Facteur d'étirement et application à la visualisation de graphes

Cyril Faure

Mémoire de DEA
mai 2003

Résumé

Ce mémoire s'inscrit dans le domaine de la visualisation de graphes. Plus précisément, il concerne l'étude et l'exploitation d'un paramètre pour améliorer la qualité d'affichage de gros graphes : le facteur d'étirement. Le but étant, pour un graphe et un arbre couvrant de ce graphe donné, d'utiliser le facteur d'étirement afin de donner une priorité à l'affichage des arêtes du co-arbre. Le deuxième objectif est de trouver un ordonnancement des fils de chaque noeud tel que l'affichage en soit amélioré.

Table des matières

1 Introduction

La visualisation de graphes est l'art de pouvoir dessiner un graphe tel que les relations entre les différents éléments soient facilement identifiables lorsque l'on visualise ce graphe. Ainsi, une définition formelle définissant la qualité d'un "bon dessin" a peu de chance d'exister. Il est de bon sens de déterminer les critères que l'on considère comme pertinents pour la compréhension avant toute élaboration d'un algorithme de dessin et ceux-ci dépendent bien entendu de l'application que l'on désire faire de l'affichage. Parmi les plus connus, on peut citer la minimisation du nombre de croisements ou bien la minimisation du nombre de courbures des arêtes dans le dessin.

Cependant, ce mémoire ayant pour contexte la visualisation de gros graphes, réaliser un affichage clair devient un problème plus compliqué comportant des contraintes supplémentaires. Effectivement, la plupart des algorithmes définissent le problème de l'affichage d'un graphe comme suit : soit un ensemble de noeuds avec un ensemble d'arêtes, calculer la position des noeuds ainsi que les courbes à tracer pour chaque arête, tout en respectant un ensemble de contraintes esthétiques. Cependant, même si l'on peut arriver à afficher tous les éléments du graphe, les résultats concernant la visibilité sont rarement satisfaisant car on n'arrive tout simplement plus à discerner les relations entre les noeuds, on pourra tout au plus avoir une indication générale sur la structure du graphe mais elle sera difficilement compréhensible d'un point de vue cognitif. Bref, la taille d'un graphe peut tout à fait rendre un algorithme d'affichage pourtant efficace en temps normal, totalement inutilisable.

On constate qu'aucun algorithme de visualisation ne peut se charger de tous les problèmes d'affichage que soulève la taille des grands graphes. Comme le précisent Ivan Herman, Guy Melançon et Scott Marshall dans leur article [10], la tache de révéler la structure d'un graphe nécessite des approches innovantes[1].

L'objectif de ce mémoire va dans ce sens car nous y proposons l'utilisation

[1]L'interressé pourra se référer à cet article pour avoir un résumé des approches existantes permettant d'améliorer les algorithmes d'affichage usuels

d'un paramètre non encore étudié dans le domaine de l'affichage de graphes. Nous allons nous servir d'un algorithme d'affichage existant et le modifier à l'aide des résultats de ce paramètre que nous aurons calculé sur les arêtes du graphe à afficher.

Ce mémoire s'inscrivant dans le domaine de la visualisation de graphe, une importante partie a été par conséquent consacrée à l'implémentation de divers algorithmes sous le logiciel de visualisation de graphes Tulip [3].

Ce mémoire sera organisé comme suit : le chapitre 2 fournira quelques définitions pour ensuite énumérer les objectifs ayant été fixés pour cette étude ainsi que le cheminement suivi. Ensuite, nous synthétiserons plusieurs articles ayant pris une place prépondérante dans cette étude (chapitre 3). Le chapitre 4 présentera la méthode employée pour calculer le facteur d'étirement du graphe ainsi que l'application que nous en avons faite pour réaliser l'affichage du graphe. Nous discuterons ensuite des perspectives d'amélioration de ce procédé et des problèmes ouverts. Enfin, dans le chapitre 5, le développeur pourra trouver les détails d'implémentations des différentes branches de l'application.

2 Généralités

2.1 Quelques définitions

Sauf précision contraire, les graphes utilisés dans ce mémoire sont des graphes simples non orientés ni valués. Pour tout graphe G, $V(G)$ dénote l'ensemble des sommets de G et $E(G)$ l'ensemble des arêtes de G. n dénotera le nombre de sommets du graphe et m son nombre d'arêtes.

Définitions 2.1.1. Le *graphe partiel* H d'un graphe G est un graphe tel que $V(H) = V(G)$ et $E(H) \subseteq E(G)$. On appelle *arbre couvrant* du graphe G un arbre T, graphe partiel de G. On appelle arêtes du *co-arbre* l'ensemble $E(G) - E(T)$.

Définitions 2.1.2. Pour toute paire de sommets $x, y \in V(G)$, $d_G(x, y)$ dénote la distance de x à y dans G, c'est à dire la longueur de la chaîne la plus courte entre x et y. Soit H un graphe partiel du graphe G, pour toute arête $(x, y) \in E(G) - E(H)$, $d_H(x, y)$ est appelé le *facteur d'étirement* de l'arête (x, y). Pour tout $t \geq 1$, H est un *graphe t-couvrant* de G si $d_H(x, y) \leq t$ pour chaque paire de sommets x et y de G. Le paramètre t est appelé *facteur d'étirement* de H [1]. L'*indice d'étirement* est le nombre minimum t pour lequel H est un t-couvrant[2].

Définitions 2.1.3. Soit un arbre enraciné T, et x, y deux noeuds, x est un *ancêtre* de y si le chemin de la racine de T à y passe par x. L'ancêtre commun le plus proche, noté $NCA(x, y)$ [3] de deux noeuds x et y est l'ancêtre commun de x et y le plus éloigné de la racine.

[1] Il est à noter que le terme de facteur d'étirement est employé aussi bien pour le graphe couvrant que pour les arêtes du co-arbre. Cependant nous leverons les ambiguités si besoin est dans les sections suivantes

[2] Précisons que nous ne nous intéresserons qu'à des valeurs entières de t

[3] pour *nearest common ancestor*

2.2 Motivations

Comme précisé précédemment, le principal problème dont souffrent la majorité des techniques d'affichage est qu'elles ne sont applicables que pour des graphes de taille relativement petite. Ceci découle de deux contraintes relatives à la nature pratique de la visualisation de graphe : tout d'abord, la plupart des algorithmes dessinant des graphes quelconques ont des complexités en temps non linéaire [4], ce qui est inabordable dans des applications nécessitant une interaction de l'utilisateur pour des graphes dépassant le millier de sommets.

Le deuxième obstacle vient du fait que la place requise par l'affichage d'un grand graphe peut s'avérer supérieure à la taille de l'écran. Ainsi, vouloir rechercher un affichage plannaire ou minimiser les croisements des arêtes d'un tel graphe n'a plus vraiment de sens, on peut donc ignorer de tels critères. Ensuite, afficher la totalité du graphe n'a que peu d'intérêt d'un point de vue cognitif, on préférerait conserver l'image mentale que l'utilisateur s'est construite sur les données ([4], p. 59).

Problème 1. *Voici donc les problèmes induits par la structure d'un grand graphe :*

1. *baisse significative des performances (temps de réponse),*
2. *dépassement de la capacité d'affichage de l'écran.*

2.3 La fragmentation de graphes

La fragmentation de graphes ou *clustering* consiste à construire un ensemble de sous-graphes (*clusters*) à partir d'un graphe.

On peut différencier deux types de clustering. Le plus connu est certainement le clustering hiérarchique sur les noeuds où l'objectif est de rassembler les noeuds "suffisamment similaires" dans un même cluster qui forme ainsi un sous-ensemble de noeuds. Ces sous-ensembles et leurs relations seront ensuite représentés à l'aide d'un algorithme de dessin de graphe usuel.

[4]les technique dites d'affichage à ressort où *force-directed methods* sont parmi les plus lourdes avec des complexité en temps de l'ordre de $O(n^3)$

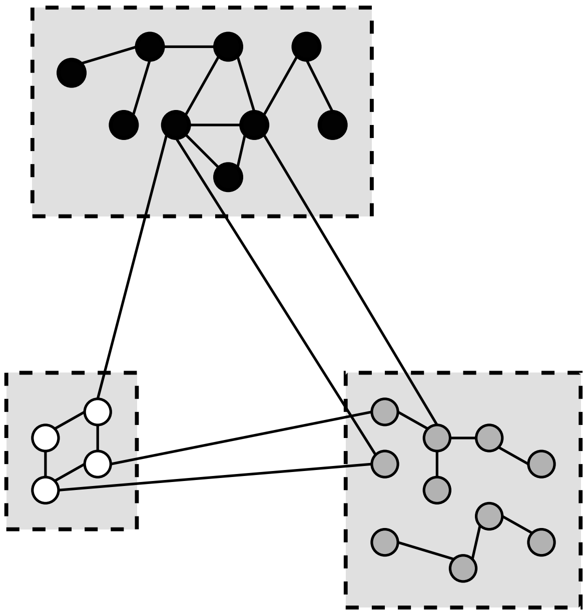


FIG. 2.1 – Le clustering hiérarchique permet de ne représenter que des sous ensembles de noeuds

Une deuxième approche consiste à filtrer les éléments à afficher en fonction d'un seuil choisit, les éléments en dessous de ce seuil étant ignorés (cf. fig 2.2(a) et 2.2(b)). L'idée est donc de n'extraire du graphe que les composantes les plus importantes de manière à ne présenter qu'une vue non détaillée de la structure.

Afin de déterminer les clusters de manière pertinente, on a besoin d'une mesure, appelée *métrique*, permettant d'effectuer les calculs entre les différents éléments du graphe.

La définition d'une métrique changeant suivant les papiers, nous utilisons dans ce mémoire la définition qu'utilisent Herman et al. dans leur article [10].

Définition 2.3.1. *Le terme* métrique *se réfère à une mesure associée à un noeud ou une arête dans un graphe.*

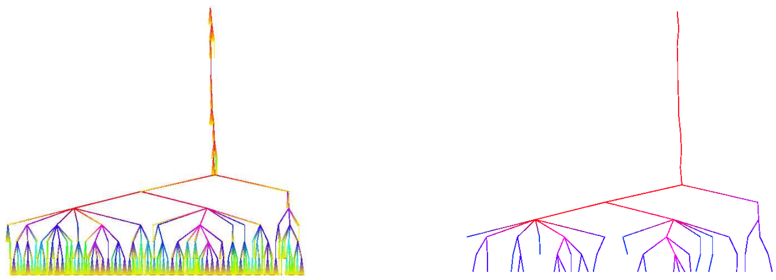
Plusieurs types de métrique existent, une métrique utilisant des données sémantiques associées avec les éléments du graphe afin de réaliser le clustering est dite "basée sur le contenu"[5]. Une métrique est dite structurelle si elle n'utilise que la structure du graphe pour information[6], l'avantage d'une telle approche est qu'elle ne nécessite aucune connaissance du domaine dont est

[5]on entendra aussi parler d'approche extrinsèque

[6]on entendra aussi le terme d'approche intrinsèque

issu le graphe. Evidemment, une métrique n'a de valeur que dans le cas où la mesure qu'elle fournit est pertinente par rapport au contexte. C'est pour cette raison que plus de recherches se concentrent sur l'étude de métriques structurelles, les clusters calculés à partir de la structure même du graphe étant pertinents pour la plupart des applications.

