est un élément essentiel de tout système de visualisation. Son rôle principal et le plus critique est d'assurer l'étape de rendu du pipeline de visualisation. Afin de ne pas gêner l'interactivité, la bibliothèque doit être capable de réaliser cette étape suffisamment rapidement. Les limites communément admises se situent entre 30 et 60 images par seconde ou *fps* (frames per second).

Un autre aspect primordial pour la visualisation de données massives est la dynamicité. Elle peut d'abord provenir d'une évolution des données à représenter, soit parce qu'on désire utiliser un autre jeu de données, soit parce que les données elles-mêmes sont dynamiques (évolution au cours du temps), soit parce qu'une nouvelle partie des données a été demandée suivant le mantra de Shneiderman. La dynamicité peut aussi se manifester par un changement de la représentation à appliquer aux données, c'est à dire un changement d'encodage visuel. La visualisation multi-échelle a la particularité de se trouver au croisement de ces deux types de dynamicités. Le sous-ensemble des données à afficher ainsi que leur représentation varient en fonction du niveau de détail choisi.

Les bibliothèques de visualisation sur le web peuvent se classer en deux catégories. D'un côté, des bibliothèques haut niveau, comme D3 ou sigma.js et de l'autre des bibliothèques bas niveau, comme WebGL.

Les bibliothèques dites haut niveau sont adaptées à un utilisateur final souhaitant créer une visualisation rapidement. La plus populaire est sans doute D3 [21]. La fonctionnalité principale de D3 est la manipulation du *DOM* (Document Object Model), la représentation hiérarchique des éléments d'une page web. Cette approche permet de capitaliser sur la connaissance préalable que les utilisateurs peuvent avoir de cet environnement. Depuis sa création, D3 a été enrichi de nombreux *plugins* qui élargissent son champ d'action. Le principe de D3 repose sur la sélection de sous-ensembles du DOM et la manipulation de leurs propriétés. D3 n'intègre pas de

moteur de rendu (partie logicielle dont le but est de transformer la représentation géométrique de la visualisation en image). Il repose pour cette partie entièrement sur le moteur SVG (scalable vector graphics) intégré aux navigateurs web. SVG est un langage de dessin vectoriel. Malheureusement, il n'a pas été conçu pour être capable de rendre des visualisations interactives. Ainsi, les performances de rendu atteignent rarement les 60 fps. De plus, la logique de manipulation directe du DOM de D3 impose à l'utilisateur de manipuler le langage SVG, dont la sémantique décrit des formes géométriques et n'est pas orientée pour créer des visualisations. Il est possible d'utiliser d'autres moteurs de rendu avec D3, comme WebGL, mais il est alors nécessaire de les maîtriser en plus de D3.

Une autre bibliothèque haut niveau populaire est sigma.js [1]. Contrairement à D3, elle ne se veut pas être une bibliothèque de visualisation généraliste. Elle est uniquement spécialisée dans la visualisation de graphe sous forme de diagramme noeud-lien. Elle intègre également son propre moteur de rendu, implémenté avec WebGL ou Canvas, ce qui lui permet de proposer des performances bien supérieures à celles de SVG. On peut aussi citer THREE.js [2], dont l'interface haut niveau rend accessible la programmation de scènes 3D. Cette bibliothèque n'est cependant pas dédiée à la visualisation et il n'est pas aisé de l'utiliser pour créer des visualisations.

Les bibliothèques dites bas niveau sont des bibliothèques qui utilisent une description géométrique. Nous avons déjà évoqué SVG, dont le moteur est intégré aux navigateurs web. Ce langage est aussi utilisé par Raphaël.js [3], dont l'interface permet de manipuler des objets SVG, mais n'est pas orienté pour la création de visualisations. De même, Processing.js [4] adopte une approche géométrique et utilise l'API Canvas d'HTML5. Enfin, WebGL est l'équivalent pour navigateurs web d'OpenGL. Cette librairie permet de faire du dessin 3D accéléré en utilisant la carte graphique de l'ordinateur. Cette technologie permet d'atteindre les meilleures performances de rendu possibles dans un navigateur web, mais impose une interface très bas niveau. Il n'est alors même plus question de dessiner des formes géométriques, mais uniquement des triangles qui seront traités ensuite en parallèle sur la carte graphique par des programmes écrits dans un langage proche du C, le GLSL. WebGL est donc assez difficile à prendre en main, mais apporte d'excellentes performances.

On voit donc que les bibliothèques haut niveau sont soit trop spécialisées, soit n'apportent pas des performances suffisantes pour de la visualisation interactive qui dépasserait les quelques milliers de points. Pour obtenir des performances satisfaisantes, il faudrait se tourner vers une bibliothèque bas niveau. Mais la prise en main de ces bibliothèques n'est pas aisée et il est compliqué de traduire la spécification d'une visualisation en terme de primitives géométriques.

Dans l'idéal, une bibliothèque de visualisation de données massives doit faire face aux deux problèmes de scalabilité déjà évoqués : perception et performance. La scalabilité de perception définit un seuil en nombre d'entités visuelles qui ne devra

1. http://sigmajs.org/
2. https://threejs.org/
3. http://dmitrybaranovskiy.github.io/raphael/
4. http://processingjs.org/

de toute façon pas être affiché en même temps sur un écran. Il s'agit alors pour la bibliothèque de pouvoir atteindre ce seuil en gardant des performances de rendu interactives. Il est difficile de quantifier ce seuil, puisqu'il dépend de l'encodage visuel choisi. Néanmoins, un ordre de magnitude de 100K entités visuelles nous semble être une estimation raisonnable. De plus, un rendu plus rapide, même avec un nombre d'entités visuelles limitées, permet de libérer du temps pour effectuer des calculs en temps réel qui peuvent soulager les besoins en précalcul. Ce temps supplémentaire peut par exemple permettre d'implémenter des interactions plus complexes.

Dans ce chapitre, nous présentons notre travail sur une nouvelle bibliothèque de visualisation qui vise à combiner une interface appropriée à la visualisation avec des performances apportées par un moteur de rendu bas niveau. En cela, notre bibliothèque est un intergiciel (*middleware*) qui permet de faire le pont entre les bibliothèques bas niveau et haut niveau. Le but est de pouvoir utiliser cette bibliothèque pour faciliter l'implémentation de systèmes de visualisation plus complets. Pour cela, nous présentons un modèle d'abstraction pour la visualisation composé de *marks* et de *connections*, chacune avec leur propriétés visuelles. Cette bibliothèque permet de gérer la dynamicité en définissant des états de la visualisation. Il est alors possible de passer d'un état au suivant en effectuant une animation prise en charge par la bibliothèque. Nous avons décidé d'appeler notre bibliothèque Fatum, pour *Fast Animated Transitions using Multi-buffers*.

6.1 *Marks* & *Connections* : un paradigme pour la visualisation d'information

Comme nous l'avons vu en introduction, il existe une corrélation entre les performances de rendu proposées par les bibliothèques et leur expressivité. En effet, les bibliothèques avec le rendu le plus efficace (sigma.js, WebGL) sont très spécialisées (noeud-lien pour sigma, uniquement des triangles pour WebGL), alors que les bibliothèques qui disposent d'un plus large éventail de possibilités ont des performances moindres. On peut en conclure que pour améliorer les performances de rendu, limiter le périmètre de la bibliothèque est une méthode efficace. Bien souvent, les modèles proposés par les bibliothèques sont trop resserrés ou au contraire, trop larges et inadaptés à la visualisation. Nous présentons donc ici un modèle cantonné à la visualisation d'information qui servira de base pour notre bibliothèque.

Notre modèle comporte deux entités visuelles de base : les *marks* et les *connections*. les marks encodent de l'information et les connections encodent des relations entre ces informations. Chacune de ces entités possède un ensemble défini de *propriétés* pour en modifier l'aspect visuel. Ce modèle est une spécialisation d'un modèle de graphe, où les marks correspondent aux sommets et les connections aux arêtes. On peut donc exprimer une visualisation avec la formule suivante, dérivée de la définition d'un graphe :

$$V = (M, C), C \in M \times M$$

Cette modélisation émerge du retour d'expérience des utilisateurs du logiciel Tulip [8]. Ce logiciel d'analyse de données possède de nombreux outils pour la gestion et la visualisation des graphes. Il utilise un modèle de graphe où chaque noeud et arête possède un ensemble de propriétés. L'utilisateur peut modifier les propriétés existantes ou en ajouter de nouvelles. Ce modèle comporte des propriétés spéciales, identifiées par le préfixe `view`, destinées à modifier l'aspect visuel des éléments du graphe. Les utilisateurs ont donc facilement repris ce système à leur avantage pour modéliser de nouvelles visualisations en encodant les éléments graphiques dans un graphe. Mais ce modèle a l'inconvénient de mélanger les données et leur représentation. Ainsi, pour toute visualisation un tant soit peu complexe, il existe dans le graphe qui la décrit certains sommets et arêtes uniquement destinés à décrire la visualisation. Ce mélange complexifie l'implémentation d'algorithmes sur ce graphe, puisqu'il faut alors penser à ignorer le cas échéant ces entités spécifiques.

Le modèle de *marks* et *connections* se propose de formaliser ce design émergent. Ce modèle permet de séparer les données à visualiser de leur représentation. Cela permet de concevoir plus facilement des représentations complexes où une donnée peut être représentée par plusieurs marks ou des glyphes, c'est à dire un agrégat de marks. L'autre avantage de ce modèle est qu'il évite de polluer les données avec des informations qui ne concernent que la représentation visuelle, et vice versa. Fatum ne propose donc aucun système pour gérer les données à visualiser, seulement leur représentation.

6.1.1 Marks

Notre modèle se base tout d'abord sur la notion de *marks* comme entités de base de la visualisation. Comme présenté en introduction de la thèse, cette notion découle des travaux de Bertin [12, 13]. Elle a depuis souvent été reprise comme base pour spécifier des visualisations. On la retrouve notamment dans Protovis [20, 58], Vega [98] et dans le livre "Visualization Analysis and Design" de Tamara Munzner [83].

6.1.1.1 Propriétés

Nous avons évoqué en introduction de la thèse les variables visuelles identifiées par Bertin. On retrouve ce principe chez Munzner sous le nom de "channels". Ce sont ces propriétés qui permettent de modifier l'aspect visuel d'une *mark*. Les propriétés issues de ces deux modèles que nous retenons sont les suivantes :

Position : contrôle la position sur les trois axes x, y et z de la visualisation. Nous utilisons une projection orthographique (sans perspective), donc l'axe z permet de décider quelle mark se trouve par dessus les autres.

Taille : contrôle la taille de la mark, en hauteur et en largeur.

Forme : La forme de la mark. Des formes de base sont prédéfinies : cercle, rectangle, diamant...

Couleur : contrôle la couleur de la mark. Cette variable est séparée en teinte et valeur chez Bertin, et teinte, luminance et saturation chez Munzner. Cett propriété comprend aussi le canal alpha, qui contrôle la transparence.

Rotation : orientation de la mark autour de son centre.

Texture : Il est possible d'ajouter une texture sous forme d'image sur une mark.

De plus, nous ajoutons de nouvelles propriétés pour compléter notre modèle.

Bordure : taille et couleur de la bordure d'une mark. La bordure suit le contour de la forme. Sa taille est définie en pixels, elle est donc constante à l'écran.

Découpe radiale, sous forme d'angle de début et de fin

Découpe centrale, exprimée comme un ratio du rayon.

6.1.2 Connections

Le rôle des connections est de modéliser une relation entre deux éléments. Ainsi, une connection relie toujours deux marks de manière orientée, elle possède donc une source et une destination. La plupart des propriétés sont donc dédoublées : une valeur du côté de la source et une valeur du côté de la destination. De plus, une connection ne possède pas de propriété de position, puisqu'elle utilise celles de ses extrémités.

6.1.2.1 Propriétés

— Taille : largeur de la connection. Une valeur côté source et une valeur côté destination. Cette valeur est ensuite interpolée linéairement le long de la connection.

— Couleur : couleur de la connection. Une valeur côté source et une valeur côté destination. Cette valeur est ensuite interpolée linéairement le long de la connection.

— Type d'extrémité : présence ou non d'une flèche au bout de la connection, du côté de la destination. On pourrait ici imaginer plusieurs types de flèches ou d'extrémité.

6.1.3 Ajout de texte

Le texte est souvent un élément important dans une visualisation. Il permet d'indiquer des informations supplémentaires, comme des noms de catégories ou des labels sur les sommets d'un graphe. Nous proposons donc un support d'éléments de texte avec les propriétés suivantes :

— Texte : Le texte à afficher.

— Position : La position dans le monde de la visualisation où le texte sera affiché.

— Ancre : Le point de la boite englobante du texte auquel correspond la position. Cette propriété permet d'attacher le texte à un endroit précis et de décider de sa direction.

— Couleur : La couleur du texte.

— Taille : La hauteur du texte. Le texte est donc un élément intégré au monde de la visualisation et sa taille est affectée par le zoom.

— Taille minimum et maximum : Ces propriétés représentent la taille minimum et maximum du texte en points, c'est à dire les tailles utilisées dans un traitement de texte. Elles permettent de garantir la visibilité du texte en évitant qu'il ne devienne trop grand ou trop petit à l'écran et reste lisible.

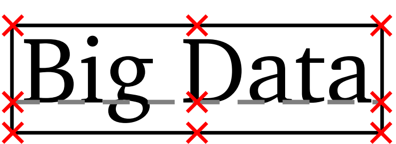
— Rotation : Un angle de rotation autour de l'ancre.

— Police : L'identifiant de la police de caractère à utiliser.

On peut voir sur la Figure 6.1 l'utilisation de ces propriétés, notamment les ancres.

Le texte est un objet complexe dont il est difficile de gérer tous les aspects, que ce soit le sens d'écriture, les ligatures qui peuvent exister dans certaines langues ou l'alignement et la décomposition en plusieurs lignes. Nous avons donc choisi de nous concentrer sur les points essentiels pour intégrer le texte à notre paradigme de visualisation. Il est par exemple rare de devoir intégrer du texte multi-ligne à une visualisation. Le texte n'est donc modélisé que sous forme de labels mono-ligne. Notre objectif étant de réaliser un intergiciel, une gestion plus complète du texte peut être implémentée par dessus.

Il aurait été possible de considérer le texte comme une propriété des marks, destiné à les étiqueter. Mais les nombreuses propriétés attachées au texte incitent à le considérer comme une entité autonome. De plus, lier le texte à une mark aurait contraint l'existence d'un texte à celle d'une mark. En considérant mark et texte de manière indépendante, il est tout à fait possible de constituer un objet plus complexe comprenant une mark et un texte pour les cas où cela est indiqué. Cette possibilité s'inscrit dans le positionnement de Fatum en tant qu'intergiciel.



(a) Les différentes ancres disponibles. Les ancres de la ligne centrale sont placées sur la ligne de base du texte.

(b) Exemple d'utilisation des propriétés du texte. Les textes ont la même position, représentée par la mark rouge, mais des ancres et des rotations différentes.

FIGURE 6.1

6.1.4 Point de vue dans la visualisation

Jusqu'à présent, nous avons considéré les éléments de la visualisation de manière isolée. Pour constituer une visualisation, il faut les assembler. Une visualisation peut être vue comme la réunion de marks, connections et textes dans un même monde abstrait. L'affichage est alors constitué par un point de vue sur ce monde. Nous considérons que ce monde doit être en deux dimensions, puisque l'usage de la troisième dimension n'est justifiable que lorsque les données à visualiser sont intrinsèquement en 3D. Ce monde définit un espace de coordonnées que nous appelons coordonnées *modèle*. L'axe des abscisses de cet espace est orienté vers la droite et son axe des ordonnées vers le haut.

L'utilisateur peut alors manipuler le point de vue pour faire apparaître d'autres parties de la visualisation. Notamment, il doit être possible de se déplacer suivant les axes horizontal et vertical et de zoomer. L'application de ce point de vue permet d'obtenir l'image finale de la visualisation. L'opération définit un nouvel espace de coordonnées que nous appelons coordonnées *écran*. Cet espace correspond aux pixels visibles à l'écran. Par convention, l'origine de ce repère est située en haut à gauche de la fenêtre de visualisation. L'axe des abscisses est orienté vers la droite et l'axe des ordonnées vers le bas. Les sens des ordonnées est donc inversé par rapport aux coordonnées modèle.

L'espace *modèle* est utilisé pour la plupart des propriétés. Cependant, certains aspects sont plus naturellement exprimés en pixels, dans l'espace écran. C'est par exemple le cas pour les tailles minimum et maximum du texte. L'environnement extérieur, notamment les autres frameworks, n'ont pas connaissance de l'espace modèle. Les évènements extérieurs, comme les interactions utilisateur, seront donc le plus souvent exprimées dans l'espace écran. Il est alors nécessaire de pouvoir convertir des coordonnées entre ces deux espaces pour traduire l'effet de chaque interaction.

6.1.5 Gestion de la dynamicité sous forme d'états

La dynamicité peut être vue comme une succession d'états. Un état de la visualisation est représenté par l'état de l'ensemble des objets qui la compose, ainsi que l'état du point de vue. Chaque évolution correspond à un nouvel état d'une partie ou de l'ensemble de la visualisation.

Les deux formes de dynamicité que nous avons vues précédemment, dynamicité des données et de la représentation, peuvent se modéliser comme trois types d'évènements :

- L'apparition d'un élément
- La modification des propriétés d'un élément
- La suppression d'un élément

Cependant, chaque évènement individuel ne doit pas correspondre à un nouvel état. Le passage à un nouvel état peut nécessiter une suite de plusieurs évènements ou bien ne correspondre qu'à un changement de point de vue. Il est donc plus facile de décider explicitement ce qui constitue un nouvel état ou non. Pour cela, on peut distinguer à un instant t deux états : l'état affiché et l'état en cours d'élaboration. L'état affiché ne peut être modifié, puisque déjà affiché. Tous les évènements de création, modification et suppression affectent donc l'état en cours d'élaboration. Une fois la description du nouvel état terminée, il est possible de valider le changement d'état.

6.1.5.1 Changements d'état : Échange & Animation

Le passage d'un état au suivant peut s'effectuer simplement en remplaçant l'état courant par le nouvel état. Le changement visuel est alors immédiat et abrupt. Ce type de passage peut être approprié lorsque les changements sont petits, comme lors d'un déplacement, ou alors si plusieurs changements d'état doivent se succéder rapidement.

On peut aussi procéder à un passage de l'état courant vers l'état suivant en utilisant une animation. L'animation a l'avantage de permettre de mieux suivre et comprendre les changements qui surviennent entre les deux états. Une animation entre deux états de la visualisation consiste en une animation de chacun de ses éléments.

Ajout et suppression d'éléments Lorsque l'utilisateur souhaite ajouter un élément (mark ou connection), cet élément ne doit apparaître que dans l'état suivant. Cependant, pour pouvoir éventuellement réaliser une animation, il est nécessaire que l'élément existe dans les deux états. Pour cela, lorsqu'un élément est ajouté à la visualisation, il est ajouté à la fois dans l'état courant et l'état suivant. Les valeurs de ses propriétés sont des valeurs par défaut. Notamment, comme rien ne doit

changer dans la visualisation au moment de l'ajout, l'élément ne doit pas être visible. Les valeurs par défaut permettent de garantir que l'élément sera bien invisible, notamment en mettant la valeur d'*alpha* à 0 pour rendre l'élément complètement transparent. L'utilisateur peut alors librement modifier les valeurs des propriétés dans l'état suivant pour réaliser l'effet d'apparition qu'il souhaite. Un effet très courant est le *fade-in*, où l'élément devient progressivement opaque. Cet effet est directement atteignable ici en modifiant la valeur de transparence. De plus, il est possible de modifier globalement ces valeurs par défaut pour éviter des modifications redondantes. Il est aussi possible de cloner une mark pour créer une nouvelle mark aux propriétés identiques et de copier l'ensemble des propriétés d'une mark vers une autre.

Cependant, les valeurs par défaut ne conviennent pas à toutes les situations. Par exemple, si un élément doit apparaître à l'emplacement de sa position finale sans déplacement, cette valeur doit être présente à la fois dans l'état courant et dans l'état suivant lors de l'animation. Il en va de même pour les autres propriétés, comme la forme, la taille ou la couleur. L'utilisateur doit donc avoir un moyen de modifier les valeurs de l'état courant pour obtenir l'effet désiré. Pour pallier ce problème, la bibliothèque permet de réaliser un changement d'état par copie restreint à un élément. De cette manière, l'état courant n'est modifié que pour cet élément sans que l'utilisateur l'ait explicitement modifié. En utilisant ce mécanisme, on peut réaliser un effet d'apparition en modifiant d'abord les propriétés qui doivent être identiques au début et à la fin de l'animation, copier l'élément dans l'état courant, puis modifier les propriétés à animer.

Lorsque l'utilisateur demande la suppression d'un élément, celle-ci ne peut être effective que dans l'état suivant, c'est à dire après la fin de la prochaine animation. Cet élément est donc seulement marqué pour suppression et son état est conservé. Une fois l'animation terminée, il pourra être réellement supprimé. Pour réaliser un effet de disparition (fade-in, rétrécissement, etc...), il suffit de modifier les propriétés de l'élément, puisque l'animation a lieu dans le même sens temporel que la disparition. En revanche, si l'élément est toujours visible à la fin de l'animation, la disparition sera abrupte.

6.1.5.2 Interpolation des éléments

Pour réaliser l'animation de chacun des éléments, il faut interpoler chacune de leurs propriétés. Pour modéliser la progression de l'animation comme une interpolation, un ratio d'interpolation est utilisé. Ce ratio démarre à 0 au début de l'animation et termine à 1 à la fin de l'animation. De cette manière, la progression de l'animation est indépendante de toute notion de durée ou de framerate. Pour connaître l'état de l'animation à un instant donné, il suffit de calculer quel doit être le ratio d'interpolation. Par défaut, la progression de ce ratio est linéaire au cours l'animation. Mais il aussi possible d'utiliser une fonction de *easing* pour modifier le déroulement de l'animation. Une fonction de *easing* peut être décrite comme suit :

$$e : [0, 1] \mapsto \mathbb{R}, e(0) = 0, e(1) = 1$$

Les conditions supplémentaires sur les valeurs de e en 0 et 1 permettent de garantir que les états initial et final seront bien ceux voulus. L'ensemble image de la fonction de easing sera généralement restreint à $[0, 1]$, comme décrit par Hudson et Stasko [63], mais il est possible de réaliser des effets d'élasticité en dépassant ces valeurs.

Interpolation des marks et connections La plupart des propriétés des marks et connections sont assez faciles à interpoler. Pour la position et la taille, l'interpolation est même triviale.

Pour interpoler la couleur, il faut la décomposer dans un espace de couleur, comme RGB ou HSV. Le choix de cet espace aura une influence sur l'aspect visuel produit lors de l'interpolation. La texture étant modélisée comme une image, il est préférable d'utiliser un effet de fondu pour réaliser l'interpolation. Pour la rotation et la découpe radiale, qui sont des propriétés cycliques, il est nécessaire de calculer dans quel sens doit évoluer l'interpolation pour emprunter le chemin le plus court.

Enfin, la seule propriété qu'il n'est au premier abord pas possible d'interpoler est la forme. A cause de son caractère discret, les valeurs intermédiaires n'ont pas de sens. Pour pouvoir réaliser une interpolation, nous exprimons la forme comme une valeur continue, au moyen d'un distance field (plus de détails à ce sujet dans la section 6.2.3). Les valeurs intermédiaires entre deux formes sont alors une déformation de la première vers la seconde et permettent de bien identifier le changement opéré.

Interpolation du point de vue L'interpolation des mouvements de caméra en 3D est un problème complexe qui nécessite des outils mathématiques avancés, comme les quaternions. Le point de vue de notre visualisation est exprimé uniquement en 2D et peut se caractériser par deux valeurs : une translation et un zoom. L'interpolation de ces deux valeurs permet d'interpoler le point de vue. Une interpolation linéaire est suffisante pour la translation, mais n'est pas correcte pour la valeur de zoom. En raison du caractère multiplicatif du facteur de zoom, une interpolation linéaire modifie aussi la translation. Il faut utiliser la formule suivante, adaptée du travail de Furnas et Bederson sur les diagrammes espace-échelle [50] :

$$z = \frac{1}{(1-t) * \frac{1}{z_1} + t * \frac{1}{z_2}}$$

Avec z_1 et z_2 les deux valeurs de zoom et $t \in [0, 1]$ le facteur d'interpolation. z est donc l'inverse de l'interpolation linéaire des inverses des valeurs de zoom z_1 et z_2.

Interpolation du texte Les propriétés continues du texte (position, couleur, taille et rotation) peuvent facilement être interpolées. En revanche, il est difficile de réaliser une interpolation des autres propriétés.

Le cas le plus complexe étant sans doute le texte lui même. Une approche naïve pourrait passer d'un texte à l'autre en parcourant l'ordre lexicographique. Une autre piste consisterait à faire correspondre chaque lettre du texte de départ avec une lettre du texte d'arrivée pour réaliser l'interpolation pour chaque lettre, mais ces textes auront le plus souvent un nombre de lettre différent et chaque lettre une taille différente. Mais dans aucune de ces approches, les valeurs intermédiaires n'ont de véritable sens et n'aident pas à comprendre le changement qui s'anime. La solution la plus sensée serait donc de traiter un changement de texte avec un simple effet de fondu pour passer d'un texte à l'autre, mais aucun des deux textes ne serait alors lisible durant une bonne partie de l'animation. L'interpolation des éléments de textes reste donc un problème ouvert. En pratique, nous n'avons pas trouvé d'usage à une animation des changements de texte.

6.1.6 Interface de programmation

L'API de Fatum s'organise autour du concept de vue. Une vue est le contexte qui contient un ensemble de marks et de connections. L'utilisateur peut créer une nouvelle vue attachée à un objet HTML5 `<canvas>`. A partir de la vue, il peut ajouter des marks et des connections.

Pour chaque propriété des marks et connection, il existe une fonction de lecture (*getter*) et une fonction de modification (*setter*) qui portent le nom de la propriété. Les méthodes d'accès ne prennent aucun argument et renvoient la valeur de la propriété. Les méthodes de modification prennent au moins un argument, plus suivant la propriété, et renvoient la mark pour permettre le chaînage de méthode. Par exemple, si `m` est une mark, il est possible de lire sa forme avec `m.shape()` et il est possible de la modifier avec `m.shape(newShape)`.

