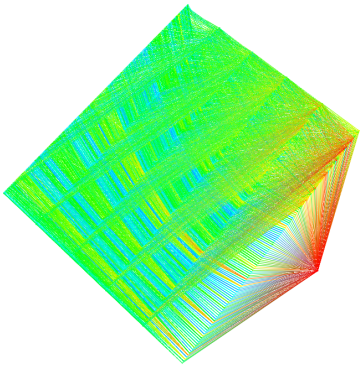
Rapport final

Fouille de données à l'aide de la visualisation



Najima Belkis Fabien Gallot Ruding Lou Mathieu Souchaud

Clients : D. Auber, N. Hanusse, S. Maabout
Chargé de TD : B. Vauquelin

Master Sciences & Technologies mention Informatique

29 avril 2005

Résumé

Des chercheurs ou scientifiques dans différents domaines (archéologie, biologie, médecine...) manipulent des bases de données multidimensionnelles (i.e. avec un nombre de tuples pouvant atteindre l'ordre du million). Due à la multitude des informations, l'analyse de telles bases de données est très compliquée.

Une des méthodes d'analyse de ces bases est de les représenter par un graphe orienté appelé cube de données. Dans ce cube chaque sommet représente une combinaison des valeurs d'un ou plusieurs tuples de la base. La particularité de ce graphe est de contenir tous les résultats de requêtes qui puissent être demandées à une base de données.

Afin d'améliorer la fouille de ces données, nous proposons dans cette étude un outil qui apporte une aide grâce à la visualisation. Cet outil utilise la représentation d'une base de données sous forme de cube de données. En visualisant tous les sommets d'un cube en même temps, une première vision globale de la base est possible.

Cet outil, un prototype destiné à la recherche, permet aussi, au sein d'une interface simple et intuitive, pensée pour des personnes sans connaissances approfondies en informatique, de se déplacer dans le cube de données ou de le simplifier.

Table des matières

Chapitre 1

Introduction au projet

Depuis quelques années avec l'évolution des technologies, les scientifiques ont pu travailler avec des bases de données de plus en plus grandes. Les bases de données multidimensionnelles qui, à la différence des bases de données classiques, ne se représente pas sous forme tabulaire mais sous forme hypercubique, sont utilisées par des analystes qui cherchent des liens entre les informations dans ces multitudes de données. Les domaines d'applications sont alors fort étendus : un médecin peut rechercher les facteurs de risques d'une maladie, un directeur de ressources humaines d'un magasin peut quant à lui rechercher les relations vendeurs-produits-magasins, ou bien un archéologue va s'intérresser aux caractériques d'objets dans le cadre de fouilles.

Le cube de données offre une abstraction très proche de la façon dont l'analyste voit et interroge les données. Il organise des données en une ou plusieurs dimensions qui déterminent une mesure d'intérêt [5]. Les attributs d'une table relationnelle sont alors considérés comme des dimensions et à chaque combinaison d'un ensemble d'attributs correspond une valeur qui peut être une des fonctions d'agrégation classique telles que la somme, le maximum, le minimum, le nombre d'élément ou la moyenne.

L'affichage d'un cube de données devient complexe en raison du nombre de sommets et d'arêtes que l'on peut obtenir : plusieurs millions de sommets et d'arcs à partir d'une base de plusieurs milliers de tuples. Le logiciel Tulip, conçu par D.Auber, résout en partie ce problème. En effet ce logiciel permet de visualiser des grands graphes avec plusieurs centaines de milliers d'éléments. L'utilisation de Tulip a donc été primordiale, dans ce projet, pour le mener à bien.

Cependant, une construction "typique" d'un cube de données ne donne pas forcément une visualisation immédiate des tendances de la base de données. Ainsi des simplifications du cube sont utiles afin de diminuer le nombre d'éléments dans le graphe. La première simplification possible est le filtrage sur certaines valeurs ou attributs de la base, qui peuvent éliminer des informations jugées inintéressantes. L'autre simplification possible, est la construction d'un résumé de cube, afin de simplifier ce dernier, et ne garder que sa structure. Un algorithme de résumé de cube a donc été partiellement implé-

menté. Une version primitive, en est présentée ici.

L'outil conçu, prévu pour des personnes n'ayant pas nécessairement des connaissances approfondies en informatique, est avant tout un prototype dédié à la recherche. Aussi Tulip ayant un domaine d'action nettement supérieur à nos besoins, une interface propre à notre projet sera indispensable. Les opérations de manipulations sur les graphes (zooms, déplacements, sélections d'éléments, ...) seront ainsi implémentés pour permettre une meilleure analyse des données.

Chapitre 2

Analyse de l'existant

2.0.1 Les cubes de données

Définition Générale

Un cube de données est un graphe orienté créé à partir d'une base de données multidimensionnelle. Il permet une analyse plus simple et plus rapide que la base de données elle-même, grâce à la possibilité de visualiser l'intrégralité des tuples de la base et leurs dépendances.

A un sommet on associe un tuple avec un degré de liberté sur les champs du tuple, en fonction de la place du sommet dans le cube. Outre le tuple, est associée au sommet une valeur calculée d'après une des fonctions d'agrégation classique sur les bases de données : MIN (minimum), MAX (maximum), SUM (somme), COUNT (nombre de tuples), AVG (moyenne).

La visualisation du cube de données permet de faire ressortir les liens entre les valeurs des sommets et les valeurs des attributs des tuples.

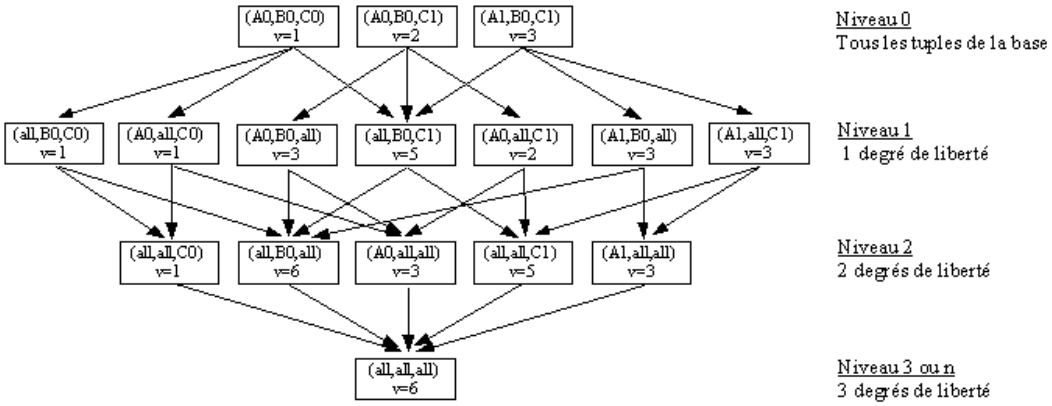
Création du cube

Dans un cube de données, les sommets sont répartis en plusieurs niveaux. Le cube contient un niveau de plus que le nombre de champs de la base de données à partir de laquelle il est construit. Sur le niveau 0 du cube, chaque sommet est associé à un tuple de la base. On compte donc autant de sommets sur ce niveau que de tuples différents dans la base. Les sommets des autres niveaux correspondent aux permutations possibles des degrés de liberté appliqués aux différents champs des tuples de la base. Autrement dit sur un niveau k, on trouve tous les tuples issus de la base, avec un degré de liberté égal à k. Pour trouver la valeur d'un des sommets de niveau k, on applique la fonction à tous les tuples de niveau k-1 qui sont contenus dans le tuple de niveau k. Sur le dernier niveau (n) du cube, on applique un degré de liberté de n sur les n attributs de la base : on trouve donc un seul sommet dont la valeur dépend de l'ensemble des attributs de la base. Les arcs relient les sommets de niveaux consécutifs. Il n'y a pas d'arcs entre les sommets d'un même niveau ni entre des sommets de niveaux non adjacents. L'origine de l'arc est toujours sur le sommet de niveau inférieur. Deux sommets sont

reliés par un arc si le tuple du sommet d'origine est contenu dans celui du second sommet.

La figure 2.1 est un exemple de cube de données avec une base de données très simple : trois attributs, trois tuples ; (A0,B0,C0 ;1), (A0,B0,C1 ;2), (A1,B0,C1 ;3). 'v' représente la valeur du tuple. On a choisi pour cette exemple la fonction SUM. Le mot 'all' indique quel attribut est libre.

FIG. 2.1 – Cube de données



Avantages

Le cube de données permet de traiter les bases de données multidimensionnelles. Ces bases ayant un nombre très important de tuples (plusieurs milliers ou millions), il est difficile de faire des requêtes en un temps raisonnable. En effet si on veut connaitre, par exemple, le nombre de tuple qui possède une certaine valeur sur un des attributs, il faut impérativement parcourir entièrement la base, ce qui est long... Grâce au cube de données, on peut retrouver en un temps constant une telle requête. Lors d'une recherche dans une base de données, on peut trier les tuples en fonction de la requête puis faire une recherche dichotomique, ce qui est assez rapide. Cependant, pour chaque requête il faut refaire un arbre binaire de recherche, ce qui malgré tout peut prendre un temps non négligeable en raison de la taille de la base et du tri des tuples. L'idée du cube de données est de construire une fois pour toutes, toutes les requêtes possibles et de les ordonner dans un graphe. Après la création du cube, une requête peut alors être exécutée en un temps constant sans avoir à parcourir de nouveau l'ensemble de la base de données.

Inconvénients et extensions

L'inconvénient majeur du cube de données est sa création. En effet, en créant toutes les permutations possibles de tous les degrés de libertés sur les attributs de la base, on doit forcément construire un graphe avec nombre

exponentiel de sommets par rapport à la taille de la base. On obtient donc des algorithmes de création de graphes qui sont (au mieux!) exponentiels.

De plus étant donné que le nombre de sommets du graphe créé est exponentiel par rapport à la taille de la base, la visualisation des liens entre les valeurs des tuples et la valeur de leurs attributs n'est pas forcément évidente. C'est pourquoi il existe des simplifications ou résumés de cube de données. L'idée est de regrouper tous les sommets ayant la même valeur, ou une valeur proche , en un seul sommet ou classe. Ce procédé réduit le nombre de sommets et permet souvent une meilleure vision de la structure du graphe. Diverses solutions sont alors possibles pour résumer un cube comme par exemple le "Quotient Cube" [8], ou le "Range Cube" [7]. Cependant certains problèmes non encore résolus peuvent se poser, comme la disparition des niveaux dans le graphe, ou l'apparition de boucles qui changent l'approche fondamentale du cube.

2.0.2 Les programmes de manipulation de graphes

L'exploration du cube de données nécessite un outil de visualisation de graphe possédant des caractéristiques précises.

Cet outil doit permettre de dessiner le graphe en 2D, ou accessoirement en 3D, puis de zoomer ou de filtrer à la demande sur les parties intéressantes du cube. La notion de filtrage conduit directement à la notion de sous-graphes. L'outil doit donc pouvoir les gérer. De plus, l'algorithme de simplification du cube de données nécessite d'avoir des fonctions de groupage de plusieurs sommets en un seul, tout en conservant en mémoire les sommets de chaque groupe.

On doit aussi pouvoir agir sur les données contenues par les éléments du graphe. Il faut donc que l'outil propose une interface utilisateur d'affichage et de recherche d'informations. Les logiciels de dessin de graphes ne sont donc pas adéquats dans notre cas. Les codes sources du logiciel doivent être distribués librement, afin d'y greffer facilement de nouveaux outils. Enfin, en raison de la taille des cubes de données, le logiciel doit pouvoir supporter des graphes de tailles importantes (supérieurs à 100 000 éléments).

Créer un tel outil est complexe et ne rentre pas dans le cadre de ce projet. Nous devons donc utiliser un des logiciels existant. Il existe beaucoup de logiciels de manipulations de graphes, mais seulement quelques uns proposent une interface utilisateur de recherche d'informations, et peu sont capables de gérer de larges structures de données. D'après David Auber [2], seuls quelques logiciels peuvent supporter des graphes contenant plus de 100.000 éléments.

Le premier, développé par Wills [11], est malheureusement un logiciel commercial, il ne peut donc être utilisé comme plateforme d'expérimentations.

Le deuxième, de Munzner [6] rassemble de nombreux avantages. Cependant ce logiciel est seulement dédié à la représentation d'arbres dans un plan hyperbolique. Munzner travaille maintenant sur le logiciel de David Auber,

celui ci rassemblant un plus grand nombre de fonctionnalités.

Le dernier, Tulip, créé par David Auber [3], semble correspondre le plus aux conditions requises par le développement de notre logiciel.

2.0.3 Tulip

Description générale

Tulip est un logiciel dédié à la visualisation de grands graphes (figure 2.2). Il permet de gérer des graphes contenant plus de 500.000 éléments sur une simple station de travail (PIII 600, 256 MO de ram) [1]. Les caractéristiques principales de Tulip sont les suivantes :

- la visualisation et la modification des graphes en 3D (ou 2D)
- le support de plug-ins pour une évolution facile
- la construction de clusters et la navigation à l'intérieur de ceux-ci
- le dessin automatique des graphes
- le clustering automatique des graphes
- la sélection automatique des éléments
- la coloration automatique du graphe en fonction de la valeur des éléments

Dans notre projet, nous utiliserons plus particulièrement les fonctions de navigations. Elles permettent à l'utilisateur de définir l'espace d'information dont il a besoin. Tulip permet donc de naviguer dans les graphes en utilisant les outils traditionnels de visualisation (déplacement, rotation, zoom). On peut aussi sélectionner les éléments et afficher leurs contenus.

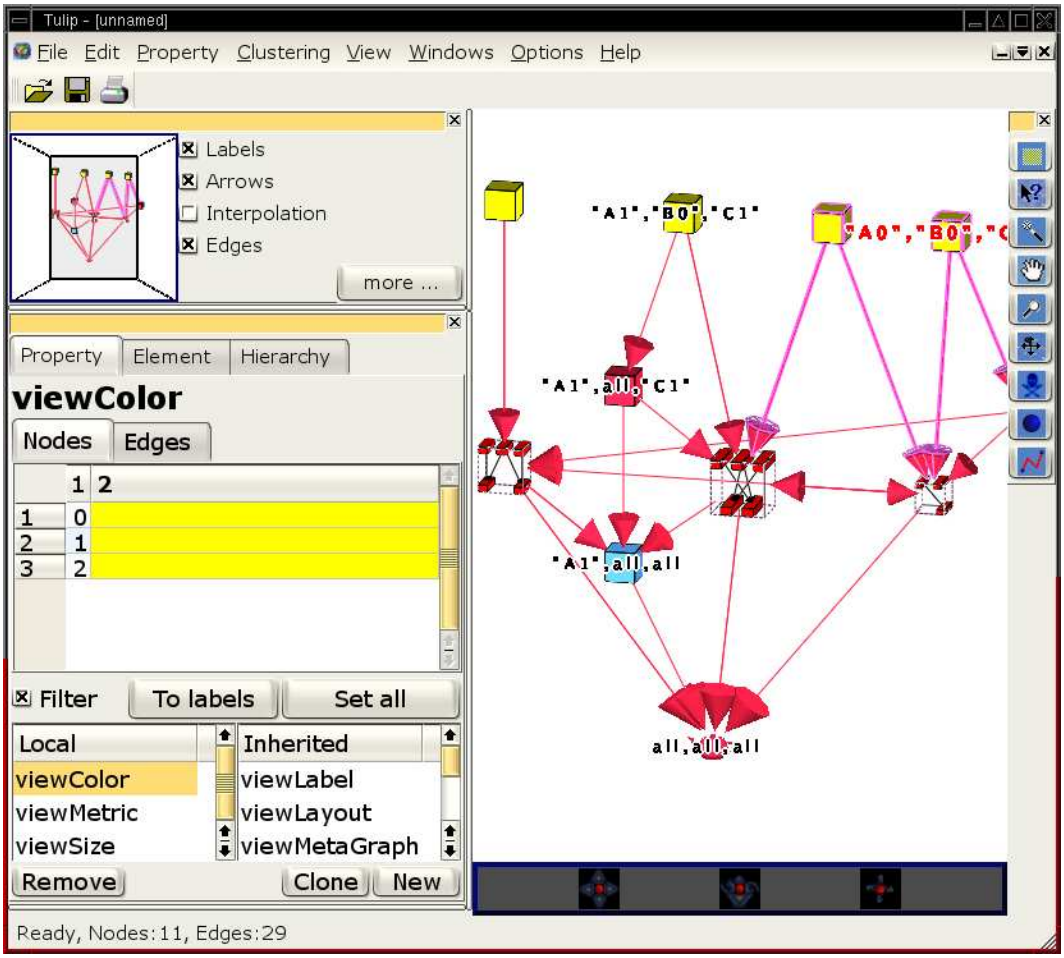
Le mécanisme de clustering permettra d'afficher une vue simplifiée du graphe en créant des groupes de sommets et d'arêtes. Les clusters peuvent être manuellement créés par l'utilisateur, ou construits automatiquement à l'aide d'algorithmes inclus dans Tulip.

Fonctionnement de Tulip

Le logiciel est écrit avec le langage C++. Il utilise la bibliothèque QT pour les interfaces graphiques et la bibliothèque OpenGL pour la visualisation en trois dimensions. Son architecture se décompose principalement en trois parties.

1. Le noyau est une librairie de manipulation de graphes. Celle-ci contient la structure du graphe, qui permet une optimisation de l'utilisation mémoire.
2. Un mécanisme permet l'intégration de différents plug-ins. Il est ainsi possible d'ajouter des algorithmes. Tulip se compose de cinq types de plug-ins. L'utilité des plug-ins d'importation, d'exportation, d'affichage de graphes est évidente. Le clustering permet de grouper plusieurs sommets en un seul et de gérer ces sous-graphes. Le plug-in glyph permet d'afficher chacun des sommets avec n'importe quel objet 3D.

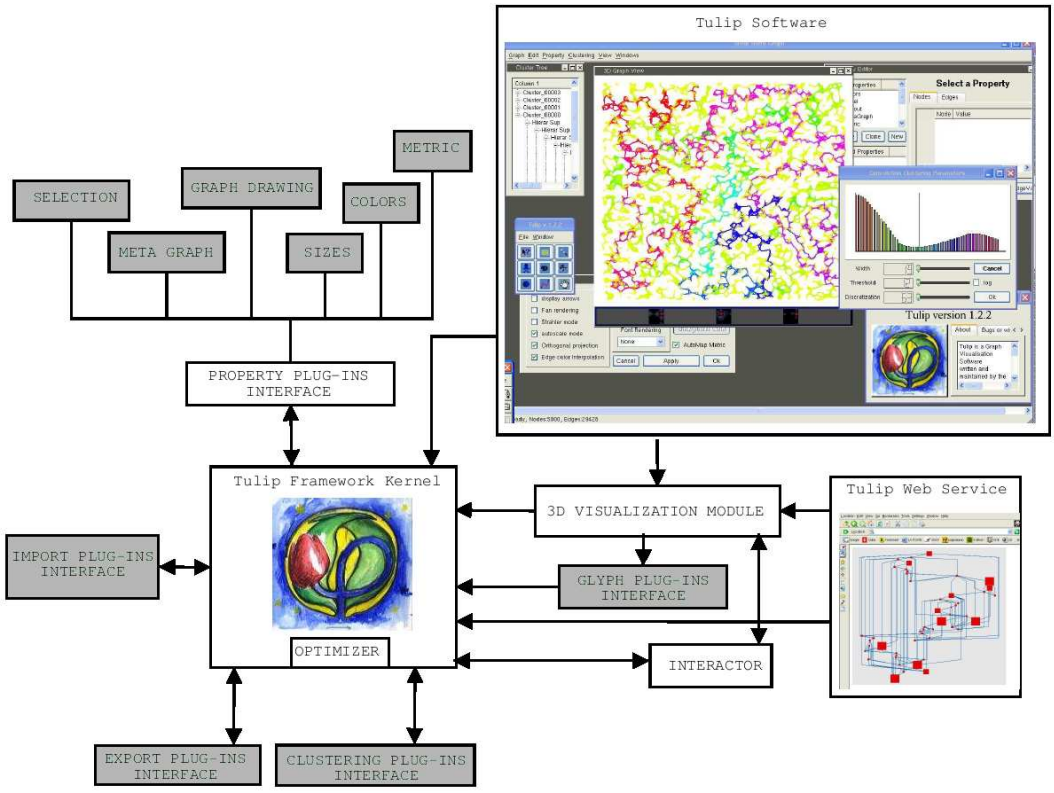
FIG. 2.2 – L'interface graphique du logiciel Tulip



3. Enfin le logiciel fournit une interface construite dynamiquement en fonction des plug-ins connectés au noyau de la plate-forme. L'interface permet de donner aux utilisateurs l'accès à toutes les extensions ajoutées, sans aucune intervention sur le logiciel lui-même.