Une des métriques structurelles utilisées dans Tulip [3] pour l'affichage d'arbres est le nombre de Strahler [17], il est calculé récursivement sur les noeuds du graphe en fonction de celui de leurs fils, les noeuds ayant le plus haut nombre de Strahler générant un affichage partiel de l'arbre appelé *squelette* de l'arbre (voir Herman et al. [10], p.39).



(a) Affichage d'un arbre surchargé

(b) Affichage d'un arbre allégé (Strahler)

FIG. 2.2 – Utilisation de la métrique de Strahler

3 Articles exploités

Dans cette partie sont décrits divers articles ayant touché de près ou de loin ce mémoire. Certains ont contribué directement aux diverses étapes du développement tandis que d'autres ont apporté des perspectives d'amélioration possibles.

3.1 Arbres t-couvrants

Un arbre t-couvrant T d'un graphe G est un arbre couvrant dans lequel la distance entre chaque paire de noeuds est au plus t fois leur distance dans G (définitions 2.1.2 page 6). L'article de Leizhen Cai et Derek G. Corneil intitulé *Tree Spanner* [6] étudie l'algorithmique des arbres t-couvrants. Plus exactement, il traite de l'existence d'arbres t-couvrants selon les valeurs de t et présente des résultats obtenus en terme de complexité.

Avant toute chose, il est à noter que pour les besoins de ce mémoire, nous nous limiterons au résumé des parties principales concernant les graphes non dirigés et non valués. L'intéressé pourra consulter l'article s'il souhaite examiner le cas des graphes valués et orientés.

Les premiers à avoir introduit la notion de graphes t-couvrants furent Peleg et Ullman en 1987 [14]. L'idée principale derrière la notion de t-couvrants est l'approximation de distances noeud à noeud dans le graphe original par des graphes partiels. La qualité de cette approximation de distance par un t-couvrant est mesurée par le paramètre $t \geq 1$, qui se réfère en fait au facteur d'étirement du t-couvrant.

3.1.1 Notations et définitions

En plus des définitions présentées dans le chapitre 2 (def. 2.1.2) nous avons besoins de quelques précisions supplémentaires.

Pour un graphe connexe, un k-séparateur est un ensemble de k sommets dont la suppression déconnecte le graphe. Un graphe G est *non séparable* s'il n'admet aucun 1-séparateur et *3-connexe* s'il n'a aucun 2-séparateur. Un

bloc d'un graphe est un sous-graphe induit maximal non séparable, et une composante 3-connexe d'un graphe est un sous-graphe induit maximal 3-connexe.
Un sommet est *universel* s'il est voisin de tous les autres noeuds du graphe.
Une *étoile* est un graphe biparti complet $K_{1,n}$ avec $n \geq 1$.
Une arête e est une *arête frontière* si ses extrémités forment un 2-séparateur minimal.

Théorème 3.1.1. *Soit H un graphe partiel d'un graphe $G = (V, E)$.*
Les affirmations suivantes sont équivalentes :

1. *H est un t-couvrant de G (i.e, $d_H(x, y) \leq t.d_G(x, y)$ pour toute paire de sommets $x, y \in V$)*
2. *Pour toute arête $(x, y) \in E - E(H), d_H(x, y) \leq t$*

Dans un premier temps nous présenterons l'algorithme de vérification de l'existence d'un arbre t-couvrant donné dans un graphe à l'aide d'un algorithme de complexité $O(m)$. Ensuite, nous exposerons la recherche d'arbres t-couvrants dans les graphes non valués. Nous montrerons qu'un arbre 2-couvrant peut être construit en temps linéaire. Egalement, nous évoquerons les résultats pour d'autres valeurs de t.

3.1.2 Vérifier un arbre t-couvrant

La définition intuitive du problème de vérification est la suivante : soit un graphe G, un arbre couvrant T, et un nombre entier $t \geq 1$, nous désirons vérifier si T est un arbre t-couvrant de G. Nous voudrions également déterminer son *indice d'étirement* (i.e le plus petit t pour lequel T est un t-couvrant).

Théorème 3.1.2. *Soit G un graphe. Soit T un arbre couvrant de G. Déterminer l'*indice d'étirement *de T peut être effectué en temps $O(m)$.*

Afin de vérifier l'existence d'un arbre t-couvrant nous nous servons de l'affirmation suivante :
Soit t un entier donné, T est un arbre t-couvrant du graphe G si et seulement si,

$$\text{pour toute arête } (x, y) \in E(G) - E(T), d_T(x, y) \leq t$$

L'idée intuitive de cette approche est de calculer le facteur d'étirement de chaque arête du co-arbre afin de vérifier qu'il ne dépasse pas un t donné. Par cette approche, il nous faut donc calculer les distances dans T de toutes les paires de noeuds définies par les arêtes du co-arbre (soit $m - n + 1$

paires). Si nous calculons la distance de chaque paire de noeuds directement et indépendamment, cela peut prendre un temps de $O(mn)$ vu que chaque distance peut prendre un temps de $O(n)$ pour être calculée.

Cependant une autre solution peut être employée afin de calculer plus efficacement la distance entre deux noeuds dans un arbre. Soit r un noeud choisi arbitrairement comme racine de T, nous étiquetons les noeuds de T tels que la distance entre deux noeuds x et y soit rapidement calculable à partir des étiquettes (*label(x)* et *label(y)*) de x, y et du $NCA(x, y)$.
Notons que, $d_T(x, y) = d_T(x, NCA(x, y)) + d_T(NCA(x, y), y)$.

Algorithme 1 Algorithme de vérification de validité d'un t-couvrant

Entrée: Un graphe G, un arbre couvrant T et un nombre positif t.
Sortie: Vrai si T est un arbre t-couvrant, Faux sinon.

1: Choisir arbitrairement un noeud r comme racine de T ;
2: calculer une étiquette $label(x)$ pour chaque noeud x de T ;
3: calculer NCA(x,y) pour chaque arête (x, y) du co-arbre ;
4: **Pour** chaque arête (x, y) du co-arbre **faire**
5:     Calculer $d_T(x, y)$ en utilisant les étiquettes de x, y et $NCA(x, y)$;
6:     **Si** $d_T(x, y) > t$ **alors**
7:         **retourner** Faux ;
8:     **Fin Si**
9: **Fin Pour**
10: **retourner** Faux ;

Par l'affirmation 2 du théorème (3.1.1) on note que l'*indice d'étirement* de T est égal à :

$$\max\{1, d_T(x, y) | (x, y) \in E(G) - E(T)\}$$

Cet algorithme peut facilement être modifié à la ligne 6 afin de calculer l'*indice d'étirement*.

Vis à vis de la complexité de l'algorithme, nous voyons que la ligne 1 est triviale. Afin d'effectuer le calcul de la ligne 3, on utilise un algorithme en temps linéaire de calcul de NCA tel que celui de Harel et Tarjan [8]. Clairement, la ligne 6 est exécutée en temps constant. Pour la ligne 6 et 2 nous étudions maintenant le cas des graphes non orientés.

La méthode d'étiquetage employée est la suivante : soit x un noeud, l'étiquette de x est égale à la distance de x à la racine[1], i.e, label(x)=$d_T(r, x)$. Voir la figure 3.1 pour un exemple.

[1]ou sa profondeur dans le cas de graphe non valué

On montre ici que pour toute paire de noeuds x, y, leur distance $d_T(x, y)$ dans l'arbre peut être calculée en temps constant à partir des étiquettes label(x), label(y) et label(NCA(x, y)). Notons que

$$label(x) = d_T(r, NCA(x, y)) + d_T(NCA(x, y), x)$$

$$label(y) = d_T(r, NCA(x, y)) + d_T(NCA(x, y), y)$$

Nous obtenons,

$$d_T(x, y) = label(x) + label(y) - 2\dot{l}abel(NCA(x, y))$$

Ainsi, $d_T(x, y)$ peut être calculé en temps constant.

Il est facile de voir que par un parcours en largeur démarré à partir de la racine r, on peut obtenir les étiquettes des noeuds de T. La ligne 2 de l'algorithme peut donc être effectuée en temps linéaire. De plus, la ligne 5 peut être exécutée en temps constant. Ainsi, la boucle *for* peut être exécutée en temps linéaire. Par conséquent, le temps global de l'algorithme est linéaire. Ainsi, le problème de la vérification de l'arbre t-couvrant prend un temps linéaire.

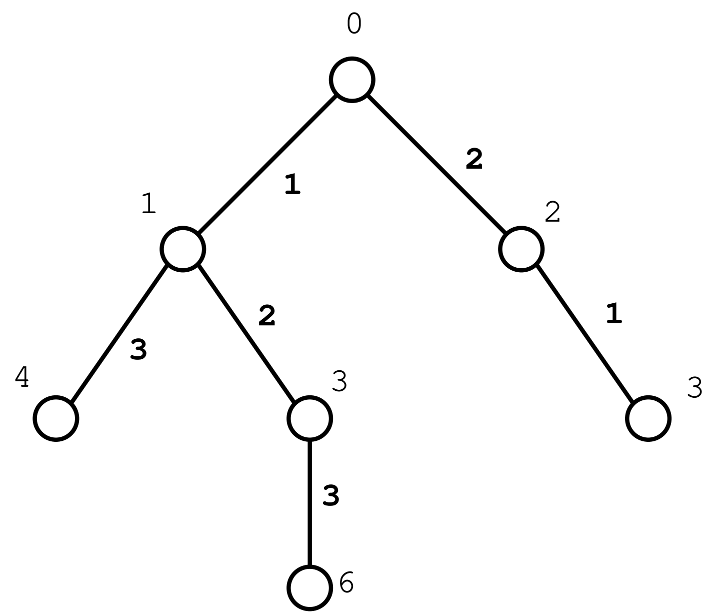


FIG. 3.1 – Etiquetage des noeuds de T par leur distance par rapport à la racine (l'arbre est ici valué).

3.1.3 Existence d'un arbre t-couvrant dans un graphe non valué

Notons que les graphes non valués ne sont qu'un cas spécial des graphes valués dont les arêtes ont un poids unitaire. Par l'affirmation 2 du théorème (3.1.1), un graphe partiel H est un t-couvrant d'un graphe $G = (V, E)$ si et seulement si pour chaque arête $(x, y) \in E - E(H)$ on a $d_H(x, y) \leq t$.

Clairement, un graphe contient un arbre 1-couvrant si et seulement s'il est lui même un arbre. Le problème de l'arbre t-couvrant ($t \geq 4$) est NP-complet et le problème pour $t = 3$ reste ouvert[2].

