N^o d'ordre : 4664

THÈSE

présentée à

L'UNIVERSITÉ BORDEAUX I

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET INFORMATIQUE

par Antoine LAMBERT

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : Informatique

Visualisation interactive de graphes : élaboration et optimisation d'algorithmes à coûts computationnels élevés.

Soutenue le : 12 décembre 2012

Après avis de :

M.	Alexandru Telea	Professeur	**Rapporteur**
M.	Alexander Wolff	Professeur	**Rapporteur**

Devant la Commission d'Examen composée de :

M.	Jean-Philippe Domenger .	Professeur	**Président**
M.	Christophe Hurter	Professeur	**Examinateur invité**
M.	Alexandru Telea	Professeur	**Rapporteur**
M.	Alexander Wolff	Professeur	**Rapporteur**
M.	Guy Melançon	Professeur	**Directeur de thèse**
M.	David Auber	Maître de conférences	**Co-directeur de thèse**

- 2012 -

Remerciements

Mes remerciements vont dans un premier temps à l'Agence Nationale pour la Recherche qui aura financée ma bourse de thèse via le projet TANGUY mené en partenariat avec Thalès Communications, Xerox, FIDAL et l'équipe projet Inria GRAVITE dans laquelle j'étais intégré. Je remercie également l'Université Bordeaux 1, l'Inria et le LaBRI qui m'ont permis de réaliser cette thèse dans de très bonnes conditions de travail. Un grand merci également à mes rapporteurs : M. Alexandru Tela et M. Alexander Wolff d'avoir accepté de relire ce manuscript et pour leurs suggestions d'amélioration et de correction.

Je tiens ensuite à remercier l'ensemble des membres de l'équipe projet Inria GRAVITE pour leur convivialité et bonne humeur quotidienne durant ces trois années passées à leur côté. Je tiens particulièrement à remercier mon encadrant de thèse David Auber pour la confiance qu'il m'a accordé durant ces années et son soutien dans les moments difficiles et de doute que tout doctorant rencontre au cours de sa thèse. Je lui signifie toute ma gratitude et mon amitié pour les bons moments passés ensemble et son goût communicatif pour la recherche innovante. Je tiens également à remercier Romain Bourqui, maître de conférences de l'équipe, pour son amitié et toute l'aide qu'il a pu m'apporter dans mon travail de recherche. Cette thèse serait bien moins riche en publications si tu n'avais pas eu les bonnes idées. Merci également à tous les autres membres et ex-membres de GRAVITE que j'ai eu la chance de rencontrer : Guy Melançon, Maylis Delest, Bruno Pinaud, Patrick Mary, Alice Rivière, Jonathan Dubois, Morgan Mathiaut, Frederic Gilbert, Pierre-Yves Koenig, Paolo Simonetto, Daniel Archambault, Arnaud Sallaberry, Faraz Zaidi, François Queyroi, Maurin Nadal, Ludwig Fiolka et Charles Huet.

Enfin je voudrai remercier Laurent Garnier, dj et producteur de musique électronique français, pour toute la bonne musique (électronique et acoustique) que j'ai pu découvrir grâce à lui durant toute mes années de thèse. Si le lecteur est un brin intéressé par la musique et sa diversité, je conseille fortement d'écouter l'émission de radio de Laurent *It is what it is* ainsi que sa radio web *Pedro's Broadcasting Basement (PBB)*, accessible depuis son site web : http://www.laurentgarnier.com/PBB.html.

Visualisation interactive de graphes : élaboration et optimisation d'algorithmes à coûts computationnels élevés.

Résumé : Un graphe est un objet mathématique modélisant des relations sur un ensemble d'éléments. Il est utilisé dans de nombreux domaines à des fins de modélisation. La taille et la complexité des graphes manipulés de nos jours entraînent des besoins de visualisation afin de mieux les analyser. Dans cette thèse, nous présentons différents travaux en visualisation interactive de graphes qui s'attachent à exploiter les architectures de calcul parallèle (CPU et GPU) disponibles sur les stations de travail contemporaines.

Un premier ensemble de travaux s'intéresse à des problématiques de dessin de graphes. Dessiner un graphe consiste à le plonger visuellement dans un plan ou un espace. La première contribution dans cette thématique est un algorithme de regroupement d'arêtes en faisceaux appelé *Winding Roads*. Cet algorithme intuitif, facilement implémentable et parallélisable permet de réduire considérablement les problèmes d'occlusion dans un dessin de graphe dus aux nombreux croisements d'arêtes. La seconde contribution est une méthode permettant de dessiner un réseau métabolique complet. Ce type de réseau modélise l'ensemble des réactions biochimiques se produisant dans les cellules d'un organise vivant. L'avantage de la méthode est de prendre en compte la décomposition du réseau en sous-ensembles fonctionnels ainsi que de respecter les conventions de dessin biologique.

Un second ensemble de travaux porte sur des techniques d'infographie pour la visualisation interactive de graphes. La première contribution dans cette thématique est une technique de rendu de courbes paramétriques exploitant pleinement le processeur graphique. La seconde contribution est une méthode de rendu nommée *Edge splatting* permettant de visualiser la densité des faisceaux d'arêtes dans un dessin de graphe avec regroupement d'arêtes. La dernière contribution porte sur des techniques permettant de mettre en évidence des sous-graphes d'intérêt dans le contexte global d'une visualisation de graphes.

Mots-clés : Visualisation de graphes, regroupement d'arêtes, bioinformatique, infographie
Discipline : Informatique.

Interactive graph visualisation : elaboration and optimisation of algorithms with high computationnal cost.

Abstract : A graph is a mathematical object used to model relations over a set of elements. It is used in numerous fields for modeling purposes. The size and complexity of graphs manipulated today call a need for visualization to better analyze them. In that thesis, we introduce different works in interactive graph visualisation which aim at exploiting parallel computing architectures (CPU and GPU) available on contemporary workstations.

A first set of works focuses on graph drawing problems. Drawing a graph consists of embedding him in a plane or a space. The first contribution in that theme is an edge bundling algorithm named *Winding Roads*. That intuitive, easyly implementable and parallelizable algorithm allows to considerably reduce clutter due to numerous edge crossings in a graph drawing. The second contribution is a method to draw a complete metabolic network. That kind of network models the whole set of biochemical reactions occurring within cells of a living organism. The advantage of the method is to take into account the decomposition of the network into functionnal subsets but also to respect biological drawing conventions.

A second set of works focuses on computer graphics techniques for interactive graph visualisation. The first contribution in that theme is a technique for rendering parametric curves that fully exploits the graphical processor unit. The second contribution is a rendering technique named *Edge splatting* that allows to visualize the bundles densities in an edge bundled layout. The last contribution introduces some techniques for emphasizing sub-graphs of interest in the global context of a graph visualization.

Keywords : Graph visualization, edge bundling, bioinformatic, computer graphics.
Field : Computer Science.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1
Introduction

La *visualisation interactive de graphes* est le domaine de recherche dans lequel s'inscrivent les différentes contributions de cette thèse. Un graphe est un objet mathématique composé d'un ensemble d'éléments ainsi qu'une relation binaire sur cet ensemble. Chaque objet de l'ensemble est appelé *sommet* et chaque élément de la relation binaire est appelé *arête*. Une arête représente une relation entre deux sommets. Les graphes sont étudiés depuis plusieurs siècles. Les fondations de la théorie des graphes ont été mises en place par Euler en 1736 avec l'étude du problème des sept ponts de Königsberg [69] (voir Figure 1.1).

(a) (b) (c)

FIGURE 1.1: Le problème des sept ponts de Königsberg étudié par Euler [69] (source des images : Wikipédia) (a) La ville de Königsberg est construite autour de deux îles situées sur la rivière Pregel. Les deux îles sont reliées entre elles par un pont. Six autres ponts relient les berges de la rivière à l'une des deux îles. Le problème était de déterminer si à partir d'un point de départ aux choix dans la ville il existait une promenade passant une et une seule fois par chaque pont pour revenir à son point de départ. (b) et (c) Euler a montré que ce problème n'avait pas de solution en le modélisant sous forme de graphe. Les sommets de ce graphe représentent les îles et les deux berges de la rivière, les arêtes les ponts. Il démontra qu'il n'existait pas de chemin dans ce graphe associé à la promenade recherchée.

Les graphes sont utilisés dans beaucoup de domaines à des fins de modélisation. La liste ci-dessous présente quelques exemples d'utilisation de graphes en fonction de leur

domaine d'application :

- **Informatique** : Les graphes permettent de représenter un réseau de machines, un système de fichier, l'architecture d'un logiciel, ...
- **Biologie** : Les réseaux d'interaction entre protéines ou entre gènes sont modélisés par des graphes.
- **Chimie** : Une structure moléculaire est modélisée par un graphe.
- **Infographie** : Les maillages de polygones définissant un modèle 3D sont des graphes.
- **Sciences Sociales** : Un réseau social est naturellement modélisable par un graphe.

De nos jours, les améliorations dans les techniques d'acquisition de données permettent de générer des graphes de très grandes tailles (contenant plusieurs milliers voire millions de sommets et d'arêtes).

La visualisation interactive de graphes fait référence à la génération d'images abstraites de ce type de structure de données relationnelles ainsi qu'aux techniques d'interaction facilitant la navigation dans les représentations produites. La Figure 1.2 présente un exemple de visualisation de graphe. Le graphe visualisé correspond au réseau social du club de karaté de Zachary [199]. Chaque sommet représente un membre du club et une arête relie deux membres si une relation d'amitié existe entre eux.

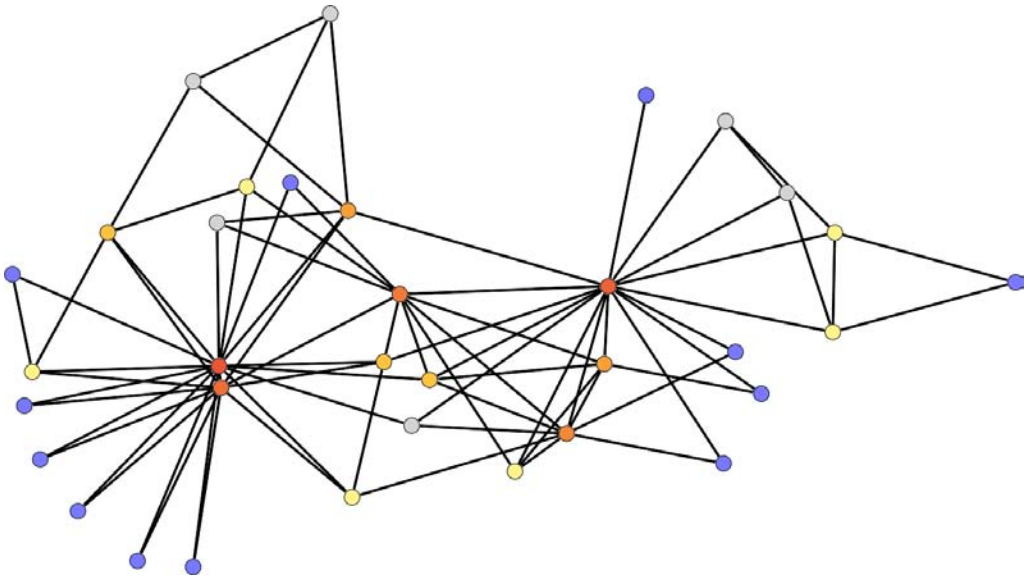


FIGURE 1.2: Exemple de visualisation de graphe. Le graphe visualisé est le réseau social du club de karaté de Zachary [199] (34 sommets/78 arêtes). Chaque sommet représente un membre du club et une arête relie deux membres si une relation d'amitié existe entre eux.

Le besoin de représenter un réseau complexe sur un support visuel afin de faciliter un processus d'analyse propre au domaine d'application fait de la visualisation de graphes un

domaine de recherche très actif depuis plusieurs dizaines d'années. De plus, l'amélioration des performances de calcul des stations de travail permet de nos jours le développement de techniques de dessin et de rendu graphique auparavant trop coûteuses. Ces gains de performances sont notamment dus à l'apparition des processeurs multi-cœurs et à l'utilisation du processeur graphique comme accélérateur. L'exploitation de ces nouvelles ressources de calcul fut d'ailleurs le point de départ de cette thèse.

La visualisation de graphes est un sous domaine de la *Visualisation d'Information*. Nous commencerons donc par introduire ce domaine englobant dans la section 1.1. Nous nous intéresserons ensuite dans la section 1.2 au domaine de la visualisation de graphes et présenterons quelques résultats célèbres. Un grand nombre des contributions de cette thèse sont basées sur l'exploitation du *processeur graphique* afin d'optimiser et générer des visualisations de graphes. Nous introduirons donc également le concept de processeur graphique ainsi que son mode de fonctionnement dans la section 1.3. Nous terminerons par détailler l'organisation globale de ce mémoire dans la section 1.4.

1.1 Visualisation d'Information

L'analyse de données est un processus essentiel à la compréhension de nombreux problèmes et phénomènes de notre société. Vu la taille et la complexité des données pouvant être récoltées de nos jours, il apparaît primordial de disposer de moyens permettant de faciliter l'exploration et l'interprétation de grands jeux de données. Dans [190], Ware explique que 70% des récepteurs de notre cerveau et plus de 40% du cortex sont dédiés au processus de la vision. Tirer profit de ce système visuel afin de faciliter l'analyse de données complexes est le contexte dans lequel s'inscrit la Visualisation d'Information. Contrairement à la Visualisation Scientifique, ce domaine de recherche porte sur l'élaboration de représentations visuelles de données abstraites où aucune référence spatiale ne leur est inhérente. La finalité de la Visualisation d'Information est de faciliter l'extraction de connaissances. Pour une plus ample description de ce domaine, le lecteur peut se référer au livre de Card *et al.* : *Readings in Information Visualization : Using Vision to Think* [35], ainsi qu'au livre de Ward *et al.* : *Interactive Data Visualization : Foundations, Techniques, and Applications* [189].

Le reste de la section est organisée de la façon suivante. Dans la sous-section 1.1.1 nous présentons le *pipeline de visualisation* décrivant les fondements d'un système de Visualisation d'Information. Dans la sous-section 1.1.2 nous présentons des exemples de visualisations pour l'analyse d'un jeu de données multi-dimensionnelles.

1.1.1 Pipeline de visualisation

La Figure 1.3 présente le pipeline de visualisation défini par dos Santos et Brodlie [57]. Ce pipeline permet de modéliser les différentes étapes permettant de passer de données brutes à une image dans un contexte de Visualisation d'Information. Il est dérivé de celui défini par Haber et McNabb pour la Visualisation Scientifique [93]. Il est composé des 5 étapes suivantes :

1. **Analyse des données** : Les données brutes en entrée sont préparées afin d'être visualisées (e.g. application d'un filtre de lissage, interpolation de valeurs manquantes, correction de valeurs erronées, ...). Cette phase est généralement automatique avec peu ou pas d'intervention de l'utilisateur.
2. **Filtrage des données** : Des portions de données sont sélectionnées voire agrégées pour être visualisées. Cette phase est généralement pilotée par l'utilisateur.
3. **Encodage visuel** : Les données sélectionnées sont associées à des primitives géométriques (e.g. points, lignes, polygones) et leurs attributs (e.g. position, couleur, taille). Cette phase est la plus critique pour élaborer des visualisations expressives et efficaces. Une visualisation est dite expressive si et seulement si elle encode uniquement toutes les relations que l'on cherchait à voir [34]. Une visualisation est dite efficace si elle parvient à exploiter les capacités du système visuel humain. Cependant l'efficacité d'une visualisation dépend des capacités de l'utilisateur.
4. **Rendu graphique** : Les données géométriques sont transformées en une image.
5. **Interaction** : l'utilisateur peut influer sur les précédentes étapes décrites au moyen de techniques d'*interaction* afin d'orienter l'analyse des données à visualiser.

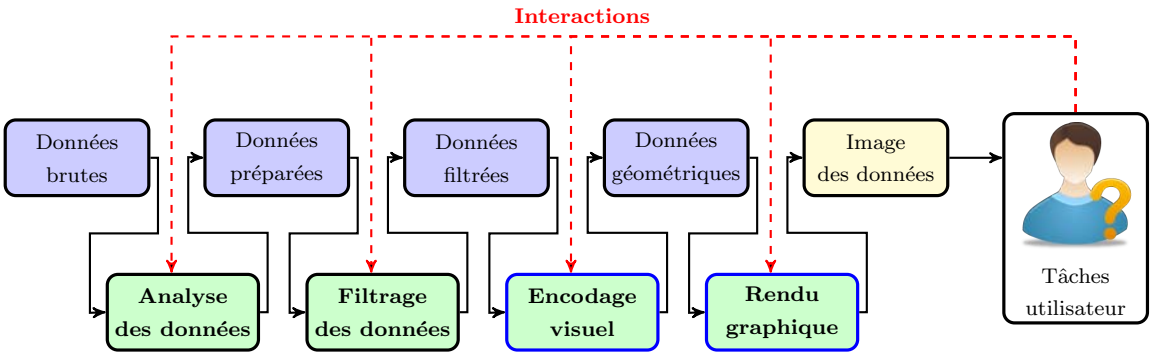


FIGURE 1.3: Pipeline de visualisation, adapté de dos Santos et Brodlie [57]. Les étapes entourées en bleu sont celles dans lesquelles s'inscrivent les différentes contributions de cette thèse.

Un système de Visualisation d'Information se doit de respecter ce pipeline et doit permettre à l'utilisateur d'interagir sur une ou plusieurs étapes. L'exploration de données

par la visualisation est généralement un processus interactif où l'on commence par étudier un jeu de données dans sa globalité avant de raffiner les vues que l'on a dessus. Le célèbre mantra de Shneiderman [168] résume bien ce processus :

« Overview first, zoom and filter, then details-on-demand. »

Un bon système de visualisation doit donc pouvoir :

- présenter une vue complète d'un jeu de données (*overview*)
- offrir la possibilité à ses utilisateurs de pouvoir se focaliser sur des données d'intérêt (*zoom*)
- écarter des données inintéressantes de l'ensemble de celles à visualiser (*filter*)
- donner le détail complet de chaque élément d'un jeu de données quand l'utilisateur en a besoin (*details-on-demand*)

Les contributions de cette thèse portent principalement sur les deux dernières étapes du pipeline de visualisation : l'encodage visuel et le rendu graphique.

1.1.2 Exemples de visualisations

Les Figures 1.4, 1.5, 1.6 et 1.7 présente quelques exemples de visualisations générées à partir du jeu de données *1983 ASA Data Exposition dataset* [150]. Ce jeu de données contient un ensemble d'informations relatives à 406 voitures de l'époque. Le nombre de dimensions attaché à chaque élément du jeu de données est de 8. Le détail de ces dimensions est le suivant :

- le nombre de cylindres dans le moteur (*cylinders*)
- la cylindrée du moteur (*displacement*)
- l'accélération de la voiture
- le nombre de chevaux du moteur (*horsepower*)
- le nombre de *miles* par galon d'essence (*mpg*)
- le poids de la voiture (*weight*)
- l'année du modèle de la voiture
- le pays du constructeur automobile de la voiture

Tous ces exemples de visualisations ont été générés à l'aide de la plateforme de Visualisation d'Information *Tulip* [9, 10, 120], développée au sein de notre équipe de recherche. Les éléments graphiques représentant les voitures ont été colorés en fonction du nombre de cylindres du moteur. La coloration utilisée est la suivante :

- 3 cylindres : bleu clair
- 4 cylindres : bleu foncé
- 5 cylindres : jaune pâle
- 6 cylindres : jaune foncé
- 8 cylindres : rouge

La Figure 1.4 présente une visualisation d'histogrammes de fréquence pour six dimensions. Ils sont présentés sous forme de *Small multiple*, soit une grille de visualisations similaires afin d'en faciliter la comparaison. Ce terme a été popularisé par Edward Tufte [180].

La Figure 1.5 montre une matrice de *scatter plot* permettant d'étudier la corrélation entre deux dimensions. Les *scatter plots* sont des représentations populaires et largement utilisés pour l'analyse de données multi-dimensionnelles. Des versions plus interactives de ce type de visualisation existent comme par exemple le projet *ScatterDice* de Elmqvist *et al.* [66].

La Figure 1.6 présente une visualisation de type *pixel-oriented* sur six dimensions. Ce type de technique de visualisation de données a été élaboré par Keim [115].

La Figure 1.7 présente une visualisation appelée *coordonnés parallèles*, élaborée par Inselberg et Dimsdale [104], représentant six dimensions du jeu de données.

Ces quelques exemples mettent en exergue l'intérêt de la Visualisation d'Information dans un contexte d'analyse de données. L'exploitation de nos capacités visuelles à reconnaître des formes et des couleurs ainsi que de différencier des positions nous permet de dégager rapidement des tendances sur les données représentées.

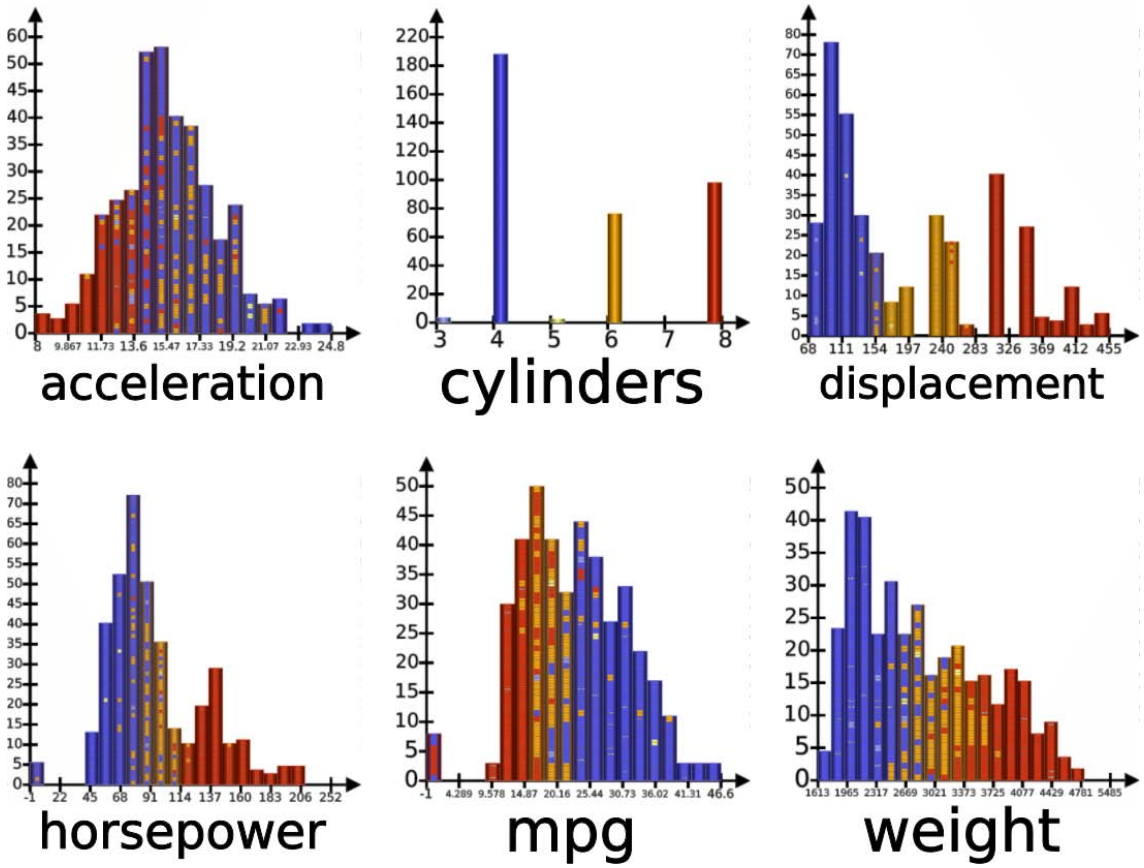


FIGURE 1.4: Histogrammes de fréquence de six dimensions du jeu de données [150]. Les éléments graphiques représentants chaque voiture (des carrés) ont été empilés dans chaque barre matérialisant une classe de l'histogramme. Ce type de visualisation permet de se faire une idée de la distribution des données sur ces dimensions.

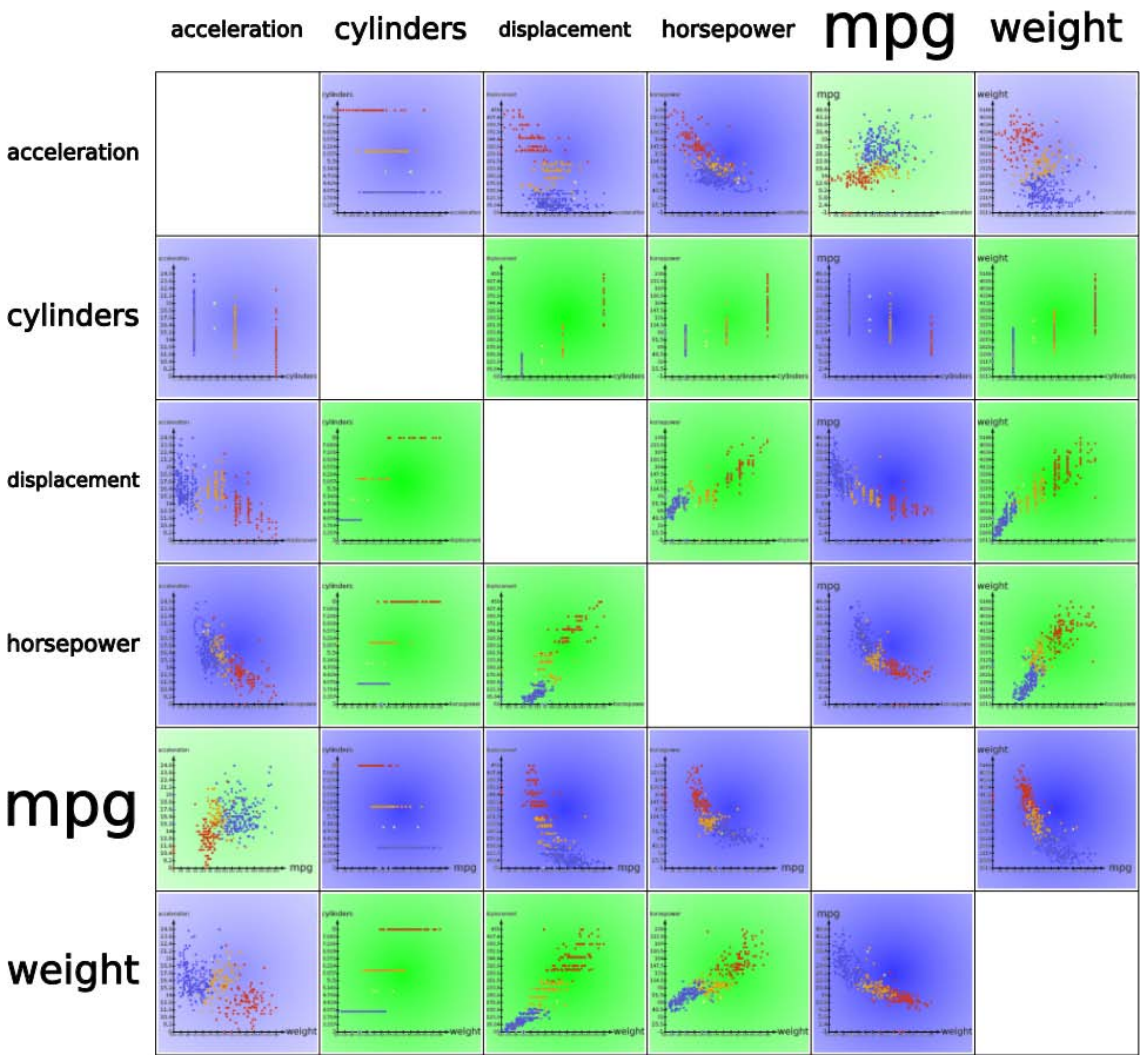


FIGURE 1.5: Matrice de *scatter plot* comparant six dimensions du jeu de données [150] deux à deux. Dans ce type de représentation, chaque élément graphique représentant une voiture est positionné dans le plan suivant deux axes matérialisant l'échelle de valeur de chaque dimension. Les représentations résultantes permettent de suggérer des corrélations positives, négatives ou nulles entre paire de dimensions. Les couleurs de fond des cases de la matrice reflètent le coefficient de corrélation entre les deux dimensions. Plus la couleur de fond est verte, plus les dimensions sont positivement corrélées. Plus la couleur de fond est bleue, plus les dimensions sont négativement corrélées. On peut ainsi observer que le nombre de chevaux et la cylindrée sont très positivement corrélées, on peut en conclure que sur les modèles de voiture dans ce jeu de données, plus la cylindrée est importante, plus le nombre de chevaux l'est également.

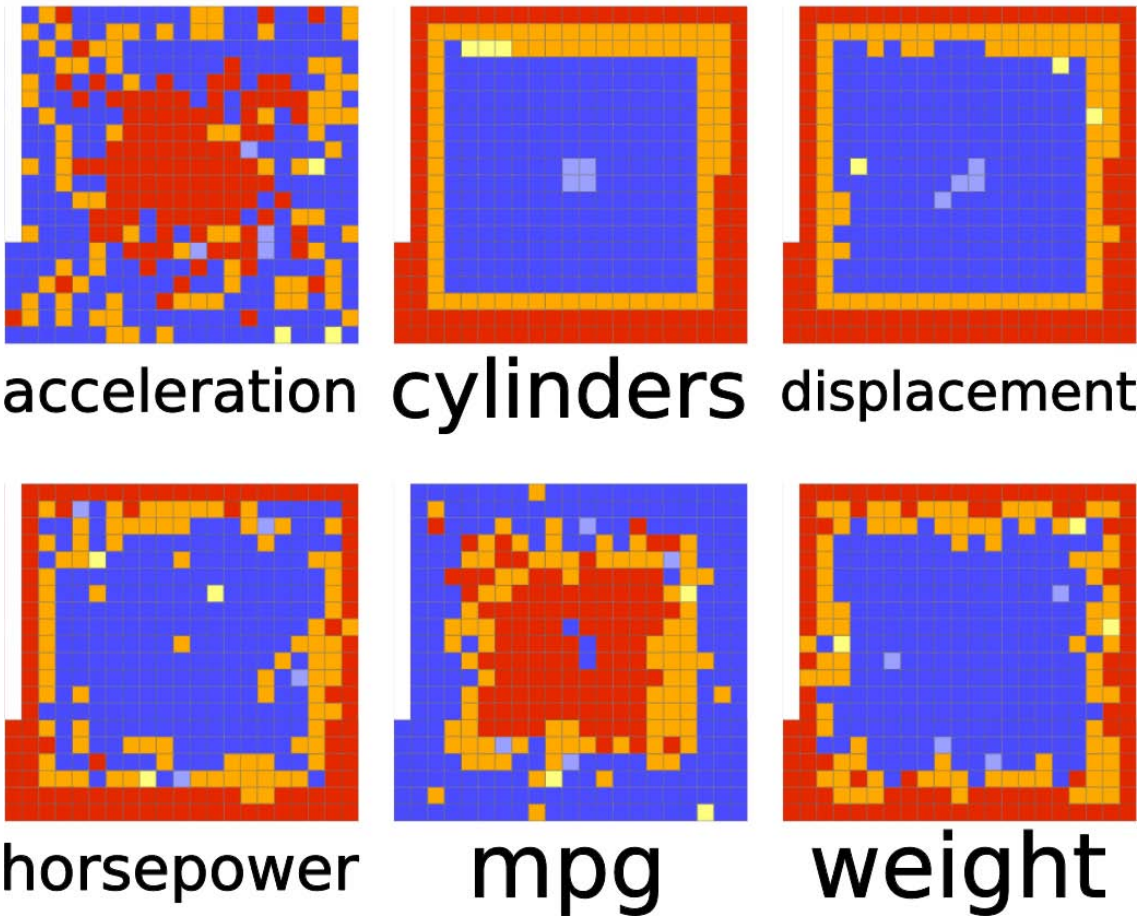


FIGURE 1.6: Visualisation de type *pixel-oriented* sur six dimensions du jeu de données [150]. Ce type de technique de visualisation de données a été élaboré par Keim [115] pour représenter une très grande masse de données en associant à chaque donnée un pixel. Les données sont généralement ordonnées en fonction d'une dimension et positionnées dans une image via l'utilisation d'une courbe de remplissage. Une coloration pertinente des pixels permet alors une analyse de tendance entre les différentes dimensions. Dans notre cas, les pixels représentant les voitures ont été positionnés en utilisant une courbe de type spirale. Ainsi pour chaque dimension les éléments ayant les valeurs les plus faibles se retrouvent au centre et ceux ayant les valeurs les plus élevées sur l'extérieur. Les données sont toujours colorées en fonction du nombre de cylindres dans le moteur des voitures. En comparant les représentations associées à chaque dimension, on peut par exemple observer que plus le moteur des voitures contient de cylindres, plus le nombre de *miles* par galon d'essence diminue.

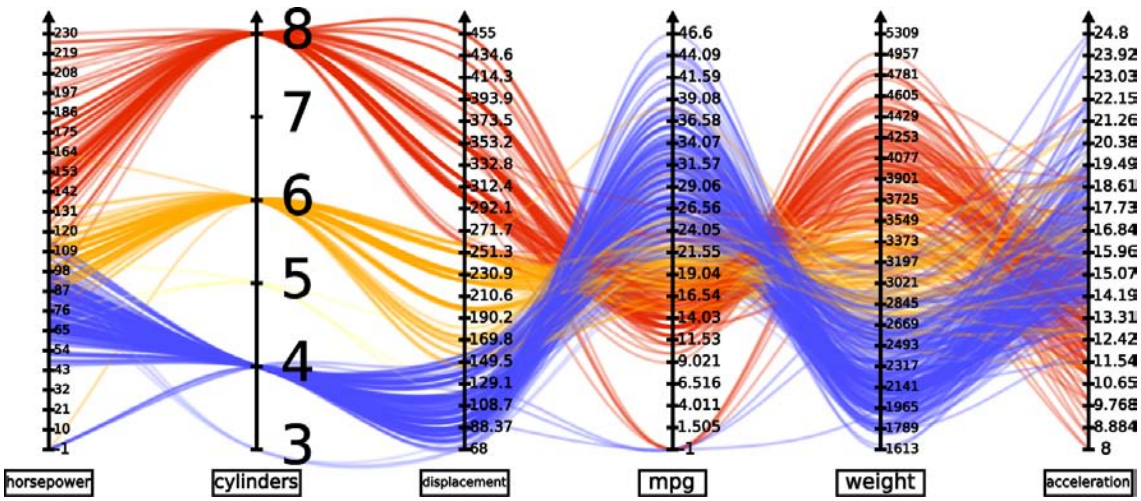


FIGURE 1.7: Visualisation de type *coordonnées parallèles* sur six dimensions du jeu de données [150]. Dans ce type de visualisation, élaboré par Inselberg et Dimsdale [104], chaque dimension est matérialisée par un axe représentant son échelle de valeurs. Les axes sont ensuite placés dans le plan de manière parallèle et sont régulièrement espacés. Une donnée est alors représentée en traçant une ligne brisée/courbe entre les différents axes en fonction des valeurs de ses dimensions. On peut de cette façon observer des corrélations entre plusieurs dimensions. Ici les données sont représentées par des courbes de Catmull-Rom (voir section 5.1.3) interpolant les points de chaque axe. On peut par exemple observer que les dimensions : nombre de chevaux, nombre de cylindres et cylindrée sont très fortement corrélées.

1.2 Visualisation interactive de graphes

La visualisation de graphes, sous-domaine de la Visualisation d'Information, s'attache à produire des représentations visuelles de ce type d'objet et à fournir des techniques d'interaction pour leur exploration. Dans cette section, nous introduisons brièvement ce domaine et nous focalisons sur les méthodes de dessin de graphe et les techniques d'interaction. Pour un état de l'art très détaillé sur la visualisation de graphes, nous recommandons l'étude de Herman *et al.* [97] ainsi que celle plus récente de von Landesberger *et al.* [186]. Toutes les images de cette section ont été générées à l'aide de la plateforme de visualisation Tulip [9, 10, 120].

La transposition du pipeline de visualisation de la Figure 1.3 dans un contexte de visualisation de graphes est la suivante :

1. *Analyse des données* : La première étape du pipeline consiste à peupler une structure de graphe à partir d'une source de données (table, base de données, système de fichiers, ...). Un ensemble d'objets et leurs éventuels attributs sont associés à des sommets et leurs relations à des arêtes.
2. *Filtrage des données* : Cette étape consiste généralement à simplifier le graphe à visualiser en réduisant sa taille, tout en maintenant sa structure principale. Le graphe ainsi modifié permet de générer une représentation plus lisible et compréhensible. Deux types de méthodes peuvent être utilisées pour cela : le filtrage et l'agrégation. Les techniques de filtrage permettent de supprimer certains éléments et sont divisées en deux groupes : stochastique et déterministe. Un filtrage stochastique est basé sur une sélection aléatoire de sommets et d'arêtes du graphe. Un filtrage déterministe se base sur les propriétés et attributs du graphe pour sélectionner des éléments. Des exemples de techniques de filtrage de graphe dans un contexte de visualisation sont donnés par Jia *et al.* dans [106]. L'autre approche pour simplifier un graphe est l'agrégation. Elle consiste à rassembler des sommets et des arêtes en d'uniques autres, réduisant ainsi la taille du graphe et révélant des relations entre des ensembles de sommets. Cette approche est utilisée dans les travaux de Abello *et al.* [1] ainsi que ceux de Archambault *et al.* [6, 8].
3. *Encodage visuel* : La phase principale de cette étape est le dessin du graphe. Dessiner un graphe consiste à positionner ses sommets et ses arêtes dans le plan (dessin en deux dimensions) où dans un espace à n dimensions (généralement $d = 3$). Si cette définition semble simple, les méthodes de dessin de graphe font généralement appel à des techniques complexes de la théorie des graphes [53] et de l'algorithmique géométrique [17]. Nous reviendrons plus longuement sur cette phase dans la sous-section 1.2.1. D'autres propriétés visuelles du graphe sont configurées lors de cette

étape, telles la couleur des éléments ou encore leur taille.

4. *Rendu graphique* : Cette étape consiste à générer l'image du graphe dessiné en utilisant des outils de rendu graphique (e.g. SVG, OpenGL) . La visualisation produite peut être statique ou interactive. Cette phase de rendu peut être plus complexe qu'une simple reproduction du dessin de graphe. Par exemple, la technique *GraphSplatting* de van Liere *et al.* [183] utilise une méthode de rendu particulière permettant de visualiser un graphe comme un champ de densité continu. Au rendu du graphe peut également se greffer le dessin de nouveaux objets résultants de techniques de filtrage ou encore d'interaction. On peut citer comme exemple l'ajout d'enveloppes pour visualiser des sous-ensembles du graphe [43, 124, 125].

L'utilisateur d'un système de visualisation de graphes peut ensuite interagir avec différentes étapes de ce pipeline pour par exemple piloter le filtrage du graphe, naviguer dans le dessin, sélectionner des éléments ou encore se focaliser sur une région particulière du dessin. Nous présenterons quelques techniques d'interaction dans la sous-section 1.2.2.

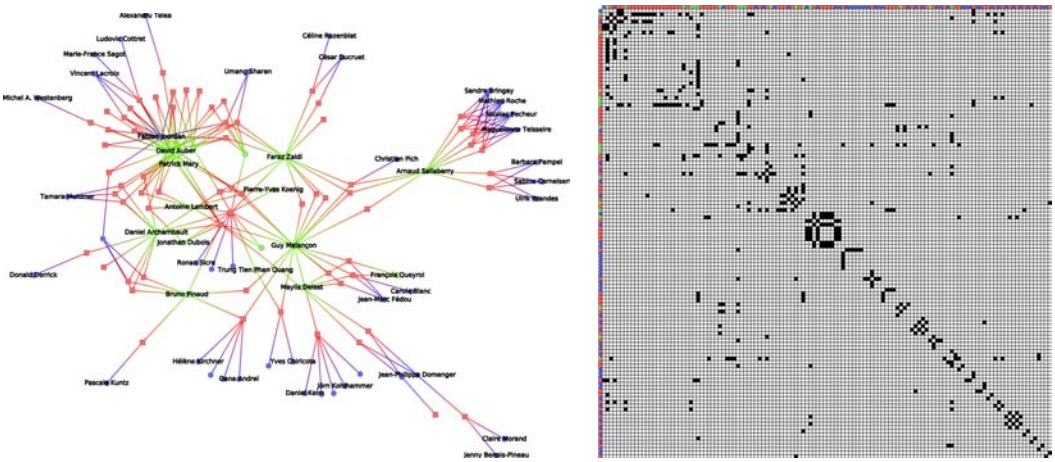
Il existe un très grand nombre de systèmes de visualisation de graphes dans l'ecosphère du logiciel. Les plus connus sont GraphViz [86] (dessins statiques) , Tulip [9, 10, 120] (visualisation interactive de tous types de graphes, développé au sein de notre équipe), Pajek [49] (orienté analyse et visualisation de grands graphes), Cytoscape [165] (orienté visualisation de réseaux biologiques), The InfoVis Toolkit [70] (pas uniquement dédié à la visualisation de graphes), ou encore GUESS [2] (le premier système à avoir intégré une fonctionnalité de script).

1.2.1 Dessin de graphe

Dans cette sous-section, nous présentons un aperçu des techniques de dessin de graphe existantes. Rappelons que le dessin de graphe consiste à plonger visuellement un graphe dans un plan ou un espace, i.e. calculer des coordonnées pour les sommets et éventuellement associer des points de contrôle aux arêtes. C'est un problème qui a toujours existé dû au fait qu'un graphe peut souvent être défini par un dessin. Euler s'est d'ailleurs basé sur un dessin de graphe pour résoudre le fameux problème des sept ponts de Konigsberg [69] (voir Figure 1.1). Deux types majeurs de représentation sont utilisés pour dessiner un graphe :

- **nœud-lien** : Les sommets sont représentés par des polygones et les arêtes par des traits/courbes entre ces polygones (voir Figure 1.8(a)).
- **matrice** : Un graphe à n sommets est représenté comme une *matrice d'adjacence* de taille $n \times n$. La case à la position (i, j) de la matrice est remplie/coloriée si il existe au moins une arête entre le sommet i et le sommet j (voir Figure 1.8(b)).

Une comparaison entre ces deux types de représentations a été faite par Ghoniem *et al.* [88]. D'après leur étude, l'avantage des représentations de type nœud-lien sont leur intuitivité, leur dessin compact ainsi que leur pertinence pour les tâches de suivi de chemin. Elles sont particulièrement adaptées pour les graphes de taille raisonnable (quelques milliers de sommets et d'arêtes) et clairsemés. Les représentations matricielles sont quant à elles particulièrement adaptées pour visualiser un graphe de grande taille (plusieurs dizaines de milliers voire millions de sommets et arêtes). Une réorganisation pertinente des sommets dans les lignes et colonnes de la matrice permet de plus de révéler des sous-graphes très connectés au sein du graphe.



(a) (b)

FIGURE 1.8: Deux représentations visuelles du même graphe. Ce graphe correspond au graphe de co-publications des membres de notre équipe INRIA projet GRAVITE sur la période 2006-2011 (cf. http://gravite.labri.fr/?Publications). Le graphe est composé de sommets représentant les auteurs (en vert les membres de GRAVITE, en bleu les auteurs externes à l'équipe) et les publications (en rouge). Une arête relie un auteur à chacune de ses publications. (a) Représentation nœud-lien obtenue par l'algorithme de dessin FM^3 [94]. (b) Représentation matricielle. Les sommets ont été ordonnés dans les lignes et colonnes de la matrice en fonction du numéro de groupe calculé par l'algorithme de fragmentation de graphes MCL [181]

Les différentes contributions de cette thèse se focaliseront sur les représentations de type nœud-lien. Pour générer ce type de visualisation de graphes, plusieurs critères de lisibilité et d'esthétique doivent être pris en considération. Ces critères incluent la minimisation du nombre de croisements d'arêtes, l'homogénéité de la longueur des arêtes ou encore la minimisation de la taille de la zone de dessin. Pour de plus amples détails sur ce sujet, le lecteur peut se référer à [50] ou aux travaux de Purchase [149, 148]. De nombreux algorithmes de dessin de graphe ont été élaborés depuis plus de 30 ans dont un

grand nombre par la communauté *Graph Drawing*. Dans la suite nous présentons quelques exemples d'algorithmes pour le dessin d'arbres et de graphes généraux.

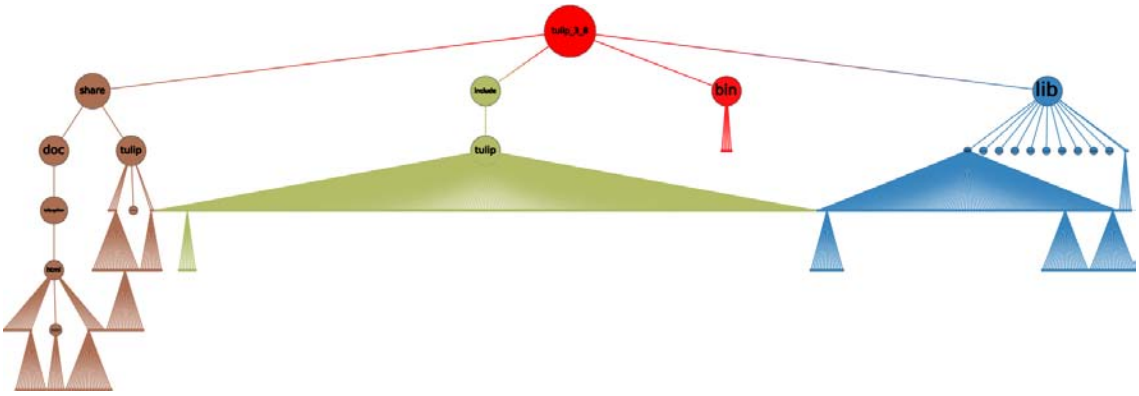
1.2.1.1 Dessin d'arbres

Les arbres sont une classe de graphe particulière (voir Définitions 2.19 et 2.20). On les utilise généralement pour modéliser une relation de hiérarchie sur un ensemble d'objets. Étant plus simples que les graphes généraux, de nombreuses techniques dédiées ont été élaborées pour les dessiner. Nous nous intéresserons aux représentations de type nœud-lien ainsi qu'à celles utilisant une méthode de remplissage d'espace.

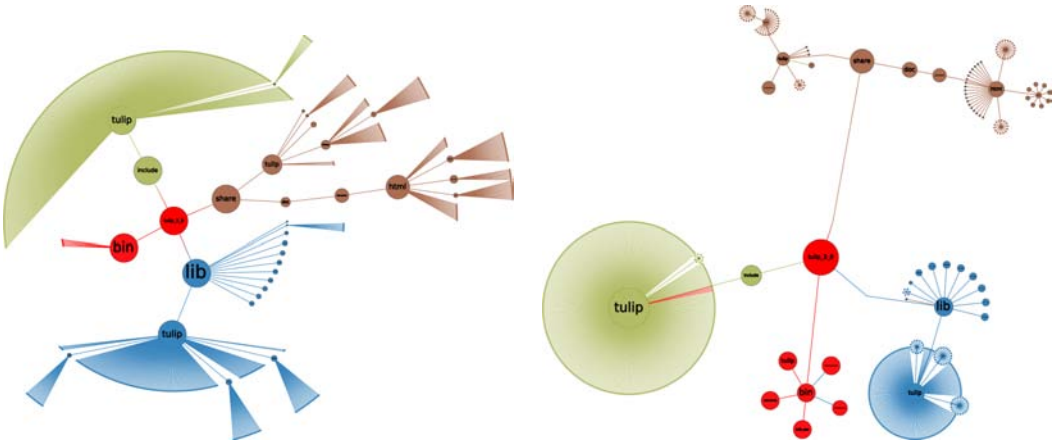
Représentation nœud-lien d'arbres Ce type de représentation utilise des liens pour matérialiser la relation de parenté entre les différents sommets du graphe. Une technique classique est celle élaborée par Reingold et Tilford [151], ensuite améliorée par Walker [188], où les sommets de l'arbre sont positionnés par niveau dans le dessin en fonction de leur profondeur dans l'arbre (voir Figure 1.9(a)). Un autre type de technique, nommé dessin radial, dispose les sommets de l'arbre sur des cercles concentriques en fonction de leur profondeur dans l'arbre [51] (voir Figure 1.9(b)). Un autre exemple de technique est le dessin d'arbre en *ballons* [129]. Dans ce type de dessin, chaque enfant d'un sommet de l'arbre est placé autour d'un cercle centré sur le sommet parent. Un exemple de méthode de dessin de ce type est l'algorithme *Bubble Tree* de Grivet *et al.* [91] (voir Figure 1.9(c)). Les dessins d'arbres en *ballons* peuvent également être obtenus en projetant sur un plan le dessin en trois dimensions obtenu avec l'algorithme *Cone tree* de Robertson *et al.* [153]. Une autre façon de représenter des arbres en trois dimensions sont les dessins hyperboliques, comme par exemple la technique de Munzner [140].

Représentation d'arbres par remplissage d'espace Ce type de méthode de dessin essaie d'utiliser la plus grande zone du support visuel afin de représenter la hiérarchie décrite par un arbre. Pour représenter les relations de parenté entre les sommets, la position des sommets est utilisée. On distingue deux types majeurs de représentation de parenté, soit par imbrication soit par adjacence. Ce type de méthode est généralement utilisé pour visualiser une partition hiérarchique d'un ensemble de données comme par exemple l'arborescence d'un système de fichier. La taille des éléments graphiques associés aux sommets de l'arbre peut être utiliser pour encoder des attributs, comme par exemple la taille d'un fichier.

Les méthodes par imbrication positionnent récursivement les enfants d'un sommet de l'arbre à l'intérieur de la zone de dessin de ce sommet parent. Les exemples les plus



(a)



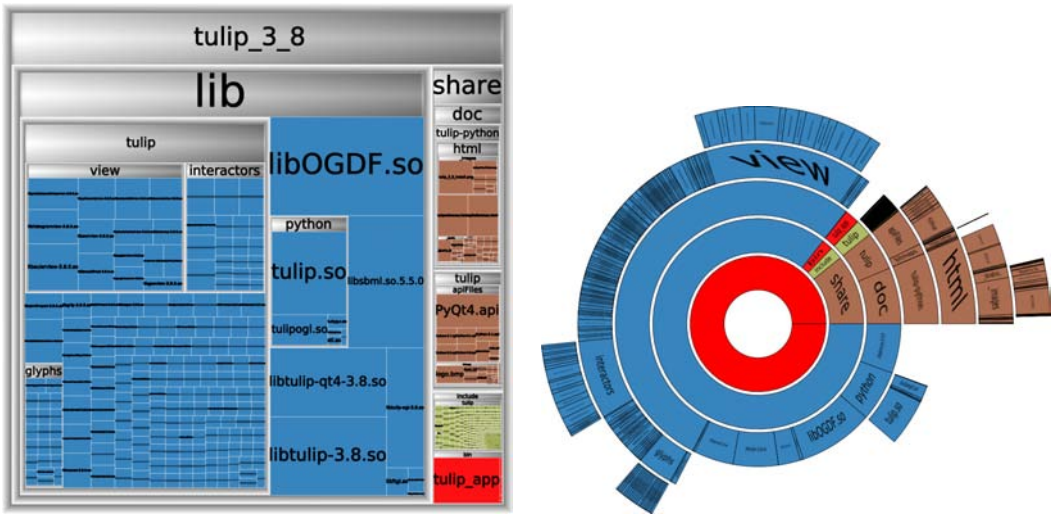
(b) (c)

FIGURE 1.9: Trois représentations nœud-lien du même arbre. Cet arbre représente la hiérarchie de fichiers depuis le répertoire d'installation du logiciel Tulip [9, 10, 120]. (a) L'arbre a été dessiné par l'algorithme *Improved Walker* [188]. (b) L'arbre a été dessiné par une technique de dessin radial [51]. (c) L'arbre a été dessiné par l'algorithme *Bubble Tree* [91], une technique de dessin d'arbre en *ballons*.

répandus de ce type de technique sont les *Treemaps*, où chaque sommet est représenté par une forme rectangulaire, subdivisée en d'autres rectangles reflétant la hiérarchie sous-jacente. La technique originale a été élaborée par Shneiderman [167] puis étendue par Bruls *et al.* [27] de façon à ce que les rectangles représentant les sommets de l'arbre approximent des carrés (voir Figure 1.10(a)). Il existe également des variations de cette méthode comme par exemple les *Voronoi Treemaps* de Balzer *et al.* [14] où des polygones plus généraux sont utilisés à la place de rectangles (voir également le récent travail de Nocaj et Brandes [142] pour calculer efficacement ce genre de représentation).

Les méthodes par adjacence quant à elles positionnent les sommets enfants à côté de leur sommet parent. Un exemple célèbre est la méthode *SunBurst* [174] de Stasko et

Zhang positionnant chaque niveau de l'arbre sur des couches circulaires du dessin (voir Figure 1.10(b)).



(a) (b)

FIGURE 1.10: Deux représentations par remplissage d'espace du même arbre. Cet arbre représente la hiérarchie de fichiers depuis le répertoire d'installation du logiciel Tulip. La taille des feuilles de l'arbre a été configurée pour être proportionnelle à la taille des fichiers qu'elles représentent. (a) L'arbre a été dessiné par l'algorithme *Squarified Tree Maps* [27]. (b) L'arbre a été dessiné par l'algorithme *SunBurst* [174].

1.2.1.2 Dessin de graphes généraux

Un très grand nombre de techniques de dessin existent pour les graphes généraux. Nous présentons quelques algorithmes célèbres en fonction de leur méthode de placement de sommets.

Dessin par modèle de forces Ce type de méthode assimile le graphe à un système physique (voir Figure 1.11). Les sommets du graphe sont considérés comme des particules exerçant des forces de répulsion entre elles. Les arêtes sont considérées comme des ressorts exerçant une force d'attraction entre les sommets connectés. L'évolution de ce système est ensuite simulée jusqu'à atteindre un état d'équilibre correspondant au dessin final. Eades fut le premier à proposer une telle approche [63]. Sa méthode a depuis été revisitée et améliorée. Parmi les algorithmes existants, on peut citer celui de Kamada et Kawai [110] (voir Figure 1.12(a)), celui de Fruchterman et Reingold [78] (voir Figure 1.12(b)) et *GEM* de Frick *et al.* [77] (voir Figure 1.12(c)). Ces méthodes emploient des modèles physiques différents débouchant sur des algorithmes de différentes complexités et produisant des dessins

de qualité variable. Les méthodes par modèle de force produisent généralement des dessins assez équilibrés avec un nombre de croisements d'arêtes minimisé. Une autre approche de dessin par modèle de force est celle de Bertault [19], depuis améliorée par Simonetto *et al.* [171]. Elle permet d'améliorer un dessin de graphe existant tout en ne créant pas et ne défaisant pas de croisements d'arêtes. Ces méthodes ne passent généralement pas à l'échelle pour de grands graphes en raison de leur complexité.

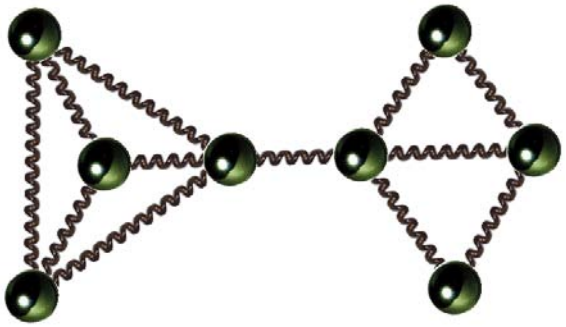
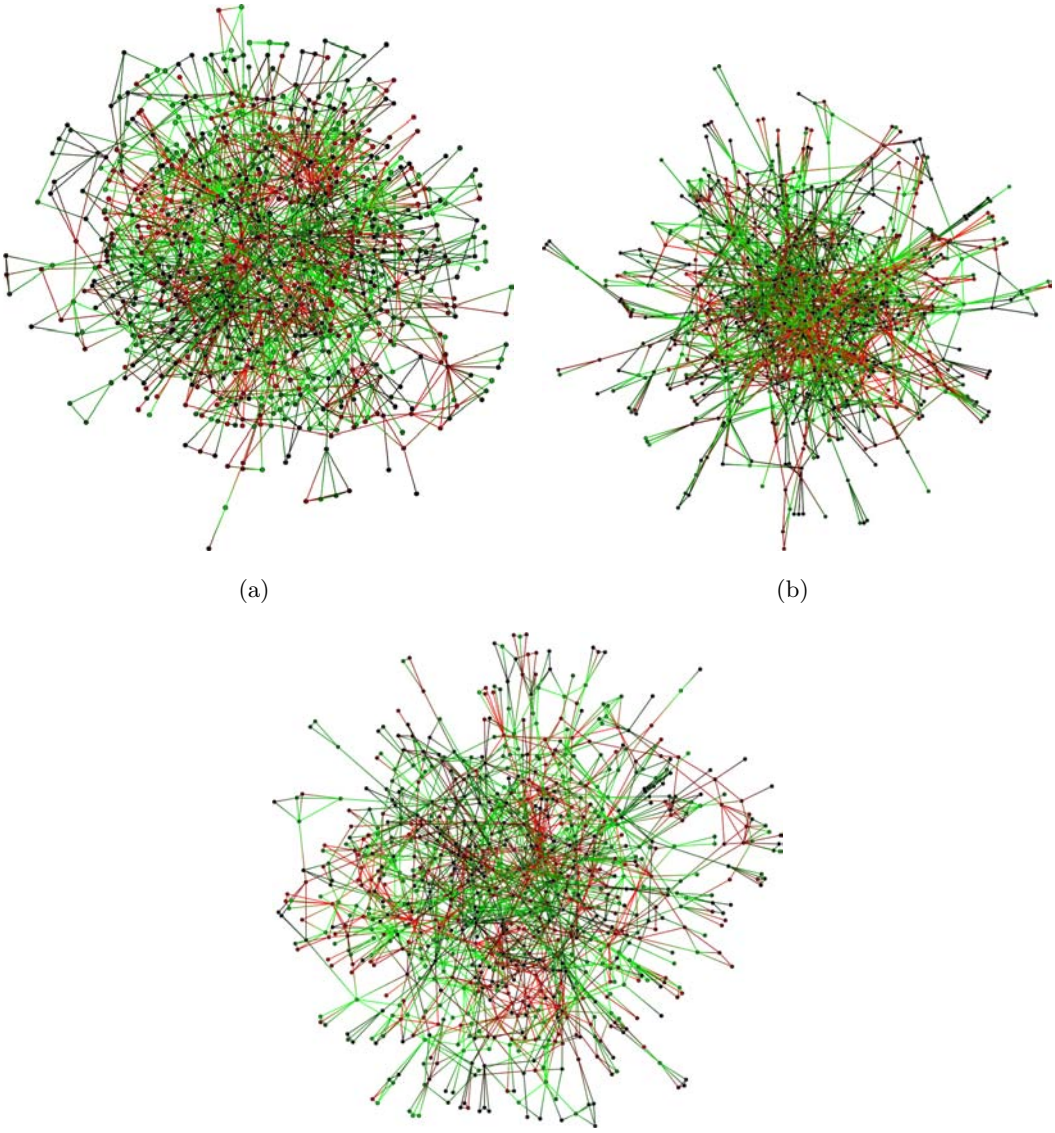


FIGURE 1.11: Principe des méthodes de dessin par modèle de force. Les sommets du graphe sont associés à des particules et les arêtes à des ressorts. Ce système physique est défini de façon à ce que les sommets non connectés se repoussent et ceux connectés s'attirent. L'évolution de ce système physique est simulée jusqu'à un état d'équilibre pour obtenir le dessin final.

Dessin par contrainte Les méthodes de ce type étendent l'approche par modèle de force en rajoutant des contraintes sur la position des sommets. Ces contraintes peuvent inclure un alignement horizontal ou vertical des sommets, le non chevauchement entre sommets ou encore l'orientation des arêtes. Un exemple classique de dessin par contrainte sont ceux orthogonaux où les arêtes sont uniquement dessinées comme des traits horizontaux ou verticaux. Des exemples d'algorithmes de dessin de ce type sont ceux élaborés par Dwyer *et al.* [59, 60, 61].

Dessin multi-niveaux Ces techniques de dessin reposent sur une décomposition hiérarchique d'un graphe en sous-graphes imbriqués. Le dessin du graphe est alors effectué niveau par niveau de la hiérarchie. Ces méthodes peuvent utiliser une approche par modèle de force pour dessiner chaque niveau. Le dessin obtenu au niveau précédent est conservé lors du calcul du dessin du prochain niveau. Suivant les approches, le premier niveau à être dessiné est soit la racine de la hiérarchie (approche descendante ou *top-down*), soit ses feuilles (approche ascendante ou *bottom-up*). Ces méthodes sont généralement plus rapides que les méthodes traditionnelles par modèle de forces et certaines peuvent dessiner des graphes contenant plusieurs millions de sommets. Comme exemple d'algorithme de dessin

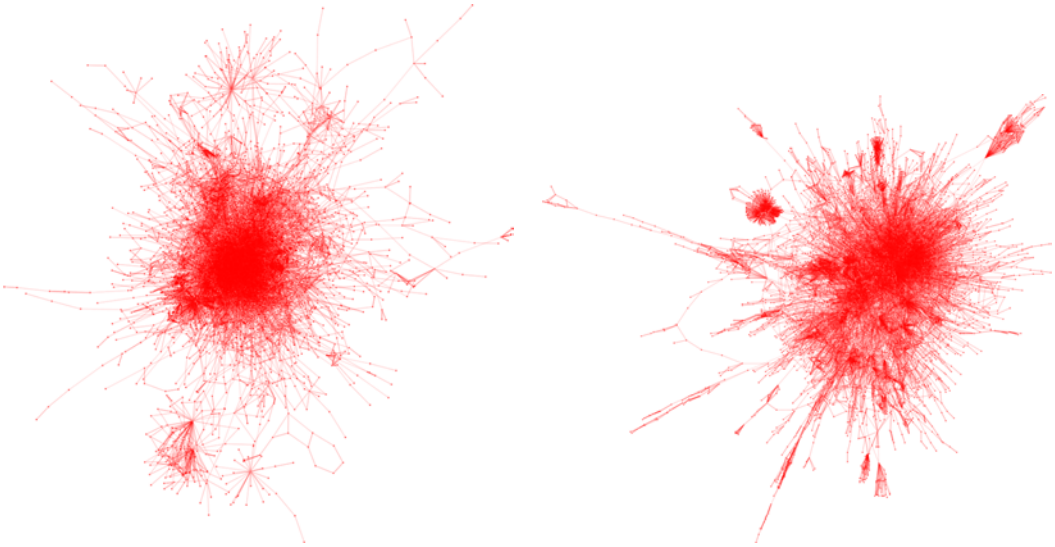


(a) (b)

(c)

FIGURE 1.12: Trois dessins par modèle de forces du même graphe. Ce graphe correspond à un réseau de joueurs de pokers (832 sommets / 2127 arêtes). Les sommets correspondent à des joueurs et deux joueurs sont connectés quand l'un d'eux a donné de l'argent à l'autre. (a) Le graphe a été dessiné en utilisant l'algorithme de Kamada et Kawai [110]. (b) Le graphe a été dessiné en utilisant l'algorithme de Fruchterman et Reingold [78]. (c) Le graphe a été dessiné en utilisant l'algorithme GEM [77].

multi-niveaux, on peut citer *GRIP* de Gajer et Kobourov [82] (voir Figure 1.13(a)), FM^3 de Hachul and Jünger [94] (voir Figure 1.13(b)) ou encore *TopoLayout* de Archambault *et al.* [7].



(a) (b)

FIGURE 1.13: Deux représentations du même graphe obtenues avec des algorithmes de dessin multi-niveaux (par modèle de forces). Ce graphe est un graphe d'appel de fonctions dans un logiciel complexe (7697 sommets / 18007 arêtes). Un sommet représente une fonction et deux fonctions sont connectées si l'une d'entre elles appelle l'autre. (a) Le graphe a été dessiné avec l'algorithme *GRIP* [82]. (b) Le graphe a été dessiné avec l'algorithme FM^3 [94].

Dessin hiérarchique Ces approches positionnent les sommets du graphe sur des niveaux horizontaux parallèles. Elles sont principalement utilisées pour les graphes orientés et dérivent de la méthode proposée par Sugiyama [175] (voir Figure 1.14). Cette dernière est composée de 4 étapes :

1. suppression des cycles dans le graphe
2. assignation des sommets dans des niveaux
3. réduction du nombre de croisements d'arêtes
4. assignation de coordonnées aux sommets.

Parmi les exemples d'algorithmes de ce type, on peut citer celui de Gansner *et al.* [85], celui de Buchheim *et al.* [28] ou encore celui de Auber [9].

Dessin des arêtes L'un des problèmes majeurs des représentations de type nœud-lien de grands graphes est l'occlusion visuelle entraînée par la grande quantité d'arêtes.

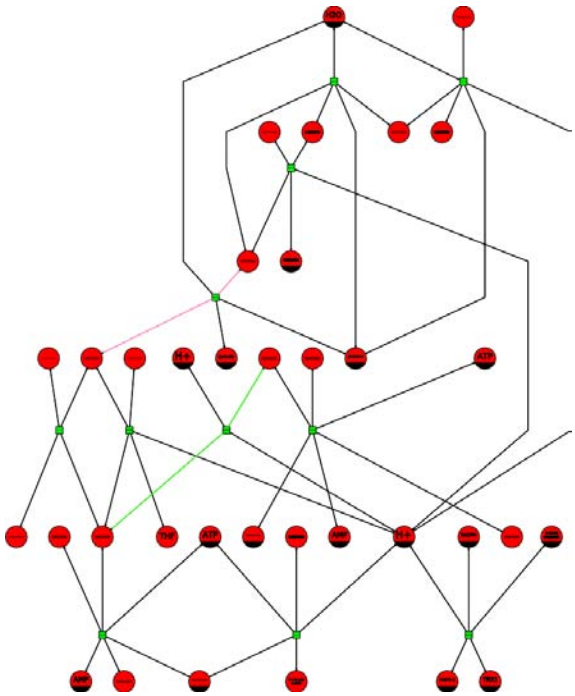
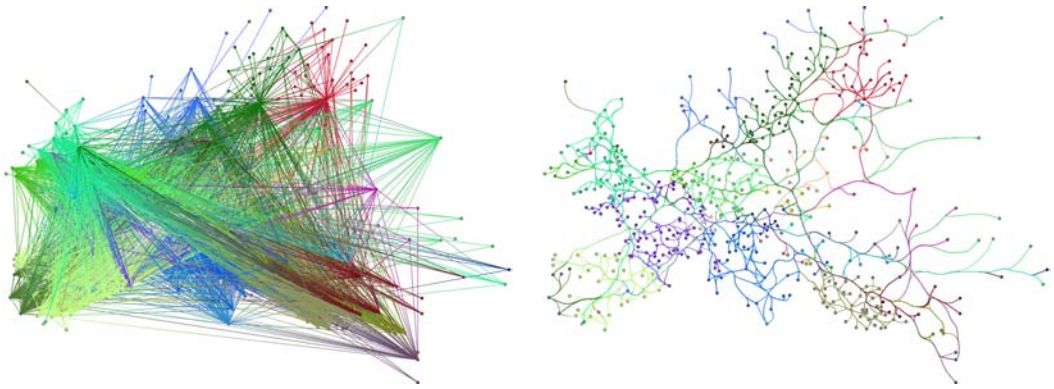


FIGURE 1.14: Exemple de dessin hiérarchique d'un graphe. Ce graphe représente la voie métabolique *Selenocompound metabolism* de l'organisme *Yarrowia lipolytica* (une levure). Une voie métabolique représente un ensemble de réactions biochimiques se produisant dans les cellules d'un organisme et effectuant une fonction biologique particulière (voir chapitre 4). Le graphe a été dessiné en utilisant l'algorithme hiérarchique de Sugiyama [175].

Comme les arêtes sont dessinées comme des segments, la lisibilité du dessin peut en pâtir. Des techniques visant à regrouper les arêtes en faisceaux (en anglais, *edge bundling*) ont alors été proposées pour tenter d'adresser ce problème. Le principe consiste à calculer un ensemble de points de contrôle qui définiront la nouvelle forme de chaque arête. Les arêtes sont alors généralement dessinées en utilisant des courbes paramétriques. La Figure 1.15 montre un exemple de résultat obtenu avec l'une de ces techniques. L'une des contributions de cette thèse porte sur ce type de méthode. Nous reviendrons donc très longuement sur le sujet dans le chapitre 3.

1.2.2 Techniques d'interaction

Générer un dessin de graphe n'est qu'une première étape pour la mise en place de visualisations à des fins d'analyse. Les utilisateurs ne veulent pas uniquement disposer d'un dessin statique, ils veulent pouvoir l'explorer et manipuler les données représentées de manière interactive. Des techniques d'interaction ont donc été élaborées pour permettre aux utilisateurs de résoudre un ensemble de tâches relatives à l'analyse d'un graphe. Ces



(a) (b)

FIGURE 1.15: Exemple d'application d'une méthode de regroupement d'arêtes (*edge bundling*) sur un graphe modélisant des connexions entre des villes européennes. (a) Dessin original. (b) Dessin après application de la méthode.

tâches peuvent être de nature variable comme par exemple celles basées sur la topologie du graphe (voir Définition 2.3) ou encore celles basées sur les attributs du graphe. Les tâches basées sur la topologie incluent l'identification de l'adjacence d'un sommet ou encore déterminer si deux sommets sont connectés. Les tâches basées sur les attributs incluent par exemple la recherche de sommets/arêtes ayant une valeur particulière. Une taxonomie exhaustive des tâches liées à la visualisation de graphes a été faite par Lee *et al.* [128], elle même basée sur celle établie par Amar *et al.* [5] pour la Visualisation d'Information. Dans la suite, nous présentons quelques unes de ces techniques d'interaction.