De plus, certaines propriétés comme la position ou la couleur sont en fait composées de plusieurs valeurs. Pour ces propriétés, plusieurs *getter* et *setter* sont disponibles. Par exemple, il est possible de modifier la position en x et y indépendamment ou encore la hauteur et la largeur. Ces accesseurs indépendants sont importants pour des valeurs qui sont souvent modifiées seules, comme la transparence `alpha` ou la position sur l'axe `z`.

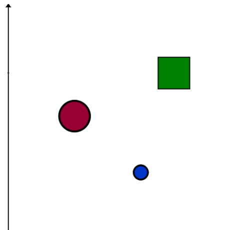
Les valeurs par défaut des propriétés sont celles qui seront appliquées lors de la création d'un élément. Il est important que l'utilisateur puisse les modifier, sans quoi il devra appliquer la modification sur chaque nouvel élément créé. Une mark et une connection spéciales sont donc disponibles pour modifier ces propriétés par défaut. Lors de la création d'un nouvel élément, les propriétés de l'élément par défaut correspondant lui sont appliquées, ce qui évite les modifications redondantes.

Enfin, comme évoqué précédemment, les méthodes de modification des propriétés peuvent être chainées. C'est à dire qu'il est possible de les appeler les unes à la suite des autres pour modifier plusieurs propriétés sur une ligne de code. Le principe du chainage de méthode est utilisé notamment dans Protovis [20] et D3 [21]. Il permet de rendre le code plus lisible en groupant les modifications de propriétés.

6.1.7 Exemples de visualisations

Maintenant que les principes de *mark* & *connection*, ainsi que l'API, ont été introduits, nous présentons comment il est possible de les utiliser pour réaliser des visualisations plus ou moins complexes. Chaque exemple est accompagné du code javascript qui permet de réaliser la visualisation.

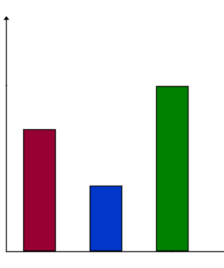
Scatterplot



Pour réaliser un scatterplot, chaque point est représenté par une mark. Les positions en x et en y dépendent de l'encodage visuel. Les propriétés de taille, couleur et forme peuvent être utilisées pour encoder plus d'information.

```
1 for(var d of data) {
2     view.addMark().x(d.x).y(d.y).color(d.color)
3         .size(d.size).shape(d.shape)
4 }
```

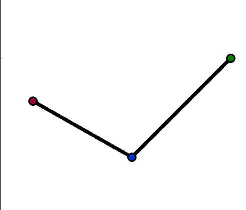
Histogramme



Pour réaliser un histogramme, chaque barre utilise une mark. La hauteur de la mark dépend de la valeur à encoder. Puisque la position de la mark est celle de son centre, la position verticale doit correspondre à la moitié de la hauteur et la position horizontale augmente linéairement avec le nombre de barres. Une version horizontale peut également être réalisée en inversant position horizontale et verticale d'une part et hauteur et largeur d'autre part.

```
1  // largeur de chaque barre
2  var w = ...
3  //espace entre chaque barre
4  var padding = ...
5  // ici, on a besoin de connaitre le rang dans les donnees pour la
       position horizontale
6  for(var i=0 ; i<data.length ; ++i) {
7      var d = data[i]
8      var pos_x = i*(padding+w)+w/2
9      view.addMark().x(pos_x).y(d.value/2)
10         .width(w).height(d.value).shape(Fatum.Shape.SQUARE)
11 }
```

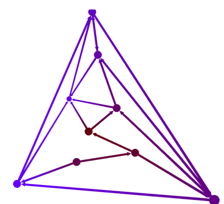
Série temporelle



Pour réaliser une courbe temporelle, la valeur à chaque pas de temps est représentée par une mark. Puis, chaque mark est reliée à celle du pas de temps suivant par une connection. Suivant l'aspect désiré, on peut utiliser ou non la flèche sur les connections ou faire une ligne brisée en n'affichant pas les marks.

```
1  // mark du pas de temps precedent
2  var previousMark = null
3  for(var i=0 ; i<data.length ; ++i) {
4      var d = data[i]
5      var current = view.addMark().x(i).y(d.value).color(d.color)
6      // creation de la connection
7      if(previousMark) {
8          view.addConnection(previousMark, current)
9              .sourceColor(previousMark.color())
10             .targetColor(current.color())
11     }
12     previousMark = current
13 }
```

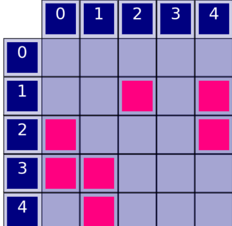
Vue noeud-lien



Le modèle de mark et connections utilisé par Fatum se prête bien à représenter les graphes sous forme de vue noeud-lien. Chaque sommet peut être représenté par une mark et chaque arête par une connection. Il est alors utile de mettre en place un index pour retrouver la correspondance entre marks et sommets du graphe.

```
1  var marksIndex = {}
2  for(var n of data.nodes) {
3      markIndex[n.id] = view.addMark().x(n.x).y(n.y)
4  }
5
6  for(var e of data.edges) {
7      view.addConnection(markIndex[e.source], markIndex[e.target])
8  }
```

Vue matrice

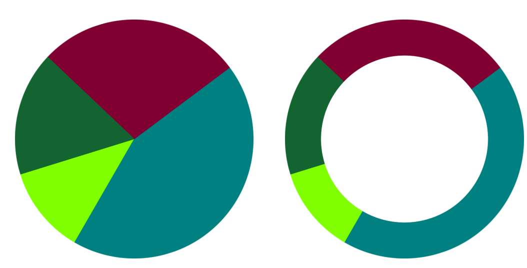


Dans une vue matrice, les sommets et les arêtes sont tous deux représentés par des marks. Pour cette visualisation, un même sommet est représenté par deux marks. Ce cas montre bien la flexibilité apportée par le fait de séparer les données et leur représentation. La position $(0, 0)$ se trouve au milieu de la case en bas à gauche de la matrice.

```
// entete des lignes et colonnes
for(var i=0; i<nbNodes ; ++i) {
    view.addMark().x(i).y(nbNodes+1)
    view.addMark().x(-1).y(nbNodes-i)
}

for(var i=0; i<graph.length ; ++i) {
    var e = graph[i]
    var m = view.addMark().shape(Fatum.Shape.SQUARE).red(255).x(e.
        source).y(nbNodes - e.target)
}
```

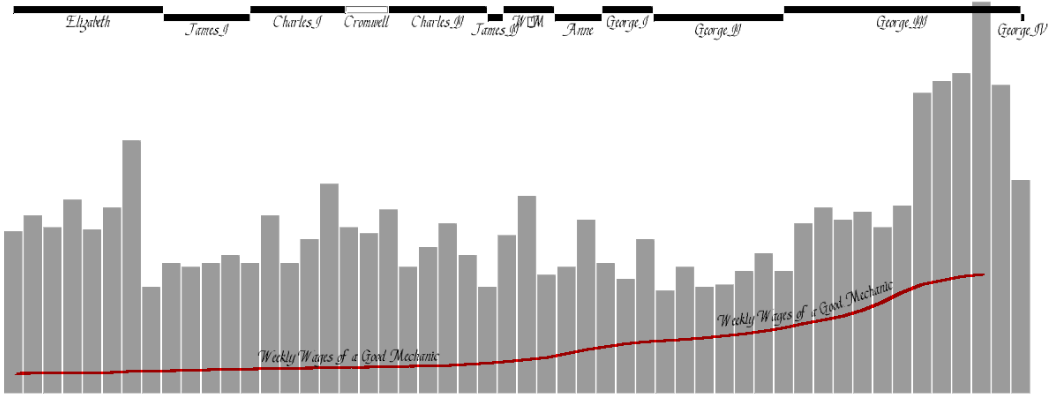
Camembert & Anneau



Les diagrammes circulaires peuvent être réalisés en utilisant une mark circulaire par secteur angulaire. Les positions et tailles sont toutes identiques. Seules les propriétés de clipping circulaire changent suivant le secteur. Pour faire un diagramme en anneau, il suffit d'ajouter un clipping interne.

```
// somme partielle pour calculer l'angle de depart de chaque secteur
var total = data.reduce((a,b)=>a+b.value, 0)
var subTotal = 0
var twoPi = 2*3.141592
for(var d of data) {
    view.addMark().color(d.color)
        .clippingStart(subTotal/total * twoPi)
        .clippingStop((subTotal+d.value)/total * twoPi)
    subTotal += d.value
}
```

Diagramme du blé de Playfair

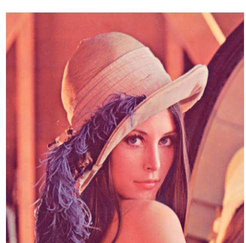


Pour reproduire le diagramme du prix du blé de Playfair, il faut combiner un histogramme et une courbe temporelle sous forme de ligne brisée. Pour reproduire la frise des monarques en haut de la visualisation, on utilise des marks rectangulaires noires. Celle correspondant à l'interrègne entre 1649 et 1660 est en blanc avec une bordure noire.

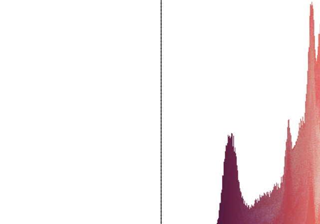
Le code de l'histogramme et de la courbe temporelle ayant déjà été donnés, voici le code pour la frise des monarques :

```
1  var white = [255,255,255,255]
2  var black = [0,0,0,255]
3  for(var i=0 ; i<monarchs.length ; ++i) {
4      var roy = monarchs[i]
5      // calcul de la duree du regne
6      var length = roy.end-roy.start
7      view.addMark().shape(Fatum.Shape.SQUARE).alpha(255)
8      // placement horizontal au milieu du regne
9      .x(roy.start+length/2)
10     // decalage vertical d'un regne sur deux, sauf l'interregne
11     .y( !roy.commonwealth && i%2 ? 95 : 97)
12     .height(2).width(length)
13     .color( roy.commonwealth ? white : black )
14     .borderColor(black).borderWidth(2)
15     view.addText().text(roy.name).x(roy.start+length/2).y((!roy.
           commonwealth && i%2) ? 94 : 96).anchor(Anchor.TOP)
16 }
```

Histomages



→



Histomages [30] propose d'éditer des images en manipulant individuellement leurs pixels. Il est possible de recréer les visualisations et les animations utilisées dans ce système avec Fatum en considérant chaque pixel de l'image comme une mark. Les animations se font ensuite en déplacant chaque mark à sa position sur l'histogramme. Le code pour visualiser l'image est le suivant :

```
1 for(var i=0 ; i < image.height ; i++) {
2     for(var j=0 ; j < image.width ; j++) {
3         var pixel = image.pixels[i][j]
4         view.addMark().x(j).y(i)
5             .color(pixel).shape(Fatum.Shape.SQUARE)
6     }
7 }
```

On peut ensuite passer à l'histogramme (ici, celui de la composante rouge) à partir des mark déjà créées :

```
1  // tableau pour stocker l'etat courant de l'histogramme
2  var histo = new Array(255)
3  histo.fill(0)
4  for(var i=0 ; i < view.numberOfMarks() ; i++) {
5      var m = view.getMark(i)
6      var red = m.red()
7      // positionnement de la mark en fonction de sa valeur de rouge et
           du nombre de mark deja dans l'histogramme a cet endroit
8      m.x(red).y(histo[red])
9      // incrementation du nombre de marks pour cette valeur de rouge
10     histo[red]++
11 }
12 view.animate()
```

Ce code définit un nouvel état de la visualisation vers lequel il est possible de passer en utilisant une animation.

6.1.8 Utilisation avec D3

Le principal interêt d'un intergiciel comme Fatum est de pouvoir être utilisé en conjonction avec d'autres frameworks. Nous montrons ici un exemple d'intégration avec D3, au travers d'une vue noeud-lien. Le code montré ici est dérivé d'un exemple D3 réalisé par Mike Bostock, disponible à l'URL suivante : https://bl.ocks.org/mbostock/4062045.

```
1  // initialisation de Fatum
2  Fatum.init();
3  var canvas = document.getElementById("fatum");
4  // creation de la vue et activation du rendu des marks et connections
5  var view = Fatum.createView(canvas);
6  view.layerOn(Fatum.MARKS | Fatum.CONNECTIONS);
7
8  var width=800, height=800;
9
10 var color = d3.scaleOrdinal(d3.schemeCategory20);
11
12 var simulation = d3.forceSimulation()
13     .force("link", d3.forceLink().id(function(d) { return d.id; }))
14     .force("charge", d3.forceManyBody())
15     .force("center", d3.forceCenter(width / 2, height / 2));
16
17 d3.json("graph.json", function(error, graph) {
18   if (error) throw error;
19
20   // fonction qui cree la visualisation de graph avec Fatum
21   createGraph(graph);
22
23   simulation
24       .nodes(graph.nodes)
25       .on("tick", ticked);
26
```

```
27    simulation.force("link")
28        .links(graph.edges);
29
30  });
```

```
1   function createGraph(graph) {
2     // definition des proprietes communes
3     view.defaultMark()
4       .alpha(255)
5       .width(1).height(1)
6       .borderWidth(2).borderColor([0,0,0,255])
7       .swap()
8
9     // index pour retrouver la mark qui correspond a un noeud
10    // utilise pour creer les connections
11    var nodesToMark = {};
12    for(var i=0 ; i<graph.nodes.length ; ++i) {
13        var n = graph.nodes[i]
14        var c = hexToRgb(color(n.cluster))
15        e.mark = view.addMark()
16          .red(c.r).green(c.g).blue(c.b)
17          .swap()
18        nodesToMark[e.id] = e.mark
19    };
20
21    graph.edges.forEach(function(e) {
22        var s = nodesToMark[e.source]
23        var t = nodesToMark[e.target]
24        e.connection = view.addConnection(s,t)
25          .sourceColor(s.color())
26          .targetColor(t.color())
27    });
28  }
```

Une fois le graphe créé, la fonction `ticked` est appelée à chaque itération de l'algorithme de dessin. Le rôle de cette fonction est de modifier la visualisation pour refléter l'évolution du dessin. Ici, il suffit de modifier la position de chaque mark, puis de changer d'état et de rafraîchir la vue.

```
1  function ticked() {
2      graph.nodes.forEach(function(e) {
3          e.mark.x(e.x).y(e.y);
4      })
5      view.swap();
6      view.refresh();
7  }
```

6.2 Implémentation

6.2.1 Choix des frameworks utilisés

L'objectif de Fatum est de se baser sur des frameworks bas niveau pour proposer une interface haut niveau adaptée à la visualisation. Étant donné son rôle critique dans un système de visualisation, l'implémentation doit être la plus efficace possible.

Nous avons donc choisi d'implémenter le coeur de Fatum en C++ et le moteur de rendu avec OpenGL. Cette combinaison permet de bénéficier d'un langage compilé très performant pour les parties les plus critiques. L'utilisation d'OpenGL permet d'avoir recours à la programmation sur GPU pour assurer un rendu rapide.

Mais ces technologies ne sont en l'état pas adaptées à l'utilisation dans un navigateur web. C'est pourquoi nous utilisons l'outil emscripten [5] [118] pour compiler le code C++ en javascript. Les appels OpenGL sont alors traduits en WebGL par le compilateur. De plus, Emscripten transforme le C++ en ASM.js. ASM.js est un sous-ensemble du langage javascript qui ne comprend que des opérations hautement optimisables par les navigateurs. Si le navigateur supporte ces optimisations, le programme bénificiera de performances accrues, sinon, le code ASM.js reste du javascript valide et peut être executé comme tel.

Le résultat est une bibliothèque javascript utilisable dans n'importe quel navigateur supportant WebGL. L'interface de programmation qu'offre le javascript est obtenue grace à l'outil *embind* qui permet de déclarer une correspondance entre des fonctions C++ et javascript. Fatum peut ainsi combiner une interface facile à utiliser dans un langage web populaire et les performances apportées par ASM.js et WebGL.

6.2.2 Gestion des animations par *double buffering*

La technique de double buffering est bien connue dans le contexte de la programmation 3D. Son but est d'éviter les artefacts graphiques qui surviennent lorsque l'écran est rafraîchi pendant la génération de la prochaine image. L'image affichée à l'écran est alors une partie de l'ancienne image et une partie de la prochaine image. Le double buffering permet d'éviter ce phénomène en maintenant deux buffers : le premier est appelé le front buffer, c'est celui qui est affiché à l'écran, le second est le back buffer, c'est celui dans lequel la nouvelle image est inscrite. Lorsque la nouvelle image est complète, les deux buffers sont échangés, c'est ce qu'on appelle le swap. L'écran peut alors être rafraichi sans mélanger les deux images.

En nous inspirant de ce principe, nous proposons une solution pour permettre l'animation des marks & connections sans surcoût. Le principe est le suivant : maintenir deux buffers avec deux états de la visualisation. Chaque buffer contient l'ensemble des valeurs des propriétés des marks de la visualisation. Le front buffer est celui dont les valeurs sont affichées avant l'animation, le back buffer est celui

5. `http://www.emscripten.org/`

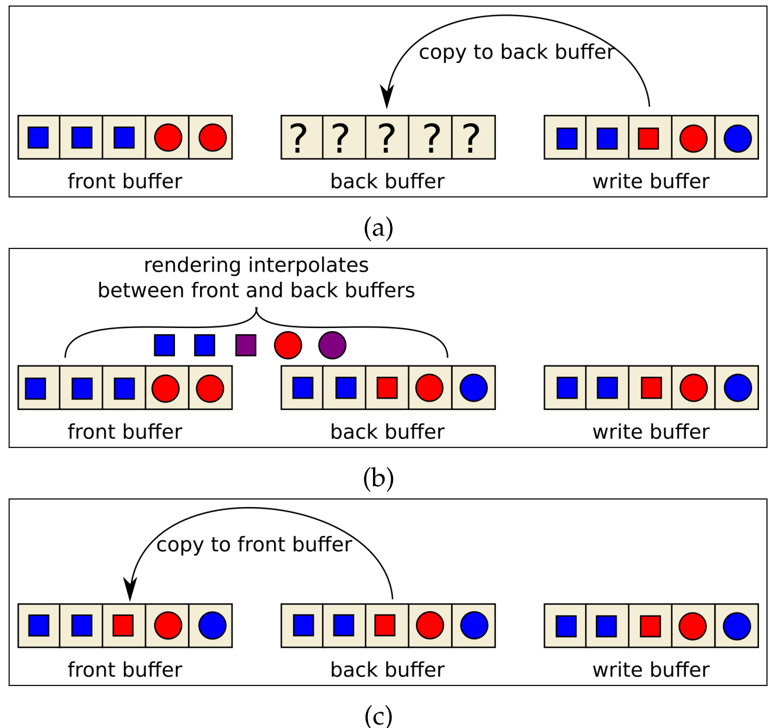


FIGURE 6.2 – Animation par *double buffering*. (a) Au lancement de l'animation, le contenu du *write buffer* est copié dans le *front buffer*. (b) Pendant l'animation, le moteur de rendu interpole chacune des propriétés entre les valeurs du *front* et *back buffer*. (c) Une fois l'animation terminée, le *back buffer* est copié dans le *front buffer*.

dont les valeurs seront affichées à la fin de l'animation. Lors du rendu, les deux états sont envoyés au moteur de rendu avec le ratio d'interpolation courant. Le moteur de rendu peut alors calculer la valeur à afficher pour chaque propriété. Ce processus est illustré par la Figure 6.2. La quantité de calcul nécessaire à un rendu est alors identique que l'on soit durant un rendu statique ou au milieu d'une animation. De plus, l'interpolation des propriétés ne doit être effectuée que pour les éléments visibles.

Pour assurer que les états de départ et d'arrivée ne changent pas pendant l'animation, il est nécessaire d'utiliser un troisième buffer, que nous apellerons write buffer. Ce buffer sert à stocker les modifications demandées par l'utilisateur. Au début de l'animation, il est copié dans le back buffer. Ainsi, si l'utilisateur modifie l'état courant pendant une animation, seul le write buffer est modifié et non le back buffer, ce qui casserait l'animation.

Ce système d'animation permet de plus de sauvegarder l'état de la visualisation à un instant donné pour pouvoir y revenir plus tard.

Pour implémenter le double buffering, les deux états sont envoyés à la carte graphique, ainsi que le ratio d'interpolation courant. L'interpolation de toutes les

Forme	Formule de distance	Distance Field
Cercle	$\sqrt{x^2 + y^2} - r$	
Carré	$max(\|x\|, \|y\|) - r$	
Losange	$\|x\| + \|y\| - r$	
Squircle	$\sqrt[4]{x^4 + y^4} - r$	

TABLE 6.1 – Exemples de distance fields, avec leur représentation implicite et explicite. Le niveau de gris représente la valeur absolue de la distance. Le blanc correspond à une distance de 0, c'est-à-dire le bord de la forme.

propriétés peut alors se dérouler en parallèle. Lorsqu'on doit effectuer un rendu en dehors d'une animation, le même principe est utilisé, mais le ratio d'interpolation ne varie pas, il reste à 0. De cette façon, il n'y aucune différence entre le rendu statique et lors d'une animation. Ainsi, les performances de rendu ne sont pas dégradées lors d'une animation.

6.2.3 Animation de la forme par *distance field*

Pour résoudre le problème d'interpolation des formes, il est nécessaire de donner à cette propriété un caractère continu. Pour cela, chaque forme est décrite par un *distance field*, c'est-à-dire qu'il est possible de connaître en chaque endroit de l'espace la distance au bord de la forme. Ce distance field peut être implicite ou explicite.

FIGURE 6.3 – Processus d'animation des formes par interpolation de distance field. L'animation permet de passer d'une forme en squircle (courbe de Lamé d'ordre 4) à un diamant. L'image centrale montre l'état à la moitié de l'animation.

6.2.3.1 Distance field implicite

Un distance field implicite est modélisé par une fonction qui donne pour un point de l'espace sa distance au bord de la forme décrite. Par convention, les distance field sont définis dans un carré de 2 unités de côté, dont le centre est en $(0, 0)$. On peut donc définir un distance field implicite comme suit :

$$DF : [-1, 1]^2 \mapsto \mathbb{R}$$

Les avantages d'un distance field implicite sont multiples. Tout d'abord, son stockage ne requiert que très peu de mémoire. Ensuite, il est possible d'avoir une précision arbitraire sur la distance. Il est aussi possible de paramétrer la fonction pour générer toute une famille de formes. Un exemple très utile est la famille des superellipses, ou courbes de Lamé. La Table 6.1 montre des exemples de distance field avec leur formule implicite et la représentation explicite. D'autres formules peuvent être trouvées dans [95]. Dans la plupart des cas, une fonction d'approximation de la distance peut même être suffisante.

6.2.3.2 Distance field explicite

Un distance field peut aussi être stocké sous forme explicite, c'est-à-dire sous forme d'une image qui encode la valeur du distance field à chacun des pixels. Cette représentation est préférable si la forme ne peut pas s'exprimer comme une fonction de distance ou que cette fonction est coûteuse à calculer. Mais le stockage explicite limite la résolution du distance field et occupe plus de mémoire. Les distance field explicites sont aussi utilisés pour le rendu de texte avec anti-crénelage [52, 94].

6.2.3.3 Interpolation de distance field

Une fois que chaque forme est exprimée à l'aide d'un distance field, il est possible de les interpoler. En effet, l'interpolation des valeurs de deux distances field à chaque pixel donne une déformation continue entre les deux formes, comme le montre la Figure 6.3.

6.2.4 Occlusion de texte avec garantie de visibilité

Le problème d'occlusion apparaît également lorsque beaucoup de textes sont affichés. Les textes affichés ne sont alors plus du tout lisibles, donc plus exploitables. Ici, le texte apporte un complément d'information au reste de la visualisation, il est donc possible de n'en afficher que certains pour éviter l'occlusion. On peut aussi réduire la taille du texte affiché. Cependant, s'il devient trop petit, on ne pourra plus le lire. De même, un texte trop grand peut sortir de l'écran et devenir tout aussi illisible. Il est donc préférable de limiter la taille à l'écran des textes affichés. Il en découle qu'un texte peut changer de taille par rapport aux éléments de la visualisation lorsque l'utilisateur effectue un zoom. Cette dynamicité de la taille du texte implique que le calcul d'occlusion destiné à choisir les textes à afficher doit être recalculé pour chaque image rendue. Les méthodes utilisant du précalcul sont donc inutiles, puisque les données sur lesquelles le calcul doit être fait changent à chaque fois, invalidant le précalcul.

De plus, il faut assurer la stabilité des textes affichés lors des interactions. C'est-à-dire que si un texte est affiché au temps t et qu'il est possible de l'afficher au temps $t + 1$, alors il doit l'être. Plus précisément :

- Lors d'un zoom-in, les labels déjà affichés doivent le rester, sauf s'il sortent de l'écran. D'autres textes peuvent devenir visibles dans l'espace libre, mais pas au détriment de texte déjà affiché.
- Lors d'un zoom-out, les textes déjà affichés ne doivent devenir invisibles que s'ils intersectent un autre texte déjà visible.
- Lors d'un pan, seuls les textes qui sortent de l'écran doivent disparaître.

Ces conditions permettent de conserver le focus donné par l'utilisateur.

Notre solution se décompose en deux étapes : le calcul de l'occlusion proprement dite puis le réordonnancement des textes pour que la prochaine phase d'occlusion donne la priorité à ceux déjà affichés.

6.2.4.1 Calcul de l'occlusion

Pour calculer l'occlusion, chaque texte est modélisé par un rectangle qui représente sa position dans l'espace. Comme il est possible d'appliquer une rotation aux textes, ces rectangles ne sont pas alignés avec les axes. Le but de l'algorithme est de décider pour chaque rectangle s'il peut être dessiné ou non. Un rectangle ne doit pas être dessiné si il intersecte un rectangle déjà dessiné. Comme l'on cherche à savoir quels textes doivent être affichés sur l'écran, il n'est pas nécessaire de tester les textes qui sont entièrement en dehors de l'écran. L'algorithme général est donné sur l'Algorithme 6.1.