Pour $t = 2$, l'article utilise certaines propriétés de la structure des arbres 2-couvrants pour en arriver à au théorème :

Théorème 3.1.3. *(Bondy [1]). Un graphe G admet un arbre 2-couvrant si et seulement si chaque bloc H de G est soit,*

1. *un graphe 3-connexe avec un sommet universel ou*
2. *une arête e formant une paire d'articulation de deux graphes admettant un arbre 2-couvrant, chacun d'eux contenant e.*

Théorème 3.1.4. *Un arbre 2-couvrant d'un graphe (s'il existe) peut se faire en temps linéaire.*

L'algorithme de calcul d'un arbre 2-couvrant T d'un graphe G peut s'exposer comme suit : d'abord, trouver tous les blocs de G. Ensuite, pour chaque bloc, trouver tous les 2-séparateurs [3] et les composantes 3-connexes. Inclure toutes les arêtes induisant des arêtes frontières dans T. Ensuite, pour chaque composante 3-connexe H, trouver une étoile couvrante contenant toutes les arêtes de H ayant déjà été incluses dans T et l'inclure dans T [4]. Enfin, si T est un arbre couvrant, alors c'est un arbre 2-couvrant ; sinon, G n'admet pas d'arbre 2-couvrant.

Cet algorithme peut être implémenté en temps linéaire en utilisant l'algorithme de recherche des composantes 3-connexes d'Hopcroft et Tarjan [11].

3.2 Plus petit ancêtre commun

Nous résumons ici un article de Stephen Alstrup, Cyril Gavoille, Haim Kaplan et Theis Rauhe[2] (*Identifying nearest common ancestor in a distri-*

[2]Le lecteur intéressé pourra se reporter à l'article p.17-20 afin d'approfondir la preuve de NP-complétude pour $t \geq 4$

[3]s'il existe un 2-séparateur n'induisant pas d'arête frontière, alors G n'admet pas d'arbre 2-couvrant

[4]si une telle étoile couvrante n'existe pas, alors G n'admet pas d'arbre 2-couvrant

buted environment).

Cet article traite d'un algorithme qui étiquette les noeuds d'un arbre enraciné en temps linéaire et permet de retrouver l'étiquette de l'ancêtre commun le plus proche de deux noeuds (NCA) à partir de l'étiquette de ces deux noeuds en temps constant.

Il est à noter que des travaux concernant le calcul du NCA avaient déjà été accomplis. Harel et Tarjan [7, 8] furent les premiers à trouver un tel algorithme de complexité linéaire en temps. Par la suite, plusieurs recherches furent menées afin de simplifier et d'optimiser certaines constantes de cet algorithme. L'avantage de cet article est de présenter un algorithme aussi efficace que celui de Harel et Tarjan mais plus abordable à l'implémentation. De plus, cet algorithme peut être distribué.

3.2.1 Notations et définitions

Pour un arbre T :

- $|T|$ est son nombre de noeuds.
- T_v est le sous-arbre enraciné en v
- taille$(v) = |T_v|$
- pere(v) est le père de v
- fils(v) est l'ensemble des fils de v
- chaque noeud est soit *lourd* soit *léger* comme suit :
  . la racine est légère
  . pour chaque noeud interne v, nous choisissons un fils w de v, où

$$taille(w) = \max_{z \in fils(v)} taille(z)$$

  et qualifions w de *lourd*. Les fils restant de v sont *légers*
- une arête menant à un *fils léger* est appelée *arête légère*
- l'arête menant au *fils lourd* est appelée *arête lourde*
- pour un noeud v avec un fils lourd w, $lsize(v) = taille(v) - taille(w)$
- l'ancêtre le plus proche de v qui est *léger* (v lui-même si v est léger) est nommé $apex(v)$
- en supprimant les arêtes légères, T est partitionné en chemins appelés *chemins lourds* : $HP(w) = \{v|v \in T, apex(v) = apex(w)\}$

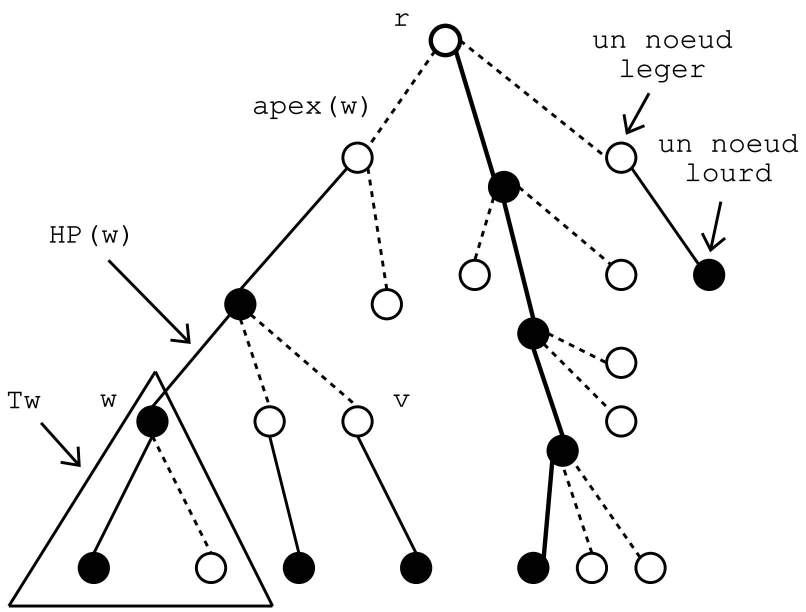


FIG. 3.2 – Exemple d'arbre ayant été traité par la première partie de l'algorithme

3.2.2 Le processus de calcul des étiquettes

Etape 1 : La première partie consiste à calculer taille(v), lsize(v), apex(v) et partitionner T en *chemins lourds*

Etape 2 : Ensuite, deux étiquettes, *llabel* (étiquette légère) et *hlabel* (étiquette lourde) sont assignées à chaque noeud comme suit :

<u>llabel</u> : pour chaque noeud w,

$llabel(w) \notin \{llabel(z) \mid z \neq w, z \in fils(pere(w))\}$,

ce qui signifie que chaque fils *léger* d'un même noeud doit avoir un llabel différent

<u>hlabel</u> : pour chaque noeud w,

$hlabel(w) <_{lex} min_{lex}\{hlabel(z) \mid z \neq w, z \in T_w \cap HP(w)\}$,

ce qui signifie que le hlabel de w doit être inférieur[5] au hlabel des noeuds se trouvant sur le même chemin lourd *au dessous* lui. (plus le hlabel est grand, plus le noeud est bas dans le chemin lourd).

Etape 3 : Puis, une étiquette $l(v)$ est assignée à chaque noeud de haut en bas comme suit [6] :

$l(v) = l(pere(apex(v))).llabel(apex(v)).hlabel(v)$.

[5] $<_l ex \longleftrightarrow$ ordre lexicographique sur les chaines binaires

[6] $llabel(root) = \varepsilon$ et $l(pere(root)) = \varepsilon$

Etape 4 : Ainsi, une étiquette $l(v)$ est composée d'une suite de concaténation de *hlabel* et de *llabel* : $l(v) = h_1 \cdot l_1 \cdot h_2 \cdot l_2 \cdots$. Pour pouvoir effectuer des requêtes pour retrouver l'ancêtre commun le plus proche, on doit savoir quand $l(v)[i]$ est le début d'une étiquette lourde ou légère. Donc, en plus de $l(v)$, une étiquette supplémentaire $k(v)$ de la même longueur est requise, où $k(v)[i] = 1$ si et seulement si $l(v)[i]$ est le début d'une étiquette lourde ou légère.

Dernière étape : L'étiquette $label(v)$ assignée à un noeud est composée de la concaténation de $l(v)$ et de $k(v)$.

3.2.3 Calcul de l'étiquette de l'ancêtre commun le plus proche

Soient x et y deux noeuds de T, $l(x)$ et $l(y)$ leur étiquette et h_i et l_i respectivement une étiquette lourde et légère.

1. Si $l(x) = h_1 \cdot l_1 \cdots h_i \cdots$ et $l(y) = h_1 \cdot l_1 \cdots h'_i \cdots$, où $h_i \neq h'_i$, alors $l(nca(x, y)) = h_1 \cdot l_1 \cdots h_{i-1} \cdot l_{i-1} \cdot min_{lex}\{h_i, h'_i\} \cdot$
   ce qui signifie que $label(x)$, $label(y)$ diffèrent sur une étiquette lourde
2. Si $l(x) = h_1 \cdot l_1 \cdots h_i \cdot l_i \cdot t$ et $l(y) = h_1 \cdot l_1 \cdots h_i \cdot l'_i \cdots$, où $l_i \neq l'_i$ ou $l_i \cdot t$ est vide, alors $l(nca(x, y)) = h_1 \cdot l_1 \cdots h_{i-1} \cdot l_{i-1} \cdot h_i$
   ce qui signifie que $label(x)$, $label(y)$ diffèrent sur l'étiquette lourde l_p.

Pour en finir avec cet article il est à noter que cet algorithme permet de retrouver l'étiquette de l'ancêtre commun le plus proche à partir des labels de deux noeuds. Si l'on veut récupérer le noeud lui même au lieu de son étiquette, il est opportun de créer une table indexée par l'étiquette du noeud et ayant pour valeur le noeud lui même.

3.3 Algorithme *multi-échelle* pour le problème de l'arrangement linéaire

Le problème de l'arrangement linéaire des sommets d'un graphe est un problème commun à diverses applications. L'article de Y. Koren et D. Harel [12] intitulé *A Multi-Scale Algorithm for the Linear Arrangement Problem* s'intéresse à une variante de ce problème connue sous le nom du *problème de l'arrangement linéaire minimum* (*minLA*[7]). Un nouvel algorithme basé sur le paradigme de *multi-échelle*[8] qui s'exécute en temps linéaire est présenté.

[7] minimum linear arrangement
[8] *multi-scale*

Le but du *minLA* est de placer n sommets en position $1 \cdots n$ sur une ligne de manière à minimiser la somme des longueurs des arêtes multipliées par leur poids.

Comme la plupart des problèmes d'ordonnancement, *minLA* a été prouvé comme étant NP-difficile et le problème de décision associé est NP-complet [9].

Ainsi, l'article présente une heuristique en temps linéaire pour le problème *minLA*. Le lecteur pourra également se référer à l'article pour avoir des comparatifs avec d'autres méthodes.

Notre résumé s'organisera comme suit : nous présenterons tout d'abord le problème de l'arrangement linéaire minimum pour ensuite exposer une vue d'ensemble de l'algorithme employé.

3.3.1 Le problème de l'arrangement linéaire minimum

Soit $G = (V, E)$ un graphe, avec $V = \{1, \cdots, n\}$ et E un ensemble d'arêtes valuées, w_{ij} représentant le poids de l'arête (i, j) .
$N(i) = \{j | (i, j) \in E\}$ est appelé le voisinage de i. Egalement, l'article prend pour hypothèse que G est connexe.

Définition 3.3.1. *Un arrangement linéaire de G est une bijection $\pi : V \rightarrow \{1, \cdots n\}$. De manière équivalente, nous pouvons définir π comme une permutation de V. $\pi(i)$ est la position du sommet i.*

Définition 3.3.2. *Le coût de l'arrangement linéaire π du graphe G est définit par :*

$$LA_\pi(G) \stackrel{def}{=} \sum_{(i,j) \in E} w_{ij} \cdot \ |\pi(i) - \pi(j)|$$

L'objectif du problème du *minLA* est de trouver une permutation π minimisant $LA_\pi(G)$.