1.2.2.1 Interactions pour la navigation

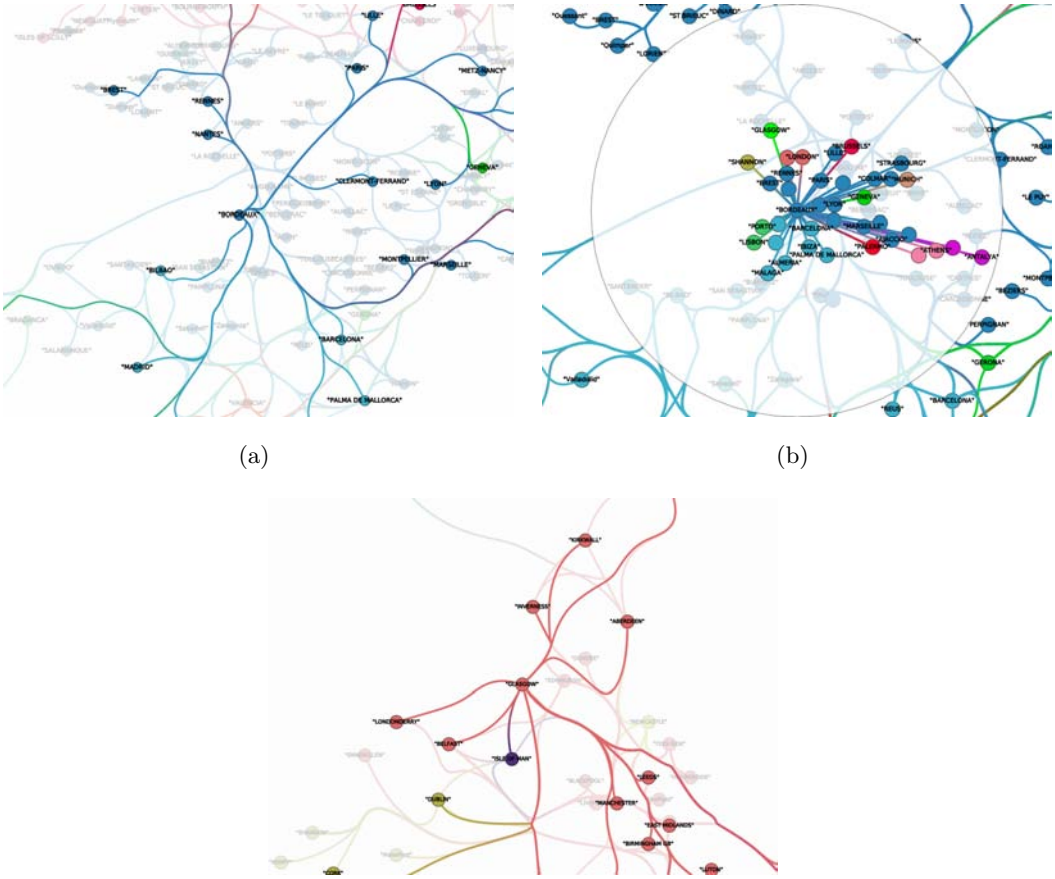
Le but de ces techniques est de faciliter l'exploration de la représentation visuelle d'un graphe. Les plus fondamentales sont celles permettant de translater la vue dans n'importe quelle direction ainsi que de changer le niveau de zoom courant. Ces opérations basiques mais essentielles pour une exploration interactive des données représentées se retrouvent dans un grand nombre de systèmes de Visualisation d'Information. Il existe d'ailleurs un algorithme élaboré par Van Wijk et Nuij [184] permettant de réaliser une animation de translation/zoom fluide et optimale entre deux régions d'un dessin. Dans un contexte de visualisation de graphes il existe aussi une technique de navigation consistant à se déplacer le long du dessin d'une arête. Dans [139], Moscovich *et al.* présentent une version améliorée de cette technique appelée *Link Sliding*.

Une autre méthode pour explorer un dessin de graphe est de naviguer de proche en proche. En effet, deux sommets connectés dans un graphe ne sont pas forcément proches

dans son dessin. Ainsi, la distance entre deux sommets dans un graphe n'est pas toujours reflétée par leur distance euclidienne dans son dessin. C'est l'un des défis que tous les algorithmes de dessin de graphe tentent de relever. La technique d'interaction *Bring & Go*, introduite par Tominski *et al* [179], depuis améliorée par Moscovich *et al* [139], permet de résoudre ce paradoxe. L'opération *Bring*, enclenchée après avoir sélectionné un sommet, rapproche temporairement les voisins du sommet de sa position dans la vue. Elle permet ainsi de résoudre des situations où l'algorithme de dessin a échoué. Les Figure 1.16(a) et 1.16(b) illustrent cette situation, le passage de l'étape 1 à l'étape 2 se faisant via une animation. Les sommets voisins sont positionnés de telle sorte que leur direction et leur distance relative par rapport au sommet sélectionné est préservée. Une fois que les voisins ont été repositionnés autour du sommet d'intérêt, l'opération *Go* laisse l'utilisateur décider d'une nouvelle direction vers laquelle se déplacer en sélectionnant un voisin. Après avoir choisi un des voisins, une animation de zoom et de translation est effectuée jusqu'à ce que la visualisation soit recentrée sur le voisin cible (voir Figure 1.2.2.1). Moscovich *et al* [139] ont réalisé une évaluation empirique de cette technique d'interaction. Leurs résultats montrent que cette technique est jugée facile à apprendre et plaisante à utiliser. Cette technique peut également être classée dans les techniques d'interaction dites Focus+Contexte.

1.2.2.2 Interactions Focus+Contexte

Dans une visualisation de graphe, l'utilisation d'opérations de zoom pour se concentrer sur une région d'intérêt du dessin se fait au détriment de la perte d'informations contextuelles. Un ensemble de techniques permettant à l'utilisateur de se concentrer sur des détails de la représentation tout en conservant le contexte global ont donc été mis au point. Ce type de techniques est dénommé par le terme *Focus+Contexte*. Des exemples de ce type de technique sont la loupe et le *Fisheye*. La loupe permet de présenter une vue détaillée d'une petite région du dessin par dessus celui lié au niveau de zoom courant (voir Figure 1.17(a)). Une généralisation de ce type de techniques a été faite par Bier *et al.* [20]. Le *Fisheye* permet également d'obtenir des détails sur une région du dessin via l'utilisation d'une déformation continue de la représentation. Ce concept a été généralisé par Furnas [80] et expérimenté pour la première fois sur une visualisation de graphe par Sarkar et Brown [159]. La référence dans ce domaine reste les travaux de Carpendale *et al* [36, 37, 38] qui ont longuement étudiés les différentes méthodes de déformation ainsi que leurs combinaisons pour les interactions de type Focus+Contexte sur tout type de visualisation. Il se présente généralement sous la forme d'une lentille circulaire. Dans un cercle de petite taille autour du centre de la lentille, le grossissement de la représentation est très important. Ce grossissement décroit ensuite continuellement jusqu'à la périphérie de la lentille. Un exemple d'interaction de type *Fisheye* est présenté à la Figure 1.17(b).



(a) (b)

(c)

FIGURE 1.16: Illustration de l'interaction *Bring & Go*. (a) *Bring* (étape 1) – La sélection d'un nœud (ici "Bordeaux") estompe les éléments du graphe ne faisant pas partie de son voisinage. (b) *Bring* (étape 2) – Les sommets voisins sont rapprochés du nœud sélectionné. La transition entre les deux étapes est animée. (c) *Go* – Après avoir sélectionné un voisin (ici le sommet "Glasgow", matérialisé en vert dans la Figure 1.16(b)), une animation de translation et de zoom change le focus vers un nouveau voisinage.

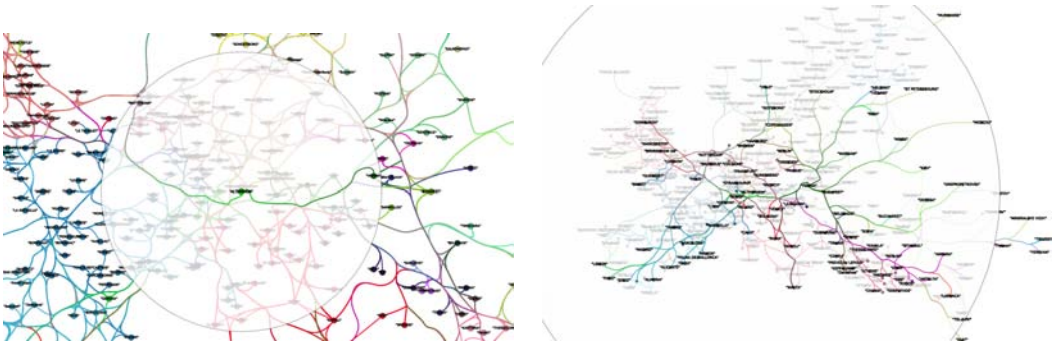
Ces techniques peuvent être utilisées pour obtenir une estimation du degré des sommets. Elles permettent également de se faire une idée de l'organisation spatiale des voisinages.

Un autre type de techniques Focus+Contexte sont celles permettant de mettre en évidence le voisinage d'un sommet d'intérêt. Plusieurs techniques ont été élaborées pour cela. Dans [95], Heer et Boyd modifient interactivement les couleurs des sommets du graphe en fonction du sommet sélectionné. Les sommets atteignables depuis ce sommet sont colorés en fonction de leur distance au sommet d'intérêt en fonction d'une échelle de couleur tandis que ceux non atteignables voient leurs couleurs désaturées. Dans le logiciel de visualisation de graphes Tulip [9, 10, 120], une autre stratégie est utilisée. Un cercle transparent, centré sur le sommet d'intérêt, est affiché par dessus la visualisation pour

(a) (b)

FIGURE 1.17: Techniques d'interaction loupe et *Fisheye*. (a) La loupe montre une vue détaillée sur une région locale du dessin. (b) Le *Fisheye* applique une déformation continue sur une petite région du dessin pour inspection locale.

mettre en exergue son voisinage. Les sommets non connectés au sommet d'intérêt sont temporairement estompés par le cercle tandis que ceux connectés (et les arêtes associées) sont dessinés par dessus (voir Figure 1.18(a)). Le rayon du cercle transparent est calculé de telle sorte que tous les voisins immédiats du sommet d'intérêt dans le dessin de graphe soient contenus dans le cercle. L'utilisateur peut ensuite choisir de visualiser le voisinage d'un sommet en fonction de différentes distances. La taille du cercle transparent s'ajuste alors en conséquence (voir Figure 1.18(b)).

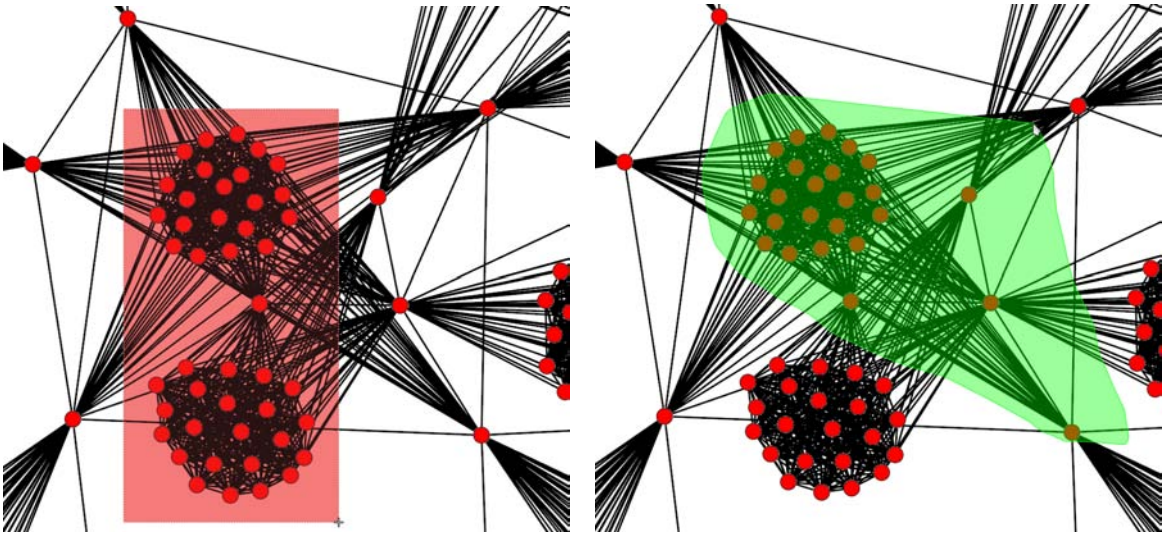


(a) (b)

FIGURE 1.18: Illustration de l'interaction de mise en évidence du voisinage implémentée dans le logiciel *Tulip* [9, 10, 120]. (a) Mise en évidence du voisinage direct – la sélection d'un sommet met en exergue ses voisins et estompe les autres éléments du graphe. (b) En utilisant la molette de la souris, le voisinage est étendu aux sommets dont la distance est supérieure à 1.

1.2.2.3 Interactions sur la représentation

Ce type de techniques permet un ajustement de la représentation visuelle du graphe et de ses paramètres. La technique la plus basique consiste à sélectionner un ensemble d'éléments et de les mettre en exergue dans la visualisation. Plusieurs stratégies peuvent être employées comme utiliser une couleur particulière pour les éléments sélectionnés ou encore l'utilisation d'enveloppes convexes ou concaves [95, 43]. La Figure 1.19 présente deux techniques classiques de sélection d'éléments dans une visualisation de graphe.



(a) (b)

FIGURE 1.19: Deux techniques d'interaction pour sélectionner des éléments dans la visualisation d'un graphe. Deux stratégies de sélection sont possibles. Soit tous les éléments situés dans la zone de sélection sont sélectionnés, soit uniquement les sous-graphes induits par les sommets sélectionnés. Les éléments sélectionnés seront ensuite mis en exergue dans la visualisation par un changement d'encodage visuel. (a) Sélection par boite : les éléments situés sous le rectangle seront sélectionnés. (b) Sélection par lasso : cette technique permet d'effectuer des sélections plus fines, pouvant être impossibles à effectuer via une sélection par boite.

Un autre type de méthode sont celles effectuant du zoom sémantique. Elles consistent à augmenter le niveau de détail de la représentation des données (en plus de celui de l'encodage visuel) lors d'une opération de zoom. Elles sont généralement utilisées sur des graphes où des ensembles d'éléments ont été agrégés selon plusieurs niveaux. L'opération de zoom sémantique dans ce cas consiste à représenter le niveau d'agrégation associé au niveau de zoom demandé. Un exemple de technique est celle utilisée par Elmqvist *et al.* [65] afin de visualiser une représentation matricielle d'un très grand graphe. Les cases de la matrice sont agrégées en groupe de quatre selon plusieurs niveaux. Un zoom sémantique

permet alors de descendre/remonter dans ces différents niveaux d'agrégation.

Il existe également des techniques permettant de modifier le dessin d'un graphe où son type de représentation interactivement. Dans [135], McGuffin et Jurisica présentent un ensemble d'outils de sélection et de manipulation de sous-graphes. Leur technique d'interaction permet entre autre d'appliquer un algorithme de dessin sur un sous-graphe sélectionné de manière interactive. Un autre exemple est la technique de Henry *et al.* [96] permettant de passer d'une représentation de type nœud-lien vers une représentation matricielle d'un graphe et vice-versa. Ce changement de représentation est visualisé à l'aide d'une animation.

1.2.2.4 Interactions avec les données

Dans ce type de techniques, l'utilisateur modifie interactivement les données couramment visualisées. Dans un contexte de visualisation de graphes, cela consiste à modifier la structure du graphe ou la valeur des attributs de ces éléments. Ces modifications de données peuvent être réalisées via des interactions sur la représentation (e.g. suppression d'un sommet en cliquant dessus), des interfaces graphiques d'édition dédiées ou une exécution de scripts. Les logiciels GUESS [2] et Tulip [9, 10, 120] fournissent par exemple une interface d'édition et d'exécution de scripts pour modifier interactivement les graphes visualisés.

Un autre type de techniques influant sur les données du graphe à visualiser sont celles permettant de modifier de manière interactive une décomposition hiérarchique d'un graphe. Une telle décomposition permet de visualiser une version simplifiée d'un grand graphe par des méthodes d'agrégation. Un exemple de ce type de technique peut être trouvé dans le système de visualisation de hiérarchie de sous-graphes *GrouseFlocks* [8]. Ce système permet de modifier interactivement la hiérarchie de sous-graphes soit en interagissant directement sur la visualisation soit via une interface graphique dédiée.

1.3 Processeur graphique et pipeline de rendu

Dans cette section, nous présenterons brièvement le concept de processeur graphique ainsi que son mode de fonctionnement.

1.3.1 Définition d'un processeur graphique

Un processeur graphique, souvent désigné par le terme GPU (de l'anglais *Graphics Processing Unit*), est un circuit électronique spécialisé dédié au traitement des données

graphiques. Il a été conçu pour rapidement manipuler et modifier une mémoire afin d'accélérer la construction d'images dans un tampon de trames (en anglais *frame buffer*) destinées à être envoyées vers un périphérique d'affichage. De nos jours, on trouve des processeurs graphiques dans les systèmes embarqués, les téléphones portables, les ordinateurs personnels, les stations de travail ou encore les consoles de jeux. Le terme a été popularisé par l'entreprise Nvidia en 1999, après la mise sur le marché de la carte graphique GeForce 256, qui contenait un processeur graphique capable de traiter un minimum de 10 millions de polygones par seconde.

1.3.2 Fonctionnement du pipeline de rendu

Cette section vise à introduire brièvement le fonctionnement d'un processeur graphique. Le lecteur pourra trouver des informations plus détaillées sur les pipelines graphiques traditionnels en consultant le livre de Foley *et al.* [74] : *Computer graphics : principles and practice*, ou le guide officiel de la programmation en OpenGL [169].

Avant de décrire le pipeline de rendu du processeur graphique, il est important de commencer par introduire les différents systèmes de coordonnées utilisés lors du rendu d'une scène 3D. Il sont au nombre de quatre :

- **L'espace objet :** système de coordonnées 3D local à un objet géométrique
- **L'espace monde :** système de coordonnées 3D où l'ensemble des objets de la scène sont positionnés. Pour transformer la position d'un sommet de l'espace de son objet vers une position désirée dans l'espace du monde, on utilise une matrice de transformation appelée *matrice monde*. Le sommet est multiplié par cette matrice qui le transforme en modifiant sa position et son orientation.
- **L'espace caméra :** système de coordonnées 3D défini par la position et l'orientation d'une caméra virtuelle observant la scène 3D. Tous les sommets de la scène sont transformés de l'espace du monde vers l'espace de la caméra en les multipliant par une matrice appelée *matrice vue*. L'apparence résultante de la scène est alors telle que si on l'observait à travers la caméra.
- **L'espace écran :** système de coordonnées 2D défini par l'écran qui affichera la scène. Les sommets dans l'espace de la caméra sont transformés dans cet espace en les multipliant par une *matrice de projection*. Cette projection de l'espace de la caméra vers l'espace de l'écran peut être de type orthographique ou perspective. Avec une projection en perspective, les objets éloignés de la caméra apparaîtront plus petits que ceux qui lui sont proches.

Le passage d'un système de coordonnées à l'autre est réversible par l'utilisation des matrices inverses de celles utilisées. Par exemple, pour passer de l'espace écran à l'espace de la caméra, on multiplie un sommet en coordonnées écran par l'inverse de la matrice de projection. La Figure 1.20 illustre ces quatre systèmes de coordonnées ainsi que le passage de l'un à l'autre.

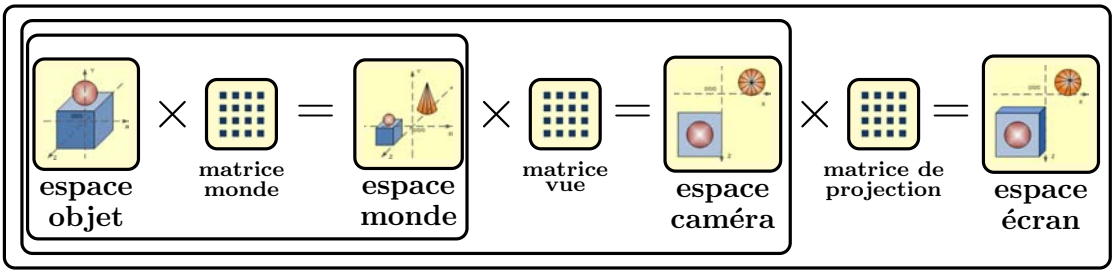


FIGURE 1.20: Illustration des quatre sytèmes de coordonnées utilisés lors du rendu d'une scène 3D (source des images : http://www.matrix44.net/cms/notes/opengl-3d-graphics/coordinate-systems-in-opengl).

Intéressons nous maintenant au fonctionnement d'un processeur graphique. La Figure 1.21 présente un schéma, inspiré de celui que l'on peut trouver dans le livre *The Orange Book : OpenGL Shading Language* [156], d'un pipeline de rendu exécuté par un processeur graphique. Ce pipeline correspond à celui décrit par les API graphiques OpenGL 2.0 / ES, versions encore très utilisées de nos jours pour le développement d'applications 3D. Nous allons brièvement décrire les différentes étapes de ce pipeline.

Une application 3D communique avec le processeur graphique via une API 3D (typiquement OpenGL ou Direct3D). Une scène 3D est décrite par une collection de primitives géométriques. Dans la plupart des cas, ces primitives sont des triangles définis par des sommets. A chaque sommet est associé un ensemble d'attributs comme la position, le vecteur normal, les coordonnées de texture ou encore la couleur.

Ces primitives sont envoyées du processeur de calcul (CPU) au processeur graphique (GPU) comme un flux de sommets. Le processeur graphique exécute alors les étapes suivantes pour générer une image affichable de la scène :

1. **Traitements des sommets :** Les sommets envoyés par l'application sont traités par le *processeur de sommets*. Dans un pipeline graphique traditionnel à fonctionnalités fixes, le processeur de sommets applique des opérations de transformation et d'éclairage. Les coordonnées d'un sommet dans l'espace objet sont alors transformées vers celles dans l'espace monde, puis vers celles dans l'espace caméra et enfin

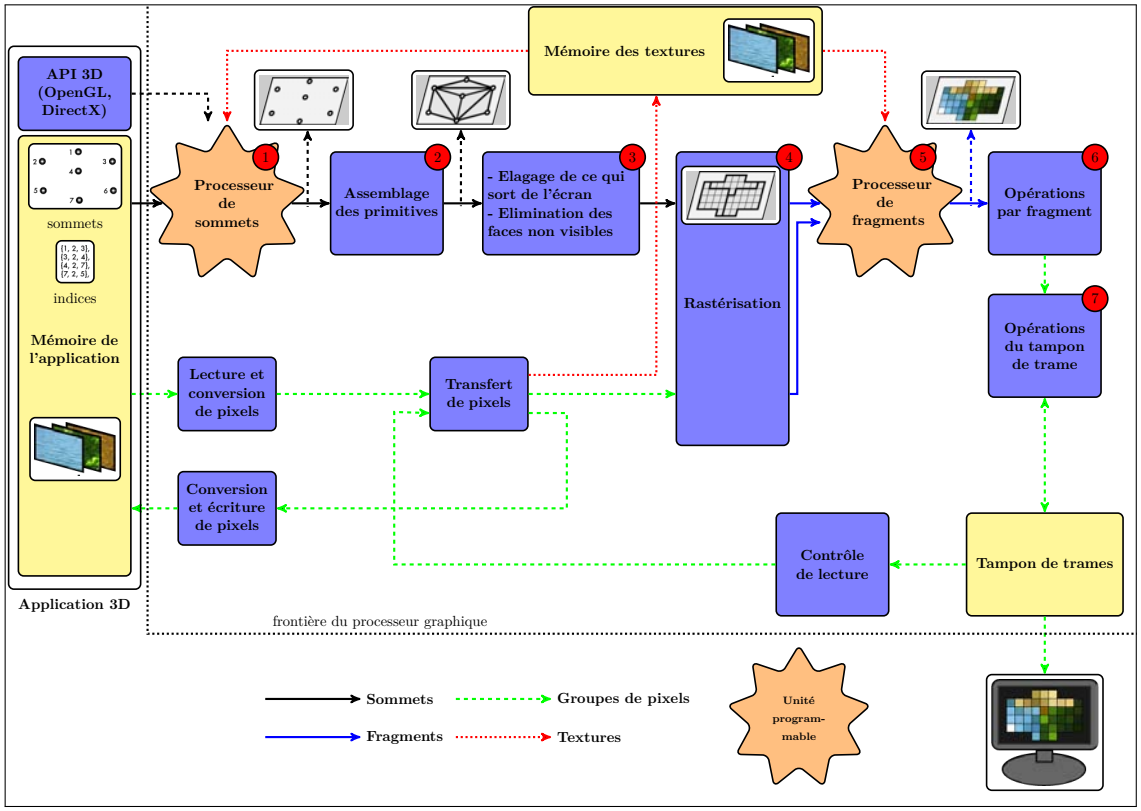


FIGURE 1.21: Le pipeline de rendu (de type OpenGL 2.0 / ES) exécuté par un processeur graphique. Ce schéma est inspiré de celui que l'on peut trouver dans le livre *The Orange Book : OpenGL Shading Language* [156]

vers celles dans l'espace écran. Les matrices monde, vue et de projection sont utilisées pour calculer ces transformations (voir Figure 1.20). Le calcul de l'éclairage est généralement basé sur le modèle de Blinn-Phong [21] et est effectué en fonction de la position de la caméra virtuelle. Les processeurs graphiques contemporains permettent aux développeurs de remplacer ces opérations fixes sur les sommets par un programme flexible appelé *vertex shader*.

2. **Assemblage des primitives :** Les sommets précédemment transformés sont assemblés en primitives géométriques simples : des triangles.

3. **Traitement des primitives :** Les primitives générées sortant de la zone d'affichage sont écartées, celles définissant des faces non visibles (car masquées par d'autres) le sont également.

4. **Rastérisation :** A cet étape, la scène 3D à afficher est convertie en fragments (portions de l'image finale pouvant être assimilées à des pixels). Par la suite, les opérations du processeur graphique seront uniquement appliquées à ces entités. Pour générer ces fragments à partir des primitives à afficher, un algorithme de rendu par

ligne de balayage est utilisé. Durant cette étape, les attributs des sommets sont interpolés pour les fragments contenus dans chaque triangle.

5. **Traitement des fragments :** La couleur de chaque fragment est assignée par le processeur de fragments en fonction des valeurs interpolées à partir des sommets. Elle peut également être combinée avec celles stockées dans des textures en mémoire vidéo. Les processeurs graphiques contemporains permettent de remplacer ces fonctionnalités fixes par un programme flexible appelé *fragment shader*.

6. **Opérations par fragment :** Le processeur graphique détermine si le fragment peut être envoyé pour écriture dans le tampon de trames en fonction des informations liées aux pixels de destination. Il exécute alors différents tests : le *alpha test* permettant d'écarter un fragment en fonction de sa transparence, le *stencil test* qui permet de limiter la zone de rendu et enfin le *depth test* qui sert à écarter un fragment en fonction d'une information de profondeur.

7. **Opérations du tampon de trame :** Les fragments sont écrits dans le tampon de trames, soit directement, soit par combinaison avec les pixels déjà écrits dans le tampon (opérations de *blending*).

Le pipeline graphique est bien adapté au processus de rendu car il permet au processeur graphique de fonctionner comme un processeur de flux vu que les sommets et les fragments peuvent être considérés comme indépendants. Par conséquent, toutes les étapes du pipeline peuvent être exécutées en simultané pour différents sommets et fragments y transitant. De plus, l'indépendance de ces données permet aux processeurs graphiques de disposer de plusieurs unités parallèles pour traiter plusieurs sommets ou fragments sur la même étape en même temps.

Les processeurs graphiques contemporains permettent de programmer certaines étapes du pipeline de rendu (typiquement le traitement des sommets et fragments). Ces programmes destinés à être exécuté sur le processeur graphique sont appelés *shaders* et sont écrits à l'aide de langage de haut niveau comme le GLSL [156] (*OpenGL Shading Language*) ou encore le HLSL [44] (*High Level Shading Language*, langage de *shading* pour Direct3D). Ils permettent de manipuler sommets, fragments ou encore textures avec la plupart des opérations disponibles sur un processeur de calcul classique. Vu que la plupart de ces calculs impliquent des opérations sur des vecteurs et des matrices, les processeurs graphiques sont de nos jours exploités pour leur puissance de calcul parallèle afin de traiter des données non graphiques. Un domaine de recherche appelé *GPU Computing* ou encore GPGPU (*General Purpose Computing on GPU*) s'emploie à proposer des solutions de calcul performant implémentées sur processeur graphique dans des domaines d'application

tels que la bio-informatique, les statistiques, la simulation ou encore la finance. La plateforme CUDA [143] de Nvidia a été le premier modèle de programmation pour le calcul sur GPU. Elle a donné suite à un standard, OpenCL [92], plus largement utilisé de nos jours. Ce standard ouvert pour la programmation sur GPU a été défini par le Khronos Group (aussi responsable d'OpenGL) et est supporté par les processeurs construits par Intel, AMD, Nvidia et ARM.

1.3.3 Exploitation du processeur graphique pour la visualisation

Les nouvelles fonctionnalités et capacités de programmation des processeurs graphiques ont été en premier lieu exploitées par la communauté de la Visualisation Scientifique. Le livre de Weiskopf [192] : *GPU-Based Interactive Visualization Techniques* présente un résumé des approches courantes de l'utilisation de shaders pour améliorer les performances et la qualité visuelle de visualisations scientifiques (e.g. visualisation de champs scalaires ou de vecteurs). Le rendu de volume en temps réel (typiquement pour des données médicales) a également largement bénéficié de l'évolution des processeurs graphiques programmables. Une étude de ces techniques de rendu peut d'ailleurs être trouvé dans le livre de Engel *et al.* [67] : *Real-time Volume Graphics*. Une des raisons de l'adoption rapide des processeurs graphiques programmables vient sûrement du fait que les données en entrée d'une visualisation scientifique ont généralement un arrangement spatial en deux ou trois dimensions et peuvent ainsi être facilement exprimées en tant que primitives graphiques manipulables par les langages de *shading* actuels. Afin de faciliter le développement de telles visualisations, des plateformes et des langages spécifiques ont été proposés permettant la génération automatique de *shaders* de visualisation. On peut citer par exemple Scout de McCormick *et al.* [133] ou le système de génération dynamique de *shaders* pour le rendu de volumes de Röβler *et al.* [154].

Si l'exploitation des fonctionnalités des processeurs graphiques a été largement adoptée dans le domaine de la Visualisation Scientifique, ce n'est pas vraiment le cas dans celui de la Visualisation d'Information. On trouve relativement peu de références en étudiant la littérature du domaine. D'après [134], cette situation serait due au fait qu'il est difficile de faire correspondre les types de données abstraits de haut niveau usuellement visualisés dans ce domaine au modèle bas niveau en points flottants supportés par les langages de *shading* actuels. Une des premières tentatives d'exploitation du processeur graphique, avant l'avènement des *shaders*, est le travail de Fekete et Plaisant [71] où ils présentent des techniques interactives de rendu permettant de visualiser plus d'un million éléments applicables à tous types de visualisation 2D (*scatter plot*, *tree map*, ...). Il existe des travaux en Visualisation d'Information utilisant des *shaders* mais ces derniers sont généralement

présentés comme un détail d'implémentation et non comme une contribution principale. Par exemple, Johansson *et al.* [107] utilisent des textures OpenGL pour stocker des données et un fragment shader pour rendre des coordonnées parallèles. Florek et Novotnỳ [73] utilisent également des shaders pour améliorer les performances de rendu des *scatter plots* et des coordonnées parallèles. Un autre exemple est le système de visualisation de graphes ZAME [65] qui dessinent des glyphes via des *shaders* pour des données multi-valuées. Il existe cependant quelques exemples de travaux où les *shaders* sont mis en avant et utilisés pour calculer des attributs visuels des données à afficher. Dans [11], Auber et Chiricota présentent une implémentation au *shader* de l'algorithme de dessin de graphe *Spring Embedder* améliorant de manière significative le temps de calcul. L'utilisation de *shaders* est également employée par Ingram *et al.* dans [103] pour implémenter un algorithme de positionnement multidimensionnel (en anglais, *Multidimensional scaling* ou MDS) multi-niveaux. Enfin, dans [134] McDonnel *et al.* présentent une approche plus généraliste de l'utilisation de *shaders*, appliquée à la visualisation de données multi-dimensionnelles, en proposant un raffinement du traditionnel pipeline de la Visualisation d'Information. Ce raffinement consiste en une étape finale dans l'espace image implémentable via l'utilisation de *fragment shaders*.

Le lecteur pourra trouver en annexe A un exemple d'exploitation du processeur graphique dans un contexte de Visualisation d'Information, à savoir l'implémentation d'une interaction visuelle de type *Fisheye*. Ce type d'interaction de type Focus+Contexte, généralisé par Furnas [80], permet de mettre en exergue une région d'intérêt dans une visualisation tout en gardant le contexte globale visible.

1.4 Organisation du mémoire

Nous décrivons dans cette section l'organisation de ce mémoire de thèse. Pour chaque chapitre, nous présentons brièvement son contenu ainsi que les contributions associées.

Le chapitre 2 présente des définitions et notations essentiels à la bonne compréhension de cette thèse. Ces dernières se rapportent majoritairement à la théorie des graphes.

Les travaux de cette thèse s'inscrivent dans le domaine de la visualisation interactive de graphes. Nous les avons séparés suivant deux parties. La première partie couvre les chapitres 3 et 4 et présente des travaux de dessin de graphe. Le dessin de graphe consiste à plonger visuellement un graphe dans un espace à n dimensions. Nous présenterons des méthodes de dessin de graphe dans le plan et dans l'espace tridimensionnel. La seconde partie couvre les chapitres 5, 6 et 7. Elle présente des techniques d'infographie pour la

visualisation interactive de graphes. La plupart de ces techniques s'attachent à exploiter le processeur graphique pour optimiser des algorithmes coûteux.

Le chapitre 3 s'intéresse aux méthodes de regroupement d'arêtes en faisceaux dans un dessin de graphe. Sa principale contribution est une méthode de ce type intuitive et efficace applicable aux dessins de graphe dans le plan. Une extension de cette méthode aux dessins de graphe dans l'espace sera également présentée.

Dans le chapitre 4, nous présentons un exemple d'application de techniques de dessin de graphe pour représenter des réseaux biologiques complexes que sont les réseaux métaboliques. Ce type de réseau modélise l'ensemble des réactions biochimiques se produisant dans les cellules d'un organisme vivant. La contribution de ce chapitre est une méthode permettant de dessiner un réseau métabolique complet. Cette méthode a l'avantage de préserver la topologie du réseau ainsi que l'information relative à des sous-ensembles particuliers de ce dernier : les voies métaboliques. Elle présente également un cas d'application concret de notre méthode de regroupement d'arêtes présentée au chapitre 3.

Le chapitre 5 s'intéresse aux méthodes de rendu de courbes paramétriques, courbes définies par un ensemble de points de contrôle comme par exemple les courbes de Bézier. Ce type de courbes est couramment utilisé pour représenter les arêtes dans un dessin de graphe, en particulier lorsque un algorithme de regroupement d'arêtes a été appliqué. Le problème est que le calcul des points interpolant une courbe est coûteux. La mise en place d'interactions fluides avec un dessin de graphe de ce type nécessite ainsi d'optimiser le temps de rendu de ces courbes. La contribution de ce chapitre est une technique de rendu de ce type de courbes exploitant pleinement le processeur graphique. Nous montrons qu'elle permet de rendre efficacement un grand nombre de courbes définies par un nombre arbitraire de points de contrôle.

Dans le chapitre 6, nous présentons une technique de rendu applicable sur les dessins de graphe avec regroupement d'arêtes. La finalité de cette contribution est de pouvoir visualiser l'information relative à la densité des faisceaux d'arêtes. L'analyse de cette information permet d'identifier les grands flux d'échange au sein d'un réseau.

Le chapitre 7 présente des méthodes pour visualiser des sous-graphes d'intérêt dans le contexte global d'une visualisation de graphe. Il est très courant de décomposer un graphe en sous-ensembles afin de mieux l'analyser. On peut citer par exemple, la fragmentation d'un réseau social en communautés ou encore la décomposition d'un réseau métabolique en voies métaboliques. Une décomposition de graphe peut être chevauchante, i.e. des sommets/arêtes sont partagés entre plusieurs groupes. Pouvoir visualiser efficacement le résultat d'une décomposition est ainsi un réel besoin pour faciliter le processus d'analyse. Les contributions de ce chapitre sont des techniques permettant de mettre visuellement

en exergue des sous-graphes dans une visualisation de graphe. Nous avons investigué deux sortes de technique. La première utilise des enveloppes concaves pour entourer des sous-graphes. La seconde utilise une déformation 3d pour mettre clairement un sous-graphe en évidence.

Enfin, nous dressons une conclusions sur les différentes travaux effectués au cours de cette thèse ainsi que les futures directions de recherche en découlant dans le chapitre 8.

Chapitre 2
Définitions et notations

Dans cette section, nous introduisons les définitions, notations et concepts importants nécessaires à la compréhension de cette thèse. Le contenu de ce chapitre est librement inspiré de Wikipédia ainsi que des définitions pouvant être trouvées dans la thèse de Romain Bourqui [25].

2.1 Graphes

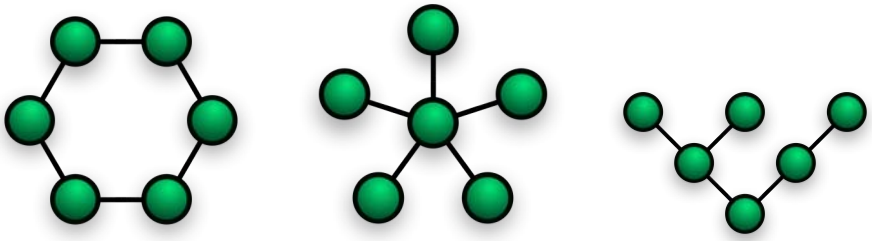
Définition 2.1 (Graphe non orienté) *Un graphe non orienté est un couple $G = (V, E)$ où V est un ensemble d'objets appelés les sommets du graphes (V de l'anglais "vertex", ou sommet en français), et $E \subseteq P_2(V)$, $P_2(V)$ désignant l'ensemble des parties de cardinalité 2 de V, un ensemble de paires d'éléments de V appelés les arêtes du graphe (E de l'anglais "edge", ou arête en français). Soit $e = \{u, v\}$ une arête, les sommets u et v sont appelés les extrémités de l'arête e.*

Définition 2.2 (Graphe orienté) *Un graphe orienté est un couple $G = (V, A)$ où V est un ensemble d'objets appelés les sommets du graphes, et $A \subseteq V \times V$, un ensemble de couple d'éléments de V appelés les arcs du graphe. Soit $a = (u, v)$ un arc, le sommet u (resp. le sommet v) est appelé la source (resp. la destination) de l'arc a.*

Définition 2.3 (Topologie d'un graphe) *La topologie est une branche des mathématiques qui fournit un cadre général pour traiter des notions de limite, de continuité et de voisinage. Elle s'attache entre autres à étudier des déformations spatiales de structures géométriques par des transformations continues, soit sans arrachage ni recollement de ces structures.*

La topologie d'un graphe correspond à la façon dont ses sommets sont connectés et reflète sa structure générale. La Figure 2.1 présente quelques exemples de topologies de graphe fréquemment observées.

Les définitions ci-après sont pour la plupart données pour les graphes non orientés mais peuvent bien entendu être adaptées pour les graphes orientés.



(a) (b) (c)

FIGURE 2.1: Quelques exemples de topologie de graphe (source des images : Wikipédia). (a) Topologie en anneau. (b) Topologie en étoile. (c) Topologie hiérarchique.

Définition 2.4 (Sous-graphe) *Soit $G = (V, E)$ un graphe. On appelle $G' = (V', E')$ un sous-graphe de G si et seulement si les conditions suivantes sont vérifiées :*

- $V' \subseteq V$
- $E' \subseteq E$

Définition 2.5 (Sous-graphe induit) *Soit $G = (V, E)$ un graphe. On appelle $G' = G[V'] = (V', E')$ un sous-graphe induit de G si et seulement si les conditions suivantes sont vérifiées :*

- *G' est un sous-graphe de G*
- $E' = \{\{u, v\} \,|\, \{u, v\} \in E, u \in V', v \in V'\}$

Définition 2.6 (Voisinage d'un sommet) *Soit $G = (V, E)$ un graphe. On appelle voisinage d'un sommet u de G, noté $N_G(u)$, l'ensemble $\{v|\,\{u, v\} \in E\}$.*

Définition 2.7 (Adjacence d'un sommet) *Soit $G = (V, E)$ un graphe. On appelle adjacence d'un sommet u de G, noté $adj_G(u)$, l'ensemble $\{\{u, v\} \,|\, \{u, v\} \in E\}$*

Définition 2.8 (Adjacence entrante ou sortante d'un sommet) *Soit $G = (V, A)$ un graphe orienté. On appelle adjacence entrante (resp. sortante) d'un sommet u de G, noté $adj^-_G(u)$ (resp. $adj^+_G(u)$), l'ensemble $\{(v, u)|(v, u) \in A\}$ (resp. $\{(u, v)|(u, v) \in A\}$).*

Définition 2.9 (Degré d'un sommet) *Soit $G = (V, E)$ un graphe. On appelle degré d'un sommet u de G, noté $deg_G(u)$, la quantité $|adj_G(u)|$.*

Définition 2.10 (Degré entrant et sortant d'un sommet) *Soit $G = (V, A)$ un graphe orienté. On appelle degré entrant (resp. sortant) d'un sommet u de G, noté $deg^-_G(u)$ (resp. $deg^+_G(u)$), la quantité $|adj^-_G(u)|$ (resp. $|adj^+_G(u)|$).*

Définition 2.11 (Valuation) *Soit $G = (V, E)$ un graphe et K un ensemble. On appelle valuation des sommets (resp. des arêtes) du graphe toute application $f : V \to K$ (resp. $f : E \to K$). On dit alors que G est sommet-valué (resp. arête valué) et on le note $G = (V, E, f)$.*

Définition 2.12 (Chemin non orienté) *Soit $G = (V, E)$ un graphe non orienté. On appelle chemin non-orienté (ou chemin) dans G, une séquence $(v_1, e_1, v_2, e_2, \ldots, e_{k-1}, v_k)$ avec :*

- $\forall i \in [1..k], v_i \in V$
- $\forall i \in [1..k[, e_i = \{v_i, v_{i+1}\} \in E$
- $\forall i, j \in [1..k] \mid i \neq j, v_i \neq v_j$
- $\forall i, j \in [1..k[\mid i \neq j, e_i \neq e_j$

Définition 2.13 (Chemin orienté) *Soit $G = (V, A)$ un graphe orienté. On appelle chemin non-orienté (ou chemin) dans G, une séquence $(v_1, e_1, v_2, e_2, \ldots, e_{k-1}, v_k)$ avec :*

- $\forall i \in [1..k], v_i \in V$
- $\forall i \in [1..k[, e_i = (v_i, v_{i+1}) \in A$
- $\forall i, j \in [1..k] \mid i \neq j, v_i \neq v_j$
- $\forall i, j \in [1..k[\mid i \neq j, e_i \neq e_j$

Un chemin ne passe pas deux fois par le même sommet, le même arc ou la même arête.

On dit que l'on peut atteindre un sommet v depuis un sommet u si et seulement si il existe un chemin tel que $u = v_1$ et $v = v_k$.

Définition 2.14 (Longueur d'un chemin) *Soit $G = (V, E)$ un graphe et $p = (v_1, e_1, v_2, e_2, \ldots, e_{k-1}, v_k)$ un chemin dans G. On définit la longueur de p comme la quantité $| \{e_1, \ldots, e_{k-1}\} | = k-1$, soit le nombre d'arêtes du chemin.*

Définition 2.15 (Longueur valuée d'un chemin) *Soit $G = (V, E, w)$ un graphe arête-valué et $p = (v_1, e_1, v_2, e_2, \ldots, e_{k-1}, v_k)$ un chemin dans G. On définit la longueur valuée de p comme la valeur $\sum_{i=1}^{k-1} w(e_i)$.*

Définition 2.16 (Distance dans un graphe) *Soit $G = (V, E)$ un graphe, $u \in V$ et $v \in V$. On appelle distance dans G entre u et v, notée $d_G(u, v)$, la longueur du plus court chemin dans G de u à v.*

Définition 2.17 (Distance valuée dans un graphe) *Soit $G = (V, E)$ un graphe arête valué, $u \in V$ et $v \in V$. Soit $P = \{p_1, p_2, \ldots, p_n\}$ l'ensemble de tous les chemins dans G de u à v. On appelle distance valuée dans G entre u et v, notée $d_{G,w}(u, v)$, la plus petite longueur valuée des chemins de P.*

Définition 2.18 (Cycle et graphe acyclique) *Soit $G = (V, E)$ un graphe non orienté et $u \in V$. On appelle cycle tout chemin de u à u. Si il n'existe pas de tel chemin dans G alors le graphe G est un graphe acyclique, noté **GA**.*

*Si $G = (V, A)$ est un graphe orienté, alors G est un graphe orienté acyclique, noté **DAG** pour* Directed Acyclic Graph.

Définition 2.19 (Arbre enraciné) *Soit $G = (V, A)$ un graphe orienté. On dit que G est un arbre enraciné si et seulement si :*

- G *est un DAG*
- $|A| = |V| - 1$
- $\exists! r \in V | deg_G^-(r) = 0$
- $\forall v \in V \setminus \{r\}, deg_G^-(v) = 1$

Le sommet r est appelé racine de l'arbre.

L'ensemble des sommets u tels que $deg_G^+(u) = 0$, noté $feuilles(G)$, est appelé feuilles de l'arbre.

Définition 2.20 (Arbre libre) *Soit $G = (V, E)$ un graphe non orienté. On dit que G est un arbre libre si et seulement si :*

- G *est un GA*
- $|E| = |V| - 1$

Définition 2.21 (DAG enraciné) *Soit $G = (V, A)$ un DAG. On dit que G est un DAG enraciné si et seulement si $\exists! r \in V | deg_G^-(r) = 0$.*

Définition 2.22 (Profondeur d'un sommet) *Soit $G = (V, A)$ un DAG enraciné de racine r et $u \in V$. On définit la profondeur de u dans G, notée $prof_G(u)$, comme suit :*

$$prof_G(u) = d_G(r, u)$$

Définition 2.23 (Graphe connexe) *Soit $G = (V, E)$ un graphe non orienté. Le graphe G est connexe si il existe un chemin entre toute paire de sommets dans G.*

Définition 2.24 (Graphe complet) *Soit $G = (V, E)$ un graphe non orienté. On dit que le graphe G est un graphe complet si et seulement si $\forall u \in V$ et $\forall v \in V$, $\{u, v\} \in E$.*

Un graphe complet à n sommets, noté K_n, contient $\frac{n(n-1)}{2}$ arêtes.

Définition 2.25 (Stable (ou ensemble indépendant)) *Soit $G = (V, E)$ un graphe. On appelle le sous-graphe $G' = (V', E')$ un stable de G si et seulement si la condition suivante est vérifiée :*

$$\forall u \in V', \forall v \in V', \neg\exists \{u, v\} \in E$$

Un stable G' de G est donc un sous-graphe induit de G sans arêtes, soit $G' = G[V'] = (V', \emptyset)$.

Définition 2.26 (Graphe biparti) *Soit $G = (V, E)$ un graphe. On dit que G est biparti si il existe une partition de son ensemble de sommets en deux sous-ensembles V_1 et V_2 telle que chaque arête ait une extrémité dans V_1 et l'autre dans V_2. Autrement dit, G est biparti si les conditions suivantes sont vérifiées :*

- $V_1 \in V$ *et* $V_2 \in V$
- $V_1 \cap V_2 = \emptyset$
- $V_1 \cup V_2 = V$
- $G[V_1]$ *et* $G[V_2]$ *sont des stables*

Définition 2.27 (Graphe biparti complet) *Soit $G = (V, E)$ un graphe. On dit que G est biparti complet s'il est biparti et contient le nombre maximal d'arêtes. Autrement dit, il existe une partition de son ensemble de sommets en deux sous-ensembles V_1 et V_2 telle que chaque sommet de V_1 est relié à chaque sommet de V_2. Si V_1 est de cardinalité m et V_2 de cardinalité n, le graphe biparti complet est noté $K_{m,n}$.*

Définition 2.28 (Graphe décomposé) *Soit $G = (V, E)$ un graphe et $H = (V_H, A_H)$ un DAG enraciné tel que $feuilles(H) = V$. Le graphe décomposé (G, H) est défini comme suit : tout sommet de $u \in V_H \backslash feuilles(H)$ représente un groupe C_u de sommets de G tel que $C_u = \{v \in V \mid deg^+_H(v) = 0$ et il existe un chemin de u à v dans $H\}$ (voir Figure 2.2).*

Le graphe H est appelé DAG de décomposition.

Lorsque le DAG de décomposition n'est pas un arbre, on parle alors de décomposition chevauchante.

On appelle niveau i de (G, H), l'ensemble des sommets u de H tels que $prof_H(u) = i$.

Définition 2.29 (Graphe de dépendance) *Soit (G, H) un graphe décomposé, $\{u_1, u_2, \ldots, u_p\}$ le niveau i de (G, H) et $C = \{C_1, C_2, \ldots, C_p\}$ l'ensemble des groupes correspondants dans G. Le graphe de dépendance $Dep_i(G, H) = (V_{Dep}, E_{Dep})$ du niveau i de (G, H) est défini comme suit :*

- *$V_{Dep} = \{u_1, u_2, \ldots, u_p\}$.*
- *$e = \{u_j, u_k\} \in E_{Dep}$ si et seulement si $j \neq$ et $C_i \cap C_k \neq \emptyset$*

Un exemple de graphe de dépendance est donné à la Figure 2.2.

Définition 2.30 (Graphe partitionné) *Soit (G, H) un graphe décomposé. (G, H) est un graphe partitionné si et seulement si H est un arbre enraciné. Un exemple de graphe partitionné est donné à la Figure 2.3(a).*

Le graphe H est appelé arbre de partition.

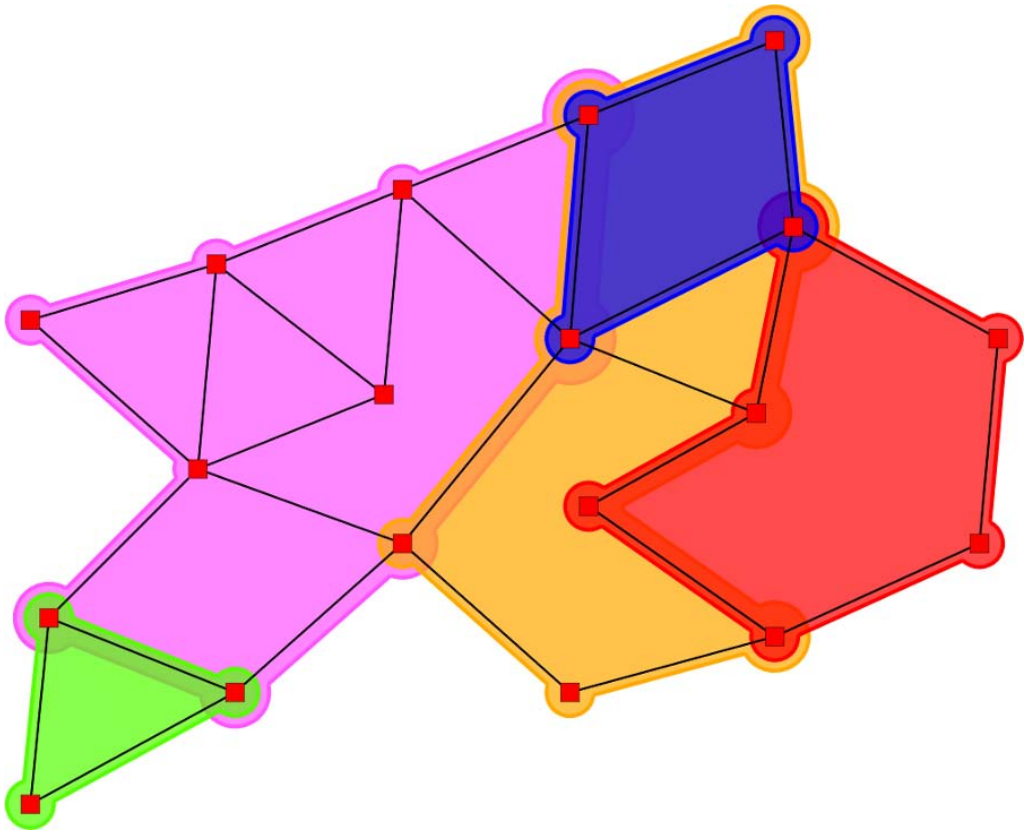
Définition 2.31 (Arête inter-groupes / intra-groupe) *Soit $G = (V, E)$ un graphe et $C = \{C_1, C_2, \ldots, C_n\}$ une partition des sommets de G. On dit que $e = \{u, v\} \in E$ est une arête inter-groupes (resp. intra-groupe) si il existe C_i et C_j tels que $u \in C_i$ et $v \in C_j$ (resp. s'il existe C_i tel que $u, v \in C_i$) (voir Figure 2.3(a)).*

Définition 2.32 (Graphe quotient) *Soient (G, H) un graphe partitionné, $\{u_1, u_2, \ldots, u_p\}$ le niveau i de (G, H) et $C = \{C_1, C_2, \ldots, C_p\}$ l'ensemble des groupes correspondants dans G. Le graphe quotient $Q_i = (V_{Q_i}, E_{Q_i}, A_{Q_i})$ du niveau i de (G, H) est défini comme suit :*

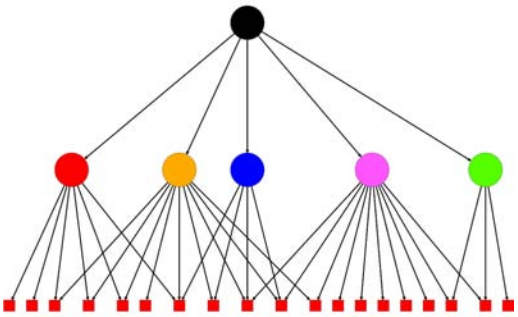
- *$V_{Q_i} = \{u_1, u_2, \ldots, u_p\}$.*
- *$e = \{u_j, u_k\} \in E_{Q_i}$ si et seulement si $j \neq k$ et $\exists u \in C_j, v \in C_k$ tels que $\{u, v\} \in E$.*
- *$a = (u_j, u_k) \in A_{Q_i}$ si et seulement si $j \neq k$ et $\exists u \in C_j, v \in C_k$ tels que $(u, v) \in A$.*

*Un sommet (resp. arête et arc) de Q_G est appelé **méta-sommet** (resp. **méta-arête** et **méta-arc**).*

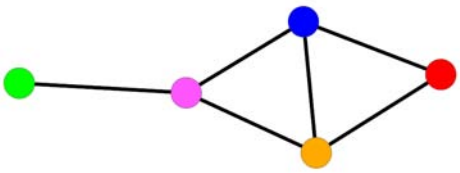
Un exemple de graphe quotient est donné à la Figure 2.3(b).



(a)

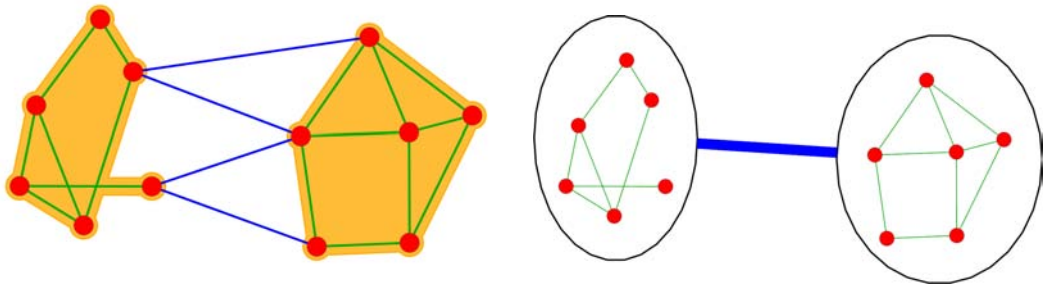


(b)



(c)

FIGURE 2.2: Exemple de décomposition de graphe (G, H). La figure (a) montre le graphe G, la (b) le DAG de décomposition H et la (c) le graphe de dépendance du niveau 1 de (G, H).



(a) (b)

FIGURE 2.3: (a) Exemple de graphe partitionné en deux groupes (matérialisés en orange dans le dessin). Les arêtes vertes sont les arêtes intra-groupe tandis que les bleues sont les arêtes inter-groupes. (b) Graphe quotient correspondant.

2.2 Autres définitions

2.2.1 Graphes de visibilité

Graphe de visibilité : En géométrie algorithmique, un graphe de visibilité est un graphe de positions mutuellement visibles entre elles, typiquement pour un ensemble de points et d'obstacles dans le plan. Chaque sommet du graphe représente une position et une arête représente une connexion visible entre elles. Ainsi, si le segment de ligne connectant deux positions ne traverse aucun obstacle, une arête est dessinée entre elles dans le graphe (voir Figure 2.4).

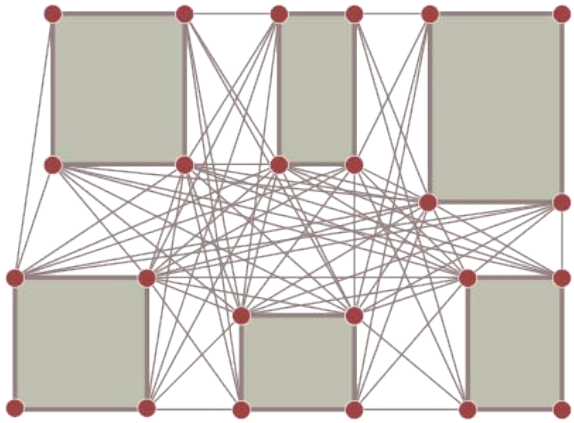


FIGURE 2.4: Exemple de graphe de visibilité pour un ensemble de points et d'obstacles.

Graphe de visibilité tangent : Le graphe de visibilité tangent est le sous-graphe d'un graphe de visibilité qui ne contient que les arêtes correspondant à des tangentes communes entre deux objets.

2.2.2 Quadtree

Un *quadtree* est une structure de données arborescente dans laquelle chaque nœud interne a exactement quatre enfants. Les *quadtrees* sont généralement utilisés pour partitionner un espace à deux dimensions, soit un plan, en le subdivisant récursivement en quatre parties ou régions. Ces régions peuvent être des carrés, des rectangles mais peuvent également avoir des formes arbitraires. Cette structure de données a été nommée quadtree par Finkel and Bentley en 1974 [72]. Un *quadtree* possède les propriétés suivantes :

- Il décompose le plan en cellules adaptables.
- Chaque cellule de la structure a une capacité maximum. Quand cette capacité maximum est atteinte, la cellule est subdivisée en quatre parties.
- L'arborescence de la structure représente les décompositions successives du plan.

La Figure 2.5 présente une illustration d'un *quadtree* décomposant le plan en cellules jusqu'à ce que chaque cellule contienne au plus un point pris dans un ensemble prédéfini.

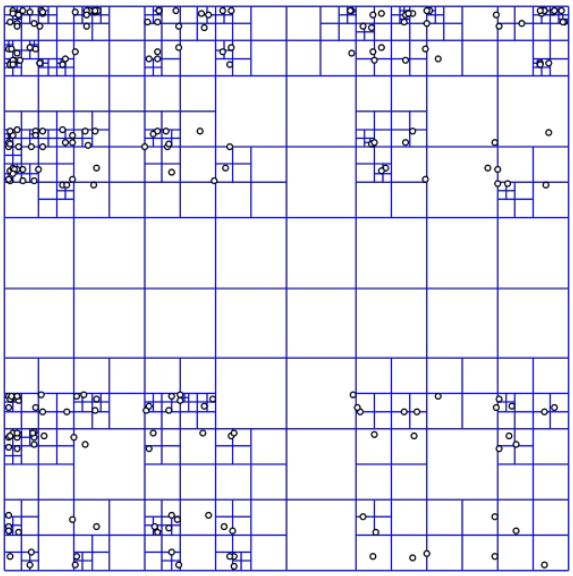


FIGURE 2.5: Exemple de *quadtree* construit à partir d'un ensemble de points dans le plan (source de l'image : Wikipédia). Ce dernier est récursivement divisé en quatre cellules jusqu'à ce que chaque cellule contienne au plus un de ces points.

2.2.3 Octree

Un *octree* est l'analogie en trois dimensions du quadtree. C'est une structure de données arborescente dans laquelle chaque nœud interne a exactement huits enfants. Les octrees sont le plus souvent utilisés pour partitionner un espace tridimensionnel en le subdivisant récursivement en huit octants. Un *octree* possède les propriétés suivantes :

- Il décompose l'espace en cellules adaptables.
- Chaque cellule de la structure a une capacité maximum. Quand cette capacité maximum est atteinte, la cellule est subdivisée en huit parties.
- L'arborescence de la structure représente les décompositions successives de l'espace.

La Figure 2.6 présente une illustration d'un *octree* décomposant l'espace en cellules jusqu'à ce que chaque cellule contienne au plus un point pris dans un ensemble prédéfini.

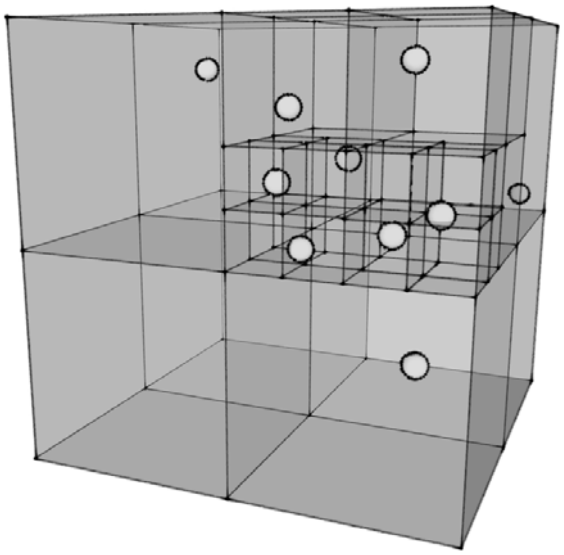


FIGURE 2.6: Exemple d'*octree* construit à partir d'un ensemble de points dans l'espace (source de l'image : Wikipédia). Ce dernier est récursivement divisé en huit cellules jusqu'à ce que chaque cellule contienne au plus un de ces points.

2.2.4 Diagramme de Voronoï

Soit E un espace euclidien et S un ensemble fini de n points de E ; les éléments de S sont appelés sites, centres ou encore germes. On appelle cellule de Voronoï associée à un élément p de S l'ensemble des points de E qui sont plus proches de p que de tout autre point de S. Si on dénote par $Vor_S(p)$ une cellule et $d(x, y)$ une fonction renvoyant la distance entre deux points x et y de E, alors :

$$Vor_S(p) = \{x \in E \ / \ \forall q \in S \ d(x, p) \leq d(x, q)\}$$

Pour deux points a et b de S, l'ensemble $\Pi(a, b)$ des points équidistants de a et b est un hyperplan affine. Cet hyperplan définit la frontière entre l'ensemble des points plus poches de a que de b, et l'ensemble des points plus proches de b que de a.

$$\Pi(p, q) = \{x \in E \ / \ d(x, p) = d(x, q)\}$$

On note $H(a, b)$ le demi espace délimité par cet hyperplan contenant a, il contient alors tous les points plus proches de a que de b. La cellule de Voronoï associée est alors l'intersection des $H(a, b)$ où b parcourt $S \setminus \{a\}$.

$$H(p, q) = \{x \in E \ / \ d(x, p) \leq d(x, q)\}$$
$$Vor_S(p) = \bigcap_{q \in S \setminus \{p\}} H(p, q)$$

En dimension 2, les cellules de Voronoï correspondent à des polygones convexes et en dimension 3 à des polyèdres convexes. L'ensemble de ces polygones/polyèdres partitionne E et définit la partition de Voronoï correspondant à l'ensemble S. Il est également facile de dessiner ces partitions en dimension 2 et 3, et le dessin résultant est appelé diagramme de Voronoï. La Figure 2.7 présente une illustration d'un diagramme de Voronoï en deux dimensions et la Figure 2.8 une illustration d'un diagramme de Voronoï en trois dimensions.

Ce type de diagramme est nommé d'après son inventeur : Georgy Voronoï [187]. Le diagramme de Voronoï est le dual de la triangulation de Delaunay. On peut définir la triangulation de Delaunay à partir du diagramme de Voronoï, deux sites p et q créent une arête dans le graphe de Delaunay si et seulement si les cellules de Voronoï associées à p et q sont adjacentes.

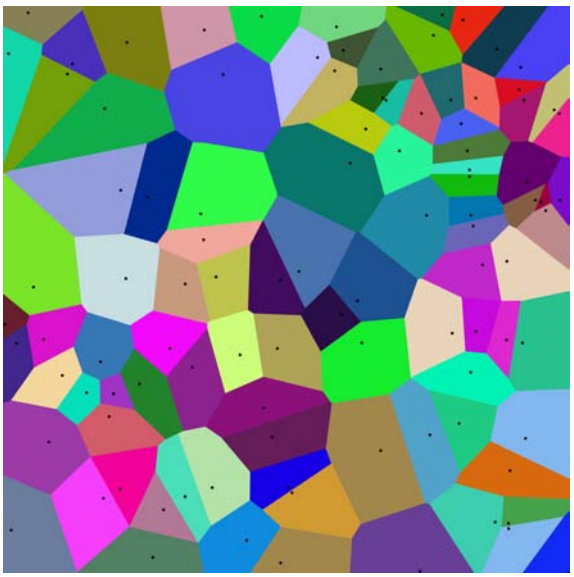


FIGURE 2.7: Exemple de diagramme de Voronoï en deux dimensions (source de l'image : Wikipédia).

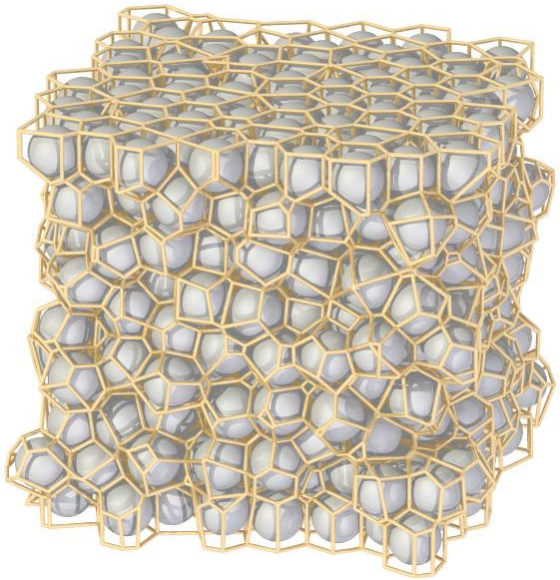


FIGURE 2.8: Exemple de diagramme de Voronoï en trois dimensions (source de l'image : thèse de Chris H. Rycroft [157])

Première partie

Dessin de graphes

Dans cette partie, nous présentons les différentes contributions de cette thèse en matière de dessin de graphe. Dessiner un graphe consiste à calculer des positions pour ses sommets ainsi que des éventuels points de contrôle pour ses arêtes dans un espace à n dimensions. Afin de pouvoir être affichable sur un support visuel, cet espace est soit un plan ($n = 2$) soit un espace tridimensionnel ($n = 3$).

Dans le chapitre 3, nous nous focalisons sur les méthodes de regroupement d'arêtes dans un dessin de graphe. Ces méthodes permettent de résoudre les problèmes d'occlusion visuelles dus au dessin des arêtes dans les représentations de type *nœud-lien* de grands graphes. Nous dressons une état de l'art exhaustif et présentons un algorithme de regroupement d'arêtes que nous avons élaborés pour les dessins de graphe dans le plan. Une extension de cet algorithme pour les dessins de graphe dans l'espace, et en particulier les dessins sphériques, est également détaillée.

Dans le chapitre 4, nous présentons une méthode permettant de dessiner un réseau métabolique complet. Ce type de graphe modélise l'ensemble des réactions biochimiques se produisant dans les cellules d'un organisme vivant. Notre méthode combine de la décomposition de graphe ainsi que des méthodes de dessin de graphe classiques et récentes pour générer automatiquement une représentation d'un réseau complet. En particulier, notre appliquons une version dédiée de notre méthode de regroupement d'arêtes afin de produire une visualisation du réseau avec un faible niveau d'occlusions visuelles. La méthode est de surcroît flexible car elle permet à l'utilisateur de piloter le processus de dessin en fonction de l'analyse biologique à effectuer.

Chapitre 3

Réduction des problèmes d'occlusion par regroupement d'arêtes

Visualiser efficacement un graphe contenant un grand nombre de sommets et d'arêtes est un vrai défi. Les dessins d'un tel graphe souffrent généralement d'occlusions visuelles induites par la grande quantité d'arêtes. Dans une représentation de graphe de type *nœud-lien*, les arêtes sont souvent dessinées avec des segments reliant deux sommets voisins. Cette approche pour dessiner les arêtes peut alors altérer la lisibilité du dessin. Bien que les algorithmes de dessin de graphe essaient de manière générale de minimiser le nombre de croisements d'arêtes, il y a des cas où ils ne peuvent produire un résultat optimal. Une autre situation où les croisements d'arêtes ne peuvent être évités est quand il existe des arêtes connectant des sommets dessinés loin l'un de l'autre. Dans ce cas, il y a une forte probabilité que ces longues arêtes soient dessinées au dessus d'autres plus courtes. Une autre difficulté est quand une arête est dessinée par dessus un sommet, ce qui entraîne une confusion pour distinguer cette arête de celles entrantes et sortantes du sommet recouvert. Il peut ainsi être difficile d'identifier les connexions entre les sommets ainsi que les principaux flux d'échanges pouvant exister au sein du graphe.

Une solution à l'approche classique du dessin d'arêtes en segments est d'utiliser des lignes brisées ou des courbes. De cette façon, les croisements d'arêtes peuvent être évités et les arêtes peuvent être dessinées sans qu'elles chevauchent les sommets. A notre connaissance, Gansner *et al.* [85] ont été les premiers à utiliser des courbes pour dessiner les arêtes afin de réduire certains problèmes d'occlusion. Cependant, leur algorithme a été conçu pour être utilisé sur des graphes de taille relativement petite où l'occlusion due au dessin des arêtes n'entraîne pas de problèmes quant à la lisibilité du dessin. Ce type d'occlusion devient par contre problématique lorsque l'on s'attaque à dessiner des graphes de taille importante, pour lesquels les méthodes de dessin par modèle de forces (voir section 1.2.1.2) sont les plus pratiques et les plus couramment utilisées (en opposition

à d'autres stratégies basées sur le classement et le tri de sommets, comme par exemple dans l'algorithme de dessin de graphe orienté de Gansner *et al.* [85]).

Des techniques visant à regrouper les arêtes en faisceaux (en anglais *edge bundling*) ont alors été proposées pour aider à résoudre ces problèmes d'occlusion dus aux arêtes. De façon grossière, l'idée derrière ce type de techniques est de regrouper les arêtes afin de réduire les croisements et faire apparaître les différents flux d'information entre les régions du dessin de graphe. Ainsi, différentes arêtes peuvent émerger de plusieurs sommets positionnés dans une région A du dessin et en même temps être connectées à d'autres sommets situés dans une région B. Ces arêtes sont alors regroupées sur les parties où elles partagent leur route afin de montrer les régions où les flux se concentrent. Le principe de ces techniques de regroupement d'arêtes est donc de calculer des points de contrôle afin de donner une nouvelle forme à chaque arête.

Le reste de ce chapitre est organisée de la façon suivante. Dans la section 3.1, nous commençons par dresser un état de l'art des différentes méthodes qui ont été proposées afin de réduire les problèmes d'occlusion dus au dessin des arêtes. Un algorithme de regroupement d'arêtes pour un dessin de graphe dans le plan que nous avons élaboré, nommé *Winding Roads*, est ensuite détaillé dans la section 3.3.1. Cet algorithme a été publié dans les actes de la conférence *EuroVis 2010* via un numéro spécial du journal *Computer Graphics Forum* [123]. Nous terminons par présenter dans la section 3.3 une extension du précédent algorithme applicable aux dessins de graphe dans l'espace et en particulier aux dessins sphériques. Cette extension a été publiée dans les actes de la *14ème conférence internationale sur la Visualisation d'Information (IV'10)* [122].

