MOHAMED KAHLA
KAHM12098104
kahlamoh

25 août 2006

IFT 3051

Rapport sur l'implantation de l'algorithme de Sugiyama dans Ptidej

Supervisé par : Yann-Gaël Gueuneuc.

IFT 3051 – Projet informatique.

1. Résumé.

Il est indispensable qu'un programme fournisse des résultats clairs et lisibles. Plusieurs critères de lisibilités des diagrammes et des graphes ont été présentés par plusieurs sources dont la principale est Kozo Sugiyama [1]. Il s'agit principalement de la hiérarchisation de graphe, de la réduction des croisements, de la disposition des sommets de façon balancée et proche et du dessin des lignes droites représentant les arêtes. **Ptidej,** qui affiche des diagrammes de classes UML calculées par rétro conception, était déficient par rapport à ces critères d'affichage de résultat.

Notre travail a donc consisté à implémenter l'algorithme de Sugiyama dans **Ptidej**, en tenant compte des particularités d'un diagramme de classes par rapport à une représentation plus abstraite en graphe multi niveau ou en hiérarchie propre. Comme la nature du problème est combinatoire, il en découle que le problème de mise en pages de graphes est un problème NP-difficile. Aussi, nous avons choisi d'implémenter les heuristiques proposées par Sugiyama [1]. Quoique les heuristiques ne garantissent pas des solutions optimums, nous avons obtenu de très bons résultats tout en gardant un temps d'exécution raisonnable.

Nous avons également rencontré des difficultés pendant le développement du projet dû au manque de clarté des heuristiques présentées par Sugiyama. Je me suis donc basé sur d'autres références et j'ai fini par élaborer mes propres algorithmes qui suivaient au maximum les algorithmes proposés dans la littérature. Pour la hiérarchisation du graphe, j'ai développé mon propre algorithme puisqu'il n'y était pas présent dans les articles étudiés. J'ai également dû modifier la bibliothèque graphique de Ptidej pour qu'il reste compatible avec les nouvelles heuristiques développées.

Finalement, ce projet était très intéressant pour moi et je propose plusieurs possibilités d'extensions et d'améliorations à adapter dans Ptidej pour améliorer sa performance et sa robustesse et de mieux vérifier sa justesse.

2. Introduction et mise en contexte.

2.1 Qualité des résultats et coût associés.

Par définition, un programme est une application qui transforme un ensemble finit de données vers un ensemble de résultats. Cet ensemble de résultats peut varier d'un simple scalaire jusqu'aux données les plus complexes et imaginables. Ces résultats peuvent, en effet, être représentés dans n'importe quels formats de données (vidéo, son, image, fichier, …). Cependant, il arrive qu'un même résultat puisse être représenté dans un même format mais d'une manière plus ou mois « clair » ou compréhensible. Par exemple, l'encodage d'un fichier wave en un fichier mp3 fournie comme résultat un fichier mp3 dont le format peut varier de 24 kibts jusqu'à 320 kibts. Il est évidant que le premier format est nettement moins souhaitable pour un humain que le second pour sa faible qualité. Cependant, si un utilisateur d'un tel programme souhaite un fichier de meilleur qualité, il devra accepter le fait que l'amélioration de la qualité va de pair avec une augmentation du temps d'exécution par exemple (d'autres compromis peuvent aussi êtres considérés également comme la taille du fichier). C'est ainsi qu'un utilisateur devra toujours trouver le meilleur compromis entre qualité et coût.

Pour certains programmes, la qualité du format des résultats a un impact très important sur son utilisation. En effet, plusieurs critères doivent êtres satisfaits lorsqu'un programme fournit des résultats. Le plus important, de ces critères est la lisibilité de ses résultats. Un programme très performant qui fait les calculs les plus précis ne peut être utilisable si les résultats de ces calculs ne sont pas clairement retournés. D'où l'importance d'avoir une représentation claire et lisible des données calculées dans un temps d'exécution relativement raisonnable.

2.2 Ptidej et affichage des résultats.

Le développement de **Ptidej (P**attern **T**race **I**dentification, **D**etection and **E**nhancement in **J**ava**)** à commencer en 2001.

Ptidej est écrit en **Java** et est le résultat du développement de plusieurs théories, méthodes et outils pour l'évaluation et amélioration de la qualité des programmes orientés objets par l'utilisation des idiomes, patrons de conceptions et patrons architecturaux

[www.ptidej.net]. Grâce à **Ptidej**, il est possible de générer, par rétro conception, des diagrammes des classes à partir du code source de programmes.