L'algorithme qui permet de déterminer si deux rectangles ont une intersection est l'algorithme de la droite séparatrice qui peut être utilisé pour n'importe quel polygone. Cet algorithme cherche une droite qui sépare les deux polygones. Une telle droite peut être caractérisée par ces trois tests :

Algorithme 6.1 : Occlusion de texte

```
Data : Rectangles : collection des rectangles représentant les textes.
/* structure permettant d'enregistrer les rectangles dessinés et
   de vérifier les intersections                                  */
1 var intersector;
2 forall the Rectangles r do
3 |  if r.intersects(screen) && !intersector.intersects(r) then
4 |  |  intersector.add(r);
```

- Tous les sommets du rectangle A sont du même côté de la droite.
- Tous les sommets du rectangle B sont du même côté de la droite.
- Il existe un sommet de chaque rectangle de part et d'autre de la droite.

Pour vérifier de quel côté d'une droite se trouve un point, la méthode la plus efficace est d'utiliser le produit vectoriel entre un vecteur normal à la droite et un vecteur entre un point de la droite et le point à considérer. Le signe du résultat indique de quel côté le point se trouve.

Ensuite, nous avons besoin d'une structure de données capable de vérifier si un rectangle candidat intersecte un rectangle déjà dessiné. Le choix de cette structure de données est primordial pour la complexité de l'algorithme précédent. En effet, la solution la plus simple serait de faire la liste des rectangles déjà choisis. Mais cette solution implique une complexité de $O(n^2)$, puisque pour chaque candidat, tous les rectangles préalablement choisis sont examinés. Pour accélérer cette vérification, nous utilisons une structure d'index sous forme de grille afin de ne tester qu'un nombre réduit de rectangles face à un candidat.

6.2.4.2 Priorisation des textes déjà dessinés

La seconde partie de notre algorithme d'occlusion sert à maintenir la continuité temporelle. Pour cela, la priorité est donnée aux textes déjà dessinés. Nous choisissons de réordonner la liste des textes de manière à ce que ceux qui sont dessinés soient placés au début de la liste. De cette manière, lors du prochain calcul d'occlusion, les textes déjà dessinés ne pourront entrer en conflit qu'avec des textes eux aussi déjà dessinés.

Le réordonnancement de la liste revient à faire un tri selon un critère booléen. Il n'est pas requis que le tri soit stable, puisque l'ordre au sein de chaque classe (dessiné ou non) importe peu. Même si n'importe quel algorithme de tri peut être utilisé, il est possible d'utiliser un algorithme plus spécifique.

Puisque les valeurs sont triées suivant un critère booléen, on peut remarquer qu'après une seule itération de quicksort, le tableau sera trié. En effet, quicksort choisit un pivot et place tous les éléments inférieurs avant et les éléments supérieurs après par échange. De plus, les textes dessinés se trouvant au début de la liste et

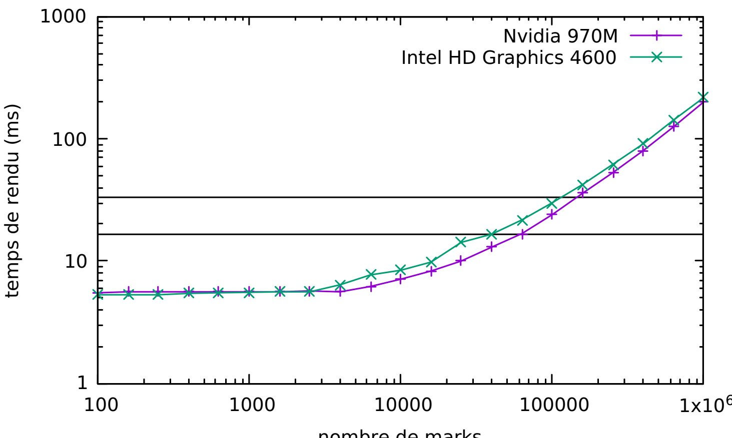


FIGURE 6.4 – Temps de rendu moyen mesurés lors de 5s d'animation d'un scatterplot réalisé avec Fatum, sur une échelle log/log. Les deux lignes horizontales correspondent au seuil de 60 fps (16.6ms) et 30 fps (33.3ms). On peut voir que les temps de rendu sont sous la barre des 60fps jusqu'à plus de 50K éléments et en dessous de la barre des 30fps jusqu'à environ 100K.

ceux non dessinés qui se trouvent à la fin sont déjà à la bonne place et n'ont pas besoin d'être considérés pour la procédure d'échange. En conservant lors de la phase de detection de collisions l'indice du premier texte non dessiné et celui du dernier dessiné, il est possible de restreindre la procédure d'échange à cette sous-liste.

6.2.5 Performances

Il n'est pas aisé de mesurer les performances d'une application qui utilise OpenGL ou WebGL car les instructions de rendu sont en fait asynchrones. C'est à dire qu'après l'appel à la dernière fonction de rendu, il n'est pas certain que le dessin ait bien été effectué par la carte graphique. De plus, nous voulons prendre en compte le temps utilisé globalement par la bibliothèque et non uniquement le temps utilisé par le GPU.

On peut trouver deux solutions à ce problème. La première consiste à forcer la bibliothèque de rendu à terminer le dessin avant de continuer au moyen d'un appel à la fonction `glFinish()`. Lorsque cette fonction rend la main au contexte appelant, toutes les instructions de dessin ont été effectuées. Cette solution permet alors de mesurer le temps de rendu d'une frame, mais les performances peuvent être dégradées car l'utilisation de la bibliothèque de rendu ne correspond pas à un cas réel. La seconde solution consiste à rendre plusieurs frames sans blocage, puis à mesurer le temps total. Cette solution correspond plus à un usage réel que la précédente, mais elle est alors susceptible d'être affectée par des éléments extérieurs.

Pour mesurer les performances de Fatum, nous nous sommes donc orientés vers

la seconde solution. Nous avons choisi de réaliser le test sur un scatterplot de points placés aléatoirement. Chaque point est ensuite animé pour se déplacer vers une autre position, aléatoire elle aussi. Durant l'animation, le nombre de frames rendues est compté, ce qui permet d'avoir une moyenne de la durée de rendu d'une frame. En faisant varier le nombre de points, on peut mesurer les performances de Fatum.

Les test ont été effectués sur un ordinateur portable équipé d'un processeur Intel i7-4710HQ et 16Go de mémoire vive. Les deux processeurs graphiques ont été testés : le chipset Intel HD Graphics 4600 intégré au processeur et la carte graphique Nvidia 970M. Le navigateur utilisé pour ces tests est Firefox 55. Les résultats visibles en Figure 6.4 montrent que Fatum permet de gérer jusqu'à 100K éléments en conservant des temps de rendu sous les 30fps.

6.3 Conclusion

Nous avons présenté dans ce chapitre l'élaboration d'une bibliothèque de visualisation sous forme d'intergiciel. Elle se base sur le paradigme de *marks* & *connections* que nous avons présenté. Ce paradigme permet d'exprimer une grade variété de visualisation avec un formalisme restreint.

De plus, la bibliothèque permet de gérer la dynamicité à la fois de la visualisation et des données à représenter au travers de transitions entre états. Ces transitions peuvent éventuellement être animées automatiquement en interpolant les propriétés visuelles définies pour les marks et connections.

Enfin, nous avons montré qu'en implémentant cette bibliothèque avec des technologies web modernes, comme WebGL et emscripten, les performances obtenues atteignent le seuil de perception fixé en début de chapitre.

Dans le cadre de cette thèse, sa bibliothèque Fatum nous a permis de construire des applications de visualisation, comme celles présentées aux chapitres 7 et 8.

Troisième partie

Applications

7

HeatMapReduce : Visualisation de carte de chaleur à grande échelle

Dans ce chapitre, nous montrons comment utiliser les concepts et techniques présentés dans la partie précédente pour permettre de visualiser des cartes de chaleur à grande échelle.

7.1 Introduction

L'idiome carte de chaleur présenté dans le chapitre 1 permet une très bonne scalabilité de perception en évitant l'occlusion. Il est donc un candidat idéal pour la visualisation de grands nuages de points.

Une carte de chaleur est créée à partir d'un nuage de points en appliquant une méthode appelée *Kernel Density Estimation* (KDE). Le KDE permet de transformer le nuage de points en une fonction continue qui estime en tout point du plan la densité des points de données. Pour être visualisée, cette fonction de densité est associée à une échelle de couleurs qui fait correspondre une couleur à chaque valeur de densité.

7.1.1 Kernel Density Estimation

La technique de *Kernel Density Estimation* a été introduite indépendamment par Rosenblatt [93] et Parzen [85]. Le principe consiste à remplacer chaque point par une fonction noyau [1] puis à faire la somme de ces fonctions, comme l'illustre la Figure 7.1. La fonction ainsi obtenue est l'estimation de la fonction de densité de l'ensemble de points de départ. Le KDE peut être résumé par la formule suivante, où n est le

1. Une fonction noyau modélise une distribution de probabilité et son intégrale sur $\mathbb{R}$ doit être égale à 1.

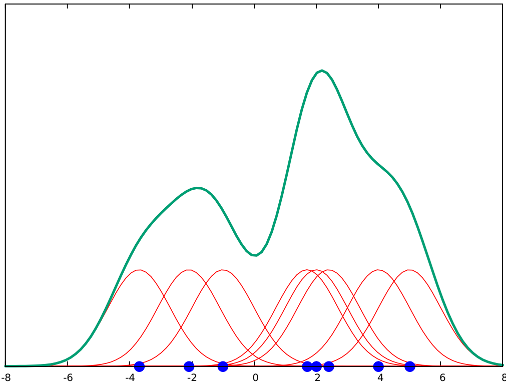


FIGURE 7.1 – Principe du Kernel Density Estimation. Chaque point est remplacé par la fonction noyau. La somme des noyaux donne l'estimation de la densité.

nombre de points, $w(p_i)$ est le poids associé au point i et K_σ est la fonction noyau :

$$KDE(p) = \frac{1}{n}\sum_{i=1}^{n} w(p_i)K_\sigma(dist(p, p_i))$$

Informellement, on peut voir le noyau comme servant à étaler le poids de chaque point autour de lui. Chaque point va alors contribuer une partie de son poids à chaque point du plan. Cet étalement donne l'effet de flou caractéristique du KDE. Le noyau possède un paramètre σ appelé fenêtre qui permet de contrôler l'intensité du lissage. Plus σ est élevé, plus le lissage est important. A partir d'une fonction noyau K de fenêtre unitaire, la version utilisant σ est définie par :

$$K_\sigma(x) = \frac{1}{\sigma}K(\frac{x}{\sigma})$$

Plusieurs fonctions noyau sont couramment utilisées, quelques exemples sont visibles dans la Table 7.1. Le choix de la fonction affecte l'apparence de la visualisation finale. Le noyau le plus souvent utilisé est le noyau Gaussien.

Pour produire une image à partir d'une estimation de densité, il faut calculer la valeur du KDE pour chaque pixel de l'image. Le calcul de la valeur du KDE pour un pixel consiste à faire la somme de la contribution de chaque point à ce pixel. Le calcul global de l'image a donc une compléxité en $O(np)$, avec n le nombre de points de données et p le nombre de pixels de l'image. L'application du KDE à des données massives ne permet donc pas d'espérer un rendu en temps réel d'une carte de chaleur.

Plusieurs techniques ont été développées pour apporter la scalabilité de performance au KDE. Michailidis et Margaritis comparent l'implémentation du KDE sur plusieurs frameworks dans un environnement multi-coeur [82]. Ils étudient également la possibilité d'utiliser le calcul GPGPU via Cuda [81]. Le calcul de KDE

Noyau	Formule	Courbe	Visualisation
Gaussien	$\frac{1}{\sqrt{2\pi}} \exp(-\frac{1}{2}x^2)$		
Uniforme	$\begin{cases} \frac{1}{2} & \text{si } \lvert x \rvert < 1 \\ 0 & \text{sinon} \end{cases}$		
Epanechnikov	$\begin{cases} \frac{3}{4}(1 - x^2) & \text{si } \lvert x \rvert < 1 \\ 0 & \text{sinon} \end{cases}$		
Triangulaire	$\begin{cases} 1 - \lvert x \rvert & \text{si } \lvert x \rvert < 1 \\ 0 & \text{sinon} \end{cases}$		

TABLE 7.1 – Exemples de noyaux utilisés pour le Kernel Density Estimation. Le noyau gaussien est le plus fréquemment utilisé. Les noyaux sont ici présentés avec une fenêtre σ unitaire. Le noyau gaussien produit un lissage beaucoup plus important pour un σ identique.

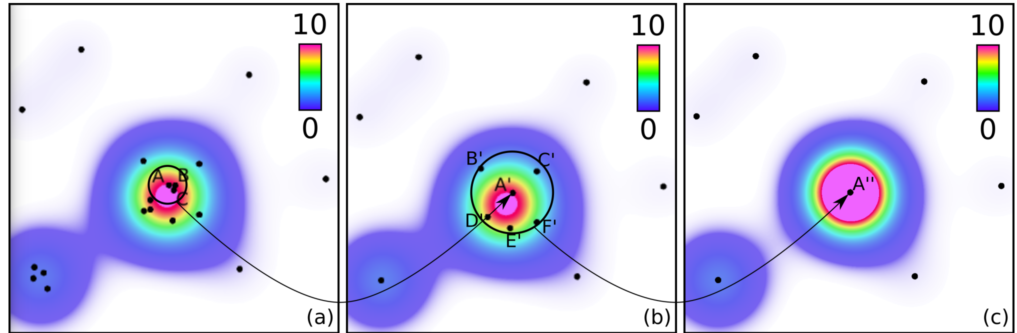


FIGURE 7.2 – Principe de l'Approximate Kernel Density Estimate. (a) L'ensemble de points de départ, ainsi que sa fonction de densité. (b) Les points A, B et C ont été groupés en A'. La fonction de densité du nouvel ensemble de points est très proche de l'originale, mais moins coûteuse à calculer. (c) Plus la distance entre les points groupés est importante, plus la différence avec la fonction de densité originale est grande.

en parallèle en utilisant un GPU est aussi étudié par Lampe & Hauser [70]. Ils proposent une méthode qui dessine chaque noyau un à un. La taille du noyau est limitée pour que le rendu ne concerne qu'une partie des pixels. L'accumulation des différents noyaux donne le KDE final. Mais aucune de ces méthodes ne permet d'envisager un rendu en temps réel d'une carte de chaleur de plusieurs milliards de points.

7.1.2 Approximate Kernel Density Estimate

Afin de réduire le temps de calcul du KDE et de la carte de chaleur associée, nous allons utiliser le principe du *Approximate Kernel Density Estimate* (AKDE), introduit par Zheng et al. [119]. Le but est de trouver à partir de l'ensemble de départ P un ensemble de points Q, tel que :

$$\|KDE_P - KDE_Q\| = max_{p \in \mathbb{R}^2} |KDE_P(p) - KDE_Q(p)| \leq \varepsilon$$

c'est-à-dire que la différence maximale entre le KDE calculé sur P et celui calculé sur Q est bornée. Le plus souvent on aura $Q \subset P$, mais ce n'est pas obligatoire. On cherche par contre à ce que $|Q| < |P|$, de façon à ce que le KDE de Q soit moins coûteux à calculer que celui de P.

Les auteurs de l'article original introduisent plusieurs façons d'obtenir un ensemble de points qui respecte ces conditions. Notamment, ils prouvent qu'en déplaçant chaque point de P d'au plus γ pour obtenir Q, la différence entre les deux KDE est bornée par $\frac{\gamma}{\sigma}$. La Figure 7.2 montre trois étapes successives d'agrégation pour obtenir des approximations de la densité d'un ensemble de points.

En ce qui concerne l'évaluation du KDE, déplacer des points $p_0, p_1, ..., p_n$ vers une même position revient à les remplacer par un seul point q tel que $w(q) = \sum_{i=0}^{n} w(p_i)$.

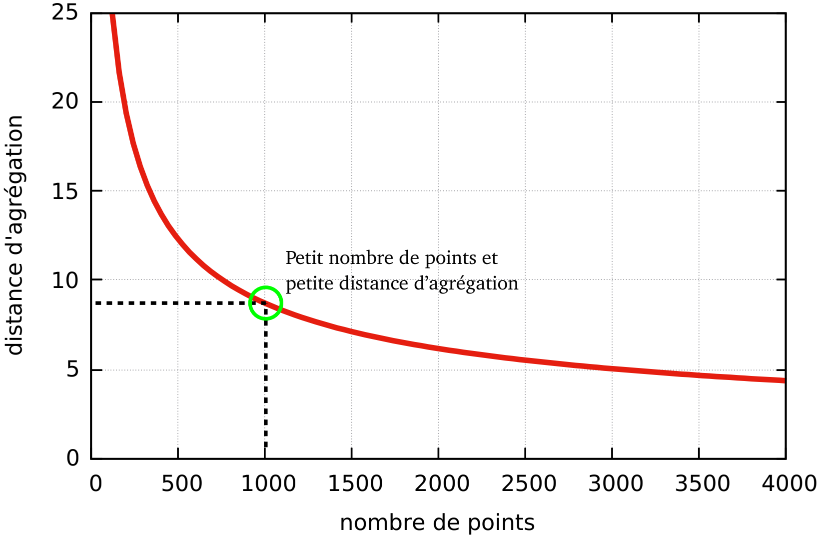


FIGURE 7.3 – Courbe du nombre de représentants en fonction de la distance d'agrégation en pixels.

L'ensemble des représentants générés par l'algorithme de canopy clustering respecte donc les conditions pour permettre un AKDE.

7.2 Vue d'ensemble du système

Pour permettre de visualiser des cartes de chaleur à grande échelle, nous allons tirer parti de la propriété d'approximation du AKDE. Un ensemble de niveaux de détail construits avec le canopy clustering, tel que décrit dans le chapitre 5, permet de visualiser des cartes de chaleur de plus en plus proches de celle du jeu de données initial à mesure que le niveau de détail augmente. Notre système implémentera donc la méthode de visualisation de données massives décrite au chapitre 4 en utilisant le canopy clustering multi-échelle du chapitre 5.

Deux paramètres doivent être déterminés pour implémenter les méthodes présentées précédemment : r le ratio de distance entre les niveaux et c_0 le nombre de représentants du niveau de détail 0. Pour déterminer r, nous prenons en compte la méthode d'indexation des données par une pyramide de tile. Dans cette pyramide, chaque niveau présente 4 fois plus de tile que le précédent. En prenant $r = 2$, chaque niveau de détail aura 4 fois plus de représentants que le précédent et donc il y aura un nombre maximal de représentants par tile de c_0 quelque soit le niveau de détail. Comme c_0 influe directement sur la distance d'agrégation, il est possible d'avoir une distance d'agrégation constante à l'écran si chaque tile occupe une surface constante. A partir de là, on peut tracer le nombre de représentants en fonction de la distance d'agrégation en pixels, visible sur la figure 7.3. Un compromis doit être fait entre l'erreur du KDE qui augmente avec la distance et le nombre de représentants, donc

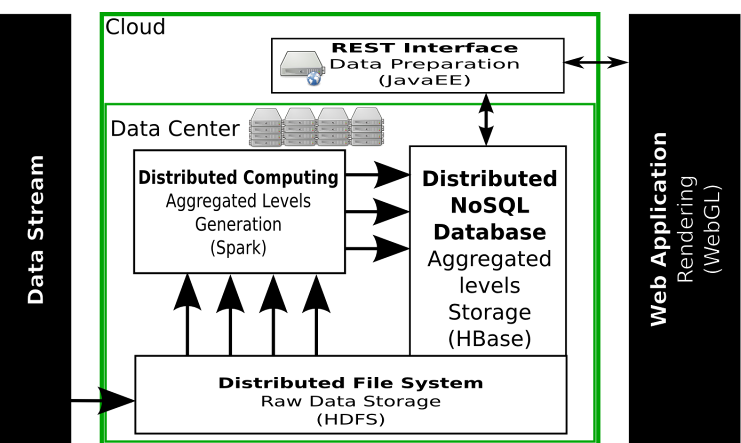


FIGURE 7.4 – Vue d'ensemble de l'architecture du système. Les données brutes sont d'abord stockées sur le système de fichiers distribué HDFS, puis agrégées récursivement avec Spark. Le résultat de l'agrégation est ensuite stocké dans HBase. Lors de la visualisation, le serveur frontal héberge l'application web et transmet les requêtes à la base de données.

la quantité d'information à traiter qui diminue avec la distance. Ici, nous avons choisi $c_0 = 1000$.

Notre système implémente la méthode de visualisation de données massives présentée au chapitre 4. Les différents composants et leurs relations sont visibles en Figure 7.4. Le système est basé sur l'écosystème Hadoop. Les données brutes sont stockées sur HDFS. Le traitement des données est implémenté avec Spark. Une fois les niveaux de détail calculés, les données indexées sont stockées dans HBase, où elles sont prêtes à être requêtées. Un serveur Web implémenté en Java expose une interface REST [2] pour permettre de communiquer avec l'exterieur sans exposer un accès direct à HBase. Enfin, le client de visualisation est implémenté sur dans un navigateur web. Le rendu s'effectue localement sur le client en WebGL.

7.3 Calcul en batch

Le calcul en batch des niveaux de détail requiert de calculer l'entièreté de la pyramide. Nous avons utilisé pour cela l'approche par partitionnement de l'espace présentée au chapitre 5. Nous l'avons préférée à l'approche par ensemble indépendant maximal puisqu'elle est plus rapide. La position relative des représentants choisis n'a ici pas d'importance puisque seule la fonction de densité sera représentée.

L'utilisation de Spark permet de conserver en mémoire vive le résultat de l'agrégation d'un niveau pour l'utiliser comme entrée du niveau suivant. L'implémentation est donc plus efficace qu'avec MapReduce, qui nécessite d'écrire ces données sur disque, de terminer le processus, puis de relancer un nouveau processus qui va

2. Representational State Transfer, permet d'accéder à des données à travers le protocole HTTP.

relire les données depuis le disque. Cet avantage accélère aussi le passage entre les deux jobs à implémenter.

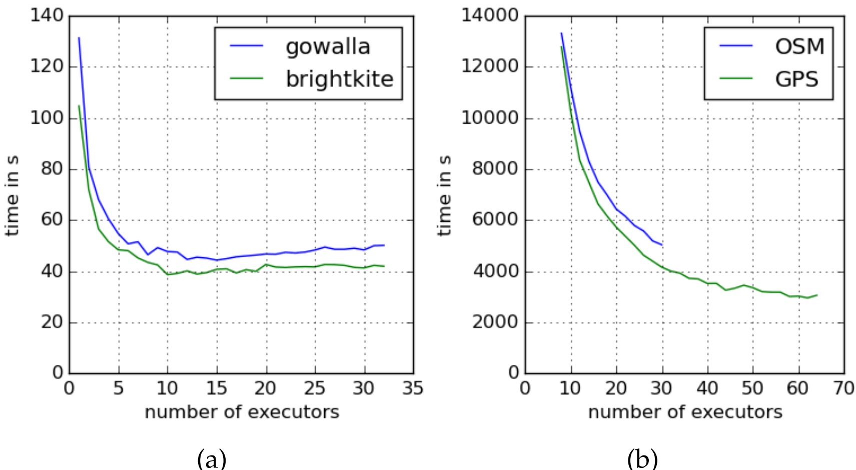
Spark permet d'utiliser les opérations `map` et `reduce`, mais permet également d'autres opérations plus complexes. L'implémentation utilise l'opération `map` pour calculer la bande de chaque point, puis `groupByKey` pour regrouper tous les points d'une même bande. Enfin, l'opération `flatMap` permet d'appliquer l'algorithme de canopy clustering : l'entrée est un tableau contenant tous les points de la bande et en sortie, chacun des éléments correspond à un point marqué pour savoir s'il est ou non représentant. Les mêmes opérations sont utilisées pour implémenter le second job. Seule l'opération `groupByKey` provoque un transfert réseau des données.

7.3.1 Performances

Jeux de données : Pour mesurer les performances de notre implémentation, nous avons utilisé plusieurs jeux de données. Les deux jeux de données les plus petits sont issus du Stanford Large Network Dataset Collection [72]. Ce sont deux collections de localisations GPS d'utilisateurs d'un réseau social. Le premier concerne le réseau social brightkite entre avril 2008 et octobre 2010. Il comprend 4.7 millions de lignes pour 693 mille positions différentes. Ce jeu de données a été utilisé par Liu et al. [74] et Li et al. [73]. Le second provient du réseau social gowalla entre février 2009 et octobre 2010. Il contient 6.4 millions de lignes pour 1.2 millions de positions différentes. Les deux autres jeux de données sont issus d'OpenStreetMap. Le premier, que nous appellerons OSM contient l'ensemble des points d'intérêt de sa base de données, pour 2.2 milliards de lignes et 1.7 milliards de positions uniques. Le second est un ensemble de traces GPS anonymisées enregistrées sur le site, pour 2.7 milliards de lignes et 2.2 milliards de positions uniques. La taille des jeux de données est indiquée en terme de positions uniques puisque l'agrégation groupe automatiquement les points situés à la même position. Ici, une première passe a été effectuée pour regrouper tous les points qui partagent une même position.

Infrastructure : L'infrastructure sur laquelle les tests ont été effectués est hébergée au sein de notre laboratoire. Elle comprend 16 machines équipées chacune de 64Go de mémoire vive, 2 processeurs de 6 cœurs hyperthreadés cadencés à 2.1GHz et 2 disques à plateau d'1To chacun. Durant les tests, une machine tient le rôle de noeud maître, tandis que les 15 autres exécutent les calculs.

Les tests ont été menés en faisant varier le nombre d'exécuteurs Spark utilisés pour réaliser l'agrégation. Chaque exécuteur possède des ressources définies en nombre de coeur et quantité de mémoire vive. Pour les tests des jeux de données brightkite et gowalla, chaque exécuteur dispose de 4 coeurs et 2Go de mémoire vive. Pour le test des traces GPS, chaque exécuteur dispose de 2 coeurs et 6Go de mémoire vive et 2 coeurs et 20Go pour le jeu de données OSM. En faisant varier le nombre d'exécuteurs utilisés, on peut observer le comportement du programme à mesure que de nouvelles ressources lui sont allouées. Les courbes qui montrent l'évolution



(a) (b)

FIGURE 7.5 – Résultats des test de performance pour les quatre jeux de données.

du temps de calcul en fonction du nombre d'exécuteurs pour ces quatre jeux de données sont visibles en Figure 7.5. Pour tous les jeux de données, 21 niveaux de détail ont été calculés.