3.1 État de l'art

Cette section vise à énumérer les différents travaux qui ont été effectués pour tenter de réduire les problèmes d'occlusion dans les visualisations de graphe dus au dessin des arêtes. Ces travaux sont classés en fonction des méthodes utilisées. Pour de plus amples informations sur les techniques de réduction d'occlusions dans les dessins de graphe et pas seulement celles dues aux arêtes, nous recommandons l'étude de Ellis et Dix [64].

Réduction d'occlusions par routage d'arêtes Un des premiers efforts pour réduire les problèmes d'occlusions dans les dessins de graphe a été fait par la communauté *Graph Drawing*. Pour améliorer la lisibilité d'un dessin de graphe, il est important d'essayer de borner le nombre de croisement d'arêtes mais également d'éviter des chevauchements entre sommets et arêtes. Dans [55], Dobkin *et al.* proposent une nouvelle méthode utilisant des

graphes de visibilité (voir section 2.2.1) et du routage d'arête par plus court chemin afin de supprimer les chevauchements entre sommets et arêtes. Cette technique sera ensuite portée aux graphes de visibilité tangents (voir section 2.2.1) par Wybrow *et al.* [197]. Dwyer et Nachmanson [58] ont d'ailleurs proposé une heuristique rapide pour calculer une approximation du graphe de visibilité afin de réduire la complexité en temps de ces approches et ainsi pouvoir router les arêtes de grands graphes. Ces méthodes réduisent efficacement les problèmes d'occlusions dus aux chevauchements entre sommets et arêtes. Cependant, elles ne cherchent pas à regrouper les arêtes similaires et n'aident pas à identifier les principaux flux d'arêtes qui peuvent exister au sein du graphe. Ce n'est pas le cas du récent travail de Pupyrev *et al.* [147] dans lequel ils créent des regroupements d'arêtes de haute qualité en routant les arêtes originales sur une grille (voir Figure 3.2(f)). La grille de routage qu'ils utilisent est basée soit sur l'utilisation d'un graphe de visibilité soit sur un raffinement d'une triangulation de Delaunay. Leur technique permet également de séparer les arêtes contenues dans chaque faisceau créé et produit ainsi des représentations très esthétiques de type "carte de métro". L'algorithme que nous présentons dans la section 3.3.1 est similaire à cette méthode (publiée postérieurement à la nôtre) dans le fait que nous utilisons également une grille pour router les arêtes en faisceaux.

Réduction d'occlusions par techniques d'interaction Wong *et al.* proposent dans [195, 196] des techniques d'interaction pour enlever l'occlusion due aux arêtes autour de points focaux définis par l'utilisateur. Les arêtes proches d'un point focal sont repoussées à la manière d'un *Fisheye* (voir Annexe A) tandis que les positions des sommets sont préservées. Les problèmes d'occlusions sont localement résolus autour de chaque point focal, mais ces techniques ne permettent pas de les enlever sur la globalité de la représentation.

Réduction d'occlusions par dessin confluent La communauté *Graph Drawing* s'est focalisée sur une représentation particulière d'un graphe appelée *dessin de graphe confluent*. Dans un dessin de graphe confluent, un graphe est représenté sans croisement d'arêtes. Avec cette technique, les arêtes se croisant dans le dessin de départ sont dessinées en tant que courbes se chevauchant. Par exemple, Dickerson *et al.* [52] donnent un algorithme pour calculer un dessin de graphe confluent qui est basé sur la détection de cliques maximum et de bi-cliques (graphe complet biparti). Ensuite, les arêtes sont regroupées pour obtenir une représentation planaire, i.e. un dessin sans croisements d'arêtes, de ces sous-graphes non planaires. Bien que les techniques de dessin de graphe confluent donnent des résultats intéressants, elles ne peuvent pas être appliquées à toutes les classes de graphe (voir [52] pour plus de détails).

Réduction d'occlusions par fragmentation de sommets Phan *et al.* présentent dans [146], une technique de dessin de *carte de flux* basée sur une fragmentation géométrique et hiérarchique des sommets. Ce type de carte représente un ensemble de flux depuis un sommet source vers un ensemble de sommets cibles. Dans [146], les arêtes sont alors routées le long des branches de l'arbre de la hiérarchie calculée. Une autre méthode plus récente pour générer ce type de cartes a été proposé par Buchim *et al.* [30]. Elle est basée sur l'utilisation de *Spiral trees* [29] qui permettent de connecter plusieurs sommets cibles à un sommet source via un arbre de longueur minimale dont les arcs obéissent à une certaine restriction avec l'angle qu'ils forment avec la source. Le principe de [146] a également été utilisée par Holten dans [99] afin d'améliorer la lisibilité des relations dans les données hiérarchiques et relationnelles (voir Figure 3.1). Le principal inconvénient de ces méthodes est que les arêtes sont routées en utilisant un arbre de hiérarchie ce qui peut être restrictif dans le cas général.

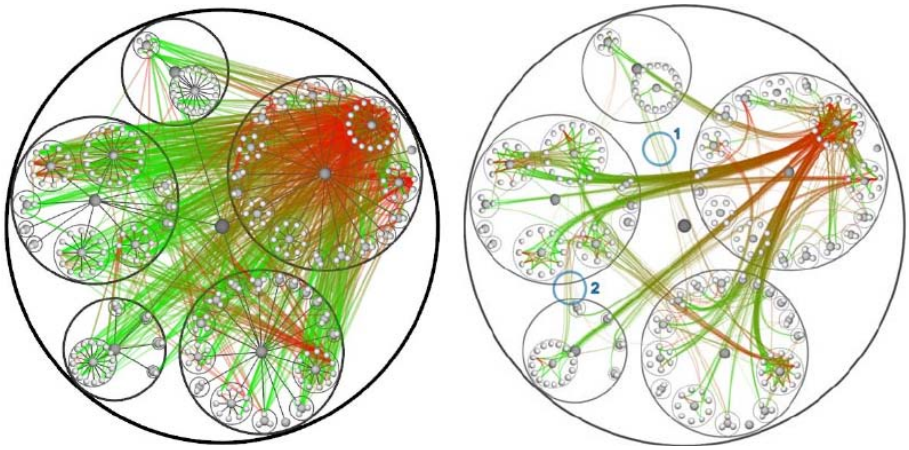
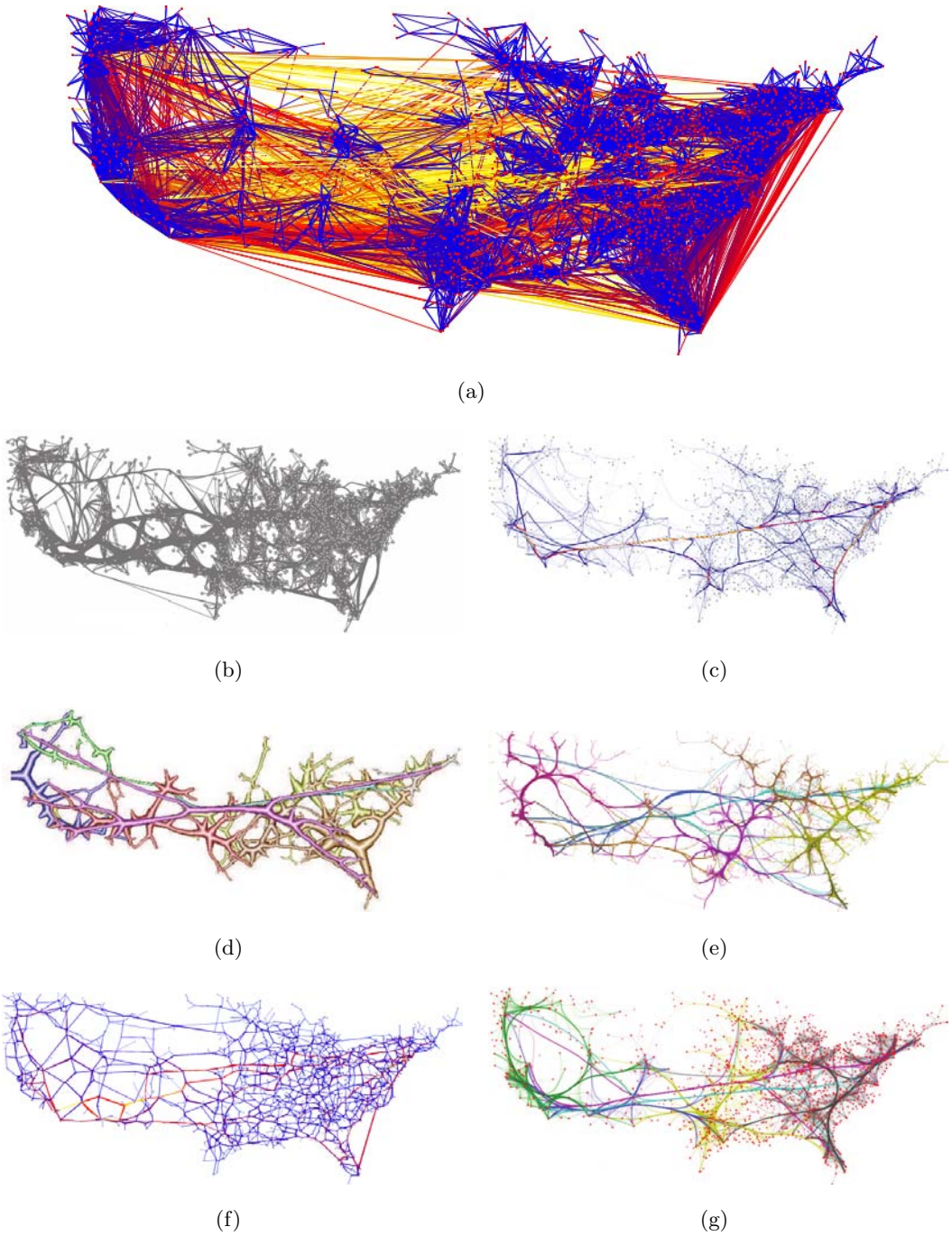


FIGURE 3.1: Exemple de représentation obtenue avec la méthode de regroupement d'arêtes *Hierachical Edge Bundles* de Holten [99]. A gauche le dessin de départ, à droite celui après application de l'algorithme. ©2006 IEEE.

Réduction d'occlusions par fragmentation d'arêtes Dans [83], Gansner et Koren proposent une technique améliorée de dessin de graphe circulaire où les arêtes sont routées soit à l'extérieur du cercle soit à l'intérieur. Les arêtes routées à l'intérieur du cercle sont regroupées en utilisant un algorithme de fragmentation d'arêtes qui essaie d'optimiser l'utilisation de la zone. Une autre technique de fragmentation d'arêtes est proposée par Cui *et al.* [46] (voir Figure 3.2(b)). Ils présentent une approche géométrique pour créer des faisceaux d'arêtes. Leur principale idée est de construire un maillage de contrôle basé sur une interaction de l'utilisateur ou une triangulation de Delaunay. Le maillage est ensuite utilisé pour calculer les régions du dessin où les arêtes devraient être regroupées. Le regroupement des arêtes est effectué en fonction d'un algorithme de fragmentation basé



(a)

(b)

(c)

(d)

(e)

(f)

(g)

FIGURE 3.2: Comparaison des résultats obtenus par différents algorithmes de regroupement d'arêtes sur le graphe des migrations de travailleurs aux États Unis entre les années 1995 et 2000 [31]. (a) Dessin original du graphe sans regroupement d'arêtes. (b) *Geometry-Based Edge Clustering*, Cui *et al.* [46]. ©2008 IEEE. (c) *Force-Directed Edge Bundling*, Holten et van Wijk [100]. ©2009 IEEE. (d) *Image-Based Edge Bundles*, Telea et Orsoy [178]. ©2010 IEEE. (e) *Skeleton-Based Edge Bundling*, Ersoy *et al.* [68]. ©2011 IEEE. (f) *Edge routing with ordered bundles*, Pupyrev *et al.* [147]. ©2011 Springer-Verlag. (g) *Graph Bundling by Kernel Density Estimation*, Hurter *et al.* [101]. ©2012 IEEE.

sur l'orientation des arêtes. Une étape supplémentaire est ensuite exécutée afin de réduire l'effet zig-zag sur la forme des arêtes.

Holten et van Wijk ont présenté dans [100] une heuristique par modèle de forces pour regrouper des arêtes, et ainsi réduire les problèmes d'occlusions, d'un graphe où les positions des sommets sont fixées (voir Figure 3.2(c)). Dans cette heuristique, des sommets temporaires sont insérés pour diviser les arêtes en plusieurs segments. Une mesure de similarité entre arêtes est calculée pour déterminer quelles arêtes devraient être regroupées. Les sommets temporaires de deux arêtes devant interagir sont alors reliés par une arête. Les faisceaux d'arêtes sont alors obtenus en exécutant un algorithme de modèle de forces sur le graphe formé par les sommets et arêtes temporaires. Les positions des sommets originaux sont bien entendus préservées lors de l'exécution de cette phase. L'inconvénient majeure de cette méthode est qu'elle repose sur une mesure de similarité plutôt coûteuse qui compare chaque arête à toutes les autres, soit une complexité en temps de $O(|E|^2)$. Une extension de cette technique a été proposée par Selassie *et al.* [163]. La version originale de l'algorithme ne prenait pas en compte la direction des arêtes lors d'un regroupement. L'amélioration majeure introduite dans [163] permet, par une modification des forces utilisées dans la simulation physique pour regrouper les arêtes, de séparer un regroupement d'arêtes en deux faisceaux "parallèles" (un par orientation d'arêtes). Un autre travail est celui de Pupyrev *et al.* [164] ou ils présentent une méthode de regroupement d'arêtes pour améliorer la lisibilité des dessins hiérarchiques produits par des algorithmes suivant le principe proposé par Sugiyama [175]. Ils utilisent une fragmentation des arêtes calculée à partir du dessin en entrée pour créer des regroupements à l'aide d'un algorithme de minimisation d'encre dédié.

Une autre technique basée sur la fragmentation des arêtes est proposée par Gansner *et al.* [84] (voir Figure 3.3). Leur solution est d'utiliser une mesure de similarité entre arêtes simple où chaque arête est considérée comme un point dans un espace à 4 dimensions et la similarité entre arêtes par leur proximité dans cet espace. A partir de ce paramétrage, ils peuvent alors construire un graphe de proximité entre arêtes de manière efficace via une méthode de décomposition d'espace. Les arêtes qui sont voisines dans ce graphe de proximité sont alors regroupées si la "quantité d'encre" nécessaire pour les dessiner est minimisée. A la fin de cette étape, le graphe de proximité est alors contracté de telle sorte que les sommets reliant deux arêtes qui ont été regroupées est remplacé par un seul représentant le groupe d'arêtes. Le processus de regroupement est alors de nouveau exécuté sur ce nouveau graphe de proximité. Ce processus multi-niveaux se termine lorsque la "quantité d'encre" nécessaire pour dessiner le graphe ne peut plus être minimisée.

Enfin, il existe des techniques basées sur de la fragmentation d'arêtes et du traitement d'image pour améliorer ou générer une visualisation de graphe avec regroupement

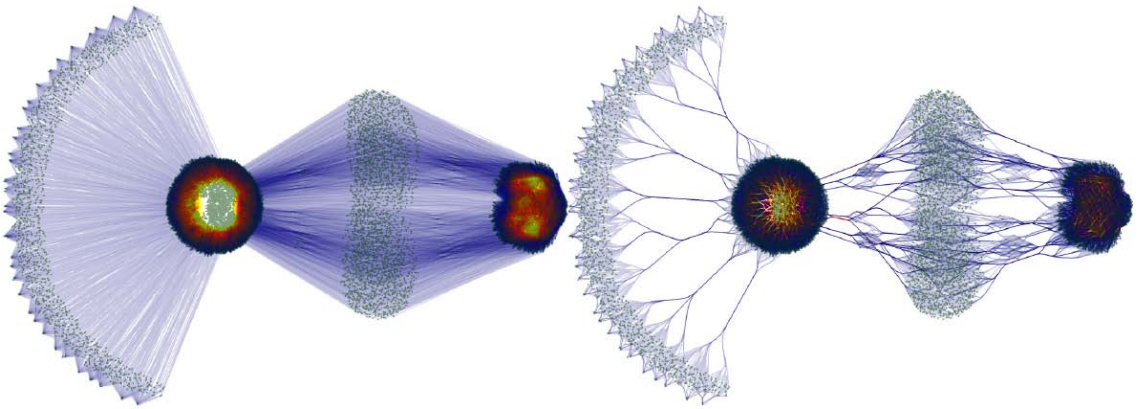


FIGURE 3.3: Exemple de représentation obtenue en appliquant la méthode multi-niveaux de regroupement d'arêtes *Multilevel agglomerative edge bundling* de Gansner *et al.* [84]. A gauche le dessin de départ, à droite celui après application de l'algorithme. ©2011 IEEE.

d'arêtes. Telea et Orsoy présentent dans [178] une méthode pour améliorer les visualisations de graphe avec regroupement d'arêtes (voir Figure 3.2(d)). Leur technique repose sur une fragmentation hiérarchique des arêtes groupant celles similaires ensemble. Ces groupes sont alors rendues en utilisant des techniques de traitement d'images combinant du *splatting* basé sur la distance et de la squelettisation de forme. Dans [68], Ersoy *et al.* présentent une technique de regroupements d'arêtes basée sur une fragmentation hiérarchique agglomérative des arêtes (voir Figure 3.2(e)). Leur processus pour créer les regroupements d'arêtes utilise des techniques de traitement d'image. Leur fragmentation d'arêtes produit des groupes d'arêtes qui ont de fortes similarités géométriques. Pour chacun de ces groupes, une fine enveloppe les entourant est calculée. Le squelette de l'enveloppe (soit une courbe localement centrée par rapport à la forme de l'enveloppe) est alors utilisé pour guider un processus de regroupement d'arêtes. Une technique plus récente de Hurter *et al.* [101] reposent sur des méthodes de traitement d'image pour générer des regroupements d'arêtes (voir Figure 3.2(g)). Leur méthode transforme un dessin de graphe en une carte de densité en utilisant une estimation de densité par noyau. Cette carte est ensuite utilisée pour créer des regroupements d'arêtes en déplaçant des points de contrôle des arêtes convoluées vers les maxima locaux de la carte. Ce processus est répété itérativement pour augmenter l'effet de regroupement. Leur méthode permet également de créer des regroupements d'arêtes en évitant des obstacles dans le dessin.

3.2 *Winding Roads* : un algorithme de regroupement d'arêtes en faisceaux dans le plan.

Cette section vise à présenter et détailler *Winding Roads* : notre algorithme de regroupement d'arêtes en faisceaux pour les dessins de graphe dans le plan. Cet algorithme est intuitif dans son fonctionnement et flexible quant au niveau de réduction d'occlusion souhaité. Son implémentation peut de plus tirer profit des architectures multi-cœurs pour optimiser grandement son temps d'exécution.

3.2.1 Vue d'ensemble de la méthode

L'idée principale de notre technique est de faire du routage d'arêtes afin d'en former des faisceaux. Dans un premier temps, nous calculons une discrétisation du plan en cellules à partir des positions des sommets du graphe à traiter et fabriquons ainsi une grille . Les sommets originaux du graphe sont alors connectés aux sommets de la grille qui leur sont les plus proches. Cette grille est ensuite utilisée pour router les arêtes en utilisant un algorithme de plus court chemin. La métaphore des routes et autoroutes est alors utilisée pour regrouper les arêtes. Ainsi, comme les autoroutes attirent plus de conducteurs que les routes classiques, nous utilisons les chemins les plus empruntés pour former des faisceaux d'arêtes. Cette opération est réalisée en exécutant plusieurs fois l'étape de routage par plus court chemin et en modifiant les poids des arêtes de la grille entre chaque itération. La Figure 3.4 présente un diagramme résumant les différentes étapes de notre méthode.

3.2.2 Détails de l'algorithme

Nous présentons dans cette section les détails de l'implémentation de notre méthode de regroupement d'arêtes. Nous appliquerons notre algorithme sur les graphes introduits à la Figure 3.5, à savoir le graphe biparti complet $K_{5,5}$ (voir Définition 2.27) et le sous réseau européen du réseau mondial des transports aériens de l'année 2000 (source des données : projet ANR Spangeo), afin d'illustrer les différentes étapes.

3.2.2.1 Calcul de la grille

Afin de router les arêtes par une méthode de plus court chemin, nous calculons une grille sur laquelle nous connectons les sommets originaux du graphe. Ce type de graphe est calculé en discrétisant le plan en cellules à partir de la position des sommets. Dans ce paragraphe, nous présentons les différentes approches que nous avons expérimentées pour calculer cette grille.

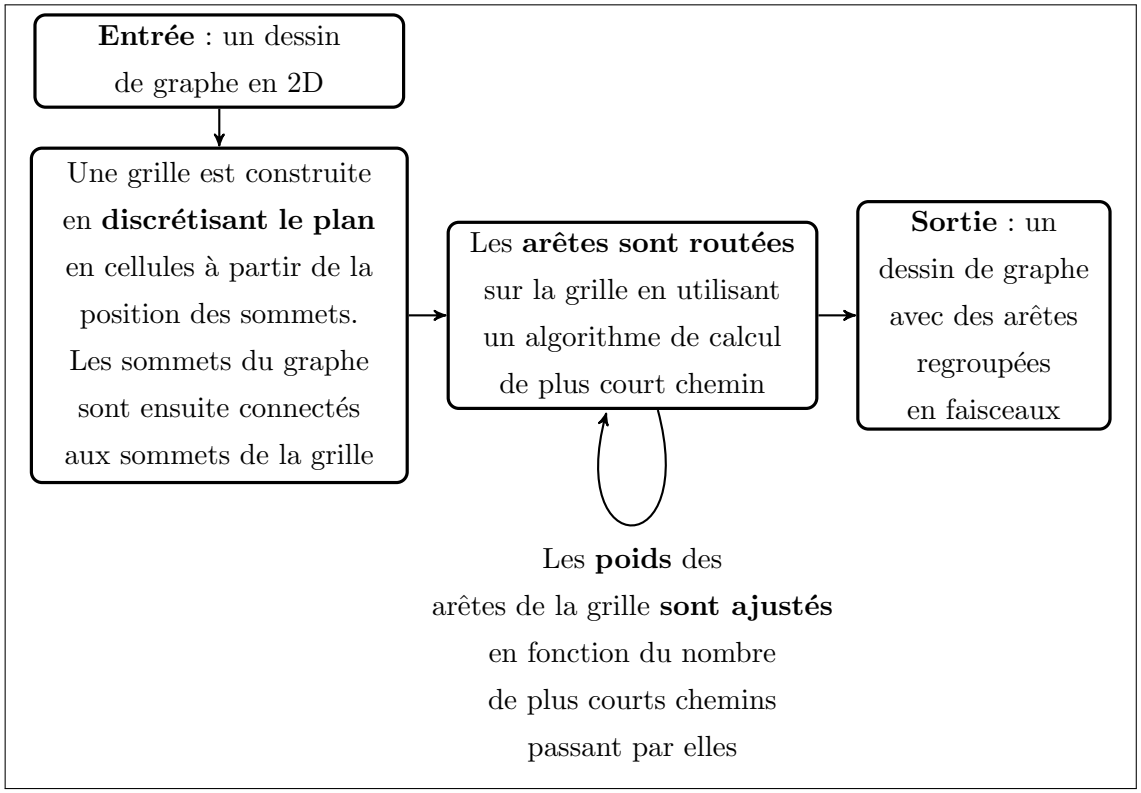
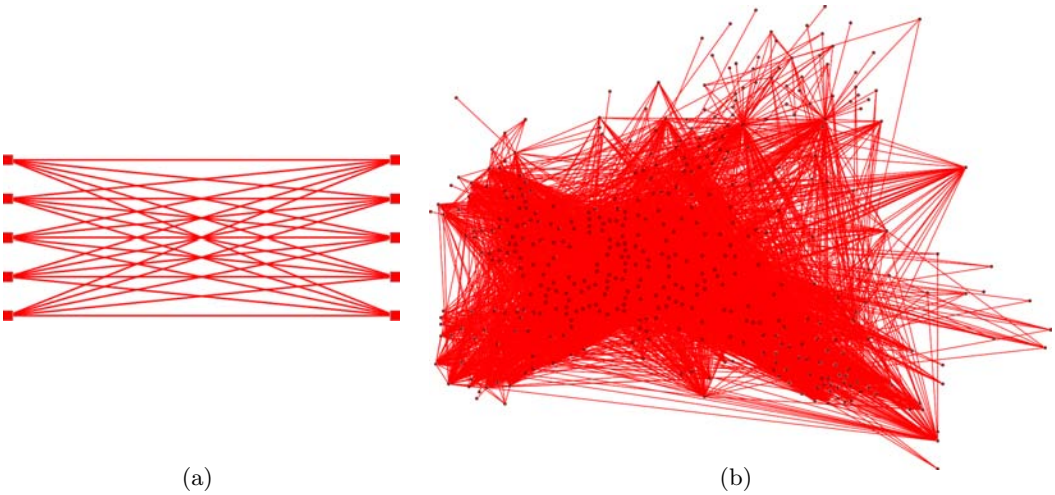


FIGURE 3.4: Diagramme illustrant les différentes étapes de notre méthode de regroupement d'arêtes en faisceaux.

Dans [46], Cui *et al.* utilisent une grille régulière pour discrétiser le plan. Cette grille est utilisée pour agréger les arêtes qui ont la même orientation. L'utilisation d'une grille régulière à granularité fine permettraient en effet un routage de haute qualité des arêtes. En effet, la grille se doit d'être précise afin de pouvoir router les arêtes à travers des régions denses du dessin du graphe. Cependant, disposer d'une grille de très large taille soulèvent deux problèmes majeurs. Premièrement, une multitude de routes possibles est générée ce qui peut nuire au regroupement de longues arêtes. Deuxièmement, la grille peut contenir un très grand nombre de sommets, facteur de $|V|^2$ ($|V|$ étant le nombre de sommets originaux), rendant l'approche coûteuse voire inacceptable en termes de calcul de plus courts chemins et de consommation mémoire.

Pour obtenir une grille multi-résolutions, un *quadtree* [72] peut être utilisé (voir section 2.2.2). Avec ce type de structure, le plan est récursivement décomposé en quatre parties jusqu'à ce que l'une d'entre elles contiennent au plus un élément. Dans notre cas, cet élément correspond à l'un des sommets originaux du graphe. La Figure 3.6 présente des exemples de grilles obtenues avec cette méthode. Une telle approche est efficace en termes de temps de calcul puisque sa complexité est en $O(|V| \cdot log(|V|))$. La taille de la grille générée est relativement raisonnable mais le principal inconvénient est que les chemins



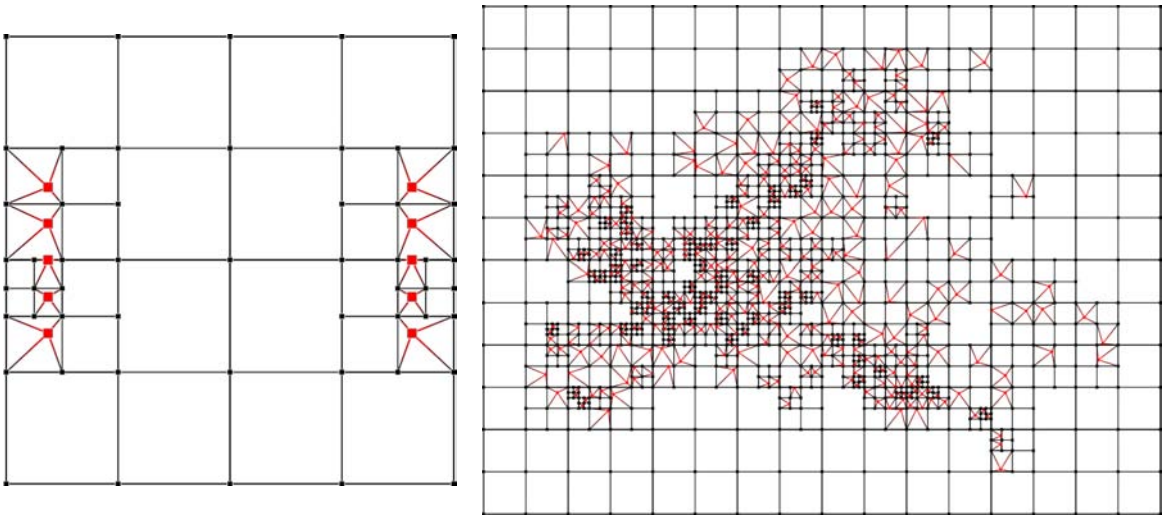
(a) (b)

FIGURE 3.5: Les graphes qui nous serviront à illustrer les différentes étapes de notre méthode de regroupement d'arêtes. (a) Le graphe biparti complet $K_{5,5}$. (b) Le sous-réseau européen du réseau mondial des transports aériens de l'année 2000 (source des données : projet ANR Spangeo)

promus lors du routage seront majoritairement horizontaux et verticaux ce qui induira un effet zigzag sur la forme finale du dessin des arêtes.

Les diagrammes de Voronoï [187] (voir section 2.2.4) peuvent également être utilisés pour générer la grille. Dans un tel diagramme, les cellules sont des régions du plan contenant les points les plus proches de leur site (correspondant dans notre cas à la position d'un des sommets du graphe) que de tout autre site. Pour plus de détails sur les diagrammes de Voronoï, le lecteur peut consulter l'étude de Aurenhammer [13]. La Figure 3.7 montre des grilles obtenues en utilisant cette approche. L'utilisation d'un diagramme de Voronoï classique ne garantit pas que les chevauchements entre sommets et arêtes seront évités dans le cas où les sommets ont des tailles supérieures à celle d'un point. Cependant, ce problème peut être résolu en utilisant un diagramme de Voronoï contraint (prenant en compte la taille des sommets). Cette méthode génère une grille de taille relativement petite. De plus, son calcul est rapide grâce à l'utilisation de l'algorithme de Fortune [76] qui possède une complexité en temps de $O(|V| \cdot log(|V|))$. Cependant, la grille peut contenir de très grandes cellules dans les régions clairsemées en sommets du dessin original. A cause de notre méthode de routage d'arêtes, ces grandes cellules créeront de larges détours ce qui pose problème quant à la qualité du dessin final.

L'approche que nous avons retenue pour générer la grille est basée sur l'utilisation combinée d'un *quadtree* et d'un diagramme de Voronoï. Dans un premier temps nous calculons un *quadtree* en posant une contrainte sur la taille des cellules qui seront générées, à savoir qu'une cellule continuera à être subdivisée en quatre, même si elle ne contient



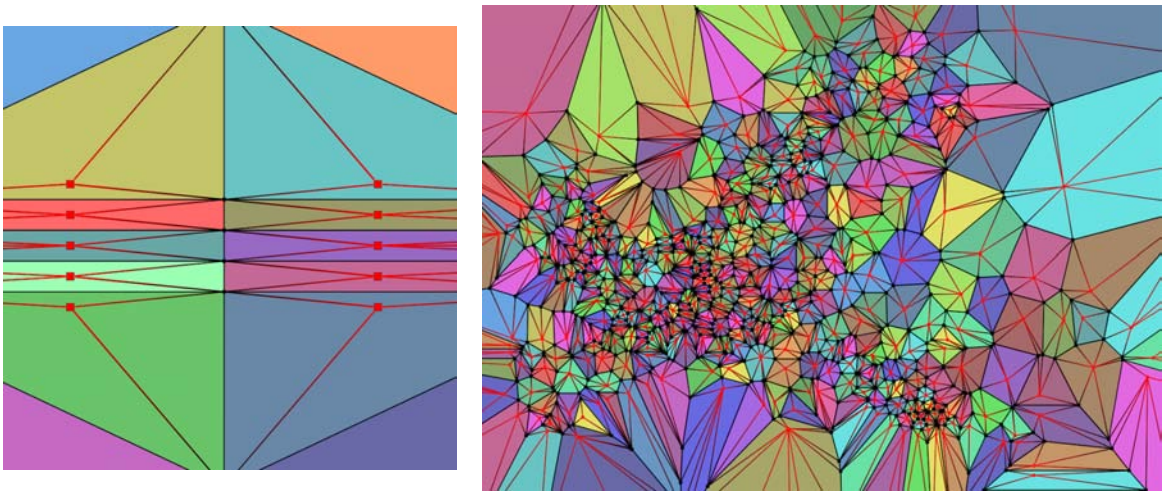
(a) (b)

FIGURE 3.6: Illustrations de grilles obtenues en utilisant un *quadtree*. Une cellule est subdivisée en quatre tant qu'elle ne contient pas au plus un sommet original du graphe. (a) Grille obtenue sur le graphe $K_{5,5}$ présenté à la Figure 3.5(a) (63 sommets / 176 arêtes). (b) Grille obtenue sur le réseau de transports aériens présenté à la Figure 3.5(b) (1695 sommets/5738 arêtes).

aucun sommet original du graphe, tant que la longueur de sa diagonale est supérieure à un certain seuil. Dans un second temps, nous calculons le diagramme de Voronoï à partir de l'ensemble des sites défini par le centre des cellules du *quadtree* précédemment calculé et la position des sommets originaux du graphe. Puisque le *quadtree* ajoute $O(|V|)$ nouveaux sommets, la complexité en temps $O(|V|\cdot log(|V|))$ du calcul global de la grille est préservée. De plus, la taille de la grille résultante est plutôt raisonnable. La Figure 3.8 présente des exemples de grilles obtenues avec cette technique.

3.2.2.2 Routage des arêtes

L'étape suivante de notre méthode consiste à router les arêtes du graphe original sur la grille obtenue dans l'étape précédente. Nous pouvons directement utiliser un algorithme de calcul de plus court chemin pour effectuer cette opération. Puisque la grille est planaire, nous obtiendrons un dessin avec les arêtes dessinées avec des lignes brisées pouvant se chevaucher sur certaines portions de leurs routes. Cependant, cette technique ne garantit pas que les arêtes suivront forcément les mêmes routes sur la grille et par conséquent crée un nombre faible de faisceaux d'arêtes. Pour augmenter l'effet de regroupement, nous utilisons la métaphore des routes et autoroutes. L'idée est de transformer les petites



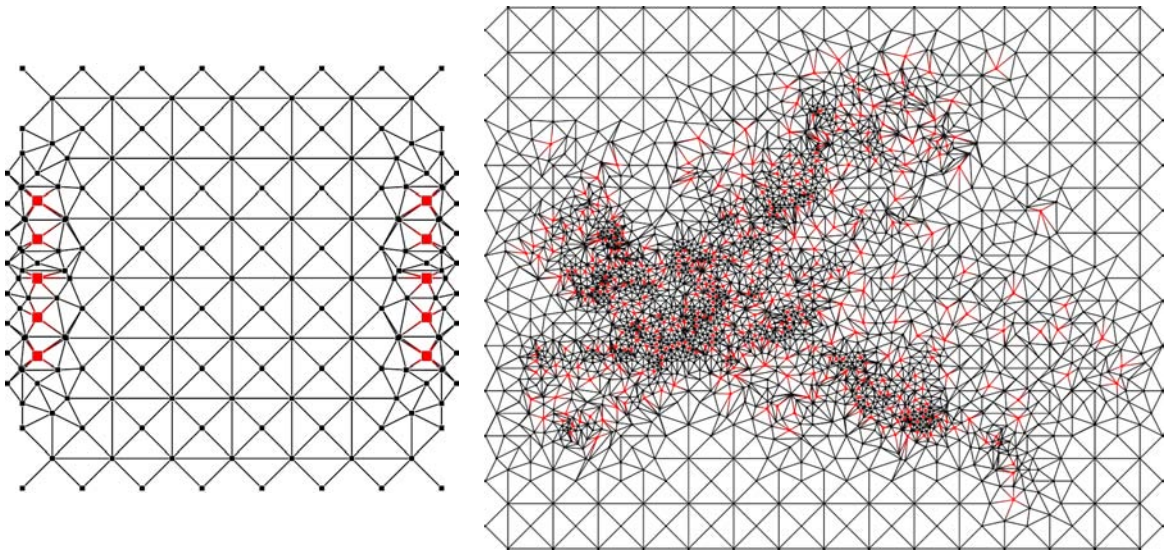
(a) (b)

FIGURE 3.7: Illustrations de grilles obtenues en utilisant un diagramme de Voronoï. Les sites des cellules correspondent aux positions des sommets originaux du graphe. Les cellules de Voronoï calculées sont repérables grâce à leur couleur. (a) Grille obtenue sur le graphe $K_{5,5}$ présenté à la Figure 3.5(a) (36 sommets / 97 arêtes). (b) Grille obtenue sur le réseau de transports aériens présenté à la Figure 3.5(b) (1305 sommets/3904 arêtes).

routes en de plus grandes si elles sont très utilisées. Nous reproduisons cet effet de la façon suivante. Nous commençons par associer à chaque arête de la grille un poids correspondant à la valeur de la distance euclidienne entre les deux sommets qu'elle relie. Puis nous calculons les plus courts chemins entre les paires de sommets connectés par une arête dans le graphe original. Une fois ces deux premières étapes effectuées, nous exécutons le processus itératif suivant. Nous ajustons les poids $w(e)$ des arêtes de la grille en fonction du nombre de plus courts chemins m passant par chacune d'entre elles de la façon suivante :

$$w(e)_{i+1} = \begin{cases} w(e)_i, \text{ if } m = 0 \\ w(e)_i/(log(m) + 1), \text{ if } m > 0 \end{cases}$$

Réduire le poids d'une arête est équivalent à la transformer en autoroute puisque passer par cet arête permet d'aller plus vite d'un point à un autre. L'étape de calcul des plus courts chemins pour chaque arête du graphe original est alors de nouveau exécutée. Le fait d'avoir ajusté les poids crée de nouveaux faisceaux d'arêtes ou renforce ceux existants car la nouvelle matrice de distances de la grille favorise la promotion des arêtes les plus fréquemment empruntées lors du routage. D'après nos expérimentations, en moyenne deux



(a) (b)

FIGURE 3.8: Illustrations de grilles obtenues en utilisant notre approche hybride *quad-tree*/Voronoï. (a) Grille obtenue sur le graphe $K_{5,5}$ présenté à la Figure 3.5(a) (36 sommets / 97 arêtes). (b) Grille obtenue sur le réseau de transports aériens présenté à la Figure 3.5(b) (2880 sommets/8561 arêtes).

itérations de notre méthode de routage sont suffisantes pour obtenir un effet de regroupement satisfaisant. La Figure 3.9 illustre l'exécution de ce processus itératif. Des exemples de dessins finaux obtenus à partir de ceux introduits à la Figure 3.5 sont également présentés à la Figure 3.10. Pour calculer les plus courts chemins sur la grille, nous utilisons le célèbre algorithme de Dijkstra [54] impliquant une complexité en temps théorique en $O(|V_{grille}|^2)$. Afin accélérer notre processus de routage, nous introduisons dans la section suivante plusieurs optimisations qui peuvent être apportées à l'implémentation.

3.2.3 Optimisation de l'implémentation

Une implémentation naïve de notre approche peut générer un dessin de graphe avec regroupement d'arêtes dans un temps raisonnable. Dans la Table 3.1, on peut voir que notre approche prend 75 secondes pour générer un dessin avec regroupement d'arête du graphe de migrations de travailleurs aux États-Unis [31] utilisé dans [46, 68, 100, 101, 146, 147, 178]. Ce temps d'exécution est plus rapide que [100] mais plus lent que [46]. Pour améliorer l'efficacité de notre méthode et la rendre utilisable sur de plus grands graphes, plusieurs optimisations peuvent être apportées.

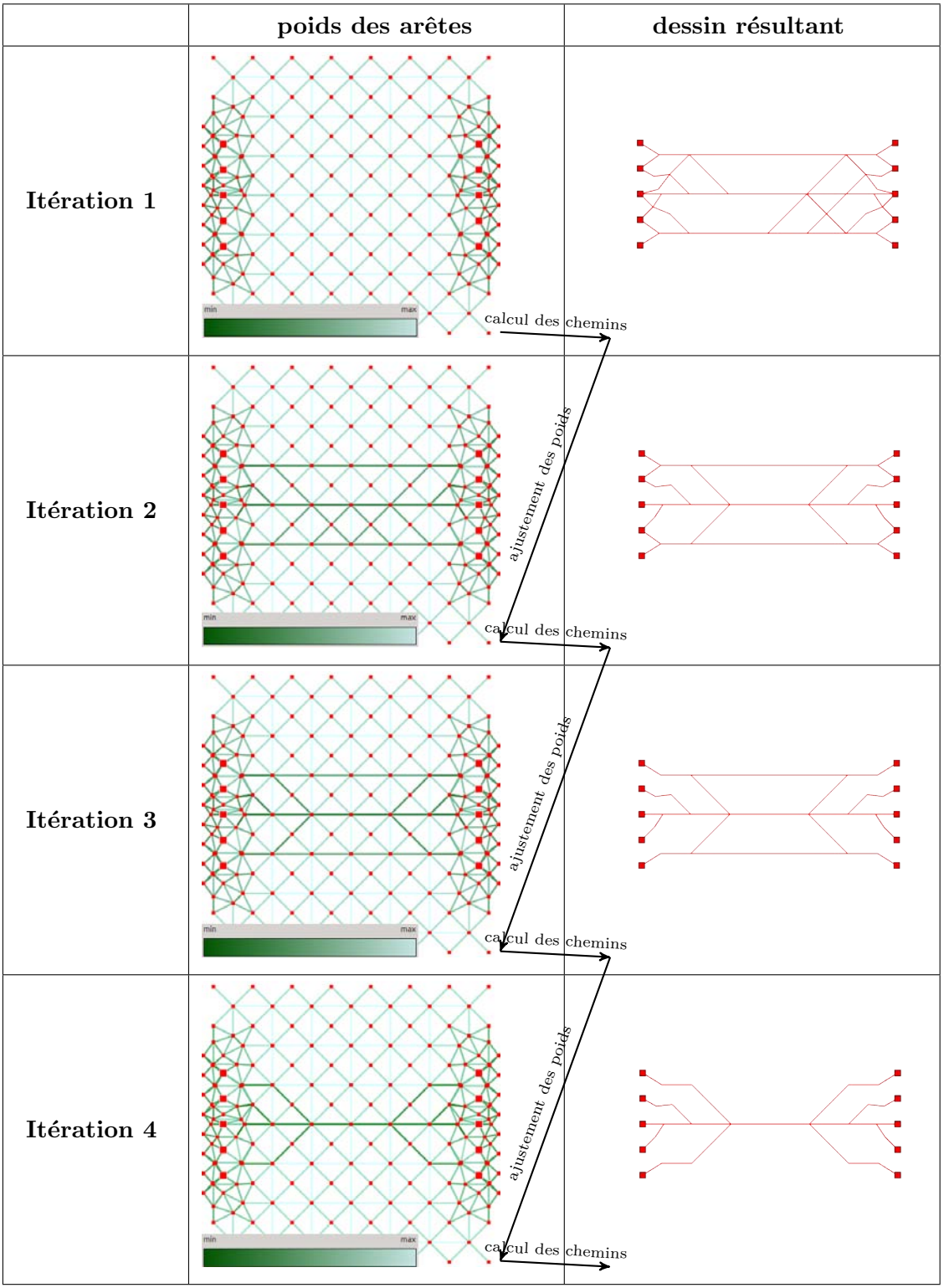
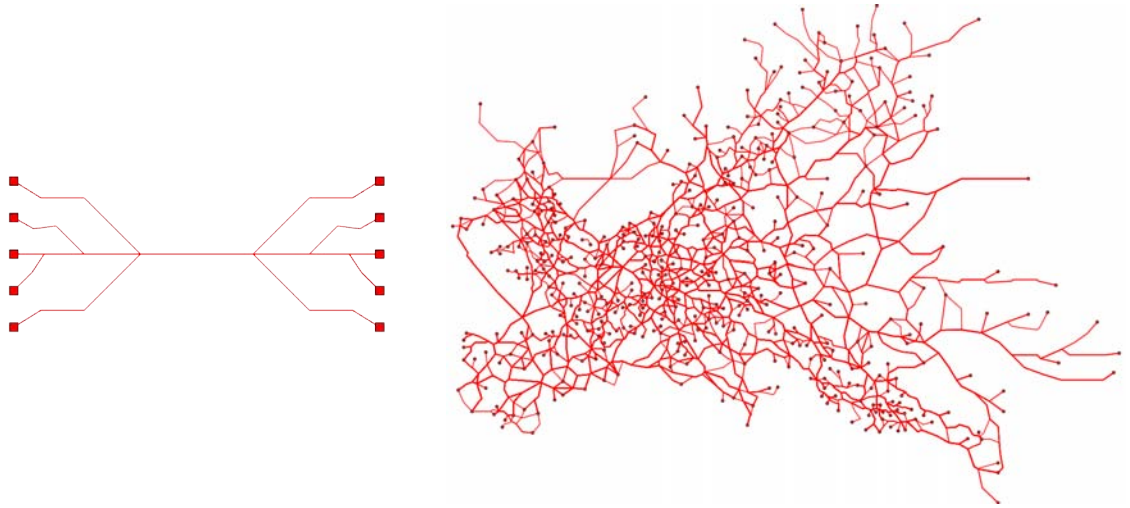
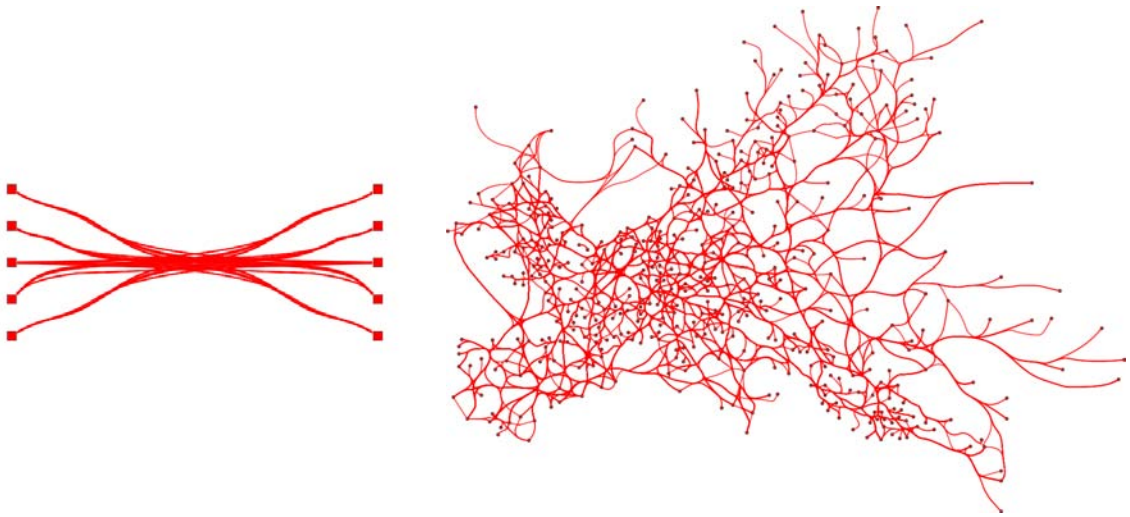


FIGURE 3.9: Illustration du processus de routage itératif pour former des faisceaux d'arêtes à partir du dessin du graphe $K_{5,5}$ introduit à la Figure 3.5(a). La colonne de gauche explicite la valeur du poids associée à chaque arête de la grille par itération. Plus une arête est verte, plus son poids est faible. La colonne de droite présente le dessin du graphe $K_{5,5}$ après chaque itération. Sur cet exemple, après quatre itérations, toutes les arêtes ont été regroupées dans un unique faisceau central.



(a) (b)



(c) (d)

FIGURE 3.10: Illustrations des dessins avec regroupement d'arêtes obtenus avec notre méthode pour les graphes introduits à la Figure 3.5. Quatre itérations de notre processus de routage ont été effectuées pour les générer. Dans les images (a) et (b), les arêtes sont rendues comme de simples poli-lignes. Dans les images (c) et (d), la forme des arêtes a été lissée en utilisant des courbes B-Spline cubiques uniformes.

(a)

	1 thread	2 threads	3 threads
sans optimisations	75.18	41.83	29.19
après 1ère optimisation	49.11	32.03	20.97
après 2nde optimisation	20.05	14.17	6.46
après 3ème optimisation	20.05	13.41	6.26

(b)

	1 thread	2 threads	4 threads
sans optimisations	98.76	56.65	36.70
après 1ère optimisation	47.20	28.64	25.68
après 2nde optimisation	24.29	14.37	10.94
après 3ème optimisation	23.65	13.41	8.74

(c)

	1 thread	2 threads	4 threads
sans optimisations	1476.68	796.15	430.24
après 1ère optimisation	229.79	176.30	115.84
après 2nde optimisation	208.05	97.36	58.70
après 3ème optimisation	208.05	95.63	55.40

TABLE 3.1: Temps d'exécution en secondes en fonction du nombre de threads utilisés de notre méthode de regroupement d'arêtes appliquée (a) au graphe de migrations de travailleurs aux États-Unis [31] utilisé dans [46, 68, 100, 101, 146, 147, 178] (1715 sommets/9780 arêtes) (b) au réseau mondial des transports aériens de l'année 2000 (1525 sommets/16479 arêtes, source des données : projet ANR Spangeo), (c) à un graphe d'appel de fonctions dans un programme (5741 sommets/11442 arêtes). La machine utilisée pour ces tests de performance disposait d'un processeur Intel(R) Core(TM)2 Extreme CPU Q9300 cadencé à 2.53GHz.

Dans cette section, nous proposons trois optimisations afin de réduire de manière significative le temps de calcul global de la méthode. Ces optimisations consistent à minimiser le nombre de plus courts chemins à calculer et à tirer profit des architectures multi-cœurs des processeurs contemporains. Ces améliorations ne changent pas le dessin en sortie de notre méthode par rapport à une implémentation basique. Elles permettent de réduire le temps d'exécution par un facteur pouvant aller jusqu'à dix, dépendant du nombre de processeurs disponibles sur la machine hôte et de la topologie du graphe.

3.2.3.1 Première optimisation

La première optimisation consiste à réduire le temps de calcul des plus courts chemins. Dans notre implémentation, nous utilisons l'algorithme de Dijkstra [54]. Les poids des arêtes sont modifiées en fonction des chemins que nous voulons promouvoir. Ainsi, nous devons utiliser un algorithme de plus court chemin valué et ne pouvons pas utiliser la technique plus rapide de calcul de plus courts chemins dans un graphe euclidien [162]. L'implémentation classique de l'algorithme de Dijkstra implique le calcul des plus courts chemins entre chaque sommet original du graphe et chaque sommet de la grille. Ces opérations ont une complexité en temps de $O(|V_{graphe}|.|E_{grille}|.log|V_{grille}|)$. Cependant, nous avons seulement besoin de calculer les plus courts chemins entre chaque sommet adjacent du graphe original sur la grille. En modifiant légèrement l'algorithme de Dijkstra, nous pouvons arrêter le calcul des chemins lorsque tous les candidats dans la file de priorité de Dijkstra sont à une distance plus grande que tous les voisins du sommet source. Dans les Tables 3.1(a), 3.1(b) et 3.1(c), on peut observer que cette modification réduit de manière significative le temps d'exécution. En particulier, dans la Table 3.1(c) on peut voir que le temps d'exécution a été réduit par un gain de facteur 6.4. Ce graphe d'appel de fonctions a été originalement dessiné avec un algorithme multi-niveaux par modèle de forces [94] qui essaie de positionner de façon aussi proche que possible les sommets connectés. Ainsi, cette optimisation permet de restreindre l'exploration des chemins sur de petites portions de la grille à chaque itération ce qui rend notre méthode efficace sur les graphes dessinés avec de tels algorithmes.

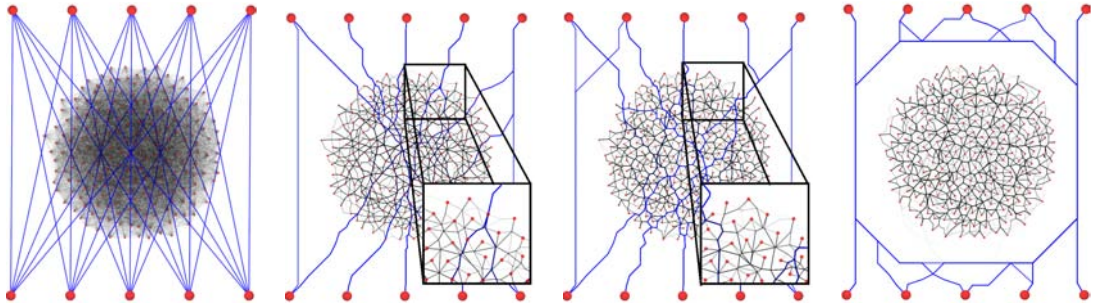
3.2.3.2 Deuxième optimisation

La deuxième optimisation vise à réduire le nombre d'appels de l'algorithme de calcul des plus courts chemins mais aussi augmenter l'efficacité de l'optimisation précédente. Après avoir calculé les plus courts chemins d'un sommet vers l'ensemble de ses voisins, il n'est plus nécessaire de considérer ce sommet dans les calculs restants. Minimiser le

nombre de sommets à traiter dans le but de considérer chaque arête du graphe correspond au problème appelé couverture de sommets [98]. Malheureusement, ce problème est NP-complet. Cependant, il est possible de calculer une couverture minimale (mais pas optimale) des sommets d'un graphe. Au lieu d'utiliser le graphe original pour récupérer les voisins d'un sommet dans la première optimisation, nous construisons une copie de ce graphe, appelée graphe de couverture de sommets, dans laquelle nous supprimons un sommet après qu'il ait été traité. Supprimer ce sommet dans le graphe de couverture fait diminuer le degré de ses voisins et va ainsi réduire l'ensemble des sommets considéré par la première optimisation. Dans la Table 3.1, on peut observer que cette amélioration réduit de manière significative le temps d'exécution de notre méthode sur chaque jeu de test.

3.2.3.3 Troisième optimisation

La troisième optimisation vise à réduire le nombre de sections critiques dans l'implémentation parallèle de notre algorithme et à créer des tâches de taille équivalente pour chaque thread. Notre méthode requiert l'exécution d'un algorithme de plus courts chemins pour chaque sommet de notre ensemble de couverture. Cette approche peut être parallélisée en calculant plusieurs plus courts chemins en simultané. Cependant, il y a des sections critiques qu'il faut traiter. Par exemple, la mise à jour du nombre de plus courts chemins passant par une arête particulière peut être effectuée par plusieurs threads en parallèle. Pour enlever ces sections critiques, nous effectuons une étape de prétraitement qui crée des ensembles de sommets qui n'entrent pas en conflit l'un avec l'autre. Dans un premier temps, nous avons essayé d'utiliser l'heuristique de coloration de graphe de Welsh et Powell [193]. Nos expérimentations ont montré que le temps d'exécution n'était pas amélioré et qu'il semblait préférable d'utiliser des sections critiques à la place. Après plusieurs expériences, nous avons trouvé que la clé était de créer une coloration locale avant d'exécuter une section parallèle. Notre algorithme maintient ainsi une liste de sommets ordonnés selon la distance dans le graphe original à leur voisinage. Avant chaque section parallèle, nous sélectionnons les k (où k correspond au nombre de threads) premiers sommets qui ne sont pas connectés. Cette opération permet d'éviter de traiter la même arête plusieurs fois et génère un ensemble de tâches de taille plus homogènes. En effet, la première optimisation accélère le calcul des plus courts chemins si les voisins sont proches l'un de l'autre dans le dessin original du graphe. Ainsi, si on traite en parallèle des sommets ayant des voisins proches et éloignés, le temps d'exécution de chaque thread peut être significativement différent. En utilisant notre ordonnancement de sommets, nous garantissons que le temps de calcul des plus courts chemins sur les ensembles de sommets que nous sélectionnons est similaire. Dans la Table 3.1, on peut voir que même si l'ordonnancement des sommets



(a) (b) (c) (d)

FIGURE 3.11: (a) Les différents niveaux de réduction d'occlusions sur une représentation de graphe. (b) Réduction des occlusions dues aux croisement d'arêtes, (c) aux chevauchements entre sommets et arêtes et (d) réduction d'occlusions dans les zones denses.

ajoute des étapes de calcul supplémentaires, il permet quand même d'accélérer le temps d'exécution global de notre méthode.

3.2.4 Niveaux de réduction d'occlusion

Comme décrit dans les sections précédentes, notre méthode consiste à router les arêtes originales d'un graphe sur une grille en utilisant l'algorithme bien connu de Dijkstra. L'utilisation de plus courts chemins valués nous permet de définir plusieurs niveaux de réduction d'occlusions, soit en adaptant les poids des arêtes de la grille soit en évitant un chemin en particulier. Dans la Figure 3.11, on peut observer les différents niveaux de réduction d'occlusions que nous proposons.

3.2.4.1 Réduction d'occlusions arête-arête

La réduction d'occlusions arête-arête que nous définissons correspond à celle utilisée dans [46, 100], à savoir seulement l'occlusion due aux croisements d'arêtes est réduite. Dans ce cas, les faisceaux d'arêtes peuvent chevaucher les sommets du graphe original. Par exemple, la Figure 3.11(b) montre le résultat obtenu sur le graphe de la Figure 3.11(a) en réduisant uniquement les occlusions arête-arête. On peut voir dans la vue détaillée que les arêtes bleues ont été routées sur les sommets originaux du graphe. Pour obtenir ce niveau de réduction d'occlusions, il suffit simplement de considérer chaque arête de la grille lors du routage des arêtes originales. En particulier, les arêtes *sommet-grille* (i.e. les arêtes reliant les sommets originaux du graphe aux sommets de la grille) peuvent être utilisées lors du calcul des plus courts chemins sur la grille.

3.2.4.2 Réduction d'occlusions sommet-arête

La réduction d'occlusions arête-arête permet uniquement de réduire l'occlusion due aux croisements d'arêtes. Mais il est également intéressant de pouvoir réduire l'occlusion due aux chevauchements entre sommets et arêtes. Notre méthode permet de le faire simplement en ne considérant pas les sommets du graphe original lors du calcul des plus courts chemins sur la grille. Par exemple, on peut observer dans la Figure 3.11(c) que les arêtes bleues on été routées autour des sommets originaux du graphe alors qu'elles les chevauchaient en utilisant seulement la réduction d'occlusions arête-arête. Nous obtenons ce niveau de réduction d'occlusions en interdisant la promotion d'arêtes sommet-grille lors de l'exécution de l'algorithme de Dijkstra. Comme ces arêtes sommet-grille ne sont pas prises en compte lors du calcul des plus courts chemins, aucune arête originale ne peut être routée sur un sommet original du graphe.

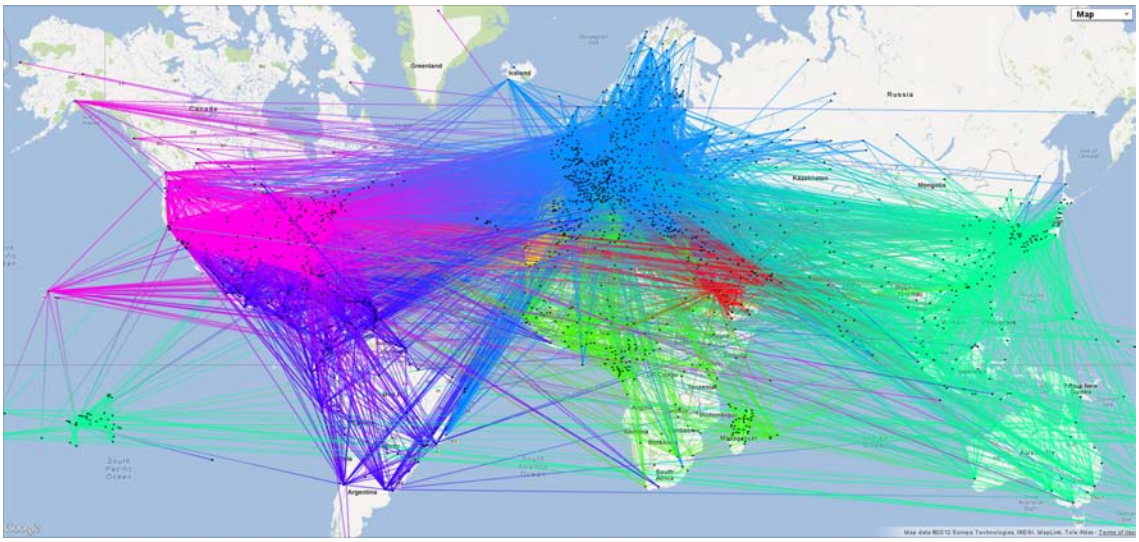
3.2.4.3 Réduction d'occlusions dans les zones denses

Nous pouvons encore réduire les problèmes d'occlusion en forçant des chemins à passer par des régions éparses du dessin. Par exemple, dans la Figure 3.11(d), on peut voir que toutes les arêtes bleues ont été routées autour du sous-graphe dense situé au milieu du dessin. En effet, si ces arêtes avaient été routées à travers ce sous-graphe, leurs formes finales auraient été très sinueuses car la région correspondante de la grille contient beaucoup de cellules. Ainsi, la longueur d'une arête routée à travers cette zone du graphe est plus grande que la distance euclidienne entre ses extrémités. Pour éviter de router des arêtes dans des zones très denses, nous adaptons les poids initiaux des arêtes de la grille. Soit $length(e)$ la distance euclidienne entre deux sommets de la grille (i.e. la longueur d'une arête). Nous calculons les nouveaux poids selon la formule suivante : $w(e) = length(e)^{\alpha}$, le paramètre α servant à augmenter ou réduire les occlusions dans les zones denses. Une valeur de α plus petite que 1 permet de promouvoir des chemins en dehors des régions denses.

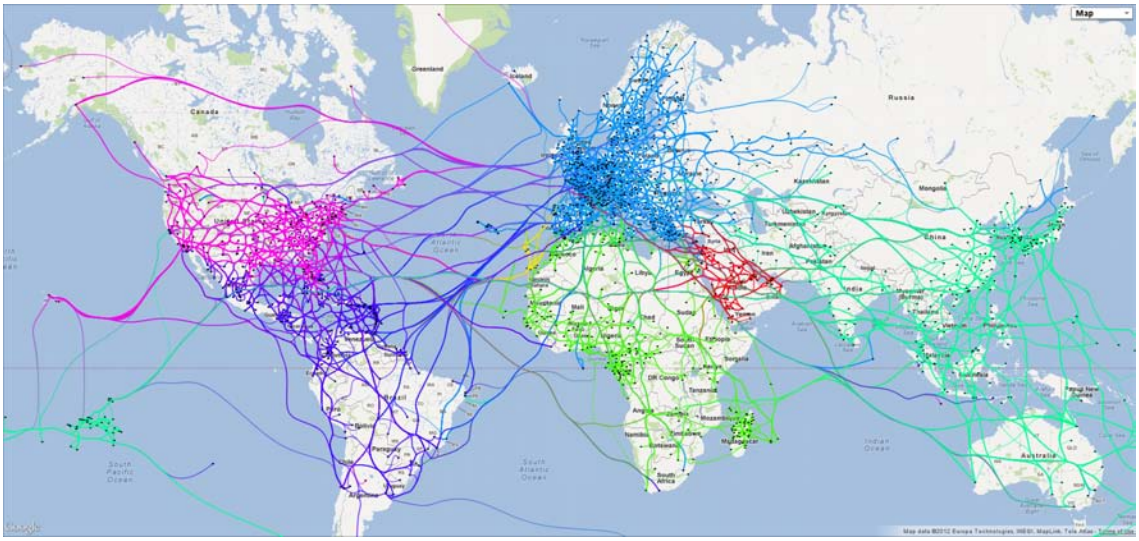
3.2.5 Exemples d'application sur des réseaux géographiques

La Figure 3.12 présente le dessin original du réseau d'interconnexions des aéroports internationaux en 2004 (1519 sommets/16096 arêtes, source des données : projet ANR Spangeo) ainsi que le résultat de l'application de notre méthode de regroupement d'arêtes dessus. Les deux dessins ont été superposés sur une carte *Google Maps*. On peut observer que le niveau d'occlusions visuelles est considérablement réduit, comme par exemple au niveau de la mer Méditerranée.

La Figure 3.13 montre le dessin original du graphe de flux de travailleurs (du domicile au lieu de travail) en France pour l'année 1975 ainsi que le résultat obtenu après avoir appliqué notre méthode de regroupement d'arêtes dessus. Ce graphe contient 36085 sommets et 316859 arêtes. On observe que sur le dessin original, le dessin des arêtes en ligne droite induit un niveau d'occlusion vraiment très élevé. Après application de notre méthode, il devient possible de distinguer la position de sommets qui étaient auparavant recouverts par de nombreuses arêtes. On peut alors identifier les grands bassins d'emploi en France à cette époque sur ce type de représentation (tels la région parisienne, toulousaine ou lyonnaise).

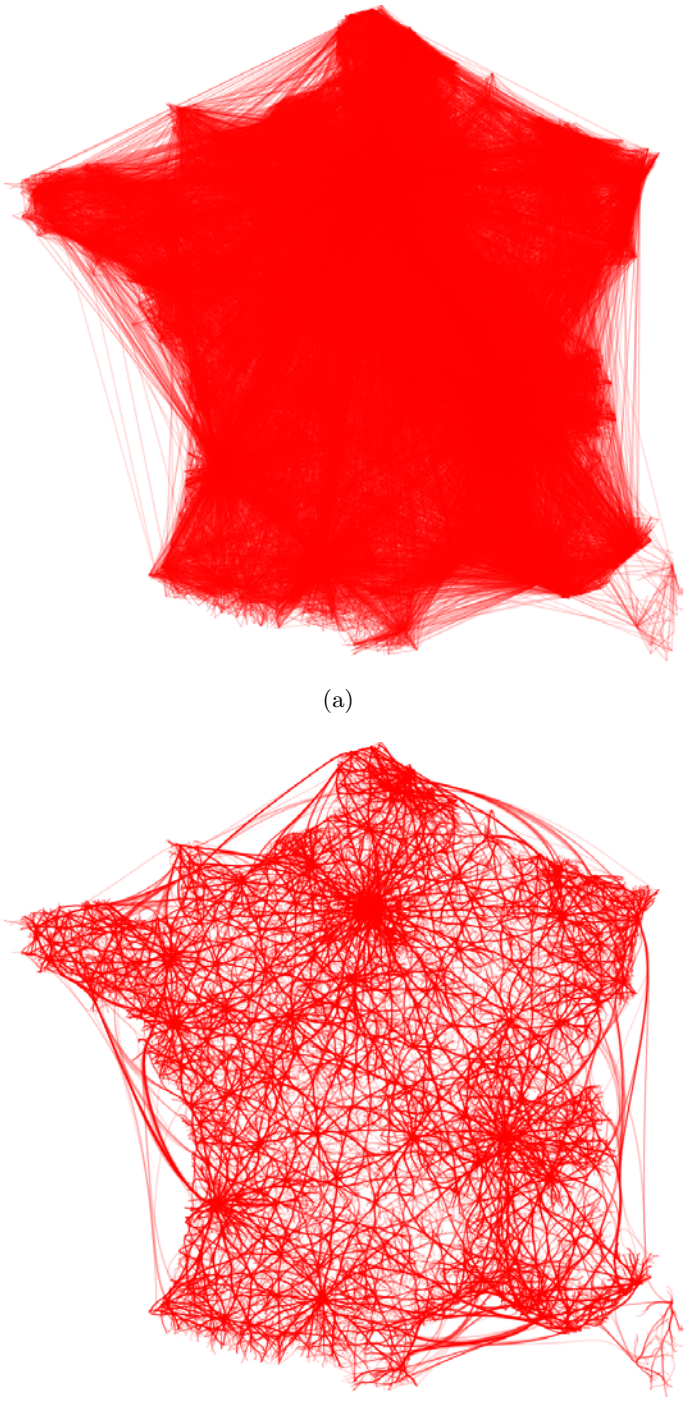


(a)



(b)

FIGURE 3.12: Application de notre méthode de regroupement d'arêtes sur le réseau d'interconnexions des aéroports internationaux en 2004 (1519 sommets/16096 arêtes, source des données : projet ANR Spangeo) (a) Dessin original. (b) Dessin avec regroupements d'arêtes. Les arêtes sont dessinées en utilisant des courbes B-Spline cubiques uniformes.



(a)

(b)

FIGURE 3.13: Application de notre méthode de regroupement d'arêtes sur le réseau de flux de travailleurs en France établi à partir des données de recensement de l'INSEE pour l'année 1975 (36085 sommets / 316859 arêtes). Ce réseau représente les déplacements effectués par chaque travailleur de son domicile à son lieu de travail. (a) Dessin original. (b) Dessin avec regroupements d'arêtes. Les arêtes sont dessinées en utilisant des courbes de Bézier.

3.3 Généralisation de l'algorithme pour les dessins de graphe dans l'espace

L'algorithme précédemment introduit fonctionne uniquement sur un dessin de graphe dans le plan. Cette section vise à introduire comment généraliser la méthode pour regrouper des arêtes sur un dessin de graphe dans l'espace. Peu de travaux se sont intéressés à ce type de dessin. Cependant, les techniques de Holten [99, 100] devraient fonctionner sur un dessin de graphe dans l'espace avec peu de modifications (voire aucunes). Une adaptation en trois dimensions de [99] a d'ailleurs été effectuée par Caserta *et al.* [39] pour faciliter la visualisation des relations entre les différents composants d'un logiciel complexe.

Dans un premier temps, nous détaillerons les modifications à apporter pour regrouper des arêtes sur n'importe quelle dessin de graphe en trois dimensions. Nous présenterons ensuite une adaptation de la technique prenant en entrée un dessin de type sphérique (représentant typiquement des lieux sur un globe terrestre).

3.3.1 Méthode de regroupement d'arêtes pour un dessin de graphe en trois dimensions

Notre méthode généralise l'algorithme présenté à la section et de la même façon utilise une technique de routage pour regrouper les arêtes. La différence majeure vient de la méthode utilisée pour générer la grille qui sera utilisée pour router les arêtes. Pour illustrer les différentes étapes, nous appliquerons la méthode sur le graphe biparti complet $K_{9,9}$ (voir Définition 2.27), introduit à la Figure 3.14, ayant été dessiné dans l'espace.

De façon similaire à la version de la méthode pour les dessins de graphe dans le plan, l'utilisation d'une grille régulière pour décomposer l'espace est à éviter. En effet, elle peut découler sur la génération d'une très grande grille afin d'avoir une granularité suffisante pour router les arêtes dans les régions denses du dessin. Nous avons donc opté, comme pour la version en deux dimensions, pour l'utilisation d'une grille multi-résolutions. Dans un premier temps, nous avons expérimenté l'utilisation d'un *octree* (voir section 2.2.3), qui est l'équivalent du *quadtree* dans l'espace. Avec ce type de structure, l'espace est récursivement décomposé en 8 parties jusqu'à ce que l'une d'elles contienne au plus un élément. Le calcul d'une telle grille est efficace car sa complexité en temps est en $O(|V| \cdot log(|V|))$. De plus, la taille de la grille générée est relativement petite. Cependant, l'utilisation d'une telle grille entraîne un routage de type orthogonal des arêtes ce qui peut nuire à la qualité du dessin final (en particulier, à cause de l'effet zig-zag sur la forme des arêtes). Un exemple de grille obtenue est présenté à la Figure 3.15.