Ainsi, comme résultat**, Ptidej** fourni une image en deux dimensions représentant un diagramme de classes rétro-conçu à partir des données d'entrée, principalement du code source (ou du byte code Java). L'utilisateur à la possibilité d'afficher sélectivement les différents types de liens existants entre ces classes et de super-imposer des informations sur les diagrammes.

Ce pendant, une faiblesse souvent identifiée par les différents utilisateurs de **Ptidej** était l'absence d'une interface qui représentait clairement et lisiblement les résultats finaux du programme. En effet, **Ptidej** utilise un algorithme naïf, qui plaçait les classes en les décalant vers le bas et à droite. Ce qui résultait en une très faible visibilité d'un diagramme de classes, voir la Figure 1.

2.3 Solution : mise en page des diagrammes de classes UML.

Théoriquement, un diagramme de classes UML est un graphe dont les sommets sont les classes et les arêtes sont les liens entre les classes. Donc n'importe quel algorithme qui affiche un graphe pourrait être utilisé dans **Ptidej**. La première source de l'amélioration de la lisibilité d'un graphe est la minimisation des croisements des arêtes. En effet, par isomorphisme, un même graphe (sommets et arêtes) a plusieurs configurations possibles qui diffèrent seulement par l'emplacement des sommets. De plus, d'autres critères de visibilités, peuvent êtres considérés, comme la minimisation des distances entre les sommets, la réduction des déviations des arêtes, le balancement des liens sur un sommet (les arêtes qui rentrent = les arêtes qui sortent sur un sommet) et les structures de hiérarchies entre les différents sommets.

C'est ainsi que l'algorithme de Sugiyama [1] (algorithme de mise en pages de graphe hiérarchiques) a attiré notre attention pour améliorer la mise en pages des diagrammes de classes affichées dans **Ptidej** après la rétro conception de code source, voir la Figure 2.