La figure 2.3 présente une partie des différents modules composant cette plate-forme, les parties grisées représentent les points d'entrées pour l'ajout de nouveaux plug-ins.

FIG. 2.3 – L'architecture de Tulip



2.0.4 Langages de développement

Le langage C++

La gestion de graphes de taille importante nécessite une gestion mémoire optimisée. De plus, le temps de réponse de l'interface utilisateur doit être quasiment immédiat. Les algorithmes s'effectuant sur de larges structures de données, l'implémentation de ceux-ci doit se faire à l'aide d'un langage produisant du code s'exécutant rapidement. Enfin, le code devra être facilement réutilisable, ce qui nous amène à préférer un langage orienté objet. En effet,

l'approche orientée objet produit en général un code source plus clair, et une modularité meilleure que l'approche fonctionnelle.

Le C++ est un langage objet de haut niveau, dérivé du langage C [10]. Ce langage permet une gestion manuelle de la mémoire occupée par le programme. Cela permet d'optimiser l'utilisation de la mémoire. En contrepartie, le développement des programmes est plus complexe [9]. Le C++ est actuellement un des langages les plus rapides, en terme de vitesse d'exécution du code créé.

De plus, le code source de Tulip est en C++. Etant donné l'étroite relation entre le logiciel que nous devons concevoir et Tulip, il est préférable de l'implémenter en C++ pour des questions de compatibilité et afin d'éviter de réécrire du code existant déjà sous Tulip. Ainsi il sera aisé de reprendre les structures de graphes de Tulip, et toutes les fonctions les manipulant.

Afin que le logiciel soit complété, modifié ou amélioré avec de nouveaux algorithmes plus performants ou de nouveaux outils d'aide à l'analyse de bases de données, l'architecture du logiciel devra être pensée de manière évolutive. Ainsi, le code et les commentaires seront en anglais, et des conventions de codages seront données par le client. D'autre part, la documentation du code devra être effectuée avec D-oxygène

L'interface graphique

QT est une librairie de développement d'interface graphique [4]. Elle se base sur le langage C++, elle est donc orientée objet. Cette librairie est portable sous différents systèmes (Unix, Windows, Macintosh). Elle sert notamment de base à KDE, l'un des environnements graphiques de bureau les plus utilisé sur Linux. QT est protégé par la licence GPL. La création d'interface graphique QT peut être automatisée et facilitée à l'aide de QT Designer.

Nous allons donc utiliser QT, premièrement puisque c'est une demande du client, et deuxième car c'est un choix judicieux. En effet, Tulip utilise cette librairie et propose des plug-ins pour QT Designer, permettant de créer rapidement les interfaces génériques de visualisation de graphes. Utiliser QT pour créer notre interface permet de réutiliser les fonctions et les plug-ins déjà implémentés. L'intégration de notre logiciel avec Tulip sera donc facilitée.

Chapitre 3

Besoins non fonctionnels actualisés

3.0.5 Qualités globales

Les deux qualités essentielles de notre logiciel doivent être : une utilisation simple pour des personnes sans connaissances approfondies en informatique et une évolutivité logicielle, afin que des chercheurs en informatique puissent y greffer le fruit de leur recherche.

Cependant, la réalisation d'un logiciel d'exploration de cubes de données est un projet de taille importante (et ne peut s'envisager dans le cadre du projet de programmation du Master 1 d'Informatique). Notre logiciel est donc en réalité un prototype permettant d'étudier la faisabilité de l'exploration des cubes de données.

3.0.6 Domaine d'action

L'interface graphique du logiciel est prévu pour des chercheurs ou scientifiques, n'ayant pas nécessairement des connaissances approfondies en informatique. Un effort a donc été réalisé sur l'interface graphique, de manière à la plus simple possible. De plus, ce logiciel étant dédié à la recherche informatique, l'architecture du logiciel a été pensée de manière à pouvoir rajouter à ce prototypes divers outils de création, de manipulation ou de simplification de cube de données. De même le code source a été rendu lisible afin qu'il puisse être repris.

3.0.7 Temps de réponse

En raison d'une complexité exponentiel incontournable, le chargement du cube peut être (très) long, suivant la grandeur de la base de données d'origine. Le plug-in de visualisation emphHGDataCube appelant le plug-in emphHerarchical Graph de la librairie Tulip, a une exécution lente, due à l'algorithme implémenté dans emphHerarchical Graph. Ensuite, mis à part

l'algorithme de résumé de cube présentée dans cette étude, toutes les autres fonctionnalités du logiciel ont une complexité au pire linéaire. Ce qui signifie que l'interaction entre l'utilisateur et le programme sera assez rapide, si la base de données d'origine ne dépasse pas une certaine taille.

3.0.8 Portabilité

Afin que le logiciel puisse servir facilement aux utilisateurs de divers domaines, il devrait posséder une bonne portabilité. En raison de la variété des utilisateurs, leurs préférences de systèmes d'exploitation ne sont pas uniformes, donc le logiciel devrait être capable de fonctionner sous plusieurs systèmes. Cependant, comme ce logiciel est étroitement lié à Tulip, il ne peut fonctionner que sous les systèmes où Tulip fonctionne. D'autres parts, pour une raison de temps, les tests de portabilité du logiciel ont été restreints à Linux. Le fonctionnement du logiciel sous d'autres systèmes d'exploitations n'est donc pas garanti.

Chapitre 4

Besoins fonctionnels actualisés

Le projet se compose de trois parties, chacune étant d'une difficulté croissante. Les deux premières devaient être impérativement réalisées, elles ont donc été implémentées. La troisième (la réalisation du plug-in de clustering) devait être commencée mais non nécessairement finie. Nous n'avons malheureusement pas pu finir cette partie.

En général, nous avons respecté les fonctionnalités définies dans le mémoire intermédiaire. Cependant, cette étude faisant parti d'un sujet de recherche, les clients ont donc eu de nouvelles idées au cours du projet, et nous ont demandés s'il était possible de les appliquer.

4.0.9 Création des plug-ins de base

Le plug-in d'import

La première fonction du logiciel est de transformer la base de données de l'utilisateur sous forme *csv* en un cube de données. La création du cube a été développée dans un plug-in d'import Tulip.

Le cube est stocké à l'aide de la librairie Tulip. La base de données peut contenir un nombre aléatoire de champs et de lignes. Seul le dernier champ doit être un nombre réel. Les autres champs sont considérés comme des chaînes de caractères.

Le plug-in d'affichage

La seconde fonction affiche les sommets du cube de données de façon cohérente et lisible. Elle constitue le plug-in d'affichage ou plug-in de layout.

Le nombre d'éléments d'un cube de données peut être de l'ordre du million. Ainsi l'affichage d'un sommet occupe à l'écran en général un pixel. Chaque niveau du cube doit être clairement séparé à l'affichage. Ils sont donc positionnés sur des lignes différentes. Au sein du même niveau, les sommets ont été ordonnés d'une manière cohérente.

Un niveau du cube peut contenir aisément plusieurs milliers de sommets. Même avec un écran haute résolution, ces sommets ne peuvent être affichés sur une même ligne. Ils auraient donc pu être dessinés à l'écran sur plusieurs

lignes adjacentes. Cette fonctionnalité n'a pas été implémentée car durant le projet, nous avons douté de l'efficacité d'un tel plug-in. Nous avons donc abandonné son développement au profit du plug-in *Hierarchical Graph*, implémentant l'algorithme de Sugiyama.

Enfin, pour faire apparaître efficacement les valeurs de mesures, les couleurs des sommets et des arêtes dépendent de ces valeurs.

4.0.10 L'interface graphique

Le second point est la mise en place d'une interface graphique spécifique à la manipulation de cube de données. Cette interface s'appuie sur la librairie Tulip, mais est indépendante de la partie logicielle de Tulip. Les plug-ins créés précédemment sont donc directement réutilisables.

Le logiciel Tulip offre de nombreuses fonctionnalités de manipulation de graphes. Dans notre cas, seul quelques unes nous intéressent. De plus, certaines fonctions utiles à la visualisation de cube de données ne sont pas présentes. On a donc créé une nouvelle interface graphique, plus adaptée aux besoins.

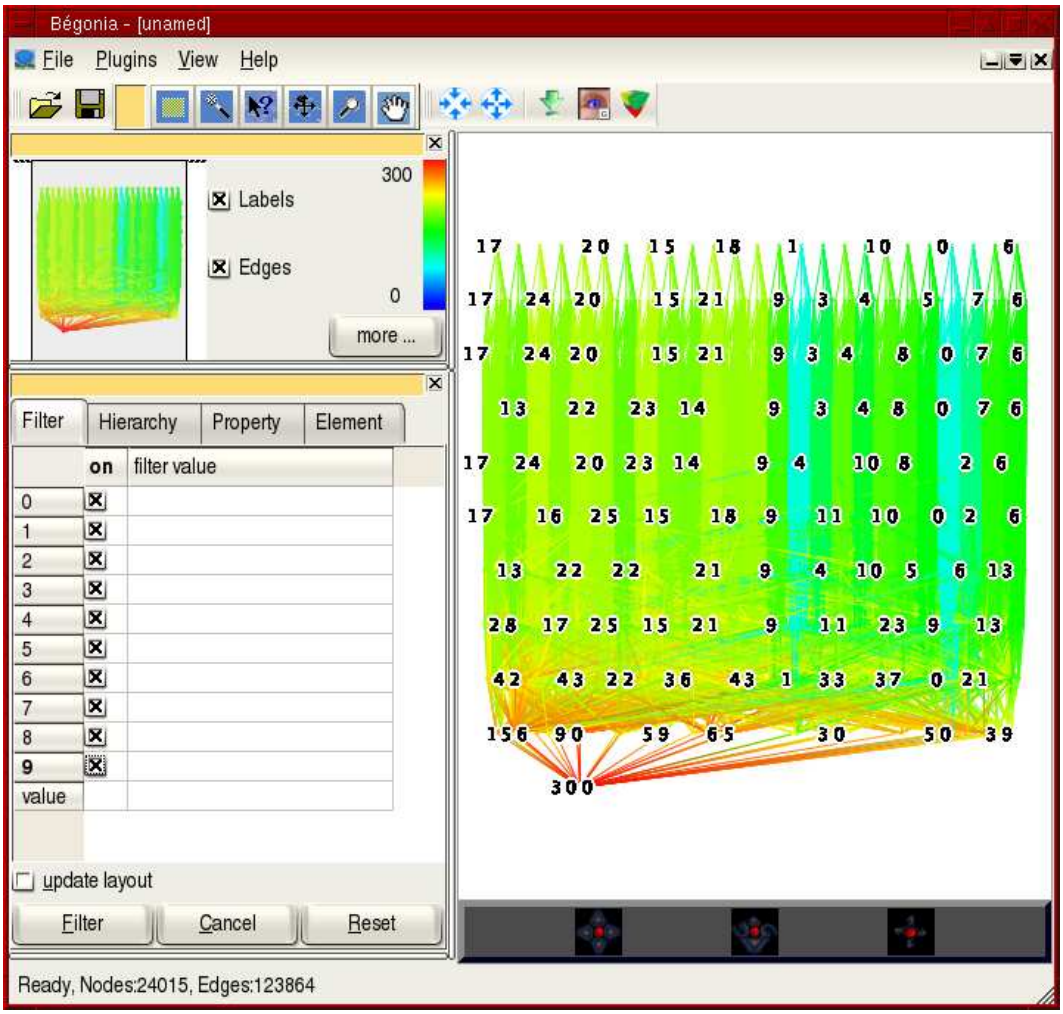
Notre interface propose les fonctionnalités suivantes

- Fonctions basiques de navigation dans un graphe : rotation, déplacement, zoom
- Sauvegarde et chargement du cube
- Fenêtre affichant une vue globale du graphe, et de la position actuelle dans celui-ci
- Affichage des informations des sommets se situant sous la souris
- Filtres des attributs de la base de données
- Filtres des valeurs d'un attribut de la base

Légende de l'interface graphique

1. Barre des menus.
   - menu File :
     Open
     Save
     Save As
     Exit
   - menu Plugins->Import
     Datacube (import un cube de données d'une base csv)
   - menu Plugins->Layout :
     DataCube
     HGDatacube
   - menu Plugins->Clustering :
     QuotientCube
   - menu View
     Center view
     Auto layout (applique un layout après avoir chargé un graphe)

FIG. 4.1 – L'interface



Dialogs (permet de réafficher des fenêtres cachées)
- menu Help
  About

2. Vue générale du graphe. Un cadre indique la partie du graphe affichée dans la fenêtre principale.
3. Echelle de couleurs utilisée pour représenter la valeur des sommets.
4. Fenêtre principale pour l'affichage et la manipulation du graphe.
5. Fenêtres permettant d'explorer le contenu du cube de données.
   Ces fenêtres ont été ajoutées durant le projet. Elles permettent de naviguer et d'explorer le graphe.
6. Fenêtre permettant d'appliquer des filtres aux graphes.
   Dans le tableau, l'ensemble des attributs de la base sont listés. Pour chaque attribut, on peut filtrer soit l'attribut dans son intégralité (boîte de sélection), soit avec un intervalle de valeurs (champ d'édition). De plus, nous avons vu au cours du projet, avec les clients, que l'on pourrait filtrer aussi en fonction de la valeur de mesure.
   Le bouton *filter* applique le filtre décrit par le tableau du dessus.
   Le bouton *cancel* annule la dernière modification du tableau.
   Le bouton *reset* efface tous les filtres décrit dans le tableau.
7. Barre d'affichage d'information : Affiche des informations sur le graphe en cours.
8. Barres d'outils :
   - Barre de chargement
     ouvrir un graphe enregistré
     enregistrer un graphe
   - Barre d'outils de manipulation du graphes :
     sélection
     obtention d'information d'un élément
     déplacement des éléments sélectionnés
     zoom (définition du cadre de zoom)
     navigation
     groupement de noeud en un méta-noeud
     dégroupement des méta-noeud sélectionnés

   Les fonctions de groupage / dégroupage ont été demandées par les clients après avoir rendu l'architecture du projet. Car l'utilisateur doit pouvoir modifier les groupements non judicieux effectués par le plug-in de clustering.
   - Barre de raccourci vers les plug-ins par défaut
     importer un cube de données d'une base csv
     appliquer un layout
     appliquer un clustering

   La barre d'outils de raccourci a été ajoutée afin d'avoir un accès plus rapide aux plug-ins.

4.0.11 Création de plug-ins de clustering

Une fois le plug-in d'affichage développé, avec des données censées, le cube de données n'est pas lisible. En effet, le graphe contient plusieurs millions de sommets avec de grandes bases de données. Un écran haute résolution peut afficher au maximum environ trois millions de pixels (2048 * 1600). La visualisation de l'intégralité des sommets est donc impossible. La mise en place d'algorithmes de simplification de graphes est donc indispensable.

Nous avons commencé à implémenter un algorithme résumant un cube. Plusieurs stratégies étaient possibles. Nous avons choisi la plus simple, c'est-à-dire regrouper en un même sommet, les sommets ayant la même valeur. Malheureusement nous n'avons pas pu finir cette fonctionnalité. Le plug-in fonctionne, mais dans certain cas, il résume trop le cube de données. Il met dans un méta-noeud des noeuds qui ne devraient pas en faire partie. De plus son intégration avec les filtres n'a pas été réalisée. Ils ne sont pas compatibles.

Les algorithmes de création de résumé de cube font encore partie du domaine de la recherche. Bruno Baruque, étudiant stagiaire au Labri et à l'Enseirb, développe actuellement un plug-in de clustering permettant de regrouper des noeuds appartenant à un ensemble de valeurs. Au moment même ou nous écrivons ces phrases, ce dernier nous a fait parvenir deux plug-ins de résumé de cube. Le premier regroupe les sommets de même valeurs. Le deuxième regroupe les sommets appartenant à un ensemble de valeurs (spanner). Nous n'avons pas eu le temps de les tester.

Chapitre 5

Architecture

Tout au long du projet, nous nous sommes appuyés sur l'architecture définie au début. Nous n'avons eu aucune modifications majeures à effectuer sur celle-ci. Ceci est dû en partie à la bonne conception de l'architecture de Tulip.

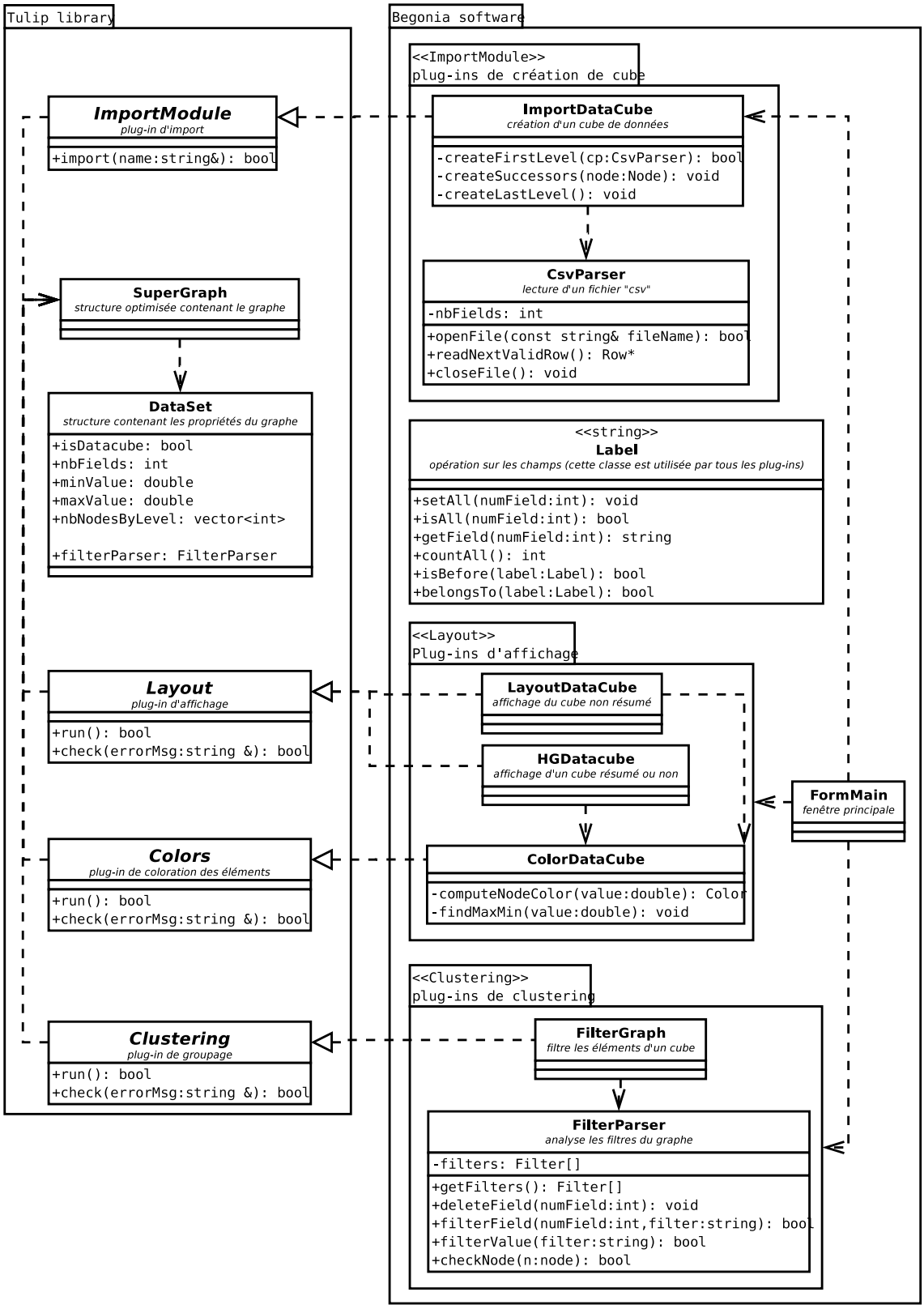
Les noms des classes ont été modifiés afin de suivre les conventions de codage de Tulip. Dans la partie plug-in la classe *Label* a été ajoutée. Cette classe est utilisée par tous les plug-ins. Au moment de la conception, nous n'avions pas pensé qu'elle serait si importante, car nous l'utilisions seulement au niveau du plug-in d'import. De plus, la coloration des éléments en fonction des valeurs de ceux-ci a finalement été implémentée dans un plug-in à part *ColorDataCube*. Ceci permet une meilleure modularité du code. Aussi les deux plug-ins de layout implémenté utilise le même plug-in de coloration.