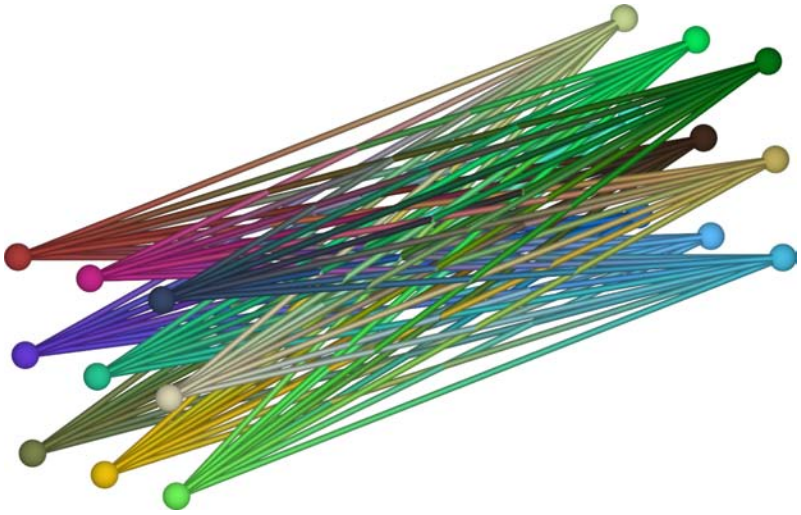


FIGURE 3.14: Le graphe biparti complet $K_{9,9}$, dessiné dans l'espace, qui nous servira à illustrer les différentes étapes de notre méthode de regroupement d'arêtes pour les dessins de graphe en trois dimensions.

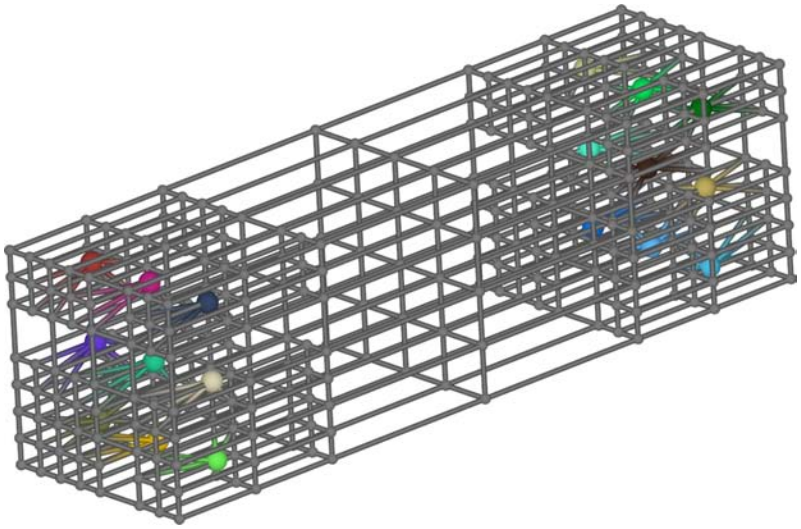


FIGURE 3.15: Illustration de la grille obtenue (427 sommets / 1342 arêtes) à partir du graphe $K9, 9$ présenté à la Figure 3.14 en utilisant un octree. Une cellule est subdivisée en huit tant qu'elle ne contient pas au plus un sommet original du graphe.

L'utilisation d'un diagramme de Voronoï en trois dimensions peut également permettre de générer une grille multi-résolutions. Dans ce diagramme, les cellules sont des régions de l'espace, sous forme de polyèdres, contenant les points les plus proches de leur site (correspondant dans notre cas à la position d'un des sommets du graphe) que de tout autre site. La grille résultante est de petite taille et peut être calculée en $O(|V| \cdot log(|V|))$. Cependant, comme pour le cas à deux dimensions, de grosses cellules peuvent être générées dans les régions éparses du dessin menant à de grands détours lors du routage des arêtes.

Un exemple de grille obtenue est présenté à la Figure 3.16.

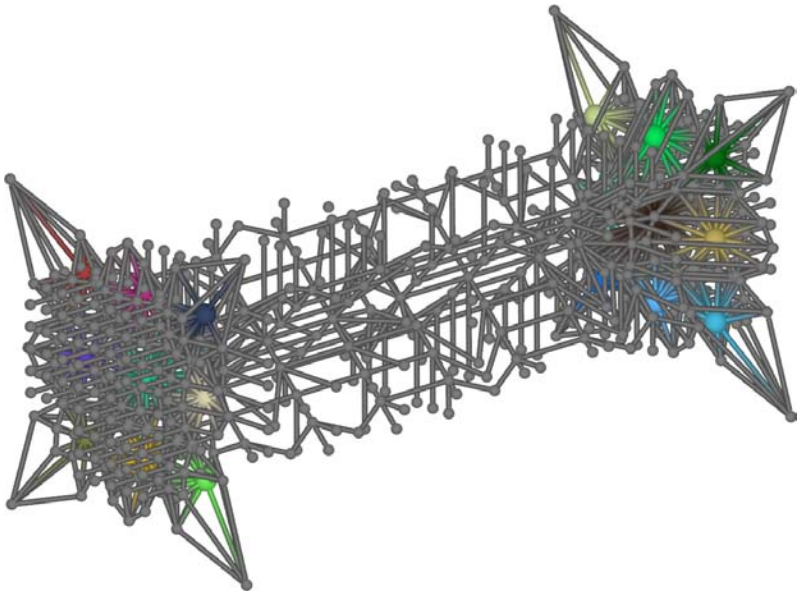


FIGURE 3.16: Illustration de la grille obtenue (879 sommets / 3434 arêtes) à partir du graphe $K9,9$ présenté à la Figure 3.14 en utilisant un diagramme de Voronoï en trois dimensions. Les sites de Voronoï correspondent aux positions des sommets du graphe original. Afin de restreindre la décomposition de l'espace, des sites ont également été rajoutés le long d'une boite englobant le dessin (non représentés sur l'image). A noter que sur cette image, les arêtes "infinies" de Voronoï ne sont pas représentées, déconnectant les cellules à la périphérie du dessin.

De manière similaire à notre méthode dans le plan, nous avons choisi une approche hybride *octree*/Voronoï pour générer la grille de routage. Dans un premier temps nous calculons un *octree* en posant une contrainte sur la taille des cellules qui seront générées, à savoir qu'une cellule continuera à être subdivisée en huit, même si elle ne contient aucun sommet original du graphe, tant que la longueur de sa diagonale est supérieure à un certain seuil. Dans un second temps, nous calculons le diagramme de Voronoï en trois dimensions à partir de l'ensemble des sites défini par le centre des cellules de l'octree précédemment calculé et la position des sommets originaux du graphe. La grille résultante possède une granularité assez fine pour effectuer un routage de qualité. Sa taille est par contre bien plus importante que celles générées avec les deux précédentes approches. Cependant, les optimisations présentées à la section 3.2.3 permettent d'effectuer le routage des arêtes de manière efficace. Un exemple de grille obtenue est présentée à la Figure 3.17.

Une fois la grille calculée, nous appliquons notre méthode de routage introduit dans la section 3.2.2 pour créer des regroupements d'arêtes et ainsi réduire l'occlusion du aux croisements d'arêtes dans le dessin original. Le résultat de la méthode appliquée au graphe $K_{9,9}$ introduit à la Figure 3.14 est présenté à la Figure 3.18.

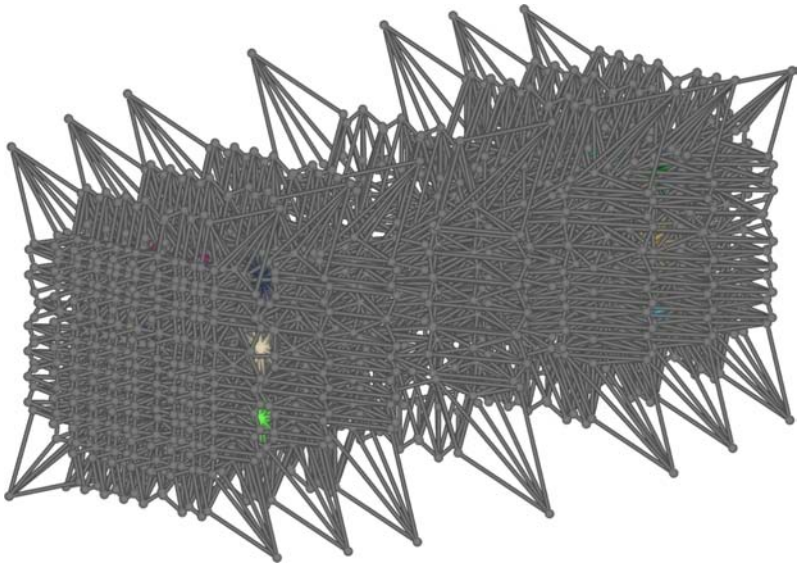
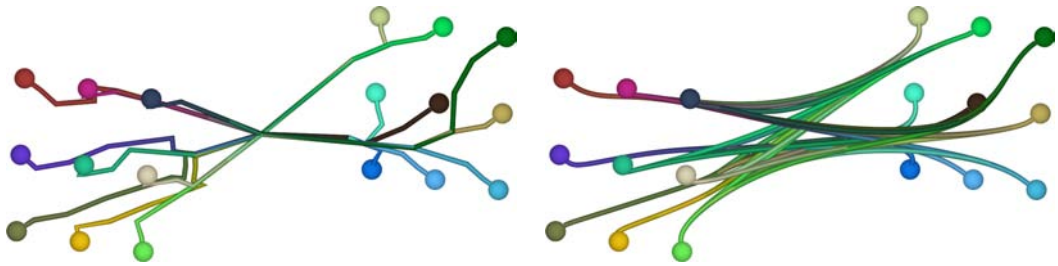


FIGURE 3.17: Illustrations de la grille obtenue (3290 sommets / 15190 arêtes) à partir du graphe $K9,9$ présenté à la Figure 3.14 en utilisant notre approche hybride octree/Voronoï 3d.

3.3.2 Adaptation de la méthode pour un dessin de graphe sphérique

Une des applications possibles de notre méthode est la visualisation des interconnexions entre les différents aéroports internationaux dans le contexte du globe terrestre. Dans ce cas, nous avons besoin de légèrement adapter notre technique de génération de la grille de routage afin de router les arêtes autour du globe et non à travers. La Figure 3.19 nous servira à illustrer nos propos. Nous devons ainsi garantir que pour toute arête originale du réseau, il existe au moins une route dans la grille qui ne traverse pas le globe. Pour y parvenir, nous effectuons une étape supplémentaire de prétraitement ajoutant des sommets temporaires avant de calculer la grille via un diagramme de Voronoï 3d. Cette étape intervient après celle de la génération de l'*octree* (voir Figure 3.19(b)). Des sommets temporaires sont dans un premier temps ajoutés à l'extérieur du globe de manière régulière et sphérique (voir Figure 3.19(c)). Cela permet de garantir que le diagramme de Voronoï contiendra pour chaque sommet original du graphe une cellule finie avec des frontières partiellement à l'extérieur du globe. Cependant, l'ajout de ces sommets n'est pas suffisant pour garantir qu'aucune arête ne sera routée à travers le globe. Pour solutionner ce problème, nous ajoutons également des sommets à l'intérieur du globe sur une sphère ayant un rayon inférieur à celui du globe. En paramétrant correctement le rayon des sphères internes et externes, nous pouvons garantir qu'il existe au moins une route pour chaque arête original du graphe qui ne traverse pas le globe. Nous avons utilisé les valeurs de rayons suivantes dans notre implémentation, r étant la valeur du rayon de la sphère sur

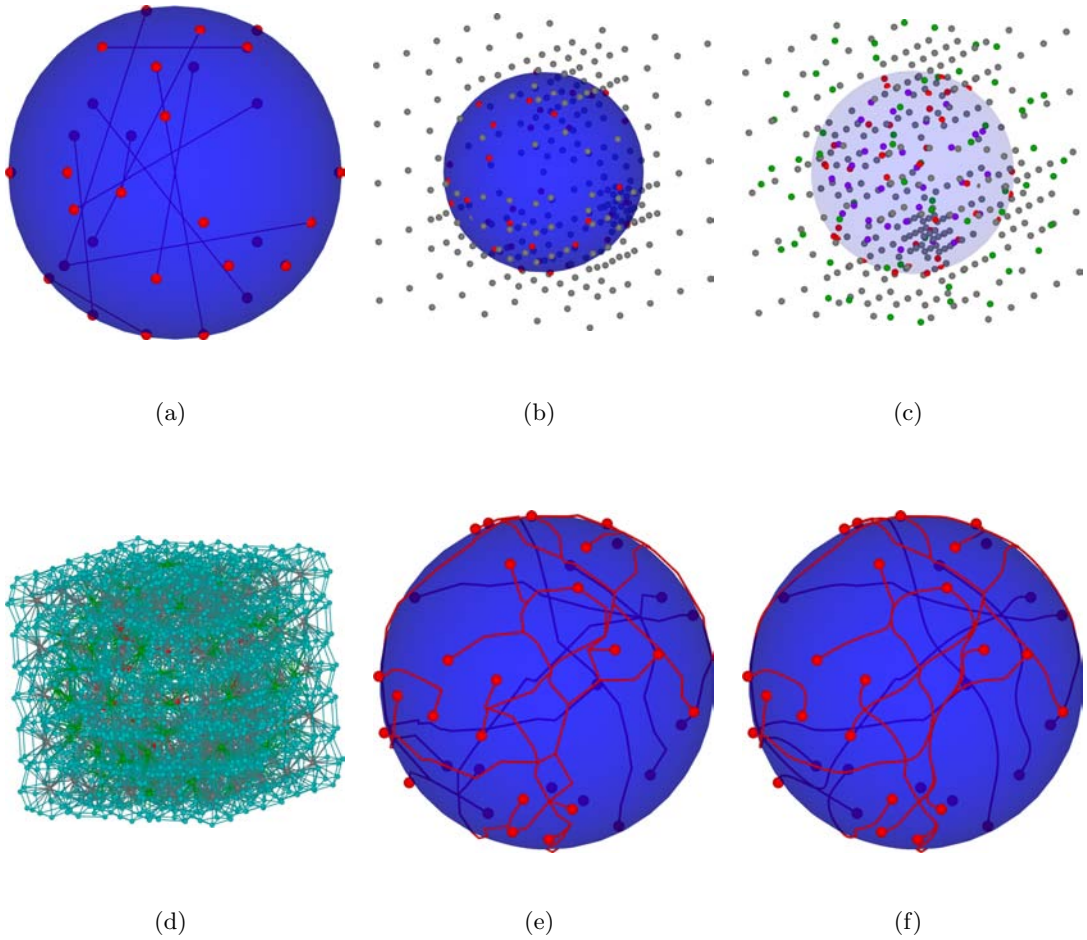


(a) (b)

FIGURE 3.18: Illustrations du dessin avec regroupement d'arêtes en trois dimensions obtenues avec notre méthode pour le graphe $K_{9,9}$ introduit à la Figure 3.14. Quatre itérations de notre processus de routage ont été effectuées pour le générer. Dans l'image (a), les arêtes sont rendues comme de simples poli-lignes. Dans l'image (b), la forme des arêtes a été lissée en utilisant des courbes de Bézier.

laquelle sont positionnés les sommets du graphe : $r-0.2*r$ pour la sphère interne, $r+0.5*r$ pour la sphère externe. La grille est ensuite calculée via la génération du diagramme de Voronoï à partir de l'ensemble des sites défini par les sommets originaux du graphe, ceux ajoutés par l'*octree* et ceux des sphères internes et externes (voir Figure 3.19(d)). Afin accélérer le processus de routage regroupant les arêtes mais aussi interdire de router à travers le globe, nous supprimons tous les sommets de la grille à l'intérieur du globe (en fonction d'un seuil sur la distance au centre). Une fois l'étape de routage effectuée, nous ajustons la position de chaque point de contrôle de chaque arête en le déplaçant sur le point du globe le plus proche (voir Figures 3.19(e) et 3.19(f)).

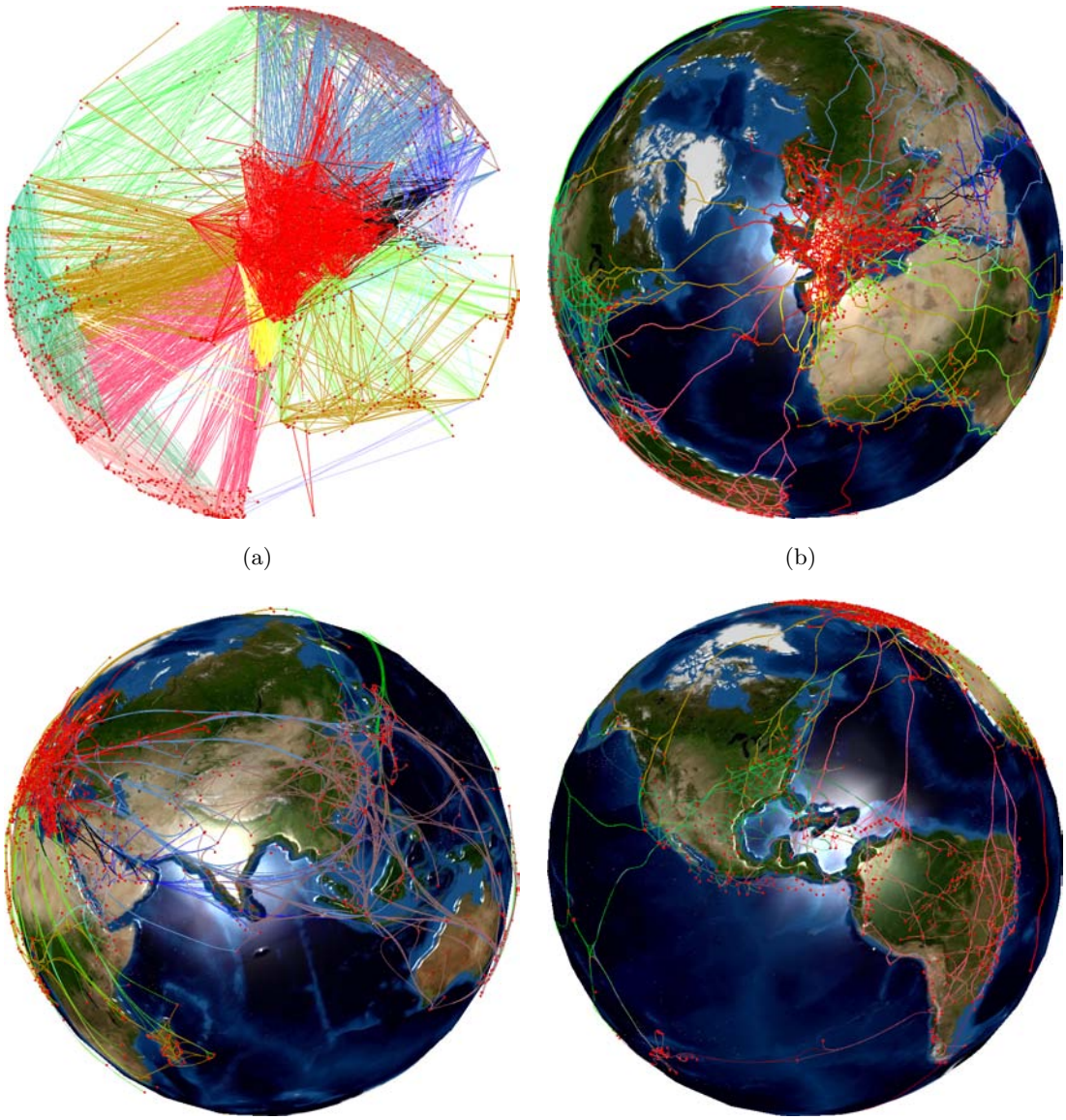
Un exemple d'application sur le réseau des interconnexions entre les aéroports internationaux de 2004 est présenté à la Figure 3.20.



(a) (b) (c)

(d) (e) (f)

FIGURE 3.19: Illustrations de l'adaptation de notre méthode de regroupement d'arêtes dans l'espace pour router des arêtes sur la surface d'une sphère. (a) Dessin original. (b) Calcul d'un *octree* et ajout de sommets temporaires positionnés aux centres des cellules générées. (c) Ajout de sommets temporaires à l'intérieur (sommets violets) et à l'extérieur (sommets verts) de la sphère de façon régulière et sphérique. (d) Calcul de la grille de routage en utilisant un diagramme de Voronoï 3d. Pour garantir qu'aucune arête ne sera routée à travers la sphère, les sommets de Voronoï à l'intérieur de la sphère (suivant un seuil sur la distance au centre) ont été supprimés. (e) et (f) Dessin final obtenu après déplacement des points de contrôle des arêtes sur le point le plus proche de la sphère (rendu poli-lignes et courbes B-Spline cubiques uniformes).



(a) (b)

(c) (d)

FIGURE 3.20: (a) Le réseau d'interconnexions des aéroports internationaux de 2004 représenté sur le globe (source des données : projet ANR Spangeo). (b), (c) et (d) Dessin obtenu après avoir routé et regroupé les arêtes autour du globe (rendu polilignes, courbes de Bézier et courbes B-Spline).

Chapitre 4

Représentation de réseaux métaboliques

Dans ce chapitre, nous présentons un exemple d'application de techniques de dessin de graphe pour représenter des réseaux biologiques particuliers : les réseaux métaboliques. Ce type de réseau permet de modéliser le métabolisme d'un organisme vivant, soit l'ensemble de réactions biochimiques se produisant dans les cellules de cet organisme. Chaque réaction transforme un ensemble de molécules (ou métabolites) appelées *substrats* en un autre ensemble de molécules appelées *produits*. Dans les systèmes biologiques, ces réactions sont activées (catalysées) par des enzymes, composées de protéines qui sont codées par des gènes (voir le travail de Lacroix *et al.* [119] pour de plus amples informations).

Quand on s'intéresse au métabolisme, on peut considérer différentes échelles qui varient en fonction des données et des questions biologiques. Par exemple, les toxicologues étudient souvent la dégradation d'une molécule particulière ; dans ce cas ils se concentrent seulement sur un petit nombre de réactions. A une plus large échelle, les biologistes vont se concentrer sur des ensemble particulier de réactions biochimiques permettant à l'organisme d'effectuer des fonctions biologiques spécifiques. Un tel ensemble de réactions est appelé *voie métabolique*. Un exemple de voie métabolique est le cycle de Krebs (ou cycle de l'acide citrique) dont le rôle est de produire des intermédiaires énergétiques nécessaires au bon fonctionnement de la chaîne respiratoire (voir Figure 4.1). Enfin, la plus grande échelle pour l'étude du métabolisme est le *réseau métabolique*, résultat de l'intégration de toutes les voies métaboliques d'un organisme dans un seul et même réseau [112, 116].

Les améliorations dans l'acquisition de données biologiques et de séquençage de génomes permettent de nos jours de reconstruire des réseaux métaboliques complets de nombreux organismes vivants. La taille et la complexité de ces réseaux empêchent de les dessiner manuellement. Il est donc nécessaire de développer des techniques de visualisation dédiées. Une représentation efficace de tels réseaux se doit de préserver la topologie des

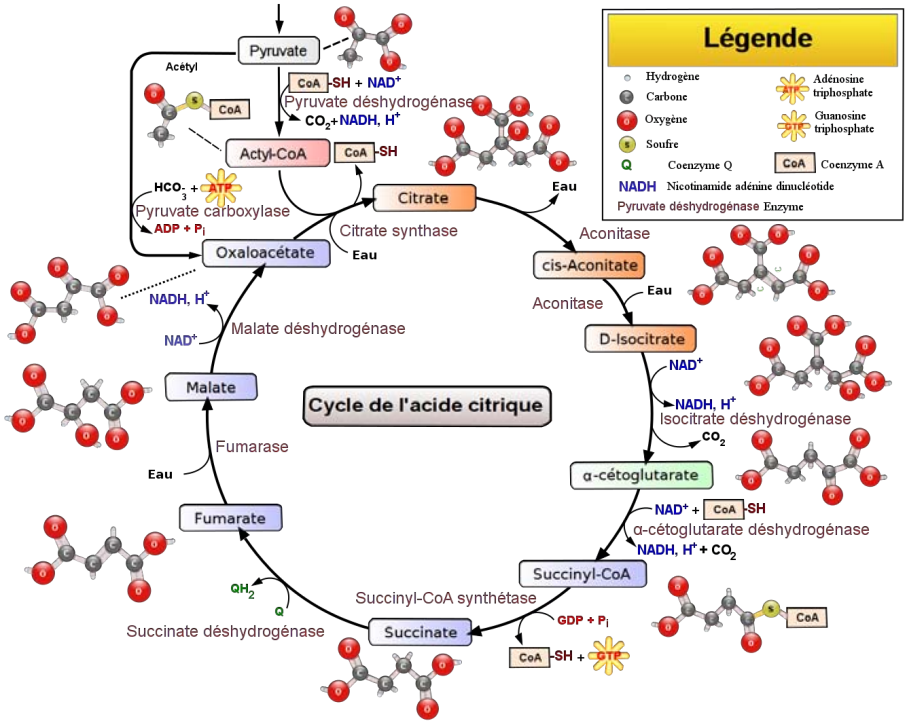


FIGURE 4.1: Exemple de voie métabolique : le cycle de Krebs ou cycle de l'acide citrique (source de l'image : Wikipédia). Cette voie permet de générer de l'énergie par l'oxydation d'acétate en dioxyde de carbone. Elle forme un cycle en raison de l'acide citrique (citrate). Cette molécule est la première consommée puis elle est régénérée par la séquence de réactions pour compléter le cycle.

voies métaboliques (voir Définition 2.3) tout en respectant les conventions de dessin biologique (e.g. dessin de cascades de réaction en ligne droite). Ces contraintes compliquent la génération automatique de telles visualisations car elles entraînent des problèmes de dessin de graphe.

Nous proposons une méthode pour dessiner un réseau métabolique entier tout en préservant le plus possible l'information relative aux voies métaboliques. Cette méthode est flexible car elle permet à l'utilisateur de piloter le dessin du réseau en fonction de l'analyse biologique à effectuer. Elle donne également la possibilité à l'utilisateur de définir si des duplications de sommets doivent être effectuées, afin de préserver ou non la topologie du réseau. Notre méthode combine de la décomposition hiérarchique de graphe ainsi que du dessin de graphe multi-niveaux pour positionner les sommets du réseau. Elle utilise ensuite une version dédiée de notre algorithme de regroupement d'arêtes afin de réduire les problèmes d'occlusions visuelles et de fournir une visualisation du réseau métabolique avec des arêtes dessinées de manière "pseudo-orthogonale". Un dessin "pseudo-orthogonale" d'une arête la représente comme une ligne brisée avec chaque angle interne entre deux segments successifs compris entre 45° et 90°. Afin de retrouver facilement l'information relative aux

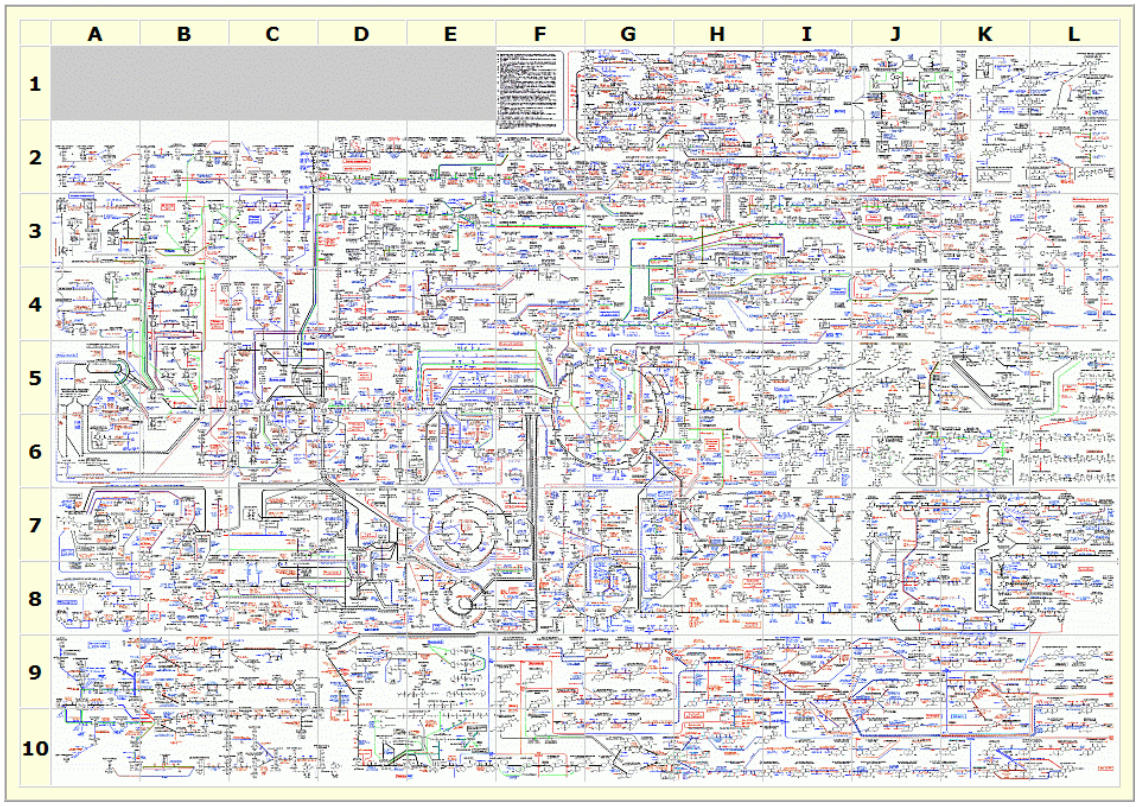
voies métaboliques, nous proposons également deux techniques d'interaction afin de mettre en exergue des voies d'intérêt. Cette méthode a été publiée dans les actes de la conférence *EuroVis 2011* via un numéro spécial du journal *Computer Graphics Forum* [124].

4.1 État de l'art

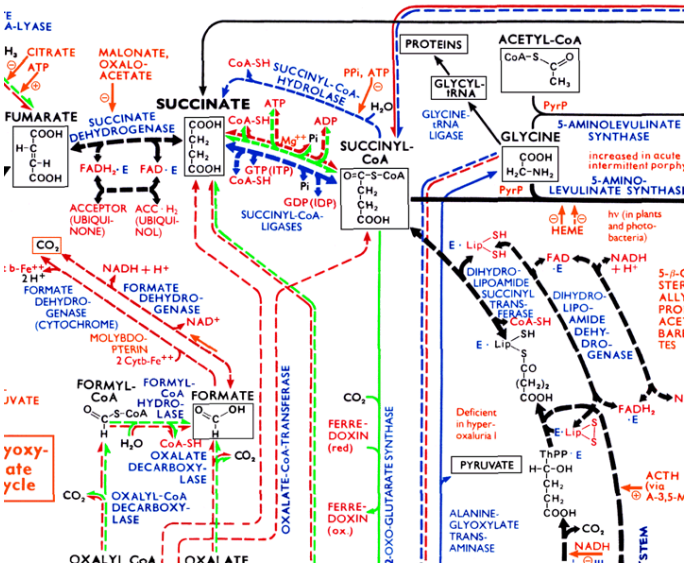
Avant la tentative de développement de techniques de dessin automatique, de nombreuses représentations manuelles ont été produites par des biochimistes pour aider à la compréhension du métabolisme. Les plus célèbres sont celles de Michal [137, 138] (voir Figure 4.2) ainsi que celles de Nicholson [141]. Ces dessins manuels et très esthétiques représentent un grand ensemble de voies métaboliques communes à de nombreux organismes.

Au niveau des méthodes de dessin automatique, la plupart des travaux sur la visualisation du métabolisme ont été faits au niveau des voies métaboliques [16, 26, 56, 81, 111, 136, 161, 191]. En raison de l'organisation hiérarchique des voies métaboliques, une approche commune consiste à utiliser des algorithmes de dessin hiérarchique (voir section 1.2.1.2) pour les représenter. Par exemple, les algorithmes de Sirava *et al.* [172] (voir Figure 4.3(a)), de Schreiber [161] (voir Figure 4.3(b)) mais également de Brandes *et al.* [26] (voir Figure 4.3(c)) dessinent des voies métaboliques de façon hiérarchique. Néanmoins, ces approches ne conviennent pas forcément aux attentes des biologistes. Elles ne respectent pas l'une des conventions de dessin biologique la plus couramment utilisée : la représentation des cycles de réactions de façon circulaire. Afin de répondre à ce problème, d'autres algorithmes [16, 81, 191] de dessin ont été élaborés. Dans un premier temps, ils détectent les éventuels cycles dans la voie métabolique. Ils les dessinent ensuite avec un algorithme de dessin circulaire et les remplacent par un méta-sommet (voir Définition 2.32). Ce processus est répété jusqu'à ce que plus aucun cycle ne soit détecté. Enfin, ils dessinent le graphe résultant en utilisant soit un algorithme de dessin par modèle de forces [16, 191] (voir section 1.2.1.2 et Figure 4.4) soit un algorithme de dessin hiérarchique dédié [81]. Enfin, récemment Meyer *et al.* [136] ont proposé un nouveau type de visualisation ne représentant pas les voies métaboliques comme des graphes mais plutôt comme des segments correspondants à des séquences d'éléments calculées à l'aide d'un procédé de linéarisation.

Ces méthodes de dessin permettent de représenter plusieurs voies métaboliques. Par exemple, dans [16] Becker *et al.* représentent jusqu'à cinq voies avec leur méthode de dessin. Cependant ces méthodes ne peuvent pas être utilisées pour représenter la plus grande échelle de l'étude du métabolisme qu'est le réseau métabolique. Le problème d'analyser des procédés biologiques couvrant plusieurs voies métaboliques apparaît dans plusieurs contextes. En particulier, il est très probable que cette situation se produise dans une

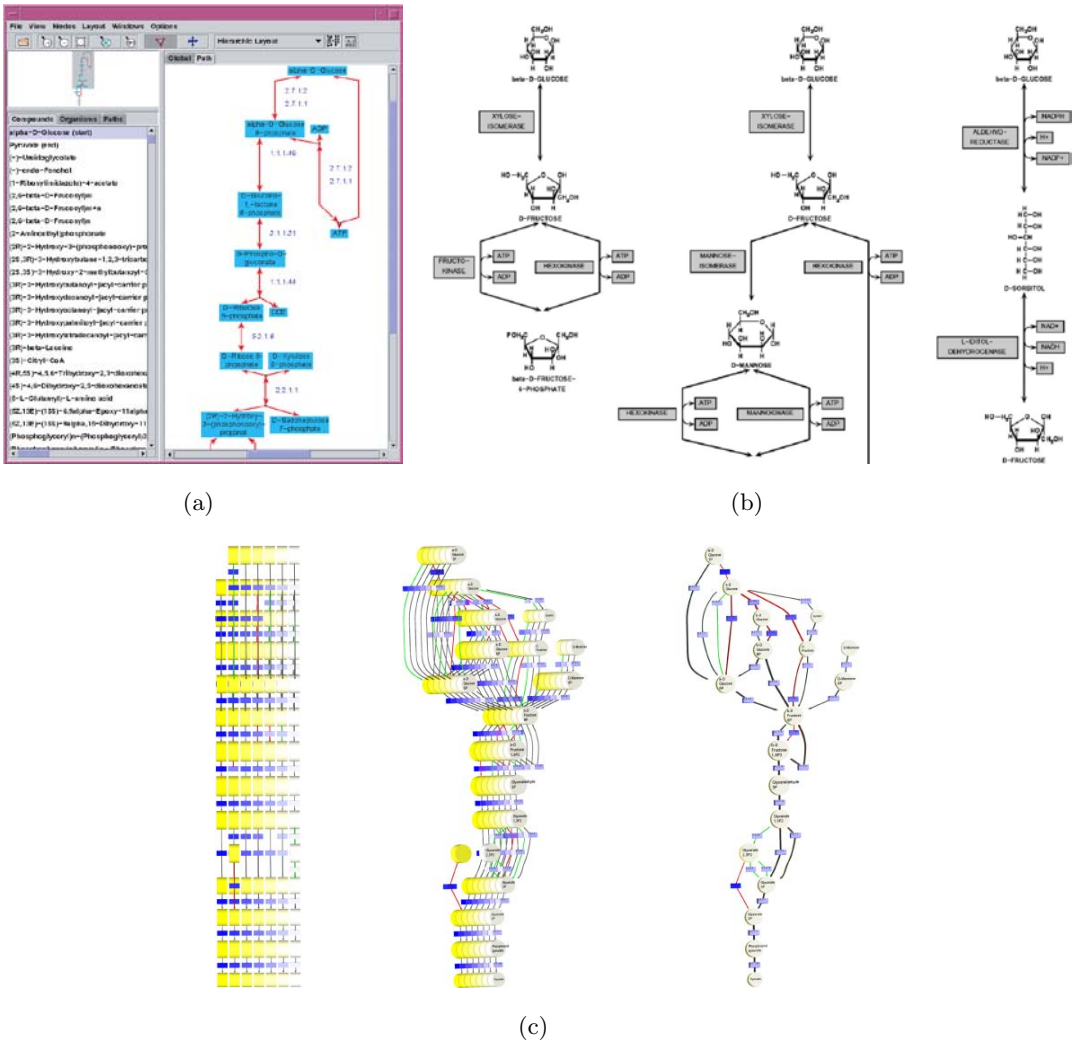


(a)



(b)

FIGURE 4.2: Le célèbre poster *Biochemical Pathways Wall Chart* produit par Michal [137] – ©1993 Boehringer Mannheim GmbH - Biochemica. (a) Vue d'ensemble du poster. (b) Vue zoomée sur la partie haute du grand cycle de réactions localisé au milieu du poster.

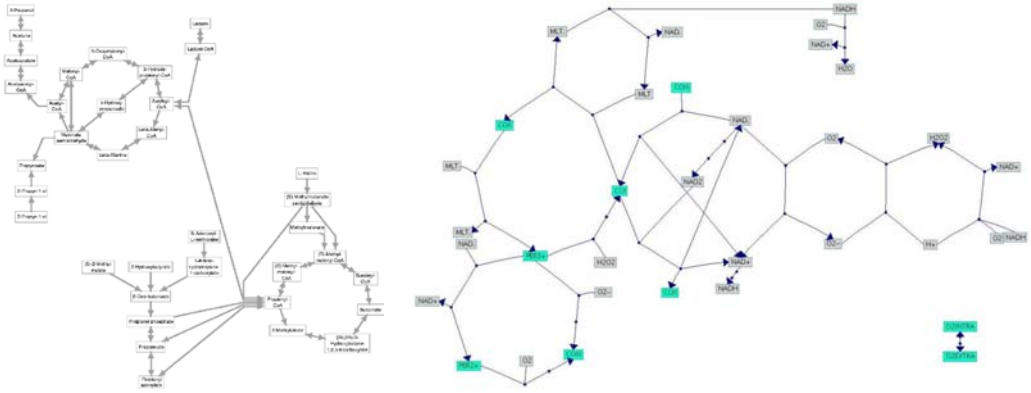


(a) (b)

(c)

FIGURE 4.3: Exemple de techniques de dessin hiérarchique de voies métaboliques. (a) Sirava *et al.* [172] – ©2002 Oxford University Press (b) Schreiber *et al.* [161] – ©2003 Australian Computer Society (c) Brandes *et al.* [26] – ©2004 Springer-Verlag

expérience non focalisée sur des voies métaboliques particulières, comme par exemple en métabolomique (science étudiant l'ensemble des métabolites présents dans une cellule, un organe ou un organisme). Ainsi, la visualisation de voies métaboliques n'est pas adaptée pour de telles tâches tout comme la visualisation du réseau métabolique ne prenant pas en compte les informations sur les voies. En effet, afin d'être utile pour visualiser le résultat d'une expérience, il est nécessaire de représenter la structure complète du réseau tout en conservant l'information contextuelle fournie par sa décomposition en voies métaboliques. Les premières approches pour représenter un réseau métabolique entier consistent en l'utilisation d'algorithmes de dessin classiques comme dans *SBML Viewer* [105], *Cell*

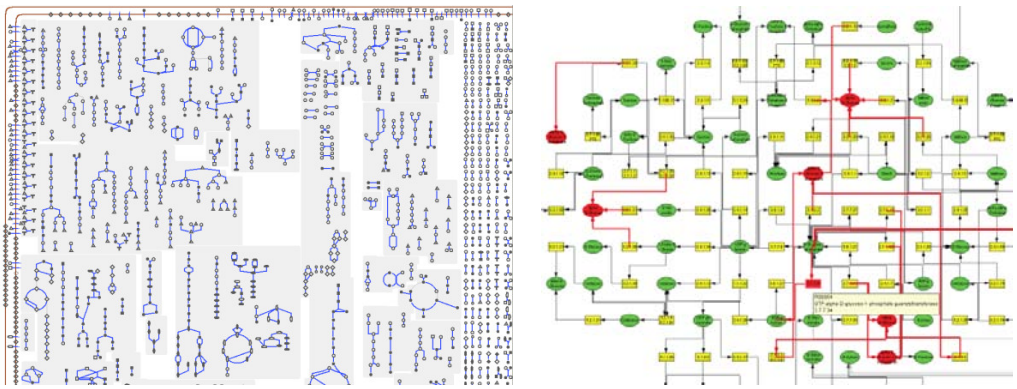


(a) (b)

FIGURE 4.4: Exemples de techniques de dessin de voies métaboliques respectant les conventions de dessin biologique. (a) Becker et Rojas [16] (b) Wegner et Kummer [191]

Designer [79] ou *Cytoscape* [165]. Avec ces outils, les réseaux métaboliques sont généralement dessinés en utilisant des algorithmes de dessin par modèle de forces (voir section 1.2.1.2). Cependant, les dessins calculés par ces algorithmes ne respectent pas les conventions de dessin biologique. Ainsi, de telles représentations ne sont pas pleinement satisfaisantes [158] du point de vue d'un biologiste. Très peu d'outils respectant les conventions de dessin biologique existent [24, 109, 144, 155]. Dans *Reactome* [109] et *Pathway Tools cellular overview diagram* [144] (voir Figure 4.5(a)), les sommets partagés par plusieurs voies métaboliques sont dupliqués et chaque voie est dessinée indépendamment. Tous les dessins des voies sont ensuite intégrés dans une seule et même visualisation. Ces dessins sont positionnés en fonction du procédé biologique auxquels ils se rapportent. Dans [155], Rohrschneider *et al.* effectuent également des duplications de sommets (voir Figure 4.5(b)). Leur approche positionne les sommets sur une grille régulière orthogonale et les arêtes sont routées sur cette grille via l'utilisation d'une fonction de minimisation de coût. Afin de réduire les problèmes d'occlusion, les auteurs autorisent de router plusieurs arêtes sur les mêmes segments résultant en une visualisation où les arêtes ont été regroupées en faisceaux. Finalement, chaque voie métabolique est remplacée par un méta-sommet et la visualisation d'un graphe quotient (voir Définition 2.32) est proposée à l'utilisateur. Ce dernier peut ensuite ouvrir/fermer les voies métaboliques si besoin est. L'un des désavantages des représentations générées par ces outils est qu'elles ne reflètent pas la topologie du réseau métabolique vu que les sommets partagés par plusieurs voies sont dupliqués.

A notre connaissance, un seul outil existe, appelé *MetaViz* [24] (voir Figure 4.6) et développé par Romain Bourqui membre de notre équipe de recherche, gardant la topologie du réseau intacte tout en mettant en exergue sa décomposition en voies métaboliques. Pour y parvenir, dans un premier temps, la décomposition chevauchante des voies métaboliques



(a) (b)

FIGURE 4.5: Exemples de méthodes dessinant un réseau métabolique complet en dupliquant des sommets. (a) Pailey et Karp [144] – ©2006 Oxford University Press (b) Rohrschneider *et al.* [155] – ©2010 Springer-Verlag

est transformée en une partition des sommets et les sous-réseaux induits sont dessinés en utilisant des algorithmes de dessin dédiés. Ensuite, le graphe quotient résultant est calculé puis dessiné via un algorithme de dessin planaire. Un tel type de visualisation permet de réduire les problèmes d'occlusion mais n'est pas pleinement satisfaisant car les utilisateurs non experts (et en particulier les biologistes) ne sont pas habitués à naviguer dans ce genre de représentation abstraite.

4.2 Dessin de réseaux métaboliques : les défis à relever

Des conventions de dessin biologique ont été établies pour faciliter l'extraction d'informations dans les représentations de processus biologiques. De telles conventions peuvent être observées dans des représentations manuelles [137, 138, 141]. Dans ces dessins, très peu de métabolites et réactions ont plus de une occurrence, c'est à dire que le nombre de duplications d'éléments est minimisé. Les cycles de réactions sont représentés de manière circulaire et les cascades de réaction en ligne droite. Il est également intéressant de noter, comme mentionné dans [155], que les arêtes sont dessinées de manière "pseudo-orthogonales". On remarque aussi dans les représentations manuelles que l'information liée à la décomposition du réseau en voies métaboliques est préservée du mieux possible. Les voies qui partagent des éléments ou effectuent des fonctions similaires sont ainsi positionnés dans une région proche du dessin. Enfin de manière générale, la longueur des arêtes ainsi que le nombre de croisements d'arêtes sont minimisés.

Le principal obstacle pour dessiner un réseau métabolique de manière automatique est de respecter ces conventions. Afin d'y parvenir, nous avons traduit ces exigences de dessin

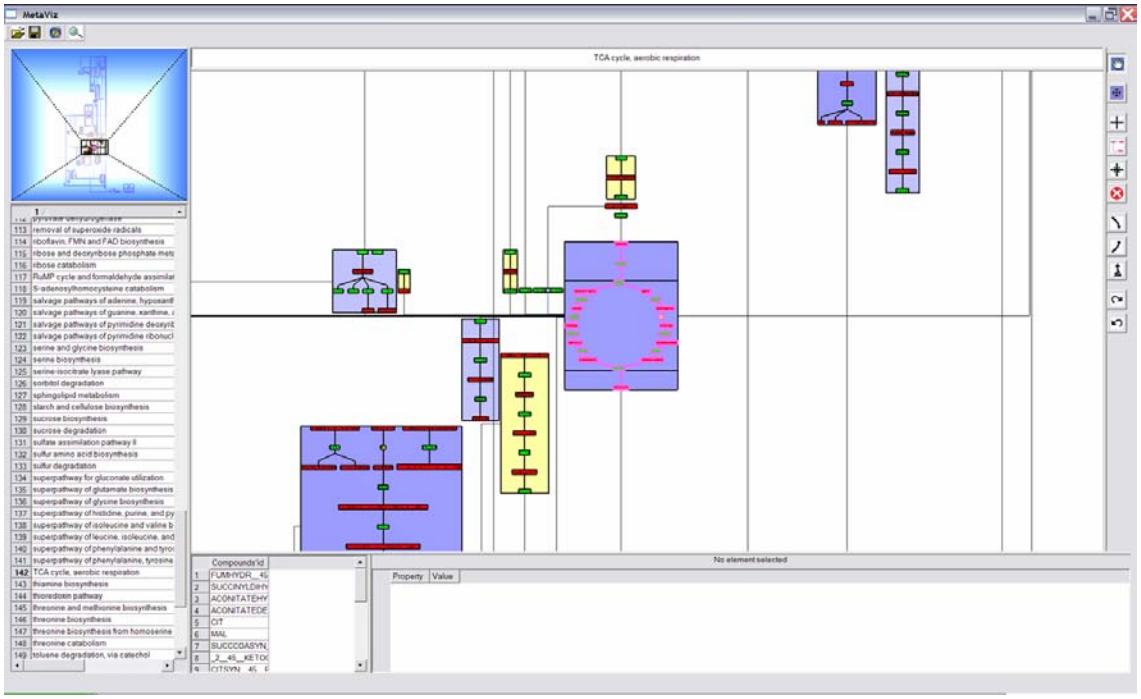


FIGURE 4.6: Capture d'écran du logiciel *MetaViz* [24] développé par Romain Bourqui. Cet outil est à notre connaissance le seul permettant de visualiser un réseau métabolique complet gardant intact la topologie du réseau tout en mettant en évidence sa décomposition en voies métaboliques.

en 5 contraintes :

- **Contrainte de proximité** : L'information relative aux voies métaboliques doit être préservée autant que possible.
- **Contrainte de duplication** : Éviter autant que possible de dupliquer des sommets afin de respecter la topologie du réseau.
- **Contrainte de dessin biologique** : Les cycles et cascades de réactions doivent être représentés suivant les conventions de dessin biologique, soit en cercle et en ligne droite.
- **Contrainte de dessin d'arêtes** : Les arêtes doivent être dessinées de manière "pseudo-orthogonales", soit comme des lignes brisées avec chaque angle interne entre deux segments successifs compris entre 45° et 90°.
- **Contrainte d'occlusion** : Le nombre de croisements d'arêtes doit être minimisé.

Chacune de ces contraintes soulève des problèmes bien connus de calcul ou de dessin de graphe. Pour respecter la *Contrainte de dessin biologique*, nous avons besoin de rechercher les plus longs cycles dans le réseau, problème connu pour être NP-complet [113]. Minimiser le nombre de croisements d'arêtes est également connu pour être NP-complet [87].

Concernant la *Contrainte de proximité*, le problème principal vient du fait que les voies métaboliques partagent souvent des métabolites et/ou réactions. Supposons par exemple que notre réseau contient 5 voies métaboliques, chacune partageant des éléments avec les 4 autres. Alors il n'existe pas de représentation dans le plan de ce réseau sans chevauchements entre les dessins des voies. Ainsi, respecter cette contrainte tout en évitant de dupliquer des sommets n'est pas si simple.

4.3 Technique de dessin automatique d'un réseau complet préservant les voies métaboliques

Dans la suite, les réseaux métaboliques sont modélisés comme des graphes bipartis $G = (V, E)$ où $V = R \cup M$, R est l'ensemble de réactions, M l'ensemble des métabolites, et $E \subseteq R \times M$. Pour une discussion plus détaillée sur les différentes manières de modéliser des réseaux métaboliques via des graphes, nous recommandons la lecture de l'article de van Helden *et al.* [182].

Notre méthode dessine un réseau métabolique entier tout en essayant de respecter simultanément les contraintes précédemment exprimées. Dans de nombreux cas, il n'existe pas de représentation pouvant remplir toutes ces exigences. En particulier, un compromis doit être trouvé entre la *Contrainte de proximité* et la *Contrainte de duplication*. En fonction de la tâche, l'utilisateur peut décider si oui ou non des duplications de sommets doivent être effectuées. Cela permet de garantir que l'une des deux contraintes sera respectée.

La Figure 4.7 présente le pipeline des différentes étapes de notre méthode. C'est un compromis entre l'approche de Bourqui *et al.* [24] et l'approche de Rohrschneider *et al.* [155]. Elle combine du partitionnement de sommets (comme dans [24]) afin de générer une décomposition hiérarchique du réseau, du placement de sommets et du regroupement d'arêtes en faisceaux (comme dans [155]) pour produire une visualisation "pseudo-orthogonale" du réseau métabolique. L'étape de placement de sommets suit une règle ascendante (*bottom-up*), signifiant que les niveaux de la hiérarchie calculée sont dessinés des plus bas à la racine.

4.3.1 Construction d'un graphe quotient multi-niveaux

Comme évoqué précédemment, un réseau métabolique peut être décomposé en voies métaboliques chevauchantes. Un graphe partitionné est usuellement modélisé par un couple (G, T_G) (voir Définition 2.30 et les travaux de Eades et Feng [62]) où G est un graphe et T_G

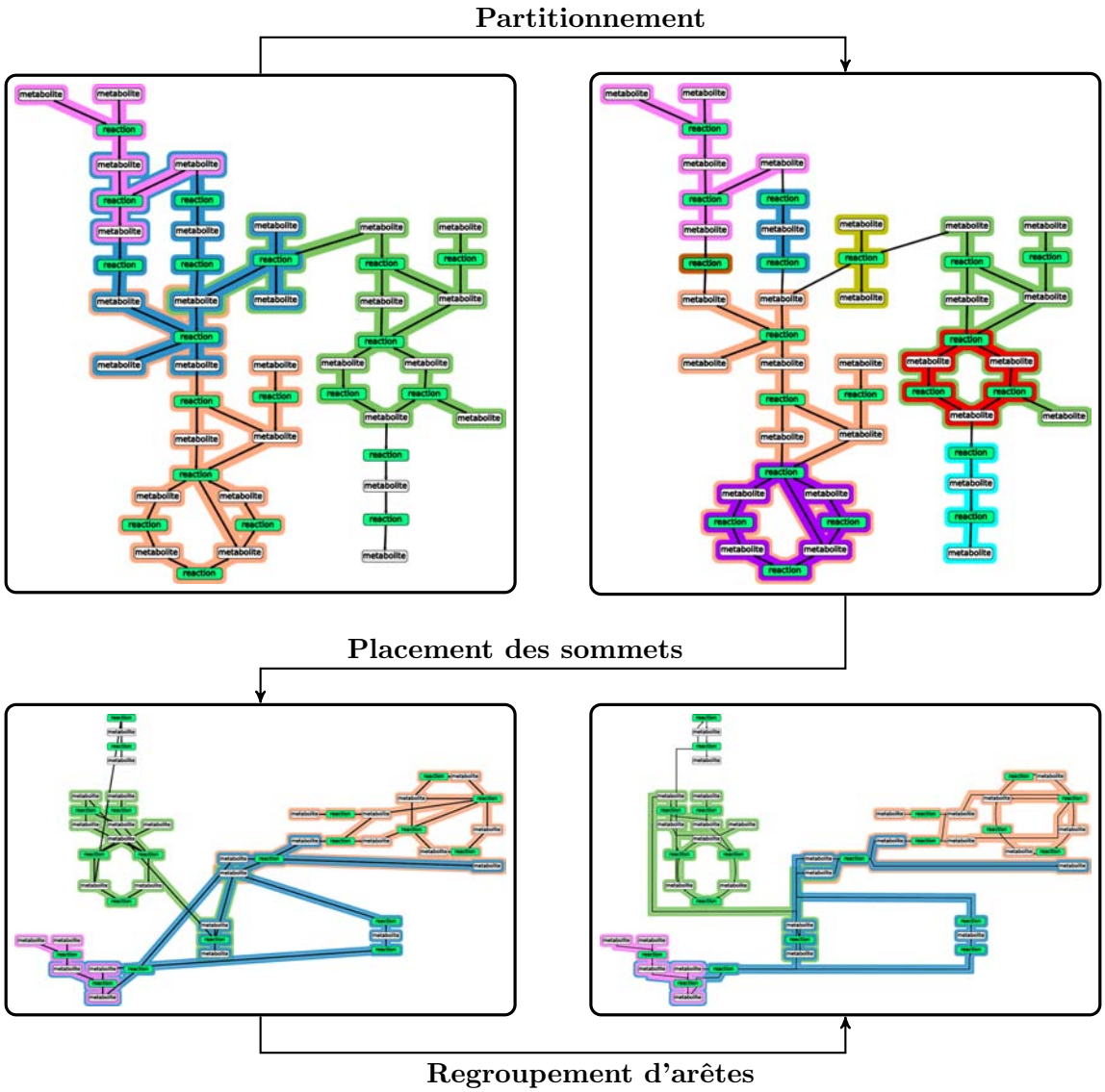


FIGURE 4.7: Pipeline de notre méthode de dessin d'un réseau métabolique. La première étape transforme une décomposition chevauchante du réseau en une partition. Ensuite les sommets sont positionnés et finalement les arêtes reliant deux sous-ensembles de la partition sont regroupées en faisceaux.

est l'arbre de hiérarchie représentant la partition multi-niveaux des sommets de G. Dans le cas d'une décomposition chevauchante, il est alors nécessaire d'utiliser un couple (G, D_G) où D_G n'est pas un arbre de hiérarchie mais plutôt un graphe acyclique orienté (DAG) de hiérarchie(voir Définition 2.28 et la thèse de Romain Bourqui [25]). Dans la Figure 4.8 (à gauche), on peut voir 4 voies métaboliques partageant des éléments. Par exemples, les voies p_1 et p_2 partagent 4 éléments (3 métabolites et 1 réaction). Pour modéliser cette relation d'inclusion, les sommets représentant les voies métaboliques p_1 et p_2 (en rose, resp. en bleu) dans la hiérarchie sont tous les deux parents de ces 4 éléments partagés (ces multiples relations de parenté sont dessinées en rouge dans le DAG de hiérarchie).

Cependant, dessiner un tel graphe décomposé sans créer de chevauchements visuels non voulus entre voies métaboliques n'est pas simple, voire impossible (voir les travaux de Simonetto *et al.* [170] sur la visualisation d'ensembles chevauchants). Pour résoudre ce problème, nous avons légèrement modifié l'algorithme de décomposition de Bourqui *et al.* [24] qui transforme un DAG de hiérarchie en un arbre de hiérarchie tout en préservant le plus possible l'information sur les voies métaboliques. Cette phase de partitionnement est composé de 3 étapes :

1. **Calcul d'un ensemble maximal de voies métaboliques indépendantes** et extraction des *sous-graphes propres* des voies restantes (i.e., le sous-graphe d'une voie métabolique induit par les éléments appartenant uniquement à cette voie). Ensuite, les éléments n'appartenant à aucune voie métabolique sont regroupés ensemble. Dans la Figure 4.8 (en haut à droite), les voies p_1 et p_3 ont été choisies pour être dans l'ensemble de voies indépendantes, et des groupes ont été créés pour les sous-graphes propres de p_2 et p_4 mais aussi pour les sommets restants.
2. **Calcul des composantes connexes** : Chaque groupe est décomposé suivant ses composantes connnexes. Dans la Figure 4.8 (en bas à gauche), un sous-graphe propre de p_2 (entouré en rouge) a été décomposé en deux composantes connexes.
3. **Détection de structures topologiques** : Chaque groupe est décomposé en cycles et cascades de réactions le cas échéant. Nous recherchons dans un premier tempe le plus long cycle de manière itérative jusqu'à ne plus en trouver. Il est en effet courant qu'un petit cycle de réactions soit contenu dans un plus grand d'où notre stratégie de recherche de cycles. Dans la Figure 4.8 (en bas à droite), deux cycles ont été détectés.

Une des particularités importantes de cette phase de partitionnement est qu'elle laisse la possibilité à l'utilisateur de piloter le calcul des voies métaboliques indépendantes. En choisissant a priori des voies d'intérêt pour être dans l'ensemble des voies indépendantes, l'utilisateur peut forcer la *Contrainte de proximité* à être respectée pour ces voies particulières.

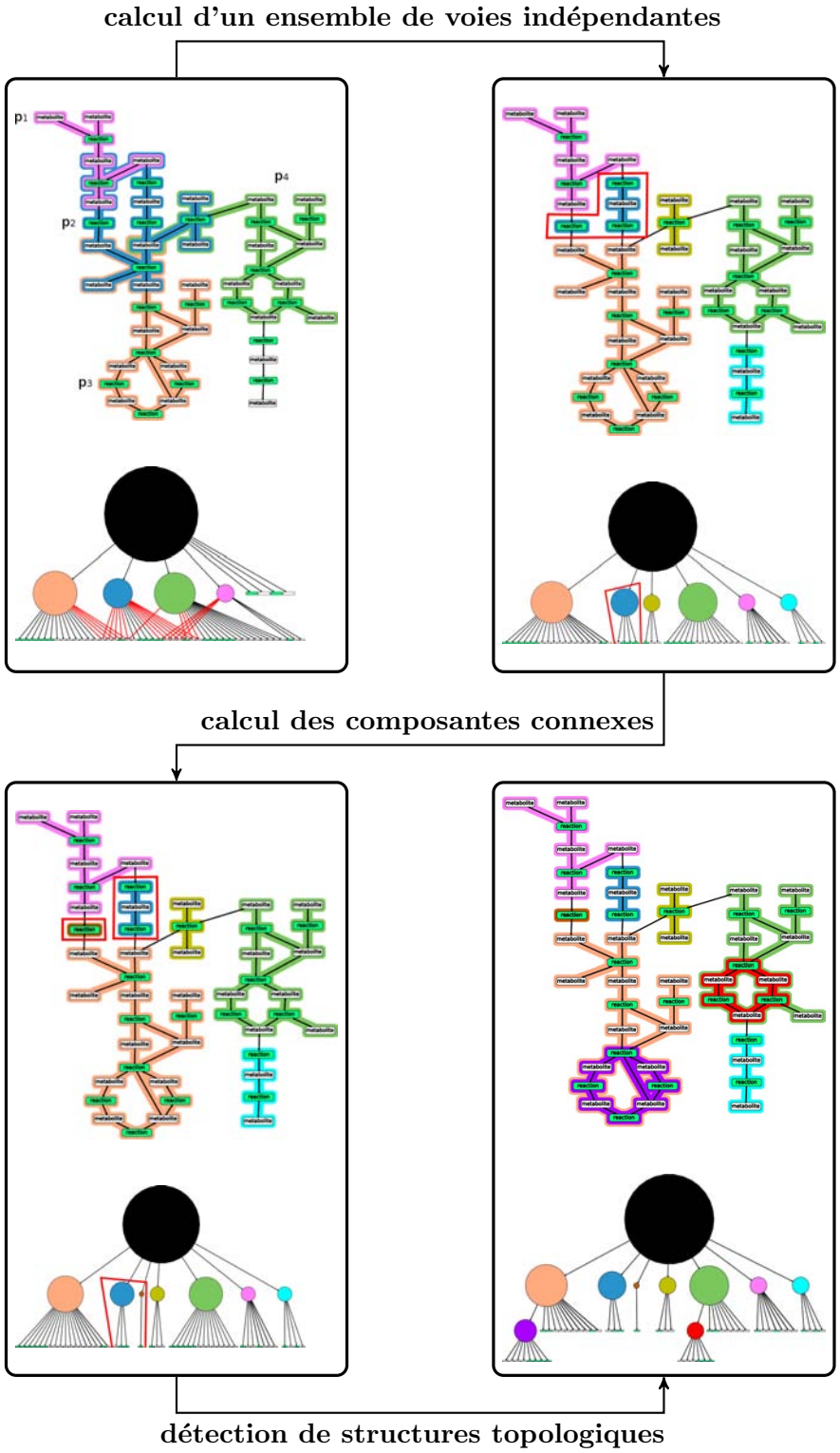


FIGURE 4.8: Illustration du processus de partitionnement. Dans un premier temps, un ensemble de voies métaboliques indépendantes est calculé et les métabolites/réactions n'appartenant à aucunes voies sont regroupés ensemble. Ensuite les composantes connexes de chaque groupe résultant sont calculées (entourées en rouge dans l'illustration). Enfin, des structures topologiques sont détectées au sein de chaque groupe.

Avant de pouvoir passer à l'étape de dessin proprement dite, nous avons besoin dans un premier temps de construire un graphe quotient associé à la décomposition hiérarchique du réseau précédemment calculée. A partir de la partition des sommets du réseau, le graphe quotient est construit de la façon suivante. Pour chaque sous-ensemble de la partition, un méta-sommet est créé dans le graphe quotient et deux méta-sommets sont reliés par une méta-arête si et seulement si au moins une arête existe entre les deux sous-ensembles de sommets associés.

Dans notre approche, le graphe quotient correspondant au plus haut niveau de l'arbre de hiérarchie est construit en premier. Le procédé est ensuite répété itérativement pour les niveaux plus bas, jusqu'à ce que tous les niveaux aient été abstraits, résultant en un graphe quotient multi-niveaux. La Figure 4.9 montre le graphe quotient multi-niveaux associé au graphe de la Figure 4.8 et à la décomposition hiérarchique calculée.

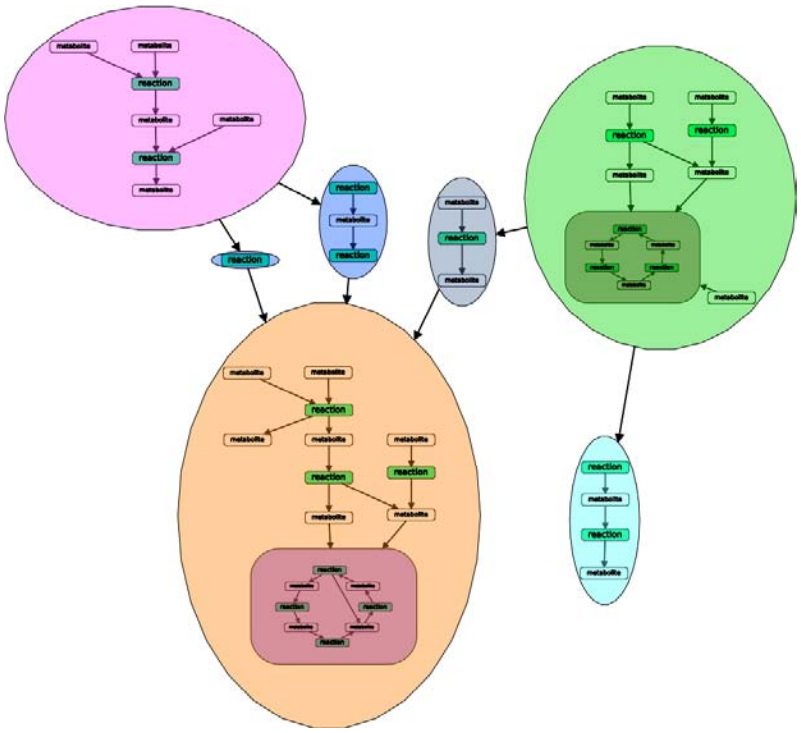


FIGURE 4.9: Graphe quotient multi-niveaux associé au graphe de la Figure 4.8 et à l'arbre de hiérarchie calculé.

4.3.2 Dessin des niveaux les plus bas de la hiérarchie

A l'issue de la phase de partitionnement, les groupes les plus bas dans la décomposition hiérarchique résultante sont soit des (sous-ensembles de) voies métaboliques, soit des structures topologiques particulières. Afin de dessiner ces derniers, notre méthode utilise

deux algorithmes de dessin. Premièrement, les cycles de réactions sont dessinés en utilisant un algorithme de dessin circulaire. Deuxièmement, les groupes restants sont dessinés via l'utilisation d'un algorithme de dessin hiérarchique (e.g. [9, 175]). Un tel algorithme est particulièrement adapté car il permet de mettre en exergue les cascades de réactions ainsi que l'organisation hiérarchique des voies métaboliques. Dans l'étape suivante, notre processus de dessin positionnera chacun de ces groupes dans le plan.

4.3.3 Dessin du graphe quotient de plus haut niveau

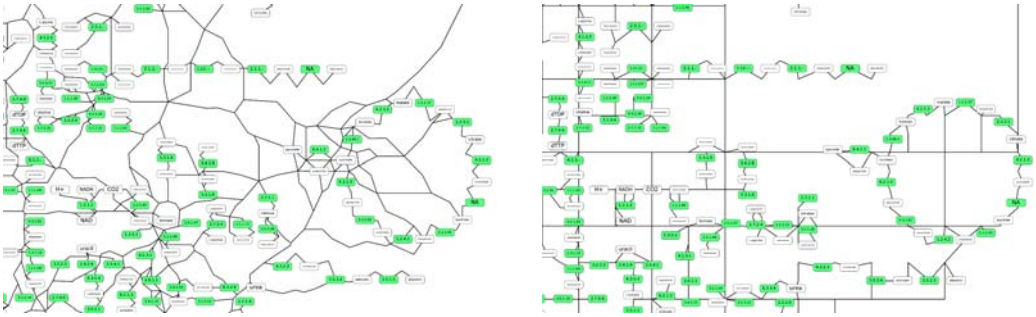
Avant de dessiner le graphe quotient de plus haut niveau, la taille des méta-sommets est fixée en fonction de celle de l'espace nécessaire pour dessiner les groupes sous-jacents. Ce graphe quotient est ensuite dessiné en utilisant un algorithme de dessin par modèle de forces adapté de GEM [77].

Une autre particularité de notre algorithme de dessin par modèle de forces est qu'il effectue des rotations des méta-sommets afin d'obtenir un dessin le plus compact possible. Une approche possible serait de combiner un algorithme de dessin par modèle de forces avec un algorithme de *moment de force* (en anglais *torque*, voir [7, 176]). Cependant, notre algorithme a seulement besoin de vérifier quatre configurations car pour respecter les conventions de dessin biologique les groupes doivent être dessinés verticalement (de haut en bas ou de bas en haut) ou horizontalement (de gauche à droite et de droite à gauche). Ainsi à chaque itération de l'algorithme de dessin par modèle de force, chacune des configurations est testée pour chaque groupe et celle qui minimise la longueur d'arête moyenne est retenue. Notre algorithme dessinent donc les groupes verticalement ou horizontalement. Les dessins en résultant sont ainsi similaires aux représentations manuelles. Il permet également d'augmenter la densité de l'information en fournissant des dessins plus compacts.

4.3.4 Réduction de l'occlusion : regroupement des arêtes entre les groupes

Au cours de l'étape précédente, les groupes de plus bas niveau ainsi que le graphe quotient de plus haut niveau ont été dessinés. Afin de fournir une visualisation classique de type *nœud-lien*, l'étape suivante consiste à ouvrir les méta-sommets. Cependant, notre algorithme par modèle de forces calcule un dessin en ligne droite des arêtes et ainsi les arêtes inter-groupes, i.e. les arêtes abstraites par des méta-arêtes dans le graphe quotient, sont représentées comme de simples segments. Ce type de représentation des arêtes peut déboucher sur des visualisations contenant beaucoup d'occlusions visuelles et ne respectant pas la *Contrainte de dessins d'arêtes* définie dans la section 4.2.

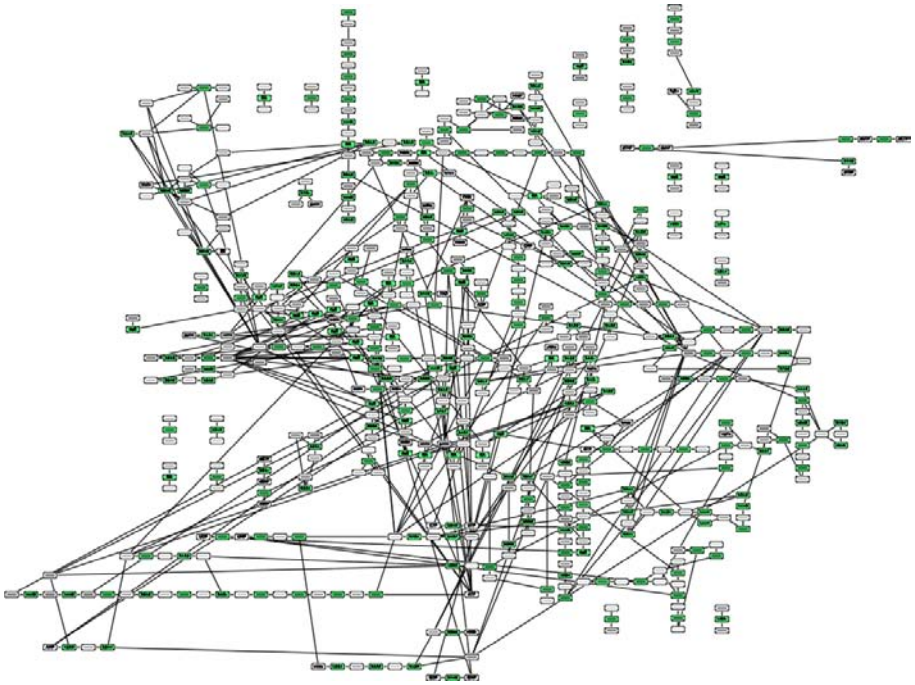
Afin de résoudre simultanément les problèmes d'occlusions et d'obtenir un dessin "pseudo-orthogonal" des arêtes, notre technique va regrouper les arêtes inter-groupes en faisceaux. Cette opération est réalisée à l'aide d'une version dédiée de notre algorithme de regroupement d'arêtes détaillé dans la section 3.3.1 et publié dans [123]. Pour rappel, la méthode discrétise le plan à l'aide d'une grille multi-résolutions et route ensuite les arêtes sur cette grille. Afin d'éviter d'obtenir des routes très sinueuses et de respecter la *Contrainte de dessin d'arêtes*, nous avons modifié l'étape de discrétisation de façon à utiliser uniquement un *quadtree* pour générer la grille. La Figure 4.10 explicite la différence dans le dessin des arêtes résultant. On peut aisément observer que l'utilisation d'un *quadtree* pour générer la grille de routage permet de réduire le nombre de points de contrôle par arêtes mais également de respecter la *Contrainte de dessin d'arêtes*.