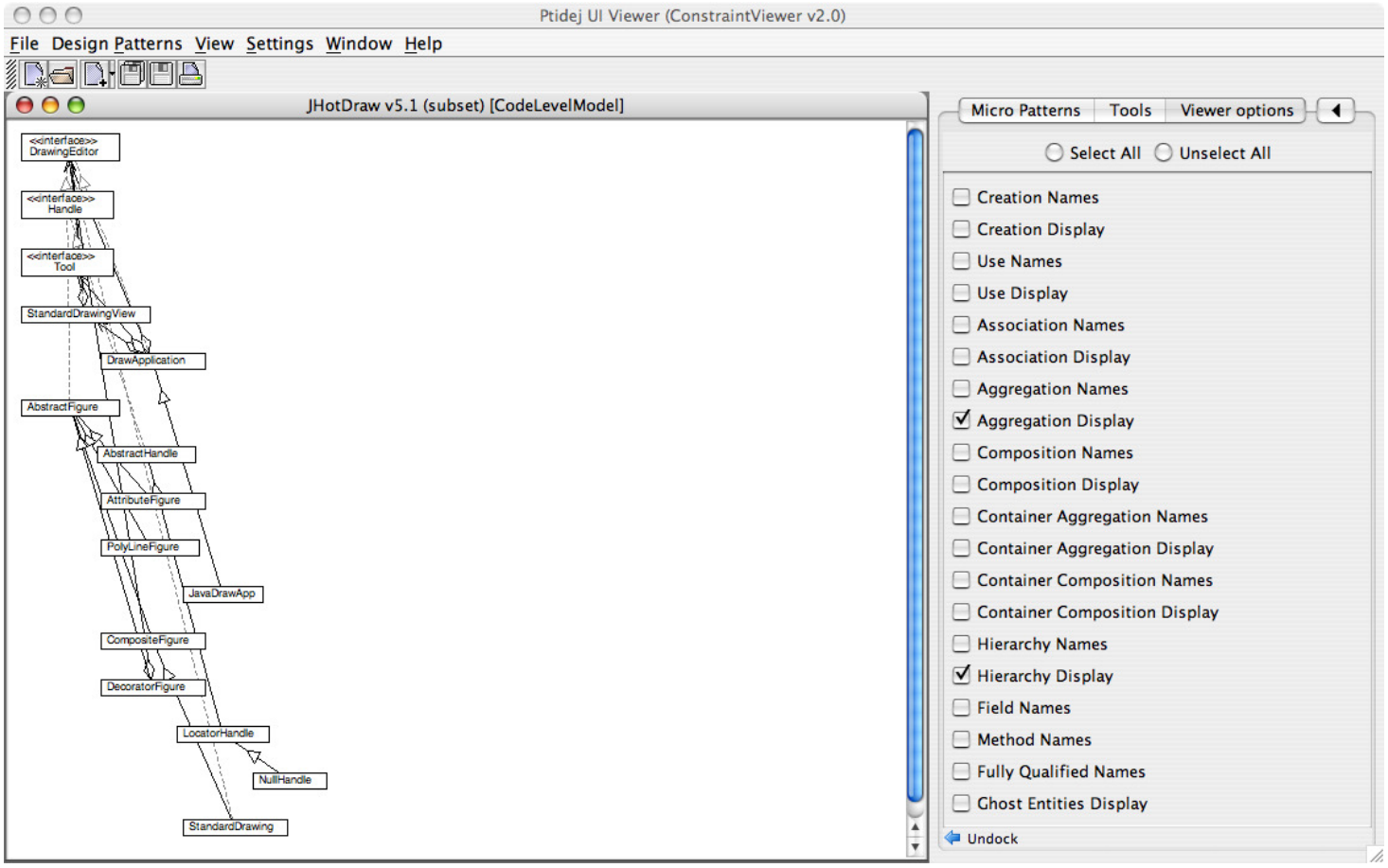

Figure 1 : l'implantation de l'algorithme initial des mise en pages des diagrammes de classes UML dans Ptidej. On remarque la faible clarté de l'image. De plus les flèches ont été modifiés (originellement elles ne peuvent être incliné).

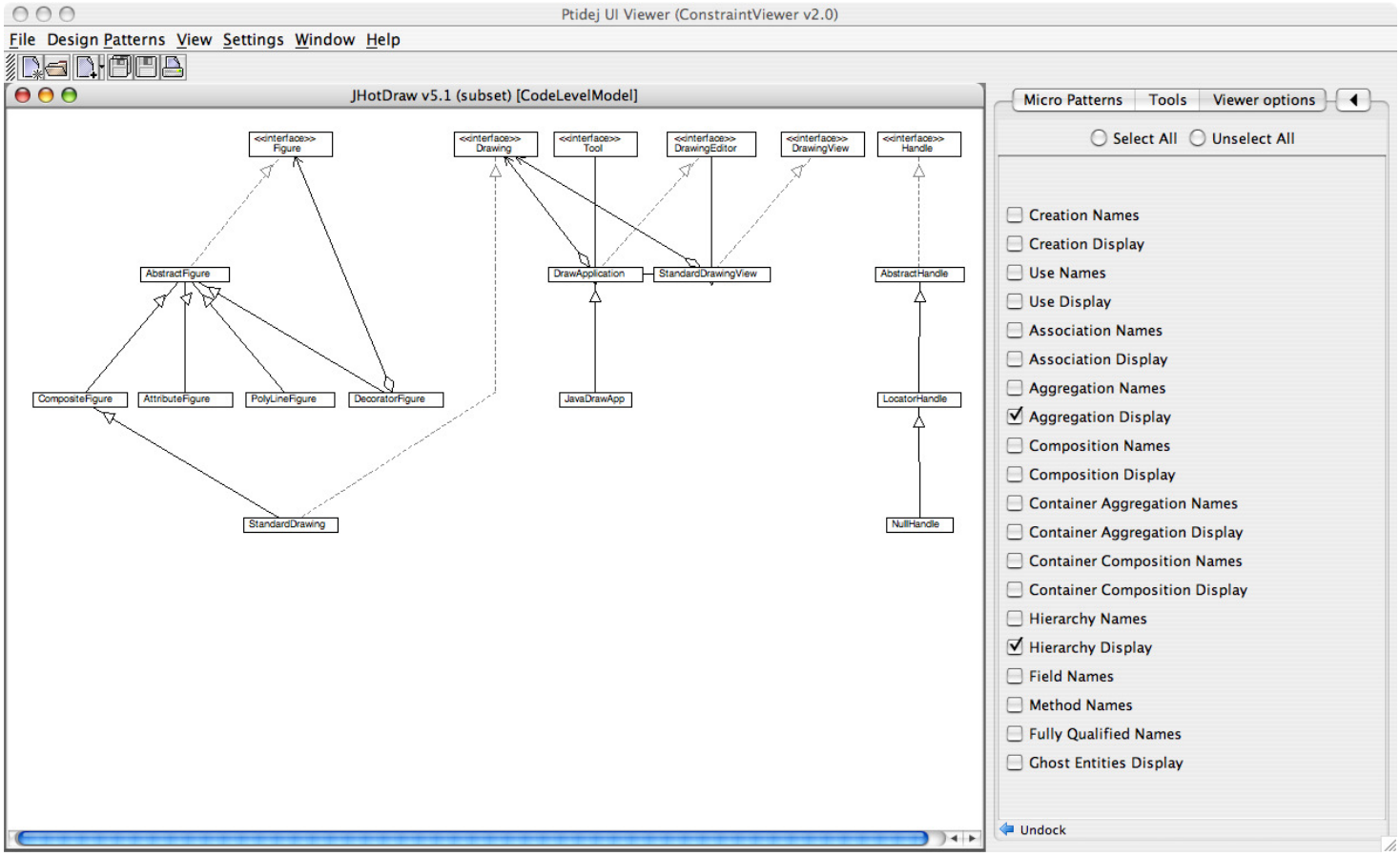

Figure 2 : Résultat de la mise en page du même exemple présenté dans la figure 1 suite à l'implémentation de l'algorithme de Sugiyama. Plus clair et plus lisible!