Dans la partie interface graphique, des classes ont été ajoutées afin de faciliter l'exploration des éléments (*PropertyDialog*, *TulipElementProperties*) et l'exploration des sous-graphes (*ClusterTree*). Ces trois classes font parties de la librairie Tulip. Leurs intégrations ne posaient donc a priori aucun problème.

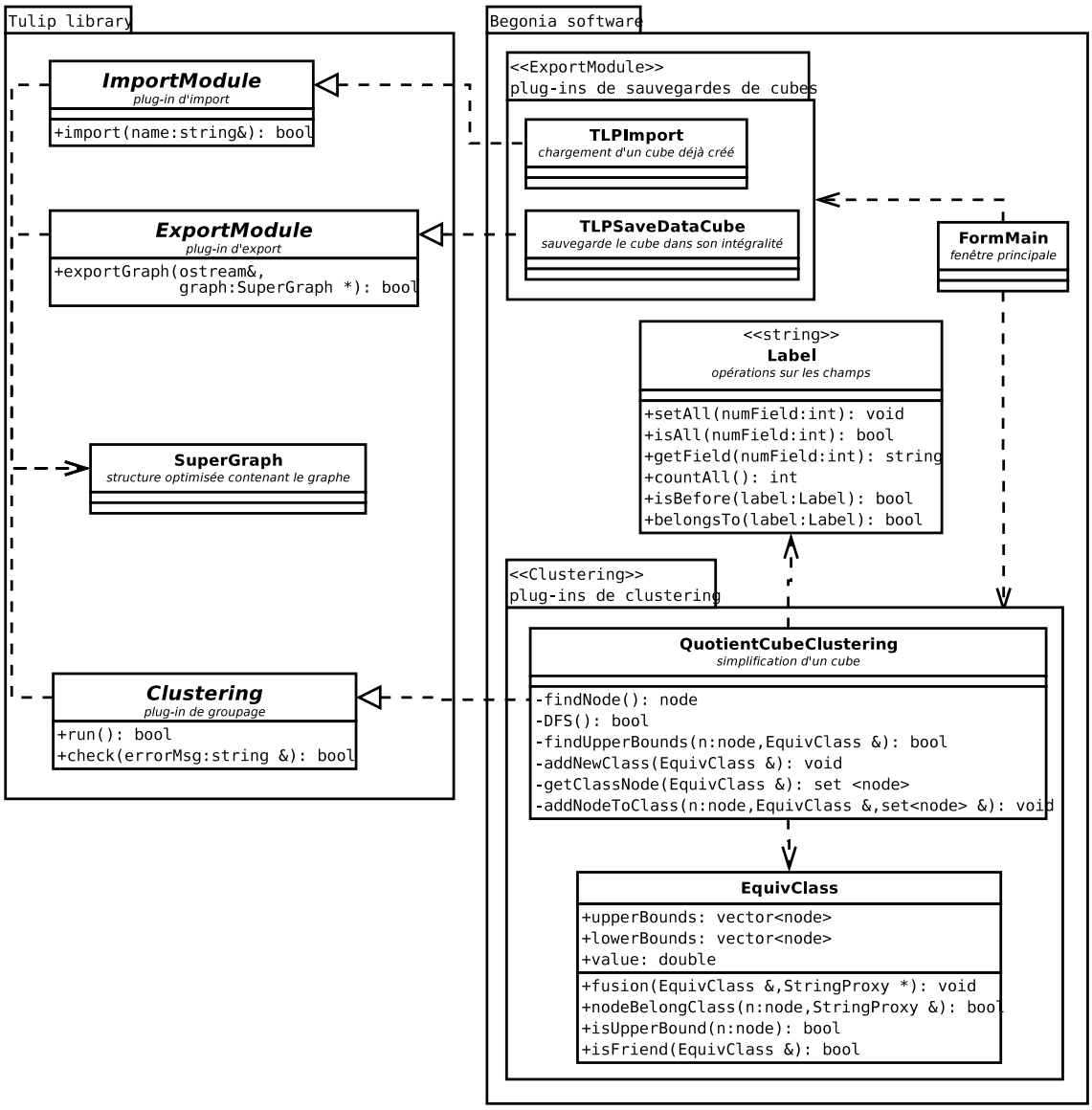
Le logiciel est divisé en deux parties principales : l'interface graphique et les plug-ins. Le dernier diagramme décrit les classes que nous n'avons pas finies d'implémenter.

5.1 Diagrammes de classes simplifié du logiciel

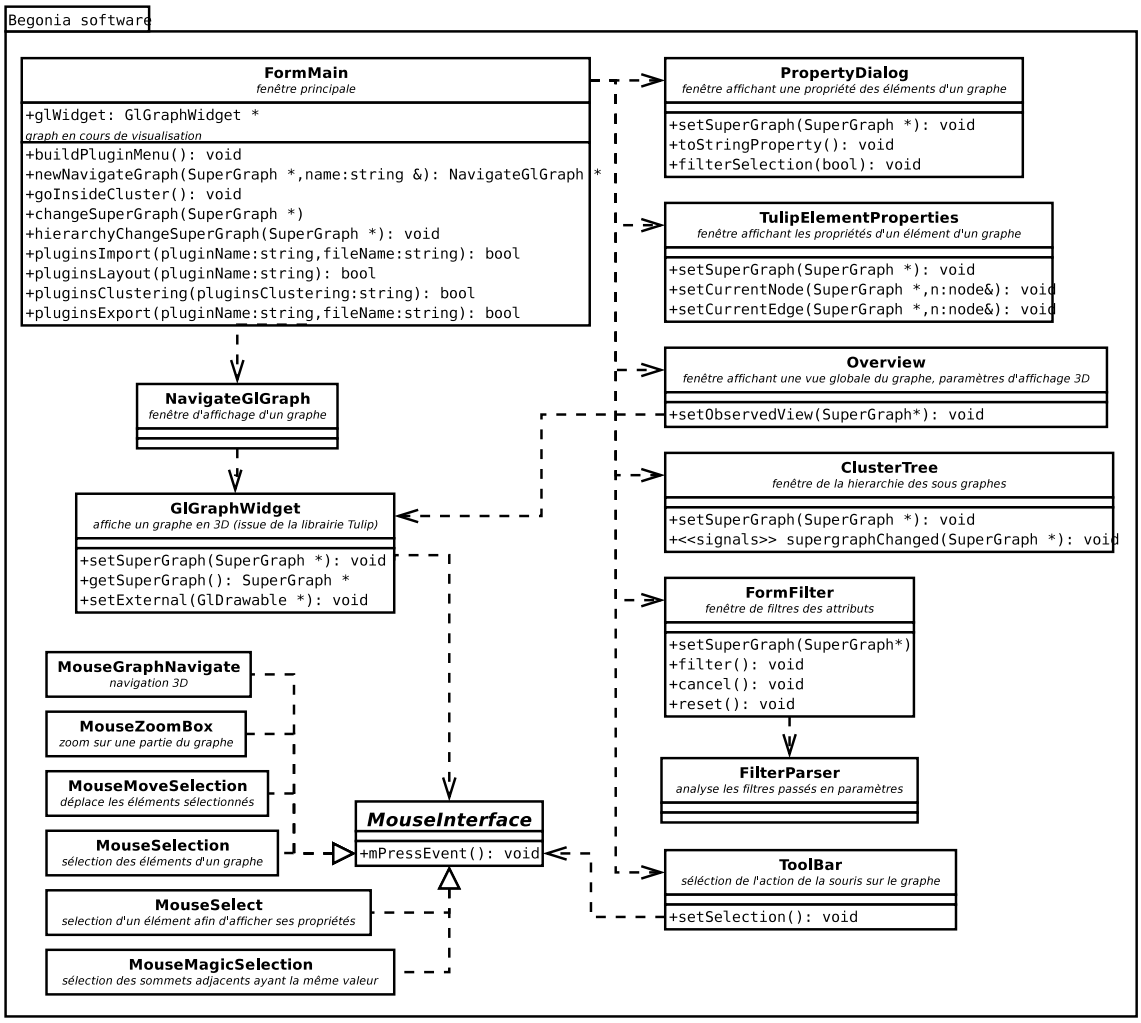
5.1.1 Plug-ins développés



5.1.2 Plug-ins à terminer



5.1.3 Interface Graphique



5.2 Légende des diagrammes

5.2.1 Description des plug-ins

Dans l'ensemble des classes suivantes, il existe des méthodes ou attributs privés. Ici, seules les méthodes publiques seront developées. En effet, les méthodes privées relèvent des choix utilisés pour l'implémentation de la classe, et n'ont pas leur place dans une architecture de logiciel. De plus, ces méthodes ne pouvant être utilisées hors de la classe, il est inutile de les expliciter.

Plug-in ImportDataCube

La seule méthode public de la cette classe (*import*) est une méthode héritée de la classe *Import* : il s'agit de la fonction d'exécution du plug-in.

Cette classe utilise le parser de fichier "CsvParser" pour lire le fichier csv, et la classe Label pour traiter la valeurs des tuples.

Conditions d'applications :

Pour que le plug-in fonctionne il lui faut passer en paramétre une base de données sous format csv. Le fichier doit être cohérent : il doit avoir le même nombre de dimensions pour chaque tuple, toutes les lignes doivent comporter une valeur de mesure placée après la dernière dimension. Si une ligne ne respecte pas ces conditions, elles sera considérée invalide et ne sera pas traitée pour la création du cube. Si deux lignes ou plus comportent les mêmes valeurs à chaque dimension, seule la première ligne sera prise en compte. En donnant le nom du fichier, il faut aussi préciser les séparateurs de textes et de champs utilisés dans le fichier. Par défaut le séparateur de texte est '"' et celui de champs ';'. Après la création du cube, les séparateurs utilisés dans le logiciel sont ceux par défaut, quels que soient ceux du fichier initial. Le dernier paramètre dont le plug-in a besoin est la fonction d'agrégation. Seules quatre sont disponibles : SUM, COUNT, MAX et MIN (respectivement : somme, nombre d'éléments, maximum et minimum). La fonction d'agrégation par défaut est SUM.

Classe CsvParser

Cette classe a pour seul but de faciliter la lecture du fichier csv par le plug-in d'import.

Méthodes publiques :

- `openFile` : Ouvre le fichier csv
- `closeFile` : Ferme le fichier csv
- `getNextValidRow` : renvoie un pointeur vers la prochaine ligne valide contenue dans le fichier csv (voir section précédente pour le contrôle de validité). La destruction de la ligne est à la charge du receveur. Renvoi `null` si on a atteint la fin du fichier. La ligne renvoyée est un objet de la classe `Row`. Cette classe est une extension de la classe `vector <string>` et comporte un attribut `value` qui contient la valeur de mesure du tuple et une méthode `toLabel` qui renvoie le tuple dans un objet de la classe `Label`.

Classe Label

Cette classe est une extension de la classe String. Cette classe sert à traiter les labels du cube de données. Les objets de cette classe doivent donc être d'un format particulier pour assurer un bon fonctionnement.

Un label est composé de plusieurs champs séparés par le caractère ','. Chaque champs est soit une valeur d'une dimension de la base et commence et fini alors par '"'. Un degré de liberté sur une dimension est marqué par la chaine de caractère 'all'

Méthodes Publiques :

- `setAll` : met le champ passé en paramètre à `all`
- `isAll` : teste si le champ passé en paramètre est à `all`
- `countAll` : compte le nombre de `all` dans le tuple

- `isBefore` : teste si l'objet, auquel est appliquée cette méthode, est avant le tuple passé en paramètre selon un certain ordre. Un label `l1` précède un label `l2` si au premier champ différent, `l1` est à `all` et pas `l2`. Si aucun de ces deux champs n'ait à `all`, c'est l'ordre lexicographique habituel qui s'exerce.
- `getField` : renvoie la chaîne de caractères du champ demandé en paramètre

Plug-in ColorDataCube

Ce plug-in permet de colorier un graphe suivant les valeurs des sommets. Il a pour seul méthode publique la méthode `run`, qui permet d'appliquer ce plug-in sur un objet de la classe `SuperGraph`. Le graphe sur lequel il est appliqué doit comporter deux attributs, `minValue` et `maxValue`, indiquant respectivement, les valeurs minimales et maximales des sommets du graphe. Ces valeurs doivent être contenues dans une propriété du graphe de type réel `MetricProxy` de nom `viewValue`. On peut indiquer dans les paramètres de ce plug-in si l'on souhaite une échelle de couleur linéaire ou logarithmique par rapport aux valeurs des sommets.

Plug-in LayoutDataCube

Ce plug-in permet la visualisation d'un cube de données de telle sorte que :

- tous les sommets d'un même niveau sont sur la même ligne
- les sommets d'un même ligne sont triés par ordre lexicographique sur la valeur des champs, excepté le champ `all` qui est le premier élément.

Ce plug-in a pour seule méthode : `run`, qui permet d'exécuter le plug-in. Ce plug-in doit être appliqué sur un cube de données avec des propriétés spécifiques.

- Pour deux noeuds `n1` et `n2`, $n1.id < n2.id$ <=> `n1` est sur un niveau inférieur ou égal à `n2`.
- Le graphe a une propriété `viewValue` de type `MetricProxy`
- Le graphe a une propriété `viewLabel` de type `MetricString`
- un attribut `nbNodesByLevel` indiquant le nombre de noeuds par niveaux
- un attribut `dataCube` ayant pour valeur `true`

Le cube de données doit aussi respecter les conditions d'applications du plug-in `ColorDataCube` qui permet la coloration du graphe et qui est appelé à la fin de ce plug-in.

Attention, la convention choisie ici est : un niveau `k` possède `k` degrés de liberté. Le niveau 0 comporte donc les tuples de la base et le niveau `n` (si la base contient `n` dimensions) le tuple (`all`,...,`all`)

Plug-in HGDataCube

Ce plug-in permet la visualisation d'un cube de données de telle sorte

qu'un minimum d'arêtes se croisent Il a pour seule méthode : `run`, qui permet d'exécuter le plug-in. Ce plug-in fait appel aux deux plug-ins suivants :

- `Herarchical Graph` : plug-in Tulip implémenté par D.Auber qui permet de visualiser un graphe de telle sorte qu'un minimum d'arêtes se croisent.
- `ColorDataCube` : expliqué dans le paragraphe précédent, ce plug-in permet la coloration du cube de données. Pour l'application de ce plug-in il faut donc que le cube de données respecte ses conditions d'applications.

Plug-in FilterGraph

Ce plug-in permet de filtrer des éléments du graphe suivant les labels des sommets. La seule méthode du plug-in, `run`, doit être appelée avec un graphe comportant un certain nombre de propriétés.

Tout d'abord le graphe doit comporter les propriétés `viewLabel` et `viewValue` respectivement de type `StringProxy` et `MetricProxy`. Ensuite il faut passer en paramètre de nom `filterParser` un objet de la classe `FilterParser` décrivant les différents filtres (voir section suivante).

Classe FilterParser

Cette classe permet de traiter facilement les descriptions des filtres que le plug-in `FilterGraph` doit appliquer. Elle contient en attribut privé la liste des filtres du graphe auxquels celle-ci est associée.

Méthodes publiques :

- `getFilters` : Récupère les filtres positionnés.
- `deleteField` : Supprime une dimension du cube de données.
- `filterField` : Ajoute un filtre à la liste des filtres pour la dimension passée en paramètre.
- `filterValue` : Ajoute un filtre sur la valeur de mesure.
- `checkNode` : Teste si un noeud vérifie les conditions des filtres de la liste.

Plug-in QuotientCube

Ce plug-in permet de résumer un cube de données. La méthode d'exécution est `run`. Pour que ce plug-in soit appliqué, le cube de données doit posséder les deux propriétés `viewLabel` et `viewValue` respectivement de type `StringProxy` et `MetricProxy`.

Ce plug-in va créer un sous-graphe qui sera le résumé et autant de sous-graphes que le cube résumé contient de classes. Ces sous-graphes contiennent les éléments des classes du résumé de cube.

Fichier marcros.h

Ce fichier contient toutes les constantes de chaînes de caractères utilisées pour nommer des objets tels que les plug-ins ou des attributs du cube de

données. Cela permet une harmonisation et une cohérence dans l'ensemble du logiciel, aussi bien sur les plug-ins que dans l'interface graphique.

5.2.2 Description de l'interface Graphique

FormMain

Squelette de l'application.

glWidget : Pointeur vers l'élément affichant le graphe en cours de visualisation. Cela permet de récupérer plus rapidement le graphe courant.

buildPluginMenu : Construit les menus en fonctions des plugins chargés en mémoire. Cependant, seuls les plug-ins utilisables sur les cubes de données seront affichés.

newNaviguateGraph : Crée une nouvelle fenêtre affichant un graphe.

goInsideCluster : Permet d'afficher le sous-graphe d'un méta-noeud

changeSuperGraph : Synchronise toutes les fenêtres lors d'un changement de graphe. Appelle la fonction *setSuperGraph* de toutes les fenêtres.

hierarchyChangeSuperGraph : Méthode appelée quand la fenêtre affichant la hiérarchie des sous-graphes d'un graphe, sélectionne un sous-graphe différent de celui en cours.

pluginsImport : Déclenche l'exécution d'un plug-in d'import. Si l'import réussi, on appelle la fonction *newNaviguateGraph* en y passant en paramètre le graphe créé.

pluginsLayout : Déclenche l'exécution d'un plug-in de layout sur le graphe courant. les fonctions *pluginsImport* et *pluginsExport* fonctionne de la même façon.