Pour les quatre jeux de données, les tests montrent la scalabilité de cette approche. Pour les deux plus petits jeux de données, le temps de calcul diminue fortement jusqu'à 5 exécuteurs, atteint un minimum à 10 et augmente ensuite légèrement. Ces jeux de données bénéficient donc dans un premier temps d'une accélération grâce à la distribution des calculs, mais atteignent rapidement un point où l'ajout de ressources n'est plus bénéfique voire fait augmenter le temps de calcul. Ce phénomène est du à l'augmentation du coût de communication entre les exécuteurs qui n'est pas compensée par un plus grand partage du travail. En revanche, pour les deux autres jeux de données, on observe une bonne scalabilité, jusqu'à pouvoir calculer l'agrégation du jeu de données GPS en moins d'une heure. Les calculs du jeu de données OSM ont du être interrompus à 30 exécuteurs puisque les ressources nécessaires dépassent ensuite la taille de notre infrastructure.

On peut remarquer que les temps de calcul ainsi que la mémoire nécessaire sur le jeu de données OSM sont plus importants que pour le jeu de données GPS, malgré un nombre de points initial inférieur. Ceci est dû à la distribution des points dans l'espace. Les données OSM sont plus uniformément réparties alors que les traces GPS sont plus concentrées. Lors de l'agrégation le nombre de représentants produits diminue donc plus rapidement pour les données GPS, ce qui affecte le temps de calcul total.

7.3.2 Stockage

La pyramide de tiles permet d'indexer les données. Elles sont stockées dans HBase pour pouvoir être requêtées et transférées rapidement. Pour chaque tile,

repéré par son triplet (l, i, j), il doit être possible de retrouver les points qui appartiennent à ce tile. Nous avons comparé trois formats de stockage différents :

— Points : Chaque ligne de la table contient un point sous forme d'une position x, y et un poids.

— Tile Gzip : Chaque ligne contient l'ensemble des points d'un tile sous forme d'une chaîne de caractères organisée en csv. Elle est ensuite compressée au format Gzip.

— Tile binaire : Comme le format précédent, tous les points d'un tile sont stockés sur la même ligne, mais en binaire. L'information de chaque point contient deux nombres flottants sur 8 octets et un entier sur 4 octets. Chaque point occupe donc 20 octets.

7.3.2.1 Clés et équilibrage de charge

Le choix des clés à utiliser pour chaque format influe sur les requêtes qu'il est possible de faire. Il doit être possible de récupérer en une seule requête tous les points d'un tile. De plus, les clés sont utilisées par HBase pour décider sur quelle machine du cluster la valeur correspondante sera stockée. Il faut alors s'assurer que les données seront uniformément réparties pour équilibrer la charge de stockage et de requêtes entre les machines. Un déséquilibre dégraderait les performances du système.

Pour les deux formats de stockage sous forme de tile, la même clé est utilisée. Elle correspond au triplet (l, i, j) du tile correspondant. Pour permettre la répartition du stockage, la méthode du "salage" est utilisée. Elle consiste à rajouter un préfixe à la clé pour répartir les clés uniformément dans l'espace des clés possibles. Cela permet d'obtenir un équilibrage au sein du cluster, puisque HBase utilise l'ordre lexicographique des clés pour décider de leur emplacement. Mais ce préfixe ne doit pas être aléatoire, puisqu'il fait partie de la clé et doit pouvoir être retrouvé pour faire une requête. Nous utilisons la fonction de hachage MurmurHash pour hacher le triplet du tile en guise de préfixe. De cette manière, la clé d'un tile peut être retrouvée uniquement à partir de son triplet.

Cette manière de procéder n'est en revanche pas suffisante pour le format de stockage sous forme de points individuels, puisque toutes les clés doivent être différentes. Pour ce format, est ajouté à cette clé la position x, y de chaque point. Grâce à l'agrégation, il n'y a pas deux points à la même position. Chaque clé est donc unique. De plus, dans l'ordre lexicographique, tous les points d'un tile sont contigus, puisqu'ils ont le même préfixe correspondant à l'identifiant du tile. Pour récupérer tous les points du tile, on utilise alors une requête du type "SCAN" qui permet d'obtenir les valeurs d'une plage de clés contiguës. Il suffit alors de calculer la boite englobante du tile et d'utiliser les valeurs minimales et maximales de x et y pour pouvoir construire les clés de début et de fin de la plage à requêter.

7.3.2.2 Taille du stockage

Jeu de données	Brut	Points	Gzip	Binaire
brightkite	364 MB	379 MB	381 MB	300 MB
gowalla	376 MB	668 MB	661 MB	515 MB
OSM	62 GB	609 GB	277 GB	259 GB
GPS	72 GB	321 GB	100 GB	128 GB

TABLE 7.2 – Taille occupée par les données dans les différents formats. Il s'agit de l'espace disque occupé sur HDFS par les tables HBase, sauf pour les données brutes qui indique la taille du fichier d'origine.

La Table 7.2 détaille l'espace disque occupé par chacun des formats de stockage. Par rapport aux données brutes, il y a bien sûr un surcoût lié au stockage de la totalité de la pyramide de tiles. Le stockage par points individuels a le surcout le plus important, puisqu'il utilise beaucoup plus de clés que les deux autres (une par point) et qu'elles sont plus longues. L'efficacité des deux autres formats dépend de la répartition des points dans l'espace. Le format gzip permettra une meilleure compression dans les tiles qui comportent beaucoup de points, ce qui explique son efficacité sur le jeu de données GPS, permettant de réduire l'espace disque utilisé à 100Go. Nous pensons que c'est ce format qui est le mieux adapté en général. La différence de taille avec le format binaire est relativement faible par rapport au gain apporté par un format texte qui permet une meilleure interopérabilité avec le reste du système.

7.4 Calcul en flux

Le calcul en batch est certes efficace, mais possède un inconvénient inhérent au mode de calcul : la latence. Même si le calcul peut être accéléré en augmentant les ressources allouées, il faut tout de même attendre potentiellement plusieurs heures avant de pouvoir accéder au résultat. La lambda architecture introduite par Marz & Warren [77] et présentée au chapitre 2 propose de combiner le calcul en batch à du calcul en flux pour améliorer la latence du système. Dans cette section, nous présentons une méthode de calcul en flux qui peut être utilisée en complément de notre approche batch au sein d'une lambda architecture.

7.4.1 Principe

L'objectif du calcul en flux est de minimiser le temps qui s'écoule entre la réception d'un nouveau point de données par le système et le moment où ce point est disponible pour les requêtes du client. Dans notre cas, nous voulons maintenir une pyramide de tiles qui décrit les points reçus depuis le lancement du système.

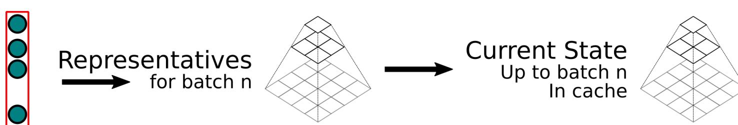


FIGURE 7.6 – Procédure de traitement d'un micro-batch. Deux étapes d'agrégation successives sont effectuées pour obtenir une pyramide de tiles décrivant les points reçus depuis le lancement du système.

Le calcul des niveaux de détail de manière récursive prend trop de temps pour être utilisé en flux. Les différends niveaux doivent donc être calculés en parallèle. Cela permet d'augmenter la parallélisation des calculs et donc de diminuer la latence totale. Il n'existe par contre plus de relation hiérarchique entre les représentants de niveaux successifs. De même, la résolution des conflits demande beaucoup de calculs et ajoute des étapes qui augmentent encore la latence. En augmentant la surface totale considérée dans chaque partition, il est possible de grandement diminuer le besoin de résoudre des conflits.

La visualisation n'est pas un domaine où la latence est critique au point de devoir atteindre les quelques millisecondes. Le résultat final étant destiné à une analyse par un humain, des latences de quelques secondes (au plus une dizaine) sont tout à fait acceptables. C'est pourquoi nous avons choisi d'utiliser le modèle de calcul en flux dit "micro-batch" qui permet de traiter plus de données au prix de latences légèrement plus grandes que le modèle "one-at-a-time".

Dans le modèle "micro-batch", les données reçues dans chaque intervalle sont regroupées et traitées ensemble. Le traitement d'un "micro-batch" se décompose en deux étapes : le calcul d'une pyramide décrivant les données du micro-batch courant et l'intégration de cette pyramide à celle décrivant l'état depuis le lancement du système.

Calcul de la pyramide pour le "micro-batch" : Chaque point entrant est dupliqué pour chaque niveau de détail, de manière à pouvoir traiter chaque niveau en parallèle. Ensuite, les points sont groupés par tile et un canopy clustering séquentiel est appliqué sur chacun. Chaque tile peut donc être traité séparément. Comme le niveau 0 ne possède qu'un seul tile, tous les points reçus seront regroupés au sein de ce tile et devront être à la fois stockés et traités séquentiellement sur la même machine. Le résultat de cette étape est une pyramide de tile qui décrit les données reçues durant ce "micro-batch".

Fusion des pyramides : Une fois la pyramide pour le "micro-batch" courant calculée, elle peut être fusionnée avec la pyramide globale. Pour cela, chaque tile est comparé avec son équivalent dans la pyramide globale. Un nouveau canopy clustering séquentiel est effectué en donnant la priorité aux représentants de la pyramide globale. Le résultat est conservé pour être réutilisé lors du prochain "micro-batch".

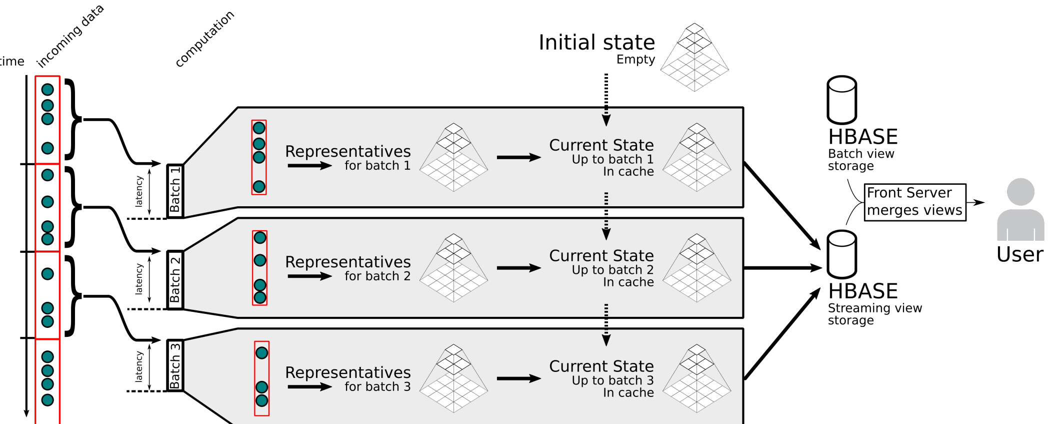


FIGURE 7.7 – Processus complet pour le calcul en flux. Lors de chaque micro-batch, des représentants sont choisis puis fusionnés avec l'état courant. Chaque tile est ensuite inséré dans HBase pour être requété par la visualisation.

7.4.2 Implémentation

L'implémentation de notre solution de calcul en flux a été réalisée avec la bibliothèque Spark Streaming, qui utilise le modèle de "micro-batch". Chaque micro-batch est ensuite traité comme un RDD, la structure de donnée normalement utilisée par Spark, ce qui permet de réutiliser le code déjà développé.

L'état courant de la pyramide est conservé dans le cache de Spark. Les nouveaux points sont fusionnés en utilisant l'opération *mapWithState*, qui prend en charge la gestion de l'état. Seuls les tiles non vides sont stockés et un calcul n'est lancé pour un tile que lorsque de nouveaux points doivent lui être ajoutés. Pour rendre cet état accessible depuis l'extérieur du système, les tiles sont insérés dans HBase. Lorsqu'un tile est modifié, il écrase alors la version précédente.

Le cache peut être configuré pour utiliser la mémoire vive, le disque ou les deux. La mémoire vive est évidemment la solution la plus rapide, mais peut être trop limitée si la taille de la pyramide devient trop importante. Dans le cadre de la lambda architecture, l'état généré par les calculs en flux n'a pas vocation à être conservé indéfiniment. Il sera réinitialisé une fois que les nouvelles données auront été prise en charge par un calcul en batch.

Si il n'est pas possible de conserver l'état courant dans le cache de Spark, HBase peut aussi remplir ce rôle. A chaque modification d'un tile, il faut alors envoyer une requête pour récupérer l'état courant, effectuer la modification et l'insérer à nouveau dans HBase. Cette solution aura une latence plus importante puisque les données devront être récupérées depuis une source externe.

Une propriété cruciale d'un système de gestion de données en flux est la stabilité. C'est à dire la capacité du système à pouvoir traiter les données plus rapidement qu'elles n'arrivent. Dans le cas du micro-batch, cela consiste à maintenir le temps de

traitement moyen en dessous de la taille de l'intervalle de micro-batch. Dans ce cas, le système sera prêt à traiter le micro-batch suivant dès son lancement. Dans le cas contraire, du retard s'accumulera.

Pour tester la stabilité de notre approche, nous avons réalisé plusieurs tests de montée en charge, dont les résultats sont visibles dans la Table 7.3. Ces tests ont été réalisés en simulant la réception de points à un rythme prédéterminé, avec de petites fluctuations aléatoires. Les données sont issues du jeu de données brightkite présenté précédemment. On peut voir que même avec un petit intervalle de micro-batch et un taux de réception des données élevé, le système reste stable. Au taux de 10000 points reçus par secondes, il faut moins de 8 minutes pour collecter la totalité des données brightkite, alors qu'elles concernent 2 ans et demi d'activité. Le vitesse de calcul est donc plus rapide que la vitesse de collecte, ce qui indique que notre algorithme peut être utilisé dans le cadre d'un système à volumétrie élevée. De plus, il est possible de gérer plus de données à la fois en adaptant la taille du cluster utilisé.

Intervalle	Taux de réception	Temps de traitement moyen
5s	5000 p/s	2.534 s
2s	5000 p/s	1.485 s
2s	10000 p/s	1.426 s

TABLE 7.3 – Résultats des tests de stabilité de notre système de calcul en flux. Dans les trois cas, le temps de traitement est inférieur à l'intervalle de micro-batch, donc le système est stable.

7.5 Visualisation

La dernière étape du système est de dessiner la carte de chaleur. Cette étape se déroule sur le client de visualisation. Les données à visualiser ont été réduites par l'agrégation, mais le rendu final peut encore prendre trop de temps suivant la puissance de calcul disponible. C'est pourquoi nous proposons un algorithme qui permet de borner le temps de rendu de la carte de chaleur. Ensuite, nous montrons quel est l'impact des approximations introduites tout au long du traitement par rapport à une carte de chaleur utilisant la totalité des données.

7.5.1 Rendu de carte de chaleur en temps borné

Le principe de notre algorithme est de maintenir le nombre d'opérations nécessaires au calcul de la carte de chaleur en dessous d'un certain seuil. Ce seuil peut être modulé en fonction de la puissance du GPU utilisé. La complexité d'un calcul de KDE est $O(np)$, avec n le nombre de points et p le nombre de pixels. L'agrégation des données a déjà réduit n. Pour maintenir le nombre d'opérations quand n devient trop important, il faut donc agir sur p. Lorsque le nombre de points augmente, la

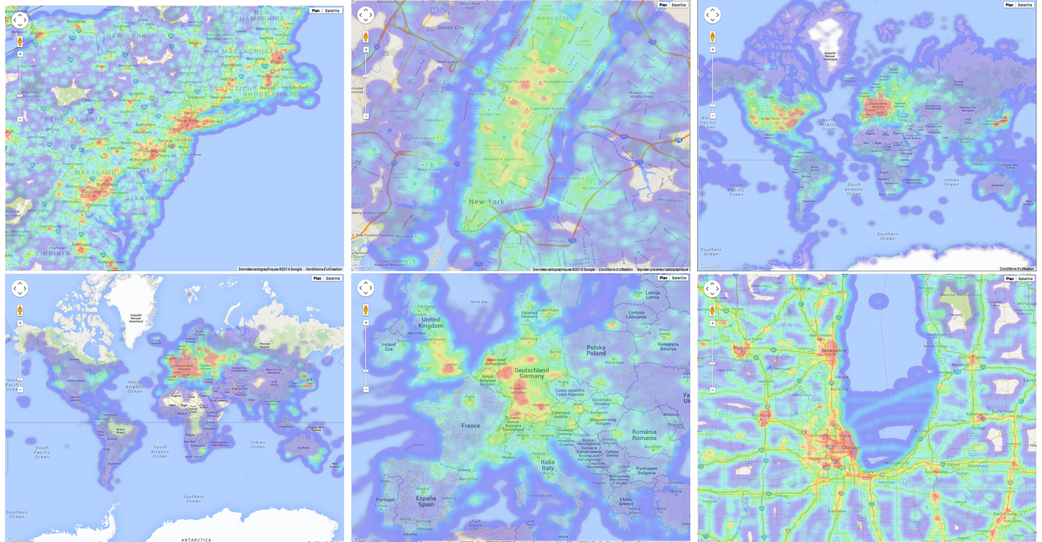


FIGURE 7.8 – Vues des quatre jeux de données utilisés, avec le nombre de points affichés. Sur la rangée supérieure, de gauche à droite : brightkite (1671), gowalla (1334) et OpenStreetMap (2873). Sur la rangée inférieure, le jeu de données GPS, de gauche à droite : vue globale (3185), Europe (3218) et Chicago (1468).

résolution sur laquelle le KDE est calculé diminue. On obtient alors une carte de chaleur sur une résolution inférieure qui peut ensuite être agrandie pour retrouver la résolution d'origine. Le principe sous-jacant est un compromis entre temps de calcul et précision de l'image obtenue.

7.5.1.1 Algorithme

Nous définissons un seuil N sur le nombre de points qui peuvent être rendus à pleine résolution sur p pixels en conservant un temps de rendu satisfaisant. Ce temps peut être exprimé par $O(pN)$. Lorsque plus de points doivent être rendus, la résolution est abaissée pour conserver un nombre d'opérations de $O(pN)$.

Soit $m = \left\lceil \frac{n}{N} \right\rceil$, avec n le nombre de points à rendre. Ce nombre mesure la différence de magnitude entre les données à rendre et la quantité maximale possible à pleine résolution. On peut définir w et h comme étant les dimensions de la zone de rendu. On a donc $p = wh$. Soit p' la nouvelle résolution. Pour conserver le ratio d'aspect du dessin, on a :

$$p' = \frac{p}{m} = w'h'$$

$$w' = \frac{w}{\sqrt{m}} \qquad h' = \frac{h}{\sqrt{m}}$$

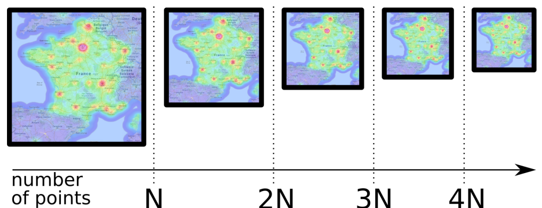


FIGURE 7.9 – Illustration de la baisse de résolution de la carte de chaleur lorsque le nombre de points augmente.

Pour chaque ensemble de N points supplémentaires, la résolution diminue pour conserver le même temps de rendu.

A cette résolution, le coût de rendu de n points est $p'n = \frac{pn}{\lceil n/N \rceil}$. On peut facilement montrer que $p'n \leq pN$, donc la complexité du rendu sur cette résolution réduite est $O(pN)$. Pour une machine donnée, ces deux valeurs peuvent être fixées et considérées comme constantes. Le temps total de rendu pour une machine spécifique est donc constant, mais au détriment de la résolution de l'image finale.

7.5.1.2 Implémentation

Nous avons implémenté cet algorithme en utilisant WebGL pour permettre un rendu exploitant le processeur graphique. Dans cette implémentation, le seuil N est défini par la constante `MAX_FRAGMENT_UNIFORM_VECTORS` qui est dépendante du matériel. Cela implique qu'il n'est pas possible de traiter plus de N points à la fois. Pour pallier cette limitation, nous avons employé une technique inspirée de MapReduce, dont les principes s'appliquent aussi au calcul parallèle.

Les points de données sont séparés en m partitions, comprenant chacune au plus N points. Un premier rendu est effectué pour chaque partition sur p' pixels. On obtient alors m KDE partiels. Cette partie s'apparente à la phase de `map`. Un deuxième rendu calcule ensuite la somme des m KDE partiels pour obtenir le KDE total sur p' pixels. C'est la phase de `reduce`.

La dernière partie consiste à agrandir l'image obtenue sur p pixels. Il existe de nombreuses techniques permettant d'agrandir des images en limitant la perte de qualité. Nous utilisons le filtre de magnification `GL_LINEAR` intégré à WebGL. Ce filtre prend comme valeur du pixel à l'écran une moyenne pondérée des valeurs des pixels proches dans l'image à agrandir.

De plus, nous ne voulons utiliser qu'une seule image de p pixels pour tous les rendus. Or, même si le nombre de pixels correspond, m images de p/m pixels ne peuvent être contenues dans p pixels que si m est un carré parfait. Dans notre implémentation, les changements de résolution se font donc uniquement avec de telles valeurs de m.

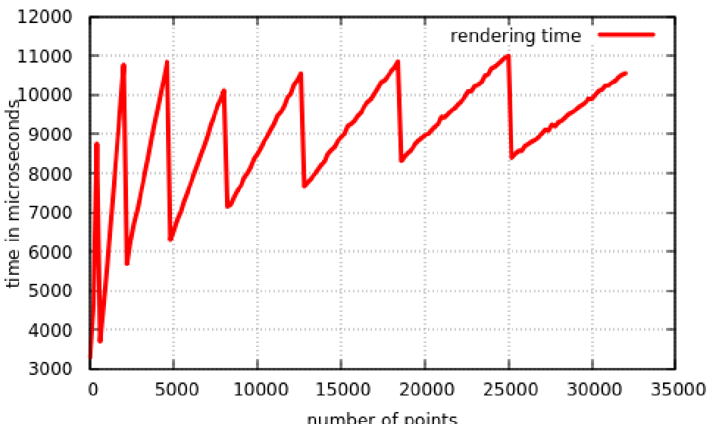


FIGURE 7.10 – Temps de rendu d'une carte de chaleur, sur un ensemble de points aléatoires. Le GPU utilisé est un Nvidia GTX 970M.

Intégration à Fatum Cet algorithme a été implémenté au sein de la bibliothèque Fatum présentée au chapitre 6. Pour ce faire, nous avons ajouté une propriété `weight` aux *marks* pour représenter le poids associé. La carte de chaleur peut ensuite être activée ou désactivée par l'utilisateur.

7.5.1.3 Performances

Le temps de rendu de notre algorithme est borné, mais cette borne dépend du matériel utilisé. Nous avons testé le temps de rendu effectif sur un ordinateur équipé d'un GPU Nvidia GTX 970M. Les résultats sont visibles sur la Figure 7.10. On peut remarquer que le temps de rendu se situe toujours en dessous de 11ms. Les motifs en dents de scie sont dus aux changements de résolution. Le temps de rendu augmente linéairement à cause de l'augmentation du nombre de points, puis redescend brutalement lors du changement de résolution. Les pics sont de plus en plus espacés, puisque nous n'utilisons que les valeurs de m qui sont des carrés parfaits.

7.5.2 Qualité de la visualisation

Tout au long du processus de traitement des données décrit dans ce chapitre, des approximations ont été introduites. D'abord, l'agrégation qui regroupe les points pour produire une abstraction multi-échelle. Puis les baisses de résolution provoquées par l'algorithme de rendu. Nous allons maintenant mesurer l'impact global de ces approximations. Pour ce faire, nous comparons les images produites par notre système avec celles qui auraient été produites avec les données brutes et un algorithme de rendu exact. Les comparaisons ont été faites sur les niveaux de détail du jeu de données *brightkite*, puisqu'il est suffisamment petit pour permettre des calculs séquentiels.

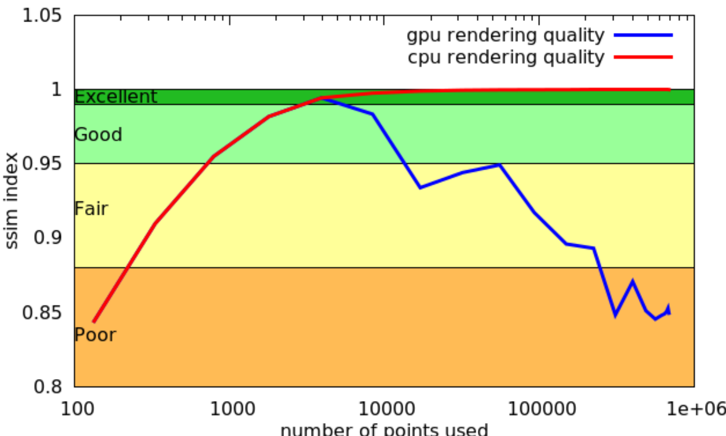


FIGURE 7.11 – Comparaison des rendus CPU et GPU des niveaux de détail du jeu de données brightkite avec la carte de chaleur exacte utilisant tout le jeu de données.

Nous cherchons ici à quantifier la différence perçue entre le rendu final de notre système et ce qu'on aurait pu visualiser en dessinant directement la carte de chaleur à partir du jeu de données complet. Pour obtenir une comparaison d'images objective, nous avons utilisé l'indice SSIM [111]. Cet indice reflète bien les propriétés de la perception humaine. Le résultat de la comparaison de deux images est un score de similarité compris entre 0 et 1. Il est difficile d'interpréter ce score pour savoir si la différence perçue est négligeable ou non. Pour cela, nous utilisons les classes définies par Zinner et al. [120] pour mieux appréhender la signification du score SSIM.