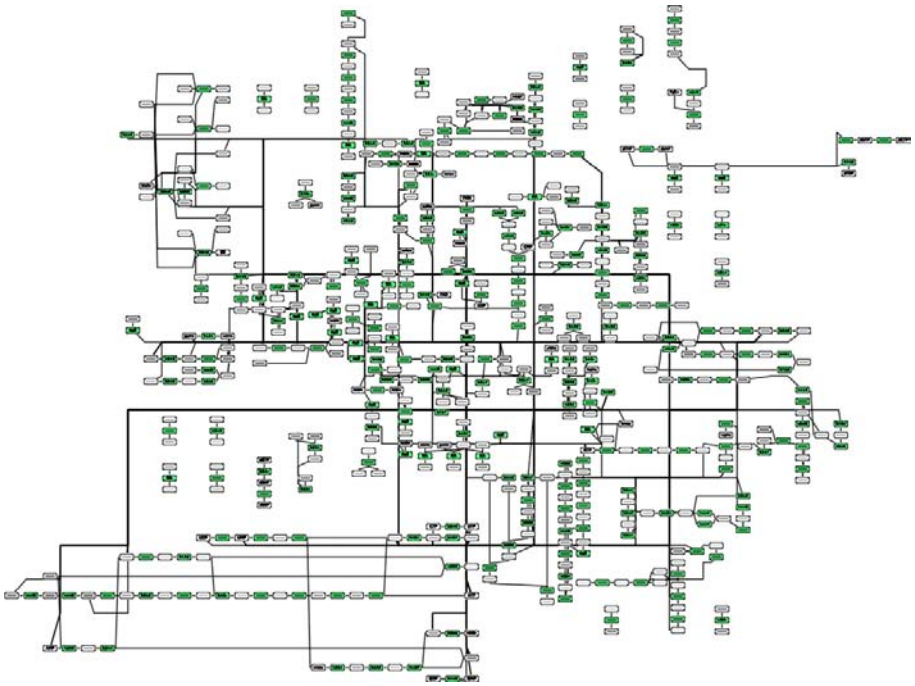
(a) (b)

FIGURE 4.10: Vue détaillée sur le réseau métabolique de l'organisme *Saccharomyces cerevisiae* (levure). (a) Les arêtes (inter-groupes et intra-groupe sur ces exemples) ont été regroupées en faisceaux via l'algorithme original détaillé dans la section 3.3.1. (b) Résultat du regroupement d'arêtes en utilisant uniquement un *quadtree* pour générer la grille de routage.

La Figure 4.11 illustre la nécessité de regrouper les arêtes inter-groupes en faisceaux. Elle présente deux visualisations d'un réseau métabolique simplifié de l'organisme *Buchnera aphidicola* [45] (protéobactérie) : une sans regroupements d'arêtes et une avec regroupements d'arêtes. Dans le dessin sans regroupement, on peut observer que le niveau d'occlusion est assez important, introduisant du bruit dans le milieu de la représentation. Dans le dessin avec regroupements, l'occlusion due aux arêtes a été considérablement réduite et la visualisation est beaucoup plus esthétique.



(a)



(b)

FIGURE 4.11: Visualisations d'un réseau métabolique simplifié de l'organisme *Buchnera aphidicola* [45] (protéobactérie) dessiné avec notre méthode. (a) Dessin du réseau sans regroupements d'arêtes inter-groupes. (b) Dessin du réseau après avoir regroupé les arêtes inter-groupes en faisceaux en utilisant une version dédiée de notre algorithme [123].

4.3.5 Exemple de résultat : visualisation du réseau métabolique de la levure

La Figure 4.12 montre le dessin résultant de l'application de notre méthode sur le réseau métabolique de l'organisme *Saccharomyces cerevisiae* (levure) fourni par la base de données BioCyc [112]. Ce réseau contient 836 sommets (422 métabolites, 414 réactions) et 936 arêtes réparties en 164 voies métaboliques.

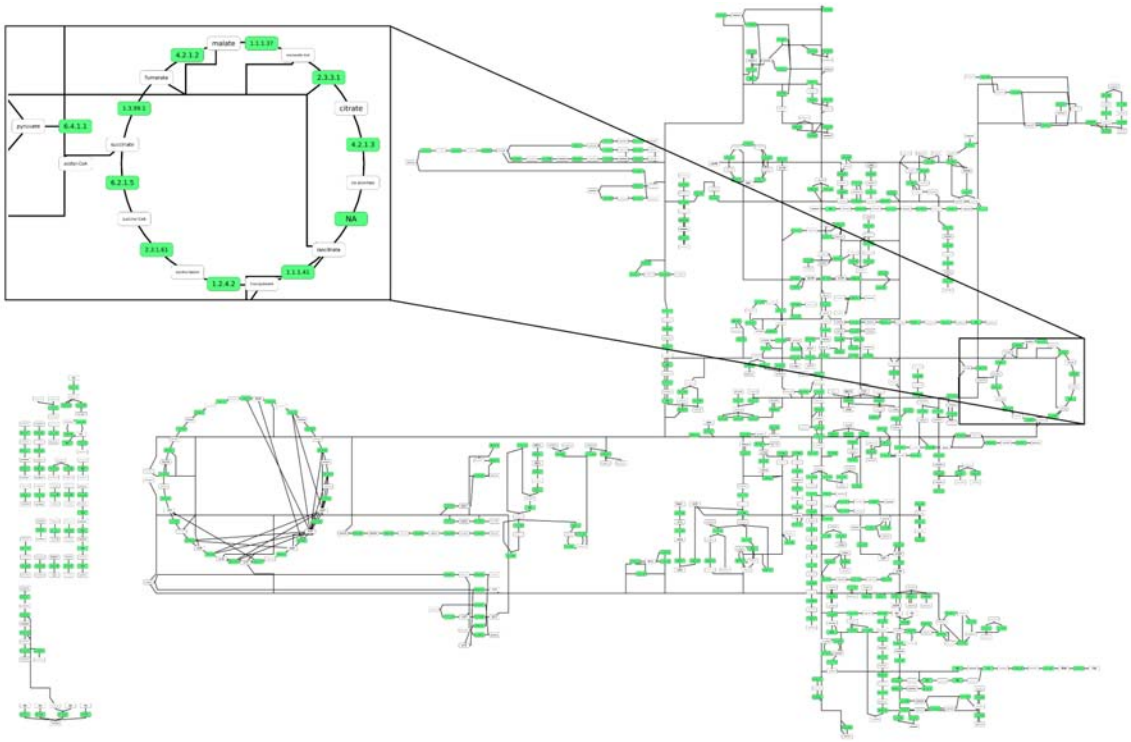


FIGURE 4.12: Le réseau métabolique de l'organisme *Saccharomyces cerevisiae* (levure) dessiné avec notre méthode. La voie métabolique du cycle TCA (responsable de la respiration aérobie) a été insérée a priori dans l'ensemble des voies indépendantes avant la phase de partitionnement. Dans la vue détaillée, on peut observer que le cycle de réactions contenu dans la voie du cycle TCA a été correctement détecté et représenté.

On peut remarquer que les contraintes que notre algorithme essaie de respecter ont bien été prises en compte :

- aucune duplication de sommets n'a été effectuée,
- la représentation ne contient pas d'occlusions visuelles
- les arêtes sont dessinées de manière "pseudo-orthogonales"
- des structures topologiques ont été détectées et correctement représentées.

Néanmoins, l'ensemble de voies métaboliques indépendantes calculé durant la phase de partitionnement contient 33 voies métaboliques et les autres ont été réparties sur l'ensemble de la zone de dessin. Comme mentionné dans la sous-section 4.2, ceci est principalement dû au respect de la *Contrainte de duplication*. Notre méthode peut résoudre ce problème en permettant à l'utilisateur de renseigner un ensemble de voies métaboliques pour lesquelles la *Contrainte de proximité* sera strictement respectée. Par exemple, si il est intéressé par la *respiration aérobie*, l'utilisateur peut décider d'insérer la voie métabolique du cycle des acides tricarboxyliques (aussi appelé cycle TCA ou cycle de Krebs ou cycle de l'acide citrique) dans l'ensemble de voies indépendantes comme cela a été fait pour générer le dessin de la Figure 4.12. On peut d'ailleurs remarquer dans la vue détaillée que la *Contrainte de proximité* et la *Contrainte de dessin biologique* ont été respectées pour la voie métabolique du cycle TCA car cette voie est dessinée de façon compacte et le plus long cycle de réactions contenue dans cette voie a été détecté et correctement représenté.

4.3.6 Relaxation de la contrainte de duplication

Précédemment, nous avons décrit comment notre technique peut dessiner un réseau métabolique complet tout en évitant de dupliquer des sommets. Cependant, en considérant des réseaux denses et/ou complexes, la phase de partitionnement peut calculer un très petit ensemble de voies métaboliques indépendantes menant à des représentations où de nombreuses voies ont été réparties sur l'ensemble de la zone de dessin. De plus, la visualisation résultante peut également souffrir de problèmes d'occlusion et de lisibilité. Pour résoudre ce problème, nous proposons de relaxer la *Contrainte de duplication* suivant deux niveaux :

1. **Niveau voie métabolique** : Quand un métabolite est impliqué dans de nombreuses réactions, alors ne pas dupliquer ce dernier peut créer un grand nombre de croisements d'arêtes inutiles. Pour résoudre ce problème, nous donnons la possibilité à l'utilisateur de dupliquer les métabolites en fonction du nombre de réactions où ils sont impliqués.
2. **Niveau réseau** : Quand quelques voies métaboliques partagent des métabolites et/ou des réactions avec beaucoup d'autres, alors le calcul de l'ensemble des voies indépendantes peut produire de mauvais résultats. Dans ce cas, nous donnons également la possibilité à l'utilisateur de dupliquer des métabolites et des réactions en fonction du nombre de voies métaboliques les contenant.

Par exemple, considérons le réseau métabolique de la levure décrit dans la section 4.3.5. Lors de la phase de partitionnement, notre algorithme calcule un ensemble de 33 voies indépendantes, i.e. 20% de l'ensemble de toutes les voies. En dupliquant toutes les réactions

et métabolites appartenant à plus de 3 voies différentes, alors notre algorithme calcule un ensemble de 86 voies indépendantes, i.e. 52% de l'ensemble de toutes les voies. Notre technique de dessin va ainsi garantir que la représentation résultante respectera strictement la *Contrainte de proximité* pour 86 voies métaboliques (au lieu de 33). La représentation du réseau après relaxation de la *Contrainte de duplication* est plus lisible et contient moins d'occlusions que celle calculée sans aucune duplication. Cette approche peut être vue comme un compromis entre la *Contrainte de duplication* et la *Contrainte de proximité*.

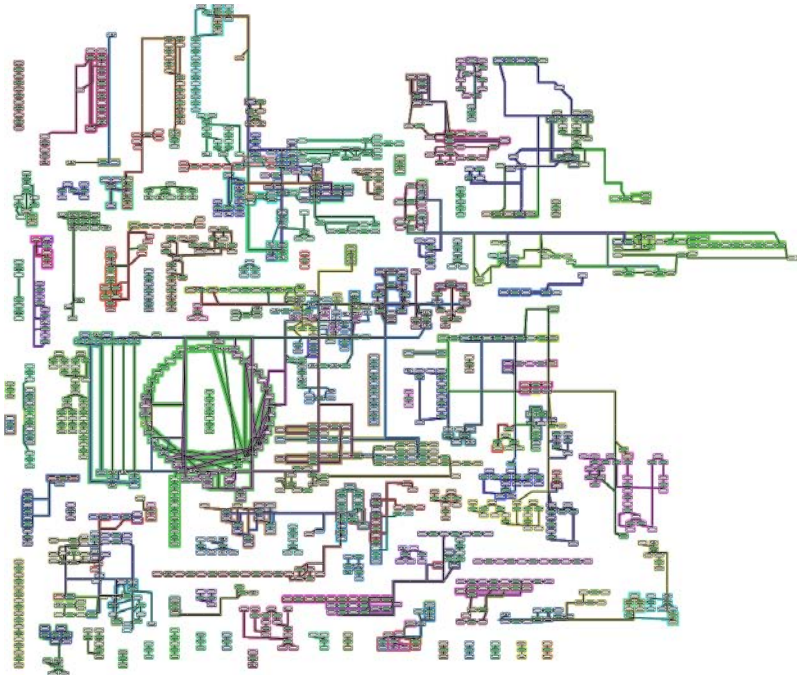
La Figure 4.13 montre deux visualisations du réseau métabolique de la levure après duplication de sommets. Dans la Figure 4.13(a), les sommets appartenant à plus de 3 voies métaboliques ont été dupliqués. Dans la Figure 4.13(b), les sommets appartenant à plus de 1 voie métabolique ont été dupliqués. Dans cette Figure, les voies métaboliques ont été mis en exergue par des enveloppes concaves (voir section 7.2).

4.3.7 Complexité et temps de calcul

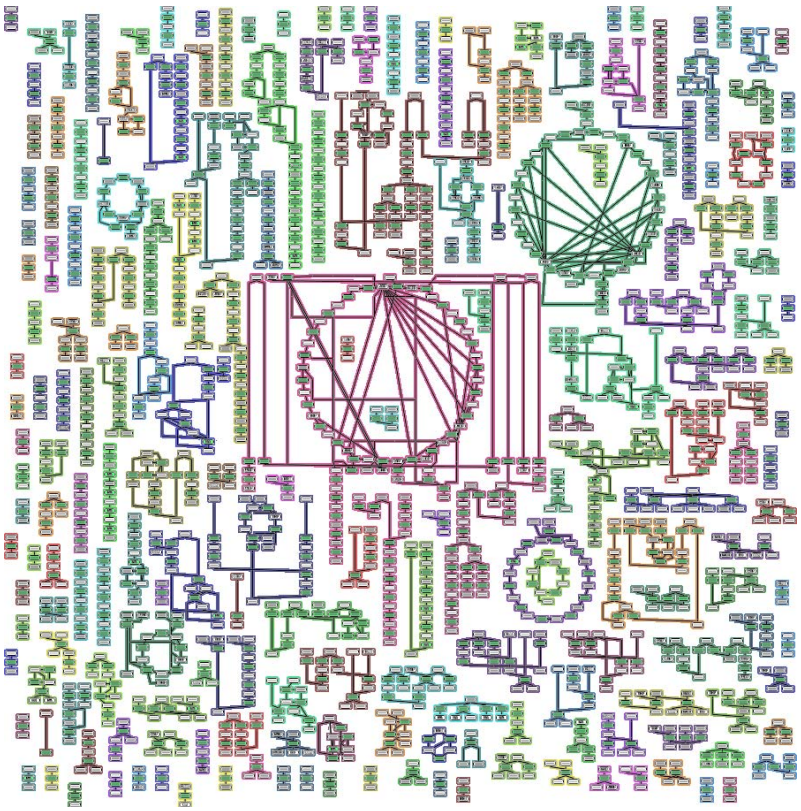
En terme de complexité en temps, le goulot d'étranglement de notre méthode se trouve dans la phase de partitionnement en raison de la détection de plus long cycle, problème connu pour être NP-Complet [113]. Nous utilisons une méthode exacte, via un parcours en profondeur (DFS) bornée par le temps, pour déterminer le plus long cycle. Ainsi si le réseau en entrée contient seulement une voie métabolique, l'algorithme de détection de plus long cycle est appliqué sur le réseau entier menant à une complexité en temps exponentielle .

Néanmoins ce pire des cas n'arrive jamais dans des scénarios réels et notre méthode peut dessiner un réseau entier en quelques secondes. La Figure 4.14 montre les boîtes à moustaches des temps de calcul en secondes de notre technique de dessin pour les 113 organismes de la base de données MetExplore [45]. Cette base de données contient des réseaux métaboliques simples comme celui de *Candidatus Hodgkinia cicadicola* (144 sommets, 133 arêtes et 21 voies métaboliques) ou complexes comme celui de l'*Homo sapiens* (1932 sommets, 2140 arêtes et 272 voies métaboliques).

Les temps de calcul de la méthode complète varient de 0.5 secondes (pour *Candidatus Hodgkinia cicadicola*) à 66.7 secondes (pour *Mycobacterium smegmatis*) et le temps de calcul moyen (resp. médian) est de 11.3 secondes (resp. 6.5 secondes). On peut également remarquer dans la Figure 4.14 que le temps de calcul de la phase de partitionnement est très faible (le temps de calcul moyen est de 0.33 secondes et la médiane vaut 0.27 secondes) par rapport au temps de calcul global. Bien que la détection du plus long cycle soit le goulot d'étranglement en terme de complexité en temps, il semble que la phase de



(a)



(b)

FIGURE 4.13: Représentations du réseau métabolique de l'organisme *Saccharomyces cerevisiae* (levure). (a) après duplication de sommets appartenant à plus de 3 voies métaboliques. (b) après duplication des sommets appartenant à plus de 1 voie métabolique. Les voies métaboliques sont mises en exergue en utilisant des enveloppes concaves (voir section 7.2).

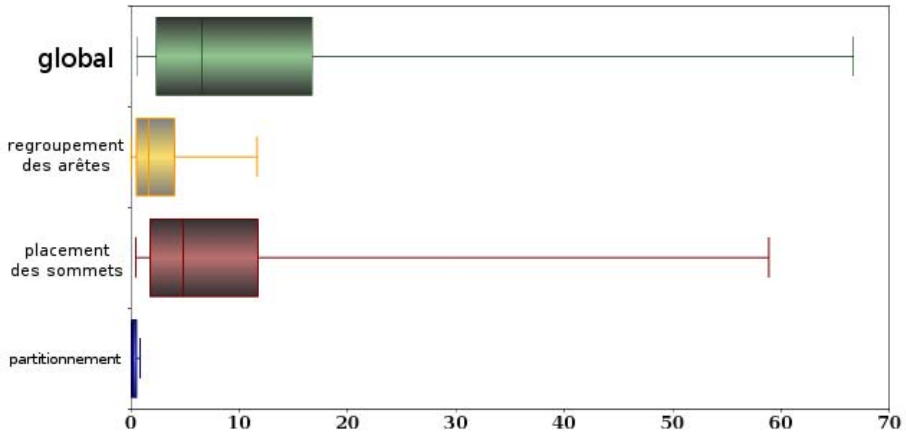


FIGURE 4.14: Boîtes à moustaches des temps de calcul en secondes de notre méthode de dessin d'un réseau métabolique entier pour les 113 organismes de la base de données MetExplore [45]

placement de sommets soit la plus longue en terme de temps de calcul (le temps de calcul moyen est de 8.7 secondes et la médiane vaut 4.8 secondes).

4.4 Mise en évidence de voies métaboliques dans la représentation

La représentation d'un réseau métabolique produite avec notre méthode préserve la topologie du réseau. Cependant, la phase de regroupement d'arêtes rend difficile la tâche d'identifier chaque voie métabolique même si certaines de ces dernières sont dessinées dans une région compacte du dessin. Il peut ainsi être difficile de suivre une arête de sa cible à sa source vu que les arêtes regroupées sont dessinées l'une sur l'autre. De plus les nombreux croisements entre faisceaux d'arêtes induisent des ambiguïtés sur les routes suivies par chaque arête.

Pour résoudre ce problème, nous proposons deux techniques d'interaction de type Focus+Contexte (voir section 1.2.2.2) permettant de mettre en exergue des voies métaboliques d'intérêt dans la représentation. La première technique aide à identifier la position des voies concernées dans le dessin en les entourant par des enveloppes concaves (voir définition et détails de la technique pour les générer à la section 7.2) . La seconde vise à se concentrer sur une voie en particulier et obtenir une information précise de sa structure topologique en appliquant une déformation 3d sur les primitives géométriques représentant ses éléments. Cette déformation agit à un échelle locale et fait en sorte que le dessin de la voie soit plaqué sur une surface cylindrique. Cette technique d'interaction sera plus amplement détaillée dans la section 7.3 de ce manuscrit.

Un exemple d'application est la mise en évidence des voies métaboliques contenant un métabolite particulier ou une réaction particulière. Supposons par exemple que l'on sélectionne l'élément *malate* dans la représentation du réseau métabolique de l'organisme *Saccharomyces cerevisiae*. Cet élément est contenu dans 3 voies métaboliques associées à ce réseau. La Figure 4.15 montre une illustration de la technique d'interaction entourant les voies d'intérêt avec des enveloppes concaves. On peut observer que cette technique permet d'identifier facilement les éléments de chaque voie ainsi que la position de ces dernières dans le dessin. Les éléments en commun entre voies peuvent également être déterminés en regardant les intersections entre les enveloppes. Quelques ambiguïtés persistent tout de même. Il est par exemple difficile d'avoir une idée précise de la topologie de chaque voie, principalement dû au fait que les arêtes sont regroupées. La Figure 4.16 illustre la technique d'interaction permettant de mettre clairement en évidence une voie et sa topologie en utilisant une déformation 3d. L'utilisation de cette technique permet de lever clairement les ambiguïtés qui persistaient avec la technique de mise en évidence par enveloppes concaves.

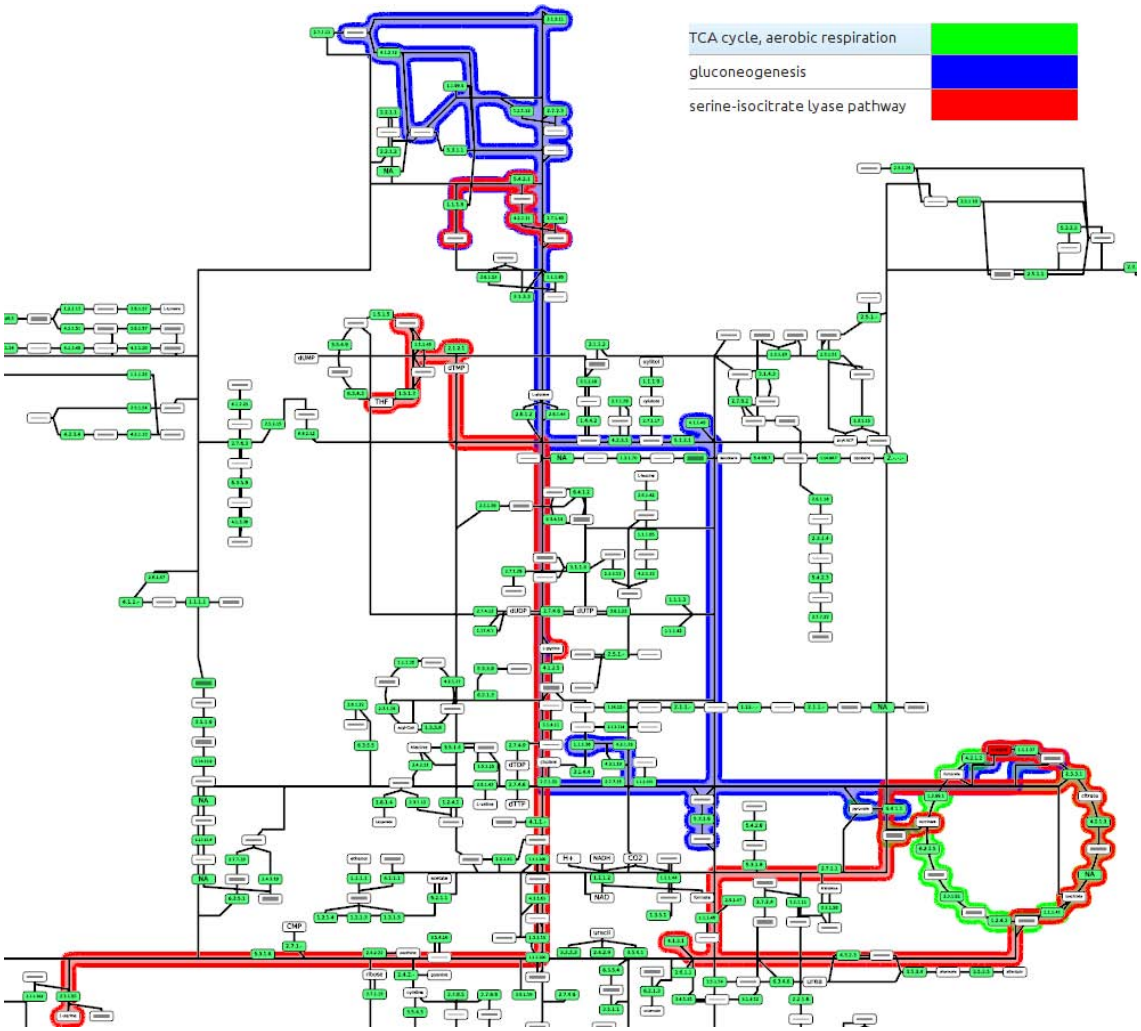
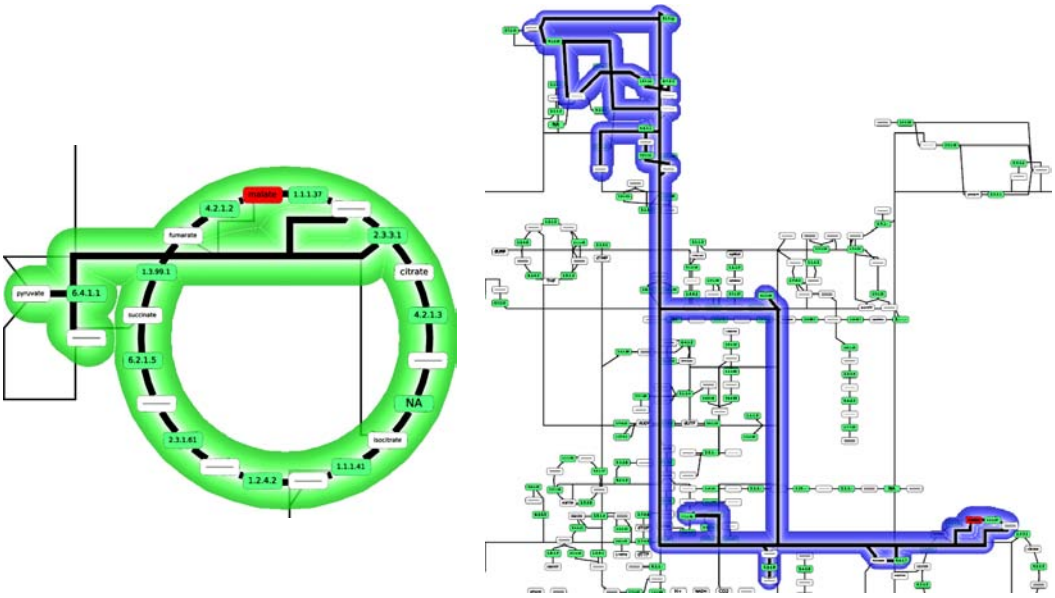
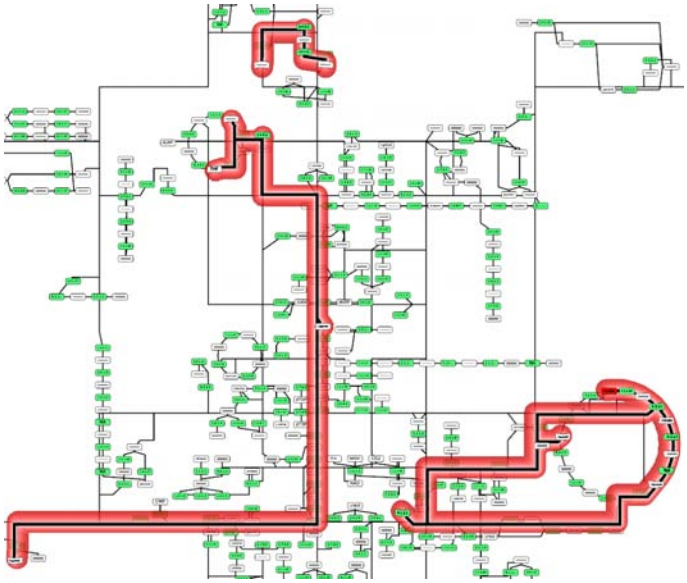


FIGURE 4.15: Illustration de la mise en évidence de voies métaboliques contenant un élément particulier du réseau. Ici, le métabolite *malate* a été sélectionné (en rouge dans le cycle à droite) dans le dessin du réseau métabolique de l'organisme *Saccharomyces cerevisiae*. Cet élément est contenu dans 3 voies métaboliques dont les dessins ont été entourés à l'aide d'enveloppes concaves : cycle TCA (en vert), gluconeogenesis (en bleu), serine-isocitrate lyase (en rouge).



(a) (b)



(c)

FIGURE 4.16: Illustration de la mise en exergue de voies métaboliques particulières à l'aide d'une déformation 3d. Les voies présentées ici sont les même que celles de la Figure 4.15, soit celles contenant l'élément *malate* (coloré en rouge). (a) cycle TCA (b) gluconeogenesis (c) serine-isocitrate lyase

Deuxième partie

Infographie pour la visualisation interactive de graphes

Cette partie vise à présenter les différents travaux en infographie (en anglais, *Computer Graphics*) pour la visualisation de graphes réalisés au cours de cette thèse. La majorité de ces travaux s'attachent à exploiter la puissance de calcul offerte par les processeurs graphiques contemporains.

Dans le chapitre 5, nous présentons une technique pour optimiser le temps de rendu de courbes paramétriques (e.g. courbes de Bézier). Développée originellement pour accélérer le temps de rendu des visualisations de graphe où les arêtes sont représentées par des courbes, la méthode est générique et peut être utilisée dans d'autres contextes que la Visualisation d'Information.

Le chapitre 6 introduit un exemple de technique de rendu exploitant pleinement le processeur graphique pour la Visualisation d'Information. Cette technique permet d'extraire l'information de densité des faisceaux d'arêtes dans une visualisation de graphe où un algorithme de regroupement d'arêtes a été appliqué.

Enfin, le chapitre 7 présente différentes techniques pour visualiser un ou plusieurs sous-graphes d'intérêt dans le contexte global d'une visualisation de graphe.

Chapitre 5

Utilisation de courbes paramétriques pour lisser la forme des arêtes

Dans ce chapitre, nous présentons les travaux autour du dessin des arêtes dans un contexte de visualisation de graphe avec regroupement d'arêtes. Après avoir appliqué un algorithme de regroupement d'arêtes comme celui présenté dans la section 3.3.1, une arête est représentée par une ligne brisée en raison de l'étape de routage lui associant un ensemble de points de contrôle. En fonction de la taille du graphe et de la grille de routage utilisée, ce nombre de points de contrôle peut être assez important (de l'ordre de plusieurs dizaines voire centaines). Dans le dessin de graphe résultant, ce grand nombre de points de contrôle induit un effet "zig zag" sur les arêtes les rendant difficile à suivre d'une extrémité à l'autre. La qualité esthétique de la visualisation pâtit également de cet effet. Afin de lisser la forme des arêtes, nous avons choisi de les représenter comme des courbes paramétriques. Nous avons investigué trois types de courbes (voir Figure 5.1) :

- les courbes de Bézier
- les courbes B-Splines
- les courbes de Catmull-Rom

L'utilisation de courbes paramétriques produit une visualisation de graphe bien plus esthétique par rapport à l'utilisation de lignes brisées. De plus, les arêtes partageant des points de contrôle successifs restent groupées sur ces portions donnant une plus belle impression de flux entre différentes régions du dessin. Un des désavantages de l'utilisation de courbes est que quelques chevauchements entre sommets et arêtes peuvent réapparaître dans le dessin (principalement avec les courbes de Bézier qui manquent de contrôle local). Mais le principal problème vient surtout du fait que le coût de calcul des points interpolant la courbe le long de son tracé est assez important et augmente en fonction du nombre de points de contrôle. Pour palier à cette situation, nous avons développé une implémentation

sur processeur graphique pour rendre efficacement un très grand nombre de courbes paramétriques définies par un nombre arbitraire de points de contrôle. L'autre intérêt majeur de cette implémentation est qu'elle donne la possibilité de mettre en place des interactions fluides avec le dessin de graphe. Les prémisses de ces travaux ont été publiés dans les actes de la *14ème conférence internationale sur la Visualisation d'Information (IV'10)* [121].

5.1 Introduction aux courbes paramétriques

Une courbe paramétrique est définie par des points de contrôle et est analytiquement décrite par un polynôme $P(t)$. Pour calculer les points de la courbe, ce polynôme doit être évalué pour t prenant ses valeurs dans l'intervalle $[0, 1]$. Pour dessiner la courbe, on évalue le polynôme pour un certain échantillonnage de t dans $[0, 1]$ et on trace une ligne entre chaque $P(t)$ successifs. La définition et le degré du polynôme $P(t)$ varient suivant le type de la courbe. Pour de plus amples informations sur les différents types de courbes paramétriques, nous recommandons la lecture du cours du professeur Shene [166].

5.1.1 Courbes de Bézier

Les courbes de Bézier sont des courbes polynomiales paramétriques décrites pour la première fois en 1962 par l'ingénieur français Pierre Bézier. Soit $(P_0, P_1, \ldots, P_n)$ les points de contrôle d'une courbe de Bézier. Le degré du polynôme définissant la courbe est de $n - 1$. Ce polynôme défini par les points de contrôle est le suivant :

$$P(t) = \sum_{i=0}^{n} B_{i,n}(t) P_i \tag{1}$$

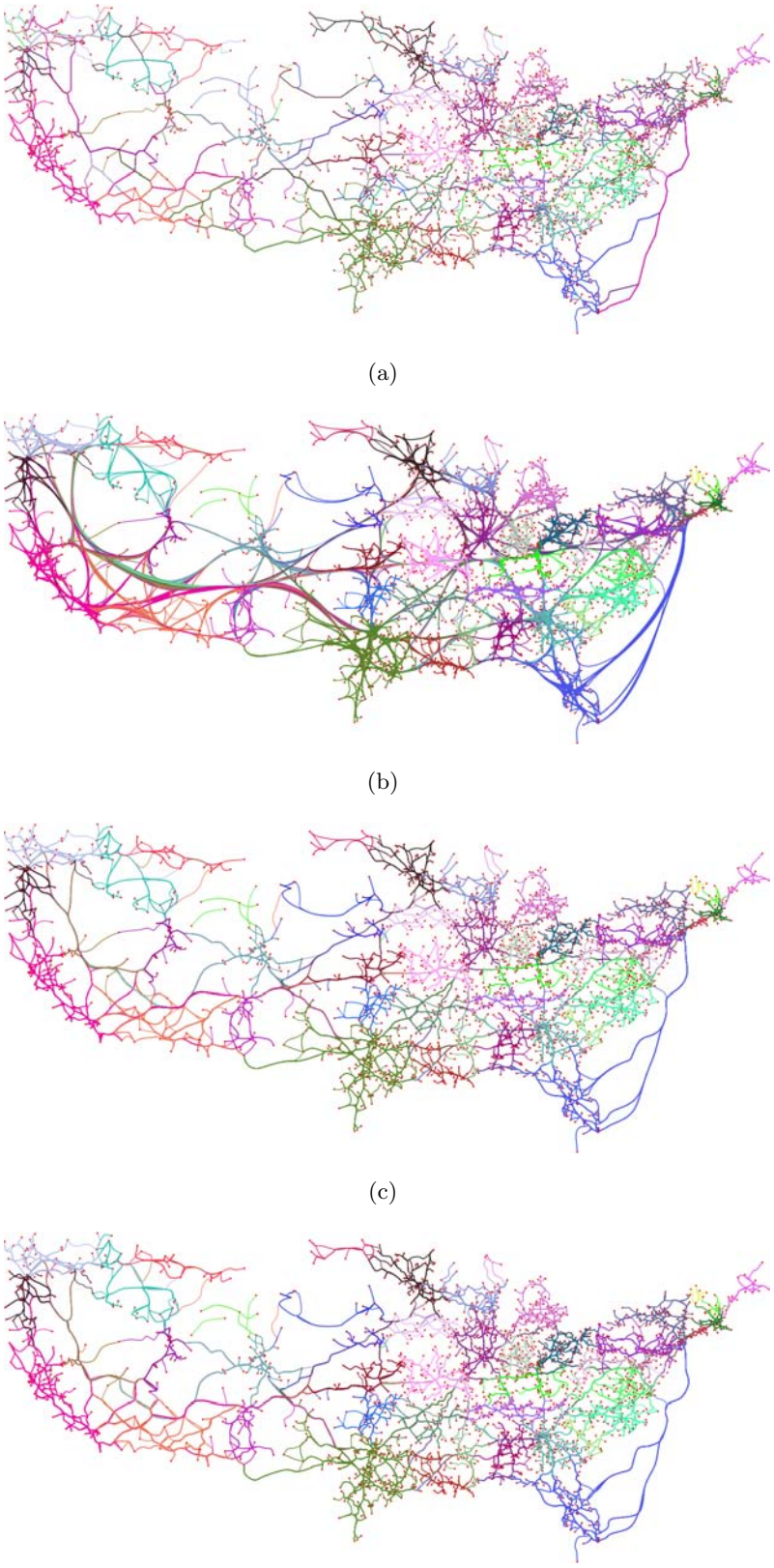
où :

$$B_{i,n}(t) = \binom{n}{i} (1-t)^{n-i} t^i, \ 0 \leq t \leq 1 \tag{2}$$

correspondent aux polynômes de *Bernstein* et $\binom{n}{i} = \frac{n!}{i!(n-i)!}$ dénotent les coefficients binomiaux usuels. Un exemple de courbe est donné à la Figure 5.2.

Une courbe de Bézier possède les propriétés suivantes :

- La courbe est à l'intérieur de l'enveloppe convexe définie par ses points de contrôle.
- La courbe commence au point P_0 et se termine au point P_n, mais ne passe pas à priori par les autres points qui déterminent cependant son allure générale.
- $\overrightarrow{P_O P_1}$ est le vecteur tangent à la courbe en P_0 et $\overrightarrow{P_{n-1} P_n}$ au point P_n.



(a)

(b)

(c)

(d)

FIGURE 5.1: Différents dessins d'un graphe avec regroupements d'arêtes en fonction du type de représentation des arêtes. (a) poli-lignes (b) courbes de Bézier (c) courbes B-Spline (d) courbes de Catmull-Rom

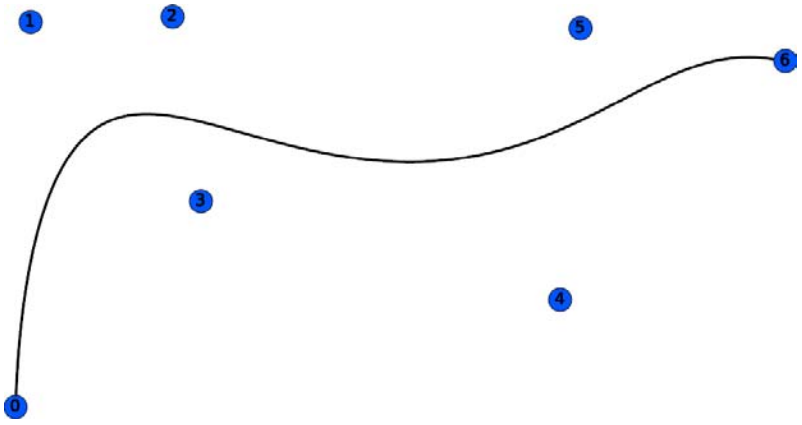


FIGURE 5.2: Exemple de courbe de Bézier de degré 6 définie par 7 points de contrôle.

- La courbe est infiniment dérivable (de classe C^∞).

Ce type de courbes est particulièrement coûteux à calculer, surtout quand le nombre de points de contrôle est élevé. Pour les courbes de degré faible, l'évaluation de $P(t)$ peut être optimisée par la technique des *forward difference* [15] où les multiplications sont confinées dans une phase d'initialisation et le calcul des points de la courbe repose uniquement sur des additions. Bien que cette approche est plus efficace en termes de temps de calcul, elle est plus sensible aux erreurs de précision et n'est pas adéquate pour les courbes définies par plus de quatre points de contrôle. L'algorithme de Casteljau [48] est une autre alternative pour évaluer les points d'une courbe de Bézier, reposant sur une approche récursive pour évaluer les polynômes en forme de *Bernstein*. Cet algorithme est plus lent à s'exécuter (complexité quadratique) que l'évaluation polynomiale classique mais il est plus stable numériquement.

5.1.2 Courbes B-Spline

Les courbes B-Spline sont une généralisation des courbes de Bézier. Soit $(P_0, P_1, \ldots, P_n)$ les points de contrôle d'une courbe B-Spline et une séquence de nœuds non décroissante $T = \{t_0, t_1, \ldots, t_m\}$ avec $0 \leq t_i \leq 1$. Une courbe B-Spline de degré p définie par ces points de contrôle et le vecteur de nœuds T est analytiquement décrite par le polynôme :

$$P(t) = \sum_{i=0}^{n} N_{i,p}(t) P_i \tag{3}$$

où les $N_{i,p}(t)$ correspondent aux fonctions de bases B-Spline de degré p. L'évaluation de ces fonctions peut être effectuée de façon stable numériquement grâce à la formule de récurrence de Cox-de Boor [47] :

$$
\begin{aligned}
N_{i,0}(t) &= \begin{cases} 1 \text{ si } t_i \leq t \leq t_{i+1} \\ 0 \text{ sinon} \end{cases} \\
N_{i,p}(t) &= \frac{t - t_i}{t_{i+p} - t_i} N_{i,p-1}(t) + \frac{t_{i+p+1} - t}{t_{i+p+1} - t_{i+1}} N_{i+1,p-1}(t)
\end{aligned} \tag{4}
$$

Un exemple de courbe B-Spline de degré 3 est donné à la Figure 5.3.

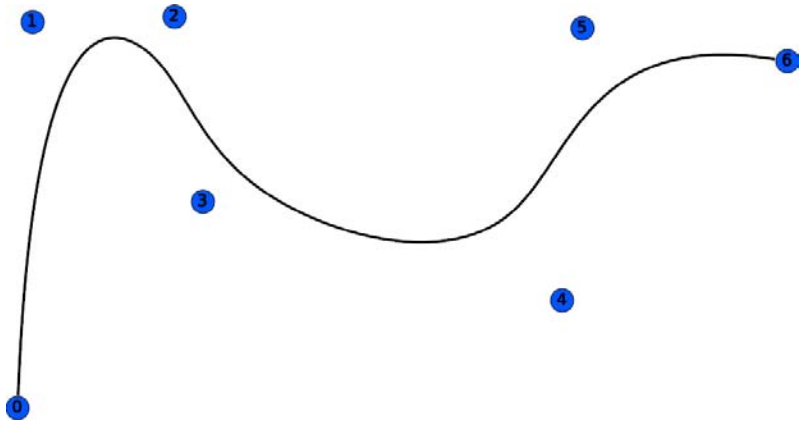


FIGURE 5.3: Exemple de courbe B-Spline de degré 3 définie par 7 points de contrôle.

Contrairement à une courbe de Bézier, une courbe B-Spline implique plus de données en entrée, soit :

- un ensemble de $n + 1$ points de contrôle
- une séquence de $m + 1$ nœuds
- un degré p

Les propriétés d'une courbe B-Spline sont les suivantes :

- n, m et p doivent satisfaire $m = n + p + 1$. Plus précisément, si on veut définir une courbe B-Spline de degré p avec $n+1$ points de contrôle, $n+p+2$ nœuds doivent être donnés. Inversement, si une séquence de $m+1$ nœuds et $n+1$ points de contrôle sont donnés, le degré de la courbe B-Spline est $p = m - n - 1$. Le degré d'une courbe B-Spline est donné en entrée, contrairement à une courbe de Bézier où le degré dépend du nombre de points de contrôle.
- Sur l'intervalle défini par deux nœuds successifs $[t_i, t_{i+1}[$, au plus $p + 1$ fonctions de bases B-Spline sont différentes de 0, à savoir : $N_{i-p,p}(t), N_{i-p+1,p}(t), \ldots, N_{i,p}(t)$.
- Le point sur la courbe correspondant à un nœud t_i, $P(t_i)$, est appelé *point nœud*. Ces points nœud divisent la courbe B-Spline en des segments de courbe, chacun d'entre eux étant défini sur l'intervalle entre deux nœuds successifs. Ces segments correspondent à des courbes de Bézier de degré p.

- Une courbe B-Spline offre plus de contrôle quant à sa forme par rapport à une courbe de Bézier. Ainsi, changer la position d'un point de contrôle ne modifie que localement l'allure générale de la courbe.
- Une courbe B-Spline est contenue dans l'enveloppe convexe de ses points de contrôle. Plus précisément, pour une valeur de t contenu dans l'intervalle de nœuds $[t_i, t_{i+1}[$, $P(t)$ est contenu dans l'enveloppe convexe des points de contrôle $P_{i-p}, P_{i-p+1}, \ldots, P_i$.
- Si la séquence de nœuds n'a pas une structure particulière, la courbe générée ne passera pas par le premier et dernier point de contrôle. Afin d'obtenir une telle propriété, le premier et dernier nœud doit être de multiplicité $p+1$. Plus précisément, les $p+1$ premiers nœuds doivent être égaux à 0 et les $p+1$ derniers à 1. Par exemple, pour la courbe de la Figure 5.3, le nombre de points de contrôle est $n+1=7$, et le degré de la courbe vaut $p=3$. Ainsi, m doit valoir 10 car $6+3+2=11$ nœuds doivent être fournis. Afin que la courbe passe par le premier et dernier point de contrôle, les 4 premiers et 4 derniers nœuds doivent être identiques. Les $11-4*2=3$ nœuds restants peuvent prendre n'importe quelles valeurs dans $[0,1]$. Dans notre cas, ils sont définis comme étant uniformément espacés. La séquence de nœuds utilisée pour générer la courbe est donc $\{0,0,0,0,0.25,0.5,0.75,1,1,1,1\}$.

5.1.3 Courbes de Catmull-Rom

Les courbes de Catmull-Rom ont été introduites dans [40] et portent le nom de leurs concepteurs : Edwin Catmull et Raphael Rom. Soient $(P_0, P_1, \ldots, P_n)$ les points de contrôle d'une telle courbe. Pour chaque P_i, un paramètre t_i dans $[0,1]$ est associé $(0 \leq t_i \leq t_{i+1} \leq 1)$. Une courbe de Catmull-Rom a les propriétés suivantes :

- La courbe interpole (passe par) ses points de contrôle, ce qui donne beaucoup de flexibilité pour définir sa forme.
- La courbe n'est pas obligatoirement contenue dans l'enveloppe convexe de ses points de contrôle.
- La courbe a un support local, ce qui signifie que chaque point de contrôle n'affecte seulement qu'une petite portion de la courbe.
- La courbe a une représentation polynomiale par morceaux. Ainsi une courbe de Catmull-Rom C^k continue est constituée de segments de Bézier de degré $2k+1$ entre chaque point de contrôle successif. Ces morceaux polynomiaux sont seulement influencés par un ensemble local de points de contrôle. Ainsi, le morceau de la courbe entre les paramètres t_i et t_{i+1} est influencé par les points de contrôle $P_{i-k}, \ldots, P_{i+1+k}$. De plus, le morceau de courbe interpole ses extrémités (i.e. à t_i (resp. t_{i+1}), la courbe s'évalue en P_{i-k} (resp. P_{i+1+k})).

Les courbes de Catmull-Rom C^1 continues sont les plus simples et les plus populaires. Elles sont composées de segments de Bézier cubiques. Un exemple d'une telle courbe est donnée à la Figure 5.4.

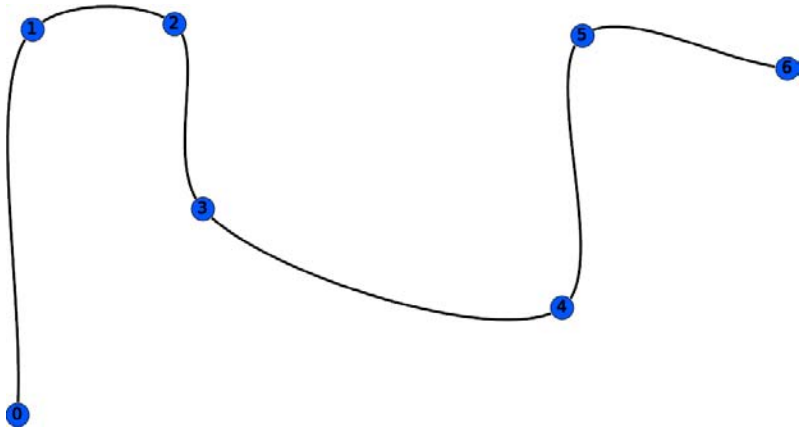


FIGURE 5.4: Exemple de courbe de Catmull-Rom de degré 3 (C^1*continue*) définie par 7 points de contrôle.

Le choix des t_i réfère à la paramétrisation d'une courbe. Ils peuvent être généralisés par la formule suivante :

$$\begin{aligned} t_0 &= 0 \\ t_{0<i\leq n} &= \frac{\sum_{j=1}^{i} |P_j - P_{j-1}|^\alpha}{\sum_{j=1}^{n} |P_j - P_{j-1}|^\alpha} \end{aligned} \tag{5}$$

Le paramètre α prend ses valeurs dans $[0, 1]$. Quand $\alpha = 0$, la paramétrisation est dite uniforme, quand $\alpha = 0.5$, la paramétrisation est dite centripétale et $\alpha = 1$ définit une paramétrisation cordale. Suivant la paramétrisation choisie, l'allure de la courbe varie. Dans [198], Yuksel *et al.* ont prouvé que seule une paramétrisation centripétale garantie que la courbe générée ne comportera pas de point de rebroussement (en anglais, *cusp*) et ne s'intersectera pas avec elle même.

Soit $0 \leq t_i \leq t < t_{i+1} \leq 1$ le paramètre pour lequel calculer un point de la courbe. Pour $0 < i < n - 1$, les points de contrôle influençant la forme de la courbe sont $Q_0 = P_{i-1}$, $Q_1 = P_i$, $Q_2 = P_{i+1}$ et $Q_3 = P_{i+2}$. Pour $i = 0$, on peut prendre les points suivants : $Q_0 = P_0 - (P_1 - P_0)$, $Q_1 = P_0$, $Q_2 = P_1$ et $Q_3 = P_2$. Et pour $i = n - 1$, on peut prendre : $Q_0 = P_{n-2}$, $Q_1 = P_{n-1}$, $Q_2 = P_n$ et $Q_3 = P_n + (P_n - P_{n-1})$.

Les points de contrôle B_j $(0 \leq j \leq 3)$ définissant le segment de Bézier cubique sur l'intervalle $[t_i, t_{i+1}]$ peuvent être obtenus via les formules suivantes :

$$
\begin{aligned}
B_0 &= Q_1 \\
B_1 &= \frac{d_1^{2\alpha}Q_2 - d_2^{2\alpha}Q_0 + (2d_1^{2\alpha} + 3d_1^{\alpha}d_2^{\alpha} + 2d_2^{2\alpha})Q_1}{3d_1^{\alpha}(d_1^{\alpha} + d_2^{\alpha})} \\
B_2 &= \frac{d_3^{2\alpha}Q_1 - d_2^{2\alpha}Q_3 + (2d_3^{2\alpha} + 3d_3^{\alpha}d_2^{\alpha} + 2d_2^{2\alpha})Q_2}{3d_3^{\alpha}(d_3^{\alpha} + d_2^{\alpha})} \\
B3 &= Q_2
\end{aligned}
\tag{6}
$$

où $d_i = |Q_i - Q_{i-1}|$.

Le polynôme $P(t)$ définissant la courbe sur l'intervalle $[t_i, t_{i+1}]$ est donc (après expansion des coefficients binomiaux) :

$$
\begin{aligned}
P(t) = B_0 * (1 - \frac{t - t_i}{t_{i+1} - ti})^3 + 3 * B_1 * \frac{t - t_i}{t_{i+1} - ti} * (1 - \frac{t - t_i}{t_{i+1} - ti})^2 \\
+3 * B_2 * (\frac{t - t_i}{t_{i+1} - ti})^2 * (1 - \frac{t - t_i}{t_{i+1} - ti}) + B_3 * (\frac{t - t_i}{t_{i+1} - ti})^3
\end{aligned}
\tag{7}
$$

5.2 Optimisation du temps de rendu de courbes paramétriques

Le défi auquel nous étions confronté est que la génération de courbes paramétriques a un coût relativement élevé dans le cas où l'on veut interagir avec les faisceaux d'arêtes d'un dessin de graphe. Bien que les courbes peuvent être dessinées en un temps raisonnable pour des dessins statiques en utilisant des techniques de rendu standards, le problème devient plus complexe lorsque une technique d'interaction requiert de modifier la forme des arêtes de façon continue. Par exemple, dans l'interaction *Bring & Go* (voir section 1.2.2.1 et Figure 1.16), le rapprochement des voisins d'un sommet implique de modifier les points de contrôle des arêtes concernées à chaque étape de l'animation et donc de recalculer les points de la courbe à la volée. De plus, nous ne voulions pas imposer une borne supérieure sur le nombre de points de contrôle définissant la forme d'une arête. Comme le calcul des points interpolant les courbes posait un vrai poids sur le système, nous avions besoin d'une approche vraiment efficace pour les dessiner. La solution que nous avons conçus évite le plus possible d'effectuer les calculs sur le CPU. Elle repose sur l'utilisation du processeur graphique pour le calcul des points interpolant les courbes.

Dans cette section, nous présentons les implémentations sur processeur graphique pour optimiser le temps de rendu de courbes paramétriques. Après avoir rappelé les techniques classiques pour le rendu de courbes paramétriques, nous détaillons le fonctionnement de notre implémentation optimisée.

5.2.1 Techniques classiques pour le rendu de courbes paramétriques

Cette section vise à introduire les méthodes classiques utilisées pour dessiner des courbes paramétriques ainsi que leurs limitations.

Une approche classique pour le dessin de courbes est de précalculer les points interpolant une courbe au CPU, les stocker en mémoire puis envoyer ces points au processeur graphique pour qu'il la dessine à l'écran. Si cette approche est convenable lorsque le nombre de courbes à dessiner est peu élevé, elle l'est beaucoup moins lorsque l'on veut en rendre finement un grand nombre surtout en termes d'occupation mémoire. Par exemple, pour dessiner 10^5 courbes, avec 200 points par courbes – un point étant stocké comme 3 nombres en point flottant (4 octets chacun), la quantité de mémoire nécessaire pour stocker les points des courbes serait de 240Mo. Si cette quantité peut paraître anecdotique comparée à la quantité de mémoire vive disponible sur les stations de travail actuelles, elle l'est beaucoup moins pour les systèmes embarqués ou mobiles où l'optimisation de l'occupation mémoire d'une application est cruciale.

Une autre solution pour rendre des courbes est d'utiliser les composants intégrés aux API graphiques de haut niveau. Par exemple, en OpenGL, cette tâche peut être achevée en utilisant une fonctionnalité standard appelée *évaluateurs*. Les évaluateurs peuvent être utilisés pour construire des courbes et surfaces basées sur les polynômes en forme de Bernstein. Cela inclue les courbes/surfaces de Bézier ainsi que les courbes/surfaces B-Splines. Un évaluateur est initialisé à partir d'un tableau de points de contrôles et permet de calculer les points interpolant la courbe sur le processeur graphique en envoyant le paramètre t au pipeline de rendu. Cependant, la plupart des implémentations OpenGL ont restreint le nombre maximum de points de contrôle autorisés à huit. Ainsi pour dessiner une courbe de Bézier ou une B-Spline cubique définie par plus de huit points de contrôle, cela doit être fait par morceaux en subdivisant la courbe en courbes définies par moins de points de contrôle. Par conséquent, la performance pour dessiner des courbes paramétrique avec cette technique décroît fortement quand le nombre de points de contrôle augmente. De plus, les évaluateurs ont été dépréciés à partir de la version 3 d'OpenGL.

Si ces deux approches fonctionnent bien pour rendre un petit nombre de courbes définies par un petit nombre de points de contrôle, elles ne sont pas adéquates pour résoudre notre problème de dessiner un grand nombre de courbes définies par plusieurs dizaines de points de contrôle efficacement.

5.2.2 Techniques exploitant le processeur graphique pour le rendu de courbes paramétriques

Nous présentons dans cette section les différentes implémentations sur processeur graphique que nous avons expérimentées pour le rendu de courbes paramétriques. Le principe est de calculer l'ensemble des points interpolant une courbe sur le processeur graphique vu que ce dernier est particulièrement adapté pour le calcul vectoriel. En utilisant l'API graphique de haut niveau OpenGL, nous pouvons encapsuler ces opérations dans un *vertex shader*. Ce type de programme, écrit dans un langage proche du C appelé GLSL (*OpenGL Shading Language*), permet de customiser l'étape de traitement des sommets en entrée du pipeline de rendu (voir Figure 1.21). L'objectif initial de cet étape est de transformer les coordonnées 3D des sommets vers des coordonnées écran en 2D. Nous allons modifier cette étape afin de calculer les points interpolant une courbe. Le principal avantage de ces implémentations, outre la rapidité des calculs en points flottants, vient des capacités de calcul massivement parallèles offerts par les processeurs graphiques contemporains. Ainsi, plusieurs instances du même *vertex shader* peuvent être exécutés en parallèles. Sur les processeurs contemporains, il est possible de calculer une dizaine de points interpolant une courbe en parallèle.

Le principe des différentes implémentations que nous allons présenter est le même pour toutes. Pour chaque courbe à rendre, les points de contrôle la définissant sont accessibles depuis le *vertex shader*. Le paramètre t définissant quel point de la courbe à calculer est transmis dans la composante x des sommets en entrée du pipeline de rendu. Le *vertex shader* va alors interpoler le point de la courbe correspondant puis le transmettre à la prochaine étape du pipeline de rendu. La principale différence entre nos implémentations provient de la manière dont les points de contrôle sont transmis au *shader*. Nous avons ainsi testé trois techniques pour le stockage de ces points :

1. **stockage des points dans un tableau uniforme** : Il est possible de déclarer des variables uniformes de type tableau dans un *shader*. Avant de rendre une courbe, les points de contrôle sont transmis au shader et stockés dans ce tableau. L'avantage de cette technique est que le temps d'accès aux points est extrêmement rapide car on accède à une mémoire locale en lecture seule. L'inconvénient est que le nombre de points de contrôle pouvant être stockés varie selon le modèle du processeur graphique et la quantité de mémoire vidéo disponible. Sur des processeurs graphiques performants, cette solution est tout à fait viable. Par exemple, sur une NVDIA Quadro FX 1800M, il est possible de stocker un millier de points de contrôle dans la mémoire d'un *vertex shader*. Le détail de cette implémentation est présenté en annexe B.1.
2. **stockage des points dans une texture 2D** : Une autre solution, plus générique,

est de stocker l'ensemble des points de contrôle des courbes à rendre dans une texture 2D. Ce type de texture correspond à un tableau de pixels à deux dimensions, chaque pixel étant défini par un vecteur à 4 composantes. La majorité des processeurs graphiques gèrent les pixels définis par des valeurs en points flottants. En fixant la taille de la texture et en transférant un index au *vertex shader*, il est alors possible de récupérer les bons points de contrôle pour chaque courbe à dessiner. Cette technique permet de passer un très grand nombre de points de contrôle au *shader* et devrait pouvoir fonctionner sur un grand nombre de processeurs graphiques. Cependant, le temps d'accès à la mémoire des textures est bien plus long que celui d'accès à la mémoire locale du *shader*. Le code source de cette solution est donné en annexe B.2.

3. **stockage des points dans un *texture buffer object*** : Cette dernière implémentation fait appel à des extensions OpenGL disponibles sur les processeurs graphiques récents. Les points de contrôles sont stockés dans un *texture buffer object*, qui peut être vu comme un grand tableau à une dimension indexable par des entiers. Un très grand nombre de points de contrôle peut y être stocké et le temps d'accès à cette mémoire est significativement plus rapide que celui pour lire une texture 2D. Les courbes sont rendues en utilisant une technique d'*instancing*, permettant de rendre la même géométrie de multiples fois, en modifiant certains paramètres depuis le *shader* via un index généré automatiquement. L'implémentation détaillée de cette solution est présentée en annexe B.3.

Nous proposons également l'implémentation d'un *geometry shader* permettant d'extruder une courbe à la volée afin de lui donner une épaisseur. La courbe est alors rendue sous la forme d'un maillage de triangles. Un *geometry shader* permet de générer de nouvelles primitives à partir de celles envoyées au pipeline de rendu. Il est par exemple possible à partir d'une ligne de générer un ensemble de triangles. L'avantage de cette technique est qu'elle permet d'éviter de générer l'extrusion côté CPU, économisant ainsi du temps CPU et de la mémoire. Le détail de l'implémentation de ce *geometry shader* est donné en annexe B.4.

Détaillons maintenant comment sont calculés les points d'interpolation en fonction du type de courbe à rendre :

- **courbe de Bézier** : Pour calculer un point de ce type de courbe, notre implémentation se contente d'évaluer le polynôme $P(t)$ de l'équation (1). Cependant, l'une des astuces utilisées est le calcul des coefficients binomiaux à la volée, évitant ainsi de transmettre d'autres données au *vertex shader*. Pour une valeur de n fixée, les

coefficients sont calculés itérativement via la formule de récurrence suivante :

$$\begin{aligned} v_0 &= \binom{n}{0} = 1 \\ v_{0<i\leq n} &= \binom{n}{i} = v_{i-1}\left(\frac{(n+1)-i}{i}\right) \end{aligned} \tag{8}$$

L'implémentation en GLSL est donnée en annexe B.5.1. Les tests effectués ont montré que cette implémentation reste numériquement stable pour calculer des courbes de Bézier ayant jusqu'à 120 points de contrôle. Des problèmes de précision de calcul en point flottant peuvent apparaître au delà. D'autres techniques ont été proposées pour rendre des courbes de Bézier au GPU. Par exemple, Loop et Blinn [130] ont présenté une méthode pour rendre des chemins et des régions définis par des courbes de Bézier cubiques et quadratiques. Leur approche, principalement dédiée pour le rendu de police de caractères, est basé sur l'utilisation d'un *fragment shader*. Un polygone dont les sommets correspondent aux points de contrôle est rendu et le *fragment shader* va écarter les pixels ne se trouvant pas dans le polygone défini par la courbe.

- **courbe B-Spline** : Pour calculer les points interpolant une courbe B-Spline, notre implémentation va évaluer le polynôme $P(t)$ introduit à l'équation (3). Le calcul des fonctions de base B-Spline est aussi effectué par le *vertex shader* en utilisant une version optimisée de l'algorithme de Cox-de Boor (voir [166] et Algorithme 1). Nous avons développé deux versions de notre implémentation pour les courbes B-Spline : une optimisée pour les B-Splines uniformes (i.e. les nœuds internes de multiplicité 1 sont uniformément espacés) et une autre générique où les valeurs des nœuds sont définies par l'utilisateur. Les codes sources en GLSL sont présentés en annexe B.5.2.

- **courbe de Catmull-Rom** : Pour rendre ce type de courbe, notre implémentation va dans un premier temps déterminer les points de contrôle influençant la forme de la courbe pour le paramètre t. Les points de contrôle du segment de Bézier cubique associé sont ensuite calculés selon l'équation (6). Enfin, le point de la courbe associé au paramètre t est calculé en évaluant le polynôme défini à l'équation (7). A noter qu'un paramètre supplémentaire est transmis au *vertex shader*, à savoir la somme des distances entre chaque point de contrôle successif (voir dénominateur dans l'équation (5)). Ceci afin d'éviter de la recalculer à chaque fois qu'un point de la courbe doit être interpolé. Le code source en GLSL est donné en annexe B.5.3.

Algorithme 1 Algorithme optimisé de Cox-de Boor (voir [166])

Entrées:
- n : nombre de point de contrôle
- p : degré de la courbe
- $m+1$ noeuds dans $[0,1]$: $\{t_0, \ldots, t_m\}$
- t : le paramètre dans $[0,1]$ définissant le point de la courbe à interpoler

Sorties: - les coefficients $N_{0,p}(t), N_{1,p}(t), \ldots, N_{n-1,p}(t)$ dans un tableau N

Pour i de 0 à $n-1$ **faire**
    $\mathrm{N}[i] = 0$
Fin Pour
Si $t == t_0$ **alors**
    $\mathrm{N}[0] = 1.0$ **Retourner**
Sinon Si $t == t_m$ **alors**
    $\mathrm{N}[n-1] = 1.0$ **Retourner**
Fin Si
Soit t dans l'intervalle de nœuds $[t_k, t_{k+1}[$
$\mathrm{N}[k] = 1.0$
Pour d de 1 à p **faire**
    $\mathrm{N}[k-d] = \dfrac{t_{k+1} - t}{t_{k+1} - t_{(k-d)+1}} * \mathrm{N}[(k-d)+1]$
    Pour i de $k-d+1$ à $k-1$ **faire**
        $\mathrm{N}[i] = \dfrac{t - t_i}{t_{i+d} - t_i} * \mathrm{N}[i] + \dfrac{t_{i+d+1} - t}{t_{i+d+1} - t_{i+1}} * \mathrm{N}[i+1]$
    Fin Pour
    $\mathrm{N}[k] = \dfrac{t - t_k}{t_{k+d} - t_k} * \mathrm{N}[k]$
Fin Pour

5.2.3 Performances

Pour évaluer et comparer les performances des différentes implémentations de rendu de courbes paramétriques, nous avons utilisé trois jeux de données. Le premier (voir Figure 5.5(a)) contient 5000 courbes, chacune définie par 40 points de contrôle générés aléatoirement. Le second (voir Figure 5.5(b)) est un dessin du sous-réseau du réseau mondial des transports aériens de l'année 2000, contenant plus de 4000 arêtes. Notre algorithme de regroupement d'arêtes [123] a été appliqué, générant un nombre moyen de 30 points de contrôle par arête. Le dernier jeu de données (voir Figure 5.5(c)) est un dessin par modèle de force d'un graphe d'appel de fonctions où notre algorithme de regroupement d'arêtes a été appliqué. Ce graphe contient plus de 11000 arêtes et le nombre moyen de points de contrôle par arête est de 16.

Nous avons comparé les performances de rendu (ligne et polygone) entre les trois implémentations au GPU présentées dans la section 5.2.2 mais aussi avec celles plus classiques consistant à calculer les points interpolant une courbe au CPU (avec ou sans mise en cache). Pour chaque courbe à dessiner, le nombre de points d'interpolations a été fixé à 200. Les performances sont évaluées en terme de nombre d'images par seconde mais également en termes d'occupation mémoire des données à transmettre au pipeline de rendu graphique. Dans le cas des implémentations au GPU, ces données correspondent aux points de contrôle des courbes à rendre ainsi que quelques autres annexes : couleur, taille, ... Dans le cas des implémentations au CPU, ces donnés correspondent à l'ensemble des points interpolant les courbes à rendre ainsi que les données de couleur associées.

La Table 5.1 présente les résultats de ces tests de performance. La machine utilisée pour effectuer ces tests dispose d'un processeur Intel(R) Core(TM) i7 CPU X940 @ 2.13GHz et d'un processeur graphique Nvidia Quadro FX 1800M. La taille de la fenêtre de rendu OpenGL est de 1920×1080 pixels. Pour le rendu B-Spline, l'implémentation optimisée pour les B-Splines uniformes a été utilisée. A noter que les résultats de rendu des courbes B-Splines et de Catmull-Rom pour le premier jeu de données ne sont pas vraiment exploitables. Ceci est dû au fait que le nombre d'opérations par pixels pour ce jeu de données et ces types de courbe est bien plus important, masquant les performances en amont du pipeline graphique.

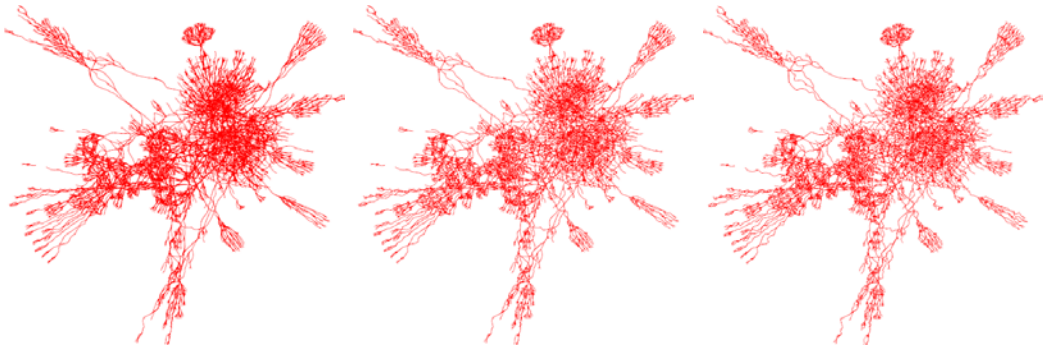
Les résultats montrent que l'implémentation au GPU la plus efficace est celle où les données sont stockées dans un tableau uniforme, suivi de près par celle avec stockage dans un *texture buffer object*. L'implémentation avec stockage des données dans une texture 2D est quant à elle bien moins efficace. Au niveau du type de courbes, les B-Splines sont les plus rapides à rendre, suivi par les Bézier. Les courbes de Catmull-Rom sont les moins rapides à calculer au GPU, dû au grand nombre de branchements dans l'implémentation. Néanmoins les performances de rendu des implémentations GPU restent très bonnes. Par exemple, pour le deuxième jeu de données, le nombre d'images par seconde pour le rendu des courbes de Bézier est de 25, celui des B-Splines de 34 et celui des courbes de Catmull-Rom de 22. L'implémentation la plus rapide est bien évidemment celle où les points interpolant la courbe ont été précalculés au CPU et mis en cache. Cependant la quantité de mémoire nécessaire pour stocker les données nécessaires au rendu des courbes est de 5 à 17 fois plus importante que celle requise par les implémentations au GPU. L'implémentation la moins efficace est celle où chaque courbe est interpolée au CPU à chaque passe de rendu (moins de 1 image par seconde), mettant en exergue le coût de calcul relativement élevé des courbes paramétriques.

Ces résultats montrent que le rendu de courbes paramétriques au GPU permet de mettre en place des visualisations interactives, grâce à un bon taux de rafraîchissement

d'image, tout en économisant une quantité importante de mémoire. De plus, les implémentations au GPU devraient se révéler plus efficace dans le contexte d'animations de courbes car seuls les points de contrôle doivent être modifiés à chaque passe de rendu. Outre la visualisation de graphe, d'autres type de visualisation pourrait bénéficier de ces implémentations, comme par exemple les coordonnées parallèles où il est courant de représenter les données comme un ensemble de courbes.