NaviguateGlGraph

Fenêtre affichant un graphe. Classe provenant du logiciel Tulip.

GlGraphWidget

Elément QT provenant de la librairie Tulip. Il permet d'afficher et de naviguer en trois dimensions dans un graphe.

setSuperGraph : Positionne le graphe à afficher. Cette fonction est présente dans toutes les fenêtres affichant des informations sur le graphe.

getSuperGraph : Récupère le graphe.

setExternal : Cette fonction est utilisée par la fenêtre d'emphoverview lorsqu'elle doit afficher la position de la vue courante par rapport à la vue générale du graphe.

FilterParser

Classe définie dans la partie des plug-ins.

FormFilter

Fenêtre permettant de modifier les paramètres de filtres d'un graphe. Cette fenêtre est utilisable que lorsque le graphe est un cube de données.

filter : Applique les dernières modifications faites dans la fenêtre au super-graphe.

cancel : Annule les dernières modifications faites.

reset : Repositionne le filtre à des valeurs par défaut.

Overview

Fenêtre visualisant l'intégralité du graphe et la position actuelle de zoom de la vue courante. Elle permet aussi de configurer les paramètres d'affichages du graphe en cours. Par exemple, l'affichage ou non des valeurs contenues par les sommets, ou l'affichage des arêtes... De plus si le graphe est un cube de données, on affiche les valeurs minimum et maximum du cube.

setObservedView : Associe la fenêtre avec la vue courante.

ToolBar

Sélection de l'action de la souris sur la vue en cours.

MouseInterface

Cette classe abstraite déclare les différentes interactions possibles entre l'utilisateur et la vue. Par exemple la méthode *mPressEvent* est appelée quand un clique de la souris survient dans la vue.

MouseGraphNaviguate

Implémente la classe MouseInterface. Elle permet de déplacer la caméra dans les trois dimensions.

ClusterTree

Elément QT provenant de la librairie Tulip. Cet élément permet d'afficher la hiérarchie des sous-graphes d'un graphe.

superGraphChanged : La classe envoie ce signal quand on choisit un sous-graphe différent de celui en cours.

TulipElementProperties

Elément QT provenant de la librairie Tulip. Cet élément permet de lister toutes les propriétés d'un élément du graphe.

setCurrentNode : Méthode appelée pour obtenir les propriétés d'un sommet.

setCurrentEdge : Méthode appelée pour obtenir les propriétés d'une arête.

PropertyDialog

Fenêtre affichant une seule propriété mais pour tous les éléments du graphe. Classe provenant du logiciel Tulip.

toStringProperty : Copie la propriété en cours de visualisation dans la propriété *viewLabel*. Cette dernière est utilisée par défaut pour afficher des propriétés dans la vue.

filterSelection : Liste seulement les éléments sélectionnés.

Chapitre 6

Description technique

6.1 Création d'un cube

Algorithme

- Entrée : base de donnée de dimension `n` sous format `csv`
- Sortie : cube de données

```
Algorithme :
Étape 1 : (Création du niveau 0)
Parcourir le fichier csv.
Pour chaque ligne lue faire :
    Créer un sommet s
    Mettre à s pour label, les valeurs des attributs de la ligne
    Mettre à s pour valeur de mesure, la dernière valeur de la ligne
    Stocker s dans une structure S1

Etape 2 : (Création des niveaux 1 à n-1)
    Créer une nouvelle structure vide S2
    Pour chaque sommet s de S1 faire :
     Pour chaque dimension i de s non à ‘all‘ faire :
            Créer un label l avec ‘all‘ pour valeur à la dimension i
            Si un sommet s’ de S2 a pour label l
                Créer une arête entre s et s’
                Mettre à jour la valeur de mesure de s’ avec celle de s
            Sinon
                Créer un sommet s’’
                Mettre à s’’ le label l
                Initialiser la valeur de mesure de s’’ grâce à celle de s
                Créer une arête entre s et s’’
                Ajouter s’’ à S2
Etape 3 :
Si S2 possède un unique sommet avec le label (‘all‘, ... , ‘all‘) s’arrêter
```

`Sinon reprendre à l'étape 2`

Précisions : La valeur `all` représente le degré de liberté du tuple

Structures de données

Dans notre implémentation le fichier csv doit être de la forme suivante :

- Chaque ligne représente un tuple de la base de données
- Une ligne est composée des valeurs des attributs d'un tuple suivies de la valeur de mesure de ce dernier

Les structures `S1` et `S2` utilisées dans l'algorithme ont été implémentées sous forme de tables de hachage. Elles stockent les sommets et ont les labels de ceux-ci pour clés. Un sommet sous Tulip est juste un entier. Ainsi les tables de hachage contiennent des entiers et n'occupent pas trop de places en mémoire. Le plus grand avantage des tables de hachage est sa rapidité en accès (complexité en $O(1)$). Ainsi, le grand nombre d'accès aux tables de hachage durant la création du cube, ne rallonge pas le temps de calcul.

Le cube de données est stocké dans un objet de la classe `SuperGraph` de la librairie Tulip. Cette classe permet de représenter de façon optimisée les graphes de grandes tailles. Les labels et les valeurs de mesures des sommets sont stockés comme des propriétés du graphe.

Explication techniques

Cet algorithme de création de cube a été implémenté sous la forme d'un plug-in d'import de Tulip.

Afin de gagner un peu de temps lors du chargement du cube, on peut traiter le dernier niveau de façon différente. On sait qu'un cube de données, d'une base de données de dimension `n`, aura `n+1` niveaux. Le dernier niveau est composé d'un unique sommet avec un tuple de degré de liberté égal à `n`. C'est-à-dire que ce sommet a pour label `(all,... ,all)`. Le parcours des sommets du niveaux `n-1` doit donc simplement servir à calculer la valeur de mesure du sommet du niveau `n`.

Au démarrage de ce plug-in d'import, plusieurs paramètres sont demandés. Outre le nom du fichier, sont demandés les séparateurs de textes et de champs utilisés dans le fichier `csv`. De cette manière, ce plug-in accepte un nombre plus important de base de données. Le dernier paramètre demandé, est la fonction d'agrégation avec laquelle on souhaite calculer le cube de données. Quatre ont été implémentée : `SUM`, `MAX`, `MIN` et `COUNT` (respectivement somme, maximum, minimum et nombre d'éléments). Une fois cette fonction choisie, l'utilisateur ne pourra plus la modifier.

Afin d'éviter des calculs sur le cube ultérieurement, divers paramètres sont enregistrés sur le graphe. Ainsi, on peut retrouver en temps constant, le nombre de champs de la base de données (ou le nombre de niveaux de cube), le nombre de sommets par niveaux, les valeurs minimales et maximales

des valeurs de mesure des sommets. Un attribut de type booléen indique également que le graphe est bien un cube de données.

Exemple précis du chargement du cube

Prenons la base de données de dimension 3 composée des trois tuples `\{ (A0, B0, C0)` ; `(A0, B0, C1)` ; `(A1, B0, C1) \}` (on ne s'occupe pas ici des valeurs de mesure)

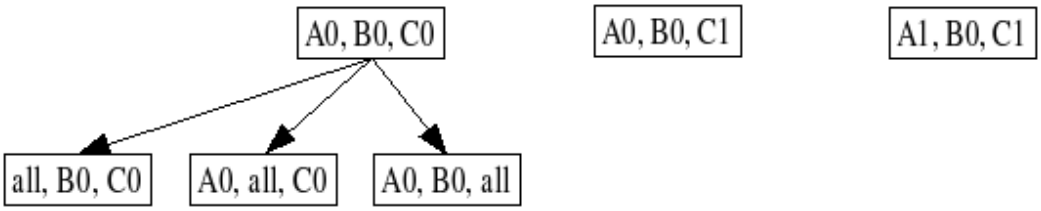
Le niveau 0 sera donc composé des sommets suivant :

```
s1 = (A0, B0, C0)
s2 = (A0, B0, C1)
s3 = (A1, B0, C1)
```

(On identifie ici un sommet, juste par son label)

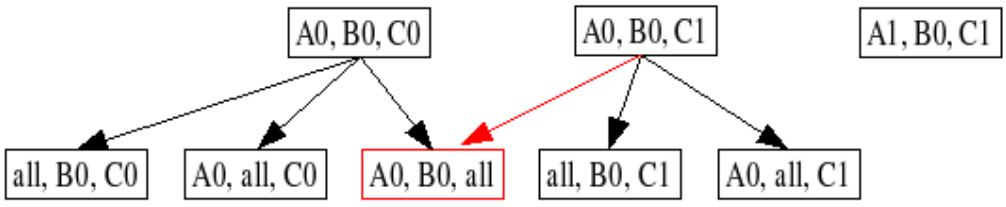
Pour la création du niveau 1, on va parcourir les sommets du niveau 0. Le sommet s1 va créer les sommets : `(all, B0, C0)` `(A0, all, C0)` `(A0, B0, all)`. On obtient le cube de la figure 6.1.

FIG. 6.1 – Etat du cube après le parcours des attributs de s1



Le sommet `s2` va créer deux nouveaux sommets : `(all, B0, C1)` `(A0, all, C1)`. `s2` devrait créer un sommet `(A0, B0, 'all')`, mais qui a déjà été créé par `s1`.Il suffit alors de rajouter une arête entre `s2` et le sommet `(A0, B0, all)` créé par `s1`. Le cube est alors comme en schéma 6.2.

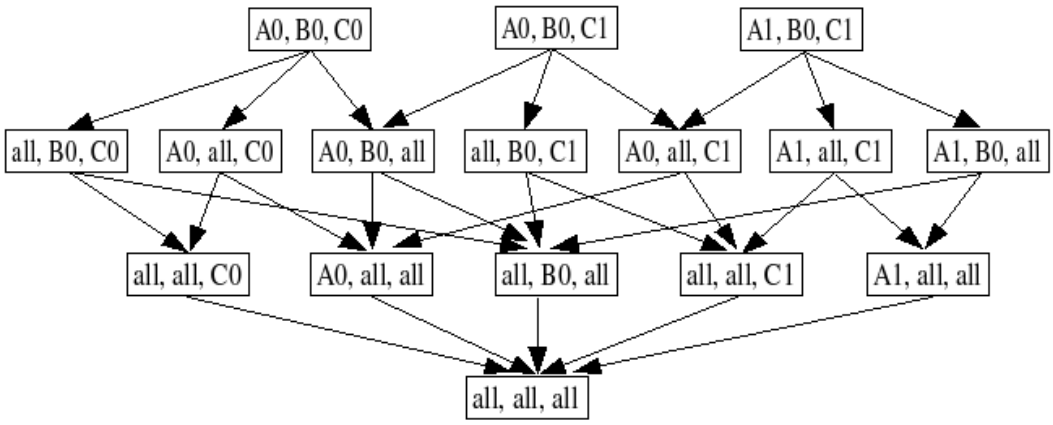
FIG. 6.2 – Etat du cube après le parcours des attributs de s2



Ensuite, de la même façon, le sommet `s3` va créer les deux sommets `(A1, all, C1)` `(A1, B0, all)` et une arête entre lui et le sommet `(all, B0, C1)` créé par `s1`

Et ainsi de suite pour les autres niveaux. On doit alors trouver le graphe de la figure 6.3

FIG. 6.3 – Etat final du cube



Complexité

Il est clair que le nombre de sommets créés est exponentiel par rapport au nombre de tuples dans la base de données. Tout algorithme de création de graphe a alors une complexité exponentielle. Ceci dit, afin de ne pas augmenter cette complexité, toutes les autres opérations réalisées lors de la création du graphe, doivent être de complexité minimale. Par exemple, le fait d'utiliser des tables de hachages permet de diviser la complexité par n (qui certes restent exponentielle, mais peut permettre de gagner quelques heures de calculs...)

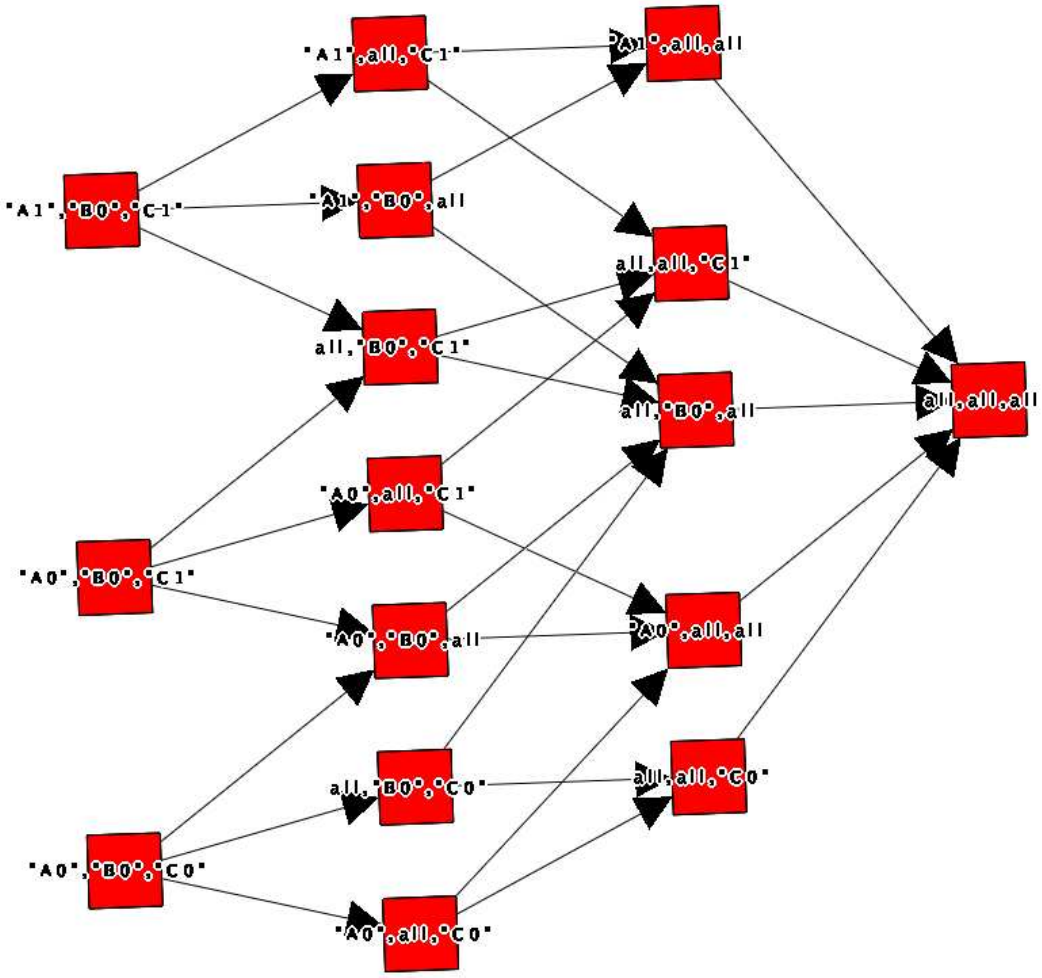
Test de validation

La première version implementée de ce plug-in a été testée et approuvée par les clients du projet, ce qui nous a permis de l'améliorer au point de vue de sa complexité.

Les différents tests que nous avons effectués ont porté sur des bases de données réelles ou non. La cohérence des cubes de données créés a pu être vérifiée sur de petites bases de données. Sur de grandes bases, les vérifications deviennent trop difficiles et coûteuses en temps. Voici l'exemple d'un test réalisé sous Tulip (l'interface graphique ne fonctionnait pas alors).

La base de données d'origine est celle présentée en section 6.1. L'affichage se fait alors avec le plug-in de layout de Tulip *hierarchical Graph*.

FIG. 6.4 – Cube importé avec le plug-in *ImportDataCube*



6.2 Affichage d'un cube

6.2.1 LayoutDatacube

Algorithme

1. Trouver le nombre maximum `nMax` de sommets contenu dans un niveau du cube
2. (a) Pour chaque niveau du cube : mettre les sommets dans un vecteur
   (b) Trier le vecteur suivant les labels des sommets
   (c) Parcourir le vecteur trié, donc dans l'ordre croissant :
   Pour chaque sommet `s`, calculer sa position en fonction de `nmax` et du nombre de sommets dans le niveau de `s`
3. Pour chaque sommet du cube calculer sa couleur en fonction de sa valeur de mesure

Précisions :

- Le tri suivant les labels (étape 2b) s'effectue comme suit :
  On parcours les différents attributs du label dans l'ordre (gauche à droite). Si l'attribut d'un label l1 est à *all* et celui d'un label l2 non, alors l1 précéde l2. Si les deux attributs de l1 et l2 pour une même dimension sont identiques on passe aux suivants. Si les deux attributs sont différents (mais non à *all*), on les trie suivant l'ordre lexicographique habituel.
- Le calcul de la position d'un sommet se fait comme suit :
  Connaissant, le nombre maximal de sommets sur un niveau, on dispose les sommets d'un même niveau sur une ligne de telle sorte que toutes les lignes de chaque niveau soient de la même taille. Puis entre chaque niveau on laisse suffisemment d'espace pour que la hauteur du cube soit égale à sa largeur, sur l'écran.
- Pour le calcul des couleurs, se référer à la sous-section 6.2.4.

Structures de données

Le cube de données est un objet de la classe SuperGraph définie dans la librairie Tulip. Différentes données sont alors présentes en tant que *propriétés* sur le graphe, comme la valeur de mesure de chaque sommet ou encore son label. Cela permet donc d'accéder directement à ces données à partir d'un sommet.

Le graphe contient aussi un attribut calculé lors de la création du cube : le nombre de noeuds par niveau. Ainsi l'étape 1 se fait en temps constant par rapport au nombre de sommets du graphe. Si cet attribut, n'avait pas été disponible avec le cube, il aurait fallu pour l'étape 1, parcourir de nouveau entièrement le graphe une autre fois, et pour chaque sommet vérifier en fonction de son label (et du nombre de *all* sur ce label) sur quel niveau le sommet est situé. Le traitement d'une chaîne de caractères n'étant pas immédiat (il faut regarder caractère par caractère), cette solution a été écartée,

pour favoriser un temps de calcul plus rapide au dépend de quelques entiers supplémentaires stockés en mémoire.

Commentaires techniques

Cet algorithme ainsi que la plupart des algorithmes du logiciel ont été implémentés sous forme de plug-ins Tulip, afin de bien les distinguer de l'interface graphique. Cet algorithme était plus particulièrement un plug-in de Layout. Il permet l'affichage des graphes sous Tulip.

Le tri de l'étape 2b se fait avec la fonction `sort()` de la librairie standard de C++. La complexité de ce tri est $O(n*\log n)$ mais peut aller jusqu'à $O(n^2)$ dans certaines situations rarissimes. Cependant, dans notre cas, il aurait été délicat d'utiliser la fonction `stable_sort()`. Elle propose un tri qui est dans tous les cas de complexité $O(n * \log n * \log n)$, de plus cette fonction a besoin de plus de mémoire. Comme nous pouvons avoir à trier des vecteurs de plusieurs milliers, voire de dizaines de milliers d'éléments, nous avons préféré la fonction `sort()`.

Le bon fonctionnement de ce plug-in nécessite l'existence de propriétés et d'attributs particuliers. Ceux-ci sont positionnés sur le cube de données à la création par le plug-in d'import. Lors de l'appel du plug-in de visualisation, on vérifie au préalable que le graphe contient les bonnes propriétés. Par exemple, un attribut nommé *DataCube*, de type booléen, précise s'il s'agit bien d'un cube de données. Si cet attribut est manquant ou s'il a pour valeur `false`, alors le plug-in stoppe et renvoi un message d'erreur. Ceci empêche l'exécution du plug-in sur des graphes non valides tels que le résumé de cube.