Nous avons comparé une image exacte du jeu de données complet aux différents niveaux de détail, rendu à la fois avec un algorithme de rendu exact au CPU et notre algorithme au GPU. La Figure 7.11 montre l'indice SSIM de ces comparaisons en fonction du nombre de points. On remarque que lorsque le nombre de points est petit, les deux méthodes ont un score identique, qui augmente avec le nombre de points utilisés. Lorsque la baisse de résolution devient trop importante, le score de notre rendu GPU diminue alors que celui du rendu CPU se rapproche de plus en plus du jeu de données entier. Idéalement, les images générées par le système doivent se situer vers le sommet de la courbe. Par exemple, une visualisation de $1000px \times 1000px$ contient 16 tiles de $256px \times 256px$. Avec $c_0 = 1000$, cela fait donc au maximum 16000 points à l'écran quel que soit le niveau de détail, ce qui correspond bien à la partie supérieure de la courbe.

La Table 7.4 permet de comparer les images produites à différents niveaux de détail par les deux méthodes, ainsi que les temps de rendu. On peut mesurer l'apport de notre système en comparant l'image encadrée en rouge en bas à gauche, qui utilise tout le jeu de données, avec celle encadrée en vert en haut à droite qui représente une carte de chaleur très similaire avec beaucoup moins de points. La différence entre les deux images est à peine visible, alors que les temps de rendu de notre système

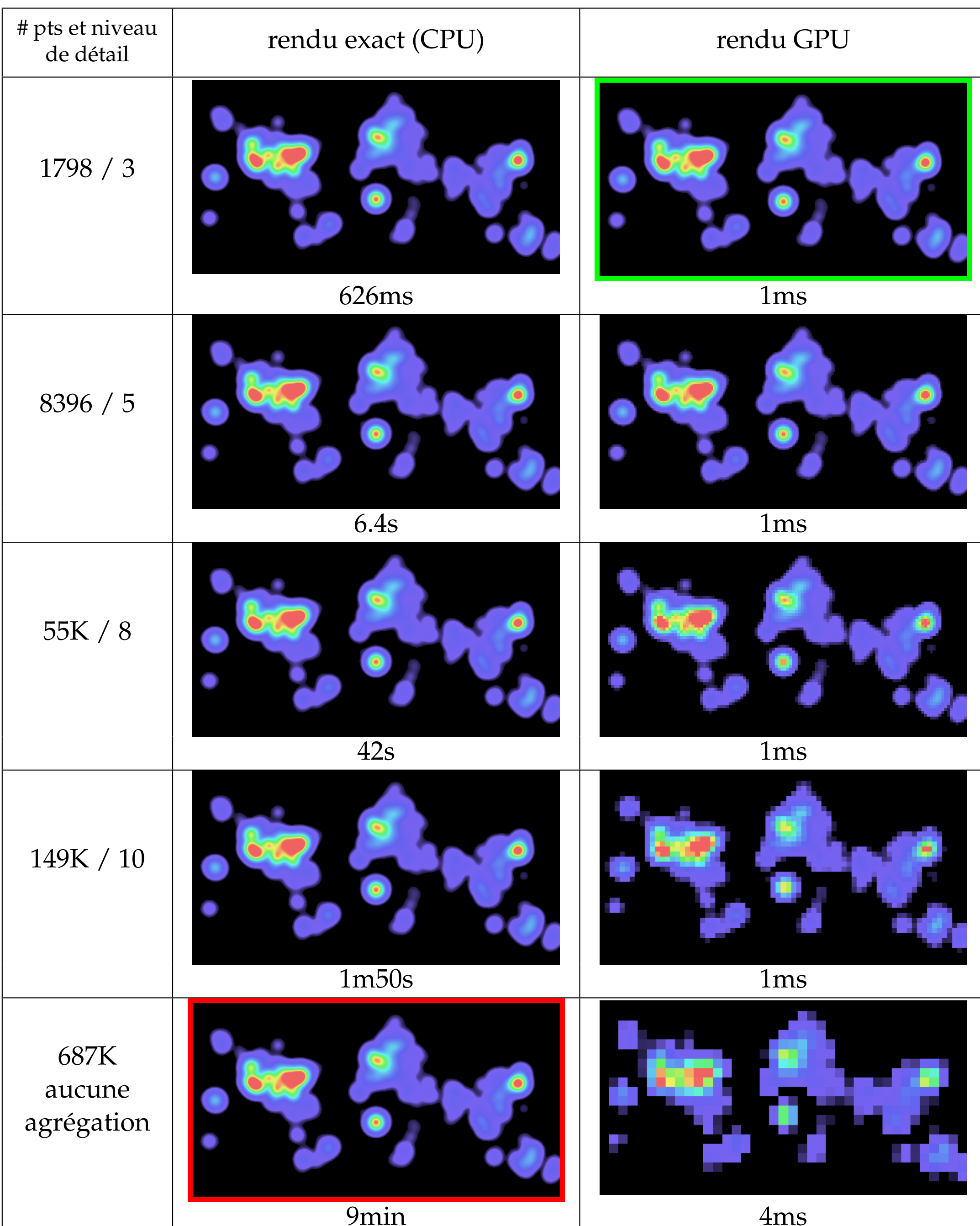
sont beaucoup plus courts. Notre système permet donc d'explorer interactivement une carte de chaleur à très grande échelle en produisant une visualisation identique à celle utilisant tout le jeu de données.

7.6 Conclusion

Dans ce chapitre, nous avons montré une application de notre méthode de visualisation de données massives à la visualisation de cartes de chaleur. Le principe d'approximation de Kernel Density Estimation permet de réduire la quantité de données utilisée pour la carte de chaleur tout en garantissant que la différence par rapport au jeu de données original soit petite. En utilisant l'approche par partitionnement de l'espace pour implémenter le canopy clustering, il est possible d'effectuer le précalcul d'une carte de chaleur de plusieurs milliards de points en moins d'une heure.

Nous avons aussi présenté un algorithme de rendu de carte de chaleur en temps constant. En diminuant la résolution du calcul de la visualisation lorsque les données deviennent trop importantes, le temps total est borné.

Nous avons montré que la combinaison de ces deux parties permet d'obtenir un système de visualisation de carte de chaleur à grande échelle. Les images qu'il produit sont très proches de la carte de chaleur du jeu de données complet, mais il permet en plus d'explorer interactivement les données grâce à la génération de plusieurs niveaux de détail.



# pts et niveau de détail	rendu exact (CPU)	rendu GPU
1798 / 3	626ms	1ms
8396 / 5	6.4s	1ms
55K / 8	42s	1ms
149K / 10	1m50s	1ms
687K aucune agrégation	9min	4ms

TABLE 7.4 – Exemples d'images obtenues avec chaque méthode de rendu. En dessous se trouve le temps de rendu. L'image en bas à gauche (encadrée en rouge) correspond à tout le jeu de données et est donc le point de comparaison global. L'image en haut à droite (encadrée en vert) est le résultat de notre système. La différence est à peine visible, alors que le temps de rendu est bien inférieur.

8 Cornac : Visualisation de graphes à grande échelle

Dans ce chapitre, nous présentons une deuxième application dédiée à la visualisation de grands graphes. L'agrégation des données concerne à la fois les sommets et les arêtes du graphe. La visualisation finale utilise un client web implémenté à l'aide de la bibliothèque Fatum présentée au chapitre 6.

8.1 Introduction

La révolution du Big Data a entraîné la mise à disposition de jeux de données plus grands et plus variés. Des données provenant de plusieurs sources peuvent être mises en relation. Sans schéma commun pour croiser les données, la structure de graphe est la solution la plus courante. Elle permet d'exprimer n'importe quels ensembles d'entités et les relations qui les unissent.

On retrouve dans la visualisation de graphes les deux obstacles à la visualisation de données massives : la scalabilité de perception et la scalabilité des performances. L'occlusion apparaît d'abord entre les noeuds du graphe. Les solutions qui jouent sur les variables visuelles sont rapidement limitées, car elles réduisent les informations qui peuvent être encodées sur chaque noeud. Ces informations sont pourtant essentielles pour permettre de relier la structure de graphe au jeu de données initial. Une solution couramment utilisée peut être de changer la représentation du graphe. Comme nous l'avons déjà vu, la carte de chaleur est une représentation particulièrement efficace pour contrer l'occlusion. Van Liere et de Leeuw [109] ont introduit l'idée de représenter un graphe sous forme de champ continu pour repérer les plus fortes concentrations de noeuds. Cette idée a par la suite été reprise par Lampe et Hauser [70] pour visualiser aussi les arêtes du graphe. Cependant, le mélange des champs dédiés aux noeuds et aux arêtes peut prêter à confusion et gêner l'analyse visuelle. Pour contrer ce problème, Zinsmaier et al. [121] utilisent la carte de chaleur uniquement pour représenter les noeuds et déplacent les extrémités des arêtes aux maximums locaux de la fonction de densité. Cela permet aussi de

fortement diminuer l'occlusion en regroupant des arêtes. L'agrégation géométrique utilisée dans notre système produit un effet similaire tout en bornant le déplacement maximal subit par les arêtes.

8.2 Vue d'ensemble

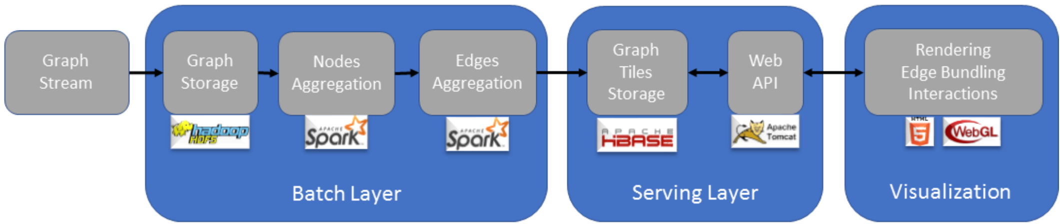


FIGURE 8.1 – Pipeline de traitement des données graphe.

L'objectif de notre système est de permettre la visualisation interactive de grands graphes en appliquant la méthode décrite au chapitre 4. Nous considérons que le dessin du graphe à visualiser est donné en entrée. L'étape de dessin de grands graphes n'est donc pas traitée ici. La séparation du dessin et de la visualisation permet d'avoir un système plus flexible qui permet de changer l'algorithme de dessin en fonction des besoins. Le fonctionnement du système se décompose donc en deux étapes : l'agrégation du graphe pour produire les différents niveaux de détail et la visualisation du graphe sur client léger. La Figure 8.1 présente les différentes étapes du traitement.

L'étape d'agrégation doit gérer l'agrégation des deux ensembles qui constituent le graphe : les noeuds et les arêtes. L'agrégation des noeuds utilise l'algorithme de canopy clustering dont plusieurs implémentations ont été présentées au chapitre 5. Les paramètres de l'agrégation c_0, le nombre maximum de représentants au niveau de détail 0, et r, le rapport entre les distances d'agrégation de deux niveaux successifs, sont déterminés comme dans le chapitre 7, nous conservons donc les valeurs $c_0 = 1000$ et $r = 2$. On voit ici l'intérêt de fixer le nombre de représentant c_0 plutôt que la distance d'agrégation d, puisque d dépend de la taille de la boite englobante du graphe considéré. Le fait de fixer c_0 permet de paramétrer l'agrégation indépendamment du graphe à agréger. La distance d'agrégation d est calculé spécicifiquement pour chaque graphe à partir de c_0.

L'agrégation utilisée est exclusivement géométrique, c'est-à-dire que la structure du graphe, donc les arêtes, n'est pas prise en compte pour l'agrégation des noeuds. Les noeuds proches dans le dessin seront regroupés. Cependant, on peut considérer qu'un bon dessin de graphe fait en sorte que les noeuds proches dans le graphe soient proches dans le dessin. De plus, une fois le dessin calculé, les noeuds proches créent de l'occlusion, qu'ils soient proches dans le graphe ou non. L'agrégation géométrique permet donc de réduire l'occlusion indépendamment des critères utilisés pour le dessin.

L'agrégation des noeuds permet de diminuer le nombre de noeuds du graphe, mais n'a pas d'effet sur les arêtes. La fusion de sommets créé des arêtes dont les deux extrémités correspondent au même sommet, c'est-à-dire des boucles, ainsi que des groupes d'arêtes reliant les deux mêmes sommets, appelées arêtes multiples. La première étape d'agrégation des arêtes consiste à supprimer les boucles et fusionner les arêtes multiples. Cela permet de ramener le nombre d'arêtes potentiel du graphe de $|V|^2$ à $|V'|^2$, avec V l'ensemble des sommets avant l'agrégation et V' l'ensemble des sommets agrégés. Cette borne est toujours quadratique et peut donc représenter un nombre considérable d'arêtes à stocker, transférer et visualiser, ce qui nuit aux performances globales du système et à l'interactivité de la visualisation.

Afin de réduire d'avantage le nombre potentiel d'arêtes, nous ajoutons une seconde étape d'agrégation, qui ne s'applique que sur les arêtes considérées comme "longues". En partant du principe que ces arêtes apportent visuellement moins d'information que les arêtes courtes, il est possible d'en supprimer sans perturber la navigation dans le graphe. Cette seconde étape nous permet de réduire le nombre maximal d'arêtes du graphe agregé à $|V'| \times (c_0 + S)$, S étant le nombre maximal d'arêtes longues conservées pour chaque sommet du graphe agrégé.

Les données agrégées sont ensuite indexées en utilisant le système de tiles décrit au chapitre 4. Pour permettre de déterminer la taille et l'origine de l'espace des tiles, la boite englobante du graphe est calculée au début de l'étape d'agrégation et est stockée avec les données. Chaque tile est décomposé en deux parties : les noeuds et les arêtes, qui peuvent être requêtées indépendamment.

La visualisation du graphe s'effectue sur client léger en utilisant la bibliothèque Fatum, présentée au chapitre 6. Les données obtenues sous forme de tile permettent de recréer le graphe sous forme de marks et connections. L'utilisation du pipeline orienté abstraction de données permet de déporter une partie des calculs sur le client de visualisation. Grâce au performances de rendu apportées par Fatum, il est possible d'ajouter un algorithme de faisceautage d'arêtes en temps réel et en espace écran pour réduire d'avantage l'occlusion générée par les arêtes, comme le détaille la section 8.4.4.

8.3 Agrégation

L'agrégation de chaque niveau de détail se décompose en deux étapes : l'agrégation des noeuds puis l'agrégation des arêtes.

8.3.1 Agrégation de noeuds

Lors de la visualisation du graphe, les noeuds peuvent être affichés individuellement. La distribution spatiale des représentants choisis par le canopy clustering aura donc un impact sur l'aspect de la visualisation finale. Nous allons donc comparer les

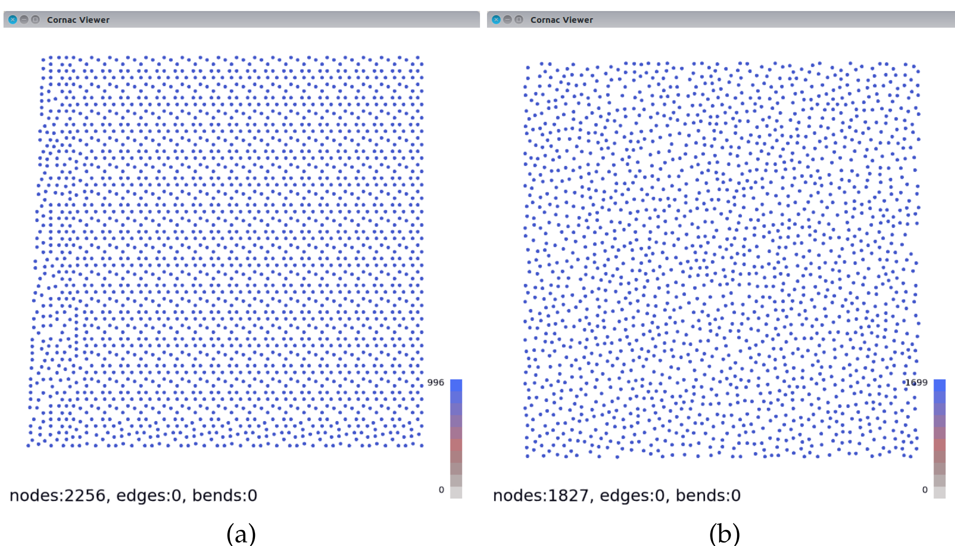


FIGURE 8.2 – Résultat de l'agrégation d'une grille de $10^5 \times 10^5$ points avec (a) la technique par partitionnement de l'espace et (b) la technique par ensemble indépendant maximal.

distributions générées par l'approche par partitionnement de l'espace et l'approche par ensemble indépendant maximal, présentées au chapitre 5.

La Figure 8.2 montre le résultat de l'agrégation d'une grille de $10^5 \times 10^5$ points par les deux techniques. Une telle densité de points permet de tester les distributions engendrées par les deux techniques lorsque l'espace est saturé et permet de mettre en exergue les limites de chaque algorithme. Sur la Figure 8.2a, les points ont été agrégés avec l'approche par partitionnement de l'espace. On remarque que la distribution des points est régulière. Les limites des bandes utilisées pour le partitionnement deviennent visibles. Ces motifs ne sont pas gênants lorsque les points ne sont pas représentés, comme pour la visualisation de carte de chaleur présentée au chapitre 7, mais donnent une impression trompeuse de structure au sein du dessin du graphe.

Au contraire, l'agrégation par ensemble indépendant maximal (Figure 8.2b) produit une distribution des représentants beaucoup plus uniforme (ce qui est attendu dans le cas d'une grille). Cette distribution rappelle celle obtenue par l'échantillonnage de Poisson [22]. Ce processus est utilisé lorsqu'on désire obtenir une distribution uniforme avec un aspect "naturel". Cette distribution est préférable à la précédente lorsque les représentants doivent être affichés. C'est donc l'approche par ensemble indépendant maximal que nous utiliserons pour l'agrégation des noeuds, même si elle s'avère plus lente que l'approche par partitionnement de l'espace.

Lors de l'agrégation des noeuds, deux graphes cohabitent : le graphe d'origine $G = (V, E)$ que l'on cherche à agréger et le graphe de disque $G_d = (V, E_d)$ qui est utilisé pour le canopy clustering. Les deux graphes partagent le même ensemble de sommets, qui est l'ensemble de points à agréger, mais les deux ensembles d'arêtes sont différents. La structure considérée pour le choix de l'ensemble indépendant

maximal est celle de G_d et non celle de G, comme le montre la Figure 8.3. Les arêtes communes aux deux graphes, qui appartiennent donc à $E \cap E_d$, sont les arêtes du graphe G dont la longueur est inférieure à la distance d'agrégation. Deux noeuds peuvent donc être agrégés même s'ils ne sont pas connectés dans G.

8.3.2 Agrégation d'arêtes

Une fois les sommets agrégés, il est possible d'agréger les arêtes du graphe. Cette étape utilise deux méthodes complémentaires. La première consiste à regrouper les arêtes en suivant la fusion des sommets réalisée par l'agrégation. La seconde applique une agrégation plus forte aux arêtes considérées comme longues.

8.3.2.1 Agrégation d'arêtes par fusion de sommets

Après l'agrégation des sommets, on peut définir une fonction qui associe à un sommet son représentant :

$$\begin{aligned} r : & V \to V_r \\ & s \mapsto r(s) \end{aligned}$$

L'ensemble $V_r \subset V$ est l'ensemble des représentants. On a la propriété suivante : $s \in V_r \Leftrightarrow r(s) = s$, puisque seuls les représentants sont leur propre représentant.

L'agrégation des arêtes suite à la fusion des sommets revient à appliquer la fonction r à chaque extrémité. Une arête (u, v) devient alors $(r(u), r(v))$. Intuitivement, cela revient à déplacer chaque extrémité de l'arête vers le représentant correspondant. Cette transformation peut créer de nouvelles arêtes et ainsi de nouveaux chemins dans le graphe. Cependant, ces chemins émanent de l'ambiguïté visuelle du dessin due à l'occlusion et auraient aussi vraisemblablement étés interprétés comme tels sans agrégation. Visuellement, le résultat de cette agrégation revient à déplacer chaque sommet s vers son représentant $r(s)$. La longueur de ce déplacement correspond au maximum à la distance d'agrégation. Par exemple, avec $c_0 = 1000$, nous avons vu au chapitre 7 sur la Figure 7.3 que le déplacement maximal est de 8 pixels.

Après cette transformation des extrémités, des arêtes multiples peuvent apparaître. Elles sont donc regroupées pour ne conserver qu'une seule arête par paire de noeuds. Un poids est ajouté à chaque arête pour conserver le nombre d'arêtes multiples représentées. Certaines arêtes peuvent aussi avoir vu leurs extrémités fusionnée avec le même représentant. Elles deviennent donc des boucles. Ces arêtes renseignent sur le nombre d'arêtes qui reliaient les sommets agrégés au sein de ce représentant.

On obtient un nouveau graphe $G_r = (V_r, E_r)$. Le nombre maximum d'arêtes générées de cette façon ($|E_r|$) est $|V_r|^2$. Même si $|V_r|$ est borné grâce à l'agrégation par canopy clustering, cela peut représenter un nombre considérable d'arêtes qu'il

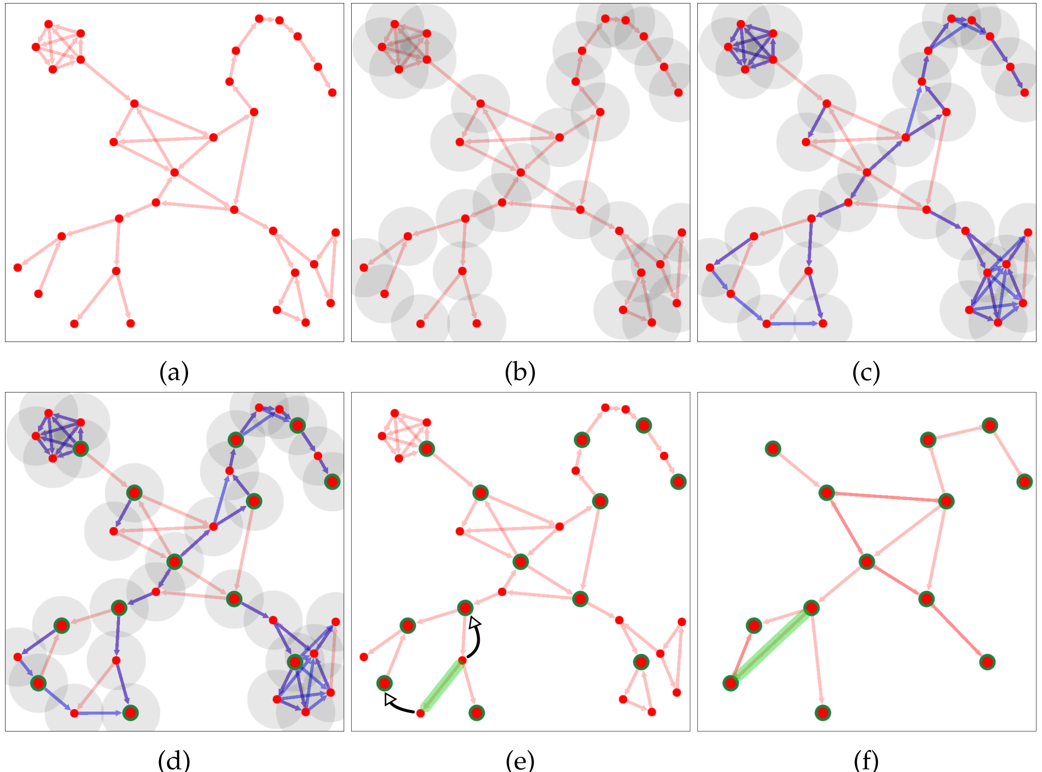


FIGURE 8.3 – Exemple du processus d'agrégation des noeuds en utilisant l'approche par ensemble indépendant maximal. (a) Le graphe original G. (b) Un disque est disposé sur chaque sommet pour construire le graphe de disque G_d. (c) Les graphes G (arêtes rouges) et G_d (arêtes bleues) superposés. Certaines arêtes sont communes, d'autres n'existent que dans l'un des deux graphes. (d) Les représentants sont choisis en considérant uniquement G_d. Ils sont ici aggrandis et entourés de vert. (e) L'ensemble des représentants sur le le graphe G. Chaque sommet sera fusionné avec le représentant le plus proche, qu'ils soient voisins dans G ou non. L'arête surlignée en vert est un cas particulier, puisque ses deux extrémités sont fusionnées avec des représentants différends, qui n'étaient pas connectés. (f) Le graphe final agrégé. L'arête verte a créé une nouvelle connexion entre deux représentants, comme ce qui se produit visuellement à cause de l'occlusion lors d'un dézoom géométrique.

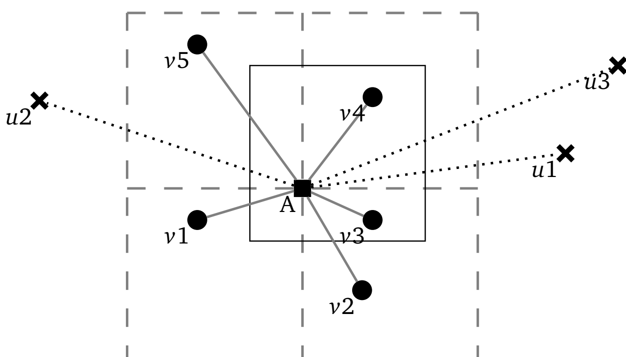


FIGURE 8.4 – Distinction entre les arêtes longues et les arêtes courtes autour d'un sommet A. Les quatre carrés en pointillé représentent les positions extrêmes de la fenêtre de visualisation par rapport à A. Les voisins de A à l'intérieur de ce rectangle (v_1 à v_5) correspondent à des arêtes courtes. Les voisins en dehors (u_1, u_2 et u_3) correspondent à des arêtes longues. On peut remarquer que les extrémités des arêtes courtes peuvent sortir de l'écran, comme le montre le rectangle noir qui modélise une position possible de la fenêtre de visualisation autour de A.