3.3.2 L'algorithme exact

Comme cité précédemment, la résolution du problème *minLA* est NP-difficile ainsi, les recherches se sont plutôt axées sur des algorithmes d'approximations et des heuristiques. Avant d'entrer dans la méthode employée ici, nous présentons un algorithme exact, meilleur que l'algorithme naïf de complexité en temps $O(n!)$.

Cet algorithme s'exécute avec une complexité en temps de $O(2^n \cdot |E|)$ et en espace de $O(2^n)$ en recherchant la solution exacte du problème *minLA*. Soit un ensemble S placé à la droite des sommets d'un ensemble L et à la gauche des sommets d'un ensemble R. L'algorithme affirme que le meilleur ordre de S est indépendant de l'ordre interne de L et de R. L'idée principale est de minimiser le coût de l'arrangement linéaire de manière locale à tous les sous-ensembles des sommets du graphe. Le lecteur désirant une description plus formelle de ce problème pourra se reporter à l'article page 2 à 4.

Algorithme 2 Algorithme exact de minLA

Entrée: Un graphe $G = (V, E)$.

Sortie: Un vecteur *Ordre* contenant l'arrangement linéaire minimum de V.

Une table T indexée par les sous-ensembles de$V$.

Pour un $S \subseteq V, T[S].cout$ contient le coût de l'arrangement local minimal de S,

$T[S].coupe$ est la somme des poids des arêtes connectant S avec $V - S$,

$T[S].sommet\_droit$ est le sommet le plus à droite dans le meilleur arrangement local de S.

La fonction $Coupe^G(v, Noeuds)$ retourne la somme des poids des arêtes connectant le sommet v et $Noeuds \subset V$.

Pour chaque $S \subset V$ **faire**
- $table[S].cout = \infty$

Fin Pour

$table[\phi].cout = 0$

$table[\phi].coupe = 0$

Pour $i = 1$ à n **faire**
- **Pour** chaque $S \subset V, |S| = i - 1$ **faire**
  - $coupe^S = table[S].coupe$
  - $nouveau\_cout = table[S].cout + coupe^S$
  - **Pour** chaque $j \notin S$ **faire**
    - **Si** $table[S \cup \{j\}].cout > nouveau\_cout$ **alors**
      - $table[S \cup \{j\}].cout = nouveau\_cout$
      - $table[S \cup \{j\}].sommet\_droit = j$
      - $table[S \cup \{j\}].coupe = coupe^S - Coupe^S(j, S) + Coupe^S(j, V - S)$
    - **Fin Si**
  - **Fin Pour**
- **Fin Pour**

Fin Pour

Bien qu'impraticable au vu de sa complexité, cet algorithme tient quand même un rôle dans l'algorithme présenté dans l'article.

3.3.3 L'approche *multi-échelle*

Le paradigme de l'approche *multi-échelle* est une technique permettant l'exploration de propriétés liées à la structure globale d'objets complexes. Ces techniques expriment un problème originalement de haute dimension en un problème de dimension de plus en plus basse durant un procédé appelé diminution[9] du problème. Les parties du problème les plus réduites sont résolues exactement, il s'entame alors un processus de *raffinement* dans lequel la solution est progressivement ramenée à des dimensions plus haute en faisant des mises à jours et ce, jusqu'à ce que le problème orignal soit reproduit.

Autres algorithme d'approximation du *minLA* Nous décrivons ici brièvement les divers algorithmes résolvant le problème du *minLA* existant. Notons que ces algorithmes tiennent également une place dans la méthode présentée dans l'article.

L'itération du médian : Cette méthode est un algorithme aléatoire itératif réduisant le coût de l'arrangement linéaire. Le coeur de la méthode effectue un relâchement continuel du problème *minLA*, on autorise notamment à deux sommets de partager la même position où d'être placés sur des positions non entières. Le problème résolu se nomme le problème de minimisation du coût du *placement linéaire.*

Définition 3.3.3. *Le placement linéaire* p *du graphe* G *est une fonction* $p : V \longrightarrow \mathbb{R}$ *dont le coût se définit par :*

$$LP_p(G) \stackrel{ref}{=} \sum_{(i,j)\in E} w_{ij} \cdot \; |p(i) - p(j)|$$

Notons que le problème du *placement linéaire* est une généralisation de l'*arrangement linéaire* à des positions réelles. Le placement minimum de $LP_p(G)$ est lorsque tous les sommets sont au même endroit cependant, on ne retire aucune information d'un tel placement. Ce qui est utile est le *processus* de minimisation de $LP_p(G)$.

[9] le terme original est *coarsening* qui signifie *rendre grossier*

Tout d'abord définissons le médian des places des voisins de i.

Définition 3.3.4. *Soit $G = (V, E)$ un graphe et p un placement linéaire. Soit un sommet $i \in V$, le médian des voisins de i est noté $med_p^G(i)$. $med_p^G(i)$ doit satisfaire :*

$$\sum_{j \in N(i), p(i) \leq med_p^G(i)} w_{ij} \geq \sum_{j \in N(i), p(i) > med_p^G(i)} w_{ij}$$

$$\sum_{j \in N(i), p(i) \geq med_p^G(i)} w_{ij} \geq \sum_{j \in N(i), p(i) < med_p^G(i)} w_{ij}$$

Lorsqu'il y a égalité, le médian est n'importe quel point dans un intervalle entre deux voisins de i, sinon sa position est unique et correspond à un voisin de i.

Voici, l'observation principale à noter :

Définition 3.3.5. *Soit $G = (V, E)$ un graphe et p un placement linéaire. On fixe les places de tous les sommets sauf un ($i \in V(G)$). La position qui minimise le coût du placement linéaire est $med_p^G(i)$. (Cette place n'est pas forcément unique.)*

Nous présentons ici l'algorithme du médian :

Algorithme 3 Algorithme de l'itération du médian

Entrée: Un graphe $G = (V, E)$, un placement linéaire p, un nombre d'itérations k.
Sortie: Le placement p.

```
Répéter k fois faire
    Pour chaque i ∈ V(G) faire
        p(i) ← med_p^G(i)
    Fin Pour
Fin Répéter
Pour chaque i ∈ V(G) faire
    p(i) ← place de i une fois triée
Fin Pour
```

On suppose que la valeur initiale de p est un arrangement linéaire valide. Notons que pour des valeurs de k pas trop grandes on obtiendra certainement un grand nombre de petits clusters placés au même endroit. A partir de ce placement linéaire on peut générer un arrangement

linéaire en décidant aléatoirement de la position des sommets à la même place.
Au total, on remarque donc que cette méthode effectue son ordonnancement en utilisant la structure globale du graphe alors que les décisions qu'elle prend lors des arrangements locaux sont faites aléatoirement.

Le séquencement spectral : L'idée du séquencement spectral est de calculer le vecteur propre du Laplacien correspondant à la deuxième plus petite valeur propre, ce vecteur minimisant l'énergie de Hall.

$$E(v) = \sum_{(i,j)\in E(G)} w_{ij} \cdot (v_i - v_j)^2$$

Nous remarquons que la formule de l'énergie de Hall est proche de celle de la réduction du coût de l'arrangement linéaire (def. 3.3.2) à ceci près que la distance est élevée au carré. En utilisant le séquencement spectral avec l'itération du médian on arrive selon l'article à des résultats satisfaisants.

Le recuit simulé : Le recuit simulé est en fait une technique générale pouvant s'appliquer au problème du *minLA*. L'idée est : soit un arrangement initial π, on génère un autre arrangement π' proche de π. Si le nouvel arrangement π' à un coût inférieur au précédent, alors il est accepté. Sinon, π' peut être accepté avec une probabilité qui diminue au fur et à mesure que le processus se déroule. Le fait d'accepter un ordre "pire" permet en fait d'échapper à de mauvais minimum locaux. Le défaut de cette technique est qu'elle est relativement lente cependant, d'après Petit [15] elle est très efficace.

Les s-graphes Un s-graphe (pour segment-graphe) est un graphe dont les sommets sont des segments de droite. Lors du traitement de s-graphes on conserve la longueur des sommets ainsi que les positions des points de départ et d'arrivée des arêtes au sein d'un sommet. L'article généralise en fait le problème d'arrangement linéaire initial. L'objectif étant de préserver la nature du problème pendant sa réduction.

L'algorithme effectue les arrangements linéaires sur des s-graphes en plaçant les sommets à arranger sur une ligne de manière à ce que deux sommets ne s'intersectent pas et de manière à minimiser la longueur des arêtes. Les algorithmes usuels utilisés étant prévus pour des graphes standard, l'article décrit comment les modifier de manière à ce qu'ils soient applicables aux s-graphes.

Le procédé de diminution Pendant la réduction, on représente un s-graphe initial en s-graphes de plus en plus petits. Lors d'une itération de réduction on remplace un s-graphe G par un autre s-graphe G^R ayant un plus petit nombre de noeuds (typiquement $n/2$). Bien entendu, la structure de G^R devrait être très liée à celle de G de manière à ce que les deux graphes décrivent la même entité mais à différentes échelles. Dans ces conditions, un bon arrangement linéaire de G^R devrait correspondre alors à un bon arrangement linéaire de G.

Notons que ce procédé est différent du *diviser pour régner* dans le sens où le graphe n'est pas découpé mais plutôt *transformé* en un graphe plus petit respectant la structure globale du graphe original.

L'attitude adoptée pendant ce procédé est de poser des restrictions sur les arrangements possibles de manière à transformer le problème en *dimension plus basse*. La restriction adoptée est d'imposer à certaines paires de sommets d'être placées côte à côte. Le tout en espérant que ces paires de sommets soient relativement proche dans l'arrangement minimal de G.

Restriction de l'espace de recherche : les buts à satisfaire sont les suivants.

1. Les degrés de liberté devraient être réduit de manière significative pour permettre une représentation par un graphe plus petit.
2. L'arrangement minimal est affecté localement seulement, alors que sa structure globale est préservée.

Formulation du processus de réduction : Soit un s-graphe $G = (V, E)$, et un ensemble $R \subset V \times V$ de paires de sommets restreintes, on construit un graphe réduit $G^R = (V^R, E^R)$ tel qu'il y ait une correspondance 1-1 entre les arrangements linéaires de G^R et ceux de espace restreint de G.

G^R est fabriqué en contractant des paires de sommets restreints. La longueur des sommets et le poids des arêtes sont préservés en les accumulant. La place des nouvelles arêtes sont calculées en faisant la moyennes des anciennes positions, pondérée par leur poids respectif.

Ainsi, on espère qu'un bon arrangement linéaire G^R correspond à un assez bon arrangement de G. Cet "assez bon" arrangement peut maintenant être amélioré via le *processus de raffinement local.*

Le processus de rafinement local : Soit un arrangement linéaire π, on cherche une méthode qui améliore la qualité de π sur un petit nombre de sommets. Etant donné que l'arrangement à améliorer se situe dans un contexte local avec peu d'éléments, on peut se permettre d'utiliser l'algorithme exact de résolution du problème du *minLA*(algo. 2). L'idée est qu'effectuer un raffinement local en combinaison avec l'itération du médian donne de bons résultats, la technique du médian opérant sur la structure globale du graphe.