Le résumé de cube comporte des meta-noeuds et la notion de niveau n'existe plus. Ce plug-in ne peut donc fonctionner sur les résumés de cubes créés en section 6.3.

Complexité

1. $O(1)$: ceci est du au grand nombre de sommets contenus dans le graphe par rapport au nombre de niveau.
2. (a) $O(n)$
   (b) $O(n * \log n)$: voir paragraphe précédant.
   (c) $O(n)$
3. $O(n)$: voir sous-section 6.2.2

La complexité de cet algorithme est donc en $O(n * \log n)$.

6.2.2 HGDatacube

HGDatacube pour *Hierarchical Graph on a Data Cube*. Cet algorithme repose essentiellement sur le plug-in de layout *Hierarchical Graph* de Tulip. L'algorithme de ce dernier est basé sur celui de Sugiyama.

Algorithme

1. Algorithme de Sugiyama :
   Grâce à un balayage successif des niveaux du cube, un sommet est placé au barycentre de ces prédécesseurs ou successeurs d'un des deux niveaux adjacents.
2. Calcul des couleurs des sommets

Structures de données

Cet algorithme n'a besoin d'aucune structure particulière. Il faut juste s'assurer lors du calcul des couleurs que l'objet de la classe SuperGraph contenant le cube données possède bien les données nécessaires pour pouvoir appliquer le calcul des couleurs.

Commentaires

Tout comme le plug-ins de visualiation précédent, le calcul des couleurs se fait par l'appel d'un plug-in de *Colors* de Tulip, décrit plus loin.

Complexité

Ce plug-in applique le plug-in de couleurs pour cubes de données puis celui de Hierarchical Graph qui implémente l'algorithme de Sugiyama.

Le plug-in de couleurs étant de compléxité linéaire, la complexité de cet algorithme est donc la même que celle de l'algorithme de Sugiyama.

6.2.3 Tests et comparaisons des algorithmes

Dans un premier temps, seul le permier algorithme de visualisation a été implémenté (avec le calcul des couleurs). Il a été testé sur différents graphes de différentes tailles (de 20 à 60.000 sommets).

La visualisation indiquait que l'algorithme était correctement inplémenté. Cependant, sur de petits graphes, il ne paraissait guère intéressant. En effet, une des opérations que l'on souhaite faire pour fouiller les données, est de simplifier le cube de données, en regroupant des sommets connexes ayant des valeurs de mesures identiques en une même classe (voir section 6.3). Après l'application de ce premier plug-in, les classes n'apparaissent pas suffisamment. En appliquant le plug-in *Hierarchical Graph* de Tulip et ensuite notre plug-in de couleurs, la vision des classes est beaucoup plus efficace.

Le cube suivant (figure 6.5 et 6.6) est extrait d'une base de données contenant quatres champs :

"A0","B0","C0",2
"A0","B0","C1",2
"A1","B0","C1",3
"A1","B1","C0",2

FIG. 6.5 – Visualisation du cube avec le plug-in *LayoutDataCube* : les classes d'équivalence des sommets ne sont pas visibles

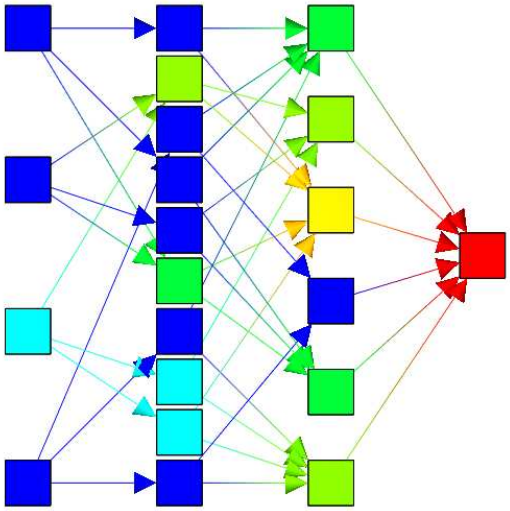
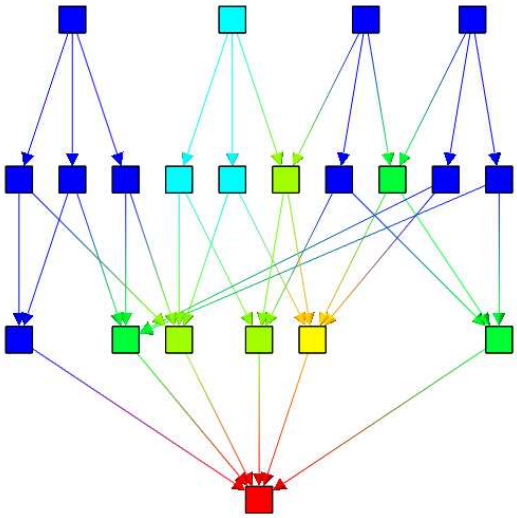


FIG. 6.6 – Visualisation du cube avec le plug-in *HGDataCube* : les classes d'équivalence des sommets sont immédiates

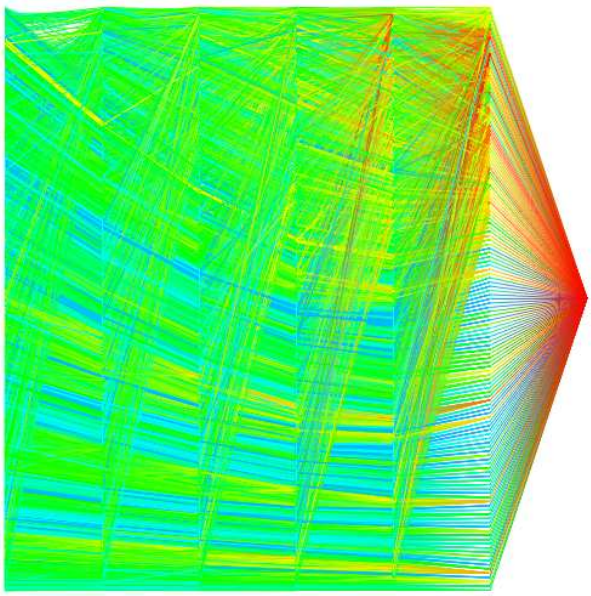


Le cube créé possède 21 sommets et arêtes.

Cependant, sur de plus grands graphes, la visualisation des classes est impossible, quel que soit l'algorithme. Ceci est dû au nombre trop important de sommets. Le plug-in *LayoutDataCube* devient donc ici plus intéressant dans la mesure où il est plus rapide que le plug-in *HGDataCube*. Et sur certaines grandes bases, il peut sembler que le premier permet une meilleure vision du cube.

Le cube suivant (figure 6.7 et 6.8) est extrait d'une base de données réelle d'archéologie. La base contenait 1846 tuples de dimension 6 (sans compter la valeur de mesure). Le cube créé comporte 13907 sommets et 52610 arêtes.

FIG. 6.7 – Visualisation du cube avec le plug-in *LayoutDataCube*

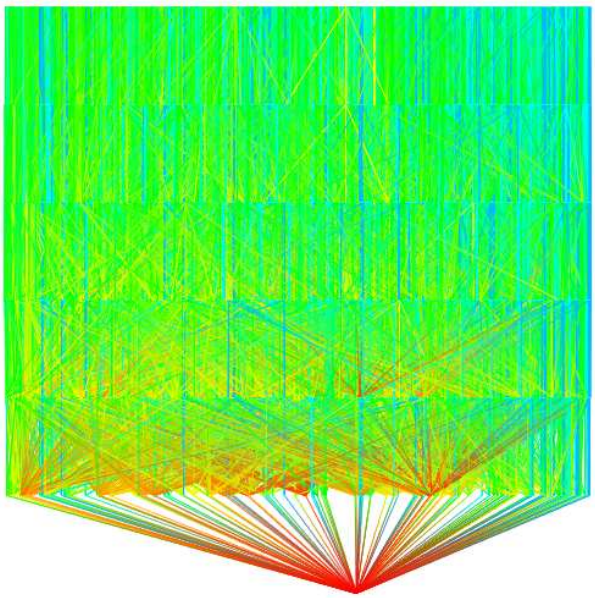


En résumé, voici les avantages et inconvénients des deux algorithmes :

- LayoutDataCube
  Avantages : Algorithme rapide sur des grands graphes ; permet une meilleure visualisation sur des grands graphes
  Inconvénients : Ne peut être appliqué qu'à un cube de données issues du plug-in d'import *ImportDataCube* (décrit en 6.1) ; ne permet pas de vision immédiate des classes d'équivalence sur de petits graphes. Ne peut-être appliqué à un *Quotient Cube* (voir section 6.3)
- HGDataCube
  Avantages : Algorithme applicable sur tous types de cubes ; permet la vision immédiate des classes sur de petits graphes
  Inconvénients : Algorithme très lent sur des grands graphes.

Nous laissons donc dans l'interface graphique les deux plug-ins de visualisation. L'utilisateur pourra choisir alors, suivant sa base de données, quelle

FIG. 6.8 – Visualisation du cube avec le plug-in *HGDataCube*



vision du cube il préfère.

6.2.4 Calcul de la couleurs des sommets

Algorithme

1. Stocker les valeurs de mesure minimales et maximales contenues dans le graphe.
2. Parcourir tous les sommets du graphe : pour chaque sommet calculer sa couleur de manière linéaire avec les valeurs minimales et maximales trouvées à l'étape 1.

Précisions : Si une échelle de couleur logarithmique est souhaitée : il suffit, si et seulement si toutes les valeurs de mesures du graphes sont strictement positives, de prendre au lieu de la valeur de mesure, son logarithme en base 10.

Commentaires

Les couleurs sont calculées sur une base de 1020 couleurs disponibles allant du bleu vers le rouge en passant par le vert, le jaune puis l'orange. Le bleu correspond aux valeurs minimales et le rouge aux maximales. Cette échelle de couleurs a été choisie de telle sorte qu'elle coïncide avec les couleurs proposées par Tulip.

Pour éviter de devoir parcourir deux fois l'ensemble des sommets du cube, les valeurs de mesures minimales et maximales sont passées en attribut dans

le graphe lors de la création du graphe 6.1, et peuvent donc être récupérées dans cet algorithme implémenté sous forme de plug-ins *colors* de Tulip.

Complexité

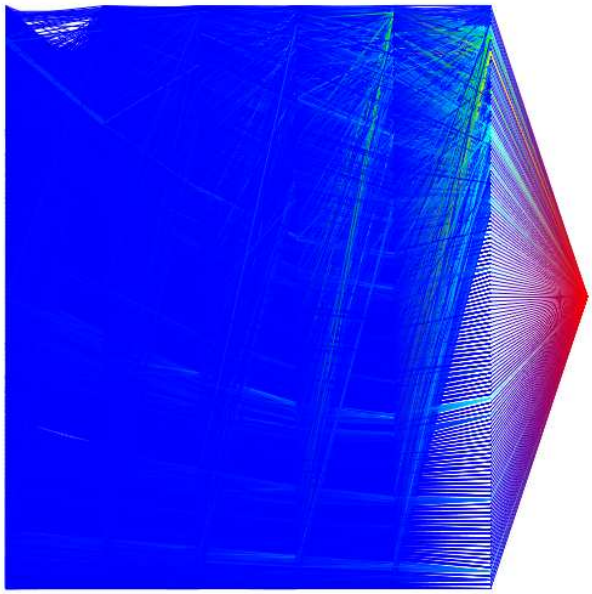
La complexité de l'algorithme est en $O(n)$, puisque tous les sommets du graphes sont parcourus une seule fois.

Tests de validations et de fonctionnement

Ce plug-in a été testé sur différentes bases de données réelles ou non. Avec l'échelle logarithmique des couleurs, cela permet une meilleure distinction des nuances au sein du cube, surtout pour les grands graphes.

Le cube de données suivant (figure 6.9 et 6.10) est issu de la base de données d'archéologie présentée dans la sous-section 6.2.3)

FIG. 6.9 – Visualisation du cube avec une échelle linéaire

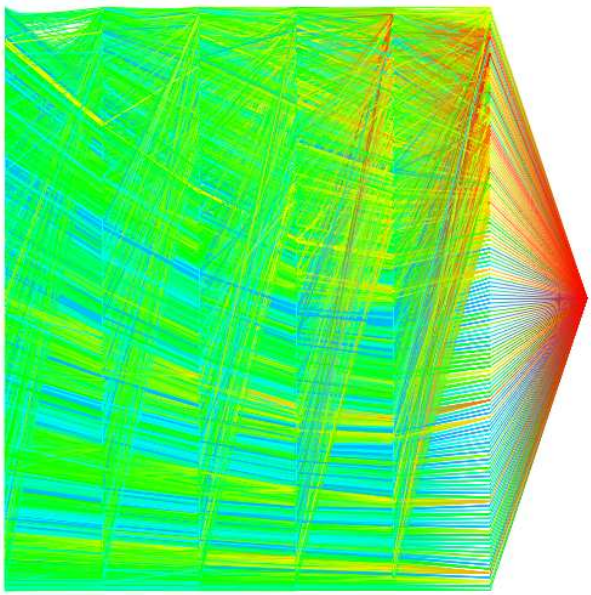


6.2.5 Applications des filtres

Algorithme

1. Déchiffrage des données de filtrages :
   enabledCond est un tableau de booléen indiquant si un filtre est posé pour une dimension
   filterCond est un tableau contenant pour chaque dimension, une opération logique si le filtre s'applique à certaines valeurs de la dimension, sinon rien.

FIG. 6.10 – Visualisation du cube avec une échelle logarithmique



2. `Parcourir tous les sommets du cube à filtrer`
   `Pour chaque sommet s de label l faire :`
   `pour chaque dimension i faire :`
   `si enabledCond[i] et filterCond[i] = NULL`
   `si l[i] = "all" garder s`
   `sinon supprimer s et passer au sommet suivant`
   `si enabledCond[i] et filterCond[i] != NULL`
   `appliquer filterCond[i] à l[i]`
   `incrémenter i;`
   `Passer au sommet suivant`

Précisions :

`filterCond[i]` contient un opérateur logique et une valeur à laquelle il faut comparer `l[i]`. Si `l[i]` respecte la condition, le sommet `s` est à supprimer, sinon il faut garder `s` dans le graphe filtrer.

Structures de données

Dans l'implémentation de cet algorithme, `enabledCond` et `filterCond` sont en fait des vectors. Les formules logiques contenues dans filterCond sont des objets de la classe `FilterParser`. Cette classe comprend quatre champs :

- op : indique l'opérateur de la formule logique
- val : contient la valeur réelle à comparer si celle-ci est un nombre
- field : contient la valeur en chaîne de caractères à comparer
- type : contient une valeur indiquant les différents cas possibles :
  - Pas de filtres

- Filtre sur toute la dimension
- Filtre avec une valeur réelle
- Filtre avec une valeur en chaîne de caractères
- Filtre sur la valeur de mesure

les deux champs `val` et `fields` sont nécessaires pour éviter de devoir éventuellement traduire pour chaque sommet, la valeur en chaîne de caractères en un réel.

Commentaires

L'application des filtres est étroitement liée avec l'interface graphique. En effet, c'est l'interface graphique qui va recevoir les différents filtres. C'est elle aussi qui va déchiffrer les formules et créer l'objet `FilterParser`. Ensuite le plug-in de clustering *FilterGraph* sera appelé afin d'appliquer le filtre au cube de données.

Le plug-in *FilterGraph* va créer un sous-graphe clone du graphe principal, et travaillera sur ce sous-graphe, ainsi le cube initial est conservé.

Afin de conserver une certaine cohérence avec les filtres, chaque sous-graphe possède une instance particulière de la classe `FilterParser`.

Dans la librairie Tulip, un sous-graphe n'est pas une copie parfaite du graphe principal, mais juste une vue sur ce dernier. Ainsi, il n'y a aucune duplication de sommets ou d'arêtes. Ceci est intéressant dans notre cas, car on manipule des graphes contenant plusieurs (dizaines de) millions d'éléments.

Complexité

n représente le nombre de sommets dans le graphe à filtrer

1. $O(1)$: ses opérations sont constantes par rapport au nombre de sommets du graphe
2. $O(n)$: les opérations a effectuer sur un sommet sont de complexités constantes par rapport au nombre de sommets du graphe.

La complexité de cet algorithme est donc en $O(n)$

Tests de validations et de fonctionnement

6.3 Création d'un pseudo *Quotient Cube*

Afin de simplifier les cubes de données créés, un algorithme de résumé de cube pouvait être implémenté dans le cadre de notre projet. Celui que les clients du projet avait proposé était celui du *Quotient Cube*, décrit dans l'article de Laks V.S.Lakshmanan, Jian Pei et Jiawei Han [8].

Un plug-in de clustering a donc été implémenté. Toutefois, il ne respecte pas entièrement l'algorithme proposé. En effet l'algorithme implémenté résume "trop" le cube de données et peut provoquer certaines incohérences

sur la structure du cube. Le quotient cube doit respecter les liens de parentés successeurs-prédécesseurs entre les sommets. Notre algorithme, décrit ci-dessus, ne respecterait pas ces liens de parentés.

L'idée de l'algorithme du quotient cube est de créer des classes de sommets, appelées classes d'équivalence, où tous les sommets d'une classe doivent posséder la même valeur de mesure et doivent provenir des mêmes tuples de la base d'origine.

Dans l'algorithme que nous avons implémenté, l'idée est de regrouper en une même classe tous les sommets connexes ayant une même valeur de mesure, sans s'occuper de la valeur des attributs de chaque sommets. L'autre grande différence entre les deux algorithmes fait que celui du quotient cube part de la base initiale et non pas d'un cube de données déjà créé. Notre algorithme s'applique quant à lui, sur un cube de données, et non pas sur la base de données initiales.

Algorithme

Algorithme général :

```
Etape 1 : Rechercher le sommet s du cube contenant tous ses
          champs à all
Etape 2 : Appliquer la fonction DFS (Depth-First Search) sur le
          sommet s trouvé à l'étape 1
Etape 3 : Parcourir l'ensemble des classes créées et construire
          sur le cube, les meta-noeuds correspondants
```

Fonction DFS : (sur un sommet s)

```
Etape 1 : Marquer s comme un sommet visité
Etape 2 : Créer une classe c avec comme borne inférieure s
          (et pas de bornes supérieures)
Etape 3 : Parcourir tous les prédécesseurs de s
          Pour chaque prédecesseur p,
            si les valeurs de mesure de p et s sont différentes
              si p n'a pas encore été visité, appliquer DFS à p
              sinon ne rien faire
            sinon rechercher la borne supérieure de p
Etape 4 : Si c n'a pas de borne supérieure, détruire la classe
          Sinon parcourir l'ensemble E des classes déjà créées.
             si il existe une classe c' appartenant à E qui
                possède au moins une borne supérieure identique
                à celles de c,
                    alors fusionner c' et c
             sinon ajouter c à E
```

Recherche des bornes supérieures : (d'un sommet s)

```
Etape 1 - Marquer s comme un sommet visité
```

```
Etape 2 - Parcourir tous les prédécesseurs de s
          Pour chaque prédécesseur p,
              si les valeurs de mesure de p et s sont identiques
                rechercher la borne supérieure de p
Etape 3 - Si s n'a pas de prédécesseurs ou que ceux-ci n'ont pas
          la même valeur de mesure que s, alors s est une borne
          supérieure.
```

Structures de données

Une classe `EquivClass` a été créée pour décrire une classe de sommet. `EquivClass` posséde trois attributs : deux vecteurs comportant les bornes supérieures et inférieures ainsi qu'un réel indiquant la valeur de mesure de la classe. Les classes trouvées sont ensuites stockées dans un vecteur. Afin de savoir si un noeuds a été visité ou non, une nouvelle propriété a été créer sur le graphe, de type booléen. Cette propriété est en fait une nouvelle `PropertyProxy` de type `SelectionProxy` créée grâce à la librairie Tulip. Cela permet une optimisation mémoire, puisque, Tulip a été créé de telle sorte.