(a)

(b)



(c)

FIGURE 5.5: Les trois jeux de données utilisés pour comparer les performances des différentes techniques de rendu de courbes paramétriques. Pour chaque jeu de données, le résultat des rendus suivants sont présentés : de gauche à droite, courbes de Bézier, courbes B-Spline et courbes de Catmull-Rom. Pour chaque courbe à rendre, 200 points d'interpolation sont calculés. (a) 5000 courbes définies par 40 points de contrôle aléatoirement générés. (b) Le dessin géolocalisé du sous-réseau européen du réseau mondial des transports aériens de l'année 2000 (433 sommets / 4343 arêtes) où notre algorithme de regroupement d'arêtes a été appliqué. Les statistiques concernant le nombre de points de contrôle par arête sont les suivantes : minimum = 3, maximum = 79, moyenne = 30. (c) Le dessin par l'algorithme FM^3 [94] d'un graphe d'appel de fonctions dans un gros projet logiciel (5471 sommets / 11472 arêtes) où notre algorithme de regroupement d'arêtes a été appliqué. Les statistiques concernant le nombre de points de contrôle par arête sont les suivantes : minimum = 3, maximum = 63, moyenne = 16.

		Jeu de données 1		Jeu de données 2		Jeu de données 3	
		nombre d'images par seconde	occupation mémoire (en Mo)	nombre d'images par seconde	occupation mémoire (en Mo)	nombre d'images par seconde	occupation mémoire (en Mo)
– rendu Bézier au GPU – données stockées dans un tableau uniforme	ligne	~ 18	~ 3.74	~ 25	~ 2.45	~ 18	~ 4.03
	polygone	~ 11	~ 3.74	~ 16	~ 2.45	~ 10	~ 4.03
– rendu Bézier au GPU – données stockées dans une texture 2D	ligne	~ 6.5	~ 3.74	~ 9.5	~ 2.45	~ 6.5	~ 4.03
	polygone	~ 4	~ 3.74	~ 6	~ 2.45	~ 4	~ 4.03
– rendu Bézier au GPU – données stockées dans un *texture buffer object*	ligne	~ 14.5	~ 3.74	~ 20	~ 2.45	~ 14	~ 4.03
	polygone	~ 8	~ 3.74	~ 12	~ 2.45	~ 8	~ 4.03
– rendu Bézier classique – points de la courbe interpolés au CPU à chaque rendu	ligne	~ 0.5	~ 3.74	~ 0.7	~ 2.45	~ 0.5	~ 4.03
	polygone	~ 0.4	~ 3.74	~ 0.6	~ 2.45	~ 0.37	~ 4.03
– rendu Bézier classique – points de la courbe interpolés au CPU et mis en cache	ligne	~ 18	~ 15.26	~ 38	~ 13.12	~ 31.5	~ 34.82
	polygone	~ 11	~ 30.52	~ 33	~ 26.24	~ 36	~ 69.64
– rendu B-Spline au GPU – données stockées dans un tableau uniforme	ligne	~ 5.5	~ 3.74	~ 34	~ 2.45	~ 27	~ 4.03
	polygone	~ 3	~ 3.74	~ 22	~ 2.45	~ 11	4.03
– rendu B-Spline au GPU – données stockées dans une texture 2D	ligne	~ 5.5	~ 3.74	~ 28.5	~ 2.45	~ 18.5	~ 4.03
	polygone	~ 3	~ 3.74	~ 19	~ 2.45	~ 11	4.03
– rendu B-Spline au GPU – données stockées dans un *texture buffer object*	ligne	~ 5.5	~ 3.74	~ 33.5	~ 2.45	~ 23	~ 4.03
	polygone	~ 3	~ 3.74	~ 22	~ 2.45	~ 11	4.03
– rendu B-Spline classique – points de la courbe interpolés au CPU à chaque rendu	ligne	~ 4.5	~ 3.74	~ 5.5	~ 2.45	~ 2.5	~ 4.03
	polygone	~ 1.3	~ 3.74	~ 1.6	~ 2.45	~ 0.6	4.03
– rendu B-Spline classique – points de la courbe interpolés au CPU et mis en cache	ligne	~ 5.5	~ 15.26	~ 39	~ 13.12	~ 33	~ 34.82
	polygone	~ 3	~ 30.52	~ 33	~ 26.24	~ 32	~ 69.64
– rendu Catmull-Rom au GPU – données stockées dans un tableau uniforme	ligne	~ 3	~ 3.74	~ 22	~ 2.45	~ 16	~ 4.03
	polygone	~ 1.5	~ 3.74	~ 15	~ 2.45	~ 10	~ 4.03
– rendu Catmull-Rom au GPU – données stockées dans une texture 2D	ligne	~ 3	~ 3.74	~ 10	~ 2.45	~ 7	~ 4.03
	polygone	~ 1.5	~ 3.74	~ 6	~ 2.45	~ 4	~ 4.03
– rendu Catmull-Rom au GPU – données stockées dans un *texture buffer object*	ligne	~ 3	~ 3.74	~ 19	~ 2.45	~ 13	~ 4.03
	polygone	~ 1.5	~ 3.74	~ 12	~ 2.45	~ 8	~ 4.03
– rendu Catmull-Rom classique – points de la courbe interpolés au CPU à chaque rendu	ligne	~ 2.5	~ 3.74	~ 3	~ 2.45	~ 1.5	~ 4.03
	polygone	~ 1	~ 3.74	~ 1.3	~ 2.45	~ 0.5	~ 4.03
– rendu Catmull-Rom classique – points de la courbe interpolés au CPU et mis en cache	ligne	~ 3	~ 15.26	~ 39	~ 13.12	~ 32	~ 34.82
	polygone	~ 1.5	~ 30.52	~ 33	~ 26.24	~ 33	~ 69.64

TABLE 5.1: Comparaison des performances entre les différentes techniques de rendu de courbes paramétriques pour différents jeux de données (voir Figure 5.5).

Chapitre 6

Edge splatting : une technique pour visualiser la densité des faisceaux d'arêtes

Dans ce chapitre nous présentons une technique de rendu exploitant le processeur graphique pour extraire l'information de densité des faisceaux d'arêtes dans un dessin de graphe où un algorithme de regroupement d'arêtes a été appliqué. Nous avons publié cette méthode de rendu dans [123], en complément de notre algorithme de regroupement d'arêtes.

Quand un algorithme de regroupement d'arêtes a été appliqué sur un dessin de graphe, des faisceaux d'arêtes émergent du dessin donnant une jolie impression de flux entre différentes régions du dessin de graphe. Néanmoins, l'information concernant le nombre d'arêtes contenues dans un faisceau n'est pas facilement visible dans le dessin. Afin de distinguer les faisceaux contenant beaucoup d'arêtes de ceux en contenant peu, nous proposons une technique de rendu, que nous avons nommée *edge splatting*, permettant de les mettre visuellement en exergue.

6.1 Principe de la technique

Notre méthode est inspirée de la technique *GraphSplatting* introduite par van Liere *et al.* dans [183]. Dans ce travail, les auteurs représentent un graphe comme un champ scalaire 2D continu appelé *splat field*. Cette technique fournit un moyen de visualiser les variations continues dans la densité des sommets (en terme de position dans le dessin), ce qui peut aider à déterminer la structure du graphe sous-jacent. Un autre travail partageant des similitudes avec cette technique est celui de Chiricota *et al.* [42]. Ils proposent une technique pour sélectionner des éléments dans une représentation de type *scatter plot* en appliquant un flou gaussien sur l'image associée. Cet effet de flou transforme les points

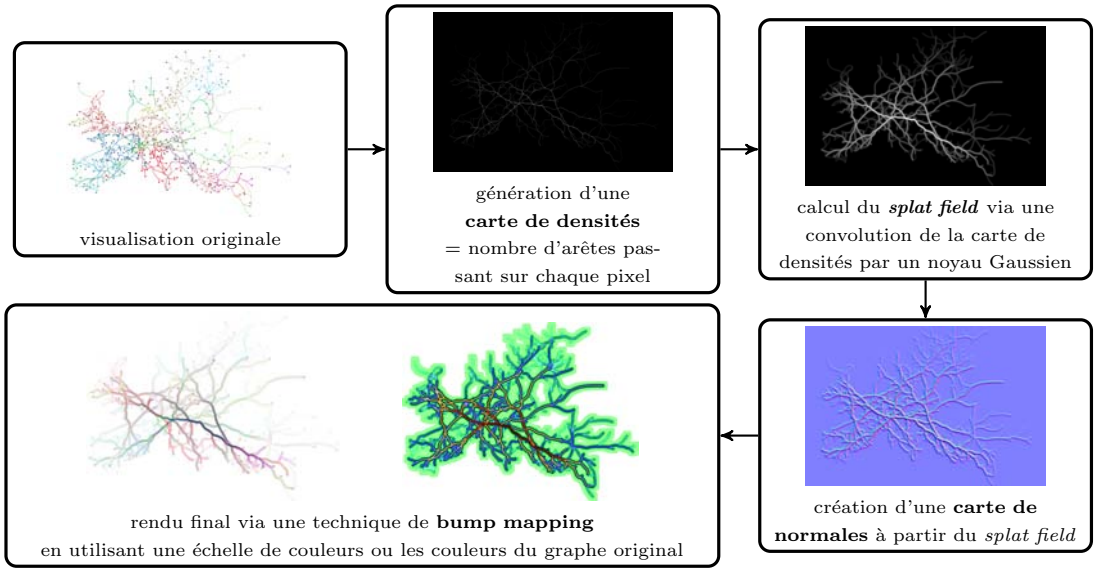


FIGURE 6.1: Pipeline de rendu de notre technique *edge splatting* permettant de visualiser la densité des faisceaux d'arêtes dans un dessin de graphe où un algorithme de regroupement d'arêtes a été appliqué.

formant une sous-région dense du scatterplot en une large tâche qui peut aider l'utilisateur à identifier et sélectionner des groupes d'éléments dans les données sous-jacentes.

La Figure 6.1 présente le pipeline de rendu pour notre technique. Il est constitué d'une combinaison de méthodes classiques de traitement d'image ainsi que des techniques de rendu graphique. Chaque étape est exécutée sur le processeur graphique. D'une façon similaire à la technique *GraphSplatting*, l'idée est de calculer un *splat field* encodant les variations continues dans la densité des arêtes regroupées. Ce *splat field* peut ensuite être visualisé de différentes manières. Nous avons exploré deux solutions pour encoder visuellement la densité des faisceaux d'arêtes. La première associe simplement les valeurs de densité à des couleurs en fonction d'une échelle de couleur définie par l'utilisateur. La seconde utilise une technique d'éclairage (*shading*) par pixel associant les valeurs de densité à des hauteurs donnant l'impression que les faisceaux contenant beaucoup d'arêtes apparaissent plus haut que ceux en contenant peu. L'avantage de cette solution par rapport à la précédente est qu'elle permet de conserver les couleurs originales des arêtes et ainsi conserver l'information qu'elles encodent.

6.2 Détails d'implémentation

Pour illustrer notre technique de rendu, nous l'appliquerons au graphe de migrations de travailleurs [31] aux États-Unis présenté à la Figure 6.2(a). Dans cette visualisation,

les arêtes ont été colorées en fonction de l'état de destination.

La première étape de notre technique de rendu *edge splatting* est de calculer le nombre d'arêtes passant sur chaque pixel du dessin et ainsi construire une *carte de densité*. Comme dans [100], cette opération peut être réalisée en effectuant un rendu hors écran des arêtes du graphe dans un tampon d'accumulation. Avec l'API graphique de haut niveau OpenGL, cela peut être implémentée en effectuant un rendu dans une texture via l'utilisation d'un *Frame Buffer Object* avec une texture en point flottant attachée. Pour générer la carte de densité, la même couleur est assignée à chaque arête et un *blending* additif est activé durant la phase de rendu.

L'étape suivante de notre pipeline est le calcul du *splat field*. La texture résultante de l'étape précédente est un champ de valeurs discrètes encodant le nombre d'arêtes passant sur chaque pixel. Le but de cette étape est de le transformer en un champ de valeurs continues. Ce procédé peut être réalisé en convoluant le champ de valeurs discrètes avec un noyau Gaussien défini par un rayon r et un écart type σ. Cette opération peut toujours être exécutée sur le processeur graphique en effectuant un rendu dans une texture et en écrivant un *fragment shader* pour réaliser la convolution. Comme expliqué dans la section 1.3.2, un *fragment shader* est un programme visant à customiser l'unité de traitement des pixels du pipeline de rendu dont le rôle est de calculer la couleur des pixels à afficher. Comme il est possible dans un *fragment shader* de lire des pixels depuis une texture stockée en mémoire, implémenter une convolution de valeurs discrètes par un noyau Gaussien est une tâche aisée. Notre implémentation tire également avantage du fait qu'un filtre Gaussien est linéairement séparable. Ainsi, l'opération de convolution peut être divisée en deux passes [173]. De cette façon, les performances ne se dégradent pas lorsque le rayon du noyau augmente. Le code source de ce *fragment shader* est généré dynamiquement en fonction du rayon du noyau choisi par l'utilisateur. L'écart type du noyau Gaussien peut également être modifié dynamiquement. Plus le rayon et l'écart type augmente, plus le *splat field* est lissé. Afin de pouvoir associer les valeurs du *splat field* à des couleurs ou des hauteurs, ses valeurs minimum et maximum sont déterminés en utilisant une opération de réduction sur processeur graphique [118].

La première solution pour visualiser le *splat field* calculé est d'effectuer un rendu basé sur un gradient, associant les valeurs de densité à des couleurs en fonction d'une échelle définie par l'utilisateur (voir Figure 6.2(b)). Un quadrilatère de la même taille que la texture contenant le *splat field* est alors rendu à l'écran et la coloration des pixels est effectuée par un *fragment shader* dédié associant les valeurs de densité au gradient défini par l'utilisateur. L'association des valeurs de densité à des couleurs peut se faire de façon linéaire ou logarithmique.

Afin d'améliorer la représentation du *splat field*, nous proposons une étape supplémentaire de rendu basée sur une technique de *bump mapping*. C'est une technique de rendu graphique introduite par Blinn [22] permettant de rendre une surface de façon plus réaliste sans modifier la géométrie. Elle consiste en une technique d'éclairage (*shading*) par pixel qui donne l'impression que la surface rendue est bosselée via une modification des normales de la surface. Les couleurs et luminosités des pixels de la surface rendue sont altérés en fonction de ces normales en utilisant un algorithme d'éclairage. Les normales modifiées pour chaque pixel sont stockées dans une texture appelée *carte de normales*, les composantes RGB des des pixels stockant les valeurs X, Y et Z des vecteurs normaux. Cette carte de normales est générée à partir d'une *carte de hauteur*, une autre texture stockant les valeurs de hauteur associée à chaque pixel de la surface. Le *bump mapping* est classé dans les méthodes d'association de déplacement (*displacement mapping*) par pixel. Pour plus de détails sur les algorithmes d'association de déplacement, nous recommandons la lecture de l'étude de Szirmay-Kalos *et al.* [177].

Récemment, Willems *et al.* [194] ont proposé une visualisation géographique de mouvements de navires où ils utilisent une technique d'éclairage (*shading*) pour mettre en exergue des zones maritimes significatives comme les autoroutes où les zones d'ancrage. Leur visualisation est basée sur des champs de densité dérivés de la convolution des positions dynamiques des navires avec un noyau. Dans notre cas, si nous associons les valeurs de densité du *splat field* à des hauteurs (de façon linéaire ou logarithmique), nous pouvons utiliser le *bump mapping* pour améliorer les visualisations de graphe avec regroupement d'arêtes. Les faisceaux contenant beaucoup d'arêtes apparaîtront plus haut que les autres avec cette technique et émergeront visuellement du dessin. Pour achever cet effet, nous avons dans un premier temps besoin de générer une carte de hauteur à partir du *splat field* calculé. Cette carte peut être générée simplement en colorant les valeurs de densité pour chaque pixel suivant une échelle de couleur allant du noir au blanc : le noir représentant la hauteur minimum et le blanc la hauteur maximum. Ensuite, nous pouvons générer la carte de normales en calculant le gradient de la carte de hauteurs via l'utilisation par exemple d'un filtre de Sobel ou de Prewitt [90]. Cette opération peut être exécutée sur le processeur graphique via un *fragment shader*. Ce programme calcule pour chaque pixel les dérivés de la carte de hauteur dans les directions horizontales et verticales en utilisant l'opérateur de gradient et construit les normales associées à partir de ces valeurs.

Une fois que la carte de normales associée au *splat field* a été générée, le rendu *bump mapping* peut être effectué. Un *fragment shader* dédié va se charger de réaliser cette opération, lisant les normales modifiées pour chaque pixel depuis la texture contenant la carte de normales et effectuant une illumination par pixel en utilisant l'algorithme d'éclairage de Blinn-Phong [21]. Les couleurs finales des pixels sont calculées à partir des propriétés

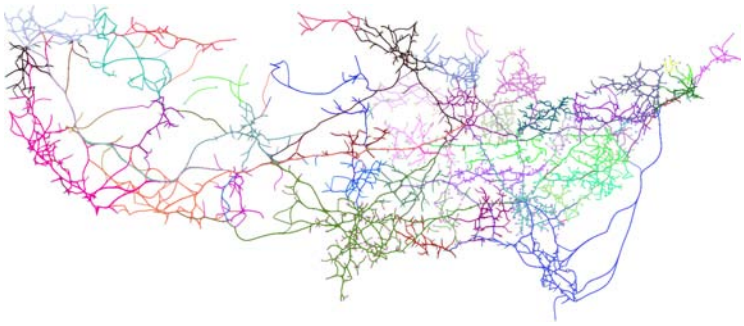
d'éclairage et d'une autre texture appelée la *carte de diffusion*. Dans notre cas, cette carte correspond à la coloration du *splat field* suivant une échelle de couleur (voir Figure 6.2(c)) où les couleurs originales du dessin de graphe (voir Figure 6.2(d)). Pour effectuer une illumination globale, la source d'éclairage est définie comme étant directionnelle avec chaque rayon de lumière parallèle à l'axe Z. Notre système de visualisation laisse également la possibilité à l'utilisateur de configurer la couleur ambiante, diffuse et spéculaire de la source de lumière.

La méthode a également été adaptée pour visualiser la densité des faisceaux d'arêtes sur des dessins de graphe sphériques [121]. Les modifications apportées à notre pipeline de rendu pour ce type de dessin particulier sont les suivantes. Pour le calcul de la carte de densité, nous restreignons le calcul du nombre d'arêtes passant sur chaque pixel à celles routées sur la face couramment visible de la sphère. La phase de rendu *bump mapping* a également été modifiée car nous voulons l'appliquer sur une surface sphérique et non plus plate. Pour cela, nous avons ajouté une étape supplémentaire à notre pipeline de rendu. Son but est de calculer pour chaque pixel de la surface visible de la sphère la transformation du vecteur d'éclairage (position de la source d'éclairage) et du vecteur de l'observateur (position de l'observateur de la scène) dans l'*espace tangent*. Cet espace est localement tangent à la surface rendue et est utilisé pour calculer les couleurs finales des pixels dans un contexte de *bump mapping*. Ce processus est réalisé en rendant une sphère dont le centre et le rayon sont les mêmes que ceux du dessin du graphe. Les informations relatives à l'espace tangent sont attachées à chaque sommet du maillage modélisant la sphère. Un *fragment shader* dédié va alors transformer le vecteur d'éclairage et le vecteur de l'observateur passés en paramètre dans l'espace tangent. Les résultats sont stockés dans deux textures en point flottant. Ces valeurs seront alors utilisés par un autre *fragment shader* simulant ainsi le rendu *bump mapping* du *splat field* sur une sphère.

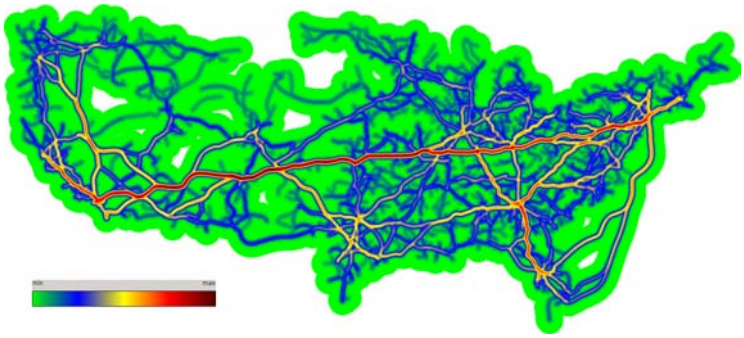
6.3 Exemples d'application sur des visualisations de réseaux géographiques

Nous présentons ici deux exemples d'application de notre méthode de rendu *edge splatting* sur les visualisations de deux réseaux géographiques où notre méthode de regroupement d'arêtes [123] a été appliquée.

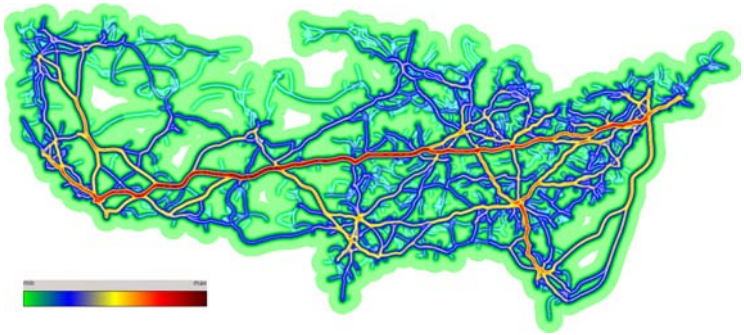
La Figure 6.3 présente le résultat de la méthode sur le réseau d'interconnexions des aéroports internationaux en 2004 (source des données : projet ANR Spangeo). La densité des faisceaux est matérialisée par des couleurs ainsi que l'effet de *bump mapping*. L'échelle de couleur utilisée est linéaire et correspond à celle de la Figure 6.2(b). On peut observer



(a) visualisation originale



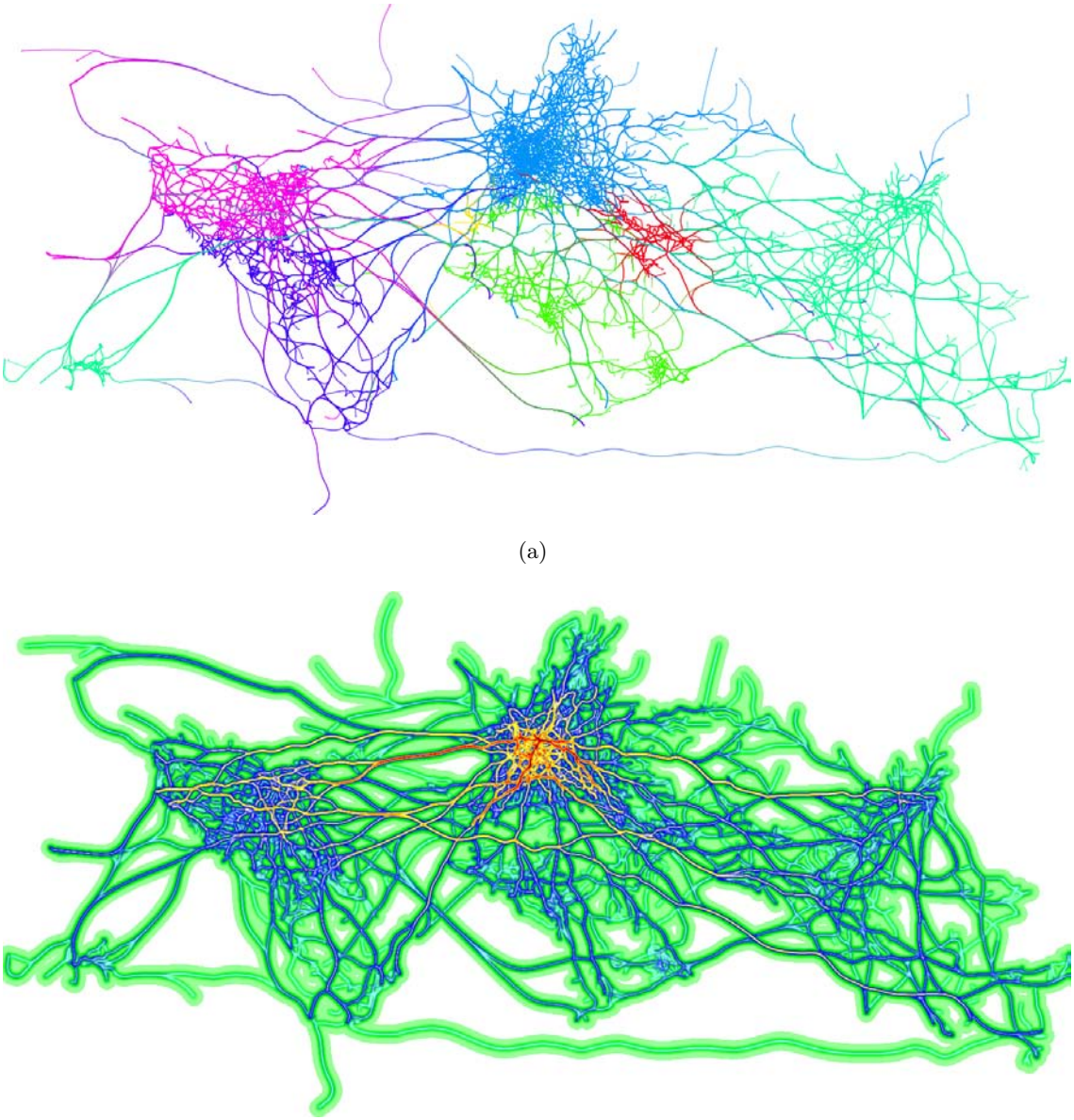
(b) rendu *edge splatting* simple associant les valeurs de densité à des couleurs



(c) rendu *edge splatting* avec *bump mapping* associant les valeurs de densité à des couleurs

(d) rendu *edge splatting* avec *bump mapping* préservant les couleurs de la visualisation originale

FIGURE 6.2: Illustrations de notre technique de rendu *edge splatting* pour visualiser la densité des faisceaux d'arêtes.



(a)

(b)

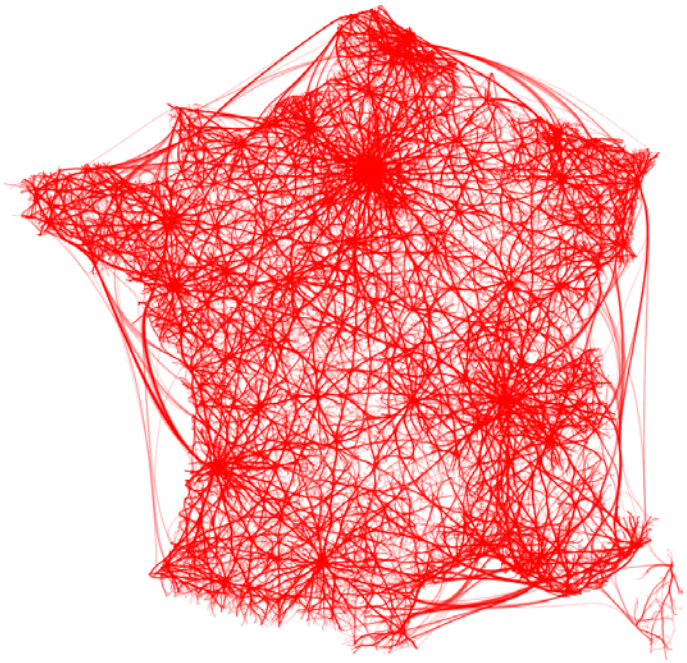
FIGURE 6.3: Application de notre méthode de rendu *edge splatting* sur la visualisation avec regroupements d'arêtes du réseau d'interconnexions des aéroports internationaux en 2004 (source des données : projet ANR Spangeo) (a) Visualisation originale. (b) Visualisation avec rendu *edge splatting*.

par exemple qu'il y a beaucoup de connexions entre les aéroports américains et européens. De même, on remarque qu'il y a beaucoup moins de vols internes en Afrique et Amérique du Sud qu'en Europe et Amérique du Nord.

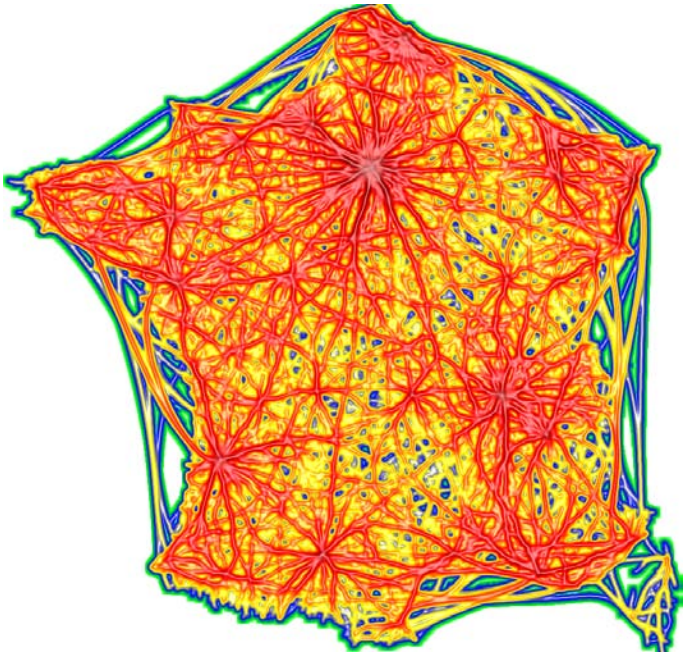
La Figure 6.4 montre le résultat de la méthode sur la visualisation avec regroupement

d'arêtes du réseau de flux de travailleurs en France établi à partir des données de recensement de l'INSEE pour l'année 1975. La densité des faisceaux est matérialisée par des couleurs ainsi que l'effet de *bump mapping*. L'échelle de couleur utilisée est logarithmique et correspond à celle de la Figure 6.2(b). On observe que les grands viviers d'emploi en France à cette époque ressortent clairement du dessin (e.g. Paris, Lyon, Bordeaux, Toulouse). De même, on peut également identifier les grands axes de déplacement de travailleurs comme par exemple la vallée du Rhône.

La Figure 6.5 présente les résultats obtenus avec notre méthode *edge splatting* adaptée pour les dessins de graphe sphériques. Le graphe visualisé est le réseau d'interconnexions des aéroports internationaux de 2004 dessiné sur le globe terrestre (source des données : projet ANR Spangeo). Les arêtes ont été regroupées en utilisant la version de notre algorithme de regroupement d'arêtes pour les dessins de graphe sphérique [121] (voir section 3.3.2).

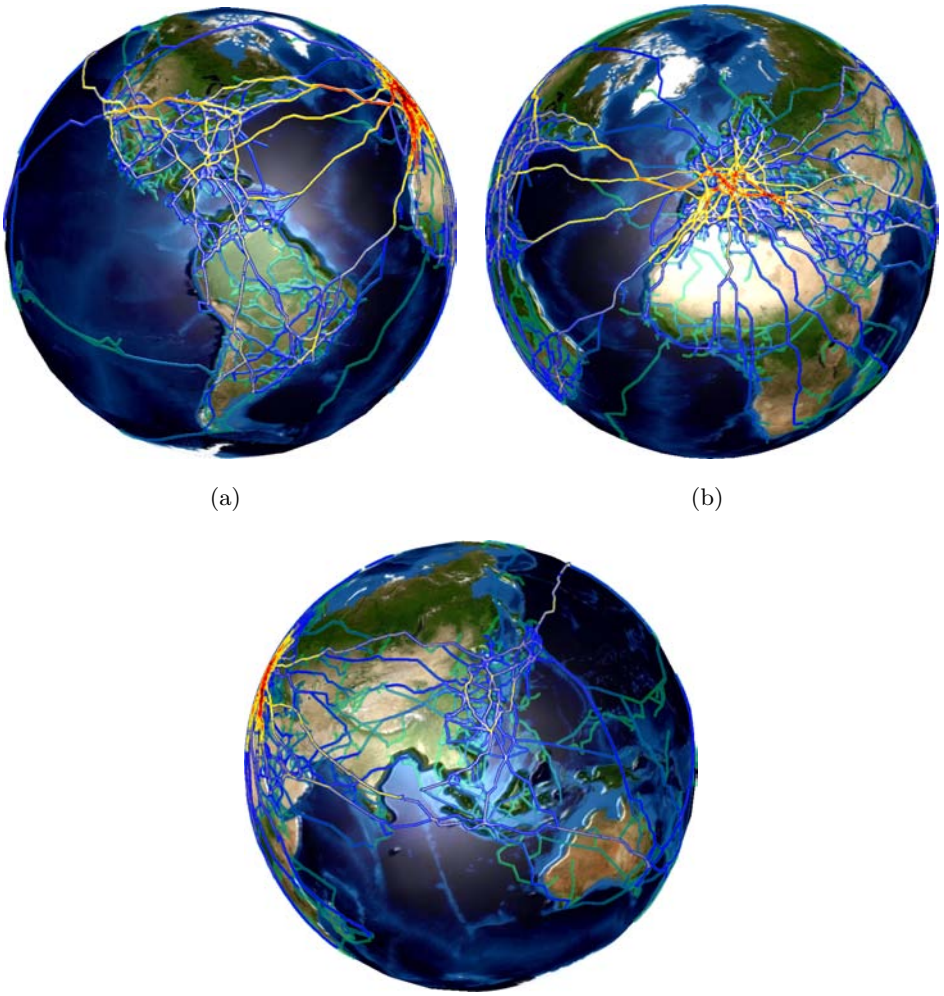


(a)



(b)

FIGURE 6.4: Application de notre méthode de rendu *edge splatting* sur la visualisation avec regroupement d'arêtes du réseau de flux de travailleurs en France établi à partir des données de recensement de l'INSEE pour l'année 1975. Ce réseau représente les déplacements effectués par chaque travailleur de son domicile à son lieu de travail. (a) Visualisation originale. (b) Visualisation avec rendu *edge splatting*.



(a) (b)

(c)

FIGURE 6.5: Exemple d'application de notre méthode de rendu *edge splatting* adaptée pour les dessins de graphe sphériques. Le graphe visualisé est le réseau d'interconnexions des aéroports internationaux de 2004 dessiné sur le globe terrestre (source des données : projet ANR Spangeo).

Chapitre 7

Visualisation de sous-ensembles d'intérêt d'un réseau dans un contexte global

Dans ce chapitre, nous nous intéressons à des techniques pour mettre visuellement en exergue des sous-graphes d'intérêt dans le contexte global d'une visualisation de graphe.

L'analyse de grands jeux de données repose souvent sur la découverte de différents *motifs*. C'est le cas avec des structures de données discrètes tels les graphes qui sont essentiels pour l'étude de données relationnelles. Un exemple connu de découverte de motifs dans l'analyse de graphe est la détection de communautés, correspondant à des sous-graphes hautement connectés. La structure de communauté est une composante importante des réseaux sociaux, internet ou d'interaction de protéines [89]. Révéler cette structure fournit une meilleure compréhension de sa dynamique. On peut par exemple considérer les sommets connectés avec différentes communautés comme médiateurs dans un réseau. La détection de communauté a fait l'objet de nombreuses investigations durant les dernières années [75].

Cependant, partitionner un réseau est un moyen parfois trop simpliste pour l'analyser en détail. Par exemple, dans un réseau social une personne peut faire partie de plusieurs groupes tels que ceux formés par les membres de sa famille ou ses collègues de travail. Une simple partition ne permet donc pas de capturer cette multi-modalité qui apparaît souvent dans ce type de réseau. Il existe donc des méthodes [3, 126, 145] permettant de détecter des communautés chevauchantes où des sommets/arêtes peuvent être partagés entre plusieurs groupes. Un autre exemple de décomposition chevauchante d'un graphe est le découpage en voies métaboliques d'un réseau métabolique (voir chapitre 4). Pouvoir visualiser efficacement le résultat d'une décomposition de graphe est donc primordial afin d'en tirer une analyse.

Après avoir fait un rapide état de l'art des solutions existantes, nous présentons deux méthodes que nous avons développées pour réaliser cette tâche. La première consiste en

la génération d'enveloppes concaves pour entourer un sous-graphe d'intérêt. La seconde, publiée dans [124] dans le cadre de notre travail sur la représentation de réseaux métaboliques (voir chapitre 4), repose sur l'utilisation d'une déformation 3D afin de faire ressortir visuellement un sous-ensemble d'un réseau. Dans le cas des enveloppes concaves, deux méthodes seront introduites. La première, également publiée dans [124], génère des enveloppes à partir de l'espace image d'une visualisation de graphe. La seconde calcule des enveloppes à partir de l'espace topologique et a l'avantage par rapport à la première d'être plus efficace pour la visualisation de sous-ensembles chevauchants dans un réseau. Cette dernière technique a été publiée dans les actes de la *16ème conférence internationale sur la Visualisation d'Information (IV'12)* [125].

7.1 État de l'art

La visualisation d'ensembles chevauchants, au sens général du terme, a été largement investiguée durant les dernières années.

Dans la communauté du dessin de graphe, ce problème revient à représenter non plus un graphe mais un hypergraphe. Un hypergraphe $H = (V, E)$ est la généralisation d'un graphe où V est toujours un ensemble de sommets mais où E est un ensemble de sous-ensembles non vides de V, appelés hyper-arêtes. L'ensemble $E \subseteq \mathcal{P}(V)$ d'hyper-arêtes est un sous-ensemble de l'ensemble des parties de V. Si les sommets sont toujours représentés comme des points dans le plan, les hyper-arêtes quant à elles peuvent avoir différents types de représentations. Par exemple, elles peuvent être dessinées comme des courbes fermées entourant uniquement les sommets qu'elles relient. Bertault et Eades expliquent dans [18] comment créer de telles représentations. Une autre méthode pour représenter une hyper-arête est d'utiliser une métaphore visuelle proche de celle utilisée dans les représentations nœuds-liens standards (voir [132]). Dans ce cas les hyper-arêtes sont représentées comme un ensemble de segments formant des arbres de Steiner (voir [29]). D'autres méthodes de représentations d'hypergraphes existent comme par exemple la méthode par subdivision de Kaufmann *et al.* [114].

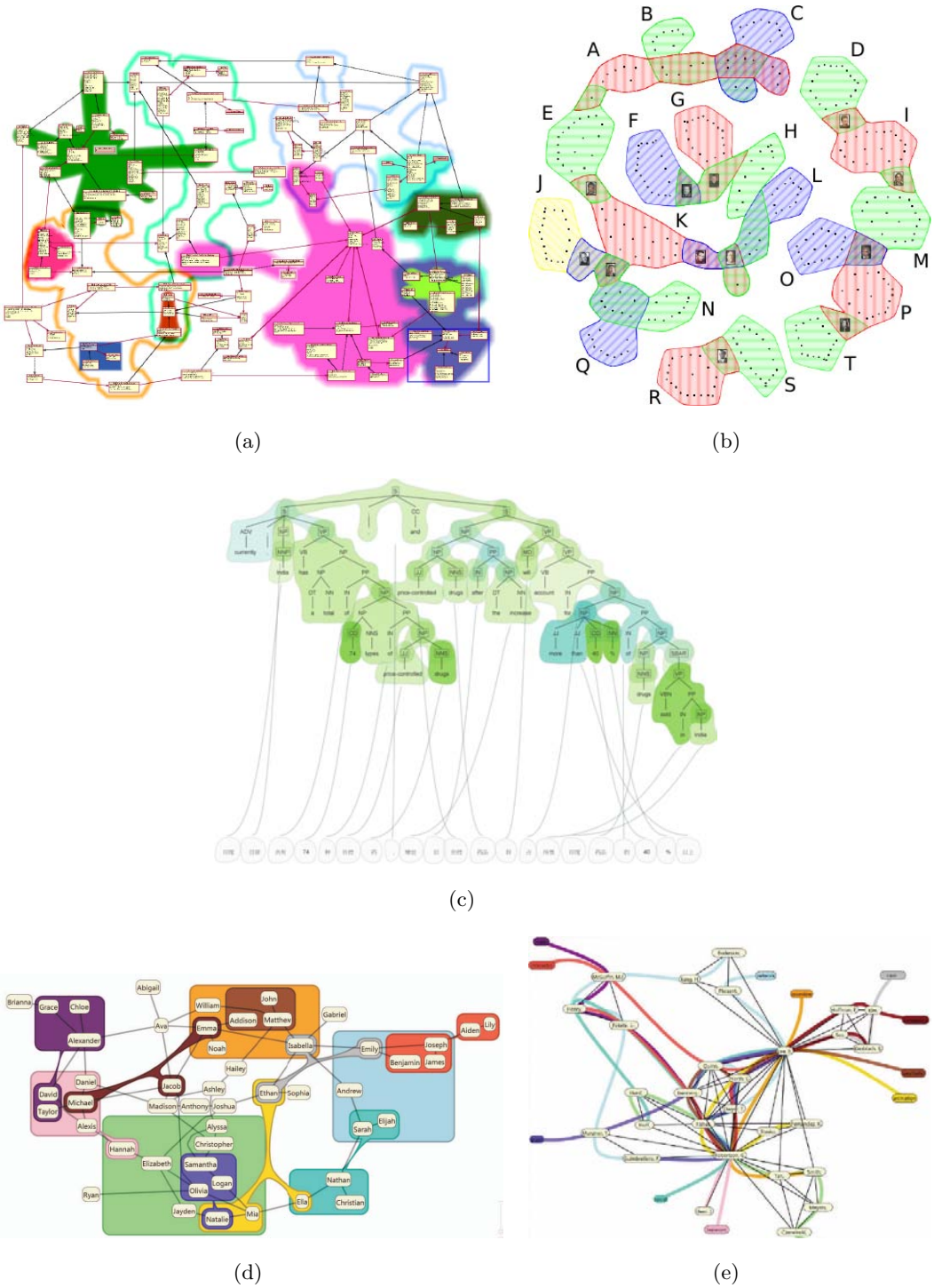
Dans la communauté de la Visualisation d'Information, des techniques de visualisation d'ensembles chevauchants ont également été proposées. Une méthode est d'organiser spatialement les éléments en fonction des ensembles auxquels ils appartiennent. Simonetto *et al.* [170] utilisent une telle approche pour générer des diagrammes topologiquement semblables à des diagrammes d'Euler (voir Figure 7.1(b)). Les ensembles sont ensuite mis en exergue via l'utilisation d'enveloppes possiblement concaves, colorées et texturées. Riche et Dwyer [152] génèrent également de tels diagrammes en utilisant une technique de dessin de graphe basée sur des contraintes (voir Figure 7.1(d)). Leur visualisation utilise

uniquement des formes rectangulaires convexes pour représenter les ensembles et améliore la lisibilité des intersections d'ensemble à travers des encodages visuels dédiés. Néanmoins, ces techniques ne peuvent pas être appliquées lorsque les éléments des ensembles ont une organisation spatiale sémantique, comme par exemple dans un dessin de graphe géolocalisé. Des méthodes ont également été proposées pour représenter des ensembles sur des visualisations existantes sans avoir à modifier la position des éléments. Byelas et Telea [32] furent parmi les premiers à matérialiser des sous-ensembles généraux via l'utilisation d'enveloppes convexes ou concaves. Dans ce travail, ils les utilisent pour visualiser des zones d'intérêt dans des diagrammes UML (voir Figure 7.1(a)). Ils génèrent ces enveloppes en calculant dans un premier temps un squelette à partir des positions et tailles des éléments d'un ensemble. Il utilisent ensuite une technique de rendu basée sur du *splatting* de texture [33] pour dessiner les enveloppes. Collins *et al.* [43] utilisent des iso-contours continus, possiblement concaves, pour matérialiser les ensembles (voir Figure 7.1(c)). Un autre type de métaphore visuelle est utilisée par Alper *et al.* [4] où ils génèrent une courbe connectant tous les éléments d'un ensemble (voir Figure 7.1(e)).

Concernant la visualisation de sous-ensembles d'un réseau dans une visualisation de graphe de type nœud-lien, l'approche classique est de les entourer avec des enveloppes convexes [95, 59]. Cependant cette méthode peut mener à des ambiguïtés car des éléments n'appartenant pas à un ensemble peuvent se retrouver inclus dans l'enveloppe convexe associée. Les techniques présentées dans [152] et [4] peuvent être utilisées mais elles se concentrent uniquement à représenter une relation secondaire sur les sommets d'un graphe. Par conséquent, elles ne sont pas adéquates pour visualiser l'intersection de sous-ensembles définis par des sous-graphes. A notre connaissance, seule la méthode présentée dans [43] permet d'effectuer cette tâche.

7.2 Utilisation d'enveloppes concaves

Dans cette section nous présentons deux techniques pour générer une enveloppe concave afin de mettre en exergue un sous-graphe d'intérêt dans une visualisation de graphes. Une enveloppe concave est un polygone non convexe, i.e. au moins un de ses angles internes est supérieur à 180°. De plus, le polygone peut comporter des trous. Les deux techniques consistent à calculer les coordonnées des différents contours constituant l'enveloppe concave à dessiner. Pour rendre une enveloppe concave à l'écran, il est nécessaire de la subdiviser en un ensemble de polygones convexes soit appliquer une opération de tesselation qui consiste à paver une surface avec le même type de primitive géométrique. Pour cela, nous utilisons les fonctionnalités de tesselation offertes par la bibliothèque GLU (*OpenGL Utility Library*) [41] permettant de découper un polygone quelconque en un



(a) (b)

(c)

(d) (e)

FIGURE 7.1: Exemples de techniques de visualisation d'ensembles chevauchants. (a) *Visualization of areas of interest in software architecture diagrams*, Byelas et Tela [32] – ©2006 ACM (b) *Fully Automatic Visualisation of Overlapping Sets*, Simonetto *et al.* [170] – ©2009 IEEE (c) *Bubble Sets*, Collins *et al.* [43] – ©2009 IEEE (d) *Untangling Euler Diagrams*, Riche et Dwyer [152] – ©2010 IEEE (e) *Line Sets*, Alper *et al.* [4] – ©2011 IEEE

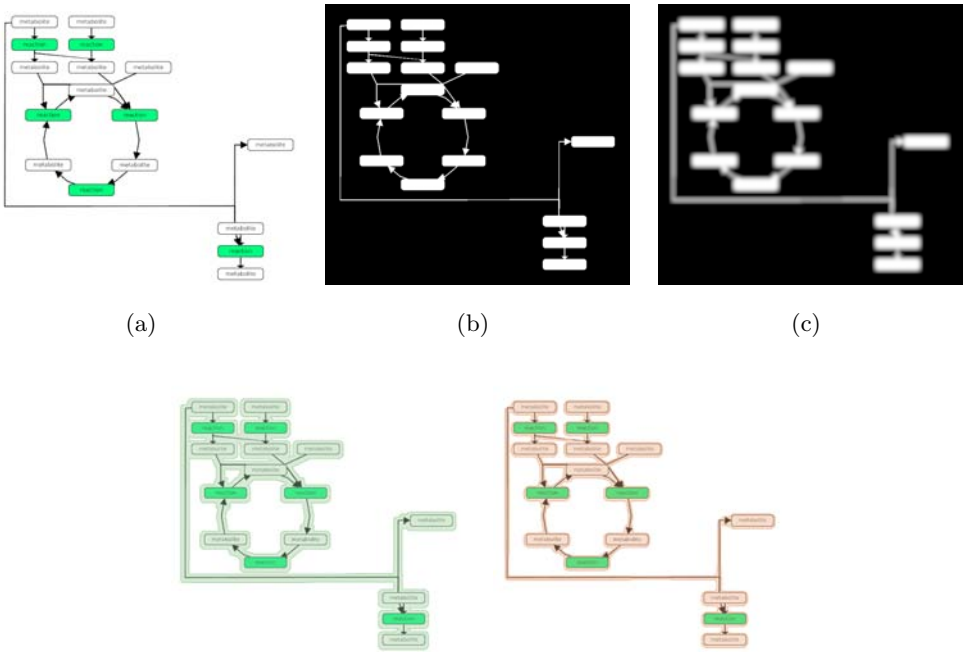
ensemble de triangles. La première technique travaille dans l'espace image d'une visualisation de graphe pour calculer l'enveloppe concave d'un sous-graphe d'intérêt. Même si cette première technique génère des enveloppes très esthétiques, elle est quelque peu onéreuse en terme de temps de calcul et n'est pas forcément efficace dans un contexte de visualisation de décomposition de graphe chevauchante. Nous avons alors élaboré une deuxième technique, travaillant dans l'espace topologique, calculant des enveloppes concaves de telle sorte que chaque sous-graphe d'intérêt soit parfaitement distinguable.

7.2.1 Génération d'enveloppes à partir de l'espace image

Cette première technique extrait des contours à partir d'une image afin de générer des enveloppes concaves pour mettre en exergue un sous-graphe. Elle est librement inspirée de celle présentée par Collins *et al.* [43].

Détail de la technique Les différentes étapes pour calculer les coordonnées des sommets définissant le polygone concave de l'enveloppe sont les suivantes. Dans un premier temps, chaque sous-graphe à entourer est rendu dans un tampon de trames hors-écran avec tous les éléments du graphe colorés en blanc (voir Figure 7.2(b)). Ensuite une convolution Gaussienne est effectuée sur l'image précédemment générée et le résultat est normalisé. De cette façon, nous obtenons un champ scalaire à partir duquel nous pouvons extraire des iso-surfaces (voir Figure 7.2(c)). Les valeurs des pixels de ce champ varient en fonction de leur distance au squelette du graphe, de 1.0 (quand la distance au squelette est de 0) à 0.0 (quand la distance au squelette est plus grande qu'un certain seuil). La taille et l'étendue des iso-surfaces pouvant être extraites varient en fonction du rayon et de l'écart type du noyau Gaussien. Les contours sont alors extraits en utilisant la version en deux dimensions du fameux algorithme *Marching Cubes* [131], appelée *Marching Squares*. Le fonctionnement détaillé de cet algorithme est présenté en Annexe C. La taille de l'enveloppe extraite dépend du choix de la valeur seuil définissant si un pixel se trouve ou non à l'intérieur d'un contour. Les Figures 7.2(d) et 7.2(e) présentent des enveloppes obtenues suivant différentes valeurs de ce seuil. La dernière étape consiste à projeter les coordonnées des sommets définissant l'enveloppe concave de l'espace image vers l'espace topologique du graphe. Le temps d'exécution de cette méthode dépend de la taille de l'image (champ scalaire) utilisée en entrée de l'algorithme *Marching Squares* ainsi que de la complexité du sous-graphe pour lequel générer une enveloppe.

Exemple d'application La Figure 7.3 montre un exemple d'application de cette technique de génération d'enveloppes concaves pour mettre en exergue des sous-graphes d'intérêt. Elle présente une vue détaillée du réseau métabolique de l'organisme *Saccharomyces*



(a) (b) (c)

(d) (e)

FIGURE 7.2: Illustration de la technique générant des enveloppes concaves depuis l'espace image. (a) Sous-graphe à mettre en exergue. (b) Rendu hors écran du sous-graphe avec tous les éléments du graphe colorés en blanc. (c) Champ scalaire normalisé obtenu après avoir convolué (b) avec un noyau Gaussien. (d) Enveloppe extraite avec un seuil de 0.1 en exécutant l'algorithme *Marching Squares* sur (c). (e) Enveloppe extraite avec un seuil de 0.5 en exécutant l'algorithme *Marching Squares* sur (c).

cerevisiae (levure). La représentation complète de ce réseau peut être trouvée à la Figure 4.12. Dans cette vue, un élément à été sélectionné : "PAPS" (en rouge), et les deux voies métaboliques auxquelles il appartient ont été mis en évidence avec notre technique. Sur cet exemple, on peut clairement identifier les deux voies ainsi que les éléments qu'elles partagent. Un autre exemple d'application de ce type a déjà été présenté à la Figure 4.15.

7.2.2 Génération d'enveloppes à partir de l'espace topologique

Même si la précédente technique permet de calculer des enveloppes très esthétiques, semblables à des enveloppes dessinées à main levée, elle ne passe pas très bien à l'échelle pour visualiser efficacement un grand nombre de groupes chevauchants. En effet, elle ne permet pas de moduler précisément la largeur d'une enveloppe et par conséquent il peut être difficile de distinguer les frontières de chaque ensemble. Pour pallier à ce problème, nous avons investigué une autre méthode qui calcule des enveloppes concaves à partir

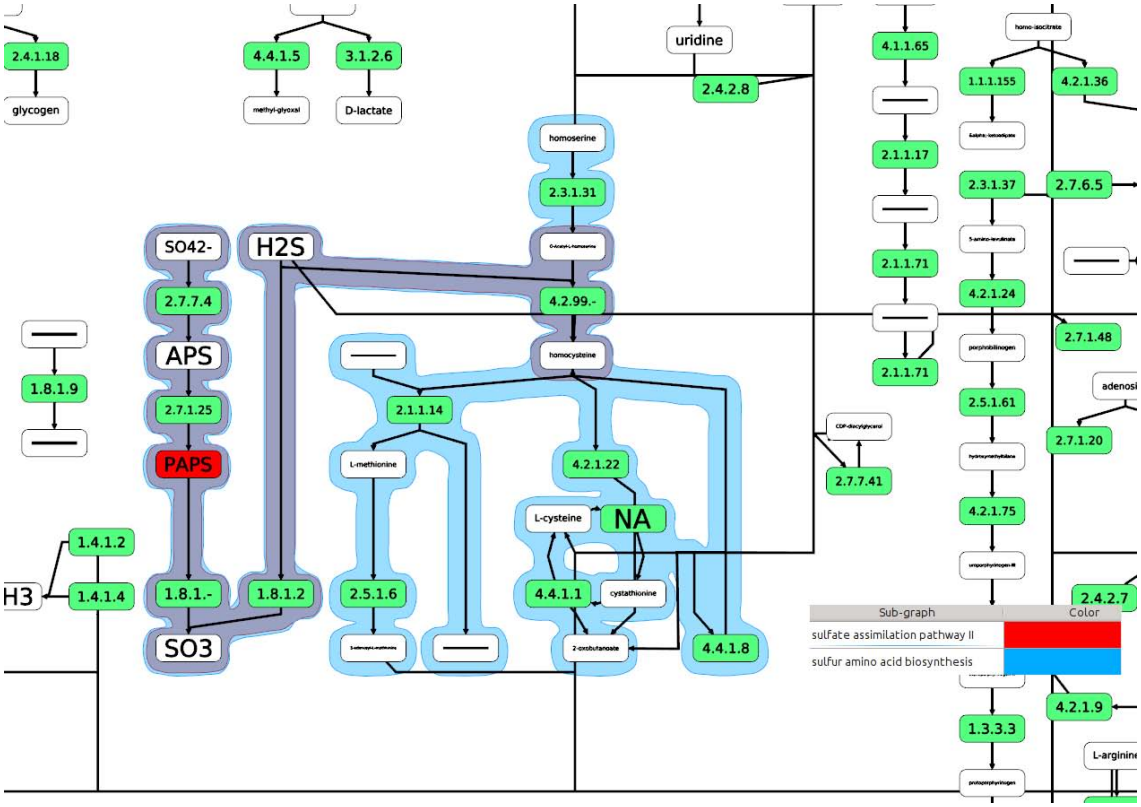


FIGURE 7.3: Exemple d'application de notre méthode de génération d'enveloppes concaves depuis l'espace image pour mettre en évidence des sous-graphes. Une vue détaillée du réseau métabolique du réseau métabolique de l'organisme *Saccharomyces cerevisiae* (levure) est présentée. Les voies métaboliques contenant l'élément "PAPS" ont été mises en évidence avec notre méthode.

de l'espace topologique du graphe à visualiser. Par espace topologique nous entendons l'espace continu de coordonnées dans lequel la représentation du graphe est plongée.

Détail de la technique La Figure 7.4 illustre les différentes étapes de notre méthode. Sur l'image la plus à gauche, on peut voir un dessin de réseau de co-occurrence des personnages du livre *Les Misérables* [117] écrit par Victor Hugo. Un ensemble de sous-graphes d'intérêt a été calculé sur ce réseau en utilisant l'algorithme *Link Communities* [3].

Dans un premier temps, un *graphe de dépendance* (voir Définition 2.29) est calculé afin de modéliser la façon dont les différents ensembles (sous-graphes) à visualiser se chevauchent. Ensuite un algorithme de coloration propre de graphe est appliqué sur ce graphe de dépendance pour assigner à chaque ensemble une valeur positive de telle sorte que deux ensembles se chevauchant aient des valeurs différentes. Cette valeur sera ensuite utilisée pour déterminer la distance dans le plan entre un sous-graphe et l'enveloppe l'entourant. Appliquer cette coloration sur le graphe de dépendance nous permet de garantir que les

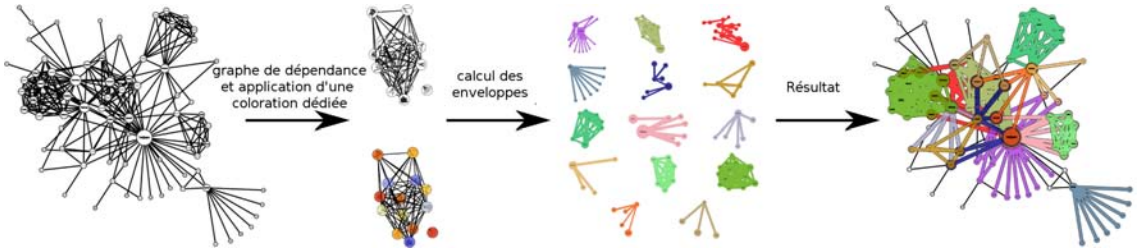
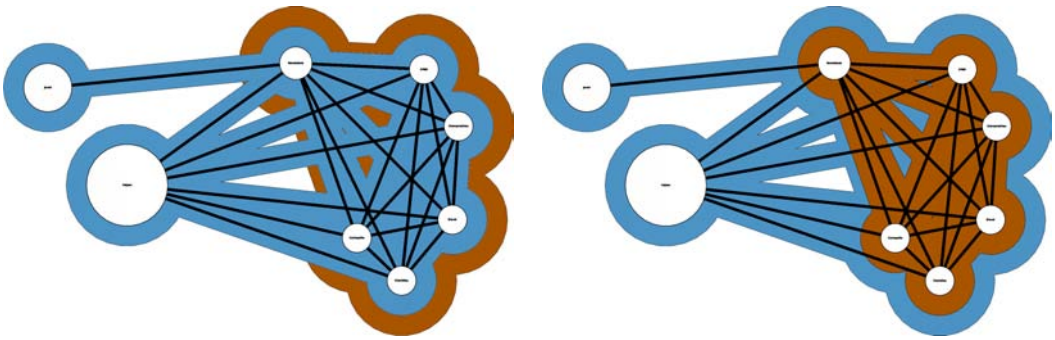


FIGURE 7.4: Illustration du pipeline de la méthode : à partir d'une décomposition de graphe chevauchante, un ensemble d'enveloppes distinguables les unes des autres est calculé.

enveloppes des différents ensembles chevauchants seront toutes distinguables en leur affectant une largeur proportionnelle à leur valeur de coloration et en les rendant dans l'ordre décroissant de ces valeurs.

N'importe quel algorithme de coloration propre pourrait être utilisé pour cette tâche. Cependant pour faciliter la visualisation d'ensembles imbriqués, nous avons besoin d'un algorithme de coloration dédié. Par exemple si l'on considère deux ensembles imbriqués, pour identifier facilement que l'un est contenu dans l'autre, son enveloppe doit être contenue dans l'enveloppe de l'autre ensemble (voir Figure 7.5).



(a) (b)

FIGURE 7.5: Illustration de la visualisation d'ensembles imbriqués. (a) La valeur de coloration assignée à l'ensemble contenu dans l'autre est plus grande que celle de l'autre ensemble. L'enveloppe de l'ensemble contenu dans l'autre est alors plus large que celle de l'autre ensemble, compliquant son identification. (b) Quand l'ensemble contenu dans l'autre a une valeur de coloration plus faible que celle de l'autre ensemble, l'imbrication des enveloppes permet de voir clairement la relation d'inclusion entre les deux.

Notre procédure de coloration va ainsi créer un ordre sur les différents ensembles de telle sorte que cet ordre reflète la complexité de chaque ensemble. Dans notre cas, nous utilisons le nombre de sommets pour évaluer la complexité d'un ensemble. Le détail de cette

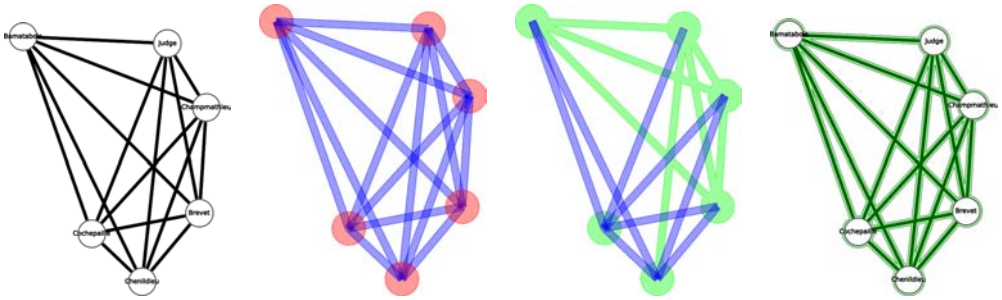
procédure est le suivant. À partir d'un graphe de dépendance $G = (V, E)$ et d'une fonction de valuation de sommets P (ici $P(u)$ est le nombre de sommets de l'ensemble associé au sommet u dans le graphe de dépendance), notre algorithme fournit une coloration C telle que $\forall (u, v) \in E$, si $P(u) < P(v)$ alors $C(u) < C(v)$. L'algorithme est basé sur un parcours de graphe en largeur, noté BFS pour *Breadth-first search*, et a donc une complexité en temps linéaire. Son détail est présenté à l'Algorithme 2

Algorithme 2 Algorithme de coloration dédiée appliqué sur le graphe de dépendance modélisant les chevauchements entre les sous-graphes à visualiser.

1: Phase d'initialisation : $\forall u \in V, C(u) = P(u)$.
2: Faire un BFS à partir d'un sommet non visité u. Lors du parcours, on insère dans la file des prochains sommets à visiter seulement les sommets v tels que $P(v) = P(u)$. Au cours de cette phase, on calcule et conserve les valeurs suivantes :
   - $\sigma(u)$ correspondant à la composante connexe (maximale sous inclusion) formée par u et tous les sommets $v \in V$ ayant $P(u)$ comme valuation.
   - $maxL$ (resp. $minG$) qui est la valeur maximum de C plus petite que $C(u)$ (resp. la valeur minimum de C plus grande que $C(u)$) dans le voisinage direct de $\sigma(u)$.
3: Assigner à tous les sommets de $\sigma(u)$ différentes valeurs dans l'intervalle $]maxL, minG[$ si $minG \neq maxL$, sinon dans l'intervalle $[1, |\sigma(u)|]$.
4: Marquer tous les sommets $v \in \sigma(u)$ comme visités.
5: Répéter l'étape 2 jusqu'à ce que tous les sommets aient été visités.

Une fois cette étape effectuée, nous pouvons alors calculer les enveloppes associées à chaque sous-graphe. Comme nous avons besoin de pouvoir moduler précisément leurs largeurs, notre solution pour les générer est basée sur du *clipping* de polygones et fonctionne dans l'espace topologique. L'idée est de calculer l'union de polygones construits à partir de la position des sommets et des arêtes du sous-graphe à mettre en exergue. Le polygone associé à un sommet peut être par exemple un cercle dont le centre est la position du sommet et le rayon est défini par la boite englobante du sommet ainsi que la largeur désirée de l'enveloppe. Le polygone associé à une arête consiste en l'extrusion de la ligne brisée la représentant paramétrée par la largeur désirée de l'enveloppe. Pour calculer l'union de tous ces polygones, nous utilisons la bibliothèque *Clipper* [108] : une implémentation efficace de l'algorithme de *clipping* de polygones mis au point par Vatti [185]. Des illustrations de ce processus de génération d'enveloppe sont présentées à la Figure 7.6.

Comparaison avec l'autre méthode de génération d'enveloppes La Figure 7.7 présente une comparaison des résultats obtenus en appliquant les deux méthodes de génération d'enveloppes concaves pour visualiser le résultat d'une décomposition chevauchante



(a) (b) (c) (d)

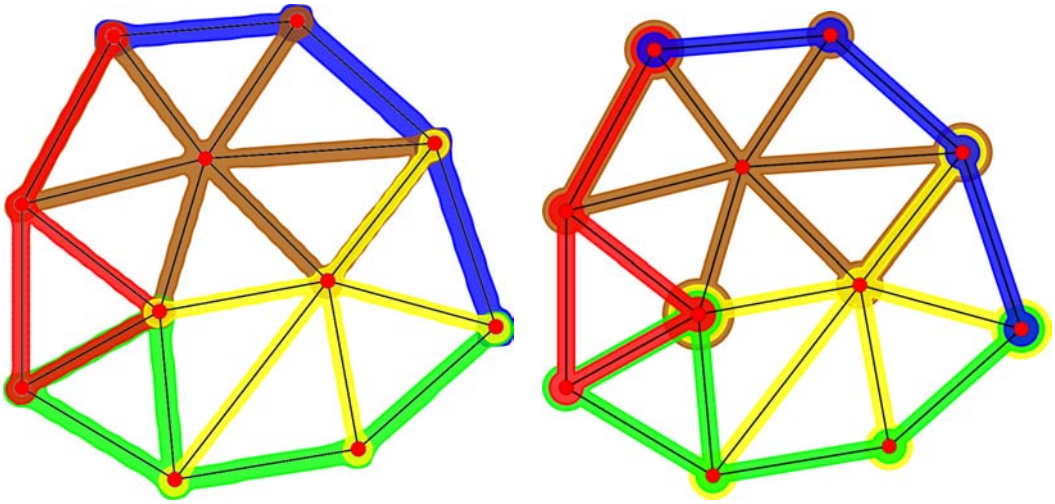
FIGURE 7.6: Illustration du processus générant une enveloppe concave pour entourer le dessin d'un sous-graphe. (a) Sous-graphe à mettre en exergue. (b) L'ensemble de polygones dont l'union doit être calculée. Les cercles rouges sont les polygones calculés à partir de la position des sommets, les quadrilatères bleus sont ceux calculés à partir de la position des arêtes. (c) Illustration d'une étape intermédiaire du processus d'union des polygones. Le polygone vert correspond à l'union déjà calculée. Les polygones bleus sont ceux restants à traiter. (d) Enveloppe concave résultante.

sur un graphe d'exemple. Dans la Figure 7.7(a), les enveloppes ont été calculées en utilisant la méthode travaillant dans l'espace image de la visualisation (voir section 7.2.1). Dans la Figure 7.7(b), les enveloppes ont été générées avec la méthode travaillant dans l'espace topologique introduite dans la section courante. On observe qu'avec la première méthode, il est difficile d'identifier précisément chaque sous-graphe de la décomposition. Le problème vient du fait que certaines frontières d'enveloppes sont parfois confondues. Ce problème n'apparaît pas avec la seconde méthode car la largeur de chaque enveloppe a pu être modulée précisément afin qu'elles soient toutes clairement distinguables les unes des autres.

Exemples d'application Nous présentons ici deux exemples d'application de cette méthode de génération d'enveloppes concaves pour la visualisation d'une décomposition chevauchante de graphe.

Le premier exemple consiste en la visualisation d'une décomposition chevauchante du réseau de co-occurence *Les Misérables* [117]. Les sommets de ce graphe représente l'ensemble des personnages de l'œuvre de Victor Hugo. Une arête relie deux personnages si ils apparaissent dans le même chapitre du livre. La décomposition a été calculée à l'aide de l'algorithme *Link Communities* [3]. Cet algorithme ne produit pas uniquement des sous-graphes hautement connectés comme le font beaucoup d'algorithmes de fragmentation de graphe. Ces sous-graphes peuvent être classés en trois catégories :

- **Sous-graphe hautement connecté** : sous-ensemble de sommets avec un grand



(a) (b)

FIGURE 7.7: Comparaison entre les deux techniques de génération d'enveloppes concaves pour la visualisation d'une décomposition de graphe chevauchante : (a) enveloppes générées depuis l'espace image, (b) enveloppes générées depuis l'espace topologique. On observe que dans (a), il est difficile d'identifier précisément chaque sous-graphe, les frontières de certaines enveloppes étant parfois confondues. Ce problème n'apparaît pas dans (b) car la largeur de chaque enveloppe a pu être précisément modulée afin qu'elles soient toutes clairement distinguables.

nombre de connexions entre eux.

- **Sous-graphe biparti** : sous-ensemble de sommets reliant des sous-graphes hautement connectés.
- **Sous-graphe arborescent :** sous-ensemble de sommets formant un arbre, pouvant également relier des sous-graphes hautement connectés.

Avec notre méthode, nous pouvons visualiser cette classification en assignant à chaque enveloppe une couleur associée à la catégorie du sous-graphe qu'elle entoure. Le résultat est présenté à la Figure 7.8. A l'échelle globale de la visualisation (voir Figure 7.8(a)), on peut observer de petits sous-graphes qui s'intersecte avec de plus grands. Les différents sous-graphes hautement connectés (en bleu) ou les sous-graphes arborescents (en vert) sont clairement identifiables. On peut voir par exemple que chaque sommet appartient au plus à un sous-graphe hautement connecté. Le centre du dessin dans la Figure 7.8(a) est plus complexe à visualiser. Il contient plusieurs sommets de fort degré appartenant à des sous-graphes bipartis (en rouge). Une vue détaillée de cette zonz est présentée à la Figure 7.8(b). Rappelons que les sous-graphes partageant des sommets sont représentés avec des enveloppes de largeur différente. Les différentes catégories de sous-graphes auquel appartient un sommet sont donc clairement identifiables. Par exemple on peut observer que

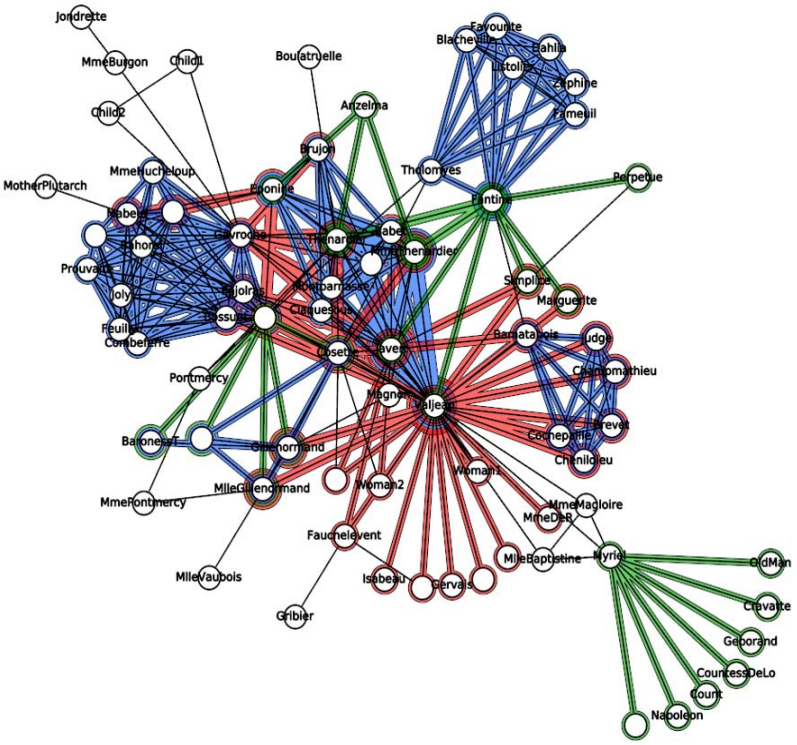
les sommets "Valjean" et "Javert" appartiennent chacun à quatre sous-graphes bipartis : ils jouent donc un important rôle de médiateur dans le réseau. On peut également voir que "MmeThenardier" est contenu dans les mêmes groupes que "Thenardier", son mari dans le livre.