L'algorithme est assez simple, il effectue plusieurs itérations (5 typiquement) où le graphe est découpé en plusieurs petits ensembles (l'article recommande 6 sommets) de sommets sur lesquels on lance l'algorithme 2 (page 19).

Le processus complet : Ainsi, nous avons tous ce qui est nécessaire à la réalisation du processus *multi-échelle*.

Etape d'initialisation : Avant de se lancer dans le processus, on veut obtenir de manière assez rapide un arrangement linéaire relativement correct. Pour ce faire, la méthode utilise le séquencement spectral puis effectue plusieurs itérations du médian afin de raffiner le résultat.

Le cycle en V : Le cycle en V est l'outil principal de la méthode, il commence par effectuer un raffinement localement de l'arrangement du graphe. L'intention étant non seulement de réduire le coût, mais également d'améliorer la qualité de la prochaine étape de diminution. On résout alors le problème dans l'espace de solution restreint en lançant le processus récursivement sur le graphe ainsi réduit. La récursion s'arrête lorsque le graphe est suffisamment réduit pour pouvoir lancer l'algorithme exact exposé page 19. Une fois que l'on a trouvé une solution dans l'espace restreint, on raffine cette solution localement puis un processus est lancé afin de faire "remonter" le raffinement dans la dimension du dessus et ce, jusqu'à que l'on revienne au graphe original.

Algorithme 4 Algorithme du Cycle en V

Entrée: Un graphe $G = (V, E)$, un arrangement linéaire $Ordre$.
Sortie: L'arrangement $Ordre$.
  $Raffinement\_Local(G, Ordre)$
  Si G est "assez" gros **alors**
    $Reduire(G, Ordre, G^R, Ordre^R)$
    $Cycle\_En\_V(G^R, Ordre^R)$
    $interpoler(G^R, Ordre^R, Ordre)$
    $Raffinement\_Local(G, Ordre)$
  Fin Si

Le cycle en V amélioré : Le cycle en V peut être amélioré en lui faisant bénéficier d'une initialisation. De plus, réaliser plusieurs passages successifs du cycle en V améliore de façon significative le résultat pour un petit nombre d'itérations. Au delà, le processus converge et les améliorations deviennent lentes.

Une deuxième amélioration notable est de perturber l'arrangement avant de faire une nouvelle itération du cycle en V via l'utilisation de l'itération du médian. Ceci à pour effet :

1. d'éviter la convergence vers des minimums locaux non désirés,
2. d'améliorer globalement l'arrangement sans se centrer sur des améliorations locales.

Ces deux avantages proviennent de la nature même de l'algorithme de l'itération du médian (algo. 3 page 21).

Algorithme 5 Algorithme multi-échelle final

Entrée: Un graphe $G = (V, E)$, un arrangement linéaire $Ordre$, un nombre d'itérations $Iteration$ pour le cycle en V.
Sortie: L'arrangement p.
  $k_1 = 40$ (nombre de passages de la première itération du médian)
  $k_2 = 10$ (nombre de passages dans les autres itérations du médian)
  $Sequencement\_Spectral(G, Ordre)$
  $Iteration\_Median(G, Ordre, k_1)$
  Pour $i = 1$ à $Iteration$ **faire**
    $Iteration\_Median(G, Ordre, k_2)$
    $Cycle\_En\_V(G, Ordre)$
    $Diminuer(k_2)$
  Fin Pour

En conclusion, l'article présente des comparatifs avec d'autres méthodes étudiées dans l'article de Petit [15]. La conclusion est que l'encapsulation du raffinement local dans l'algorithme multi-échelle a donné de très bon résultats aussi bien au niveau de la qualité que des performances. La qualité de l'algorithme multi-échelle est très comparable à celle du recuit simulé de Petit mais beaucoup plus rapide.

4 Méthode d'affichage proposée

C'est un peu dans la veine de la métrique de Strahler que nous avons conçue notre méthode (voir figures 2.2(a) et 2.2(b), page 9). Nous essayons de n'afficher qu'une partie des éléments du graphe de manière à réduire le "coût" global de l'affichage.

Nous définissons le *coût* d'affichage d'un élément comme étant proportionnel à la place que prend l'élément dans l'écran lors de son affichage.

Ainsi, l'affichage d'un noeud à un "coût" relativement faible en terme de visualisation, la place couverte n'étant que celle d'un point sur l'écran. L'affichage d'une arête par contre, coûte nettement plus cher étant donné que le tracé d'une ligne est nécessaire à sa représentation.

C'est par conséquent sur les arêtes à afficher que s'est portée notre attention.

4.1 Affichage par arbre couvrant

La base de notre méthode repose en fait sur l'affichage d'un arbre couvrant du graphe.

Sauf mention contraire, le graphe de la figure 4.1 qui servira d'exemple le long de notre étude, notons qu'il a pour seule propriété d'être un graphe simple. Il a été généré par Tulip avec 100 noeuds et 250 arêtes.

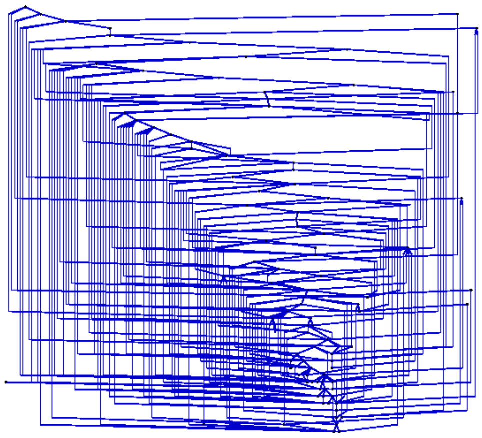


FIG. 4.1 – Graphe affiché à l'aide de l'algorithme d'affichage standard

4.1.1 Choix de l'arbre couvrant

Dans le cadre de cette étude, nous ne nous intéressons pas au choix de l'arbre couvrant. La solution choisie étant de réaliser un simple parcours en largeur de manière à obtenir un arbre de plus courte chaîne correctement équilibré (voir plugin. 1, page 42). Nous discuterons cependant de ce choix et du choix d'autres arbres couvrants dans les perspectives d'améliorations.

Voici l'arbre couvrant choisi :

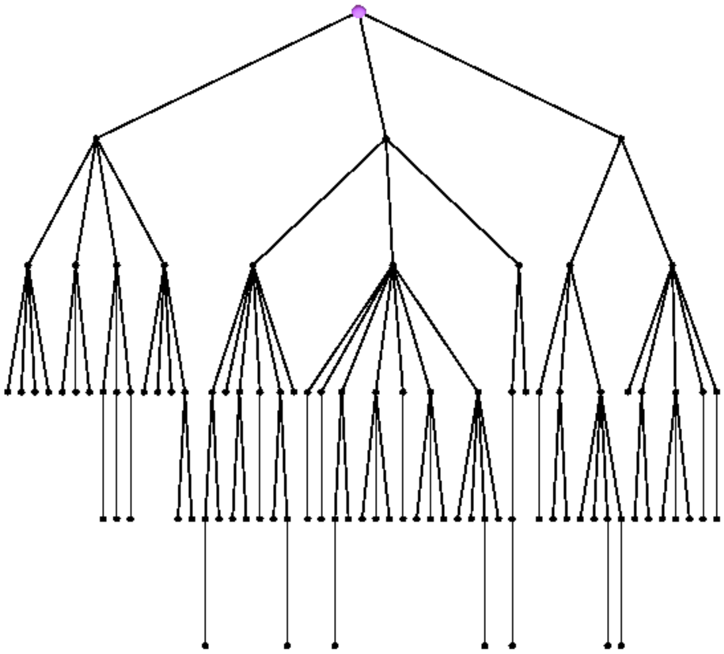


FIG. 4.2 – Arbre couvrant choisi affiché à l'aide de l'algorithme de Reingold et Tilford [16].

4.1.2 Avantage de l'affichage par arbre couvrant

L'affichage d'un arbre a pour avantage d'avoir une complexité en temps faible, la plupart des algorithmes étant de complexité en temps linéaire. De ce fait, nous résolvons la partie (1) du problème 1, page 7.

Ensuite, le nombre restreint d'arêtes d'un arbre a pour conséquence directe de ne pas surcharger l'affichage et même de laisser une marge de manoeuvre supplémentaire pour l'affichage d'arêtes supplémentaires. Ainsi, nous résolvons la partie (2) du problème 1. Egalement, si le graphe est très gros notre méthode pourra tout à fait être combinée avec une autre méthode d'affichage partiel d'arbre telle que la technique utilisant la métrique de Strahler.

La technique de l'affichage par arbre couvrant est d'ailleurs déjà employée par certains logiciels de visualisation afin d'obtenir de meilleures performances. Nous citons ici l'article de Munzner concernant le logiciel H3Viewer [13] qui utilise cette méthode :

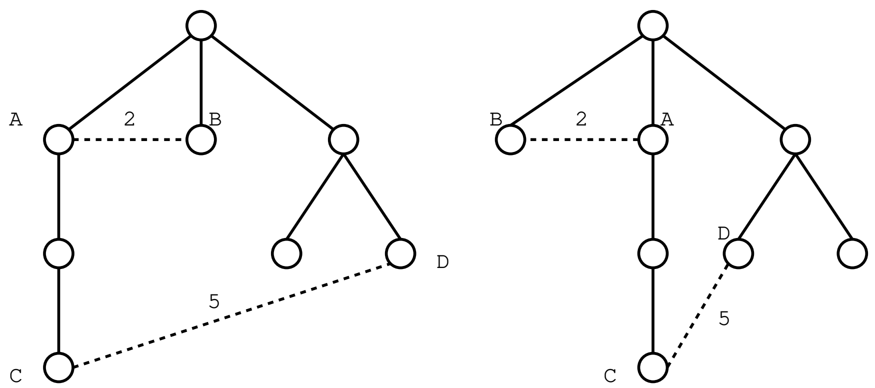
"On pourrait croire au premier abord que notre méthode n'est appropriée que pour des graphes qui ressemblent fortement à des arbres et qu'elle est inappropriée pour des graphes denses. Cependant, nous affirmons qu'il y a en fait beaucoup de graphe *quasi-hiérarchiques* pour lesquels notre algorithme est bien adapté."

4.1.3 Inconvénients induits

Cependant, en nous limitant à l'extraction et l'affichage d'un arbre couvrant quelconque nous nous trouvons face à des effets indésirables liés à *l'indice d'étirement* de l'arbre qui, s'il est élevé, induit un nouvel ensemble de problèmes :

Problème 2. *Lors de leur affichage, deux sommet liés par une arête du co-arbre de facteur d'étirement élevé ont de forte chance d'être représenté éloigné l'un de l'autre aussi bien*

1. *dans l'arbre,*
2. *que sur l'écran.*



(a) Affichage maladroit par arbre couvrant

(b) Affichage rectifié

Fig. 4.3 – Affichage d'un graphe par arbre couvrant

Le dessin figure 4.3(a) a été réalisé à partir d'un arbre couvrant. Les sommets A, B (resp. C, D) sont reliés par une arête de facteur d'étirement 2 (resp. 5), on remarque bien que plus le facteur d'étirement d'une arête est élevé, plus elle a tendance à être longue (ou étirée en l'occurrence). En permutant certains sommets entre eux, on arrive à réduire la longueur de l'arête (C, D) et on rapproche donc C et D (fig. 4.3(b)).