Commentaires

Contrairement à l'algorithme du Quotient Cube, le choix a été fait ici de fusionner les classes au fur et à mesure qu'elles sont créées, et ceci dans un esprit d'économie de mémoire. C'est pour la même raison que seules les bornes des classes sont mémorisées. Lors de la création des méta-noeuds, il suffit de repartir des bornes inférieures des classes, et grâce au cube de données déjà créées, on peut rapidement retrouver tous les noeuds d'une classe.

Tests de validations et de fonctionnement

Les différents tests effectués sur de petites bases de données non réelles, ainsi qu'une confrontation avec les clients du projet ont montrés que l'implémentation réalisée ne correspondait pas à l'algorithme du Quotient Cube désiré mais à l'algorithme explicité en section 6.3. Voici tout de même quelques exemples de résumés de cubes réalisés avec notre algorithme.

Exemple 1 : Le résumé du cube de données (figure 6.12) est issu de la base de données de quatre tuples, expliquée en 6.2.3. La fonction d'agrégation est la fonction somme.

Le cube initial (figure 6.11) comporte 21 sommets et le résumé 10. On obtient donc une réduction de 52%.

Exemple 2 : Le résumé du cube de données (figures 6.14 et 6.15) est issu de la base de données d'archéologie (voir section 6.2.3). La fonction d'agrégation est la fonction somme. Le cube initial (figure 6.13) comporte 13907 sommets et le résumé 2651. On obtient donc une réduction de 81%.

FIG. 6.11 – Cube Initial

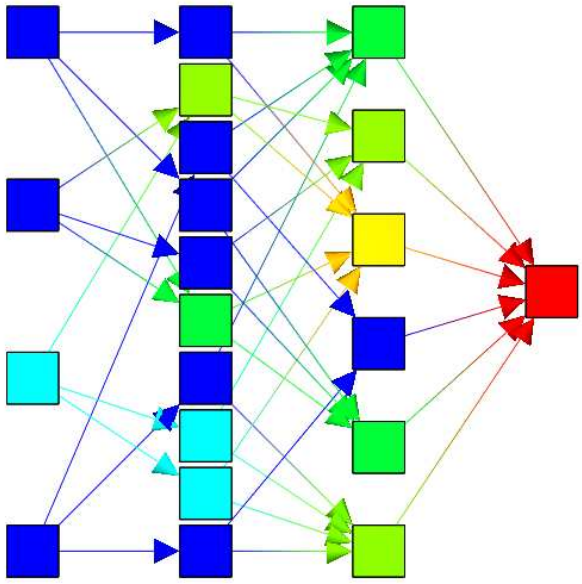


FIG. 6.12 – Cube résumé

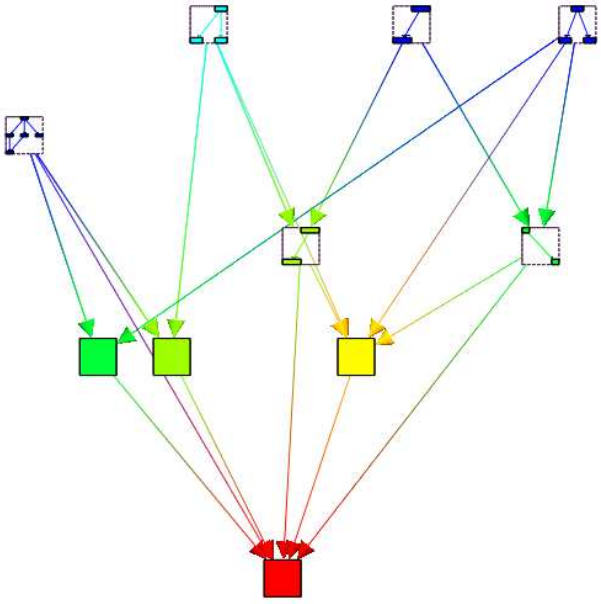


FIG. 6.13 – Cube Initial

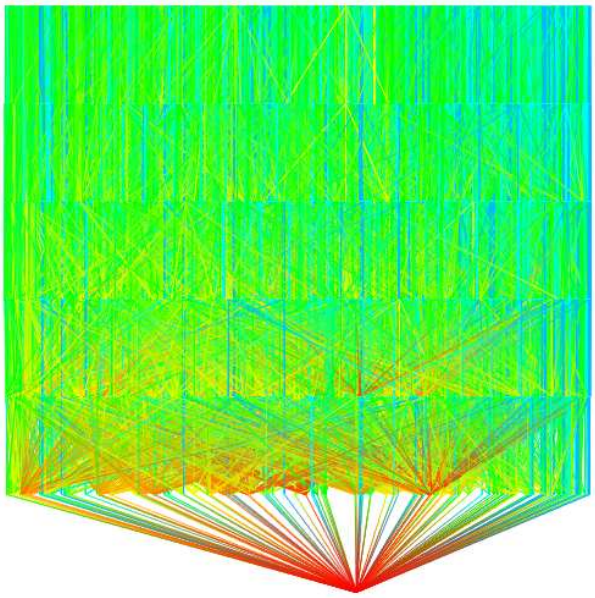


FIG. 6.14 – Cube résumé, avec les méta-noeuds placés au barycentre des sommets de la classe

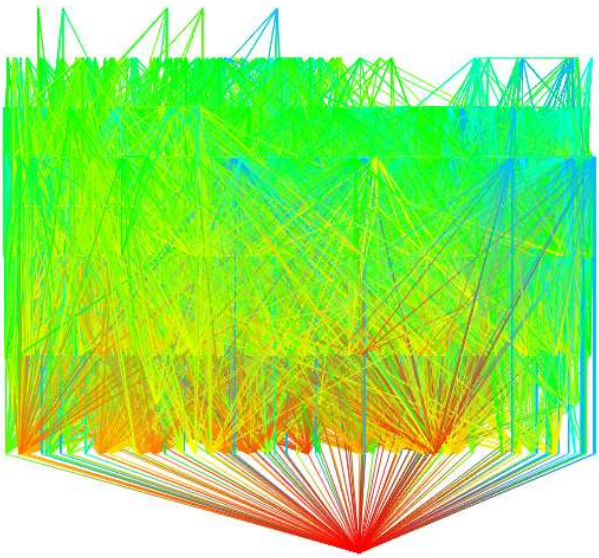
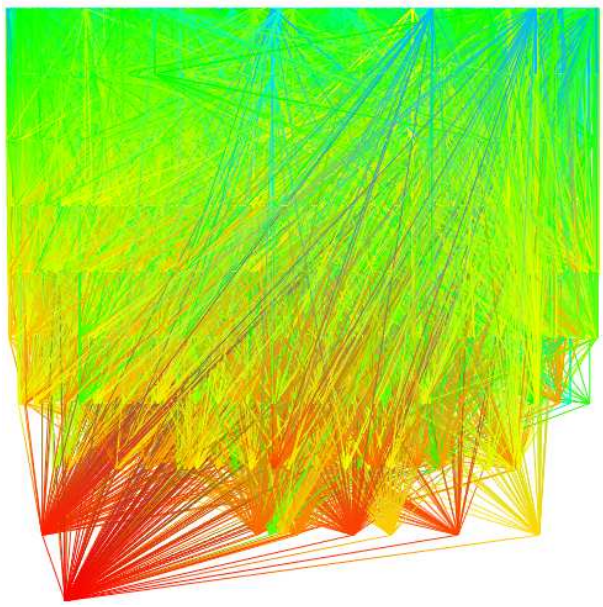


FIG. 6.15 – Cube résumé, avec l'affichage du plug-in *HGDataCube*



La conclusion que l'on peut tout de même porter est que l'algorithme `Quotient Cube` ne pourra avoir une meilleure réduction que cet algorithme.

Pour avoir une idée des réductions que pourrait effectuer l'algorithme du *Quotient Cube*, voici quelques résultats obtenus avec notre implémentation. On peut lire dans le tableau 6.16, les réductions sur le nombre de sommets, par rapport aux fonctions d'agrégations (*SUM*, *COUNT*, *MIN* et *MAX*). La plupart des bases de données initiales ont été construites de manière aléatoire uniforme.

FIG. 6.16 – Résultats des tests du résumé de cube

Nombre de noeuds du Cube	Résumé « SUM »		Résumé « COUNT »		Résumé « MIN »		Résumé « MAX »	
	Nb Noeuds	Réduction	Nb Noeuds	Réduction	Nb Noeuds	Réduction	Nb Noeuds	Réduction
21	10	52%	14	33%	4	81%	2	90%
74	18	76%	25	66%	6	92%	6	92%
108	12	89%	24	78%	3	97%	3	97%
13907	2651	81%	-	-	-	-	-	-
24015	3070	87%	8157	66%	25	99%	25	99%
67240	6719	90%	14885	78%	116	99%	352	99%

6.4 L'interface graphique

6.4.1 Structures de données

L'interface graphique développée n'a pas eu recours à des structures de données spéciales. En effet la quantité de données gérées par l'interface est assez faible. Par exemple, le nombre de fenêtres ou le nombre de champs à filtrer ne sera pas, en général, supérieur à vingt. Ainsi le stockage de ces listes se fait dans des structures de type `vector`.

Seul l'affichage de la liste des éléments du graphe pourrait demander des optimisations. Cependant la librairie Tulip propose déjà une classe *QT* optimisée pour afficher ces éléments : `PropertyDialog`.

6.4.2 Commentaires techniques

Développement de Bégonia

La création de l'interface graphique s'est fortement appuyée sur le logiciel Tulip existant. Cependant, comme il fallait développer une interface plus simple que Tulip, nous avons d'abord commencé par prendre seulement quelques lignes de code de ce dernier. Le fait de prendre juste une certaine partie du code nécessite de comprendre parfaitement ce qu'il fait. Autrement des bugs peuvent apparaître.

Au cours du projet à chaque nouvelle fonctionnalité, nous avons copié le fonctionnement de Tulip. Ce qui nous a paru comme une recopie des sources du logiciel Tulip. Cependant, une interface simple d'utilisation était une demande importante des clients, il était donc nécessaire de passer par cette étape.

Le design pattern Observer

Nous avons utilisé le design pattern observé / observeurs. Il se résume en une phrase. Quand l'observé modifie son état, il envoit un signal à tous les observateurs qui lui ont demandé d'être avertis des modifications.

Dans le cas de Tulip, l'observé est par exemple un graphe, l'observateur est la fenêtre affichant le graphe. Si un plug-in ajoute un sommet au graphe, la fenêtre en sera donc avertie, et elle affichera immédiatement le nouveau sommet. Lorsque l'on applique un layout à un graphe, à chaque modification, un signal est envoyé à la fenêtre. Dans le cas du layout le nombre de modifications est souvent supérieur au nombre d'éléments du graphe. Si l'on ne bloque pas toutes ces signaux, le temps pour calculer le layout sera beaucoup plus long. c'est pourquoi avant de lancer les plug-ins il faut mieux appeler la méthode static `Observable::holdObservers()`. Celle-ci retient tous les signaux envoyés par les observés. Les signaux seront relachés à l'appel de `Observable::unholdObservers()`

L'affichage de plusieurs fenêtres

La visualisation de plusieurs graphes dans différentes fenêtres n'était pas une demande du client. Cependant l'ajout de cette fonctionalité ne demandait pas un temps de développement important. En effet, la synchronisation de la vue avec les fenêtres affichant les informations sur le graphe en cours était obligatoire, car lorsque l'on passe du graphe racine vers un de ses sous-graphes, toutes les informations affichées doivent être modifiées. Le passage d'une fenêtre affichant un graphe vers une autre utilise le même mécanisme de synchronisation.

La classe FormFilter

Cette classe permet d'appliquer des filtres sur un cube de données non résumé.

Si un graphe de type cube de données non résumé est affiché dans Bégonia, un filtre par défaut est automatiquement créé. La sauvegarde des filtres se base sur les attributs du graphe.

Lorsque l'on applique les filtres (bouton *filter*), on crée une nouvelle instance de FilterParser et on passe à cet objet tous les filtres demandés par l'utilisateur. Le nom du nouveau graphe est calculé de telle sorte qu'il soit unique dans les sous-graphes du graphe filtré. Si les filtres sont corrects on lance le plug-in de filtrage (voir sous-section 6.2.5). Une fois le graphe créé, on recherche son nom dans les sous-graphes afin de le récupérer, et on envoit un signal afin de l'afficher. Ce dernier point pose problème. Le bon graphe est récupéré mais son affichage ne s'effectue pas, un autre graphe est affiché à la place.

6.4.3 Tests de validations et de fonctionnement

Les tests ont d'abord été effectués sur de petits cubes de données. Nous avons ensuite observé la cohérence entre les filtres appliqués et la valeur des sommets obtenus sur le graphe filtré (voir figures 6.17, 6.18, 6.19 et 6.20). Nous avons ensuite utilisé la même méthode pour des cubes de grandes tailles, même si dans ce cas nous n'avons pas pu vérifier chaque sommets.

La base de données utilisée pour les tests suivants est la même que celle utilisée lors de l'exemple de la figure 6.5.

FIG. 6.17 – Cube de données non filtrer

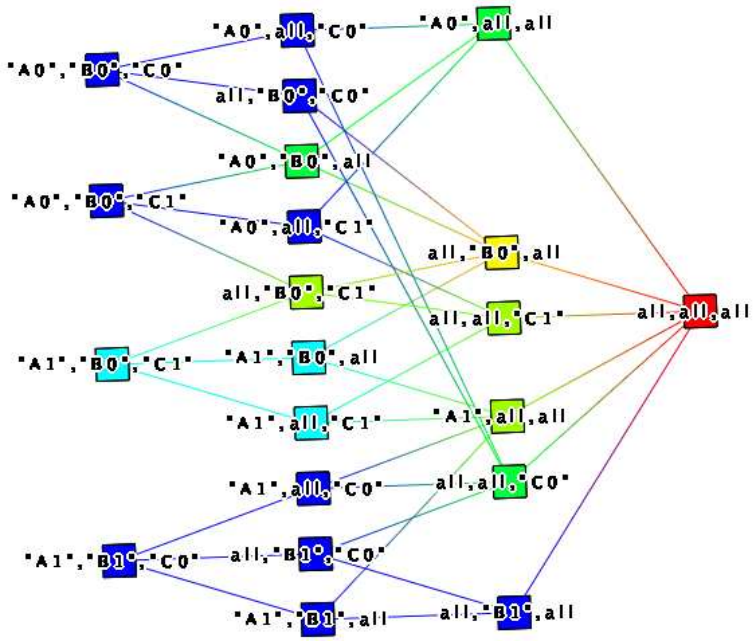


FIG. 6.18 – La deuxième dimension a été supprimée

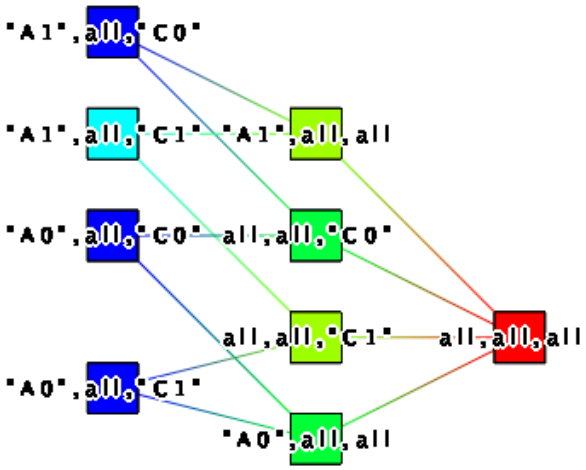


FIG. 6.19 – La première dimension a été filtrée avec le filtre $f = A0$

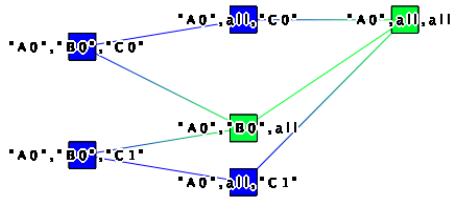
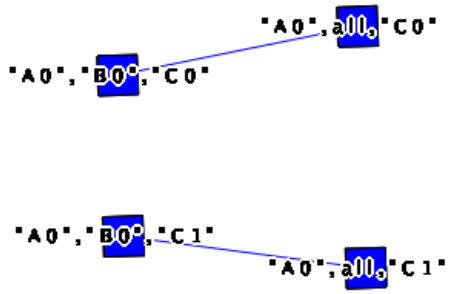


FIG. 6.20 – Le cube de la figure 6.19 a été filtrée sur la valeur de mesure avec $\$ < 4$



Chapitre 7

Exemple de fonctionnement et tests

7.1 Test complet du logiciel avec un exemple réel

Prenons pour démonstration du logiciel, une base de données réelle d'archéologie, donnée par les clients dans le fichier *tomb.csv*. Le but sera de "fouiller ces données" à l'aide de la visualisation, et plus précisément, de voir de quelle manière Bégonia peut permettre une aide à l'analyse de bases multidimensionnelles.

7.1.1 Création du cube

Le fichier recense des sépultures découvertes lors de fouilles archéologiques. En voici les premières lignes :

```
"NECROPOLE";"N__TOMBE";"RITE";"SIECLE";"REGIONS_AN";"SEXE";"AGE"
"Angera 1970-71";1;"cremation";2;"Transpadana";"INDETER.";"INDETER."
"Angera 1970-71";2;"cremation";2;"Transpadana";"INDETER.";"INDETER."
"Angera 1970-71";3;"cremation";2;"Transpadana";"INDETER.";"INDETER."
"Angera 1970-71";4;"cremation";0;"Transpadana";"INDETER.";"INDETER."
"Angera 1970-71";5;"cremation";8;"Transpadana";"INDETER.";"INDETER."
[...]
```

On peut déjà observer que ce fichier ne respecte pas les conditions d'applications du plug-in d'import d'un cube de données, pour deux raisons.

- Il ne faut pas de ligne de description du fichier (`"NECROPOLE";"N__TOMBE";...`)
- Il n'existe pas de valeurs de mesure

La première chose à faire avant d'utiliser le logiciel est donc de faire remplir les conditions d'application du plug-in au fichier `csv`. Cela est possible

facilement, grâce à un logiciel de tableur (par exemple *OpenOffice*). On supprime donc la première ligne, et on rajoute la valeur 1 à la suite de chaque tuple de la base. La base de données étant un recencement de tombes, chaque ligne représente une tombe découverte.

La base de données que nous allons donc "fouiller" se compose de 1846 6-uplets, tous différents, et chacun ayant pour valeur de mesure 1.