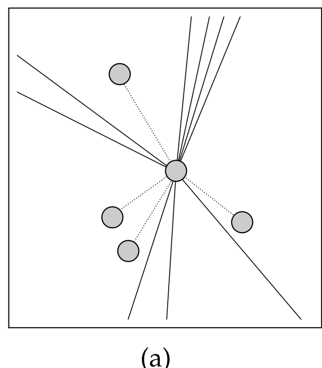
faut pouvoir stocker d'une part, transférer et dessiner à l'écran d'autre part. Le nombre d'arêtes possibles influence donc les performances globales du système. Nous verrons comment réduire ce nombre par une seconde étape d'agrégation pour les arêtes longues dans la section 8.3.2.2 et par un filtrage dans la section 8.4.1.

Chaque arête agrégée d'un niveau de détail i correspond à une ou plusieurs arêtes du niveau de détail inférieur $i + 1$. Nous associons à chaque arête agrégé du niveau i un poids qui correspond à la somme des poids des arêtes correspondantes au niveau $i + 1$. Les arêtes du graphe original ont un poids de 1, donc le poids de chaque arête correspond au nombre d'arêtes du graphe original qu'elle représente. Ce poids sera utilisé dans la visualisation pour différencier l'importance des arêtes agrégées (voir section 8.4).

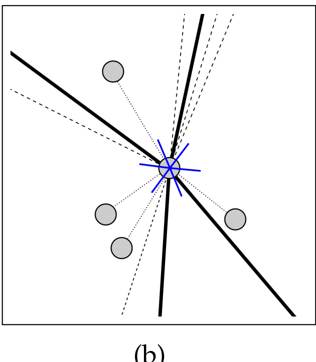
8.3.2.2 Agrégation des arêtes longues

Pour permettre de faire baisser la borne théorique sur le nombre total d'arêtes, nous allons différencier deux types : les arêtes longues et les arêtes courtes. On peut partitionner l'ensemble E_r en E_c, l'ensemble des arêtes courtes et E_l l'ensemble des arêtes longues.

La séparation entre ces deux types d'arêtes repose sur l'affichage final. Comme les opérations de zoom et de drill-down sont couplées, on peut connaître la surface occupée à l'écran par un niveau de détail grâçe à la taille des tiles qui servent à l'indexation. Comme décrit dans le chapitre 4, un tile occupe $256 \times 256px$. En fixant la taille de la fenêtre de visualisation, on détermine un nombre de tile constant affiché à l'écran indépendamment du niveau de détail, que nous appellerons k. Par



(a)



(b)

FIGURE 8.5 – Processus d'agrégation des arêtes longues. (a) Le sommet original, avec des arêtes courtes, dont les deux extrémités sont visibles, et des arêtes longues. (b) Les arêtes en gras on été choisies comme représentants pour leur secteur angulaire (en bleu). Les autres sont en pointillés.

exemple, une fenêtre de $1000 \times 1000px$ permet d'afficher 16 tiles.

Les arêtes courtes sont celles dont les deux extrémités peuvent être visibles simultanément, c'est-à-dire distantes de moins que la diagonale de la fenêtre de visualisation. Pour un sommet v, le nombre d'arêtes courtes possibles est le nombre maximum de sommets suffisamment proches, ce qui dépend du nombre de représentants par tile c_0 choisi lors de l'agrégation des sommets. Les arêtes courtes sont les arêtes comprises dans les 4 positions extrêmes de la fenêtre de visualisation par rapport au sommet considéré, comme le montre la Figure 8.4. Pour ce sommet, on a $4k \times c_0$ arêtes courtes au maximum (par exemple $4 * 16 = 64$ pour une fenêtre de $1000 \times 1000px$). Au total dans le graphe, on a donc $|E_c| = |V|4kc_0/2$ arêtes courtes, puisque chaque arête a été comptée deux fois (une fois pour chaque extrémité).

Toutes les arêtes plus longues que ce seuil sont des arêtes longues. On peut remarquer que tant que le graphe peut être affiché en totalité dans la fenêtre choisie, toutes les arêtes sont des arêtes courtes. La séparation entre arêtes courtes et arêtes longues ne concerne donc pas les niveaux d'abstraction les plus élevés.

Comme leurs extrémités ne peuvent pas être affichées simultanément, les arêtes longues n'apportent que peu d'information sur la structure du graphe autre que la direction générale d'un voisin d'un sommet visible. De plus, étant plus longues, elles participent plus à l'occlusion générale puisqu'elles couvrent une plus grande surface. Elles ajoutent aussi de l'occlusion localement autour de leurs extrémités. Il nous semble donc essentiel de réduire le nombre d'arêtes longues.

Pour ce faire, nous définissons S, le nombre maximum d'arêtes longues à conserver par noeud. Pour conserver l'information de direction, l'espace est partagé en S secteurs angulaires autour du sommet considéré. Pour chaque secteur, une seule arête longue est conservée par secteur. Son poids est adapté pour représenter les autres arêtes de son secteur. La Figure 8.5 représente la situation d'un noeud avant et après l'agrégation des arêtes longues. Cette méthode s'apparente au canopy clus-

graphe	sommets	arêtes	#niveaux	brut	HBase	ratio
USA-EAST	3.5M	8.7M	19	258MB	5.4GB	20.9
USA-WEST	6.2M	15.2M	20	462MB	11.3GB	24.4
USA-CENTRAL	14M	34.2M	20	1GB	24GB	24
USA-FULL	23.9M	58.3M	21	1.6GB	36GB	22
OSM-EUROPE	174M	348M	22	11.5GB	202GB	17.5

TABLE 8.1 – Jeux de données utilisés pour mesurer les performances de l'agrégation de graphe. Cette table montre la taille de chaque graphe, le nombre de niveaux calculés avec $c_0 = 1000$, la taille du graphe brut ainsi que la taille totale des données stockées dans HBase à l'issue de l'agrégation.

tering, mais considère une distance angulaire plutôt qu'une distance euclidienne. Aucune nouvelle arête n'est créée, seules certaines sont supprimées.

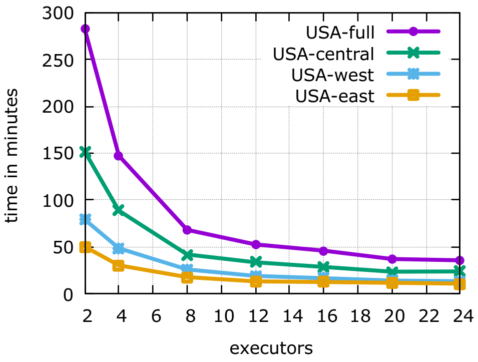
Comme les deux extrémités d'une arête longue ne peuvent être affichées simultanément, cette agrégation peut être appliquée localement autour de chaque noeud. C'est-à-dire qu'on ne considère que des demi-arêtes. Ainsi, une arête peut très bien être conservée à une extrémité et supprimée à l'autre. Le cas le plus courant est qu'une seule extrémité de l'arête ait plusieurs arêtes longues. Le cas le plus représentatif est celui d'une étoile. Si toutes les arêtes de l'étoile sont considérées comme longues, le centre aura S arêtes et chaque sommet sur une branche devra conserver l'arête qui le lie au centre. Les arêtes à afficher devront changer suivant les sommets visibles.

8.3.3 Performances

Pour mesurer les performances de notre système, nous avons choisi d'utiliser des graphes routiers. Ce type de graphe permet de s'affranchir de l'étape de dessin, puisque leurs sommets correspondent à des positions géographiques. Comme le résultat attendu est connu, il est aussi possible de contrôler visuellement la validité de l'agrégation.

Nous avons utilisé cinq graphes : les quatre graphes les plus volumineux issus du 9$^{\text{ème}}$ challenge DIMACS [1], qui représentent le réseau routier des Etats-Unis et un graphe basé sur des données OpenStreetMap, représentant l'ensemble du réseau routier d'Europe [2]. La table 8.1 montre le nombre de sommets et d'arêtes de ces jeux de données, ainsi que le nombre de niveaux calculés avec $c_0 = 1000$ et l'espace utilisé par les données brutes et agrégées. On peut remarquer aussi que le ratio entre données agrégées et données brutes (indiqué dans la dernière colonne) n'augmente pas avec la taille des données. Notre système respecte donc la contrainte de *scalabilité du stockage* énoncé au chapitre 4.

1. http://www.dis.uniroma1.it/challenge9/download.shtml
2. http://i11www.iti.kit.edu/resources/roadgraphs.php



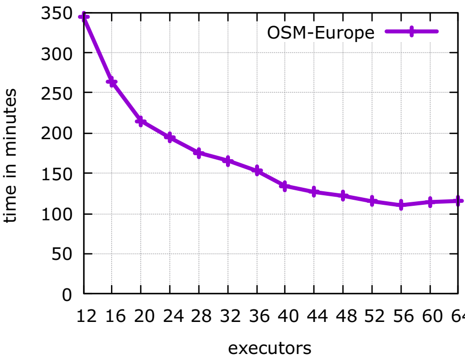


FIGURE 8.6 – Temps de calcul pour l'agrégation (noeuds et arêtes) et l'insertion dans HBase. Pour tous les jeux de données, chaque exécuteur est configuré avec 2 coeurs et 4Go de mémoire vive. La scalabilité de notre approche est clairement visible quel que soit le jeu de données.

Les calculs ont été effectués sur la plateforme du laboratoire. Elle est constituée de 16 machines. Chacune est équipée de deux processeurs à 6 coeurs hyperthreadés cadencés à 2,1 GHz et de 64Go de mémoire vive. Les résultats sont visibles sur la Figure 8.6. On constate une bonne scalabilité horizontale sur tous les jeux de données. Avec suffisamment de ressources, il est même possible d'agréger le graphe de l'Europe en moins de deux heures.

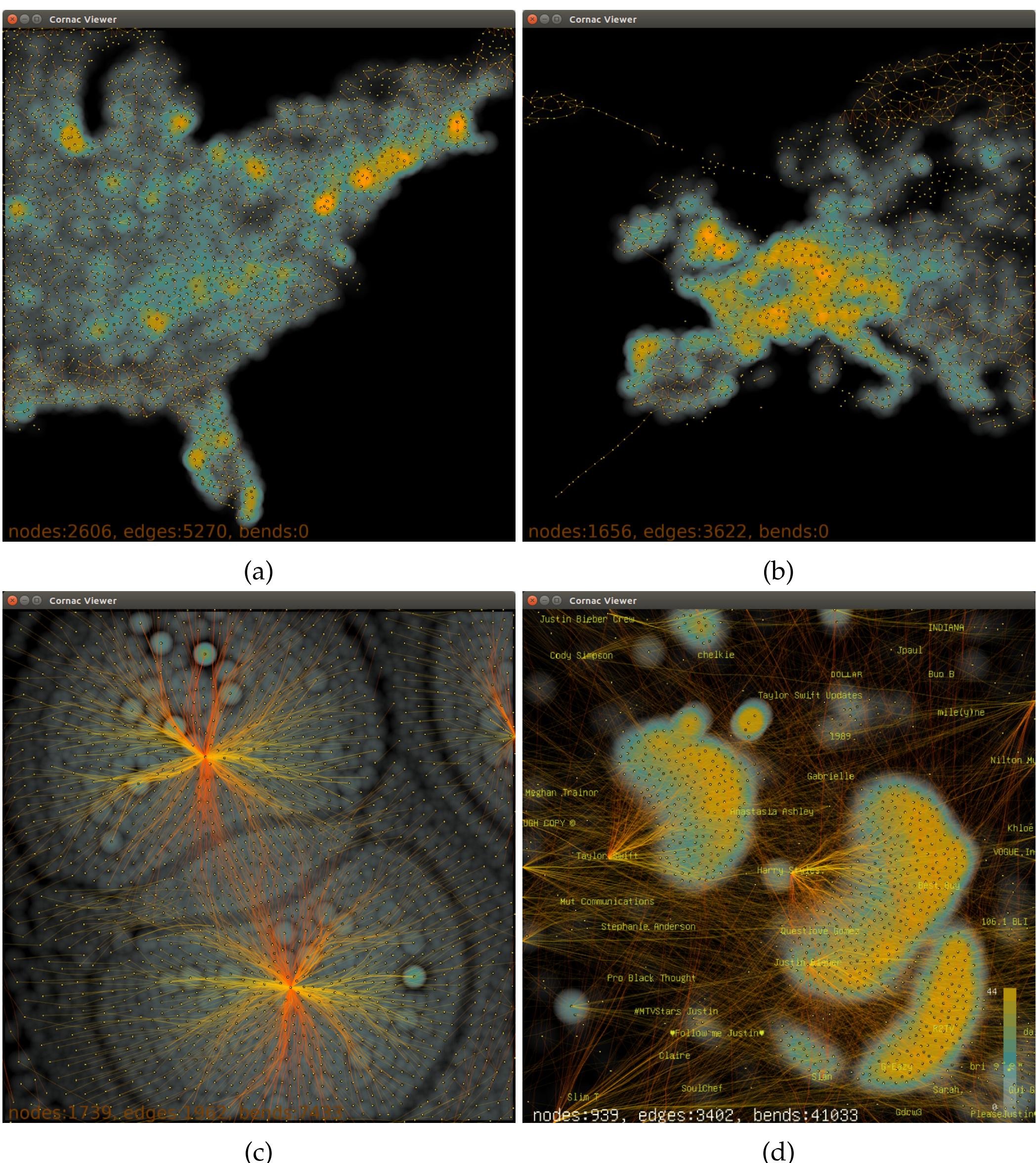
8.4 Visualisation

La dernière partie du système est celle qui permet de visualiser le résultat de l'agrégation. Le client de visualisation utilise la bibliothèque Fatum présentée au chapitre 6 pour l'affichage. La visualisation de graphes sous forme de vue noeud-lien est parfaitement adaptée au paradigme de marks & connections utilisé par Fatum.

8.4.1 Récupération des données

Lors du lancement du client de visualisation, les métadonnées du graphe à visualiser sont récupérées. Elles comportent le nombre de niveaux de détail stockés et la boite englobante du graphe. Grâce à ces informations, le système de tiling peut être initialisé pour se repérer dans la visualisation. Le niveau de détail initial est choisi de manière à ce que l'entièreté du graphe soit visible. La première vue qui est présentée à l'utilisateur est donc une vue d'ensemble du graphe.

A chaque frame, les tiles visibles sont requêtés sur l'Infrastructure Big Data. A la réception, les données sont conservées dans un cache local qui évite de requêter plusieurs fois le même tile. Le cache peut être vidé lorsqu'il atteint une certaine taille, mais cela n'a jamais été nécessaire durant nos expériences. Les tiles de sommets ne pèsent que quelques kilooctets, mais les tiles d'arêtes peuvent contenir un nombre



(a) (b)

(c) (d)

FIGURE 8.7 – Visualisation de grands graphes. (a) Le réseau routier des Etats-Unis. (b) Le réseau routier européen. (c) Détail de la visualisation d'un graphe du réseau internet [29]. Les deux étoiles sont bien visibles grâce au faisceautage d'arêtes et la carte de chaleur permet de repérer au centre un ensemble de noeud communs. (d) Détail d'un réseau de citations d'utilisateurs sur Twitter. Les utilisateurs populaires apparaissent clairement à côté de leur communauté.

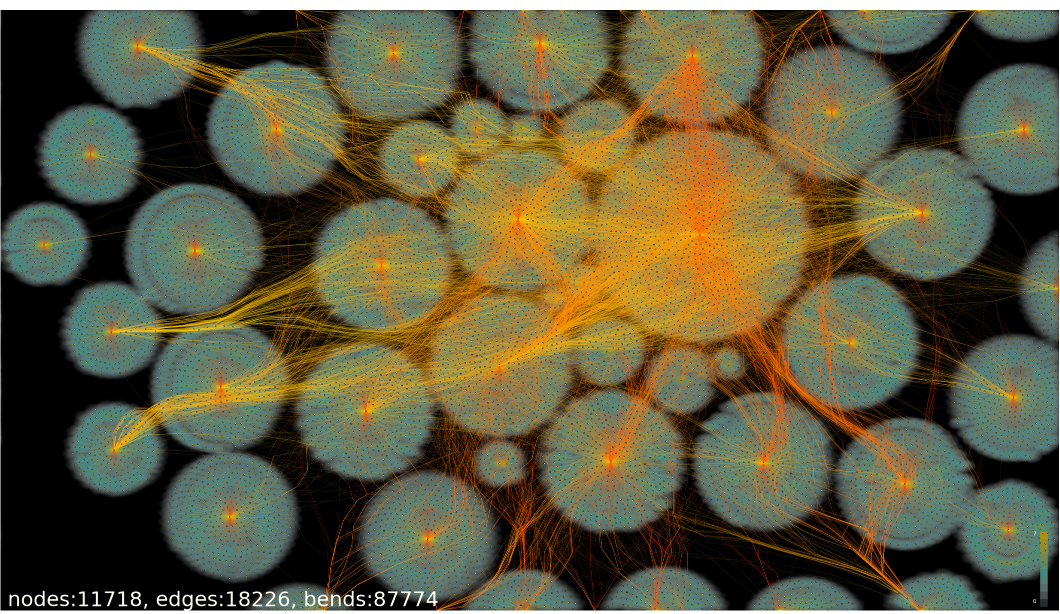


FIGURE 8.8 – Visualisation d'un réseau de machines avec notre système. La carte de chaleur permet de mettre en valeur les groupes créés par le dessin du graphe.

important d'arêtes, ce qui ralenti la réception. Nous appliquons donc un filtrage à la volée via le serveur web utilisé pour le transfert. Dans chaque tile, seules les k arêtes avec le poids le plus important sont transférées. Comme les arêtes sont triées par poids décroissant au sein de chaque tile avant le stockage, il suffit de ne retenir que les k premières. En pratique, nous utilisons $k = 4000$, ce qui signifie que les tiles qui comportent moins de 4000 arêtes sont entièrement lus. Ce filtrage ayant lieu à la réception des données, il est possible d'augmenter le nombre d'arêtes à récupérer ou de le désactiver entièrement. Cela permet alors de ne visualiser que les arêtes les plus importantes dans le graphe, de manière similaire à ce qui est réalisé lors de la génération des niveaux de détail dans GraphMaps [84] ou le regroupement d'arêtes dans LaGO [121].

8.4.2 Rendu des sommets

Chaque noeud est représenté par une mark de forme circulaire. Comme le niveau de détail affiché dépend du niveau de zoom, il y a toujours un espace garanti de quelques pixels entre deux noeuds. Cela permet d'afficher chaque noeud à une taille constante de quelques pixels quelque soit le niveau de détail.

La distribution générée par l'agrégation des noeuds tend vers une distribution uniforme des points dans l'espace. Pour permettre de retrouver l'information de la distribution spatiale des noeuds, nous utilisons le poids de chaque noeud, c'est-à-dire le nombre total de noeuds qu'il représente, pour générer une carte de chaleur qui est affichée comme fond. Comme les sommets du graphe ont été agrégés par le

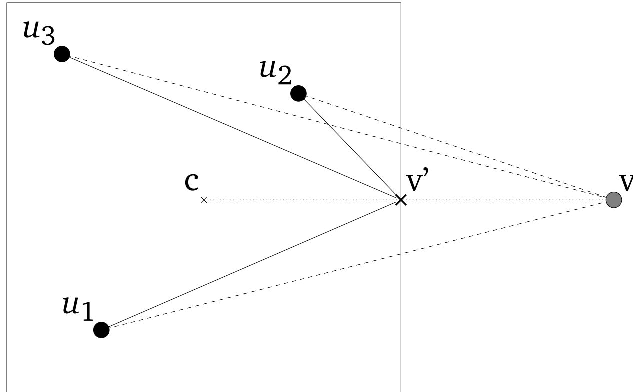


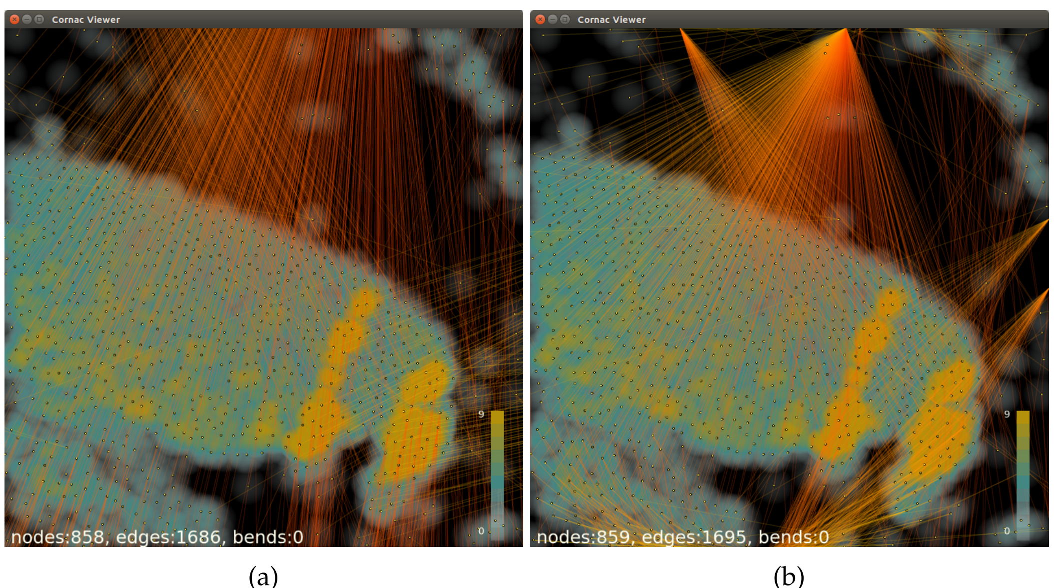
FIGURE 8.9 – Principe du clipping des arêtes au bord de l'écran. Les trois sommets u_1, u_2 et u_3 sont voisins de v. Comme v est en dehors de l'écran, les arêtes sont dessinées vers v', l'intersection du segment $[c, v]$ avec le bord de l'écran.

canopy clustering, les propriétés exposées au chapitre 7 sont toujours valables. Le même algorithme de rendu est utilisé, mais c'est le noyau d'Epanechnikov qui est utilisé ici, puisqu'il produit un lissage moins important et permet de bien souligner les groupes de noeuds dans le dessin du graphe.

8.4.3 Rendu des arêtes

Le rendu des arêtes utilise les connections de Fatum. Seules les arêtes incidentes aux sommets visibles sont affichées, ce qui évite l'occlusion des arêtes qui coupent l'écran mais dont les deux extrémités sont en dehors. Les arêtes sont traitées différemment suivant si une seule ou leurs deux extrémités sont visibles. Si les deux extrémités sont visibles, l'arête relie les deux sommets. Sinon, l'arête est traitée différemment. Soit c le centre de l'écran et l'arête (u, v), on considère que u est l'extrémité visible et que v se trouve en dehors de l'écran. On défini le point v' comme étant le point d'intersection entre le segment $[c, v]$ et le bord de l'écran. L'arête (u, v) est modifiée pour relier u à v'. La position de v' ne dépend pas de celle de u, mais uniquement de la position de v par rapport à l'écran. Toutes les arêtes incidentes à v et à un sommet visible se terminent donc au point v', comme l'illustre la Figure 8.9. Cela peut être vu comme un déplacement de v en v'. Ce déplacement permet d'identifier facilement si les sommets visibles possèdent des voisins communs en dehors de l'écran. Ce phénomène est illustré sur la Figure 8.10.

Les différentes arêtes affichées à l'écran ne représentent pas le même nombre d'arêtes dans le graphe original. Pour chaque arête, on dispose du nombre d'arêtes qu'elle représente, appelé e_{count}. Soient α_{min} et α_{max} les valeurs minimale et maximale de transparence et e_{max} le maximum de e_{count} affiché à l'écran. Pour permettre de repérer les arêtes les plus importantes, la transparence de chaque arête est modi-



(a) (b)

FIGURE 8.10 – Illustration de l'effet du clipping d'arêtes sur le bord de l'écran. (a) Sans clipping, les arêtes relient normalement leurs deux extrémités. On ne peut alors pas clairement identifier si les sommets affichés ont un voisin commun en dehors de l'écran. (b) Avec le clipping, il devient facile de repérer deux sommets très connectés au dessus de l'écran et deux autres à droite.

fiée à l'aide de la formule suivante :

$$\alpha_{min} + \frac{\log e_{count}}{\log e_{max}} \times (\alpha_{max} - \alpha_{min})$$

Avec cette formule, toutes les arêtes sont toujours visibles et l'échelle de transparence s'adapte au maximum affiché. En pratique, nous utilisons des valeurs empiriques de 50 pour α_{min} et 225 pour α_{max}, sur un maximum de 255.

8.4.4 Faisceautage d'arêtes

Le faisceautage d'arêtes est une technique visant à diminuer l'espace occupé à l'écran par les arêtes en les regroupant. Cela permet de diminuer l'occlusion et de mettre en exergue les principales connexions au sein du graphe. Introduite en 2006 par Holten [61], cette technique consiste à représenter des arêtes du graphe sous forme de courbes pour leur permettre de se regrouper aux endroits où il y a le plus d'arêtes. D'autres algorithmes de faisceautage ont étés introduits par la suite : par modèle de force [62], routage [69] ou en utilisant une fonction de densité[64].