3. Déroulement et détails du projet : théorie.

3.1 Point de départ : un article !

En 1981, une étude menée pas **Kozo Sugiyama** [1] s'intéressait exactement au problème de mise en pages de graphes **hiérarchiques,** tels les diagrammes de classes qui se transforment en un diagramme hiérarchique en se basant sur les relations d'héritage et d'implémentation. Plusieurs études ont par la suite amélioré les résultats de cet article. Par exemple « l'algorithme neuronal » [2] et l'addition d'autres critères de visibilités et mise en pages pour rassembler les informations d'un même paquetage (package) ensemble par exemple [3].

Dans mon projet, nous nous sommes basés sur l'article de Sugiyama car c'est l'article précurseur.

3.2 Critères de lisibilité et clarté d'un graphe.

D'après l'article de Sugiyama, son algorithme respecte 5 éléments de lisibilité.

1. **Disposition Hiérarchique des sommets :** le graphe est transformé en hiérarchie propre telle que les sommets sont disposés en couches basées sur une relation de généralisation par exemple.
2. **Minimisation des croisements des arêtes :** c'est la plus grande difficulté lors des traçages d'un chemin. En effet, les croisements des liens diminuent la lisibilité du graphe par la difficulté accentué de mise en évidence des relations entre les sommets.
3. **Rectitude (droiture) des lignes :** une ligne droite est facile à dessiner. On divisera plus tard ce concept en arêtes d'une seule envergure et en arêtes de longue envergure.
4. **Disposition rapprochée d'un sommet connecté à un autre sommet :** on souhaite avoir des chemins courts le plus souvent possible.
5. **Disposition balancée des lignes rentrant et sortant d'un sommet :** l'information sur la structure de branchement et de liaison entre les chemins est dessinée clairement et balancée entre arcs entrants et sortants.

3.3 Algorithme de Sigiyama : Heuristiques contre méthodes exacts (théoriques).

3.3.1 Minimisation des croisements. Il s'agit d'un problème NP-difficile. En effet, la nature du problème combinatoire est résolue exactement par l'énumération de toutes les

dispositions (ordre d'un sommet dans un niveau) possibles en gardant celles qui impliquent un nombre de croisements le plus faible. La méthode exacte est appelée **méthode de minimisation des pénalités**. On l'applique sur deux niveaux successifs de la hiérarchie du graphe et l'on peut la généraliser sur les n niveaux du graphe. Cependant, la méthode théorique, même si elle garantie la meilleure solution, reste très coûteuse en termes de temps d'exécution. La méthode heuristique quant à elle s'appelle **méthode barycentrique** et sera présentée plus loin dans ce rapport.

3.3.2 **Disposition rapprochée et balancement des sommets.** La méthode exacte s'appelle **méthode de disposition par programmation quadratique.** Elle consiste à minimiser la fonction objectif $f = cf_1 + (1 - c)f_2$ où f1 et f2 sont des fonctions quadratiques. La fonction f1 représente la disposition rapprochée des sommets, alors que la fonction f2 représente le balancement des sommets. Ainsi, la minimisation de la première fonction assure une disposition rapprochée alors que la minimisation de f_2 assure le balancement des sommets. Le facteur c est un réel entre 0 et 1, selon lequel l'une des deux fonctions pourra être priorisée sur l'autre. Un c égale à 0.5 assure un compromis égale entre le rapprochement et le balancement. Cette minimisation est sous mise aux contraintes suivantes :

- un point initial fixe ;
- une limite inférieure fixe de distance entre les sommets.
- Les sommets qui n'ont qu'une seule connexion entre eux et qui sont sur deux niveaux successif doivent avoir le même coordonné x.

Cependant, la méthode théorique, même si elle garantie la meilleure solution, reste très coûteuse en termes de temps d'exécution. La méthode heuristique compte à elle s'appelle **méthode de disposition par priorité** et sera présentée plus loin dans ce rapport.

D'après des tests de performances, Sugiyama montre que la méthode heuristique qu'il présente donne des résultats significativement effectifs en fournissant des solutions très proches de la solution exacte, d'une part, et en s'exécutant avec des performances nettement supérieures à la méthode exacte, d'autre part.