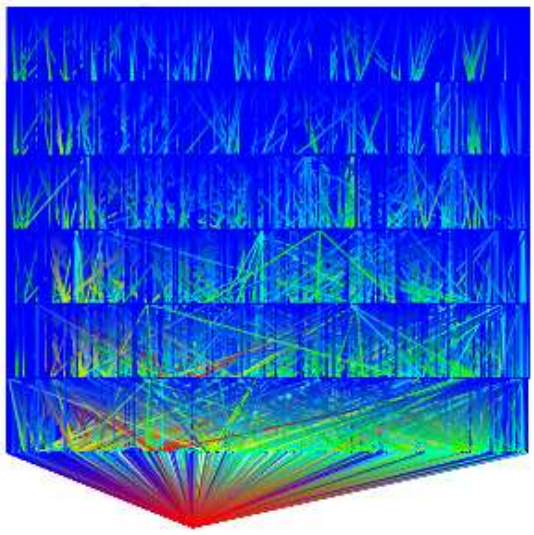
Pour la fonction d'agrégation, dans notre cas, les fonctions `MIN` et `MAX` ne sont pas intéressantes puisque tous les tuples ont la même valeur de mesure. Et comme cette valeur est 1, que l'on choisisse la fontion `SUM` ou `COUNT`, cela rendra le même résultat. Prenons la fonction `SUM` qui est par défaut.

L'importation du plug-in dure quelques secondes. On peut observer son avancée, grâce à une barre de progression qui évolue au fur et à mesure de la construction du cube.

7.1.2 Affichage du cube et premières manipulations

Lorsque l'importation est finie, le plug-in de Layout *HGDataCube* est exécutée avec une échelle de couleur logarithmique. On peut observer alors une première vue de notre cube de données qui possède alors 113.722 sommets et 474.670 arêtes(figure 7.1).

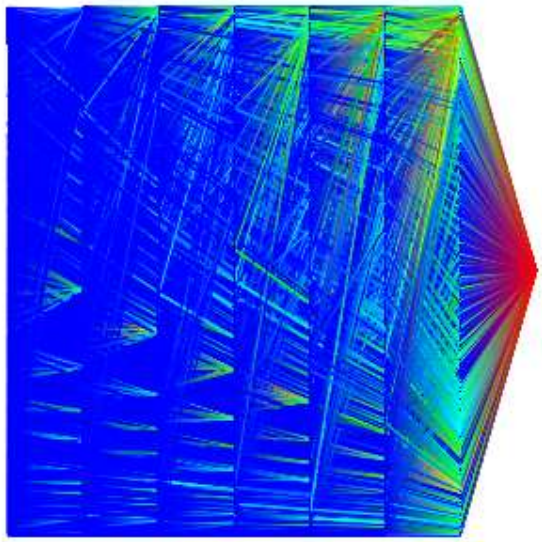
FIG. 7.1 – Première visualisation du cube



La première impression que l'on peut avoir en voyant ce cube, est justement de ne rien voir. Ce qui est affiché à l'écran est un gros bloc compact avec quelques couleurs, mais on ne perçoit pas quelles données se démarquent des autres.

On peut alors essayer de zoomer sur des sommets, afficher leurs labels tourner le graphe... Mais on ne parvient pas toujours à faire parler ce cube de données. Une des solutions est alors de changer l'affichage en essayant le plug-in de visualisation *LayoutDataCube*, qui va positionner les noeuds en fonctions de leurs labels. Ainsi, on obtient un graphe qui aurait comme un semblant de structure (figure 7.2). Cependant, il est toujours difficile d'en faire ressortir des données.

FIG. 7.2 – Visualisation du cube avec le plug-in *LayoutDataCube*



7.1.3 Filtrages d'éléments du cube

Certaines données dans le cube ne sont pas très intéressantes. En effet, si on regarde de plus près les valeurs de la propriété `viewLabel` dans la section `Porperty`, on observe que certains champs sont très souvents *indeterminé*. Grâce à la section `Filter`, on va pouvoir enlever les données non pertinantes qui génent la visibilité du cube.

En observant la base de données, on peut voir que les deux dernières dimensions (`SEXE` et `AGE`), comporte beaucoup de champs *indeterminés*. Ainsi grâce à la section `filtre`, nous pouvons supprimer ces deux dimensions qui semblent donc peu intéressantes. La dimension 2 (`N__TOMBE`) étant un identifiant de tombes, elle aussi sera filtrée.

Nous obtenons donc un nouveau graphe, avec 2457 sommets, qui correspond en fait au cube de données de la base comportant les quatre dimensions : `NECROPOLE`, `RITE`, `SIECLE` et `REGIONS_AN` (figures 7.3 et 7.4).

Supposons que nous sommes intéressés seulement, par les sépultures du I^{er} et II^{e} siécles. On peut donc filtrer ses valeurs en deux étapes. On filtre

FIG. 7.3 – Cube de données filtré sur des dimensions de la base, sans applications de plug-in de visualisation

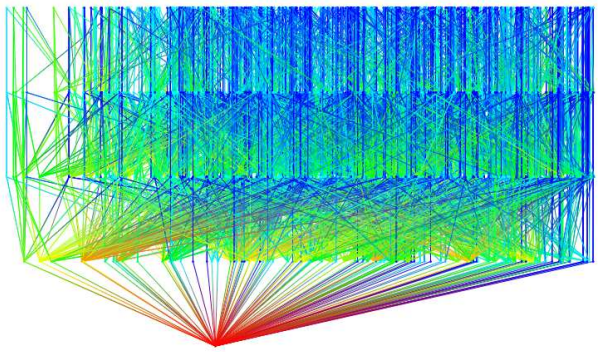
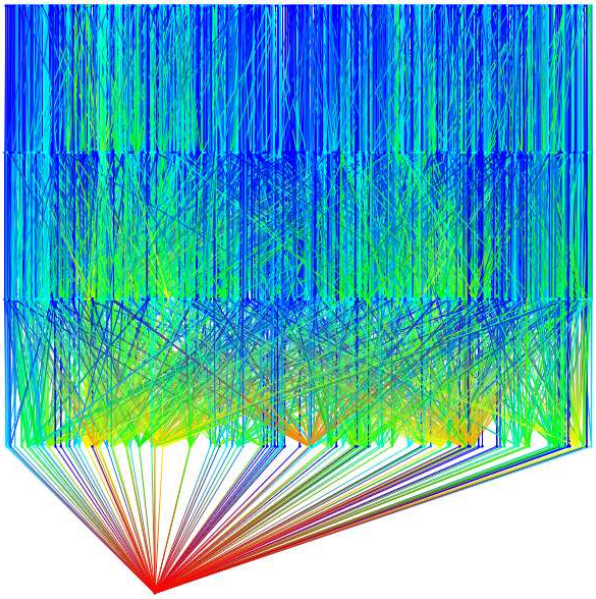
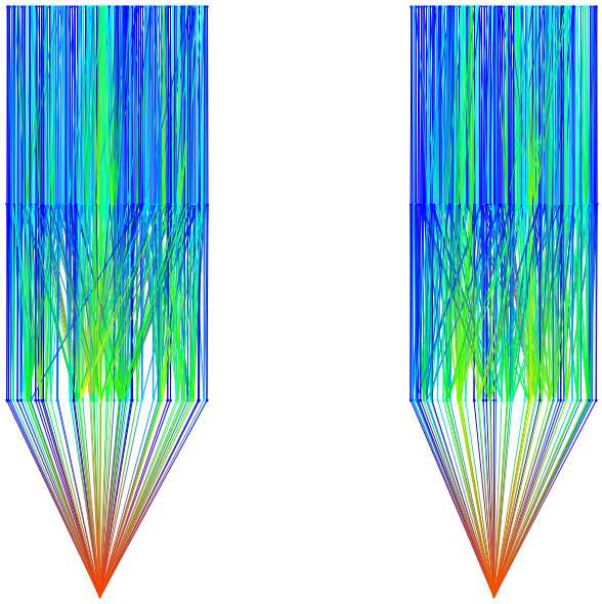


FIG. 7.4 – Cube de données filtré sur des dimensions de la base, avec l'application du plug-in *LayoutDataCube*



d'abord la troisième dimension de la base en prenant que les siécles strictement inférieurs à 3 puis ceux supérieur ou égale à 1. Le nouveau graphe, ainsi obtenu (sans appliquer de plug-ins de layout) posséde 755 sommets, et est non connexe. On peut voir en effet deux graphes distincts (figure 7.5).

FIG. 7.5 – Cube de données filtré sur des valeurs des attributs



En zoomant sur les sommets et en affichant leurs labels, on observe que, sur un graphe on retrouve tous les sommets correspondant aux tombes du I^{er} siécle et sur le second ceux du II^{e} siécle. On peut en conclure qu'aucun lieu de sépulture ne contenait des tombes des deux siécles.

7.1.4 Création d'un résumé du cube

Le plug-in de résumé de cube mis à disposition, n'est qu'une version d'expérimentation et ne fonctionne pas bien avec les graphes filtrés. L'appliquer ici serait donc dangeureux.

On peut cependant, si l'on souhaite, simplifier notre cube "à la main". En effet, on peut avec les outils de la barre de tâches, sélectionner des sommets puis les grouper. De cette manière, on peut faire un résumé de cube. Ceci dit, cela devient très vite laborieux.

7.1.5 Conclusion du test

Grâce à cet exemple, nous avons pu voir que le logiciel décrit dans cette étude, permet bien une première manipulation de cube de données, et peut donner lieu à des conclusions non évidentes en regardant simplement la base

de données. Nous voyons aussi avec cet exemple que l'implémentation d'un résumé de cube est inévitable afin de le simplifier, et pouvoir analyser correctement et plus précisement les données.

7.2 Tests de fonctionnement

Nous allons dans cette section voir comment les tests de fonctionnement ont été réalisés tout au long du projet afin de valider le fonctionnement de l'interface graphique et des plug-ins.

7.2.1 Origines des données

Les seules bases de données réellement mises à notre diposition, ont été données par les clients en fin de projet. Tous les tests, en cours de réalisation du logiciel, ont été effectués avec des bases de données réelles que nous avons créées.

Les bases de données que nous avons créées sont de deux types. Il y a tout d'abord les bases de données de petites tailles, avec quelques tuples et au plus trois dimensions. Les tuples de ces bases étaient choisis de manière judicieuse. Ainsi, elles nous ont permis de tester le bon fonctionnement des algorithmes de cube. En effet, ces bases sont assez petites pour que l'on puisse dessiner, à la main, le cube correspondant, ainsi que de faire quelques opérations dessus (tri des sommets sur un niveau, résumé de cube).

Comme les cubes de données sont utilisés pour de grandes bases de données, nous avons créé des bases de données de plus grandes tailles afin de savoir avec quelles bases de données notre logiciel fonctionne, et à quelle rapidité. Ces bases ont été créées de manière aléatoire uniforme, grâce aux fonctionnalités du tableur d'*OpenOffice*.

7.2.2 Outils de tests

Afin de chronométrer le temps d'exécution des création de cubes de données le code suivant a été utilisé :

```
#include <sys/time.h>
static struct timeval _t1, _t2;
static struct timezone _tz;
static unsigned long _temps_residuel = 0;

#define top1() gettimeofday(&_t1, &_tz)
#define top2() gettimeofday(&_t2, &_tz)

// Initialisation du chronométre
```

```
void init_cpu_time(void)
{
   top1(); top2();
   _temps_residuel = 1000000L * _t2.tv_sec + _t2.tv_usec -
                     (1000000L * _t1.tv_sec + _t1.tv_usec );
}

unsigned long cpu_time(void) /* retourne des microsecondes */
{
   return 1000000L * _t2.tv_sec + _t2.tv_usec -
           (1000000L * _t1.tv_sec + _t1.tv_usec ) - _temps_residuel;
}

int    InitSeed = 20 ;
```

Il suffit ensuite d'utiliser ces fonctions au sein du plug-in d'import afin de connaître le temps exact de calcul.

Les tests ont été réalisés principalement sur trois machines :
- Un PC avec un processeur AMD Athlon XP 2 Ghz et 256 Mo de mémoire vive.
- Les PC du CREMI : Celeron 2 Ghz avec 512 Mo de mémoire vive.
- La machine *vivaldi* du CREMI : bi-processeurs Xeon 2,2 GHz hyper-threadés avec 4 Go de mémoire vive.

Les tests ont d'abord eu lieu sur une machine relativement rapide (Athlon XP 2 Ghz) mais avec peu de mémoire vive (256 Mo). Les cubes de données ont pu être créés si ceux-ci n'était pas trop gros ($<$ 1.000.000 de sommets). Ensuite nous avons utilisé les machines du CREMI, moins puissantes, et donc moins rapides sur les petits cubes, mais avec plus de mémoire vive (512 Mo), ce qui a permis de créer des cubes plus gros qu'avec la machine précédente. Ceci dit pour des cubes de plus de 2.000.000 de sommets, ces machines ne possédaient plus assez de mémoire. Nous avons donc réalisé les tests qui n'avaient pas fonctionné jusqu'alors, sur la machine Vivaldi.

7.2.3 Résultats des tests

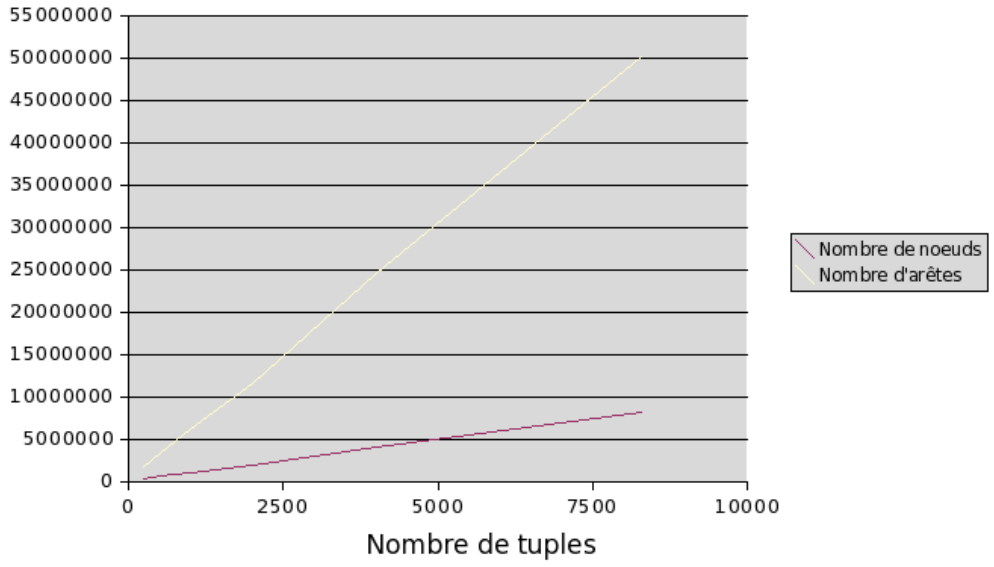
Nous proposons ici, plusieurs tableaux récapitulatifs des tests de création de cubes de données. On peut y voir le nombre de tuples et de dimensions des bases de données d'origine, le nombre de noeuds et d'arêtes dans les cubes créés, le temps de calculs, la taille des fichiers `tlp` d'enregistrement des graphes, et des informations sur les machine qui ont effectués les calculs. Tous ces tests ont été réalisés à partir d'une base de données réelle d'archéologie données par les clients, dans le fichier `objetNN.csv`. Cette base contient un recencement d'objets découverts lors de fouilles archéologiques, ainsi que des caractéristiques de ses objets.

FIG. 7.6 – Résultats des tests

Nb tuples	Nb dimensions	Origine	Tps Création	Nb noeuds	Nb arêtes	Taille fichier
250	11	Archéo.	41 s	293232	1752980	30 Mo
500	11	Archéo.	1 min 13 s	553901	3334614	170 Mo
1000	11	Archéo.	2 min 2 s	1016559	6180874	300 Mo
2000	11	Archéo.	4 min 11 s	1869658	11464809	605 Mo
4000	11	Archéo.	6 min 34 s	4016603	24554182	1,3 Go
8278	11	Archéo.	13 min 33 s	8187976	50267321	

De ce premier tableau (tableau 7.6), nous avons tracé la courbe représentant le nombre de sommets du cube de données, en fonction du nombre de tuples de la base de données initiale (courbe 7.7). Nous pouvons donc percevoir que le nombre de noeuds dans le graphe créé est proportionnel au nombre de tuples de la base. De plus d'après les chiffres du tableau, la taille du fichier, le temps d'exécution et le nombre d'arêtes sont aussi proportionnels aux nombre de sommets du cube.

FIG. 7.7 – Nombre de sommets du cube en fonction du nombre de tuples



De ces deux tableaux, nous avons voulu savoir les relations entre le nombre de sommet du cube et le nombre de dimensions de la base de données. Ainsi en extrayant les colonnes `Nb dimensions` et `Nb Noeuds` nous avons obtenu les deux courbes 7.10 et 7.11 provenant respectivement des tableaux 7.8 et 7.9. Nous pouvons voir cette fois-ci, les courbes créées sont des courbes exponentielles. Lorsqu'on ajoute une colonne à la base, on multiplie par deux

FIG. 7.8 – Résultats des tests sur des bases avec peu de tuples

Nb tuples	Nb dimensions	Origine	Tps Création	Nb noeuds	Nb arêtes	Taille fichier
10	5	Archéo.	0,1 s	224	632	29 Ko
10	6	Archéo.	0,1 s	528	1704	86 Ko
10	7	Archéo.	0,2 s	1064	3956	175 Ko
10	8	Archéo.	0,3 s	2248	9372	481 Ko
10	9	Archéo.	0,7 s	4664	21716	1 Mo
10	10	Archéo.	1,3 s	9328	48096	2 Mo
10	11	Archéo.	2,8 s	19408	108960	5 Mo

FIG. 7.9 – Résultats des tests sur des bases avec beaucoup de tuples

Nb tuples	Nb dimensions	Origine	Tps Création	Nb noeuds	Nb arêtes	Taille fichier
8278	7	Archéo.	13 s	356979	1519542	78 Mo
8278	8	Archéo.	34 s	821759	3866950	202 Mo
8278	9	Archéo.	3 min 43 s	1845314	9568494	510 Mo
8278	10	Archéo.	6 min 27 s	3701157	20960128	1,1 Go
8278	11	Archéo.	13 min 33 s	8187976	50267321	

(environ) le nombre de sommets du cube de données, et ceci aussi bien pour des petites bases que pour des grandes.

Ces tests montrent que notre logiciel peut fonctionner pour des bases de données de tailles modestes sur des machines puissantes. Finalement, le problème qui se posent n'est pas celui pensé initialement. En effet, d'après les explications des clients, nous pensions que le problème des tests, seraient le temps d'exécution de création du cube. En fait, le premier problème rencontré est celui de l'espace mémoire utilisée pour le cube et sa création.

De plus, Tulip est prévu pour la manipulation de graphe pouvant avoir plusieurs centaines de milliers d'éléments sur une station de travail "standard". Nous, nous pouvons obtenir, avec des bases de données de tailles modestes, des graphes qui peuvent atteindre plus de 10 millions d'éléments ! La visualisation de tels cubes se révélent donc difficile et lente avec l'usage de Tulip.