Le second exemple, présenté à la Figure 7.9, montre le réseau métabolique de l'organisme *Saccharomyces cerevisiae* (levure) dessiné avec notre méthode [124] détaillée dans la section 4.3. L'ensemble des voies métaboliques de ce réseau ont été mis en exergue avec notre méthode. Ce réseau contenait à la base 836 sommets et 936 arêtes répartis sur 164 voies métaboliques. Les éléments appartenant à plus de trois voies métaboliques ont été dupliqués, résultant en un réseau de 1360 sommets et 1340 arêtes. L'intérêt de notre méthode de visualisation de sous-graphes sur ce réseau est qu'un grand nombre de métabolites/réactions ainsi que les arêtes les connectant sont partagés par plusieurs voies. On peut observer dans la vue détaillée de la Figure 7.9 que chaque voie et leurs éléments communs peuvent être clairement identifiés.

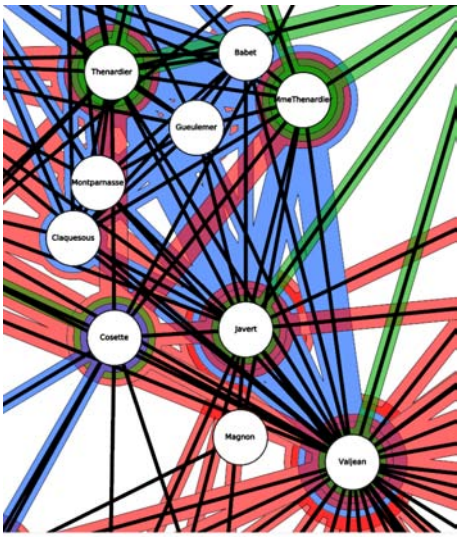
7.3 Utilisation d'une déformation 3D

Mettre en exergue des sous-graphes au moyen d'enveloppes concaves aide l'utilisateur à les identifier dans la représentation globale du réseau. Cependant, quelques ambiguïtés peuvent apparaître. Par exemple quand un sous-graphe est dense et que son dessin implique un grand nombre de croisements d'arêtes, il peut être difficile de déterminer si un sommet se trouvant dans son enveloppe appartient au sous-graphe ou non. Pour pallier à ce type de problème, nous présentons une méthode basée sur une déformation 3D visant à mettre clairement en évidence un sous-graphe dans un contexte de visualisation globale d'un réseau.

Détail de la méthode Notre méthode est proche de la technique *Graph Folding* de Carpendale *et al.* [36] à la différence que la visualisation est déformée à l'échelle locale du sous-graphe à mettre en évidence. Le principe est de modifier la coordonnée z des éléments du graphe à rendre et d'utiliser une projection en perspective pour visualiser l'effet de déformation. La zone du dessin du graphe sur laquelle la déformation sera appliquée est déterminée par le squelette du sous-graphe ainsi que le rayon de la déformation. Le squelette d'un sous-graphe est défini par un ensemble de points et de segments. Les points correspondent aux positions des sommets et les segments correspondent aux dessins des arêtes. Pour chaque sommet des primitives géométriques utilisées pour représenter les éléments du graphe entier, sa distance au squelette du sous-graphe à mettre en évidence est calculée. Le calcul de cette distance est effectuée de la façon suivante. Pour chaque



(a)



(b)

FIGURE 7.8: Application de notre méthode de mise en évidence de sous-graphes par enveloppes concaves sur le réseau de co-occurrence *Les Misérables* [117]. Le résultat de la décomposition chevauchante calculée par l'algorithme [3] est visualisé. Les différents sous-graphes résultants sont divisés en trois catégories : sous-graphe hautement connecté (en bleu), sous-graphe arborescent (en vert) et sous-graphe biparti (en rouge). (a) Vue globale. (b) Vue détaillée sur le centre du dessin.

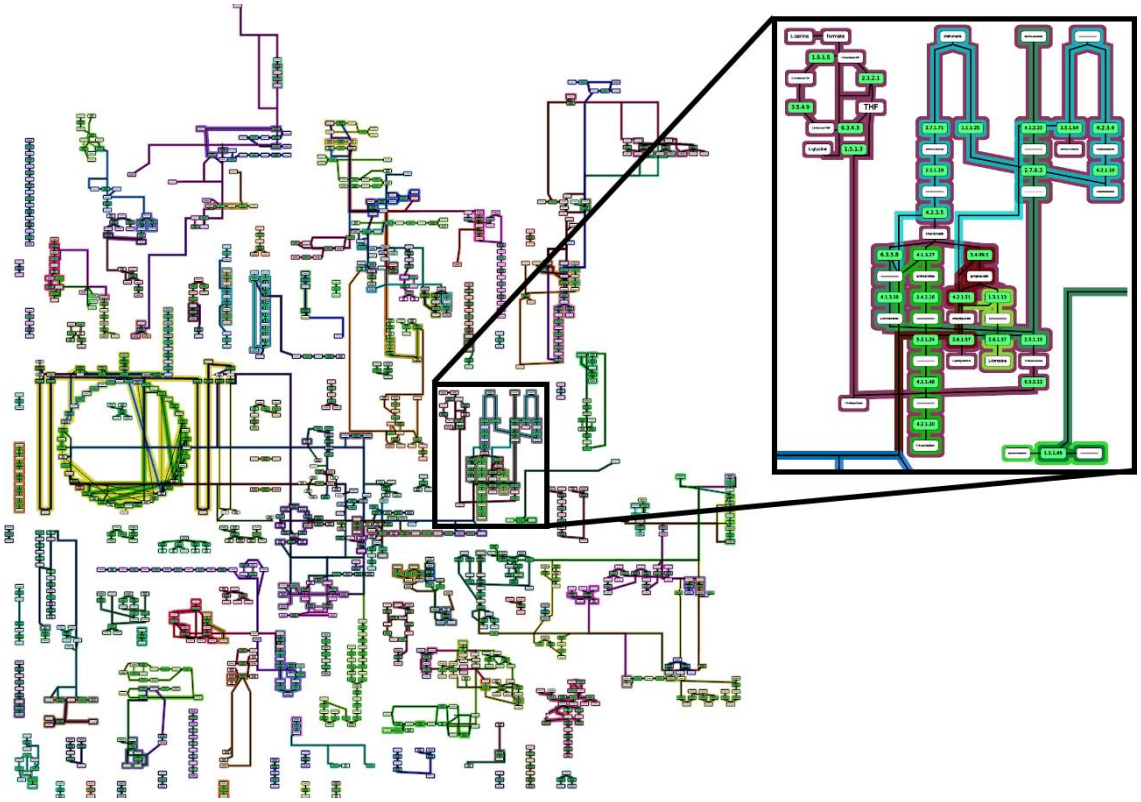


FIGURE 7.9: Représentation du réseau métabolique de l'organisme *Saccharomyces cerevisiae* (levure) où chacune des 164 voies métaboliques ont été mis en exergue avec notre méthode. Dans la vue détaillée, on peut clairement identifier chaque voie ainsi que leurs éléments communs.

point du squelette du sous-graphe, la distance entre le sommet de la primitive géométrique et ce point est déterminée. Pour chaque segment du squelette, la projection orthogonale du sommet de la primitive géométrique sur le segment est calculée et si elle se trouve sur le segment la distance entre cette projection et la position originale est déterminée. Parmi toutes les distances déterminées, celle avec la valeur minimum est retenue. Si cette distance est inférieure au rayon de la déformation, la coordonnée z du sommet est alors modifiée en fonction de cette distance via l'utilisation d'une fonction *drop-off*. Dans notre cas, nous utilisons une simple fonction *drop-off* hémisphérique $f(x) = 1 - x^2$, $0 \leq x \leq 1$ avec $x = \dfrac{distance}{rayon\ déformation}$ de telle sorte que le sous-graphe à mettre en évidence soit plaqué sur une surface cylindrique. D'autres types de fonctions *drop-off* peuvent être employées comme expliqué par Carpendale *et al.* dans [37]. Le pseudo-code de cet algorithme de déformation très simple est présenté à l'Algorithme 3. Nous l'avons implémenté au moyen d'un *vertex shader* OpenGL appliquant la déformation en temps réel lorsque le dessin du graphe est rendu à l'écran. De plus, afin d'aider l'utilisateur à percevoir clairement le sous-graphe mis en évidence, nous rendons également une surface illuminée, sous

la forme d'une grille de triangles de résolution fine, obéissant à la même déformation. Des illustrations de cette technique sur un graphe d'exemple sont présentés à la Figure 7.10.

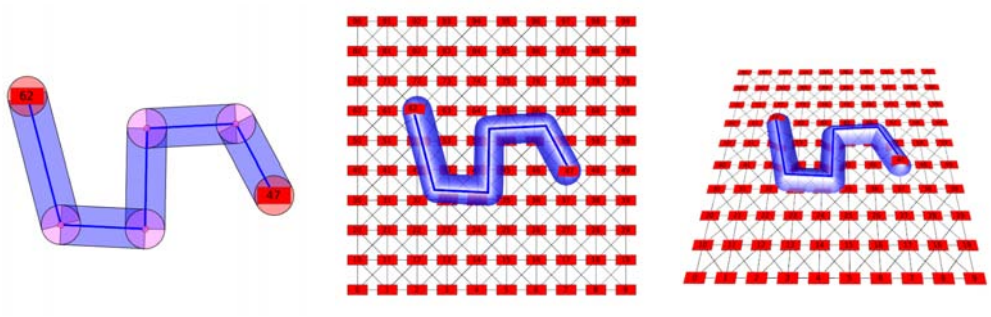
Algorithme 3 Algorithme appliquant une déformation 3d sur les sommets définissant les primitives géomètriques utilisées pour représenter les élèments d'un graphe.

Entrées:
- v : un sommet d'une primitive géomètrique
- $tabPoints$: un tableau de points
- $tabSegments$: un tableau de segments
- r : le rayon de la déformation

Sorties: le sommet v modifié suivant la déformation

```
1: maxMouvement = 0.0
2: Pour i de 0 à TAILLE(tabSegments) faire
3:     centre = PROJECTIONORTHOGONALE(v, tabSegments[i])
4:     Si centre est sur tabSegments[i] alors
5:         dist = DISTANCE(centre, v)
6:         Si dist < r alors
7:             normMouvement = 1.0 − (dist/r)^2
8:             Si normMouvement > maxMouvement alors
9:                 maxMouvement = normMouvement
10:            Fin Si
11:        Fin Si
12:    Fin Si
13: Fin Pour
14: Pour i de 0 à TAILLE(tabPoints) faire
15:    dist = DISTANCE(tabPoints[i], v)
16:    Si dist < r alors
17:        normMouvement = 1.0 − (dist/r)^2
18:        Si normMouvement > maxMouvement alors
19:            maxMouvement = normMouvement
20:        Fin Si
21:    Fin Si
22: Fin Pour
23: nouveauV = v
24: nouveauV.z = maxMouvement * r
25: Retourner nouveauV
```

Exemple d'application La Figure 7.11 présente un exemple d'application de cette méthode de mise en évidence de sous-graphe. Le graphe visualisé est un réseau de joueurs

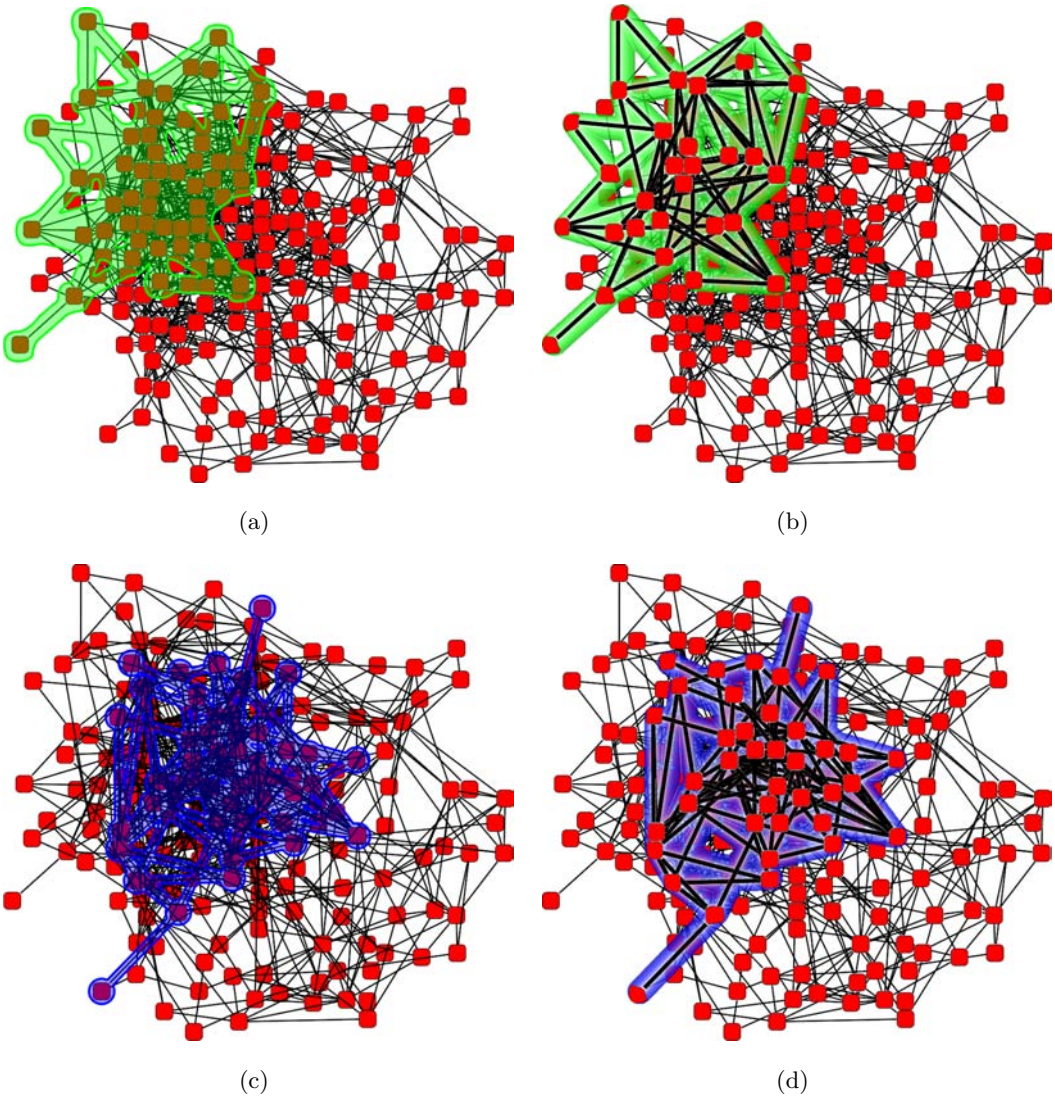


(a) (b) (c)

FIGURE 7.10: Illustrations de notre méthode basée sur une déformation 3D pour mettre en exergue un sous-graphe dans une visualisation de graphe. (a) Un sous-graphe simple extrait d'un réseau contenant deux sommets et une arête. La zone sur laquelle la déformation sera appliquée est représentée. (b) Résultat de notre technique pour visualiser le sous-graphe introduit en (a) dans le contexte du réseau global. (c) Résultat après avoir effectué une rotation de la visualisation le long de l'axe x.

de poker en ligne. Chaque sommet représente un joueur et une arête relie deux joueurs si l'un deux a payé l'autre au cours d'une partie. L'algorithme de fragmentation de graphe *Louvain* [23] a été appliqué sur ce réseau. La Figure 7.11 compare les méthode de mise en évidence par enveloppes concaves et la méthode par déformation 3D pour visualiser dans le contexte global les communautés détectées par l'algorithme [23]. Dans ce cas précis, les méthode par enveloppes concaves ne sont pas efficaces en raison du grand nombre d'arêtes et leurs croisements induisant beaucoup d'occlusions dans le centre de la représentation. Il est très difficile d'identifier précisément les groupes et leur topologie. Ce problème n'apparaît plus en utilisant la méthode de mise en exergue par déformation 3D.

Un autre exemple d'application de cette méthode pour mettre en exergue une voie métabolique d'intérêt dans une représentation de réseau métabolique a déjà été présentée à la Figure 4.16.



(a) (b)

(c) (d)

FIGURE 7.11: Exemple d'application de notre méthode de mise en exergue d'un sous-graphe par déformation 3D et comparaison avec les méthodes de mise en évidence par enveloppes concaves. Le graphe visualisé est un réseau de joueurs de poker où l'on veut voir le résultat de l'algorithme de fragmentation *Louvain* [23]. (a) Sous-graphe mis en exergue par une enveloppe concave générée depuis l'espace image (voir section 7.2.1). (b) Le même sous-graphe que (a) mis en évidence par déformation 3D. (c) Sous-graphe mis en exergue par une enveloppe concave générée depuis l'espace topologique (voir section 7.2.2). (d) Le même sous-graphe que (c) mis en évidence par déformation 3D.

Chapitre 8
Conclusion

Dans ce chapitre final, nous dressons une conclusion sur les travaux effectués au cours de cette thèse, les résultats obtenus ainsi que les futures directions de recherche en découlant.

8.1 Objectifs et réalisations

Les travaux de cette thèse sont inscris dans le domaine de la visualisation interactive de graphes. Ils peuvent être découpés en deux catégories. Le premier ensemble de travaux porte sur le domaine du dessin de graphe. Le second ensemble présente des méthodes d'infographie pour optimiser et améliorer des visualisations de graphes. Un grand nombre des travaux de cette dernière catégorie repose sur l'exploitation du processeur graphique comme accélérateur de calculs. Dans cette section, nous résumons les problèmes qui ont motivé nos travaux de recherche ainsi que les solutions que nous avons élaborées. Pour chaque contribution, nous donnerons la liste des publications associées ainsi que les sections de cette thèse détaillant le sujet.

8.1.1 Dessin de graphe

Nous résumons ici les travaux de dessin de graphe effectués au cours de cette thèse. Nos recherches se sont dans un premier temps concentrées sur le problème de la réduction d'occlusions visuelles dans les dessins de grands graphes. Elles ont débouché sur l'élaboration d'un algorithme de regroupement d'arêtes en faisceaux pour les dessins de graphe dans le plan et dans l'espace. Dans un second temps, nous nous sommes intéressés à la représentation automatique de réseaux biologiques complexes que sont les réseaux métaboliques. Ce travail présente d'ailleurs un cas d'application concret de notre méthode de regroupement d'arêtes.

8.1.1.1 Réduction des problèmes d'occlusion par regroupement d'arêtes

Dans un premier temps, nous nous somme intéressés à un problème très en vogue depuis quelques années : la réduction de l'occlusion due au dessin des arêtes en ligne droite dans les représentations de graphe de type *nœud-lien*. C'est un problème majeur qui s'est amplifié au cours des années en raison de la taille des graphes à visualiser qui ne cesse de s'accroître. Les visualisations de grands graphes souffrent généralement de problèmes d'occlusion et de lisibilité en raison du dessin des arêtes. Ces dernières sont usuellement représentées comme des segments ce qui entraîne de nombreuses occlusions visuelles lorsque leur nombre est très élevé. Ces occlusions peuvent nuire à la bonne lisibilité du dessin voire le rendre totalement inexploitable. En particulier, il arrive que de nombreux sommets soient totalement recouverts par le dessin d'un grand nombre d'arêtes. Il peut donc être difficile d'extraire des informations sur la structure générale du graphe visualisé.

Pour palier à ce problème, nous avons mis au point une méthode de regroupement d'arêtes en faisceaux (en anglais, *edge bundling*) pour les dessins de graphe dans le plan nommée *Winding Roads*. Ce type de méthode s'attache à regrouper ensemble des arêtes afin de réduire le nombre global de croisements et donc la lisibilité du dessin. L'autre bénéfice est qu'elles permettent de faire émerger les grands flux d'échanges entre différentes régions du dessin de graphe. Regrouper des arêtes consiste à modifier leur forme via le calcul de points de contrôles dans l'espace du dessin. Les arêtes partageant des points de contrôle successifs sont ainsi regroupées sur des portions de leur route. La méthode que nous proposons a les propriétés suivantes :

- intuitive dans son fonctionnement
- flexible quant au niveau de réduction d'occlusions
- facilement implémentable

Le détail de la méthode est présentée dans la section 3.3.1 de ce manuscrit. La méthode a été publiée sous la référence suivante :

> [123] Antoine Lambert, Romain Bourqui, and David Auber. Winding Roads: Routing edges into bundles. *Computer Graphics Forum*, 29(3):853–862, 2010. Proceedings of the Joint Eurographics/IEEE-VGTC Symposium on Visualization (EuroVis 2010)

Nous avons également mis au point une généralisation de cette méthode pour regrouper des arêtes dans un dessin de graphe en trois dimensions. Une version spécialement dédiée à regrouper des arêtes sur des dessins de graphe sphérique (typiquement des lieux et leurs connexions placés sur un globe terrestre) a aussi été élaborée. La finalité de cette version dédiée est de router et regrouper les arêtes uniquement sur la surface de la sphère.

Le détail de cette généralisation en trois dimensions de notre méthode *Winding Roads* [123] est présenté dans la section 3.3. La méthode a été publiée sous la référence suivante :

> [122] Antoine Lambert, Romain Bourqui, and David Auber. 3D Edge Bundling for Geographical Data Visualization. In *Proceedings of the 14th International Conference Information Visualisation*, IV'10, pages 329–335. IEEE Computer Society, 2010

8.1.1.2 Représentation de réseaux métaboliques

Notre second travail en matière de dessin de graphe porte sur la représentation automatique de réseaux métaboliques. Ce type de réseaux biologiques modélise l'ensemble des réactions biochimiques se produisant au sein des cellules d'un organisme. Ces réactions transforment un ensemble de molécules en un autre ensemble de molécules. Un réseau métabolique peut être décomposé en voies métaboliques, sous ensembles de réactions établis par les biologistes effectuant une fonction particulière. Ces voies peuvent se chevaucher, i.e. qu'elles peuvent partager des molécules/réactions. Les améliorations dans l'acquisition de données biologiques et de séquençage de génomes permettent de nos jours de reconstruire des réseaux métaboliques complets de nombreux organismes vivants. Pouvoir les visualiser efficacement est un vrai besoin pour l'analyse du métabolisme. La taille et la complexité de ces réseaux ne facilitent pas leur représentation et nécessite l'élaboration de techniques de visualisation dédiées. En particulier, un certain nombre de contrainte doivent être prises en compte pour dessiner de tels réseaux. Elles sont basées sur des observations faites sur des représentations manuelles et sont énoncées ci-dessous :

- L'information relative aux voies métaboliques doit être préservée autant que possible.
- Éviter autant que possible de dupliquer des sommets afin de respecter la connectivité du réseau.
- Les cycles et cascades de réactions doivent être représentés suivant les conventions de dessin biologique, soit en cercle et en ligne droite.
- Les arêtes doivent être dessinées de manière pseudo-orthogonales.
- Le nombre de croisements d'arêtes doit être minimisé.

Nous proposons une méthode de dessin qui essaie de respecter simultanément toutes les contraintes précédemment exprimées. En particulier, notre méthode s'attache à essayer de préserver le plus possible l'information relative aux voies métaboliques. Les représentations obtenues sont compactes, préservent la topologie du réseau et contiennent peu d'occlusions visuelles.

Le détail de cette méthode de dessin de réseaux métabolique préservant l'information relative aux voies métaboliques est présenté dans la section 4.3 de ce manuscrit. La méthode a été publiée sous la référence suivante :

[124] Antoine Lambert, Jonathan Dubois, and Romain Bourqui. Pathway Preserving Representation of Metabolic Networks. *Computer Graphics Forum*, 30(3):1021–1030, 2011. Proceedings of the Joint Eurographics/IEEE-VGTC Symposium on Visualization (EuroVis 2011)

8.1.2 Infographie pour la visualisation interactive de graphes

Les autres contributions de cette thèse portent sur des techniques d'infographie élaborées dans un contexte de visualisation de graphe. La majorité d'entre elles s'attachent à exploiter le processeur graphique pour optimiser le temps de calcul d'algorithmes coûteux. La première contribution dans cette thématique est une technique optimisée de rendu de courbes paramétriques. Cette méthode a été développée afin de mettre en place des interactions fluides sur des visualisations de graphe avec regroupement d'arêtes. La seconde contribution est une technique de rendu permettant de visualiser la densité des faisceaux d'arêtes dans un dessin de graphe avec regroupement d'arêtes. La dernière contribution porte sur des techniques de mises en évidence de sous-graphes d'intérêt dans le contexte global d'une visualisation de graphe.

8.1.2.1 Optimisation du temps de rendu de courbes paramétriques

Après avoir appliqué un algorithme de regroupement d'arêtes sur un dessin de graphe, une arête n'est plus représentée comme un segment mais comme une ligne brisée. Afin de lisser la forme des arêtes et de proposer une visualisation plus esthétique, il est courant de dessiner les arêtes en utilisant des courbes paramétriques. Les courbes paramétriques sont définies par un ensemble de points de contrôle et analytiquement décrites par un polynôme. Il en existe plusieurs types. Nous nous sommes intéressés aux types de courbes suivant :

- courbes de Bézier
- courbes B-Spline
- courbes de Catmull-Rom

L'un des problèmes majeurs est que le coût de calcul des points interpolant une courbe le long de son tracé est important et augmente en fonction du nombre de points de contrôle. Il est bien entendu possible de précalculer tous les points interpolant les différentes courbes à rendre et de les mettre en cache. Si cette technique est satisfaisante pour les dessins statiques, elle est en revanche très peu efficace lorsque l'on veut mettre en place des interactions fluides avec la visualisation de graphe. Un autre inconvénient lorsque l'on précalcule tous les points interpolant les différentes courbes à rendre est l'occupation mémoire importante qui en découle. En effet, le nombre de points interpolant une courbe

est généralement bien plus élevé que le nombre de points de contrôle. Si ce problème peut paraître anecdotique sur des stations de travail contemporaines, il l'est beaucoup moins sur des systèmes embarqués ou mobiles qui dispose de beaucoup moins de mémoire. Nous avions donc besoin d'une approche de rendu de courbes efficace afin de proposer des techniques d'interaction fluides sur une visualisation de graphe avec regroupement d'arêtes.

Nous proposons une solution qui repose sur l'utilisation du processeur graphique pour le calcul des points interpolant les courbes. Le principe est de transférer les points de contrôle des courbes dans la mémoire vidéo du processeur graphique et de laisser ce dernier évaluer le polynôme associé au type de courbe à rendre.

Les résultats que nous avons obtenu montrent que notre technique est très efficace et permet de rendre un très grand nombre de courbes définies par un nombre arbitraire de points de contrôle tout en conservant un très bon taux de rafraîchissement d'images (de l'ordre de 25 images par seconde). L'autre avantage de cette technique est qu'elle permet d'économiser beaucoup de mémoire vive car seuls les points de contrôle et quelques propriétés visuelles (couleurs, tailles, ...) sont nécessaires pour dessiner une courbe. Enfin, si cette technique a été développée dans un contexte de visualisation de graphe, elle reste générique et peut donc être appliquée sur tout type de visualisation utilisant des courbes paramétriques (e.g. coordonnées parallèles).

Le détail de ces techniques de rendu de courbes paramétriques exploitant le processeur graphique est présenté dans la section 5.2 de ce manuscrit. Les implémentations en GLSL des différents *vertex shader* pour rendre les trois types de courbes sont données en Annexe B. Les prémisses de ces techniques ont été publiées sous la référence suivante :

> [121] Antoine Lambert, David Auber, and Guy Melançon. Living Flows: Enhanced Exploration of Edge-Bundled Graphs Based on GPU-Intensive Edge Rendering. In *Information Visualisation (IV), 2010 14th International Conference*, pages 523–530. IEEE Computer Society, 2010

8.1.2.2 *Edge splatting* : une technique pour visualiser la densité des faisceaux d'arêtes

Après avoir appliqué un algorithme de regroupement d'arêtes sur un dessin de graphe, des faisceaux d'arêtes sont créés. Ils donnent alors une belle impression de flux d'échanges entre différentes régions du dessin. Cependant, l'information relative à la densité de ces faisceaux n'est pas facilement visible dans le dessin. Il est ainsi difficile d'identifier les faisceaux contenant beaucoup d'arêtes de ceux en contenant peu.

Nous proposons une méthode de rendu nommée *edge splatting* permettant de visualiser efficacement la densité des faisceaux d'arêtes. Notre méthode combine des techniques classiques de traitement d'image et de rendu graphique. Chaque étape de la méthode est exécutée sur le processeur graphique. Le principe est de calculer à partir du dessin de graphe un *splat field*, soit un champ scalaire encodant les variations continues dans la densité des arêtes regroupées. Ce *splat field* peut ensuite être rendu en utilisant différents encodages visuels. Nous en avons expérimenté deux. Le premier associe simplement les valeurs de densité à des couleurs en fonction d'une échelle définie par l'utilisateur. Le second utilise une technique d'illumination par pixel, nommée *bump mapping*, associant les valeurs de densité à des hauteurs. Les faisceaux d'arêtes denses apparaissent ainsi plus haut que ceux contenant peu d'arêtes. L'avantage de cet encodage visuel par rapport au premier est qu'il permet de conserver les couleurs originales des arêtes et ainsi de préserver l'information qu'elles encodent.

Le détail de l'implémentation de la méthode est présenté dans la section 6 de ce manuscrit. Cette méthode de rendu a été publiée dans l'article présentant notre méthode de regroupement d'arêtes *Winding Roads* [123].

8.1.2.3 Visualisation de sous-ensembles d'intérêt d'un réseau dans un contexte global

L'analyse d'un graphe repose souvent sur une décomposition de ce dernier en sous-graphes d'intérêt. Des exemples connus sont la détection de communautés au sein d'un réseau social ou encore la décomposition en voies métaboliques d'un réseau métabolique. Dans de nombreux cas, cette décomposition est chevauchante, i.e. des sommets/arêtes peuvent être partagés entre plusieurs sous-graphes. Pouvoir visualiser efficacement le résultat d'une décomposition de graphe est donc primordial pour faciliter le processus d'analyse.

Nous nous sommes intéressés à la mise en exergue de sous-graphes dans le contexte global d'une visualisation de graphe. La motivation de ce travail était de pouvoir retrouver l'information relative aux voies métaboliques dans les représentations de réseaux métaboliques que nous produisons avec la méthode de dessin présentée à la section 4.3. Nous avons élaboré deux méthodes pour cette tâche. La première est basée sur l'utilisation d'enveloppes concaves pour entourer les sous-graphes d'intérêt. La seconde utilise une déformation 3d pour mettre en évidence un sous-graphe particulier.

Pour générer des enveloppes concaves (polygones non convexes pouvant avoir des trous) entourant le dessin de sous-graphes, nous avons mis au point deux techniques. La première

calcule les enveloppes à partir de l'espace image de la visualisation. Le détail de l'implémentation de cette technique est présenté à la section 7.2.1 de ce manuscrit. Elle a été publiée dans notre article [124] en complément de notre méthode de dessin de réseaux métaboliques. Cette technique génère des enveloppes très esthétiques mais manque de flexibilité quant à la précision des enveloppes générées. En particulier, il est difficile de moduler finement la largeur d'une enveloppe avec cette technique. Par conséquent, cette méthode ne passe pas très bien à l'échelle pour visualiser efficacement un grand nombre de sous-graphes chevauchants. Pour palier à ce problème, nous avons élaboré une seconde technique qui calcule des enveloppes à partir de l'espace topologique (soit le plan dans lequel est plongé le dessin du graphe). Cette technique a l'avantage de garantir que toutes les enveloppes calculées sont distinguables, propriété primordiale dans le cas où l'on veut visualiser le résultat d'une décomposition chevauchante de graphe. Le détail de cette technique est présenté dans la section 7.2.2 de ce manuscrit. La technique a été publiée sous la référence suivante :

> [125] Antoine Lambert, François Queyroi, and Romain Bourqui. Visualizing patterns in Node-link Diagrams. In *Proceedings of the 16th International Conference on Information Visualisation*, IV'12, pages 48–53. IEEE Computer Society, 2012

L'utilisation d'enveloppes concaves pour mettre en exergue des sous-graphes aide l'utilisateur à visualiser le résultat d'une décomposition de graphe. Cependant, quelques ambiguïtés peuvent apparaître. Par exemple quand un sous-graphe contient un grand nombre de sommets et d'arêtes et que son dessin implique un grand nombre de croisements d'arêtes, il peut être difficile de déterminer si un sommet se trouvant dans son enveloppe appartient au sous-graphe ou non. Pour palier à ce problème, nous proposons une méthode pour mettre clairement en évidence un sous-graphe dans le contexte global d'une visualisation de graphe. Cette méthode est basée sur l'utilisation d'une déformation 3D qui va être appliquée à l'échelle locale du sous-graphe à mettre en exergue. Le détail de cette méthode est présenté dans la section 7.3 de ce manuscrit. La méthode a été publiée dans notre article sur le dessin de réseaux métaboliques [124].

8.2 Futures directions

Nous concluons cette thèse par donner de futures directions de recherche dérivant de ses principales contributions. Nous nous focaliserons dans un premier temps sur les améliorations qui pourraient être apportées à notre méthode de regroupement d'arêtes dans un dessin de graphe. Nous donnerons ensuite quelques suggestions pour améliorer la représentation de réseaux métaboliques. Nous donnerons ensuite quelques exemples de

visualisation et de techniques d'interaction qui pourraient bénéficier de notre technique optimisée de rendu de courbes paramétriques. Enfin, nous donnerons quelques pistes sur les futurs travaux à effectuer pour la visualisation de sous-graphes d'intérêt dans le contexte global d'une visualisation de graphe.

8.2.1 Améliorations pouvant être apportées à notre méthode de regroupement d'arêtes

Nous listons ici des possibles améliorations pour notre méthode de regroupement d'arêtes *Winding Roads* [123].

Adaptation multi-niveaux de la méthode Afin de permettre à la méthode de passer à l'échelle sur de très grands graphes, une possible amélioration serait d'effectuer un routage multi-niveaux des arêtes à regrouper. A notre connaissance, il n'existe pour le moment qu'une seule méthode de regroupement d'arêtes utilisant ce paradigme : celle de Gansner *et al.* [84]. A partir d'une décomposition hiérarchique du graphe, l'idée serait de regrouper les arêtes en travaillant de façon descendante (*top-down*) sur les graphes quotients associés à chaque niveau de la décomposition. Au lieu de router une seule arête à la fois, on routerait un ensemble d'arêtes abstraites par une méta-arête. Chaque arête verrait ainsi des portions de sa nouvelle route calculées à chaque niveau. La décomposition en entrée pourrait être liée au domaine d'application du graphe à visualiser. Par exemple dans un réseau géographique, les sommets pourraient être fragmentés suivante leur continent puis leur pays. A partir de la décomposition du graphe, la grille de routage pourrait également être décomposée en fonction de la position des sommets de chaque groupe. L'avantage serait de réduire la taille de la grille afin d'accélérer le processus de routage. Une autre possibilité serait de recalculer une grille de routage à chaque niveau en augmentant sa granularité au fur et à mesure que l'on descend dans l'arbre de hiérarchie associée à la décomposition.

Séparation des arêtes des faisceaux Une autre amélioration possible de notre algorithme serait de pouvoir séparer les arêtes des faisceaux afin qu'elles soient toutes distinguables et non confondues. La méthode de Pupyrev *et al.* [147] permet d'effectuer cette tâche. Pour implémenter une telle fonctionnalité dans notre méthode *Winding Roads*, une idée serait de rediscrétiser la grille sur les portions où les arêtes ont été routées. La finalité serait de créer autant de routes "parallèles" que d'arêtes routées. Un algorithme d'ordonnancement des arêtes à séparer serait également nécessaire pour minimiser les croisements lorsque des arêtes sortent d'un faisceau (également implémenté dans [147]).

Prise en compte d'obstacles pour le routage Notre méthode ne permet pas de prendre en compte des obstacles à contourner lors du routage des arêtes. Cependant, cette fonctionnalité serait facilement implémentable par une simple modification de la grille de routage. L'utilisateur pourrait par exemple définir manuellement des zones du dessin dans lesquelles il ne veut pas que les arêtes soient routées. Les sommets de la grille situés dans ces zones seraient ensuite supprimés. De la même façon, on pourrait forcer les arêtes à prendre une route particulière, comme par exemple pour éviter un groupe de sommets dans le dessin.

8.2.2 Suggestions pour améliorer les représentations de réseaux métaboliques

Nous donnons ici quelques suggestions pour améliorer les représentations automatiques de réseaux métaboliques. Elles découlent de travaux que nous sommes couramment en train d'effectuer dans ce domaine.

Prise en compte des compartiments cellulaires Les différentes molécules et réactions présents dans un réseau métabolique ont généralement un compartiment cellulaire qui leur est associé. Ces compartiments correspondent à différentes parties d'une cellule : noyau, mitochondrie, peroxysome, ... Une relation d'inclusion existe également sur l'ensemble des compartiments d'une cellule. La prise en compte de cette décomposition naturelle lors du processus de dessin permettrait sûrement d'améliorer la représentation du réseau.

Utilisation de standards pour le dessin biologique Les représentations que nous produisons avec notre méthode [124] n'utilisent pas de format de représentation de procédé biologiques particuliers. L'utilisation de standards de dessin biologique comme par exemple SBGN : *The Systems Biology Graphical Notation* [127] pourrait grandement faciliter leur exploitation.

Factorisation de la représentation Il existe de nombreuses réactions similaires au sein d'un réseau métabolique. Cette similarité est basée sur les groupes moléculaires auxquels appartiennent les substrats et produits des réactions. De telles réactions similaires pourraient ainsi être factorisées afin de simplifier la représentation du réseau.

8.2.3 Exploitation de notre technique optimisée de rendu de courbes paramétriques

Nous avons présenté dans cette thèse une technique de rendu optimisée pour rendre des courbes paramétriques. Nous l'avons développé pour mettre en place des techniques d'interaction fluides avec une visualisation de graphe dont les arêtes ont été regroupées en faisceaux. De nombreuses techniques d'interaction pourraient bénéficier de cette technique de rendu, en particulier celle nécessitant de modifier dynamiquement la forme courbe des arêtes. On peut citer par exemple la technique *EdgeLens* de Wong *et al.* [196], qui repousse des arêtes autour d'un point focal en les déformant. Notre technique pourrait également être utilisée pour réaliser des interpolations fluides entre un dessin avec regroupement d'arêtes et un dessin sans (et vice versa), comme dans la technique *MoleView* de Hurter *el al.* [102]. D'autres types de visualisation pourrait également bénéficier de cette technique de rendu comme par exemple les coordonnées parallèles où il est courant de représenter les données comme des courbes. Si on dispose de données dynamiques, il serait alors possible de mettre en place des animations entre deux états sur ce type de représentation.

8.2.4 Visualisation de sous-graphes d'intérêt dans le contexte global : possibles améliorations et futurs travaux

Nous avons détaillé dans cette thèse une méthode permettant de générer des enveloppes concaves pour entourer le dessin d'un sous-graphe à partir de l'espace topologique [125]. Cette méthode a été spécialement conçue pour visualiser le résultat d'une décomposition chevauchante de graphe en garantissant que chaque enveloppe sera distinguable. Pour rappel, nous calculons un ordre sur les sous-graphes à mettre en exergue qui détermine la largeur et l'ordre de rendu des enveloppes associées. Cet ordre est basé sur une coloration d'un graphe de dépendance et une heuristique évaluant la complexité de chaque sous-graphe. Dans notre cas, nous utilisons le nombre de sommets de chaque sous-graphe comme référence mais d'autres heuristiques propres au domaine d'application pourrait être utilisées. Une autre amélioration de la méthode serait la prise en compte des dessins de graphe avec regroupement d'arêtes. Notre méthode ne gère pas bien ce type de dessin, pouvant résulter en des enveloppes confondues le long des faisceaux d'arêtes. Au niveau de l'implémentation de la méthode de génération d'enveloppes, il serait intéressant d'exploiter les architectures multi-cœurs des processeurs contemporains. Comme chaque enveloppe est calculée indépendamment l'une de l'autre, il devrait être possible de calculer plusieurs enveloppes en parallèle. Enfin, il serait intéressant de conduire une évaluation empirique pour comparer l'efficacité de notre méthode avec d'autres existantes, en particulier pour visualiser le résultat d'une décomposition chevauchante de graphe.

Chapitre 8
Bibliographie

[1] J. Abello, F. van Ham, and N. Krishnan. ASK-GraphView : A Large Graph Visualisation System. *IEEE Transactions on Visualization and Computer Graphics*, 12(5) :669–676, 2006. 11

[2] Eytan Adar. GUESS : a language and interface for graph exploration. In *Proceedings of the SIGCHI conference on Human Factors in computing systems*, CHI '06, pages 791–800. ACM, 2006. 12, 26

[3] Y.Y. Ahn, J.P. Bagrow, and S. Lehmann. Link communities reveal multiscale complexity in networks. *Nature*, 466(7307) :761–764, 2010. xxi, 137, 143, 147, 149

[4] B. Alper, N. Riche, G. Ramos, and M. Czerwinski. Design Study of LineSets, a Novel Set Visualization Technique. *IEEE Transactions on Visualization and Computer Graphics*, 17(12) :2259–2267, 2011. xx, 139, 140

[5] R. Amar, J. Eagan, and J. Stasko. Low-Level Components of Analytic Activity in Information Visualization. In *Proceedings of the IEEE Symposium on Information Visualization*, INFOVIS'05. IEEE Computer Society, 2005. 21

[6] Daniel Archambault, Tamara Munzner, and David Auber. Grouse : Feature-Based and Steerable Graph Hierarchy Exploration. In Ken Museth, Torsten Möller, and Anders Ynnerman, editors, *Proceedings of the Joint Eurographics/IEEE-VGTC Symposium on Visualization (EuroVis 2007)*, pages 67–74. Eurographics Association, 2007. 11

[7] Daniel Archambault, Tamara Munzner, and David Auber. TopoLayout : Multilevel graph layout by topological features. *IEEE Transactions on Visualization and Computer Graphics*, 13 :305–317, 2007. 19, 94

[8] Daniel Archambault, Tamara Munzner, and David Auber. GrouseFlocks : Steerable Exploration of Graph Hierarchy Space. *IEEE Transactions on Visualization and Computer Graphics*, 14(4) :900–913, 2008. 11, 26

[9] David Auber. Tulip : A huge graph visualisation framework. In P. Mutzel and M. Jünger, editors, *Graph Drawing Softwares*, Mathematics and Visualization, pages 105–126. Springer-Verlag, 2003. xi, xii, 5, 11, 12, 15, 19, 23, 24, 26, 93, 189

[10] David Auber, Daniel Archambault, Romain Bourqui, Antoine Lambert, Morgan Mathiaut, Patrick Mary, Maylis Delest, Jonathan Dubois, and Guy Mélançon. The Tulip 3 Framework A Scalable Software Library for Information Visualization Applications. Technical Report RR-7860, INRIA, 2012. xi, xii, 5, 11, 12, 15, 23, 24, 26, 189

[11] David Auber and Yves Chiricota. Improved efficiency of spring embedders : taking advantage of GPU programming. In *7th IASTED International Conference on Visualization, Imaging and Image Processing*, pages 169–175. ACTA Press, 2007. 32

[12] David Auber, Noel Novelli, and Guy Melançon. Visually Mining the Datacube using a Pixel-Oriented Technique. In *Proceedings of the 11th International Conference Information Visualization*, IV'07, pages 3–10. IEEE Computer Society, 2007. 183

[13] F. Aurenhammer. Voronoi diagrams—a survey of a fundamental geometric data structure. *ACM Comput. Surv.*, 23(3) :345–405, 1991. 60

[14] Michael Balzer, Oliver Deussen, and Claus Lewerentz. Voronoi treemaps for the visualization of software metrics. In *Proceedings of the ACM Symposium on Software visualization*, SoftVis '05, pages 165–172. ACM, 2005. 15

[15] Curtis Bartley. Forward difference calculation of Bezier curves. *C/C++ Users J.*, 15(11) :19–26, 1997. 112

[16] Moritz Becker and Isabel Rojas. A Graph Layout Algorithm for Drawing Metabolic Pathways. *Bioinformatics*, 17 :461–467, 2001. xvii, 83, 86

[17] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry : Algorithms and Applications*. Springer-Verlag TELOS, 3rd ed. edition, 2008. 11

[18] François Bertault and Peter Eades. Drawing Hypergraphs in the Subset Standard. In *Proceedings of the 8th International Symposium on Graph Drawing*, GD'00, pages 164–169. Springer-Verlag, 2001. 138

[19] François Bertault. A Force-Directed Algorithm that Preserves Edge Crossing Properties. In *Proceedings of the International Symposium on Graph Drawing*, GD'99, pages 351–358. Springer Berlin / Heidelberg, 2000. 17

[20] Eric A. Bier, Maureen C. Stone, Ken Pier, William Buxton, and Tony D. DeRose. Toolglass and Magic Lenses : The See-Through Interface. In *Proceedings of SIGGRAPH '93*, 1993. 22

[21] James F. Blinn. Models of light reflection for computer synthesized pictures. In *SIGGRAPH '77 : Proceedings of the 4th annual conference on Computer graphics and interactive techniques*, pages 192–198. ACM, 1977. 29, 130

[22] James F. Blinn. Simulation of wrinkled surfaces. In *SIGGRAPH '78 : Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, pages 286–292. ACM, 1978. 130

[23] V.D. Blondel, J.L. Guillaume, R. Lambiotte, and E. Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics : Theory and Experiment*, 2008(10) :P10008, 2008. xxii, 152, 153

[24] R. Bourqui, V. Lacroix, L. Cottret, D. Auber, P. Mary, M.-F. Sagot, and F. Jourdan. Metabolic network visualization eliminating node redundance and preserving metabolic pathways. *BMC Systems Biology*, 1 :29, 2007. xvii, 86, 87, 89, 91

[25] Romain Bourqui. Décomposition et Visualisation de graphes : Applications aux Données Biologiques, 2008. 35, 89

[26] U. Brandes, T. Dwyer, and F. Schreiber. Visualizing Related Metabolic Pathways in Two and Half Dimensions. In *Proceedings of the International Symposium on Graph Drawing*, GD'03, pages 110–122, 2004. xvii, 83, 85

[27] Mark Bruls, Kees Huizing, and Jarke van Wijk. Squarified Treemaps. In *Proceedings of the Joint Eurographics and IEEE TCVG Symposium on Visualization*, pages 33–42, 1999. xi, 15, 16

[28] Christoph Buchheim, Michael Jünger, and Sebastian Leipert. A Fast Layout Algorithm for k-Level Graphs. In *Proceedings of the International Symposium on Graph Drawing*, GD'00, pages 229–240. Springer-Verlag, 2001. 19

[29] Kevin Buchin, Bettina Speckmann, and Kevin Verbeek. Angle-Restricted Steiner Arborescences for Flow Map Layout. In Takao Asano, Shin-ichi Nakano, Yoshio Okamoto, and Osamu Watanabe, editors, *Algorithms and Computation*, volume 7074 of *Lecture Notes in Computer Science*, pages 250–259. Springer Berlin Heidelberg, 2011. 54, 138

[30] Kevin Buchin, Bettina Speckmann, and Kevin Verbeek. Flow Map Layout via Spiral Trees. *IEEE Transactions on Visualization and Computer Graphics*, 17(12) :2536–2544, 2011. 54

[31] U.S. Census Bureau. County-to-County Migration Flow Files. `http://www.census.gov/population/www/cen2000/ctytoctyflow.html`, 2003. xiv, xxv, 55, 63, 66, 128

[32] H. Byelas and A. Telea. Visualization of areas of interest in software architecture diagrams. In *Proceedings of the 2006 ACM symposium on Software visualization*, SoftVis'06, pages 105–114. ACM, 2006. xx, 139, 140

[33] Heorhiy Byelas, Egor Bondarev, and Alexandru Telea. Visualization of areas of interest in component-based system architectures. In *Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications*, EUROMICRO'06, pages 160–169. IEEE Computer Society, 2006. 139

[34] Stuart Card. The human-computer interaction handbook. chapter Information visualization, pages 544–582. L. Erlbaum Associates Inc., 2003. 4

[35] Stuart K. Card, Jock D. Mackinlay, and Ben Shneiderman, editors. *Readings in information visualization : using vision to think*. Morgan Kaufmann Publishers Inc., 1999. 3

[36] M. Sheelagh T. Carpendale, David J. Cowperthwaite, F. David Fracchia, and Thomas Shermer. Graph Folding : Extending Detail and Context Viewing into a Tool for Subgraph Comparisons. In *Proceedings of the International Symposium on Graph Drawing*, GD'95, pages 127–139. Springer, 1995. 22, 148, 183

[37] M. Sheelagh T. Carpendale, John Ligh, and Eric Pattison. Achieving higher magnification in context. In *Proceedings of the 17th annual ACM symposium on User interface software and technology*, UIST '04, pages 71–80. ACM, 2004. 22, 150, 183

[38] M. Sheelagh T. Carpendale and Catherine Montagnese. A framework for unifying presentation space. In *Proceedings of the 14th annual ACM symposium on User interface software and technology*, UIST '01, pages 61–70. ACM, 2001. 22, 183

[39] Pierre Caserta, Olivier Zendra, and Damien Bodénès. 3D Hierarchical Edge Bundles to Visualize Relations in a Software City Metaphor. In *6th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT 2011)*, pages 1–8, 2011. 74

[40] E. Catmull and R. Rom. A class of local interpolating splines. *Computer Aided Geometric Design*, pages 317–326, 1974. 114

[41] Norman Chin, Chris Frazier, Paul Ho, Zicheng Liu, and Kevin P. Smith. *The OpenGL Graphics System Utility Library (Version 1.3)*, chapter 5, pages 10–18. Silicon Graphics, Inc., 1998. http://www.opengl.org/registry/doc/glu1.3.pdf. 141

[42] Y. Chiricota, F. Jourdan, and G. Melançon. Metric-Based Network Exploration and Multiscale Scatterplot. In *Proceedings of the IEEE Symposium on Information Visualization*, INFOVIS'04, pages 135–142. IEEE Computer Society, 2004. 127

[43] C. Collins, G. Penn, and S. Carpendale. Bubble sets : Revealing set relations with isocontours over existing visualizations. *IEEE Transactions on Visualization and Computer Graphics*, 15(6) :1009–1016, 2009. xx, 12, 25, 139, 140, 141

[44] Microsoft Corporation. *Microsoft DirectX 9 Programmable Graphics Pipeline*. Microsoft Press, 2003. 30

[45] L. Cottret, D. Wildridge, F. Vinson, M. P. Barrett, H. Charles, M.-F. Sagot, and F. Jourdan. MetExplore : a web server to link metabolomic experiments and genome-scale metabolic networks. *Nucleic Acids Research*, 38(suppl 2) :W132–W137, 2010. xviii, 95, 96, 99, 101

[46] W. Cui, H. Zhou, H. Qu, P. C. Wong, and X. Li. Geometry-Based Edge Clustering for Graph Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 14(6) :1277–1284, 2008. xiv, xxv, 54, 55, 59, 63, 66, 69

[47] Carl de Boor. *A Practical Guide to Splines*. Springer-Verlag, 1978. 112

[48] Paul De Casteljau. Outillage méthodes calculs. Technical report, André Citroen Automobiles SA, Paris, 1959. 112

[49] W. de Nooy, A. Mrvar, and V. Batagelj. *Exploratory Social Network Analysis with Pajek*. Structural Analysis in the Social Sciences. Cambridge University Press, 2011. 12

[50] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. *Graph Drawing : Algorithms for the Visualization Of Graphs*, chapter 2.1.2 : Aesthetics, pages 14–16. Prentice Hall, 1999. 13

[51] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. *Graph Drawing : Algorithms for the Visualization Of Graphs*, chapter 3.1.3 : Radial Drawing, pages 52–55. Prentice Hall, 1999. xi, 14, 15

[52] M. Dickerson, D. Eppstein, M. T. Goodrich, and J. Y. Meng. Confluent Drawings : Visualizing Non-planar Diagrams in a Planar Way. In *Proceedings of the International Symposium on Graph Drawing*, GD'03, pages 1–12, 2004. 53

[53] Reinhard Diestel. *Graph Theory*. Graduate Texts in Mathematics, Volume 173. Springer-Verlag, 4th edition, 2010. 11

[54] E W Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1) :269–271, 1959. 63, 67

[55] D.P Dobkin, E.R. Gansner, E. Koutsofios, and S.C. North. Implementing a General-Purpose Edge Router. In *Proceedings of the International Symposium on Graph Drawing*, GD'97, pages 262–271, 1998. 52

[56] Dogrusoz, Erson, Giral, Demir, Babur, Cetintas, and Colak. Patikaweb : a web interface for analyzing biological pathways through advanced querying and visualization. *Bioinformatics*, 22(3) :374–375, 2005. 83

[57] S. dos Santos and K. Brodlie. Gaining understanding of multivariate and multidimensional data through visualization. *Computers & Graphics*, 28(3) :311–325, 2004. ix, 4

[58] T. Dwyer and L. Nachmanson. Fast Edge-Routing for Large Graphs. In *Proceedings of the International Symposium on Graph Drawing*, GD'09, pages 147–158. Springer-Verlag, 2010. 53

[59] Tim Dwyer, Kim Marriott, Falk Schreiber, Peter Stuckey, Michael Woodward, and Michael Wybrow. Exploration of Networks using overview+detail with Constraint-based cooperative layout. *IEEE Transactions on Visualization and Computer Graphics*, 14 :1293–1300, 2008. 17, 139

[60] Tim Dwyer, Kim Marriott, and Michael Wybrow. Dunnart : A constraint-based network diagram authoring tool. In *Proceedings of the International Symposium on Graph Drawing*, GD'08, pages 420–431. Springer-Verlag, 2009. 17

[61] Tim Dwyer, Kim Marriott, and Michael Wybrow. Topology Preserving Constrained Graph Layout. In *Revised Papers From the International Symposium on Graph Drawing*, GD'08, pages 230–241. Springer-Verlag, 2009. 17

[62] P. Eades and Q.-W. Feng. Multilevel Visualization of Clustered Graphs. In *Proceedings of the International Symposium on Graph Drawing*, GD'95, pages 101–112. Springer-Verlag, 1996. 89

[63] Peter Eades. A Heuristic for Graph Drawing. *Congressus Numerantium*, 42 :149–160, 1984. 16

[64] G. Ellis and A. Dix. A Taxonomy of Clutter Reduction for Information Visualisation. *IEEE Transactions on Visualization and Computer Graphics*, 13(6) :1216–1223, 2007. 52

[65] Niklas Elmqvist, Thanh-Nghi Do, Howard Goodell, Nathalie Henry, and Jean-Daniel Fekete. ZAME : Interactive Large-Scale Graph Visualization. In IEEE Computer Society, editor, *IEEE Pacific Visualization Symposium 2008*, pages 215–222, 2008. 25, 32

[66] Niklas Elmqvist, Pierre Dragicevic, and Jean-Daniel Fekete. Rolling the Dice : Multidimensional Visual Exploration using Scatterplot Matrix Navigation. In *Proceedings of the IEEE Symposium on Information Visualization*, INFOVIS'08, pages 1141–1148, 2008. 6

[67] Klaus Engel, Markus Hadwiger, Joe M. Kniss, Christof Rezk-salama, and Daniel Weiskopf. *Real-time Volume Graphics*. A. K. Peters, Ltd., 2006. 31

[68] Ozan Ersoy, Christophe Hurter, Fernando Paulovich, Gabriel Cantareiro, and Alex Telea. Skeleton-based edge bundling for graph visualization. *IEEE Transactions on Visualization and Computer Graphics*, 17(12) :2364–2373, 2011. xiv, xxv, 55, 57, 63, 66

[69] Leonard Euler. Solutio problematis ad geometriam situs pertinentis. *Commentarii academiae scientiarum Petropolitanae*, 8 :128–140, 1741. ix, 1, 12

[70] Jean-Daniel Fekete. The InfoVis Toolkit. In *Proceedings of the IEEE Symposium on Information Visualization*, INFOVIS'04, pages 167–174, 2004. 12

[71] Jean-Daniel Fekete and Catherine Plaisant. Interactive Information Visualization of a Million Items. In *Proceedings of the IEEE Symposium on Information Visualization*, INFOVIS'02, pages 117–124. IEEE Computer Society, 2002. 31

[72] R. A. Finkel and J. L. Bentley. Quad trees a data structure for retrieval on composite keys. *Acta Informatica*, 4(1) :1–9, 1974. 42, 59

[73] M. Florek and Novotnỳ M. Interactive Information Visualization using Graphics Hardware. Poster Proceedings of Spring Conference on Computer Graphics, 2006. 32

[74] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer graphics : principles and practice (2nd ed.)*. Addison-Wesley Longman Publishing Co., Inc., 1990. 27

[75] S. Fortunato. Community detection in graphs. *Physics Reports*, 486(3-5) :75–174, 2010. 137

[76] S. Fortune. A sweepline algorithm for Voronoi diagrams. In *SCG '86 : Proc. of the second annual symposium on Computational geometry*, pages 313–322, 1986. 60

[77] A. Frick, A. Ludwig, and H. Mehldau. A Fast Adaptive Layout Algorithm for Undirected Graphs. In *Proceedings of the International Symposium on Graph Drawing*, GD'93, pages 389–403, 1994. xi, 16, 18, 94

[78] T. M. J. Fruchterman and E. M. Reingold. Graph Drawing by Force-directed Placement. *Software-Practice and Experience*, 21(11) :1129–1164, 1991. xi, 16, 18

[79] A. Funahashi, Y. Matsuoka, A. Jouraku, M. Morohashi, N. Kikuchi, and H. Kitano. CellDesigner 3.5 : A Versatile Modeling Tool for Biochemical Networks. *Proceedings of the IEEE*, 96(8) :1254–1265, 2008. 85

[80] G. W. Furnas. Generalized fisheye views. *SIGCHI Bull.*, 17(4) :16–23, 1986. 22, 32, 183

[81] S. D. Gabouje and E. Zimányi. Generic visualization of biochemical networks : A new compound graph layout algorithm. In *Poster Proc. of the 4th International Workshop on Efficient and Experimental Algorithms (WEA 05)*, 2005. 83

[82] P. Gajer and S. G. Kobourov. GRIP : Graph Drawing with Intelligent Placement. In *Proceedings of the International Symposium on Graph Drawing*, GD'00, pages 222–228, 2001. xii, 19

[83] E. R. Gansner and Y. Koren. Improved circular layouts. In *Proceedings of the International Symposium on Graph Drawing*, GD'06, pages 386–398, 2007. 54

[84] Emden R. Gansner, Yifan Hu, Stephen North, and Carlos Scheidegger. Multilevel agglomerative edge bundling for visualizing large graphs. In *Proceedings of the 2011 IEEE Pacific Visualization Symposium*, PACIFICVIS '11, pages 187–194. IEEE Computer Society, 2011. xiv, 56, 57, 162

[85] Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Kiem-Phong Vo. A Technique for Drawing Directed Graphs. *IEEE Transactions on Software Engineering*, 19(3) :214–230, 1993. 19, 51, 52

[86] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software - Practice and Experience*, 30(11) :1203–1233, 2000. 12

[87] M. R. Garey and D. S. Johnson. Crossing Number is NP-Complete. *SIAM Journal on Algebraic and Discrete Methods*, 4(3) :312–316, 1983. 88

[88] Mohammad Ghoniem, Jean-Daniel Fekete, and Philippe Castagliola. A Comparison of the Readability of Graphs Using Node-Link and Matrix-Based Representations. In *Proceedings of the IEEE Symposium on Information Visualization*, INFOVIS '04, pages 17–24. IEEE Computer Society, 2004. 13

[89] M. Girvan and M.E.J. Newman. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences*, 99(12) :7821, 2002. 137

[90] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing (3rd Edition)*, chapter 10, pages 707–710. Prentice-Hall, Inc., 2006. 130

[91] S. Grivet, D. Auber, J.P. Domenger, and G. Melançon. Bubble Tree Drawing Algorithm. In *ICCVG'04 : International Conference on Computer Vision Graphics*, pages 633–641, 2004. xi, 14, 15

[92] Khronos Group. OpenCL - The open standard for parallel programming of heterogeneous systems. http ://www.khronos.org/opencl/, 2008. 31

[93] R. Haber and D. A. McNabb. *Visualization idioms : A conceptual model for scientific visualization systems*. IEEE Computer Society, 1990. 4

[94] S. Hachul and M. Jünger. Drawing Large Graphs with a Potential-Field-Based Multilevel Algorithm. In *Proceedings of the International Symposium on Graph Drawing*, GD'04, pages 285–295, 2005. xi, xii, xix, 13, 19, 67, 124

[95] J. Heer and D. Boyd. Vizster : visualizing online social networks. In *Proceedings of the IEEE Symposium on Information Visualization*, INFOVIS'05, pages 32–39. IEEE Computer Society, 2005. 23, 25, 139

[96] Nathalie Henry, Jean-Daniel Fekete, and Michael J. McGuffin. NodeTrix : a Hybrid Visualization of Social Networks. *IEEE Transactions on Visualization and Computer Graphics*, 13(6) :1302–1309, 2007. 26

[97] Ivan Herman, M. Scott Marshall, and Guy Melançon. Graph Visualisation and Navigation in Information Visualisation : A Survey. *IEEE Transactions on Visualization and Computer Graphics*, 6(1) :24–43, 2000. 11

[98] Dorit S. Hochbaum. Approximating covering and packing problems : set cover, vertex cover, independent set, and related problems. In Dorit S. Hochbaum, editor, *Approximation algorithms for NP-hard problems*, pages 94–143. PWS Publishing Co., 1997. 67

[99] D. Holten. Hierachical Edge Bundles : Visualization of Adjacency Relations in Hierarchical Data. *IEEE Transactions on Visualization and Computer Graphics*, 12(5) :805–812, 2006. xiv, 54, 74

[100] D. Holten and J. J. van Wijk. Force-Directed Edge Bundling for Graph Visualization. *Computer Graphics Forum*, 28(3) :983–990, 2009. Proceedings of the Joint Eurographics/IEEE-VGTC Symposium on Visualization (EuroVis 2009). xiv, xxv, 55, 56, 63, 66, 69, 74, 129

[101] Christophe Hurter, Ozan Ersoy, and Alex Telea. Graph Bundling by Kernel Density Estimation. *Computer Graphics Forum*, 31(3) :865–874, 2012. Proceedings of the Joint Eurographics/IEEE-VGTC Symposium on Visualization (EuroVis 2012). xiv, xxv, 55, 57, 63, 66

[102] Christophe Hurter, Alexandru Telea, and Ozan Ersoy. MoleView : An Attribute and Structure-Based Semantic Lens for Large Element-Based Plots. *IEEE Transactions on Visualization and Computer Graphics*, 17(12) :2600–2609, 2011. 164

[103] Stephen Ingram, Tamara Munzner, and Marc Olano. Glimmer : Multilevel MDS on the GPU. *IEEE Transactions on Visualization and Computer Graphics*, 15 :249–261, 2009. 32

[104] Inselberg Alfred and Dimsdale Bernard. Parallel coordinates : a tool for visualizing multi-dimensional geometry. In *Proceedings of the 1st IEEE Symposium on Visualization*, VIS '90, pages 361–378. IEEE Computer Society, 1990. x, 6, 10

[105] Keck Graduate Institute. Sbml viewer. http://sbw.kgi.edu/layout/. 85

[106] Yuntao Jia, Jared Hoberock, Michael Garland, and John Hart. On the visualization of social and other scale-free networks. *IEEE Transactions on Visualization and Computer Graphics*, 14(6) :1285–1292, 2008. 11

[107] Jimmy Johansson, Patric Ljung, Mikael Jern, and Matthew Cooper. Revealing structure in visualizations of dense 2D and 3D parallel coordinates. *Information Visualization*, 5(2), 2006. 32

[108] Angus Johnson. Clipper - an open source freeware polygon clipping library. http://www.angusj.com/delphi/clipper.php, 2010-2012. 145

[109] G. Joshi-Tope, M. Gillespie, I. Vastrik, P. D'Eustachio, E. Schmidt, B. de Bono, B. Jassal, G. R. Gopinath, G. R. Wu, L. Matthews, S. Lewis, E. Birney, and L. Stein. Reactome : a knowledgebase of biological pathways. *Nucleic Acids Research*, 33 :D428–D432, 2005. 86

[110] T. Kamada and S. Kawai. An algorithm for drawing general undirected graphs. *Information Processing Letters*, 31(1) :7–15, 1989. xi, 16, 18

[111] M. Kanehisa, S. Goto, M. Furumichi, M. Tanabe, and M. Hirakawa. KEGG for representation and analysis of molecular networks involving diseases and drugs. *Nucleic Acids Research*, 38 :D355–D360, 2010. 83

[112] P.D. Karp, C.A. Ouzounis, C. Moore-Kochlacs, L. Goldovsky, P. Kaipa, D. Ahren, S. Tsoka, N. Darzentas, V. Kunin, and N. Lopez-Bigas. Expansion of the BioCyc collection of pathway/genome databases to 160 genomes. *Nucleic Acids Research*, 19 :6083–6089, 2005. 81, 95

[113] R.M. Karp. Reducibility Among Combinatorial Problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972. 88, 99

[114] Michael Kaufmann, Marc Kreveld, and Bettina Speckmann. Graph drawing. chapter Subdivision Drawings of Hypergraphs, pages 396–407. Springer-Verlag, 2009. 138

[115] Daniel A. Keim. Pixel-oriented Visualization Techniques for Exploring Very Large Databases. *Journal of Computational and Graphical Statistics*, 5 :58–77, 1996. x, 6, 9

[116] I.M. Keseler, J. Collado-Vides, S. Gama-Castro, J. Ingraham, S. Paley, I.T. Paulsen, M. Peralta-Gil, and P.D. Karp. EcoCyc : A comprehensive database resource for Escherichia coli. *Nucleic Acids Research*, 33 :D334–D337, 2005. 81

[117] Donald E. Knuth. *The Stanford GraphBase : a platform for combinatorial computing*. ACM, 1993. xxi, 143, 146, 149

[118] Jens Krüger and Rüdiger Westermann. Linear algebra operators for GPU implementation of numerical algorithms. In *SIGGRAPH '03 : ACM SIGGRAPH 2003 Papers*, pages 908–916. ACM, 2003. 129

[119] V. Lacroix, L. Cottret, P. Thébault, and M.-F. Sagot. An Introduction to Metabolic Networks and Their Structural Analysis. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 99(1), 2008. 81

[120] Antoine Lambert and David Auber. Graphs analysis and visualization with Tulip-Python. Poster at the *5th European Conference for Scientists using Python (EuroSciPy 2012)*, 2012. xi, xii, 5, 11, 12, 15, 23, 24, 26, 189

[121] Antoine Lambert, David Auber, and Guy Melançon. Living Flows : Enhanced Exploration of Edge-Bundled Graphs Based on GPU-Intensive Edge Rendering. In *Information Visualisation (IV), 2010 14th International Conference*, pages 523–530. IEEE Computer Society, 2010. 110, 131, 134, 159

[122] Antoine Lambert, Romain Bourqui, and David Auber. 3D Edge Bundling for Geographical Data Visualization. In *Proceedings of the 14th International Conference Information Visualisation*, IV'10, pages 329–335. IEEE Computer Society, 2010. 52, 157

[123] Antoine Lambert, Romain Bourqui, and David Auber. Winding Roads : Routing edges into bundles. *Computer Graphics Forum*, 29(3) :853–862, 2010. Proceedings of the Joint Eurographics/IEEE-VGTC Symposium on Visualization (EuroVis 2010). xviii, 52, 95, 96, 121, 127, 131, 156, 157, 160, 162

[124] Antoine Lambert, Jonathan Dubois, and Romain Bourqui. Pathway Preserving Representation of Metabolic Networks. *Computer Graphics Forum*, 30(3) :1021–1030, 2011. Proceedings of the Joint Eurographics/IEEE-VGTC Symposium on Visualization (EuroVis 2011). 12, 83, 138, 148, 158, 161, 163

[125] Antoine Lambert, François Queyroi, and Romain Bourqui. Visualizing patterns in Node-link Diagrams. In *Proceedings of the 16th International Conference on*

Information Visualisation, IV'12, pages 48–53. IEEE Computer Society, 2012. 12, 138, 161, 164

[126] A. Lancichinetti, F. Radicchi, J.J. Ramasco, and S. Fortunato. Finding statistically significant communities in networks. *PloS one*, 6(4) :e18961, 2011. 137

[127] Nicolas Le Novère, Michael Hucka, Huaiyu Mi, Stuart Moodie, Falk Schreiber, Anatoly Sorokin, Emek Demir, Katja Wegner, Mirit I. Aladjem, Sarala M. Wimalaratne, Frank T. Bergman, Ralph Gauges, Peter Ghazal, Hideya Kawaji, Lu Li, Yukiko Matsuoka, Alice Villéger, Sarah E. Boyd, Laurence Calzone, Melanie Courtot, Ugur Dogrusoz, Tom C. Freeman, Akira Funahashi, Samik Ghosh, Akiya Jouraku, Sohyoung Kim, Fedor Kolpakov, Augustin Luna, Sven Sahle, Esther Schmidt, Steven Watterson, Guanming Wu, Igor Goryanin, Douglas B. Kell, Chris Sander, Herbert Sauro, Jacky L. Snoep, Kurt Kohn, and Hiroaki Kitano. The systems biology graphical notation. *Nature biotechnology*, 27(8) :735–741, 2009. 163

[128] B. Lee, C. Plaisant, C. S. Parr, J. Fekete, and N. Henry. Task taxonomy for graph visualization. In *AVI Workshop on Beyond Time and Errors : Novel Evaluation Methods For information Visualization BELIV '06*. ACM, 2006. 21

[129] Chun-Cheng Lin and Hsu-Chun Yen. On Balloon Drawings of Rooted Trees. In *Proceedings of the International Symposium on Graph Drawing*, GD'05, pages 285–296. Springer-Verlag, 2006. 14

[130] Charles Loop and Jim Blinn. Resolution independent curve rendering using programmable graphics hardware. *ACM Trans. Graph.*, 24(3) :1000–1009, 2005. 120

[131] William E. Lorensen and Harvey E. Cline. Marching cubes : A high resolution 3D surface construction algorithm. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '87, pages 163–169. ACM, 1987. 141, 203

[132] E. Mäkinen. How to draw a hypergraph. *International Journal of Computer Mathematics*, 34 :177–185, 1990. 138

[133] Patrick S. McCormick, Jeff Inman, James P. Ahrens, Charles Hansen, and Greg Roth. Scout : A Hardware-Accelerated System for Quantitatively Driven Visualization and Analysis. In *Proceedings of the IEEE Symposium on Visualization*, VIS'04, pages 171–178. IEEE Computer Society, 2004. 31