Les deux approches théoriques (ci-haut) et heuristiques peuvent êtres implantés. Les méthodes théoriques se basent sur la nature même du problème. Les méthodes heuristiques

permettent d'élargir le nombre de sommet dans le graphe ainsi que le nombre d'arêtes entre les sommets. Par conséquent, les méthodes heuristiques, même si elles ne garantissent pas que la solution obtenue soit la meilleure, nous permettent toujours d'avoir une solution très proche de la meilleure solution dans la majorité des cas tout en gardant un temps d'exécution raisonnable. Nous avons décidé d'implémenter les méthodes heuristiques et de les intégrer dans **Ptidej** pour l'affiche et la mise en page des diagrammes de classes.

4. Déroulement du projet et détails pratique.

Une grande partie du premier mois de mon projet a été consacrée à la compréhension, l'analyse et conception des algorithmes avancés dans l'article de Sugiyama[1] . En effet, seuls les algorithmes des étapes 2 et 3 étaient discutés par l'auteur. Les difficultés rencontrées seront présentées plus loin dans ce rapport dans la section **Difficultés et choix alternatifs.**

4.1 Ptidej, familiarisation avec le code source.

Ptidej est un grand logiciel comportant plusieurs projets. Les projets qui m'intéressaient étaient **Ptidej UI, Ptidej UI Primitive AWT** et **Ptidej UI Viewer Standalone SWT** et **AWT.** Les deux derniers projets contiennent les classes de démarrage permettant d'exécuter le programme avec l'interface **Swing** ou **AWT**. **Ptidej UI Primitive AWT** contient tous les outils de dessin des instances et des éléments contenus dans les diagrammes de classes, par exemple les classes Ligne, ArraowSymbol, Rectangle, Triangle... Finalement, le projet **Ptidej UI** contient la majorité des classes développées dans ce projet. Plus particulièrement, toutes les classes se trouvant dans le paquetage **sugiyama** sont nécessaire pour la mise en pages des diagrammes des classes par Ptidej.

4.2 Architecture des classes : rôle et associations.

4.2.1 La classe SugiyamaLayout.

C'est la classe principale qui implémente l'interface IGraphLayout et la méthode doLayout. Cette méthode reçoit toutes les entités et éléments du graphe en paramètre, donne les positions verticales et horizontales pour chaque entité, les construit, puis construit les autres éléments et les retourne pour êtres dessinés. Dans cette classe, on gère toutes les étapes de l'algorithme puisque les résultats d'une étape constituent les données d'entrée de la prochaine étape, architecture « pipeline ».

4.2.2 Les classe Node et DummyNode.

Comme les entités et les éléments ne contiennent pas toute l'information nécessaire comme, par exemple, les parents et les enfants, j'ai ajouté la classe Node qui encapsule une entité en connaissant son niveau, ses enfants, ses parents ainsi que sa position. La classe **DummyNode** hérite de la classe Node et contient toutes les données d'un **Node** avec la seule différence qu'elle n'encapsule pas d'entité car elle sert à représenter les nœuds artificiels requis par l'algorithme de Sugiyama.

Toutes les étapes travailleront donc avec des instances de la classe Node comme sommets du graphe.

4.2.3 La classe Graph.

C'est la représentation d'un graphe. La classe **Graph** contient des instances de la classe **Level**, qui contiennent des instances de la classe **Node**. Ainsi, l'objet Graph représente un graphe orienté et acyclique multi-niveaux. L'avantage de garder tous les nœuds du même niveau ensemble est de trouver facilement tous les nœuds d'un même niveau et qui partagent un même parent. Ainsi, on peut facilement rejoindre ces nœuds sur la même arête dans le cas d'un lien de généralisation (implémentation ou héritage).

4.2.4 Les classes de construction.

Plusieurs classes ont été ajoutées pour construire le graphe. La première est la classe **HierarchieBuilder,** qui permet de mettre chaque entité dans un niveau en se basant sur les relations de Généralisation. Les classes **NodeBuilder** et **GraphBuilder** permettent de construire respectivement les nœuds du graphe et le graphe lui-même. La classe NodeBuilder s'occupe de supprimer les cycles du graphe en détectant si un enfant se situe sur un niveau plus haut que son parent et d'inverser le sens du lien. Ainsi, l'enfant devient parent et vice vers ça. Il est à remarquer que ce problème touche exclusivement les liens autres qu'implémentation et héritage. La classe **GraphBuilder** construit tous les niveaux du graphe. Elle s'occupe également de garder les relations entre les nœuds cohérents.