FIG. 7.10 – Nombre de sommets du cube en fonction du nombre de dimensions, dans des petites bases

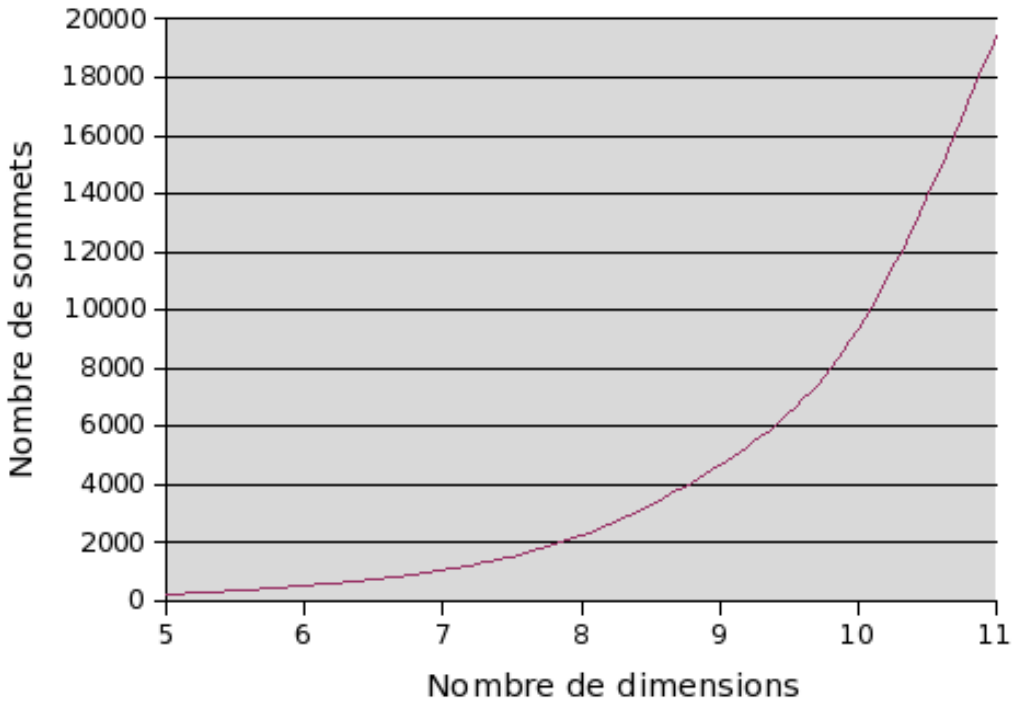
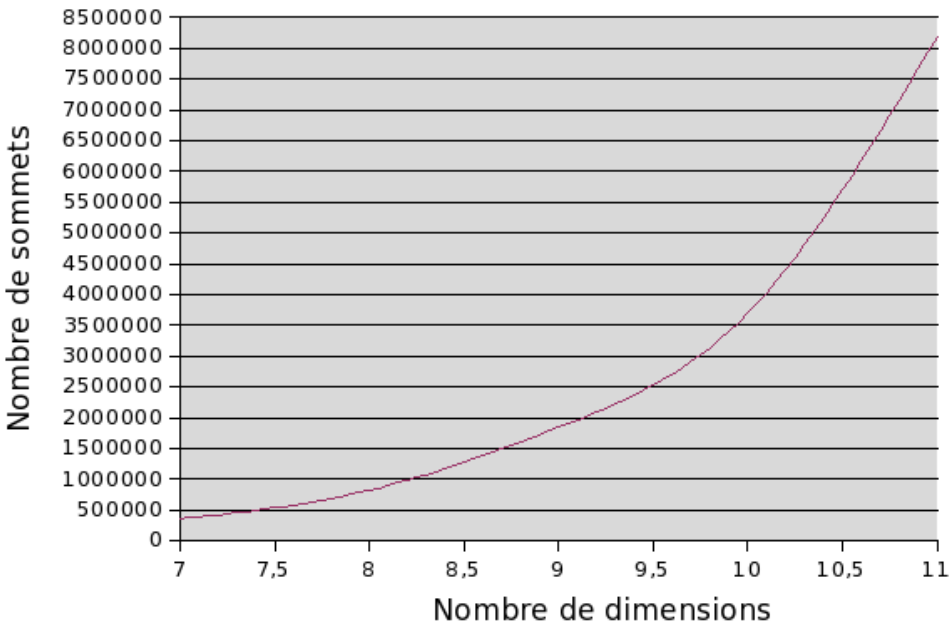


FIG. 7.11 – Nombre de sommets du cube en fonction du nombre de dimensions, dans des grandes bases



Chapitre 8

Extensions possibles et conclusions

Au cours du projet nous nous sommes rendu compte que certains points du projets auraient pu être développés, et que d'autres auraient pu être ajoutés. En raison du temps limité, nous n'avons pas pu réaliser ces différentes fonctionnalités. De plus, le logiciel à été créé afin de pouvoir intégrer de nouveaux outils d'analyse de cube de données.

8.1 Création du Quotient Cube

La création du Quotient Cube est véritablement un sujet de recherche. Les algorithmes existant sont peut être optimisables, et d'autres sont encore à créer.

Notre logiciel supporte des cube de données ayant quelques millions d'éléments au maximum. Cependant nous avons vu qu'avec des données réelles le nombre d'éléments est trop important pour pouvoir être géré par un ordinateur de puissance standard à l'heure actuelle.

Il faudrait pour supporter de tels cubes développer un plug-in d'import qui résume le cube durant sa création. Les détails de l'intérieur d'une classe du résumé de cube, seraient alors calculés seulement à la demande. La reconstruction des détails de la classe est possible si l'on stocke dans la classe tous les sommets constituant les bornes inférieures et supérieures de cette classe.

Cette solution apporterait de nombreux avantages :

- optimisation de l'utilisation de la mémoire vive
- affichage d'un cube directement résumé
- sauvegarde d'un cube de données de taille moindre.
  En effet le stockage des cubes pose aussi problème, car il peut atteindre plusieurs giga-octets (d'après N. Novelli, chercheur à l'Université de Marseille). Ici, le cube stockée serait le résumé de cube.

8.2 Interface graphique

Il aurait été intéressant de développer des fonctions qui, à partir d'un sommet, choisissent tous les sommets connectés qui ont la même valeur, ou qui appartiennent au même ensemble de valeurs.

8.3 Modularité du logiciel

Il aurait été intéressant de mettre une entête dans les fichiers *csv*. Cette entête comporterait le nom des champs et leur type. Ainsi au lieu de considérer tous les champs comme des chaines de caractères, les champs de type entier ou réel seraient au préalable convertis.

Cela économiserait de l'espace mémoire, et l'ensemble des traitements du cube devrait être plus rapide. Entre autres, cela éviterait de convertir tous les champs d'une dimension lorsque on filtre un champ en considérant celui-ci comme une valeur.

8.4 Conclusion

Ce projet nous a enrichi en plusieurs points.

Nous nous sommes perfectionnés en C++ : nous avons pu constater que l'apprentissage d'un langage est possible de façon autonome. La construction d'un planning est de l'architecture est difficile. Lorsqu'on n'est pas rigoureux, on peut négliger certaines parties qui se révèlent plus importantes que prévues. Le travail en équipe est plus difficile à gérer mais il permet de résoudre les problèmes plus facilement, grâce à la complémentarité de chacun.

Nous avons respecté le contrat du client et fournit un logiciel expérimental fonctionnel. La modularité du logiciel permettra d'intégrer les extensions décrites précédemment.

Il nous a semblé que l'exploration des cube de données est assez compliquée. Nous nous posons donc la question de l'efficacité de la méthode de fouille de données par la visualisation de cubes.

Bibliographie

[1] David Auber. http ://www.tulip-software.org/. Tulip's web site.

[2] David Auber. *Outils de visualisation de larges structures de données.* PhD thesis, Université Bordeaux I, 2002.

[3] David Auber. *Graph Drawing Software*, chapter Tulip - A Huge Graph Visualization Framework. Springer-Verlag, 2003.

[4] Trolltech company. http ://www.trolltech.com/products/qt/. QT's web site.

[5] Ladjel Bellatreche (LISI/ENSMA Futuroscope). Techniques d'optimisation des requêtes dans les data warehouses. In *Sixth International Symposium on Programming and Systems*, pages 81–98, 2003.

[6] T. Munzner. *Drawing large graphs with h3viewer and site manager.* Springer-Verlag, 1998.

[7] Ying Feng & Divyakant Agrawal & Amr El Abbadi & Ahmed Metwally (University of California). -. In *Range CUBE : Efficient Cube Computation by Exploiting Data Correlation*, 2002.

[8] Laks V.S. Lakshmanan (University of Columbia) & Jian Pei (Simon Fraser University) & Jiawei Han (University of Illinois). Quotient cube : How to summarize the semantics of a data cube. In *Prooceeding of the 28th Very Large Database (VLDB)*, 2002. Hong Kong.

[9] Jan Skansholm. *C++ from the beginning.* Pearson Education, 2002.

[10] Bjarne Stroustrup. *Le langage C++.* Pearson Education, 2003.

[11] G.J. Wills. *NicheWorks : Interactive visualization of very large graphs.* Springer-Verlag, 1991.

Annexe A

Manuel utilisateur

A.1 Introduction

Bégonia est un logiciel inspiré de Tulip. Il propose une interface adaptée à l'exploration des cubes de données.

Ce manuel s'adresse à des utilisateurs connaissant les principes de bases de Linux et des cubes de données.

A.2 Installation du logiciel

A.2.1 Configuration initiale

Pour être sûr de pouvoir utiliser Bégonia vous devez avoir les applications suivantes installées et correctement configurées :

- Linux
- Un serveur graphique supportant OpenGL
- La librairie QT3.x avec ses outils de développement (qmake, uic, QT Designer). De plus, le support des threads doit être activé.
- La librairie Tulip à partir de la version 2.0.2

A.2.2 Compilation

En suivant la méthode décrite ci dessous, le logiciel ne sera pas installé au coeur de votre système d'exploitation, mais sera seulement exécutable à partir de son répertoire.

Compilation automatique

Si toutes les applications listées ci dessus ont été correctement installées en mode root, le logiciel peut être automatiquement compilé grâce au script `install` fournit avec bégonia.

Ce script compile d'abord chaque plug-ins, puis, il compile l'interface de Bégonia. Si aucun message d'erreur n'est apparu (l'apparition de warning

n'est pas gênante), vous pouvez lancer Bégonia en tapant dans un shell : `./begonia`

Attention, à cause des informations de compilations, les messages d'erreurs ne sont pas toujours visible. Pour être sûr de ne pas avoir d'erreurs de compilation, lancez une nouvelle fois `./install`, vous devez seulement voir apparaitre `make: Rien à faire pour « all ».`

Si vous recevez des messages d'erreurs, un des éléments n'a pas été compilé. Référer vous alors au paragraphe suivant.

L'interface est fortement liée aux plug-ins. Si un des plug-ins n'a pas été compilé, l'interface se lancera, mais certaines fonctionnalités seront absentes (voir la section A.3).

Remarque : tapez `./clean` pour effacer tous les fichiers compilés et ceux parraissant superflus (voir le script pour plus de détails).

Configuration complémentaire

Si Tulip n'as pas été installé sur les répertoires Linux par défaut (nécessite normalement d'avoir les droits de root), la compilation automatique va échouer.

Le seul problème connu est que le linker Linux ne trouve pas les bibliothèques de Tulip. Il faut donc ajouter au linker le chemin vers ces librairies.

On peut pour cela ajouter dans le fichier *.bashrc* :

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/path/to/libraries
```

par exemple :

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/net/cremi/nom/
_projet/local/lib
```

Configuration de développement

Configurer QT Designer en ajoutant dans le menu *Edit->Preferences-> Plugin Paths*, le chemin vers les bibliothèques contenant les plug-ins QT de Tulip.

Par exemple : `/net/cremi/nom/misc/local/share/Tulip/plugins`

A.3 Fonctionnalités de Bégonia

A.3.1 L'interface générale

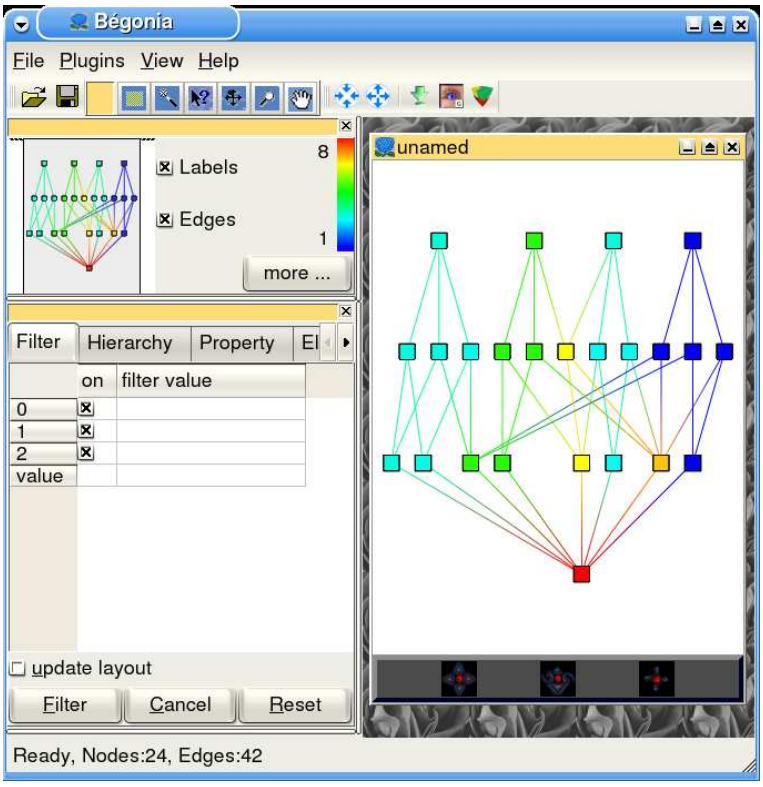
Après avoir importé un cube de données sous Bégonia, vous verrez un écran similaire à celui de la figure A.1.

A.3.2 Les différents menus

Le menu File

– open : permet d'ouvrir un graphe enregistré

FIG. A.1 – Bégonia en action



- save : permet de sauvegarder un graphe
  Les attributs du graphe n'étant pas enregistrés avec la version Tulip 2.0.2, le chargement d'un graphe sauvé ne peut donc pas être traité par Bégonia. Les versions ultérieures de Tulip le supporteront peut-être.

Le menu Plugins

- Import :
  DataCube : permet d'importer un cube de données à partir d'une base csv
  Si la base csv possède des lignes identiques, seul la première ligne est prise en compte. De plus le dernier champ doit contenir la valeur de mesure (type réel)
  Les actions suivantes ne peuvent être utilisées que si un graphe a été chargé auparavant (chargement d'un fichier csv).
- Layout :
  HGDataCube : organise la disposition des sommets de telle sorte qu'un minimum d'arêtes se croisent
  Ce plug-in utilise le plug-in Hierarchical Graph de la librairie de Tulip, puis applique le plug-in de couleur spécialisé pour les cubes de données.
  DataCube : organise la disposition des sommets d'après la valeur des champs de chaque sommet
  Ce plug-in applique ensuite le plug-in de couleur spécialisé pour les cubes de données.

Clustering : le mécanisme de clustering (groupement) permet d'afficher une vue simplifiée du graphe.
Quotient Cube : créé un résumé du cube en regroupant les sommets ayant la même valeur

le menu View

- Center View : permet de recentrer la vue du cube
- AutoLayout : applique automatiquement le layout HGDataCube après avoir importer le graphe
- Dialogs : réactive une fenêtre si elle etait désactivée
  3D Overview : réactive la fenêtre qui affiche une vue globale du graphe
  Mouse ToolBar : réactive la barre d'outils
  info Editor : réactive la fenêtre d'édition des propriétés

Le menu Help

- about : les concepteurs du logiciel

La barre d'outil Elle permet de lier une opération à la souris. Les opérations suivantes permettent d'explorer le graphe.

- Box Selection : sélectionne des éléments du graphe

- Get Information : affiche les informations d'un élément dans la fenêtre PropertyElement
- Move selection : permet de déplacer les éléments selectionnés du graphe
- Zoom box : zoom sur l'espace défini
- Move In graph : navigation autour du graphe
- Grouper : permet de regrouper manuellement l'ensemble des sommets sélectionnés, en créant une classe
- Dégrouper : dégroupe toutes les classes selectionnées

La barre de raccourci

Permet de déclencher les plug-ins : *Import DataCube*, *HGDataCube* et *QuotientCube*

A.3.3 La fenêtre de fouille du graphe

L'onglet de filtrage

Cet onglet (*Filter*) s'active et s'initialise lorsque le graphe est de type cube de données. Il affiche alors pour chaque dimension du cube une ligne de filtre.

Afin de supprimer entièrement une dimension du cube, il suffit de décocher la boite de sélection correspondant à la dimension non voulue (colonne *on*). La suppression d'une dimension supprime tous les sommets du graphe n'ayant pas cette dimension positionnée à *all*.

Pour filtrer certaine valeurs d'une dimension, il faut remplir la ligne correspondante (colonne *filter value*) avec une expression décrivant le filtre. Afin de construire cette expression, on peut considérer la dimension soit comme une chaine de caractères, soit comme une valeur réelle.

L'expression de type caractère doit être de la forme suivante : *f opération chaîne*. L'opération peut être égale à = ou *!=*. Avant de comparer le champ avec la chaîne, les guillemets entourant le champ seront supprimés.

L'expression de type réel doit être de la forme suivante : *$ opération réel*. L'opération peut être égale à $=, !=, <, >, <=, >=$. Le réel peut avoir la forme *-3.45e-45* ou plus simplement *2*.

Enfin, on peut filtrer les sommets en fonction de leur valeur. Il suffit pour cela de mettre un filtre sur la dernière ligne. L'expression doit être de type réel (décrit précédemment).

De toutes façons, si l'expression entrée est erronée, un message d'erreur explicatif s'affichera.

Le graphe créé sera un sous-graphe du graphe racine. Ainsi on peut filtrer de nouveau le sous-graphe. Par ce mécanisme on peut choisir des intervalles de valeurs.

Par exemple, si on veut tous les sommets dont les valeurs sont comprises entres 2 et 3, il suffit d'appliquer le filtre *$ >= 2* sur un graphe, puis d'appliquer sur le sous-graphe créé le filtre *$ <= 3*.

Description des boutons :

- Filter : applique les filtres

– Cancel : recharge les filtres du graphe en cours
– Reset : réinitialise les filtres à leur valeur par défaut

Attention ! Il existe un bug non résolu : lorsqu'on applique les filtres, parfois le cube créé n'est pas celui automatiquement affiché. Pour pouvoir récupérer le dernier cube créé, cliquez avec le bouton droit de la souris sur la fenêtre affichant le graphe, puis cliquez sur *Update Hierarchy Tab*. Enfin allez choisir manuellement le graphe voulu à l'aide de l'onglet *Hierarchy*.

Affichage de la hiérarchie du graphe

Cet onglet (*Hierarchy*) permet de naviguer et d'agir dans la hiérarchie des sous-graphes du graphe. Avec le menu contextuel (clique droit sur un des graphes), on peut supprimer ou renommer les graphes.

Affichage d'une propriété de tous les éléments

L'onglet *Property* permet d'afficher les propriétés des noeuds et des arêtes dans un tableau.

Elle se compose de deux parties principales :

La première partie affiche la valeur des éléments de la propriété sélectionnée. L'utilisateur ne peut pas modifier ces valeurs. Il est possible d'afficher seulement les éléments sélectionnés en cochant la boite de sélection *Filter*.

Le bouton *To labels* permet d'afficher sur les éléments du graphe les valeurs de la propriété selectionnée.

Il est possible d'afficher seulement les éléments sélectionnés en cochant la boite de sélection *Filter*.

La deuxième partie est une liste de toutes les propriétés locales et héritées d'un graphe. Les propriétés héritées sont des propriétés qui appartiennent à un graphe parent dans la hiérarchie du graphe.

Affichage des propriétés d'un élément

Dans cet onglet (*Element*), on affiche toutes les propriétés d'un élément du graphe. La sélection de l'élément s'effectue avec la souris lorsque l'opération *Get Information* est activée (voir sous-sectionA.3.2).

A.3.4 La fenêtre affichant une vue globale du graphe

Cette fenêtre permet d'avoir à tout moment, une vue globale du graphe et la position actuelle de la caméra.

Elle permet aussi de paramétrer la façon dont le graphe sera affiché.

Table des figures