Ce sont les deux problèmes (problèmes 2) que nous avons visé à résoudre via l'exploitation du facteur d'étirement.

4.2 Exploitation du facteur d'étirement

4.2.1 Filtrage des arêtes

Notre objectif ici est de réduire la distance dans l'arbre de deux sommets liés par une arête du co-arbre (2 (1)). La meilleure solution pour ce problème serait bien entendu de rajouter toutes les arêtes du co-arbre, la distance entre les paires de sommets traitées redescendrait ainsi à 1. Cependant, comme nous le voyons dans la figure 4.4 la visibilité d'un tel affichage redevient très médiocre. De ce fait, nous entrons en conflit direct avec le problème 1 (page 7).

Nous devons donc trouver un moyen "intelligent" de rajouter des arêtes de manière à trouver un affichage optimal minimisant le problème de la distance dans l'arbre (2, partie 1) sans surcharger l'affichage (1, partie 2).

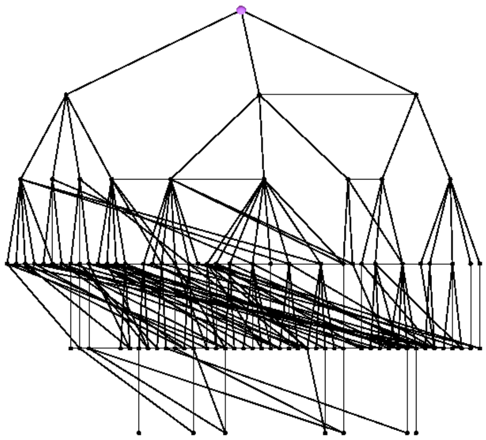


FIG. 4.4 – Affichage sans filtre : toutes les arêtes du co-arbre ont été rajoutées.

Afin d'améliorer cette technique nous proposons de ne pas afficher toutes les arêtes du co-arbre afin que l'affichage n'en soit pas alourdi. Un critère de choix s'avère donc indispensable afin de sélectionner les arêtes pertinentes à afficher (problème 1 (2)).

Critère 1. *Lors de l'affichage du graphe, la pertinence de l'affichage d'une arête du co-arbre est proportionnelle à son facteur d'étirement.*

Méthode La technique employée comporte plusieurs phases :

1. **extraction de l'arbre couvrant :** comme précisé précédemment, l'arbre couvrant choisi est un simple parcours en largeur qui sélectionne donc un arbre de plus courte chaîne enraciné en un noeud que sélectionne au préalable l'utilisateur. Quelques difficultés ont néanmoins été rencontrées ici lors de l'implémentation (voir plugin 1, page 42).
2. **calcul du facteur d'étirement des arêtes du co-arbre :** l'algorithme de calcul du facteur d'étirement provient d'une variante de l'algorithme vérifiant l'existence d'un arbre t-couvrant donné dans un graphe pour une valeur de t donnée. Cet algorithme est présenté dans l'article de Leizhen Cai et Corneil [6] résumé dans le chapitre 3.
   (a) *calcul des étiquettes de l'ancêtre commun le plus proche :* l'algorithme nécessitant un moyen de retrouver l'ancêtre commun le

plus proche d'un couple de noeud en temps constant, nous avons choisi l'algorithme d'Alstrup et al. [2] également résumé au chapitre 3. Cet algorithme a en fait été choisi pour deux raisons, la première est qu'il répond aux exigences de l'algorithme de Cai et Corneil[1] et la seconde est qu'il est d'une implémentation plus aisée que d'autres (voire le résumé).

(b) *correspondance des étiquettes vers les noeuds :* l'algorithme cité précédemment calcule en fait l'étiquette du NCA de deux noeuds en temps constant. Afin de retrouver la référence sur le NCA lui-même, on a besoin d'une table indexée par les étiquettes. Cette table est construite en temps linéaire et retourne, sous l'hypothèse d'une fonction de hachage correcte, la référence d'un noeud en temps constant.

(c) *calcul de la profondeur :* le calcul de la profondeur consiste en un simple parcours de l'arbre (temps linéaire donc). La récupération de cette valeur se faisant ensuite en temps constant.

(d) *calcul du facteur d'étirement :* nous calculons donc le facteur d'étirement de chaque arête xy du co-arbre comme suit :
$d_T(x,y) = d_T(x, NCA(x,y)) + d_T(NCA(x,y), y)$ or dans un arbre non valué $d_T(x,y) = profondeur(NCA(x,y)) - profondeur(x) + profondeur(NCA(x,y)) - profondeur(y)$. Ce calcul se fait en temps constant.

3. **dessin du graphe :** précisons que la notion de “dessin de graphe” employée ici dénote le calcul des coordonnées de chaque noeud du graphe plutôt que la représentation du graphe sur support physique.

   Nous utilisons ici un algorithme classique de dessin d'arbre, celui de Reingold et Tilford [16]. Il s'effectue en deux phases, d'abord il effectue une traversée de l'arbre en ordre postfixe, calcule des coordonnées temporaires et des modificateurs pour chaque noeud. Ensuite, une deuxième traversée est effectuée, cette fois en visitant les noeuds en ordre préfixe, les coordonnées du noeuds y sont calculées en cumulant les valeurs modificatrices de la racine jusqu'aux feuilles de l'arbre. La complexité en temps est prouvée linéaire (voir l'article pour les détails).

4. **clustering :** une fois la phase d'affichage réalisée, chaque noeud dispose d'une position, il reste maintenant à déterminer les arêtes à garder pour l'affichage. La méthode ici est de fixer une limite de facteur d'étirement pour les arêtes en dessous de laquelle une arête est ignorée.

[1] calcul des étiquettes en temps linéaire et calcul de l'étiquette du NCA(x,y) dans l'arbre en temps constant

Evidemment, plus la limite est basse et plus nous favorisons la réduction des distances dans l'arbre des sommets liés par des arêtes du co-arbre (problème 2 (1), page 29) au détriment de la visibilité (problème 1 (2), page 7). Inversement, le rehaussement de la limite diminue le nombre d'arêtes conservée et favorise la donc visibilité (en terme de surcharge à l'écran) au détriment des distances. Cette opération consistant en un simple parcours des arêtes, la complexité est linéaire.

5. **coloration :** pour des besoins esthétiques, nous avons rajouté une coloration différente sur les arêtes suivant leur facteur d'étirement. Cette coloration varie du bleu au rouge proportionnellement au facteur d'étirement de l'arête. De la même manière que le clustering, la coloration consiste en un parcours des arêtes la complexité en temps reste linéaire.

Résultats Voici quelques résultats d'affichage en prenant diverses limites en dessous desquelles les arêtes ne seront pas affichées.

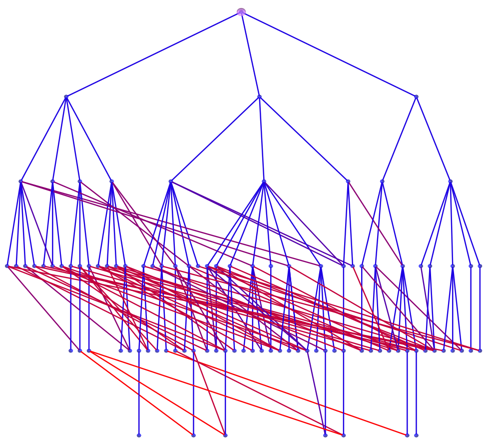


Fig. 4.5 – Affichage avec filtre de facteur d'étirement 3 : ici nous n'avons rajouté que les arêtes du co-arbre de facteur d'étirement de 3 ou plus

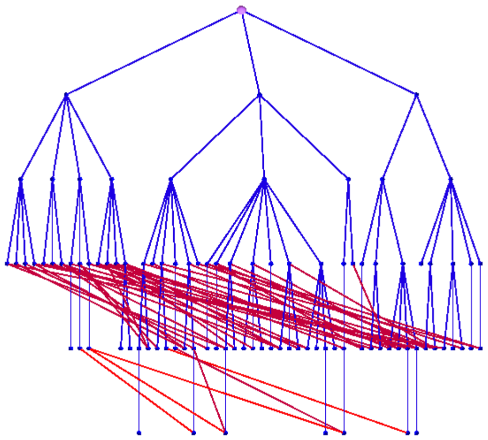


FIG. 4.6 – Affichage avec filtre de facteur d'étirement 7 : ici nous n'avons rajouté que les arêtes du co-arbre de facteur d'étirement de 7 ou plus

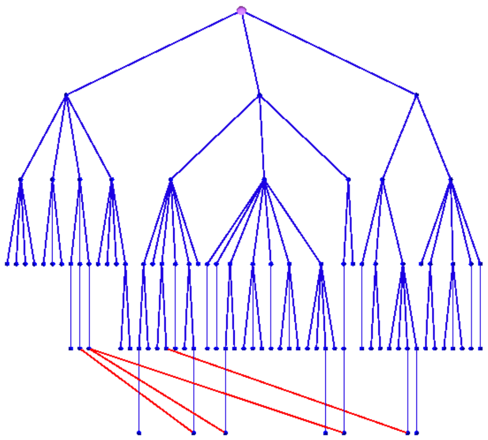


FIG. 4.7 – Affichage avec filtre de facteur d'étirement 9 : ici nous n'avons rajouté que les arêtes du co-arbre de facteur d'étirement de 9 ou plus

Notons qu'au dessus de 9, nous obtenons en fait uniquement l'arbre couvrant, aucune arête n'ayant un facteur d'étirement supérieur à neuf.

Du point de vue de la complexité, notre algorithme respecte la structure et les propriétés de celui présenté dans l'article de Cai et Corneil [6] à savoir qu'il calcule bien le facteur d'étirement des arêtes du co-arbre en temps linéaire.

4.2.2 Agencement des noeuds

Cette partie tend à résoudre la partie (2) du problème 2 à savoir réduire la distance (sur l'écran) entre deux noeuds connectés par une arête de facteur d'étirement élevé.

D'où notre second critère,

Critère 2. *Minimiser la longueur des arêtes de facteur d'étirement élevé.*

Soit T un arbre et i un sommet de T, on appelle T_i le sous-arbre enraciné en i. L'idée est la suivante. Pour chaque paire de fils x, y d'un sommet interne v, on crée une "arête virtuelle" $(xy)_{virtual}$ et on y affecte une valeur numérique reflétant la connexité entre les deux sous arbres T_x et T_y. Cette valeur se calcule en faisant la somme des facteur d'étirement des arêtes (x, y) du co-arbre liant les deux sous arbres T_x et T_y.