La classe **EdgeBuilder,** quant à elle, sert à construire les arêtes du graphe avec leurs différents types et les **DummyNodes** utilisées dans la phase de dessin du graphe et de reconstitution des arêtes de grande envergure. De plus, la classe **DummyNodesBuilder** permet

d'ajouter les nœuds artificiels sur les arêtes de grande envergure. Ces dernières sont modifiées pour considérer les enfants d'un même niveau et ayant le même parent, dans le cas des liens de généralisation, pour ne traiter qu'une seule arête pour chaque parent. En effet, toutes les arêtes émergeant de tels enfants vont se rejoindre pour se diriger vers le même parent. Un seul nœud artificiel est jouté sur chaque niveau séparant les enfants du parent. Les autres types de liens sont considérés un lien à la fois entre un enfant et un parent.

4.2.5 Les classes d'actions.

Selon les différentes étapes de l'algorithme, j'ai ajouté différentes classes dont chacune réalise une des étapes.

La classe **BarycenterCrossingMinimiser** implémente l'heuristique de minimisation des croisements par la méthode des barycentres (Étape 2). Cette classe utilise une classe **Matrix** particulière comme structure de données d'entrée. Chaque objet Matrix, représente un ordre particulier des sommets de deux niveaux successifs. Plusieurs méthodes ont été ajoutées à cette classe pour réaliser les différentes actions de l'algorithme sur deux niveaux du graphe. On a par exemple les méthodes de calcul du nombre de croisement, le calcul des valeurs des barycentres, l'ordonnancement des barycentres (selon les colonnes et les rangés), l'ordonnancement **inverse** des barycentres (selon les colonnes et les rangés) nécessaire pour éventuellement sortir l'heuristique d'une solution intermédiaire en essayant d'avoir une meilleure solution…

La classe **PriorityHorizentalSolver** implémente l'heuristique de disposition horizontale des sommets sur chaque niveau (Étape 3). Cette classe comporte les méthodes de calculs des priorités (haut et bas ou up and down) et la méthode **pushMove**. Cette méthode récursive, qui se base sur les priorités de chaque sommet, déplace les sommets pour que le sommet le plus prioritaire ait la plus grande chance d'être le plus près de sa cible en déplaçant ces voisins de plus faibles priorités.

La classe **XCoordonateSetter** permet à partir des résultats fournis par le PriorityHorizentalSolver (l'index de chaque sommet sur son niveau gradué en entier) de calculer et d'affecter à chaque entité (si nœud réel) les coordonnés réels (dans le diagramme) de chaque sommet en gardant le graphe compact et clair.

La classe **LineDrawer** contient plusieurs méthodes pour permettre le dessin correct des liens. Elle contient la méthode qui envoie la liste de nœuds artificiels permettant de régénérer

les arêtes de grandes envergures. Elle contient aussi, les méthodes d'espacement entre les sommets verticalement et d'espacement entre les différentes lignes qui se trouvent entre deux niveaux. C'est ainsi qu'on introduit la notion des canaux (Spliter). Chaque ligne de type **PlaineDoubleSquareLine** ou **DottedDoubleSquareLine** passe par un « canal » horizontal qui lui est propre et ainsi on évite les superpositions des lignes. Seuls les liens de type de généralisation sont concernés (Figure 2). La notion de canaux est d'un index sur l'axe vertical unique pour chaque lien, et permettant de dessiner les liens sous forme de « T » ou des chemins entre deux sommet qui ne contiennent que des lignes verticales ou horizontales. Finalement, cette classe construit les entités (par un appel à la méthode build()) contenant toutes les informations déjà calculées pour les préparer à la dernière phase de dessin du graphe.

4.2.6 Les classes auxiliaires.

La classe **LinkObserver** permet d'aider la classe NodeBuilder à trouver tous les liens visibles à partir d'un nœud origine vers toutes ces cibles.

La classe **SettingValue** contient tous les paramètres de l'algorithme et permet donc à l'utilisateur de l'algorithme de modifier toutes les valeurs de l'algorithme. On peut donc modifier le nombre d'itérations vides des heuristiques avant l'arrêt, les distances entre les entités (verticales et horizontales), les déplacements initiaux de gauche…

D'autres classes existent, mais leur rôle est moins important que celles qui sont présentées. Comme les classe **Parents, Children, Couple, LinkType, Edge…**

4.3 Algorithme de Sigiyama : étapes et déroulements de la solution.

L'algorithme de Sugiyama requiert un graphe dirigé et acyclique en entrée. En sortie, on retourne le même graphe avec une nouvelle configuration des sommets les uns par rapport aux autres. Cet algorithme est divisé en quatre étapes principes.