Un des intérêts des techniques de faisceautage est de pouvoir changer les paramètres de l'algorithme utilisé pour modifier l'intensité du regroupement des arêtes. Pour chaque paramétrage de l'algorithme, une nouvelle géométrie des arêtes est

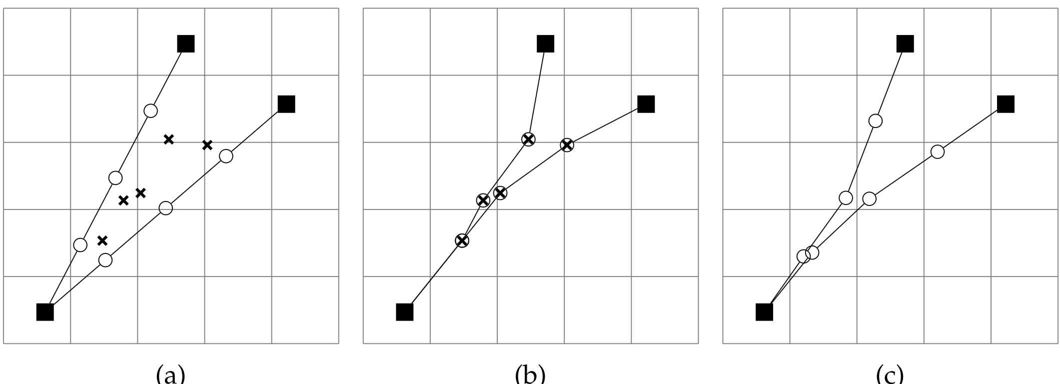
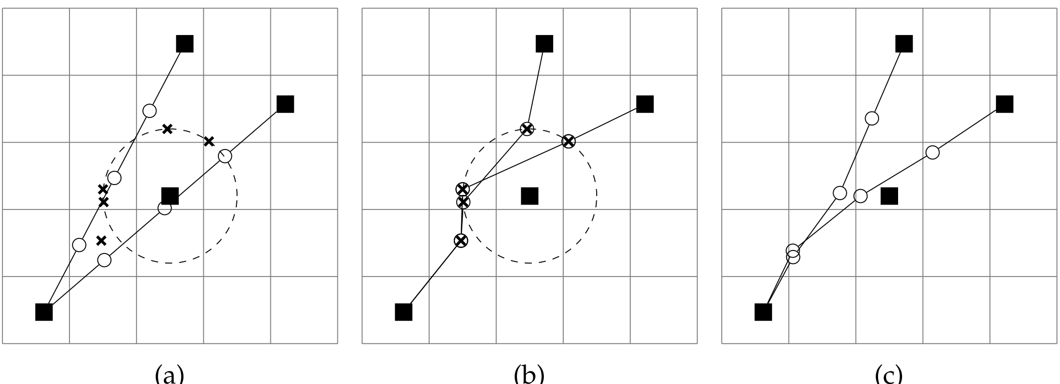


FIGURE 8.11 – Etapes de notre algorithme de faisceautage. Les extrémités des arêtes sont représentées sous forme de carrés. (a) Ajout des points de contrôle (cercles) et calcul du barycentre pour les cases qui contiennent un point de contrôle (croix). Ici, tous les barycentres affichés sont à l'interieur de la case correspondante. (b) Déplacement de chaque point de contrôle vers le barycentre de sa case. (c) Lissage des points de contrôle le long de chaque arête.

calculée. Dans notre cas, stocker la géométrie des arêtes pour chaque paramétrage de l'algorithme de faisceautage représente une quantité de données trop importante. Nous avons donc choisi de réaliser le faisceautage d'arêtes sur le client de visualisation. Le client ne disposant que de la partie visible des données, le faisceautage devra être recalculé à chaque fois que les données changent, soit potentiellement à chaque image. Notre objectif est donc d'avoir un algorithme suffisamment rapide pour être appliqué plusieurs dizaines de fois par seconde.

Notre algorithme est une simplification de l'algorithme par estimation de densité dit *Kernel Density Estimation Edge Bundling* (KDEEB) par Hurter et al. [64]. Il ne considère que les arêtes affichées à l'écran. L'écran est modélisé sous forme d'une grille. Le nombre de cases de cette grille est paramétrable et détermine la finesse du faisceautage. L'algorithme se décompose en quatre étapes :

— La première consiste à ajouter des points de contrôle sur chaque arête. Ces points seront déplacés pour modifier le chemin emprunté par l'arête. Les points sont répartis uniformément le long de l'arête, de manière à ce qu'il y ait un point par case de la grille traversée.
— La deuxième étape calcule pour chaque case de la grille le barycentre des points de contrôle de cette case et des cases adjacentes (voir Figure 8.11a). Ce barycentre peut donc se retrouver en dehors de la case qu'il représente.
— Pour la troisième étape, chaque point de contrôle est déplacé au barycentre associé à la case à laquelle il appartient (voir Figure 8.11b). Tous les points de contrôle d'une case sont alors regroupés.
— Enfin, la quatrième étape lisse les points de contrôle le long de chaque arête (voir Figure 8.11c). Chaque point est modifié pour être le barycentre du point précédent et du point suivant.



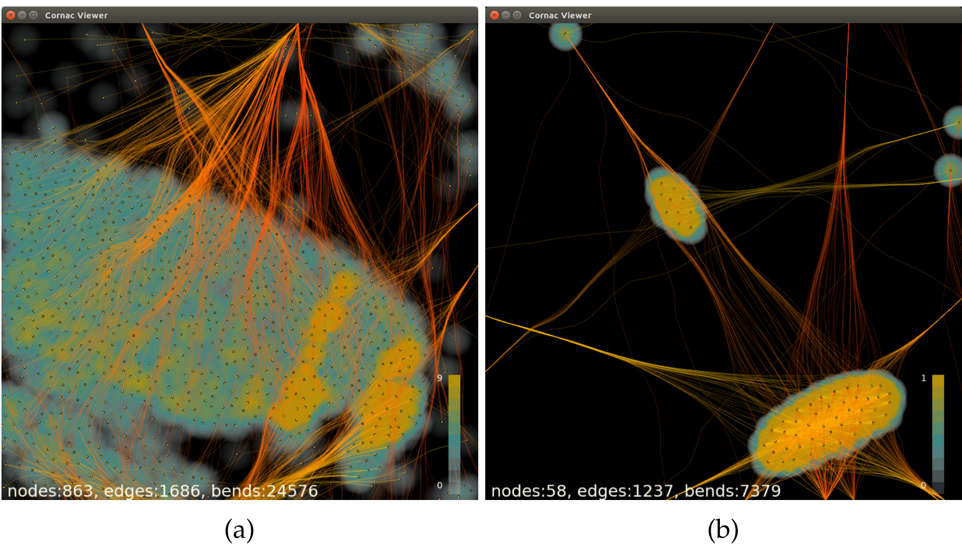
(a) (b) (c)

FIGURE 8.12 – Principe de l'évitement de sommets durant le faisceautage. (a) Les barycentres de chaque case sont déplacés à une distance unitaire du sommet à éviter. (b) Les points de contrôle sont ensuite déplacés normalement sur les barycentres. (c) Après le lissage, les points de contrôle sont légèrement plus éloignés du sommet à contourner. L'effet est plus prononcé au fur et à mesure des itérations.

L'intensité du faisceautage peut être contrôlée en répétant les étapes 2, 3 et 4. Le calcul du faisceautage à chaque image introduit une petite instabilité, c'est-à-dire que les points de contrôle calculés diffèrent légèrement d'une image à l'autre. De plus, pour limiter l'occlusion noeud/arête, les barycentres des cases qui contiennent un sommet du graphe sont déplacés à une distance suffisante du sommet. Ce principe est illustré sur la Figure 8.12. Ceci permet aux arêtes de contourner les noeuds, comme l'illustre l'exemple sur la Figure 8.13b.

Notre algorithme est suffisament rapide pour permettre le calcul du faisceautage à chaque image affichée. Par exemple, sur la Figure 8.13a, le temps de faisceautage est de 5ms, 10ms sur la Figure 8.7c et 15ms sur la Figure 8.8. Sur toutes ces figures, le nombre de points de contrôle est indiqué en bas sous l'appellation "bends". Ces points supplémentaires entraînent un coût de rendu plus élevé puisqu'il y a plus d'entité à gérer. Les performances de rendu de Fatum et la souplesse de son paradigme ont permis d'intégrer simplement et sans surcoût notre algorithme de faisceautage à la visualisation.

Notre algorithme diffère de l'algorithme de *KDEEB* de Hurter et al. [64] sur plusieurs points. Tout d'abord, notre algorithme ne considère que les arêtes affichées à l'écran, ce qui réduit les calculs nécessaires, mais ne permet pas de traiter le graphe entier. Ensuite, nous calculons une approximation de la fonction de densité au CPU là où le *KDEEB* calcule une fonction de densité beaucoup plus précise au GPU. Enfin, notre contrainte d'évitement des sommets est beaucoup moins sophistiquée que celle employée dans le *KDEEB*. Néanmoins, notre algorithme permet de réaliser du faisceautage d'arêtes en un temps suffisamment réduit pour être utilisé lors du calcul de chaque image, pour un résultat proche de celui des algorithmes de faisceautage de l'état de l'art. Des techniques plus récentes comme CUBu [122] pourraient servir à améliorer notre algorithme.



(a) (b)

FIGURE 8.13 – Exemples de résultat du faisceautage. (a) Faisceautage activé sur la vue de la Figure 8.10. (b) Effet de l'évitement de sommets autour d'un cluster. Les arêtes se séparent pour contourner le cluster.

8.5 Étude de cas : Panama Papers

Les *Panama Papers* sont une fuite de données concernant des sociétés extraterritoriales. Dévoilée en avril 2016 par l'*International Consortium of Investigative Journalists* (ICIJ), elle regroupe 11,5 millions de documents confidentiels appartenant au cabinet d'avocats panaméen *Mossack Fonseca*, pour un total de 2,6To de données. Les données contenues dans ces documents, ainsi que dans ceux de la précédente fuite baptisée *Offshore Leaks* de 2013, ont été exploitées pour extraire un graphe. Les données de ce graphe sont disponibles en téléchargement sur le site de l'ICIJ : `https://offshoreleaks.icij.org/pages/database`. Le graphe contient 839K sommets et 1,2M d'arêtes. Il regroupe des informations sur des sociétés, des actionnaires, des intermédiaires et des adresses.

8.5.1 Visualisation

Pour visualiser le graphe, nous avons extrait la composante connexe la plus importante, comprenant 750K sommets et 1,1M d'arêtes. Le reste du graphe est constitué de petits groupes de quelques sommets. Le dessin est obtenu grâce à l'algorithme FM^3 [54].

La Figure 8.14 présente différentes vues de la visualisation interactive produite par notre système. Une vue d'ensemble du graphe est visible sur la Figure 8.14a. Sur cette vue, le nombre de sommets, d'arêtes et de points de contrôle est affiché en bas à gauche. On peut remarquer que seul un petit nombre de sommets et d'arêtes sont affichés. Pourtant, on remarque facilement les zones les plus denses en sommets

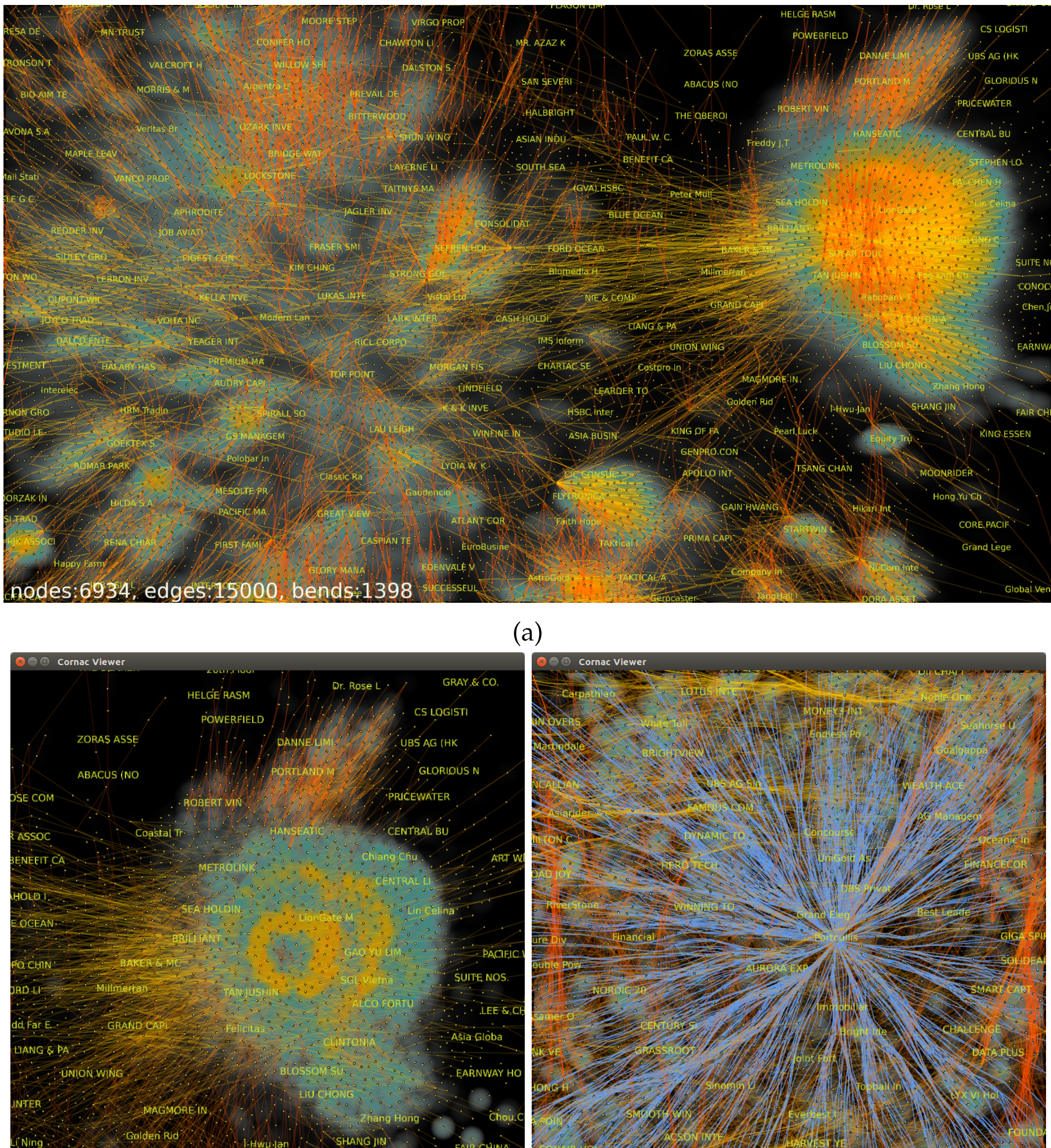


FIGURE 8.14 – Vues du graphe des Panama Papers. (a) Vue d'ensemble. La carte de chaleur permet de repérer les cluster. On remarque aussi une zone très dense en haut à droite. (b) Vue rapprochée du cluster dense. On distingue deux anneaux concentriques. (c) Zoom au centre de la structure. Le noeud central "Portcullis" est connecté à l'ensemble de l'anneau interne.

grâce à la carte de chaleur. Il est aussi possible de voir distinctement les zones denses qui correspondent à des clusters.

Le cluster le plus important se trouve en haut à droite du graphe. Il est facilement reconnaissable, puisque qu'il constitue la zone la plus dense du graphe. En zoomant sur cette partie du graphe, comme le montre la Figure 8.14b, on observe qu'il est constitué d'un noeud central très connecté et d'une structure en double anneau autour. En inspectant ce noeud central de plus près (voir Figure 8.14c), on peut s'apercevoir qu'il est connecté à l'anneau intérieur, dont les noeuds sont eux-mêmes connecté à l'anneau externe. L'étiquette du noeud nous renseigne sur son identité : il s'agit du cabinet d'avocat "Portcullis", qui est une des deux firmes dont sont issus les documents des *Offshore Leaks*, ce qui explique sa forte connectivité. On peut également repérer dans le graphe un certain nombre de sommets très connectés dont l'étiquette contient "Mossack Fonseca" ou "MOSSFON", en référence au cabinet panaméen.

8.5.2 Performances

L'étape d'agrégation a été réalisée sur la plateforme du laboratoire. Elle a permis de générer 15 niveaux de détail en 6 minutes, en utilisant 4 exécuteurs Spark avec 2 coeurs chacun. Le total des données agrégées occupe 594Mo dans HBase.

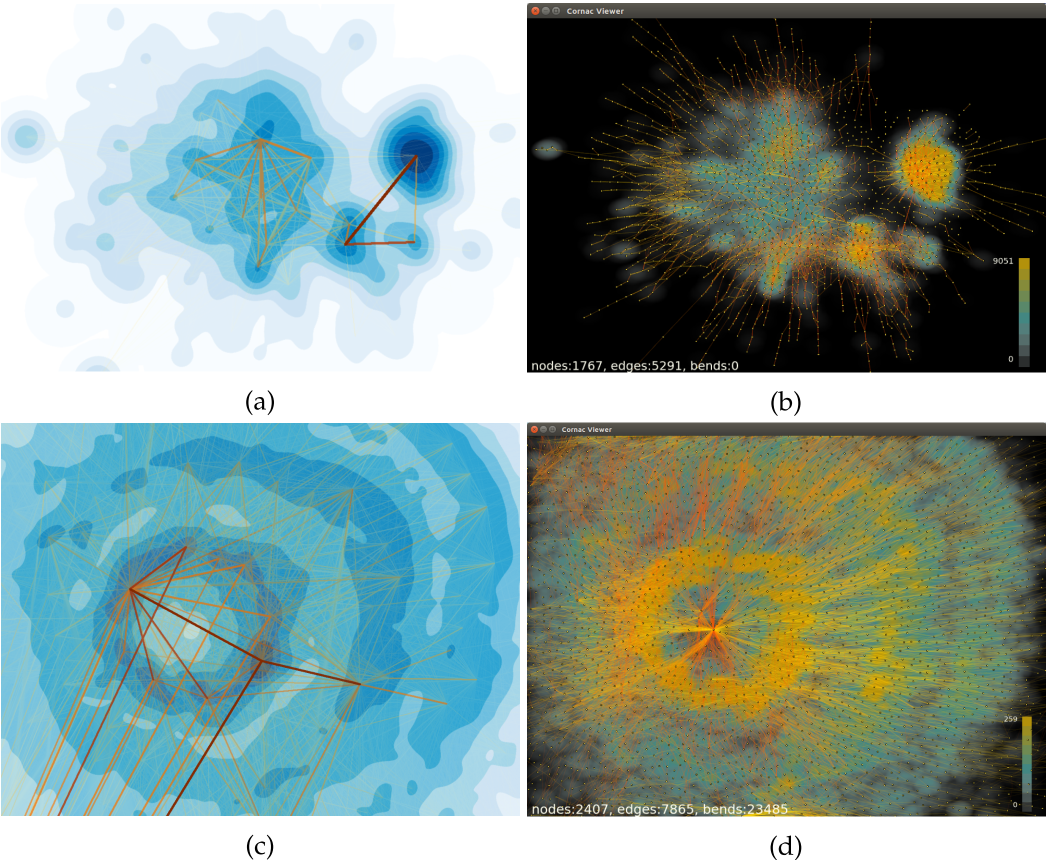
Lors de l'exploration du graphe, les tiles de sommets pèsent en moyenne 12Ko et les tiles d'arêtes 50Ko. Le temps de transfert est compris entre 10ms et 100ms. Le temps total de rendu de la visualisation, qui comprend le calcul des entités visibles et de la géométrie des arêtes ainsi que le dessin assuré par Fatum, est compris entre 10ms et 30ms, ce qui correspond à entre 100 et 30 images par seconde. En ajoutant du bundling, ce temps passe à entre 40ms et 70ms (entre 25 et 14 images par seconde), suivant le nombre d'arêtes affichées à l'écran. Notre système permet donc d'explorer interactivement le graphe Panama Papers.

8.5.3 Discussion

Nous avons choisi de comparer Cornac à la solution la plus proche dans la littérature, appelée LaGO [121]. Ce logiciel n'utilise pas de calcul distribué, mais utilise le GPU pour dessiner une fonction de densité. Il est implémenté sur un client lourd, contrairement à Cornac qui utilise un client web.

LaGO effectue d'abord une phase de précalcul. Pour le graphe des Panama Papers, elle a duré 2 minutes. C'est plus rapide que le précalcul de Cornac, mais les données ne sont pas partagées et doivent être recalculées par les autres utilisateurs.

Le résultat affiché par LaGO est visible sur la Figure 8.15. La vue d'ensemble (Figure 8.15a) montre la même structure que Cornac. Lorsque l'on zoome sur le cluster en haut à droite, on peut aussi observer la structure en double anneau à travers la fonction de densité des noeuds. En revanche, les arêtes affichées sont



(a) (b)

(c) (d)

FIGURE 8.15 – Comparaison de la visualisation du graphe des *Panama Papers* avec LaGO [121] et Cornac. (a et b) Vue d'ensemble du graphe. La fonction de densité permet de distinguer la même structure que Cornac. (c et d) Vue du cluster du noeud *Portcullis*. La structure en double anneau est bien visible dans les deux systèmes. En revanche, le noeud central fortement connecté semble avoir disparu avec LaGO. De plus, le déplacement des arêtes opéré par LaGO rend difficile d'observer la connectivité des anneaux exterieurs.

différentes de celles montrées par Cornac. En effet, LaGO déplace les extrémités des arêtes vers les maximum locaux de la fonction de densité. La connectivité de la structure s'en trouve donc visuellement altérée, puisque le noeud central n'est plus mis en évidence. Dans notre système le déplacement total des noeuds est limité par la distance d'agrégation, donc la connectivité affichée est visuellement proche de celle du graphe d'origine.

Lors d'une interaction avec LaGO (déplacement, zoom, changement des échelles de couleur), le système doit recalculer les différentes fonctions de densité. Sur le graphe des Panama Papers, ce calcul prend environ une second à chaque interaction. Ce délai induit un temps de latence entre l'action de l'utilisateur et l'affichage du résultat, ce qui nuit fortement à la fluidité de l'interaction avec la visualisation de LaGO. Cornac ne souffre pas de ce problème, puisque le nombre d'entité à l'écran est borné.

Pour les plus gros graphes, LaGO est capable d'afficher les graphes routier des USA en utilisant jusqu'à 16Go de mémoire vive. La phase de précalcul prend alors 8 minutes. En revanche, il n'a pas été possible d'afficher le graphe routier de l'europe.

8.6 Conclusion

La visualisation de grands graphes doit faire face aux mêmes problèmes que les autres types de visualisation de données massives : la scalabilité de perception et la scalabilité des performances. Dans ce chapitre, nous avons montré comment résoudre ces problèmes en appliquant la méthode présentée au chapitre 4. En utilisant le canopy clustering, il est possible d'agréger des graphes de plusieurs millions d'arêtes en quelques heures. Nous avons de plus montré comment réduire les bornes théoriques sur le nombre d'arêtes du graphe en agrégeant les arêtes longues.

La visualisation s'effectue ensuite sur un client de visualisation implémenté avec la bibliothèque Fatum, présentée au chapitre 6. L'étape d'agrégation permet de garantir un petit nombre d'éléments affichés et des interactions rapides. Nous avons également ajouté un algorithme de faisceautage d'arêtes en temps réel pour diminuer l'occlusion entre les arêtes et les noeuds.

Notre système permet de visualiser et d'explorer interactivement de grands graphes en tirant parti de l'infrastructure Big Data et du navigateur web. La taille des graphes n'est limitée que par la taille de l'infrastructure utilisée pour le calcul et le stockage des données agrégées. Les trois critères de scalabilité énoncés au chapitre 4 sont respectés : la scalabilité de perception, de calcul et de stockage.

9 Conclusion

Le phénomène du Big Data a entraîné une augmentation de la quantité de données disponibles. De telles quantités peuvent maintenant être stockées et traitées dans des temps acceptables grâce aux technologies émergentes, comme l'écosystème Hadoop. Malgré ces capacités, il n'est pas aisé d'extraire du sens de cette masse de données. La visualisation joue donc un rôle crucial dans l'exploitation des quantités faramineuses de données collectées. La visualisation repose sur les capacités et mécanismes de la perception humaine, qui, eux, ne connaissent pas la même évolution que les systèmes de traitement de l'information. La visualisation de données massives soulève donc deux problèmes fondamentaux concernant la scalabilité de la perception d'une part et des performances de calcul, de requêtage et de rendu d'autre part.

Dans cette thèse, nous avons proposé de résoudre ces deux problèmes par une approche multi-échelle. La décomposition de la visualisation en plusieurs niveaux de détail ainsi que la définition d'un budget d'entités visuelles permettent de résoudre la scalabilité de perception et des performances de requêtage et de rendu. En revanche, cela nécessite un précalcul supplémentaire. Ce coût est pris en charge par l'utilisation des technologies de calcul distribuées associées au Big Data, comme MapReduce et Spark.

Dans le chapitre 4, nous avons présenté notre méthode générale pour la visualisation de données massives s'appuyant sur le principe de décomposition multi-échelle. Cette méthode se base sur une modification du pipeline de visualisation pour l'adapter à la prise en charge de données massives stockées sur une infrastructure distante. Dans cette méthode, un algorithme d'agrégation géométrique est utilisé pour précalculer les niveaux de détail de l'abstraction multi-échelle. Grâce à l'indexation des niveaux de détail par une pyramide de tiles, le client de visualsation peut récupérer uniquement la partie visible des données. Ainsi, la taille des données transférées peut être bornée, ce qui permet d'explorer interactivement n'importe quel jeu de données.

Dans le chapitre 5, nous avons présenté comment l'algorithme de canopy clustering peut être utilisé pour produire une agrégation géométrique pour la visualisation. Cet algorithme permet d'obtenir un nombre borné d'entités représentant un

ensemble de départ. En l'appliquant récursivement, on obtient une abstraction multi-échelle d'un jeu de données à visualiser. Nous avons aussi proposé deux approches pour rendre cet algorithme distribué. D'une part, une approche par partitionnement spatial implémentable dans le paradigme MapReduce. D'autre part, une approche combinant graphes de disque et ensemble indépendant maximal, implémentable dans le paradigme de calcul de graphes Pregel.

Puis, nous avons présenté dans le chapitre 6 la bibliothèque de visualisation Fatum. Nous avons exposé le principe des *marks* et *connections* qui constituent le paradigme implémenté dans cette bibliothèque. Ce paradigme orienté visualisation permet de décrire facilement des visualisations avec des éléments complexes. Fatum gère la dynamicité de la visualisation et permet de passer d'un état de la visualisation au suivant sous forme d'animation. L'implémentation utilise des technologies de bas niveau pour obtenir les meilleures performances de rendu.

Nous avons ensuite présenté deux applications mettant en oeuvre les principes et techniques présentées précédemment. Tout d'abord, le système HeatMapReduce au chapitre 7, qui exploite le principe du *Approximate Kernel Density Estimate* pour permettre d'explorer interactivement des cartes de chaleur. Implémenté en utilisant le canopy clustering, il permet de visualiser plusieurs milliards de points. Enfin, nous avons présenté le système Cornac pour la visualisation de grands graphes dans le chapitre 8. En se basant sur les résultats présentés auparavant, ce système permet de visualiser des graphes de plusieurs millions d'arêtes.