[134] Bryan McDonnel and Niklas Elmqvist. Towards Utilizing GPUs in Information Visualization : A Model and Implementation of Image-Space Operations. *IEEE Transactions on Visualization and Computer Graphics*, 15(6) :1105–1112, 2009. 31, 32

[135] Michael J. McGuffin and Igor Jurisica. Interaction Techniques for Selecting and Manipulating Subgraphs in Network Visualizations. *IEEE Transactions on Visualization and Computer Graphics*, 15(6) :937–944, 2009. 26

[136] M. Meyer, B. Wong, M. Styczynski, T. Munzner, and H. Pfister. Pathline : A Tool for Comparative Functional Genomics. *Computer Graphics Forum*, 29 :1043–1052, 2010. Proceedings of the Joint Eurographics/IEEE-VGTC Symposium on Visualization (EuroVis 2010). 83

[137] G. Michal. *Biochemical Pathways (Poster)*. Boehringer Mannheim, 1993. xvii, 83, 84, 88

[138] G. Michal. On representation of metabolic pathways. *BioSystems*, 47 :1–7, 1998. 83, 88

[139] Tomer Moscovich, Fanny Chevalier, Nathalie Henry, Emmanuel Pietriga, and Jean-Daniel Fekete. Topology-aware navigation in large networks. In *CHI '09 : Proceedings of the 27th international conference on Human factors in computing systems*, pages 2319–2328. ACM, 2009. 21, 22

[140] T. Munzner. H3 : laying out large directed graphs in 3d hyperbolic space. In *Proceedings of the IEEE Symposium on Information Visualization*, INFOVIS '97, pages 2–10. IEEE Computer Society, 1997. 14

[141] D.E. Nicholson. Metabolic Pathways Map (Poster), 1997. Sigma Chemical Co., St. Louis. 83, 88

[142] Arlind Nocaj and Ulrik Brandes. Computing Voronoi Treemaps : Faster, Simpler, and Resolution-independent. *Computer Graphics Forum*, 31(3) :855–864, 2012. Proceedings of the Joint Eurographics/IEEE-VGTC Symposium on Visualization (EuroVis 2012). 15

[143] NVIDIA. The CUDA homepage. http ://www.nvidia.com/object/cuda_home.html, 2007. 31

[144] Suzanne Pailey and Peter Karp. The Pathway Tools cellular overview diagram and Omics Viewer. *Nucleic Acids Research*, 34(13) :3771–3778, 2006. xvii, 86, 87

[145] G. Palla, I. Derényi, I. Farkas, and T. Vicsek. Uncovering the overlapping community structure of complex networks in nature and society. *Nature*, 435(7043) :814–818, 2005. 137

[146] D. Phan, L. Xiao, R. Yeh, P. Hanrahan, and T. Winograd. Flow Map Layout. In *Proceedings of the IEEE Symposium on Information Visualization*, INFOVIS'05, pages 219–224. IEEE Computer Society, 2005. xxv, 54, 63, 66

[147] Sergey Pupyrev, Lev Nachmanson, Sergey Bereg, and Alexander E. Holroyd. Edge routing with ordered bundles. In *Proceedings of the International Symposium on Graph Drawing*, GD'11, pages 136–147. Springer-Verlag, 2012. xiv, xxv, 53, 55, 63, 66, 162

[148] H.C. Purchase. Which Aesthetic has the Greatest Effect on Human Understanding ? In *Proceedings of the International Symposium on Graph Drawing*, GD'97, pages 248–261. SpringerVerlag, 1998. 13

[149] Helen Purchase, R. F. Cohen, and M. James. Validating Graph Drawing Aesthetics. In *Proceedings of the International Symposium on Graph Drawing*, GD'95, pages 435–446. SpringerVerlag, 1996. 13

[150] Ernesto Ramos and David Donoho. 1983 ASA Data Exposition dataset. `http://lib.stat.cmu.edu/datasets/`, 1983. ix, x, 5, 7, 8, 9, 10

[151] E. M. Reingold and J. S. Tilford. Tidier drawings of trees. *IEEE Trans. Softw. Eng.*, 7(2) :223–228, 1981. 14

[152] Nathalie Henry Riche and Tim Dwyer. Untangling Euler Diagrams. *IEEE Transactions on Visualization and Computer Graphics*, 16(6) :1090–1099, 2010. xx, 138, 139, 140

[153] George G. Robertson, Jock D. Mackinlay, and Stuart K. Card. Cone Trees : animated 3D visualizations of hierarchical information. In *Proceedings of the SIGCHI conference on Human factors in computing systems : Reaching through technology*, CHI '91, pages 189–194. ACM, 1991. 14

[154] Friedemann Röβler, Ralf P. Botchen, and Thomas Ertl. Dynamic Shader Generation for GPU-Based Multi-Volume Ray Casting. *IEEE Comput. Graph. Appl.*, 28(5) :66–77, 2008. 31

[155] M. Rohrschneider, C. Heine, A. Reichenbach, A. Kerren, and G. Scheuermann. A Novel Grid-Based Visualization Approach for Metabolic Networks with Advanced Focus&Context View. In *Proceedings of the International Symposium on Graph Drawing*, GD'09, pages 268–279, 2010. xvii, 86, 87, 88, 89

[156] Randi J. Rost. *OpenGL(R) Shading Language (2nd Edition)*. Addison-Wesley Professional, 2005. xiii, 28, 29, 30

[157] Chris H. Rycroft. Multiscale modeling in granular flow, 2007. xiii, 45

[158] Purvi Saraiya, Chris North, and Karen Duca. Visualizing biological pathways : requirements analysis, systems evaluation and research agenda. *Information Visualization*, 4 :1–15, 2005. 86

[159] Manojit Sarkar and Marc H. Brown. Graphical fisheye views of graphs. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '92, pages 83–91. ACM, 1992. 22

[160] Manojit Sarkar and Marc H. Brown. Graphical fisheye views. *Commun. ACM*, 37(12) :73–83, 1994. 183

[161] Falk Schreiber. Comparison of metabolic pathways using constraint graph drawing. In *APBC 03 : Proceedings of the First Asia-Pacific bioinformatics conference on Bioinformatics*, pages 105–110. Australian Computer Society, Inc., 2003. xvii, 83, 85

[162] R. Sedgewick and J. S. Vitter. Shortest paths in euclidean graphs. *Algorithmica*, 1 :31–48, 1986. 67

[163] David Selassie, Brandon Heller, and Jeffrey Heer. Divided Edge Bundling for Directional Network Data. *IEEE Transactions on Visualization and Computer Graphics*, 17(12) :2354–2363, 2011. 56

[164] Pupyrev Sergey, Nachmanson Lev, and Kaufmann Michael. Improving layered graph layouts with edge bundling. In *Proceedings of the 18th international conference on Graph drawing*, GD'10, pages 329–340. Springer-Verlag, 2011. 56

[165] P. Shannon, A. Markiel, O. Ozierand, N. Baliga, J. Wang, D. Ramage, N. Amin, B. Schwikowski, and T. Ideker. Cytoscape : A Software Environment for Integrated Models of Biomolecular Interaction Networks. *Genome Research*, 13 :2498–2504, 2003. 12, 85

[166] C.-K. Shene. CS3621 Introduction to Computing with Geometry Notes. http ://www.cs.mtu.edu/ shene/COURSES/cs3621/NOTES/, 1997-2011. 110, 120, 121

[167] Ben Shneiderman. Tree visualization with tree-maps : 2-d space-filling approach. *ACM Transactions on Graphics*, 11(1) :92–99, 1992. 15

[168] Ben Shneiderman. The eyes have it : A task by data type taxonomy for information visualizations. In *Proceedings of the IEEE Symposium on Visual Languages*, VL '96, pages 336–343. IEEE Computer Society, 1996. 5

[169] D. Shreiner, M. Woo, J. Neider, and T. Davis. *OpenGL(R) Programming Guide : The Official Guide to Learning OpenGL(R), Version 2 (5th Edition)*. Addison-Wesley Professional, 2005. 27

[170] P. Simonetto, D. Auber, and D. Archambault. Fully Automatic Visualisation of Overlapping Sets. *Computer Graphics Forum*, 28(3) :967–974, 2009. Proceedings of

the Joint Eurographics/IEEE-VGTC Symposium on Visualization (EuroVis 2009). xx, 91, 138, 140

[171] Paolo Simonetto, Daniel Archambault, David Auber, and Romain Bourqui. ImPrEd : An Improved Force-Directed Algorithm that Prevents Nodes from Crossing Edges. *Computer Graphics Forum*, 30(3), 2011. Proceedings of the Joint Eurographics/IEEE-VGTC Symposium on Visualization (EuroVis 2011). 17

[172] M. Sirava, T. Schäfer, M. Eiglsperger, M. Kaufmann, O. Kohlbacher, E. Bornberg-Bauer, and H.-P. Lenhof. BioMiner - modeling, analyzing, and visualizing biochemical pathways and networks. *Bioinformatics*, 18 :S219–S230, 2002. xvii, 83, 85

[173] Steven W. Smith. *The scientist and engineer's guide to digital signal processing*, chapter 24, pages 404–407. California Technical Publishing, 1997. 129

[174] John Stasko and Eugene Zhang. Focus+context display and navigation techniques for enhancing radial, space-filling hierarchy visualizations. In *Proceedings of the IEEE Symposium on Information Visualization*, INFOVIS '00, pages 57–65. IEEE Computer Society, 2000. xi, 15, 16

[175] K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man, and Cybernetics*, 11(2) :109–125, 1981. xii, 19, 20, 56, 93

[176] A. Symeonidis and I. G. Tollis. Visualization of biological information with circular drawings. In *Intl Symposium on Medical Data Analysis (ISBMDA)*, pages 468–478, 2004. 94

[177] L. Szirmay-Kalos and T. Umenhoffer. Displacement Mapping on the GPU - State of the Art. *Computer Graphics Forum*, 27(1), 2008. 130

[178] Alexandru Telea and Ozan Ersoy. Image-Based Edge Bundles : Simplified Visualization of Large Graphs. *Computer Graphics Forum*, 29(3) :843–852, 2010. Proceedings of the Joint Eurographics/IEEE-VGTC Symposium on Visualization (EuroVis 2010). xiv, xxv, 55, 57, 63, 66

[179] Christian Tominski, James Abello, Frank van Ham, and Heidrun Schumann. Fisheye Tree Views and Lenses for Graph Visualization. In *IV '06 : Proceedings of the conference on Information Visualization*, pages 17–24. IEEE Computer Society, 2006. 22

[180] Edward R. Tufte. *Envisioning Information*. Graphics Press (8th printing, June 2001), 1990. 6

[181] S. van Dongen. *A Cluster algorithm for graphs*. PhD thesis, Centrum voor Wiskunde en Informatica, 2000. xi, 13

[182] J. van Helden, L. Wernisch, D. Gilbert, and S. Wodak. Graph-based analysis of metabolic networks. *Ernst Schering Research Foundation Workshop*, 38 :245–274, 2002. 89

[183] R. van Liere and W. de Leeuw. GraphSplatting : Visualizing Graphs as Continuous Fields. *IEEE Transactions on Visualization and Computer Graphics*, 9(2) :206–212, 2003. 12, 127

[184] Jarke J. Van Wijk and Wim A. A. Nuij. Smooth and efficient zooming and panning. In *Proceedings of the IEEE Symposium on Information Visualization*, INFOVIS'03, pages 15–22. IEEE Computer Society, 2003. 21

[185] Bala R. Vatti. A generic solution to polygon clipping. *Communications of the ACM*, 35 :56–63, 1992. 145

[186] T. von Landesberger, A. Kuijper, T. Schreck, J. Kohlhammer, J.J. van Wijk, J.-D. Fekete, and D.W. Fellner. Visual Analysis of Large Graphs : State-of-the-Art and Future Research Challenges. *Computer Graphics Forum*, 30(6) :1719–1749, 2011. 11

[187] Georgy F. Voronoi. Nouvelles applications des paramètres continus à la théorie de formas quadratiques. *J Reine Angew Math*, 134 :198–287, 1908. 44, 60

[188] J. Q. Walker, II. A node-positioning algorithm for general trees. *Software - Practice and Experience*, 20(7) :685–705, 1990. xi, 14, 15

[189] M. Ward, G.G. Grinstein, and D. Keim. *Interactive Data Visualization : Foundations, Techniques, and Applications*. A K Peters Ltd., 2010. 3

[190] C. Ware. *Information Visualization : Perception for Design*. Morgan Kaufmann publishers, 2000. 3

[191] Katja Wegner and Ursula Kummer. A new dynamical layout algorithm for complex biochemical reaction networks. *BMC Bioinformatics*, 6 :212, 2005. xvii, 83, 86

[192] Daniel Weiskopf. *GPU-Based Interactive Visualization Techniques (Mathematics and Visualization)*. Springer-Verlag New York, Inc., 2006. 31

[193] D. J. Welsh and M. B Powell. An upper Bound to the chromaticnumber of a graph and its application to timetabling problems. *The Computer journal*, 10 :85–86, 1967. 68

[194] N. Willems, H. van de Wetering, and J. J. van Wijk. Visualization of vessel movements. *Computer Graphics Forum*, 28(3) :959–966, 2009. Proceedings of the Joint Eurographics/IEEE-VGTC Symposium on Visualization (EuroVis 2009). 130

[195] N. Wong and S. Carpendale. Using Edge Plucking for Interactive Graph Exploration. In *Poster Proceedings of the IEEE Symposium on Information Visualization*, INFOVIS'O5, pages 51–52. IEEE Computer Society, 2005. 53

[196] N. Wong, S. Carpendale, and S. Greenberg. EdgeLens : An Interactive Method for Managing Edge Congestion in Graphs. In *Proceedings of the IEEE Symposium on Information Visualization*, INFOVIS'03, pages 51–58. IEEE Computer Society, 2003. 53, 164

[197] M. Wybrow, K. Marriott, and P.J. Stuckey. Incremental connector routing. In *Proceedings of the International Symposium on Graph Drawing*, GD'05, pages 446–457, 2006. 53

[198] Cem Yuksel, Scott Schaefer, and John Keyser. Parameterization and applications of Catmull-Rom curves. *Comput. Aided Des.*, 43(7) :747–755, 2011. 115

[199] W.W. Zachary. An information flow model for conflict and fission in small groups. *Journal of Anthropological Research*, 33 :452–473, 1977. ix, 2

Chapitre A

Exemple d'exploitation du processeur graphique dans un contexte de visualisation d'information : implémentation d'un *Fisheye*

Cette annexe présente un exemple d'exploitation du processeur graphique dans un contexte de visualisation d'information. Le cas d'étude concerne l'implémentation d'une technique d'interaction applicable à tout type de visualisation, à savoir une lentille de type *Fisheye*. Ce type d'interaction de type Focus+Contexte, généralisé par Furnas [80], permet de mettre en exergue une région d'intérêt dans une visualisation tout en gardant le contexte globale visible. Une pratique courante pour achever cet effet est d'appliquer une déformation géométrique aux éléments graphiques composant une visualisation. On peut par exemple citer les travaux de Sarkar et Brown [160], où ils utilisent des lentilles de type *Fisheye* pour explorer une visualisation de graphe. Un autre exemple sont les travaux de Auber *et al* [12] où une déformation *Fisheye* est appliquée sur une visualisation de type *pixel oriented*. La référence dans ce domaine reste les travaux de Carpendale *et al* [36, 37, 38] qui ont longuement étudié les différentes méthodes de déformation ainsi que leur combinaison pour les interactions de type Focus+Contexte sur tout type de visualisation.

Détaillons maintenant comment implémenter un effet *Fisheye* sur le processeur graphique. Nous utiliserons l'API OpenGL et son langage de *shading* GLSL pour parvenir à nos fins. Pour appliquer l'effet, nous allons écrire un *vertex shader* soit un petit programme exécuté sur le processeur graphique chargé de traiter les sommets en entrée du pipeline de rendu (voir Figure 1.21). L'idée pour achever cet effet est de modifier à la volée la position des sommets envoyés au processeur graphique. La technique peut donc être appliquée à tout type de visualisation implémentée en OpenGL. A noter que la déformation appliquée est en 2D et ne pourra donc marcher que sur des visualisations dessinées dans le plan.

Différents types de fonctions de déformation peuvent être appliquées, nous en présenterons trois :

- f_{hem} : une déformation hémisphérique
- f_{par} : une déformation parabolique
- f_{loupe} : une combinaison d'une déformation parabolique et d'un effet loupe

Les fondements mathématique de ces déformations ne seront pas développés, le but de cet annexe étant simplement de présenter un exemple d'exploitation du processeur graphique pour la Visualisation d'Information.

Chaque fonction prend les quatre mêmes paramètres en entrée :

- p : le point sur lequel appliquer la déformation
- c : le centre de la déformation : plus un élément de la visualisation sera prêt du centre, plus il sera détaillé et inversement.
- r : le rayon de la déformation : seulement les éléments de la visualisation situés dans le cercle défini par le centre et le rayon seront plus détaillés.
- h : la hauteur de la déformation : plus la hauteur sera élevée, plus les éléments proches du centre apparaîtront proches de l'observateur.

et retourne le point p après application de la déformation.

La fonction f_{hem} est définie de la façon suivante :

$f_{hem}(p, c, r, h)$:
... $d = distance(c, p)$
... **if** $d < r$:
..... $ratio = \frac{d}{r}$
..... $coeff = (h + 1) * \frac{d}{h * ratio + 1}$
..... $dir = normalize(p - c) * coeff$
..... **return** $c + dir$
... **else** :
..... **return** p

La fonction f_{par} est définie de la façon suivante :

$f_{par}(p, c, r, h)$:
... $d = distance(c, p)$
... $ratio = \frac{r}{h}$
... $coeff = d + d * \frac{r}{d^2 + 1 + ratio}$
... $dir = normalize(p - c) * coeff$
... **return** $c + dir$

La fonction f_{loupe} est définie de la façon suivante :

$f_{loupe}(p, c, r, h)$:
... $d = distance(c, p)$
... **if** $d < r$:
..... **return** $c + h * (p - c)$
... **else** :
..... **return** $c + (1 + \frac{r * (h - 1)}{d}) * (p - c)$

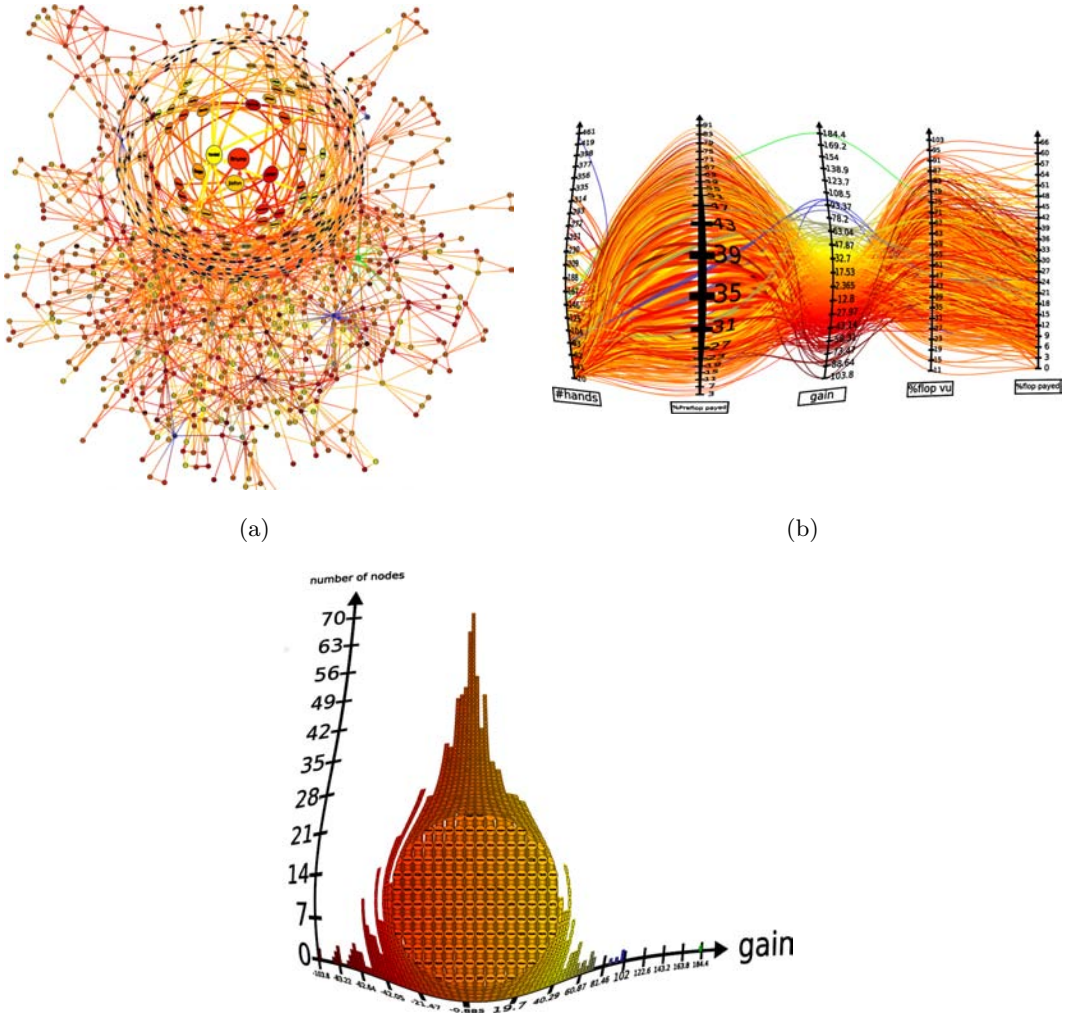
A noter que dans les définitions de ces fonctions, la fonction $distance(u, v)$ calcule la distance euclidienne entre les vecteurs u et v et la fonction $normalize(u)$ retourne le vecteur u normalisé (de même direction mais de norme égale à 1).

La Figure A.1 présente l'implémentation d'un *vertex shader* en GLSL permettant d'appliquer l'un de ces trois effets *Fisheye*. Pour appliquer un des effets, il suffit d'activer le *shader* avant d'appeler le code de rendu de la visualisation. Les paramètres de la déformation sont transmis au *shader* via des variables uniformes. A noter que le centre de la déformation est projeté dans l'espace de la caméra observant la visualisation avant d'être transmis au shader. Ceci est dû au fait que les matrices monde et vue peuvent être modifiées temporairement lors du rendu de la visualisation (e.g. lors du rendu d'un objet défini dans son propre espace).

fisheye.glsl 1

```glsl
#version 120
// Le centre de la déformation doit avoir été projeté dans l'espace caméra
// (multiplié par la matrice monde puis vue)
uniform vec4 center;
uniform float radius;
uniform float height;
uniform int fisheyeType;

void main() {
  // projection du sommet en entrée dans l'espace caméra
  gl_Position = gl_ModelViewMatrix * gl_Vertex;

  // calcul de la distance entre le centre de la déformation
  // et le sommet en entrée
  float dist = distance(center, gl_Position);

  // fisheye hémisphérique
  if (fisheyeType == 1) {

    if (dist < radius) {
      float coeff = (height + 1.0) * dist / (height * dist/ radius + 1.0);
      vec4 dir = normalize(gl_Position - center) * coeff;
      // calcul de la nouvelle position du sommet
      // et projection dans l'espace écran
      gl_Position = gl_ProjectionMatrix * (center + dir);
    } else {
      // pas de déformation
      gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
    }

  // fisheye parabolique
  } else if (fisheyeType == 2) {

    float coeff = dist + dist * radius / (dist * dist + 1.0 + radius / height);
    vec4 dir = normalize(gl_Position - center) * coeff;
    gl_Position = gl_ProjectionMatrix * (center + dir);

  // fisheye parabolique + loupe
  } else {

    if (dist < radius) {
      gl_Position = gl_ProjectionMatrix * (center + height * (gl_Position - center));
    } else {
      gl_Position = gl_ProjectionMatrix * (center + (1.0 + radius * (height - 1.0) / dist) * (gl_Position - center));
    }

  }

  gl_FrontColor =  gl_Color;
  gl_TexCoord[0] = gl_MultiTexCoord0;
}

```

FIGURE A.1: Implémentation d'un *vertex shader* en GLSL pour appliquer un effet *Fisheye* à tout type de visualisation.



(a) (b)

(c)

FIGURE A.2: Exemple d'effets de type *Fisheye* pouvant être appliqués à différents types de visualisation. (a) *Fisheye* de type hémisphérique appliqué sur une visualisation de graphe. (b) *Fisheye* de type parabolique appliqué sur une visualisation de type *coordonnées parallèles*. (c) *Fisheye* de type parabolique+loupe appliqué sur une visualisation de type *histogramme*.

Chapitre B

Détails d'implémentation pour le rendu de courbes paramétriques au GPU

Cette annexe présente les codes sources C++ et GLSL pour rendre trois types de courbes paramétriques au GPU : les courbes de Bézier, les courbes B-Splines et les courbes de Catmull-Rom. Pour les codes C++, la bibliothèque GLEW (*The OpenGL Extension Wrapper*) et le kit de développement de Tulip [9, 10, 120] doivent être installés sur la machine hôte.

Trois implémentations sous la forme de *vertex shader* ont été testées, chacune offrant une alternative différente pour le stockage des points de contrôle des courbes à rendre.

La Figure B.1 présente la manière dont les données nécessaires pour rendre une courbe sont préparées avant d'être transmis à la mémoire du processeur graphique. Toutes les données sont transmises au GPU sous la forme d'un tableau de vecteurs de quatre flottants. Dans le cas du stockage des données dans une texture 2D ou dans un *texture buffer object*, l'ensemble des données de chaque courbe à rendre sont concaténées dans le même tableau. L'index de début des donnés d'une courbe dans le tableau (texture) est transmis au GPU afin de les récupérer depuis le *vertex shader*. Les données suivantes sont envoyées au GPU pour rendre une courbe :

- **nbCp** : le nombre de points de contrôle
- **length** : la somme des distances entre chaque point de contrôle, utile pour le rendu d'une courbe de Catmull-Rom.
- **startSize** : l'épaisseur de la courbe à son point de départ, paramètre utilisé uniquement lorsque l'on rend une courbe sous forme polygonale à l'aide d'un *geometry shader*
- **endSize** : l'épaisseur de la courbe à son point d'arrivée, paramètre utilisé uniquement lorsque l'on rend une courbe sous forme polygonale à l'aide d'un *geometry shader*

– **startColor** : la couleur de la courbe à son point de départ

– **endColor** : la couleur de la courbe à son point d'arrivée

– **cp0, ..., cpN** : les points de contrôle de la courbe

– **k0, ..., kM** : les noeuds pour le rendu d'une courbe B-Spline, utile uniquement pour le *vertex shader* générique de rendu de courbe B-Spline.

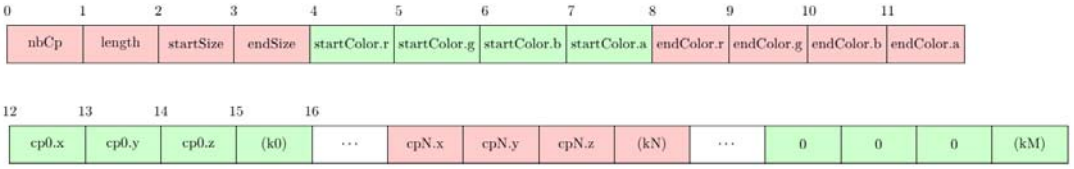


FIGURE B.1: Illustration du paquetage des données nécessaires pour rendre une courbe avant transmission à la mémoire du processeur graphique.

B.1 Implémentation avec stockage des points de contrôle dans un tableau

La première implémentation, qui se révèle également être la plus efficace, stocke les données pour rendre une courbe dans un tableau de vecteurs (à 4 composantes) uniforme. Les variables uniformes peuvent être assimilées à de la mémoire locale en lecture seule et offre un temps d'accès rapide. La Figure B.2 présente le squelette du code GLSL pour le *vertex shader* permettant d'interpoler les points d'une courbe. Le code client OpenGL en C++ est quant à lui donné à la Figure B.3.

B.2 Implémentation avec stockage des points de contrôle dans une texture 2D

La seconde implémentation utilise une texture 2D pour stocker les données des courbes à rendre. Une texture 2D consiste en un tableau de vecteurs (à 4 composantes) à deux dimensions. Cette implémentation se révèle moins efficace que la première dû au temps d'accès à la mémoire des textures depuis le *vertex shader*, bien plus long que celui pour la mémoire locale. Cependant, cette implémentation est plus générique (i.e. ne dépendant pas d'une constante de taille comme dans la première) et devrait pouvoir fonctionner sur la majorité des processeurs graphiques. La Figure B.4 présente le squelette du code GLSL pour le *vertex shader* permettant d'interpoler les points d'une courbe. Le code client OpenGL en C++ est quant à lui donné à la Figure B.5.

curvePointsArrayShader.glsl 1

```glsl
#version 120
// MAX_CONTROL_POINTS is a hardware dependant constant
uniform vec4 controlPoints[MAX_CONTROL_POINTS];

// varying variable to transmit size data
// to geometry shader (for quad curve rendering)
varying float size;

// cache variables
int nbControlPoints = 0;
float totalLength = 0;

int getNbControlPoints() {
  return int(controlPoints[0].x);
}

// Catmull-Rom specific function
float getTotalLength() {
  return controlPoints[0].y;
}

float getStartSize() {
  return controlPoints[0].z;
}

float getEndSize() {
  return controlPoints[0].w;
}

vec4 getStartColor() {
  return controlPoints[1];
}

vec4 getEndColor() {
  return controlPoints[2];
}

vec3 getControlPoint(int index) {
  return controlPoints[index+3].xyz;
}

// B-Spline specific function
float getKnot(int index) {
  return controlPoints[index+3].w;
}

//
// curve computation code goes here
//

void main () {
  // get parameter
  float t = gl_Vertex.x;

  // cache some values
  nbControlPoints = getNbControlPoints();
  totalLength = getTotalLength();

  // compute curve point
  vec3 curvePoint = computeCurvePoint(t);

  // line curve rendering
  gl_Position = gl_ModelViewProjectionMatrix * vec4(curvePoint, 1.0);

  // uncomment the line below for quad curve rendering (through geometry shader)
  //gl_Position = vec4(curvePoint, t);

  // interpolate color
  gl_FrontColor =  mix(getStartColor(), getEndColor(), t);

  // interpolate size (for quad curve rendering)
  size = mix(getStartSize(), getEndSize(), t);
}
```

FIGURE B.2: Squelette du *vertex shader*, en GLSL, interpolant les points d'une courbe avec stockage des points de contrôle dans un tableau uniforme.

curvePointsArrayShader.cpp 1

```cpp
#include <GL/glew.h>
#include <tulip/Coord.h>
#include <tulip/GlShaderProgram.h>
#include <vector>

// header file containing the source code of the
// shaders (stored in std::string)
#include "CurvesShaders.h"

const size_t nbCurveVertices = 200;
static std::vector<tlp::Coord> curveVertices;

// shader object
static tlp::GlShaderProgram *curveArrayVertexShader = NULL;

// external data describing the curves to render
//
// the curves data packed in a Vec4f array
extern std::vector<tlp::Vec4f> curvesData;
// the indexes of each curve data
extern std::vector<size_t> curveDataIdx;
// the size of the data array for each curve
extern std::vector<size_t> curveDataSize;
// the number of curves to render
extern size_t nbCurves;
// the alpha parameter for Catmull-Rom curves
extern float alpha;

void curvePointsArrayShaderRendering() {

    // curve vertices initialisation
    if (curveVertices.empty) {
      for (size_t i = 0 ; i < nbCurveVertices ; ++i) {
        curveVertices.push_back(tlp::Coord(i/static_cast<float>(nbCurveVertices-1), 0, 0));
      }
    }

    glEnableClientState(GL_VERTEX_ARRAY);

    glVertexPointer(3, GL_FLOAT, 3 * sizeof(float), &curveVertices[0]);

    // shader initialisation
    if (curveArrayVertexShader == NULL) {
        curveArrayVertexShader = new tlp::GlShaderProgram();
        curveArrayVertexShader->addShaderFromSourceCode(Vertex, curvePointsArrayVertexShaderSrc);
        // uncomment the line below for quad curve rendering
        //curveArrayVertexShader->addGeometryShaderFromSourceCode(curveExtrusionGeometryShaderSrc,
        GL_LINES_ADJACENCY, GL_TRIANGLE_STRIP);
        curveArrayVertexShader->link();
    }

    curveArrayVertexShader->activate();
    curveArrayVertexShader->setUniformFloat("alpha", alpha);

    // loop over all curves to render
    for (size_t i = 0 ; i < nbCurves ; ++i) {
        // send curve data to the shader
        curveArrayVertexShader->setUniformVec4FloatArray("controlPoints", curveDataSize[i],
        &curvesData[curveDataIdx[i]][0]);
        // line curve rendering
        glDrawArrays(GL_LINE_STRIP, 0, nbCurveVertices);
        // comment the above line and uncomment the one below for quad curve rendering
        //glDrawArrays(GL_LINE_STRIP_ADJACENCY, 0, nbCurveVertices);
    }

    curveArrayVertexShader->desactivate();

    glDisableClientState(GL_VERTEX_ARRAY);
}
```

FIGURE B.3: Implémentation en C++ du code client OpenGL pour rendre des courbes à l'aide du *vertex shader* donné à la Figure B.2

curvePointsTexture2D.glsl 1

```glsl
#version 120
// texture 2D (=vec4 array) sampler from which control points
// will be read
uniform sampler2D controlPoints;

// the index of the beginning of the curve data
// in the texture
uniform int dataIdx;

// the width and height of the texture
uniform int textureSize;

// varying variable to transmit size data
// to geometry shader (for quad curve rendering)
varying float size;

// cache variables
int nbControlPoints = 0;
float totalLength = 0;

int getNbControlPoints() {
  int col = dataIdx / textureSize;
  int row = int(mod(dataIdx, textureSize));
  return int(texture2D(controlPoints, vec2(float(row) / (float(textureSize) - 1.0), float(col) / (float(textureSize) - 1.0))).x);
}

// Catmull-Rom specific function
float getTotalLength() {
  int col = dataIdx / textureSize;
  int row = int(mod(dataIdx, textureSize));
  return texture2D(controlPoints, vec2(float(row) / (float(textureSize) - 1.0), float(col) / (float(textureSize) - 1.0))).y;
}

float getStartSize() {
  int col = dataIdx / textureSize;
  int row = int(mod(dataIdx, textureSize));
  return texture2D(controlPoints, vec2(float(row) / (float(textureSize) - 1.0), float(col) / (float(textureSize) - 1.0))).z;
}

float getEndSize() {
  int col = dataIdx / textureSize;
  int row = int(mod(dataIdx, textureSize));
  return texture2D(controlPoints, vec2(float(row) / (float(textureSize) - 1.0), float(col) / (float(textureSize) - 1.0))).w;
}

vec4 getStartColor() {
  int col = (dataIdx + 1) / textureSize;
  int row = int(mod(dataIdx + 1, textureSize));
  return texture2D(controlPoints, vec2(float(row) / (float(textureSize) - 1.0), float(col) / (float(textureSize) - 1.0)));
}

vec4 getEndColor() {
  int col = (dataIdx + 2) / textureSize;
  int row = int(mod(dataIdx + 2, textureSize));
  return texture2D(controlPoints, vec2(float(row) / (float(textureSize) - 1.0), float(col) / (float(textureSize) - 1.0)));
}

vec3 getControlPoint(int index) {
  int col = (dataIdx + 3 + index) / textureSize;
  int row = int(mod(dataIdx + 3 + index, textureSize));
  return texture2D(controlPoints, vec2(float(row) / (float(textureSize) - 1.0), float(col) / (float(textureSize) - 1.0))).xyz;
}

// B-Spline specific function
float getKnot(int index) {
  int col = (dataIdx + 3 + index) / textureSize;
  int row = int(mod(dataIdx + 3 + index, textureSize));
  return texture2D(controlPoints, vec2(float(row) / (float(textureSize) - 1.0), float(col) / (float(textureSize) - 1.0))).w;
}

//
// curve computation code goes here
//

void main () {
  // get parameter
  float t = gl_Vertex.x;

  // cache some values
  nbControlPoints = getNbControlPoints();
  totalLength = getTotalLength();

  // compute curve point
  vec3 curvePoint = computeCurvePoint(t);

  // line curve rendering
  gl_Position = gl_ModelViewProjectionMatrix * vec4(curvePoint, 1.0);

  // uncomment the line below for quad curve rendering (through geometry shader)
  //gl_Position = vec4(curvePoint, t);

  // interpolate color
  gl_FrontColor =  mix(getStartColor(), getEndColor(), t);

  // interpolate size (for quad curve rendering)
  size = mix(getStartSize(), getEndSize(), t);
}
```

FIGURE B.4: Squelette du *vertex shader*, en GLSL, interpolant les points d'une courbe avec stockage des points de contrôle dans une texture 2D.

curvePointsTexture2D.cpp 1

```cpp
#include <GL/glew.h>
#include <tulip/Coord.h>
#include <tulip/GlShaderProgram.h>
#include <vector>

// header file containing the source code of the
// shaders (stored in std::string)
#include "CurvesShaders.h"

const size_t nbCurveVertices = 200;
static std::vector<tlp::Coord> curveVertices;

// shader object
static tlp::GlShaderProgram *curveTexture2DVertexShader = NULL;

// external data describing the curves to render
//
// the curves data packed in a Vec4f array
extern std::vector<tlp::Vec4f> curvesData;
// the indexes of each curve data
extern std::vector<size_t> curveDataIdx;
// the size of the data array for each curve
extern std::vector<size_t> curveDataSize;
// the number of curves to render
extern size_t nbCurves;
// the alpha parameter for Catmull-Rom curves
extern float alpha;

// OpenGL id of curves data texture
static GLuint controlPointsPackedTexId = 0;
// size of the curves data texture
static size_t textureSize = 1024;

void curvePointsTexture2DShaderRendering() {

    // curve vertices initialisation
    if (curveVertices.empty) {
      for (size_t i = 0 ; i < nbCurveVertices ; ++i) {
        curveVertices.push_back(tlp::Coord(i/static_cast<float>(nbCurveVertices-1), 0, 0));
      }
    }

    glEnableClientState(GL_VERTEX_ARRAY);

    glVertexPointer(3, GL_FLOAT, 3 * sizeof(float), &curveVertices[0]);

    // shader initialisation
    if (curveTexture2DVertexShader == NULL) {
        curveTexture2DVertexShader = new tlp::GlShaderProgram();
        curveTexture2DVertexShader->addShaderFromSourceCode(Vertex, curvePointsTexture2DVertexShaderSrc);
        // uncomment the line below for quad curve rendering
        //curveTexture2DVertexShader->addGeometryShaderFromSourceCode(curveExtrusionGeometryShaderSrc, GL_LINES_ADJACENCY,
        GL_TRIANGLE_STRIP);
        curveTexture2DVertexShader->link();
    }

    // curves data texture initialisation
    if (controlPointsPackedTexId == 0) {
        glGenTextures(1, &controlPointsPackedTexId);
        glBindTexture(GL_TEXTURE_2D, controlPointsPackedTexId);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
        std::vector<tlp::Vec4f> curvesDataCp(curvesData);
        if (curvesDataCp.size() < textureSize * textureSize) {
            curvesDataCp.resize(textureSize * textureSize, tlp::Vec4f(0));
        }
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA32F, textureSize, textureSize, 0, GL_RGBA, GL_FLOAT, &curvesDataCp[0]);
    }

    curveTexture2DVertexShader->activate();
    curveTexture2DVertexShader->setUniformTextureSampler("controlPoints", 0);
    curveTexture2DVertexShader->setUniformInt("textureSize", textureSize);
    curveTexture2DVertexShader->setUniformFloat("alpha", alpha);
    // bind curves data texture
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, controlPointsPackedTexId);
    // loop over all curves to render
    for (size_t i = 0 ; i < nbCurves ; ++i) {
        // send curve data index in texture to shader
        curveTexture2DVertexShader->setUniformInt("dataIdx", curveDataIdx[i]);
        // line curve rendering
        glDrawArrays(GL_LINE_STRIP, 0, nbCurveVertices);
        // comment the above line and uncomment the one below for quad curve rendering
        //glDrawArrays(GL_LINE_STRIP_ADJACENCY, 0, nbCurveVertices);
    }
    curveTexture2DVertexShader->desactivate();

    glDisableClientState(GL_VERTEX_ARRAY);
}
```

FIGURE B.5: Implémentation en C++ du code client OpenGL pour rendre des courbes à l'aide du *vertex shader* donné à la Figure B.4

B.3 Implémentation avec stockage des points de contrôle dans un *texture buffer object*

La dernière implémentation utilise des fonctionnalités plus évoluées de l'API OpenGL (disponible sous forme d'extensions). Les données des courbes à rendre sont stockées dans un *texture buffer object*, assimilable à un grand tableau de vecteurs (à 4 composantes) à une dimension indexable par des entiers depuis un *shader*. Les courbes sont ensuites rendues par *geometry instancing*, technique permettant de rendre une même géométrie un grand nombre de fois en modifiant certains paramètres depuis un *vertex shader*. Une variable autogénéré (`gl_InstanceID`) est alors accessible depuis le *shader* pour récupérer le paramétrage associé à une instance. Cette implémentation se révèle plus efficace que la précédente, le temps d'accès d'un *texture buffer object* semblant être plus rapide que celui d'une texture 2D.

La Figure B.6 présente le squelette du code GLSL pour le *vertex shader* permettant d'interpoler les points d'une courbe. Le code client OpenGL en C++ est quant à lui donné à la Figure B.7.

B.4 *Geometry shader* pour extruder une courbe

Nous proposons également l'implémentation d'un *geometry shader* permettant d'extruder une courbe à la volée afin de lui donner une épaisseur. La courbe est alors rendue sous la forme d'un maillage de triangles. Un *geometry shader* permet de générer de nouvelles primitives à partir de celles envoyées au pipeline de rendu. Il est par exemple possible à partir d'une ligne de générer un ensemble de triangles. C'est cette fonctionnalité que nous allons utiliser pour générer l'extrusion, via l'utilisation d'un type de primitive OpenGL particulier (`GL_LINES_ADJACENCY`). Dans ce cas, le *geometry shader* reçoit 4 points successifs (calculés par un *vertex shader*) de la ligne couramment rendue, permettant de générer un maillage polygonale dont le squelette est définie par la courbe. L'avantage de cette technique est qu'elle permet d'éviter de générer l'extrusion côté CPU, économisant ainsi du temps CPU et de la mémoire.

Le code source de ce *geometry shader* est donné à la Figure B.8.

curvePointsTboInstancing.glsl 1

```glsl
#version 120
#extension GL_EXT_draw_instanced : enable
#extension GL_EXT_gpu_shader4 : enable

// the texture buffer (=vec4 array) sampler from which
// control points will be read
uniform samplerBuffer controlPoints;

// varying variable to transmit size data
// to geometry shader (for quad curve rendering)
varying float size;

// cache variables
int dataIdx = 0;
int nbControlPoints = 0;
float totalLength = 0;

// function to get the index of the curve data in the texture buffer.
// indexes are packed in the beginning of the texture buffer
int getDataIdx() {
  return int(texelFetchBuffer(controlPoints, gl_InstanceID/4)[int(mod(gl_InstanceID, 4))]);
}

int getNbControlPoints() {
  return int(texelFetchBuffer(controlPoints, dataIdx).x);
}

// Catmull-Rom specific function
float getTotalLength() {
  return texelFetchBuffer(controlPoints, dataIdx).y;
}

float getStartSize() {
  return texelFetchBuffer(controlPoints, dataIdx).z;
}

float getEndSize() {
  return texelFetchBuffer(controlPoints, dataIdx).w;
}

vec4 getStartColor() {
  return texelFetchBuffer(controlPoints, dataIdx+1);
}

vec4 getEndColor() {
  return texelFetchBuffer(controlPoints, dataIdx+2);
}

vec3 getControlPoint(int index) {
  return texelFetchBuffer(controlPoints, dataIdx+3+index).xyz;
}

float getKnot(int index) {
  return texelFetchBuffer(controlPoints, dataIdx+3+index).w;
}

//
// curve computation code goes here
//

void main () {
  // get parameter
  float t = gl_Vertex.x;

  // cache some values
  dataIdx = getDataIdx();
  nbControlPoints = getNbControlPoints();
  totalLength = getTotalLength();

  // compute curve point
  vec3 curvePoint = computeCurvePoint(t);

  // line curve rendering
  gl_Position = gl_ModelViewProjectionMatrix * vec4(curvePoint, 1.0);

  // uncomment the line below for quad curve rendering (through geometry shader)
  //gl_Position = vec4(curvePoint, t);

  // interpolate color
  gl_FrontColor =  mix(getStartColor(), getEndColor(), t);

  // interpolate size (for quad curve rendering)
  size = mix(getStartSize(), getEndSize(), t);
}

```

FIGURE B.6: Squelette du *vertex shader*, en GLSL, interpolant les points d'une courbe avec stockage des points de contrôle dans un *texture buffer object* et effectuant un rendu de type *geometry instancing*.

curvePointsTboInstancing.cpp 1

```cpp
#include <GL/glew.h>
#include <tulip/Coord.h>
#include <tulip/GlShaderProgram.h>
#include <vector>

// header file containing the source code of the
// shaders (stored in std::string)
#include "CurvesShaders.h"

// the number of points sampling a single curve
const size_t nbCurveVertices = 200;
static std::vector<tlp::Coord> curveVertices;

// shader object
static GlShaderProgram *curveTBOInstancingVertexShader = NULL;

// external data describing the curves to render
//
// the curves data packed in a Vec4f array
extern std::vector<tlp::Vec4f> curvesData;
// the indexes of each curve data
extern std::vector<size_t> curveDataIdx;
// the size of the data array for each curve
extern std::vector<size_t> curveDataSize;
// the number of curves to render
extern size_t nbCurves;
// the alpha parameter for Catmull-Rom curves
extern float alpha;

// OpenGL id of curves data texture buffer object
static GLuint cpTboId = 0;
// OpenGL id of curves data texture
static GLuint cpTboTex = 0;

void curvePointsTboInstancingShaderRendering() {

    // curve vertices initialisation
    if (curveVertices.empty) {
      for (size_t i = 0 ; i < nbCurveVertices ; ++i) {
        curveVertices.push_back(tlp::Coord(i/static_cast<float>(nbCurveVertices-1), 0, 0));
      }
    }

    glEnableClientState(GL_VERTEX_ARRAY);

    glVertexPointer(3, GL_FLOAT, 3 * sizeof(float), &curveVertices[0]);

    // shader initialisation
    if (curveTBOInstancingVertexShader == NULL) {
        curveTBOInstancingVertexShader = new GlShaderProgram();
        curveTBOInstancingVertexShader->addShaderFromSourceCode(Vertex, curveTboInstancingVertexShaderSrc);
        // uncomment the line below for quad curve rendering
        // curveTBOInstancingVertexShader->addGeometryShaderFromSourceCode(curveExtrusionGeometryShaderSrc, GL_LINES_ADJACENCY,
        GL_TRIANGLE_STRIP);
        curveTBOInstancingVertexShader->link();
    }

    // texture buffer initialisation
    if (cpTboId == 0) {
        // pack indexes of each curve data in the beginning of the texture buffer
        std::vector<tlp::Vec4f> curvesDataTbo;
        if (curveDataIdx.size() > 3) {
            for (size_t i = 0 ; i < curveDataIdx.size() ; i+=4) {
                curvesDataTbo.push_back(tlp::Vec4f(curveDataIdx[i], curveDataIdx[i+1], curveDataIdx[i+2], curveDataIdx[i+3]));
            }
        }
        size_t r = curveDataIdx.size() % 4;
        if (r > 0) {
            tlp::Vec4f last;
            for (size_t i = 0 ; i < r ; ++i) {
                last[i] = curveDataIdx[curveDataIdx.size() - (r-i)];
            }
            curvesDataTbo.push_back(last);
        }

        // offset each index by the current size of the texture buffer
        for (size_t i = 0 ; i < curvesDataTbo.size() ; ++i) {
            for (size_t j = 0 ; j < 4 ; ++j) {
                curvesDataTbo[i][j] += curvesDataTbo.size();
            }
        }

        // concatenate curves data to the texture buffer
        curvesDataTbo.insert(curvesDataTbo.end(), curvesData.begin(), curvesData.end());

        glGenBuffers(1, &cpTboId);
        glBindBuffer(GL_TEXTURE_BUFFER, cpTboId);
        glBufferData(GL_TEXTURE_BUFFER, curvesDataTbo.size() * 4 * sizeof(float), &curvesDataTbo[0], GL_STATIC_DRAW);

        glGenTextures(1, &cpTboTex);
        glBindTexture(GL_TEXTURE_BUFFER, cpTboTex);
        glTexBuffer(GL_TEXTURE_BUFFER, GL_RGBA32F, cpTboId);
    }

    curveTBOInstancingVertexShader->activate();
    curveTBOInstancingVertexShader->setUniformTextureSampler("controlPoints", 0);
    curveTBOInstancingVertexShader->setUniformFloat("alpha", alpha);

    // bind the curve data texture buffer object
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_BUFFER, cpTboTex);

    // line curve instancing rendering
    glDrawArraysInstanced(GL_LINE_STRIP, 0, nbCurveVertices, nbCurves);
    // comment the above line and uncomment the one below for quad curve instancing rendering
    //glDrawArraysInstanced(GL_LINE_STRIP_ADJACENCY, 0, nbCurveVertices, nbCurves);

    curveTBOInstancingVertexShader->desactivate();

    glDisableClientState(GL_VERTEX_ARRAY);
}
```

FIGURE B.7: Implémentation en C++ du code client OpenGL pour rendre des courbes à l'aide du *vertex shader* donné à la Figure B.6

curveExtrusion.glsl 1

```
#version 120
#extension GL_EXT_geometry_shader4 : enable

const float M_PI = 3.14159265358979323846264643;

// size data received from the previously
// executed curve computation vertex shader
varying in float size[4];

void computeExtrusionAndEmitVertices(vec3 pBefore, vec3 pCurrent, vec3 pAfter, float size) {
  vec3 u = pBefore - pCurrent;
  vec3 v = pAfter - pCurrent;
  vec3 xu = normalize(u);
  vec3 xv = normalize(v);
  vec3 bi_xu_xv = normalize(xu+xv);
  float angle = M_PI - acos(dot(u,v)/(length(u)*length(v)));

  // Nan check
  if(angle != angle) {
    angle = 0.0;
  }

  float newSize = size;

  float cosA = cos(angle / 2.0);

  bool parallel = false;

  if (cosA > 1e-1) {
    newSize = size / cosA;
  }

  // workaround for numerically unstable cases
  if (cosA < 1e-1 || angle < 1e-3) {
    vec3 tmp = vec3(0.0);
    tmp = normalize(pAfter - pCurrent);
    bi_xu_xv = tmp;
    bi_xu_xv.x = -tmp.y;
    bi_xu_xv.y = tmp.x;
    parallel = true;
  }

  if (parallel || cross(xu, xv)[2] < 0.0) {
    gl_Position = gl_ModelViewProjectionMatrix * vec4(pCurrent + bi_xu_xv * newSize, 1.0);
    EmitVertex();
    gl_Position = gl_ModelViewProjectionMatrix * vec4(pCurrent - bi_xu_xv * newSize, 1.0);
    EmitVertex();
  } else {
    gl_Position = gl_ModelViewProjectionMatrix * vec4(pCurrent - bi_xu_xv * newSize, 1.0);
    EmitVertex();
    gl_Position = gl_ModelViewProjectionMatrix * vec4(pCurrent + bi_xu_xv * newSize, 1.0);
    EmitVertex();
  }
}

void main() {

  if (gl_PositionIn[0].w == 0.0) {
    gl_FrontColor = gl_FrontColorIn[0];
    computeExtrusionAndEmitVertices(gl_PositionIn[0].xyz - (gl_PositionIn[1].xyz - gl_PositionIn[0].xyz),
    gl_PositionIn[0].xyz, gl_PositionIn[1].xyz, size[0]);
  }

  gl_FrontColor = gl_FrontColorIn[1];
  computeExtrusionAndEmitVertices(gl_PositionIn[0].xyz, gl_PositionIn[1].xyz, gl_PositionIn[2].xyz, size[1]);
  gl_FrontColor = gl_FrontColorIn[2];
  computeExtrusionAndEmitVertices(gl_PositionIn[1].xyz, gl_PositionIn[2].xyz, gl_PositionIn[3].xyz, size[2]);

  if (gl_PositionIn[3].w == 1.0) {
    gl_FrontColor = gl_FrontColorIn[3];
    computeExtrusionAndEmitVertices(gl_PositionIn[2].xyz, gl_PositionIn[3].xyz, gl_PositionIn[3].xyz +
    (gl_PositionIn[3].xyz - gl_PositionIn[2].xyz), size[3]);
  }
}

```

FIGURE B.8: Implémentation en GLSL du *geometry shader* permettant d'extruder une courbe à la volée pour lui donner de l'épaisseur.

B.5 Implémentation des fonctions interpolant le point d'une courbe

Cette section présente les implémentations en GLSL des fonctions interpolant le point d'une courbe.

B.5.1 Courbe de Bézier

La Figure B.9 présente le code source en GLSL de la fonction interpolant un point d'une courbe de Bézier.

bezier.glsl

```
vec3 computeCurvePoint(float t) {
  if (t == 0.0) {
    return getControlPoint(0);
  } else if (t == 1.0) {
    return getControlPoint(nbControlPoints - 1);
  } else {
    float r = float(nbControlPoints);
    float curCoeff = 1.0;
    float s = (1.0 - t);
    float t2 = 1.0;
    vec3 bezierPoint = vec3(0.0);
    for (int i = 0 ; i < nbControlPoints ; ++i) {
      bezierPoint += (getControlPoint(i) * curCoeff * t2 * pow(s, float(nbControlPoints) - 1.0 - float(i)));
      float c = float(i+1);
      curCoeff = curCoeff * (r-c)/c;
      t2 *= t;
    }
    return bezierPoint;
  }
}
```

FIGURE B.9: Implémentation en GLSL de la fonction interpolant le point d'une courbe de Bézier.

B.5.2 Courbe B-Spline

La Figure B.10 présente le code source en GLSL de la fonction optimisée interpolant un point d'une courbe B-Spline uniforme (i.e. l'ensemble des noeuds internes sont uniformement espacés).

La Figure B.11 présente le code source en GLSL de la fonction interpolant un point d'une courbe B-Spline. Dans cette version, les noeuds sont définis par l'utilisateur et lus depuis le *vertex shader*.

B.5.3 Courbe de Catmull-Rom

La Figure B.12 présente le code source en GLSL de la fonction interpolant un point d'une courbe de Catmull-Rom C^1 continue (composée de segments de Bézier cubiques).

bspline_uniform.glsl 1

```
float clampVal(float val) {
  return clamp(val, 0.0, 1.0);
}

vec3 computeCurvePoint(float t) {
  const int curveDegree = 3;
  float coeffs[curveDegree+1];
  float stepKnots = 1.0 / float(nbControlPoints - curveDegree);
  if (t == 0.0) {
    return getControlPoint(0);
  } else if (t == 1.0) {
    return getControlPoint(nbControlPoints - 1);
  } else {
    int k = curveDegree;
    float cpt = 0.0;
    while (t > (cpt * stepKnots) && t >= ((cpt+1.0) * stepKnots)) {
      ++k;
      ++cpt;
    }
    float knotVal = cpt * stepKnots;
    for (int i = 0 ; i <= curveDegree ; ++i) {
      coeffs[i] = 0.0;
    }
    coeffs[curveDegree] = 1.0;
    for (int i = 1 ; i <= curveDegree ; ++i) {
      coeffs[curveDegree-i] = (clampVal(knotVal + stepKnots) - t) / (clampVal(knotVal + stepKnots) -
      clampVal(knotVal + (-i+1) * stepKnots)) * coeffs[curveDegree-i+1];
      int tabIdx = curveDegree-i+1;
      for (int j = -i+1 ; j <= -1 ; ++j) {
        coeffs[tabIdx] = ((t - clampVal(knotVal + j * stepKnots)) / (clampVal(knotVal + (j+i) * stepKnots) -
        clampVal(knotVal + j * stepKnots))) * coeffs[tabIdx] + ((clampVal(knotVal + (j+i+1) * stepKnots) - t) /
        (clampVal(knotVal + (j+i+1) * stepKnots) - clampVal(knotVal + (j+1) * stepKnots))) * coeffs[tabIdx+1];
        ++tabIdx;
      }
      coeffs[curveDegree] = ((t - knotVal) / (clampVal(knotVal + i * stepKnots) - knotVal)) * coeffs[curveDegree];
    }
    int startIdx = k - curveDegree;
    vec3 curvePoint = vec3(0.0);
    for (int i = 0 ; i <= curveDegree ; ++i) {
      curvePoint += coeffs[i] * getControlPoint(startIdx + i);
    }
    return curvePoint;
  }
}
```

FIGURE B.10: Implémentation en GLSL de la fonction interpolant le point d'une courbe B-Spline uniforme.

bspline_generic.glsl 1

```
vec3 computeCurvePoint(float t) {
  const int curveDegree = 3;
  float coeffs[curveDegree+1];
  if (t == 0.0) {
    return getControlPoint(0);
  } else if (t >= 1.0) {
    return getControlPoint(nbControlPoints - 1);
  } else {
    int k = curveDegree;
    while (t >= getKnot(k) && t > getKnot(k+1)) {
      ++k;
    }
    for (int i = 0 ; i <= curveDegree ; ++i) {
      coeffs[i] = 0.0;
    }
    coeffs[curveDegree] = 1.0;
    for (int i = 1 ; i <= curveDegree ; ++i) {
      coeffs[curveDegree-i] = ((getKnot(k+1) - t) / (getKnot(k+1) - getKnot(k-i+1))) * coeffs[curveDegree-i+1];
      int tabIdx = curveDegree-i+1;
      for (int j = -i+1 ; j <= -1 ; ++j) {
        coeffs[tabIdx] = ((t - getKnot(k+j)) / (getKnot(k+j+i) - getKnot(k+j))) * coeffs[tabIdx] +
        ((getKnot(k+j+i+1) - t) / (getKnot(k+j+i+1) - getKnot(k+j+1))) * coeffs[tabIdx+1];
        ++tabIdx;
      }
      coeffs[curveDegree] = ((t - getKnot(k)) / (getKnot(k+i) - getKnot(k))) * coeffs[curveDegree];
    }
    int startIdx = k - curveDegree ;
    vec3 curvePoint = vec3(0.0);
    for (int i = 0 ; i <= curveDegree ; ++i) {
      curvePoint += coeffs[i] * getControlPoint(startIdx + i);
    }
    return curvePoint;
  }
}
```

FIGURE B.11: Implémentation en GLSL de la fonction interpolant le point d'une courbe B-Spline où les noeuds sont définis par l'utilisateur.

catmull.glsl 1

```glsl
uniform float alpha;

vec3 bezierControlPoints[4];

float parameter[2];

void computeBezierSegmentControlPoints(vec3 pBefore, vec3 pStart, vec3 pEnd, vec3 pAfter) {
  bezierControlPoints[0] = pStart;
  float d1 = distance(pBefore, pStart);
  float d2 = distance(pStart, pEnd);
  float d3 = distance(pEnd, pAfter);
  float d1alpha = pow(d1, alpha);
  float d12alpha = pow(d1, 2*alpha);
  float d2alpha = pow(d2, alpha);
  float d22alpha = pow(d2, 2*alpha);
  float d3alpha = pow(d3, alpha);
  float d32alpha = pow(d3, 2*alpha);
  bezierControlPoints[1] = (d12alpha*pEnd-
  d22alpha*pBefore+(2*d12alpha+3*d1alpha*d2alpha+d22alpha)*pStart)/(3*d1alpha*(d1alpha+d2alpha));
  bezierControlPoints[2] = (d32alpha*pStart-
  d22alpha*pAfter+(2*d32alpha+3*d3alpha*d2alpha+d22alpha)*pEnd)/(3*d3alpha*(d3alpha+d2alpha));
  bezierControlPoints[3] = pEnd;
}

int computeSegmentIndex(float t) {
  float dist = pow(distance(getControlPoint(0), getControlPoint(1)), alpha);
  parameter[0] = 0.0;
  parameter[1] = dist / totalLength;
  if (t == 0.0) {
    return 0;
  } else if (t == 1.0)   {
    return nbControlPoints - 2;
  } else {
    int i = 0;
    while (t >= (dist / totalLength)) {
      ++i;
      parameter[0] = dist / totalLength;
      dist += pow(distance(getControlPoint(i), getControlPoint(i+1)), alpha);
    }
    parameter[1] = dist / totalLength;
    return i;
  }
}

vec3 computeCurvePoint(float t) {
  int i = computeSegmentIndex(t);
  float localT = 0.0;
  if (t == 1.0) {
    localT = 1.0;
  } else if (t != 0.0) {
    localT = (t - parameter[0]) / (parameter[1] - parameter[0]);
  }
  if (i == 0) {
    computeBezierSegmentControlPoints(getControlPoint(i) - (getControlPoint(i+1) - getControlPoint(i)),
    getControlPoint(i), getControlPoint(i+1), getControlPoint(i+2));
  } else if (i == nbControlPoints - 2) {
    computeBezierSegmentControlPoints(getControlPoint(i-1), getControlPoint(i), getControlPoint(i+1),
    getControlPoint(i+1) + (getControlPoint(i+1) - getControlPoint(i)));
  } else {
    computeBezierSegmentControlPoints(getControlPoint(i-1), getControlPoint(i), getControlPoint(i+1),
    getControlPoint(i+2));
  }
  float t2 = localT * localT;
  float t3 = t2 * localT;
  float s = 1.0 - localT;
  float s2 = s * s;
  float s3 = s2 * s;
  return (bezierControlPoints[0] * s3 + bezierControlPoints[1] * 3.0 * localT * s2 + bezierControlPoints[2] * 3.0
  * t2 * s + bezierControlPoints[3] * t3);
}
```

FIGURE B.12: Implémentation en GLSL de la fonction interpolant le point d'une courbe de Catmull-Rom C^1 continue.

Chapitre C

Marching Squares : un algorithme d'extraction de contours

Cette annexe présente le détail de l'algorithme *Marching Squares*, que nous avons utilisé pour générer des enveloppes concaves afin de mettre en exergue des sous-graphes dans le contexte d'une visualisation globale d'un réseau (voir section 7.2.1).

Cette algorithme permet de reconstruire des surfaces implicites (ou iso-surfaces ou contours) dans un champ scalaire à deux dimensions (un tableau rectangulaire contenant des valeurs numériques). Le principe de fonctionnement est le même que l'algorithme *Marching Cubes* [131] sauf que l'algorithme travaille dans le plan. Des carrés sont donc utilisés comme forme de base au lieu de cubes.

L'algorithme est très simple à comprendre. Il consiste à exécuter une boucle et examiner un carré de 4 cases du champ scalaire à la fois. En fonction de la configuration de ce carré, dépendant d'une valeur seuil dans le champ scalaire, un déplacement dans une des 4 directions (gauche, droite, haut, bas) est effectué. La Figure C.1 présente les 16 configurations possibles, le code qui leur est associé ainsi que le déplacement qu'elles engendrent. On commence donc par examiner le carré en haut à gauche et on progresse vers la droite jusqu'à atteindre la frontière d'une iso-surface. A partir de là, on regarde la configuration du carré et on se déplace en fonction. De cette façon, on va "marcher" le long du contour de la surface jusqu'à revenir au point de départ. La Figure C.2 illustre ce fonctionnement.

L'implémentation complète de l'algorithme en pseudo-code est présentée dans les Algorithmes 4, 5, 6 et 7.

Algorithme 4 Algorithme *Marching Squares* : procédure principale

Entrées: - f : un champ scalaire à deux dimensions
- w : largeur du champ scalaire
- h : hauteur du champ scalaire
- v : seuil déterminant si une valeur de f est dans un contour ou non

Sorties: - une liste de contours/trous définis par une suite de coordonnées dans f

```
1: contours = liste()
2: celluleVisite = tableau_booléen(w − 1, h − 1, faux)
3: Pour i de 0 à h − 1 faire
4:     Pour j de 0 à w − 1 faire
5:         Si celluleVisite[i][j] == vrai alors
6:             Continuer
7:         Fin Si
8:         typeCellule = calculerTypeCellule(f, v, i, j)
9:         Si modulo(typeCelulle, 15) != 0 alors
10:            contours.ajouter(extraireContour(f, v, i, j, celluleVisite))
11:        Fin Si
12:        celluleVisite[i][j] = vrai
13:    Fin Pour
14: Fin Pour
15: Retourner contours
```

Algorithme 5 Algorithme *Marching Squares* : fonction calculant le type d'une cellule

```
1: Fonction calculerTypeCellule(f, v, i, j)
2:     type = 0
3:     Si f[i][j] > v alors
4:         type += 1
5:     Fin Si
6:     Si f[i+1][j] > v alors
7:         type += 4
8:     Fin Si
9:     Si f[i+1][j+1] > v alors
10:        type += 8
11:    Fin Si
12:    Si f[i][j+1] > v alors
13:        type += 2
14:    Fin Si
15:    Retourner type
16: Fin Fonction
```

Algorithme 6 Algorithme *Marching Squares* : fonction extrayant un contour (partie 1)

```
1: Fonction EXTRAIRECONTOUR(f, v, i, j, celluleVisite)
2:     Nord = (0, 1)
3:     Sud = (0, −1)
4:     Est = (1, 0)
5:     Ouest = (−1, 0)
6:     dir = Est
7:     precDir = Est
8:     arretMarche = faux
9:     contour = liste()
10:    posI = i
11:    posJ = j
12:    Tant que arretMarche == faux faire
13:        type = CALCULERTYPECELLULE(f, v, posI, posJ)
14:        Si type == 0 alors
15:            dir = Est
16:        Sinon Si type == 1 alors
17:            dir = Ouest
18:        Sinon Si type == 2 alors
19:            dir = Sud
20:        Sinon Si type == 3 alors
21:            dir = Ouest
22:        Sinon Si type == 4 alors
23:            dir = Nord
24:        Sinon Si type == 5 alors
25:            dir = Nord
26:        Sinon Si type == 6 alors
27:            Si precDir == Ouest alors
28:                dir = Nord
29:            Sinon
30:                dir = Sud
31:            Fin Si
32:        Sinon Si type == 7 alors
33:            dir = Nord
34:        Sinon Si type == 8 alors
35:            dir = Est
```

Algorithme 7 Algorithme *Marching Squares* : fonction extrayant un contour (partie 2)

```
36:         Sinon Si type == 9 alors
37:             Si precDir == Sud alors
38:                 dir = Ouest
39:             Sinon
40:                 dir = Est
41:             Fin Si
42:         Sinon Si type == 10 alors
43:             dir = Sud
44:         Sinon Si type == 11 alors
45:             dir = Ouest
46:         Sinon Si type == 12 alors
47:             dir = Est
48:         Sinon Si type == 13 alors
49:             dir = Est
50:         Sinon Si type == 14 alors
51:             dir = Sud
52:         Fin Si
53:         celluleVisite[posI][posJ] = vrai
54:         contour.ajouter((posI, posJ))
55:         posI += dir[1]
56:         posJ += dir[0]
57:         precDir = dir
58:         Si posI == i et posJ == j alors
59:             arretMarche = vrai
60:         Fin Si
61:     Fin Tant que
62:     Retourner contour
63: Fin Fonction
```

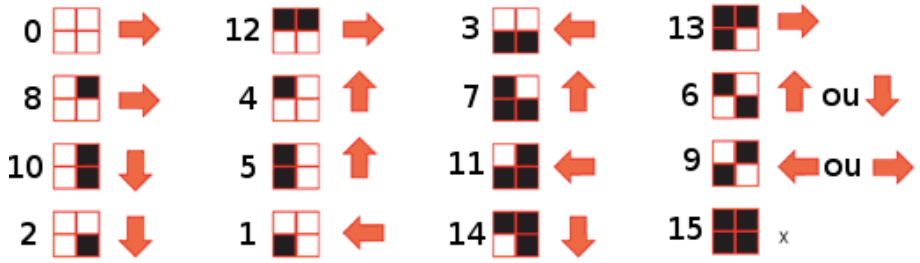


FIGURE C.1: Illustration des 16 configurations possibles pour l'algorithme *Marching Squares* (source de l'image original : Wikipédia). Une case noire signifie que la valeur dans le champ scalaire associé est supérieure à un seuil. Les codes associés à ces configurations (voir Algorithme 5) et les mouvements qu'elles engendrent sont également représentés.

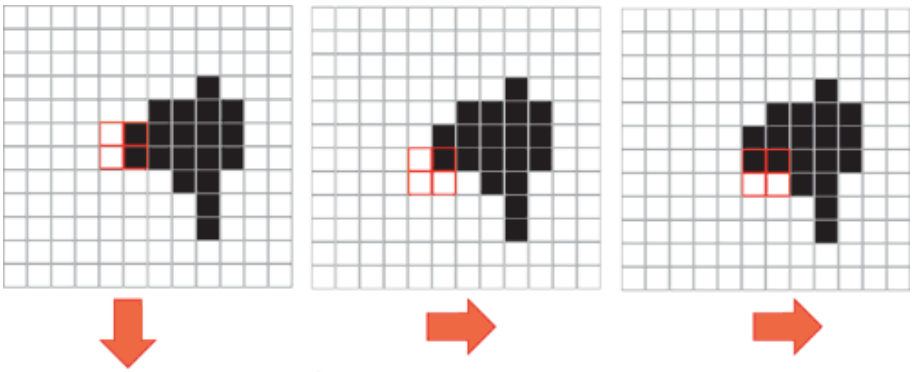


FIGURE C.2: Illustration du fonctionnement de l'algorithme *Marching Squares* (source de l'image : Wikipédia).

Visualisation interactive de graphes : élaboration et optimisation d'algorithmes à coûts computationnels élevés.

Résumé : Un graphe est un objet mathématique modélisant des relations sur un ensemble d'éléments. Il est utilisé dans de nombreux domaines à des fins de modélisation. La taille et la complexité des graphes manipulés de nos jours entraînent des besoins de visualisation afin de mieux les analyser. Dans cette thèse, nous présentons différents travaux en visualisation interactive de graphes qui s'attachent à exploiter les architectures de calcul parallèle (CPU et GPU) disponibles sur les stations de travail contemporaines.
Les contributions de cette thèse ont été divisées en deux parties. La première partie porte sur des problématiques de dessin de graphe. Un algorithme de regroupement d'arêtes et une méthode permettant de dessiner un réseau métabolique complet y sont présentés. La seconde partie présente des techniques d'infographie pour la visualisation interactive de graphes.

Mots-clés : Visualisation de graphes, regroupement d'arêtes, bioinformatique, infographie.

Discipline : Informatique.

Interactive graph visualisation : elaboration and optimisation of algorithms with high computationnal cost.

Abstract : A graph is a mathematical object used to model relations over a set of elements. It is used in numerous fields for modeling purposes. The size and complexity of graphs manipulated today call a need for visualization to better analyze them. In that thesis, we introduce different works in interactive graph visualisation which aim at exploiting parallel computing architectures (CPU and GPU) available on contemporary workstations.
The contributions of that thesis have been divided into two parts. The first part focuses on graph drawing problems. An edge bundling algorithm and a method for drawing a complete metabolic network are detailed. The second part introduces computer graphics techniques for interactive graph visualisation.

Keywords : Graph visualization, edge bundling, bioinformatic, computer graphics.

Field : Computer Science.