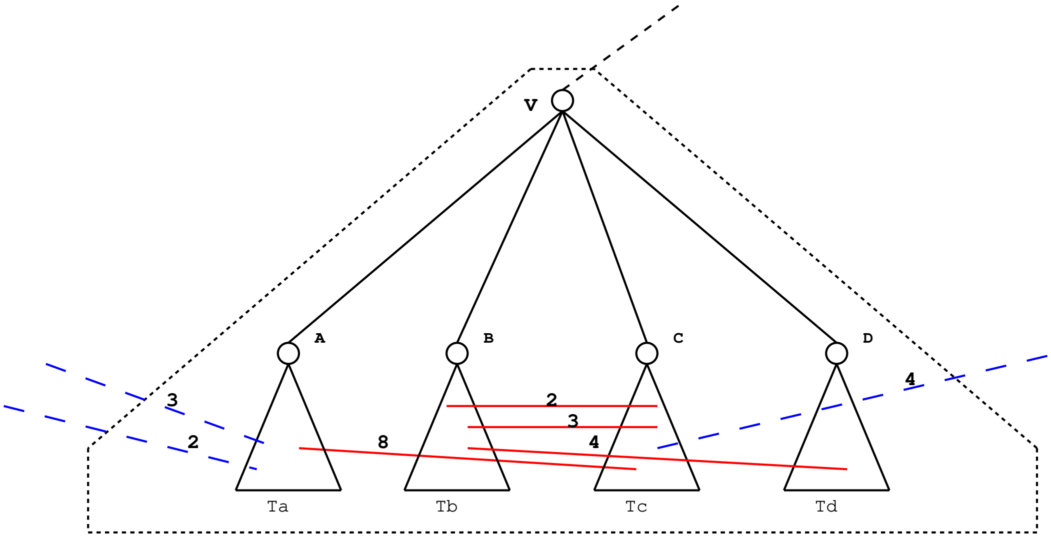


FIG. 4.8 – Sous arbre de l'arbre couvrant issu de v dont les fils sont susceptibles d'être réarrangés.

On observe dans la figure 4.8 que des arêtes du co-arbre créent des liens entre les sous-arbres T_A, T_B, T_C et T_D. Ces arêtes créent également des liens vers l'extérieur. Ces arêtes ne peuvent pas être ignorées : dans la figure 4.8 par exemple, échanger les positions de A et D réduirait la longueur de l'arête liant T_A et T_C de facteur d'étirement 8 mais étirerait également les deux arêtes issues de T_A partant vers l'extérieur de manière considérable. La solution adoptée est de créer deux "faux" sommets de part et d'autre des racines des sous-arbres traités et de rajouter une arête virtuelle dont le poids est égal à la somme du facteur d'étirement des arêtes partant du sous-arbre traité vers l'extérieur, à gauche ou à droite.

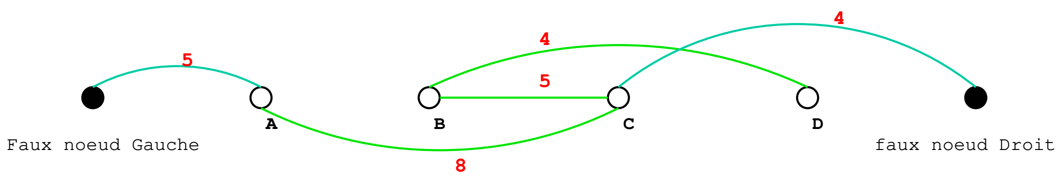


FIG. 4.9 – Graphe à réarranger.

Dans la figure 4.9 qui applique cette méthode au graphe de la figure 4.8, des arêtes virtuelles ont été créées entre les fils de v et ont pour poids la somme du facteur d'étirement des arêtes du co-arbre liant leur sous-arbre. Les arêtes virtuelles partant vers l'extérieur ont également été crées vers les "faux" sommets.

Une fois les arêtes virtuelles calculées, on cherche à ordonner chaque ensemble de frères de manière à minimiser la somme des longueurs des arêtes virtuelles proportionnellement leur poids.

On peut reformuler ce problème de la manière suivante, soit un graphe G ayant n sommets, on veut positionner les n sommets sur une ligne de manière à minimiser la somme des longueurs des arêtes multipliées par leur poids. Ce problème est plus connu sous le nom du problème de *l'arrangement linéaire minimum* [12]. La technique étudiée se base sur un article récent de Y. Koren et D. Harel qui propose une heuristique de minimisation du coût de l'arrangement linéaire ayant une complexité en temps linéaire. Afin d'atteindre de telles performances ils emploient le paradigme de multi-échelle (voir chapitre 3). Cette partie n'a malheureusement pas été implémentée cependant, une recherche nous a permis de trouver les sources C++ d'une implémentation réalisée par Yehuda Koren[2].

Calcul des arêtes virtuelles et des priorités Avant toute réorganisation des sommets, nous devons créer les arêtes virtuelles et calculer le poids leur

[2] http ://www.wisdom.weizmann.ac.il/~yehuda/index_programs.html

étant associé.

Nous présentons d'abord le squelette de l'algorithme, nous discutons des détails ensuite.

Algorithme 6 Algorithme de calcul et valuation des arêtes virtuelles

Entrée: Un graphe G, un arbre couvrant T, une table SF indexée par des arêtes renvoyant leur facteur d'étirement.

Sortie: Une valuation *poids* sur les arêtes virtuelles

Pour chaque arête xy du co-arbre **faire**
  étir est le facteur d'étirement de l'arête xy
  $(a_1, a_2)_{virtual} = filsNCA(x, y)$
  $poids((a_1, a_2)_{virtual}) += f(\acute{e}tir)$
Fin Pour
retourner $TablePriorite$;

Soient x et y deux sommets liés par une arête du co-arbre.

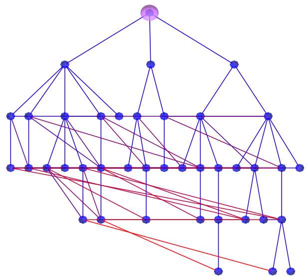
- La fonction *filsNCA(noeud x, noeud y)* a pour but de retrouver le couple de sommets (a_1, a_2) tel que a_1 (resp. a_2) soit un ancêtre de x (resp. y) et $a_1, a_2 \in fils(NCA(x, y))$ Cette opération nécessite la constitution d'une table indexée par les arêtes du co-arbre et renvoyant le couple de noeuds adéquat. Cette table peut être constituée en temps $n*m$ en prenant chaque arête et en remontant jusqu'à trouver le noeud ayant pour père l'ancêtre commun le plus proche. La linéarité se brise ici cependant, on doit pouvoir construire une variante de l'algorithme d'Alstrup et al. [2] de manière à constituer une table en temps linéaire et retrouver a_1, a_2 en temps constant.
- La fonction *f(entier sf)* est là pour permettre d'appliquer une autre méthode que la simple addition des facteurs d'étirement des arêtes liant les sous arbres de x et de y. On pourrait par exemple être tenté de donner un priorité supérieure à plusieurs arêtes de facteur d'étirement faible plutôt qu'à une seule arête de facteur d'étirement élevé (ou inversement).

4.3 Problèmes ouverts et perspectives

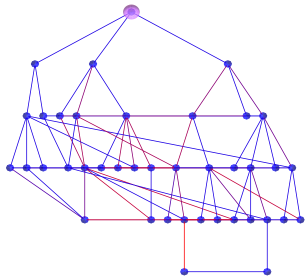
4.3.1 Le facteur d'étirement

Supposons que l'on ait calculé le facteur d'étirement des arêtes du graphe par rapport à l'arbre couvrant. Doit-on obligatoirement recalculer tous les facteurs d'étirements lorsque l'on choisi de "déplacer" la racine dans l'arbre à l'un de ses fils ? Il serait assez intéressant d'un point de vue cognitif que, lors de l'utilisation de notre méthode à des fins de visualisation, on n'ait pas à modifier tout le dessin si on se centre sur un autre sommet.

Vis à vis de ce problème, la réalisation du plugin de couleur pour le facteur d'étirement permet de réaliser l'affichage du nouvel arbre en conservant les couleurs obtenues lors de l'affichage précédent, ce qui donne une idée des arêtes gardant un facteur d'étirement similaire[3].



(a) Arbre enraciné arbitrairement



(b) Déplacement de la racine à un voisin

FIG. 4.10 – Effet d'un changement d'arbre sur le facteur d'étirement

Nous observons que dans la figure 4.10(b) le nouvel arbre affiché a conservé une partie non négligeable des arêtes de l'ancien.

4.3.2 L'affichage

Affichage radial Si l'on utilisait un algorithme de dessin d'arbre radial cela pourrait encore clarifier l'affichage, on utiliserait donc un algorithme d'arrangement sur les noeuds circulaire et non plus linéaire. Etant donné qu'il laisse un degré de liberté supplémentaire, l'arrangement circulaire permettrait d'arranger les noeuds de manière plus efficace.

[3]le lecteur ayant une version noir et blanc de ce mémoire peut ignorer les deux figures suivantes (fig. 4.10(a) et 4.10(b))

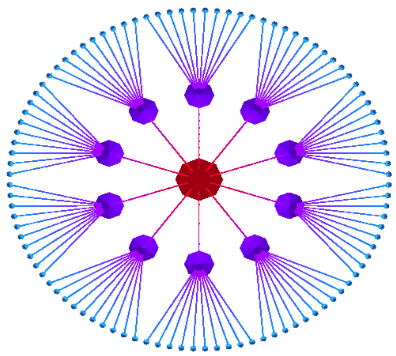


FIG. 4.11 – Exemple d'arbre affiché à l'aide d'une méthode d'affichage radiale.

L'arbre couvrant Vis à vis de l'arbre couvrant, choisir un arbre t-couvrant avec un t minimum serait l'idéal cependant, comme le précisent Cai et Corneil, ce problème est NP-difficile.

On pourrait aussi utiliser des graphes couvrants ayant d'autres propriétés. Le cas des graphes partiels t-couvrants plannaires pourrait être étudié. On pourrait les créer à partir de l'arbre trouvé en rajoutant les arêtes du co-arbre les plus pertinentes de manière à conserver la plannarité ou bien d'en rester aussi proche que possible.

4.3.3 Les PQ-arbres

La méthode actuelle pour réarranger l'arbre consiste à prendre les fils d'un sommet entre lesquels on a rajouté des arêtes virtuelles et à les réorganiser. On pourrait également penser à une autre méthode pour réorganiser l'arbre en traitant les arêtes du co-arbre dans l'ordre décroissant du facteur d'étirement et effectuer les permutations adéquates dans l'arbre afin de rapprocher les sommets reliés par cette arête en respectant les permutations faites précédemment. Pour réaliser une telle procédure, il existe une structure de donnée créée par Booth et Lueker appelée les PQ-arbres [5].

Soit U un ensemble, S inclu dans le sous-ensemble des parties de U, $S = \{S_1, \cdots, S_k\}$. On cherche une permutation π de U telle que pour tout i, les éléments de S_i doivent être consécutifs dans π.
Exemple : $U = \{1, 2, 3\}$ $S = \{\{1, 2\}, \{1, 3\}\}$. $2, 1, 3$ est la seule permutation

acceptable. Le problème que résout les PQ-arbres peut se formuler de cette manière : "Soient les permutations autorisées obtenues jusqu'à maintenant, ne permettre seulement que les permutations pour lesquelles les objets de S ($\mathbb{S} \subset U$) interviennent consécutivement".