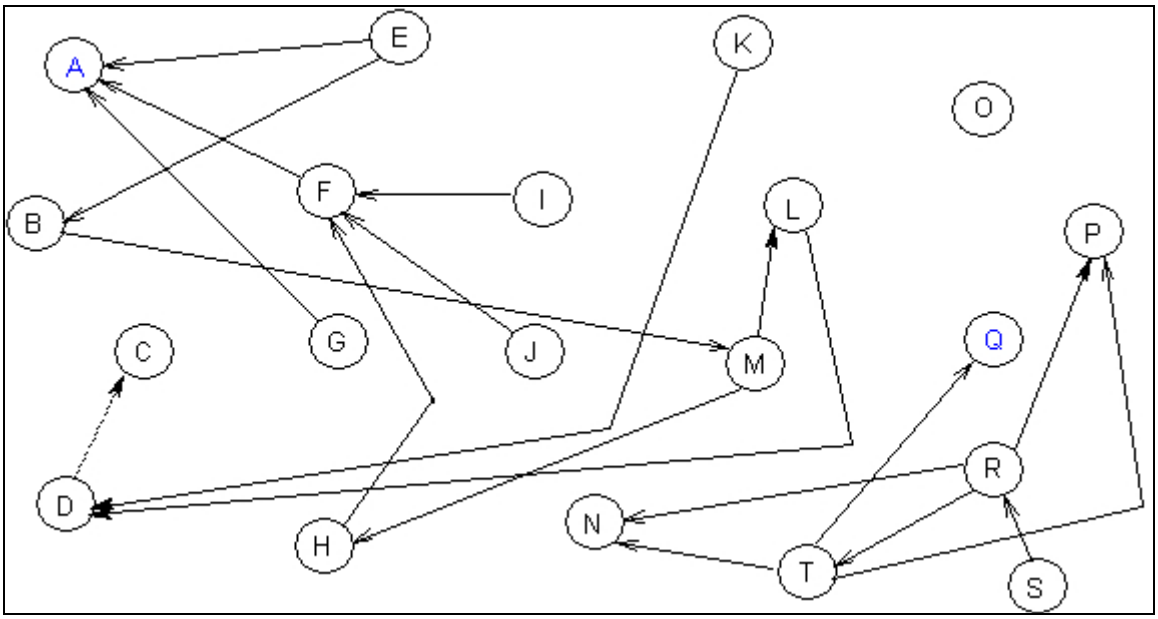

Figure 3 : Un graphe dirigé non connexe.

4.3.1 Étape1. Formation d'une **hiérarchie propre** à partir d'un ensemble de sommets (noeuds) et des relations qui les relient (arêtes). Si le graphe engendré à partir des relations contient des cycles, ils sont éliminés par inversion de sens des arêtes se dirigent d'un niveau plus bas vers un niveau plus haut. On obtient ainsi un graphe dirigé acyclique multi niveaux ou hiérarchisé. Les **arêtes** de **grande envergure** (arêtes qui lient deux sommets non situés sur deux niveaux successifs) sont divisées en arêtes d'une seule envergure pour avoir une hiérarchie propre. Par insertion des **nœuds artificiels** à chaque niveau séparant les sommets liés par une arête de grande envergure. Ainsi le nouveau chemin, entre ces paires de sommets, passe par ces nouveaux sommets artificiels ajoutés au graphe pour avoir une hiérarchie propre.

La première étape est de construire un graphe dirigé acyclique multi niveau. Dans cette perspective, nous nous sommes basés sur les liens d'implémentation et d'héritage. En effet, l'algorithme conçu recherche toutes les entités qui ne possèdent pas de parents et les place dans le niveau 0. Par la suite, pour chaque niveau i ajouté, on cherche tous les enfants des entités

ajoutées à ce niveau. Si un enfant a été ajouté à un niveau plus haut, ce nœud serait ignoré dans ce niveau et il sera ajouté dans le niveau i + 1. Si au moins un enfant du niveau i + 1 possède un enfant, on continue, sinon si tous les enfants du niveau i + 1 n'ont pas d'enfant, on arrête. On obtient ainsi **un graphe hiérarchique**.

Une fois la hiérarchie construite, nous procédons à la construction des nœuds. Pour chaque niveau et pour chaque entité dans un niveau, en commençant par les parents, je crée une instance d'un nœud contenant l'entité, son niveau, ses parents et ses enfants. Finalement, j'ajoute le nouveau nœud à la **liste** de tous les **nœuds**.