Les travaux présentés dans cette thèse constituent une première approche complète de la visualisation de données massives à l'aide d'une infrastructure Big Data. Nous avons choisi d'avoir systématiquement recours à un précalcul complet pour réduire au maximum les temps d'accès lors de la visualisation. Nous avons présenté des méthodes pour accélérer ce précalcul, mais il peut néanmoins rester prohibitif. De nouvelles méthodes pour combiner précalcul et calcul à la volée sont à développer. De plus, les interactions possibles avec la visualisation sont limitées par ce qui a été précalculé. Il serait alors intéressant de conserver plus d'informations sur les agrégats produits pour permettre des interactions plus variées.

L'agrégation et les visualisations réalisées dans cette thèse sont indépendantes des jeux de données utilisés. Pour permettre d'utiliser ces méthodes pour retirer plus d'information d'un jeu données précis, l'utilisateur devrait pouvoir spécifier quelles métadonnées agréger et comment. De plus, ces informations devraient être accessibles depuis le client de visualisation et être reflétées dans la métaphore visuelle choisie.

Enfin, nous n'avons considéré ici que des données statiques, c'est-à-dire n'évoluant pas au cours du temps. Notre méthode devra donc être adaptée aux données dynamiques, par exemple en utilisant des cubes de données en tant que représentant au sein de l'abstraction multi-échelle.

Bibliographie

[1] J. Abello, F. van Ham et N. Krishnan. « Ask-graphview : A large scale graph visualization system ». *Visualization and Computer Graphics, IEEE Transactions on* 12.5 (2006), p. 669–676.

[2] C. Ahlberg et B. Shneiderman. « Visual information seeking : Tight coupling of dynamic query filters with starfield displays ». *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM. 1994, p. 313–317.

[3] D. Archambault, T. Munzner et D. Auber. « Grouse : feature-based, steerable graph hierarchy exploration ». *Proceedings of the 9th Joint Eurographics/IEEE VGTC conference on Visualization*. Eurographics Association. 2007, p. 67–74.

[4] D. Archambault, T. Munzner et D. Auber. « GrouseFlocks : Steerable exploration of graph hierarchy space ». *IEEE transactions on visualization and computer graphics* 14.4 (2008), p. 900–913.

[5] D. Archambault, T. Munzner et D. Auber. « Tuggraph : Path-preserving hierarchies for browsing proximity and paths in graphs ». *Visualization Symposium, 2009. PacificVis' 09. IEEE Pacific*. IEEE. 2009, p. 113–120.

[6] A. Arleo, W. Didimo, G. Liotta et F. Montecchiani. « A million edge drawing for a fistful of dollars ». *International Symposium on Graph Drawing and Network Visualization*. Springer. 2015, p. 44–51.

[7] A. Arleo, W. Didimo, G. Liotta et F. Montecchiani. « A Distributed Multilevel Force-Directed Algorithm ». *International Symposium on Graph Drawing and Network Visualization*. Springer. 2016, p. 3–17.

[8] D. Auber, R. Bourqui, M. Delest, A. Lambert, P. Mary, G. Melançon, B. Pinaud, B. Renoust et J. Vallet. *TULIP 4*. Research Report. LaBRI - Laboratoire Bordelais de Recherche en Informatique, sept. 2016.

[9] D. Auber, Y. Chiricota, F. Jourdan et G. Melançon. « Multiscale Visualization of Small World Networks ». *IEEE Symposition on Information Visualisation*. 2003, p. 75–81.

[10] M. Balzer et O. Deussen. « Level-of-detail visualization of clustered graph layouts ». *Visualization, 2007. APVIS'07. 2007 6th International Asia-Pacific Symposium on*. IEEE. 2007, p. 133–140.

[11] J. L. BENTLEY et M. I. SHAMOS. « Divide-and-conquer in multidimensional space ». *Proceedings of the eighth annual ACM symposium on Theory of computing*. ACM. 1976, p. 220–230.

[12] J. BERTIN. « Sémiologie graphique : les diagrammes, les réseaux, les cartes ». *Paris, École des hautes études en sciences sociales/Mouton-Gauthier-Villars* (1967).

[13] J. BERTIN. *La graphique et le traitement graphique de l'information*. 91 (084.21) BER. 1977.

[14] E. BERTINI et G. SANTUCCI. « Give chance a chance : modeling density to enhance scatter plot quality through random data sampling ». *Information Visualization* 5.2 (2006), p. 95–110.

[15] M. BEYER. « Gartner Says Solving'Big Data'Challenge Involves More Than Just Managing Volumes of Data ». *Gartner. Archived from the original on* 10 (2011).

[16] N. BIKAKIS, J. LIAGOURIS, M. KROMMYDA, G. PAPASTEFANATOS et T. SELLIS. « graphVizdb : A scalable platform for interactive large graph visualization ». *Data Engineering (ICDE), 2016 IEEE 32nd International Conference on*. IEEE. 2016, p. 1342–1345.

[17] R. BLANCH et E. LECOLINET. « Browsing zoomable treemaps : Structure-aware multi-scale navigation techniques ». *IEEE Transactions on Visualization and Computer Graphics* 13.6 (2007), p. 1248–1253.

[18] G. E. BLELLOCH, J. T. FINEMAN et J. SHUN. « Greedy sequential maximal independent set and matching are parallel on average ». *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*. ACM. 2012, p. 308–317.

[19] V. D. BLONDEL, J.-L. GUILLAUME, R. LAMBIOTTE et E. LEFEBVRE. « Fast unfolding of communities in large networks ». *Journal of statistical mechanics : theory and experiment* 2008.10 (2008), P10008.

[20] M. BOSTOCK et J. HEER. « Protovis : A graphical toolkit for visualization ». *IEEE transactions on visualization and computer graphics* 15.6 (2009).

[21] M. BOSTOCK, V. OGIEVETSKY et J. HEER. « D^3 data-driven documents ». *IEEE transactions on visualization and computer graphics* 17.12 (2011), p. 2301–2309.

[22] R. BRIDSON. « Fast Poisson disk sampling in arbitrary dimensions. » *SIGGRAPH sketches*. 2007, p. 22.

[23] M. BURCH, C. VEHLOW, F. BECK, S. DIEHL et D. WEISKOPF. « Parallel edge splatting for scalable dynamic graph visualization ». *IEEE Transactions on Visualization and Computer Graphics* 17.12 (2011), p. 2344–2353.

[24] S. K. CARD, J. D. MACKINLAY et B. SHNEIDERMAN. *Readings in information visualization : using vision to think*. Morgan Kaufmann, 1999.

[25] M. S. T. CARPENDALE, D. J. COWPERTHWAITE et F. D. FRACCHIA. « 3-dimensional pliable surfaces : For the effective presentation of visual information ». *Proceedings of the 8th annual ACM symposium on User interface and software technology*. ACM. 1995, p. 217–226.

[26] D. B. CARR, R. J. LITTLEFIELD, W. NICHOLSON et J. LITTLEFIELD. « Scatterplot matrix techniques for large N ». *Journal of the American Statistical Association* 82.398 (1987), p. 424–436.

[27] F. CHANG, J. DEAN, S. GHEMAWAT, W. C. HSIEH, D. A. WALLACH, M. BURROWS, T. CHANDRA, A. FIKES et R. E. GRUBER. « Bigtable : A distributed storage system for structured data ». *ACM Transactions on Computer Systems (TOCS)* 26.2 (2008), p. 4.

[28] H. CHEN, W. CHEN, H. MEI, Z. LIU, K. ZHOU, W. CHEN, W. GU et K.-L. MA. « Visual abstraction and exploration of multi-class scatterplots ». *IEEE transactions on visualization and computer graphics* 20.12 (2014), p. 1683–1692.

[29] B. CHESWICK, H. BURCH et S. BRANIGAN. « Mapping and Visualizing the Internet. » *USENIX Annual Technical Conference, General Track*. Citeseer. 2000, p. 1–12.

[30] F. CHEVALIER, P. DRAGICEVIC et C. HURTER. « Histomages : fully synchronized views for image editing ». *Proceedings of the 25th annual ACM symposium on User interface software and technology*. ACM. 2012, p. 281–286.

[31] W. S. CLEVELAND et R. MCGILL. « Graphical perception : Theory, experimentation, and application to the development of graphical methods ». *Journal of the American statistical association* 79.387 (1984), p. 531–554.

[32] J. H. CONWAY et N. J. A. SLOANE. *Sphere packings, lattices and groups*. T. 290. Springer Science & Business Media, 2013.

[33] T. CORMEN, C. LEISERSON, R. RIVEST et C. STEIN. *Introduction to algorithms*. 1989.

[34] J. COTTAM, A. LUMSDAINE et P. WANG. « Overplotting : Unified solutions under abstract rendering ». *Big Data, 2013 IEEE International Conference on*. IEEE. 2013, p. 9–16.

[35] A. DAS SARMA, H. LEE, H. GONZALEZ, J. MADHAVAN et A. HALEVY. « Efficient spatial sampling of large geographical tables ». *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM. 2012, p. 193–204.

[36] J. DEAN et S. GHEMAWAT. « MapReduce : simplified data processing on large clusters ». *Communications of the ACM* 51.1 (2008), p. 107–113.

[37] S. DOS SANTOS et K. BRODLIE. « Gaining understanding of multivariate and multidimensional data through visualization ». *Computers & Graphics* 28.3 (2004), p. 311–325.

[38] P. EADES et Q.-W. FENG. « Multilevel visualization of clustered graphs ». *Graph drawing*. Springer. 1997, p. 101–112.

[39] A. ELDAWY et M. F. MOKBEL. « SpatialHadoop : A MapReduce Framework for Spatial Data ». *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*. 2015, p. 1352–1363.

[40] A. ELDAWY, M. F. MOKBEL, S. ALHARTHI, A. ALZAIDY, K. TAREK et S. GHANI. « Shahed : A mapreduce-based system for querying and visualizing spatio-temporal satellite data ». *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*. IEEE. 2015, p. 1585–1596.

[41] A. ELDAWY, M. F. MOKBEL et C. JONATHAN. « HadoopViz : A MapReduce framework for extensible visualization of big spatial data ». *Data Engineering (ICDE), 2016 IEEE 32nd International Conference on*. IEEE. 2016, p. 601–612.

[42] G. Ellis et A. Dix. « A taxonomy of clutter reduction for information visualisation ». *IEEE transactions on visualization and computer graphics* 13.6 (2007), p. 1216–1223.

[43] N. Elmqvist, T.-N. Do, H. Goodell, N. Henry et J.-D. Fekete. « ZAME : Interactive large-scale graph visualization ». *Visualization Symposium, 2008. PacificVIS'08. IEEE Pacific*. IEEE. 2008, p. 215–222.

[44] N. Elmqvist et J.-D. Fekete. « Hierarchical aggregation for information visualization : Overview, techniques, and design guidelines ». *Visualization and Computer Graphics, IEEE Transactions on* 16.3 (2010), p. 439–454.

[45] L. Fejes. « Über die dichteste Kugellagerung ». *Mathematische Zeitschrift* 48.1 (1942), p. 676–684.

[46] J.-D. Fekete et C. Plaisant. « Interactive information visualization of a million items ». *Information Visualization, 2002. INFOVIS 2002. IEEE Symposium on*. IEEE. 2002, p. 117–124.

[47] R. A. Finkel et J. L. Bentley. « Quad trees a data structure for retrieval on composite keys ». *Acta informatica* 4.1 (1974), p. 1–9.

[48] D. Fisher. « Hotmap : Looking at Geographic Attention ». *IEEE Transactions on Visualization and Computer Graphics* 13.6 (nov. 2007), p. 1184–1191.

[49] G. W. Furnas. *Generalized fisheye views*. T. 17. 4. ACM, 1986.

[50] G. W. Furnas et B. B. Bederson. « Space-scale diagrams : Understanding multiscale interfaces ». *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM Press/Addison-Wesley Publishing Co. 1995, p. 234–241.

[51] E. R. Gansner, Y. Koren et S. C. North. « Topological fisheye views for visualizing large graphs ». *IEEE Transactions on Visualization and Computer Graphics* 11.4 (2005), p. 457–468.

[52] C. Green. « Improved alpha-tested magnification for vector textures and special effects ». *ACM SIGGRAPH 2007 courses*. ACM. 2007, p. 9–18.

[53] R. B. Haber et D. A. McNabb. « Visualization idioms : A conceptual model for scientific visualization systems ». *Visualization in scientific computing* 74 (1990), p. 93.

[54] S. Hachul et M. Jünger. « Drawing large graphs with a potential-field-based multilevel algorithm ». *International Symposium on Graph Drawing*. Springer. 2004, p. 285–295.

[55] S. Har-Peled. *Geometric approximation algorithms*. T. 173.

[56] J. A. Hartigan et M. A. Wong. « Algorithm AS 136 : A k-means clustering algorithm ». *Journal of the Royal Statistical Society. Series C (Applied Statistics)* 28.1 (1979), p. 100–108.

[57] C. G. Healey, K. S. Booth et J. T. Enns. « Visualizing real-time multivariate data using preattentive processing ». *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 5.3 (1995), p. 190–221.

[58] J. Heer et M. Bostock. « Declarative language design for interactive visualization ». *IEEE Transactions on Visualization and Computer Graphics* 16.6 (2010), p. 1149–1156.

[59] A. Hinge et D. Auber. « Distributed Graph Layout with Spark ». *Information Visualisation (iV), 2015 19th International Conference on*. IEEE. 2015, p. 271–276.

[60] N. HOLLIMAN et P. WATSON. « Scalable Real-Time Visualization Using the Cloud ». *IEEE Cloud Computing* 2.6 (2015), p. 90–96.

[61] D. HOLTEN. « Hierarchical edge bundles : Visualization of adjacency relations in hierarchical data ». *IEEE Transactions on visualization and computer graphics* 12.5 (2006), p. 741–748.

[62] D. HOLTEN et J. J. VAN WIJK. « Force-Directed Edge Bundling for Graph Visualization ». *Computer graphics forum*. T. 28. 3. Wiley Online Library. 2009, p. 983–990.

[63] S. E. HUDSON et J. T. STASKO. « Animation support in a user interface toolkit : Flexible, robust, and reusable abstractions ». *Proceedings of the 6th annual ACM symposium on User interface software and technology*. ACM. 1993, p. 57–67.

[64] C. HURTER, O. ERSOY et A. TELEA. « Graph bundling by kernel density estimation ». *Computer Graphics Forum*. T. 31. 3pt1. Wiley Online Library. 2012, p. 865–874.

[65] D. JONKER, S. LANGEVIN, D. GIESBRECHT, M. CROUCH et N. KRONENFELD. « Graph mapping : Multi-scale community visualization of massive graph data ». *Information Visualization* 16.3 (2017), p. 190–204.

[66] D. A. KEIM. « Designing pixel-oriented visualization techniques : Theory and applications ». *IEEE Transactions on visualization and computer graphics* 6.1 (2000), p. 59–78.

[67] D. A. KEIM, C. PANSE, M. SIPS et S. C. NORTH. « Pixelmaps : A new visual data mining approach for analyzing large spatial data sets ». *Data Mining, 2003. ICDM 2003. Third IEEE International Conference on*. IEEE. 2003, p. 565–568.

[68] G. KINDLMANN et C. SCHEIDEGGER. « An Algebraic Process for Visualization Design ». *InfoVis*. IEEE, 2014.

[69] A. LAMBERT, R. BOURQUI et D. AUBER. « Winding roads : Routing edges into bundles ». *Computer Graphics Forum*. T. 29. 3. Wiley Online Library. 2010, p. 853–862.

[70] O. D. LAMPE et H. HAUSER. « Interactive visualization of streaming data with kernel density estimation ». *Pacific Visualization Symposium (PacificVis), 2011 IEEE*. IEEE. 2011, p. 171–178.

[71] D. LANEY. « 3D data management : Controlling data volume, velocity and variety ». *META Group Research Note* 6 (2001), p. 70.

[72] J. LESKOVEC et A. KREVL. *SNAP Datasets : Stanford Large Network Dataset Collection*. http://snap.stanford.edu/data. Juin 2014.

[73] C. LI, G. BACIU et Y. HAN. « Interactive visualization of high density streaming points with heat-map ». *Smart Computing (SMARTCOMP), 2014 International Conference on*. IEEE. 2014, p. 145–149.

[74] Z. LIU, B. JIANG et J. HEER. « imMens : Real-time Visual Querying of Big Data ». *Computer Graphics Forum*. T. 32. 3pt4. Wiley Online Library. 2013, p. 421–430.

[75] M. LUBY. « A simple parallel algorithm for the maximal independent set problem ». *Proceedings of the seventeenth annual ACM symposium on Theory of computing*. ACM. 1985, p. 1–10.

[76] G. MALEWICZ, M. H. AUSTERN, A. J. BIK, J. C. DEHNERT, I. HORN, N. LEISER et G. CZAJKOWSKI. « Pregel : a system for large-scale graph processing ». *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM. 2010, p. 135–146.

[77] N. MARZ et J. WARREN. *Big Data : Principles and best practices of scalable realtime data systems*. Manning Publications Co., 2015.

[78] A. MAYORGA et M. GLEICHER. « Splatterplots : Overcoming overdraw in scatter plots ». *IEEE transactions on visualization and computer graphics* 19.9 (2013), p. 1526–1538.

[79] A. MCCALLUM, K. NIGAM et L. H. UNGAR. « Efficient clustering of high-dimensional data sets with application to reference matching ». *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM. 2000, p. 169–178.

[80] Y. MÉTIVIER, J. M. ROBSON, N. SAHEB-DJAHROMI et A. ZEMMARI. « An optimal bit complexity randomized distributed MIS algorithm ». *Distributed Computing* 23.5-6 (2011), p. 331–340.

[81] P. D. MICHAILIDIS et K. G. MARGARITIS. « Accelerating kernel density estimation on the GPU using the CUDA framework ». *Applied Mathematical Sciences* 7.30 (2013), p. 1447–1476.

[82] P. D. MICHAILIDIS et K. G. MARGARITIS. « Parallel computing of kernel density estimation with different multi-core programming models ». *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*. IEEE. 2013, p. 77–85.

[83] T. MUNZNER. *Visualization analysis and design*. CRC Press, 2014.

[84] L. NACHMANSON, R. PRUTKIN, B. LEE, N. HENRY RICHE, A. E. HOLROYD et X. CHEN. « GraphMaps : Browsing Large Graphs as interactive Maps ». *Graph Drawing*. Springer. 2015.

[85] E. PARZEN. « On estimation of a probability density function and mode ». *The annals of mathematical statistics* 33.3 (1962), p. 1065–1076.

[86] A. PERROT et D. AUBER. « FATuM-Fast Animated Transitions using Multi-Buffers ». *Information Visualisation (iV), 2015 19th International Conference on*. IEEE. 2015, p. 75–82.

[87] A. PERROT, R. BOURQUI, N. HANUSSE et D. AUBER. « HeatPipe : High Throughput, Low Latency Big Data Heatmap with Spark Streaming ». *IV2017-21st International Conference on Information Visualisation*. 2017.

[88] A. PERROT, R. BOURQUI, N. HANUSSE, F. LALANNE et D. AUBER. « Large interactive visualization of density functions on big data infrastructure ». *Large Data Analysis and Visualization (LDAV), 2015 IEEE 5th Symposium on*. IEEE. 2015, p. 99–106.

[89] W. PLAYFAIR. *A Letter on Our Agricultural Distresses, Their Causes and Remedies : Accompanied with Tables and Copper-plate Charts, Shewing and Comparing the Prices of Wheat, Bread and Labour from 1565 to 1821...* Sams, 1822.

[90] A. QUIGLEY et P. EADES. « FADE : Graph drawing, clustering, and visual abstraction ». *Graph Drawing*. Springer. 2001, p. 197–210.

[91] M. RABIN. *Probabilistic algorithms, Algorithms and Complexity : New directions and recent trends (JF Traub, ed.)* 1976.

[92] D. RAFIEI. « Effectively visualizing large networks through sampling ». *Visualization, 2005. VIS 05. IEEE*. IEEE. 2005, p. 375–382.

[93] M. ROSENBLATT et al. « Remarks on some nonparametric estimates of a density function ». *The Annals of Mathematical Statistics* 27.3 (1956), p. 832–837.

[94] N. ROUGIER. « Higher quality 2D text rendering ». *Journal of Computer Graphics Techniques* 2.1 (2013), p. 50–64.

[95] N. P. ROUGIER. « Antialiased 2d grid, marker, and arrow shaders ». *Journal of Computer Graphics Techniques* 3.4 (2014), p. 52.

[96] J. SANSEN, F. LALANNE, D. AUBER et R. BOURQUI. « Adjasankey : Visualization of huge hierarchical weighted and directed graphs ». *Information Visualisation (iV), 2015 19th International Conference on*. IEEE. 2015, p. 211–216.

[97] M. SARKAR et M. H. BROWN. « Graphical fisheye views of graphs ». *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM. 1992, p. 83–91.

[98] A. SATYANARAYAN, D. MORITZ, K. WONGSUPHASAWAT et J. HEER. « Vega-lite : A grammar of interactive graphics ». *IEEE Transactions on Visualization and Computer Graphics* 23.1 (2017), p. 341–350.

[99] B. SHNEIDERMAN. « Tree visualization with tree-maps : 2-d space-filling approach ». *ACM Transactions on graphics (TOG)* 11.1 (1992), p. 92–99.

[100] B. SHNEIDERMAN. « The eyes have it : A task by data type taxonomy for information visualizations ». *Visual Languages, 1996. Proceedings., IEEE Symposium on*. IEEE. 1996, p. 336–343.

[101] M. SMID. « Closest-point problems in computational geometry » (1997).

[102] A. THUE. *Über die dichteste Zusammenstellung von kongruenten Kreisen in einer Ebene*. J. Dybwad, 1910.

[103] C. TOMINSKI, J. ABELLO, F. VAN HAM et H. SCHUMANN. « Fisheye tree views and lenses for graph visualization ». *Information Visualization, 2006. IV 2006. Tenth International Conference on*. IEEE. 2006, p. 17–24.

[104] C. TOMINSKI, S. GLADISCH, U. KISTER, R. DACHSELT et H. SCHUMANN. « A survey on interactive lenses in visualization ». *EuroVis State-of-the-Art Reports* 3 (2014).

[105] M. TRUTSCHL, G. GRINSTEIN et U. CVEK. « Intelligently resolving point occlusion ». *Information Visualization, 2003. INFOVIS 2003. IEEE Symposium on*. IEEE. 2003, p. 131–136.

[106] E. R. TUFTE. « The visual display of quantitative information ». *The visual display of quantitative information/Edward R. Tufte. Cheshire, Conn. : Graphics Press, c1983.* (1983).

[107] L. G. VALIANT. « A bridging model for parallel computation ». *Communications of the ACM* 33.8 (1990), p. 103–111.

[108] F. VAN HAM et J. J. VAN WIJK. « Interactive visualization of small world graphs ». *Information Visualization, 2004. INFOVIS 2004. IEEE Symposium on*. IEEE. 2004, p. 199–206.

[109] R. VAN LIERE et W. DE LEEUW. « Graphsplatting : Visualizing graphs as continuous fields ». *IEEE Transactions on Visualization and Computer Graphics* 9.2 (2003), p. 206–212.

[110] H. T. VO, J. BRONSON, B. SUMMA, J. L. COMBA, J. FREIRE, B. HOWE, V. PASCUCCI et C. T. SILVA. « Parallel visualization on large clusters using MapReduce ». *Large Data Analysis and Visualization (LDAV), 2011 IEEE Symposium on*. IEEE. 2011, p. 81–88.

[111] Z. WANG, A. C. BOVIK, H. R. SHEIKH et E. P. SIMONCELLI. « Image quality assessment : from error visibility to structural similarity ». *IEEE Transactions on Image Processing* 13.4 (2004), p. 600–612.

[112] H. WICKHAM. « Bin-summarise-smooth : a framework for visualising large data ». *had.co.nz, Technical Report* (2013).

[113] L. WILKINSON et M. FRIENDLY. « The history of the cluster heat map ». *The American Statistician* 63.2 (2009).

[114] C. WILLIAMSON et B. SHNEIDERMAN. « The Dynamic HomeFinder : Evaluating dynamic queries in a real-estate information exploration system ». *Proceedings of the 15th annual international ACM SIGIR conference on Research and development in information retrieval*. ACM. 1992, p. 338–346.

[115] N. WONG, S. CARPENDALE et S. GREENBERG. « Edgelens : An interactive method for managing edge congestion in graphs ». *Information Visualization, 2003. INFOVIS 2003. IEEE Symposium on*. IEEE. 2003, p. 51–58.

[116] M. ZAHARIA, M. CHOWDHURY, T. DAS, A. DAVE, J. MA, M. MCCAULEY, M. J. FRANKLIN, S. SHENKER et I. STOICA. « Resilient distributed datasets : A fault-tolerant abstraction for in-memory cluster computing ». *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association. 2012, p. 2–2.

[117] M. ZAHARIA, M. CHOWDHURY, M. J. FRANKLIN, S. SHENKER et I. STOICA. « Spark : cluster computing with working sets ». *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*. T. 10. 2010, p. 10.

[118] A. ZAKAI. « Emscripten : an LLVM-to-JavaScript compiler ». *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. ACM. 2011, p. 301–312.

[119] Y. ZHENG, J. JESTES, J. M. PHILLIPS et F. LI. « Quality and efficiency for kernel density estimates in large data ». *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM. 2013, p. 433–444.

[120] T. ZINNER, O. HOHLFELD, O. ABBOUD et T. HOSSFELD. « Impact of frame rate and resolution on objective QoE metrics ». *International Workshop on Quality of Multimedia Experience*. IEEE. 2010, p. 29–34.

[121] M. ZINSMAIER, U. BRANDES, O. DEUSSEN et H. STROBELT. « Interactive level-of-detail rendering of large graphs ». *Visualization and Computer Graphics, IEEE Transactions on* 18.12 (2012), p. 2486–2495.

[122] M. van der ZWAN, V. CODREANU et A. TELEA. « CUBu : universal real-time bundling for large graphs ». *IEEE transactions on visualization and computer graphics* 22.12 (2016), p. 2550–2563.