Dans notre cas, U serait l'ensemble des sommets de notre arbre couvrant et S serait constitué de paires de sommets issues des extrémités des arêtes du co-arbre traitées. Ainsi, nous pourrions rajouter des arêtes à l'arbre tant que le PQ-arbre n'est pas totalement "vérouillé".

Voici également un avantage qui viendrait d'une telle approche : lorsque l'on rajoute une arête xy du co-arbre à l'affichage, on perd la structure d'arbre en créant un cycle. Via cette arête, on réduit la distance entre x et y à 1, mais on réduit également la distance entre certaines paires de sommets en créant une nouvelle chaîne potentiellement plus courte. On pourrait donc éviter de rajouter par la suite certaines arêtes dont le facteur d'étirement a diminué.

L'algorithme des PQ-arbres est assez complexe à implémenter et par manque de temps, nous n'avons donc pas exploré plus en profondeur les possibilités de cette structure.

5 Plugins Tulip

Ce mémoire s'inscrivant dans le domaine de la visualisation de l'information, une implémentation sur machine s'imposait. Notons aussi que notre méthode nécessite d'utiliser et de modifier des algorithmes parmi lesquels certains sont devenus des standards dans les logiciels de visualisation dignes de ce nom. Il nous fallait donc un logiciel ayant certains traits,

- création et visualisation de gros graphes,
- réalisation de calculs (métriques) sur le graphe affiché,
- architecture claire et permettant l'implémentation modulaire de plusieurs algorithmes,
- une suite d'algorithmes doit être déjà implémentée,
- les algorithmes implémentés doivent être utilisables au sein d'autres parties,
- également, le logiciel doit être libre et tourner sous la plateforme Linux.

Vis à vis de toutes ces contraintes plusieurs solutions étaient possibles, l'une d'entre elles était d'utiliser un logiciel implémenté sous JAVA : JGraph. Mais d'une part, ce logiciel était constitué d'une architecture relativement obscure peu orientée vers l'algorithmique de graphes. De plus, l'implémentation sous JAVA laissait présager des performances moindres.

Après l'essai infructueux de JGraph, la recherche du logiciel adéquat nous a amené vers un logiciel récent dont la philosophie cadre avec l'esprit de notre méthode : Tulip. Ce logiciel, réalisé par David Auber en C++ au cours de sa thèse [4], permet de faire toutes les opérations nécessaires à notre étude que ce soit en matière de visualisation qu'en matière de calcul.

D'un point de vue plus technique, Tulip prend en charge l'ajout de nouvelles fonctionnalités, appelées *plugins*, de manière dynamique. Egalement, il permet d'associer à chaque élément du graphe une ou plusieurs valeurs appelées *propriétés* et de les manipuler efficacement.

Notons que l'architecture de Tulip n'est pas triviale, c'est le prix à payer pour une efficacité optimale. Aussi nous mettons en garde le lecteur sur la technicité des informations contenues dans ce chapitre nécessitant des notions sur le fonctionnement des plugins de Tulip. Au passage, notons qu'une documentation en ligne est disponible sur le site de Tulip et fournit une description exhaustive de l'architecture. Un tutorial en ligne devrait également être disponible sur la conception de plugins sous peu.

Ce chapitre expose les différents plugins créés en ne détaillant que les parties dont l'implémentation a amenée des difficultés.

Plugin 1 (Extraction de l'arbre couvrant). Une difficulté rencontrée ici vient du fait que les graphes utilisés dans Tulip sont orientés et notre extraction prend les arêtes indépendamment de leur orientation. Or, la plupart des plugins Tulip nécessitent certaines propriétés sur les graphes traités, ce qui rend l'arbre que nous extrayons invalide. La solution retenue est de modifier le graphe fournit en inversant les arêtes entrantes rencontrées de manière à faire en sorte que l'arbre que nous extrayons soit reconnu comme étant un arbre selon Tulip. Nous gardons néanmoins une trace des arêtes inversées dans une propriété pour pouvoir retrouver la structure du graphe original.

Plugin 2 (Calcul du facteur d'étirement). Ce plugin exécute séquentiellement plusieurs autres plugins. Conformément à la méthode exposé dans les chapitres précédents, il calcule un arbre couvrant de plus courte chaîne. Cependant, une rapide modification permet tout à fait d'utiliser un autre plugin d'extraction ; la contrainte majeure étant de respecter la notion d'arbre de Tulip.

Afin de calculer le facteur d'étirement de chaque arête, il faut pouvoir récupérer l'ancêtre commun le plus proche d'un couple de noeud en temps constant. Or, une fois les étiquettes calculées, on ne peut retrouver que l'étiquette du NCA en temps constant. Pour retrouver le noeud lui même, on remplit une $Hashmap$ indexée par les étiquettes et renvoyant le noeud associé. Notons aussi que nous utilisons également un plugin déjà implémenté de Tulip : le plugin de calcul de la profondeur dans un graphe orienté acyclique.

Plugin 3 (Calcul des étiquettes pour le calcul du ancêtre commun le plus proche). Ce plugin implémente de manière assez fidèle l'algorithme d'Alstrup et al. [2]. Aussi, nous ne nous étendrons pas dessus. Cependant, on peut noter que la compacité des données est légèrement moins optimale que celle exposée dans l'algorithme. En effet, les délimiteurs d' *étiquette légère / étiquette lourde* sont implémentés par des caractères séparateurs dans l'étiquette finale.

Aussi, précisons qu'une amélioration vient récemment d'être apportée à l'algorithme et qu'elle doit pouvoir être aisément implémentable.

Plugin 4 (Affichage avec ordonnancement des noeuds). Ce plugin n'a pas pu être totalement implémenté faute de temps. La partie consistant en l'implémentation de l'algorithme d'ordonnancement des noeuds n'a pas été abordé. Néanmoins, plusieurs méthodes peuvent être essayées. La première consiste à ne pas implémenter totalement l'algorithme de Koren et Harel [12] et de ne réaliser qu'un des algorithmes de calcul de l'arrangement linéaire minimum tel que *le médian*, *le séquencement spectral* ou bien *le recuit simulé*. Notons que d'autres techniques sont disponibles et ont déjà été testées par Petit [15]. Egalement, on peut tenter d'adapter l'implémentation qu'à faite Yehuda Koren[1] de l'algorithme *multi-échelle* à Tulip.

Vis à vis de la partie implémentée, l'affichage consiste en la récupération de l'arbre couvrant et le lancement du plugin de l'algorithme de Reingold et Tilford (plugin 5). Les positions ainsi calculées sont récupérées et recopiées dans la propriété résultat de notre plugin.

Plugin 5 (Dessin de l'arbre couvrant (Reingold et Tilford)). Ce plugin est disponible dans Tulip et implémente l'algorithme décrit par Reingold et Tilford dans leur article [16]. Nous ne nous étendrons pas sur les spécificité de cet algorithme mais plutôt sur les perspectives qu'il ouvre. Deux implémentations de cet algorithme sont disponibles, il existe une version améliorée permettant aux fils d'un noeud de recevoir un ordre dans lequel chaque fils d'un même sommet doit être affiché. Il suffit de renseigner une propriété nommée "nodeOrder" avec des valeurs entières. Une fois l'algorithme d'ordonnancement des noeuds implémenté, il est aisé d'appliquer les résultats à ce plugin.

Plugin 6 (Calcul des couleurs). Le plugin de couleur fournit une couleur à chaque arête du graphe variant du rouge au bleu proportionnellement au facteur d'étirement.

Plugin 7 (Clustering à l'aide du stretch factor). Ce plugin a principalement été réalisé pour permettre des captures d'images de graphes traités par notre méthode. Ce plugin est pour l'instant très sommaire, une amélioration directe serait d'ouvrir une boite de dialogue demandant les limites du facteur d'étirement au delà desquelles les arêtes ne seraient pas rajouté au graphe final.

[1]http ://www.wisdom.weizmann.ac.il/~yehuda/index_programs.html

Bilan

Au total c'est un sujet très riche que j'ai abordé durant cette étude. De part le contexte dans lequel il s'inscrit, ce mémoire m'a permis d'aborder de nombreux articles appartenant à des pans très différents de la recherche tels que l'article sur les arbres t-couvrants, celui sur le problème de l'ancêtre commun le plus proche ou encore le récent article sur le problème de l'arrangement linéaire minimum via l'approche dite de multi-échelle. Egalement, la finalité de cette étude étant d'être appliquée à la visualisation d'informations, une partie importante a été consacrée à l'implémentation des procédés étudiés. Et, bien que par faute de temps toutes ces méthodes n'aient pas pu être implémentées, cela m'a permis de faire un lien fort avec ma formation initiale de DESS Génie Logiciel ce qui a certainement été pour moi une source importante de motivation.

De plus, je suis persuadé que cette étude n'est pas vaine de suite. Il reste beaucoup de méthodes à implémenter et beaucoup de méthodes à explorer (section 4.3). Enfin, d'un point de vue plus général, il est clair que les problèmes apportés par la structure des gros graphes entraînent potentiellement une grande partie des recherches existantes à être révisées ou améliorées.

Table des figures

Bibliographie

[1] Bondy A. Trigraphs. 1989.

[2] S. Alstrup, C. Gavoille, H. Kaplan, and T. Rauhe. Nearest common ancestors : A survey and a new distributed algorithm. 2002.

[3] David Auber. Tulip web., 2001.

[4] David Auber. *Outils de visualisation de larges structures de données.* PhD thesis, Université de Bordeaux 1, Bordeaux, France, 2002.

[5] K.S. Booth and G.S. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using pq- tree algorithms.

[6] Cai and Corneil. Tree spanners. 1995.

[7] Harel D. A linear time algorithm for the lowest common ancestors problem. In *21th annual IEEE symposium on foundations of computer science(FOCS).*

[8] Harel D. and Tarjan R. fast algorithms for finding nearest common ancestors. In *SIAM journal computing*, 1984.

[9] Josep Diaz, Jordi Petit, and Maria Serna. A survey on graph layout problems.

[10] Herman, G. Melançon, and M. S. Marshall. Graph visualization and navigation in information visualization : A survey. volume 6, pages 24–43, /2000.

[11] Hopcroft and Tarjan R. Dividing a graph into triconnected components. In *SIAM journal computing 2*, 1973.

[12] Y. Koren and D. Harel. Multi-scale algorithm for the linear arrangement problem, 2002.

[13] Tamara Munzner. Drawing large graphs with H3Viewer and Site Manager. In *Graph Drawing*, pages 384–393, 1998.

[14] Peleg and Ullman. An optimal synchronizer for the hypercube. In *Proc. 6th ACM symposium on principles of distributed computing*, 1987.

[15] J. Petit. Experiments on the minimum linear arrangement problem, 2001.

[16] Reingold and Tilford. Tidier drawings of trees. In *IEEE Transactions on Software Engineering*, 1981.

[17] Strahler. Hypsomic analysis of erosional topography. In *Bulletin geological Society of America 63 1117-1142*, 1952.