À la fin de cette phase, tous les noeuds sont formés. Pour toutes les arêtes de grande envergure, on insère autant de noeuds artificiels que de niveaux séparant les deux sommets sur cette longue arête. Pour les liens d'implémentation et d'héritage, on rassemble tous les enfants d'un même niveau qui ont le même parent sur la même arrête de grande envergure. Cela permettra de tracer une seule ligne sortante du parent et allant vers le niveau où se situent les enfants. Près des enfants, la ligne se divise sur tous les enfants. À la fin de cette étape, on a **une hiérarchie propre**, voir la Figure 4.

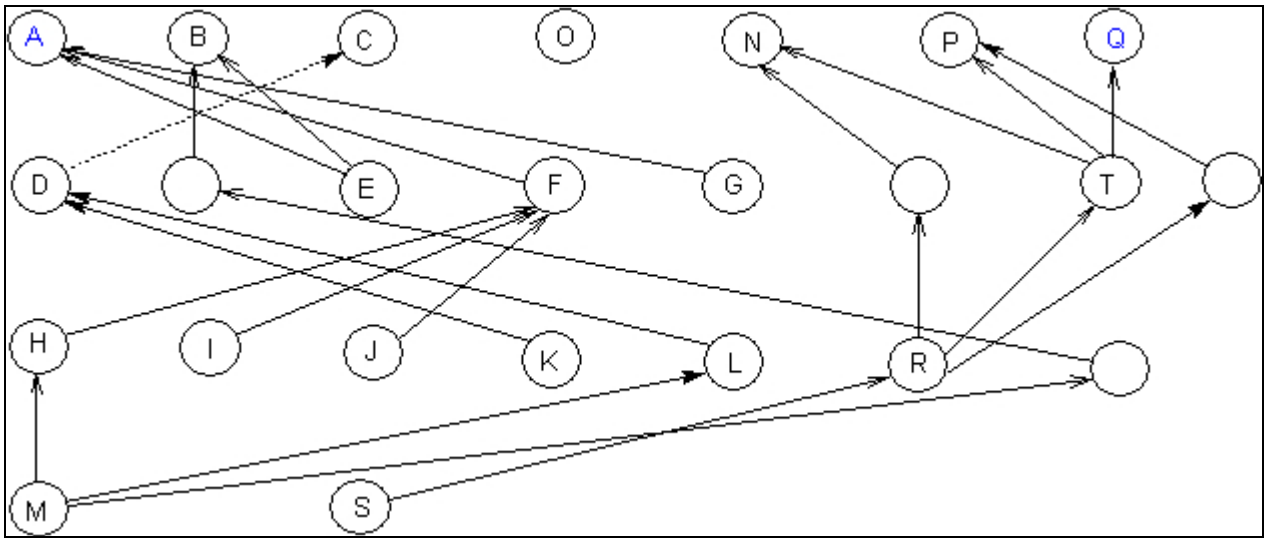

Figure 4 : Un graphe multi niveau (Graphe dirigé acyclique) ou hiérarchie propre. Les cercles sans étiquettes qui correspondent aux nœuds artificiels ajoutés sur les arêtes de grande envergure. Certaines arêtes ont été inversées pour avoir un graphe acyclique.

4.3.2 Étape2. Le nombre de croisement de toutes les arêtes dans la hiérarchie propre (artificielles ou pas) est **minimisé** par la **permutation** de l'ordre des sommets dans chaque niveau.

Nous intéressons à **minimiser** les **croisements** des arêtes en modifiant l'ordre des sommets sur chaque niveau. Cette heuristique utilise une méthode de haut en bas (**Up to Down**) et du bas en haut (**Down to Up**). On fixe le niveau 0. On calcule les barycentres de tous les sommets du niveau 1. On réordonne les sommets du niveau 1 dans un sens ascendant. En poursuit le même résonnement pour les autres niveaux. On fixe le niveau i déjà ordonné. On calcule les barycentres du niveau i+1. Et l'on réordonne les sommets du niveau i+1. Il s'agissait de la phase descendante de l'algorithme. Pour la phase ascendante, on commence par fixer le dernier niveau. On réordonne le niveau i-1, et ainsi de suite, jusqu'au niveau 0.

On recommence les étapes Up to Down suivi de Down to Up jusqu'à ce qu'aucun changement ne soit observable. Ce dernier test consiste à compter un certain nombre de boucles sans progrès. Par la suite, si le nombre de croisement égale 0, on arrête, sinon on ordonne inversement les sommets qui ont les mêmes valeurs de barycentres. En effet, par cet inversement, le nombre de croisement ne change pas, l'ordre initial des sommets au début des procédures Up to Down et Down to Up est modifié. Ainsi, on essaye de sortir d'une solution partielle dégénérée en modifiant la solution de départ de la prochaine étape de l'heuristique. Si on n'a aucun progrès après un certain nombre de boucles, on arrête. Voir la Figure 5.

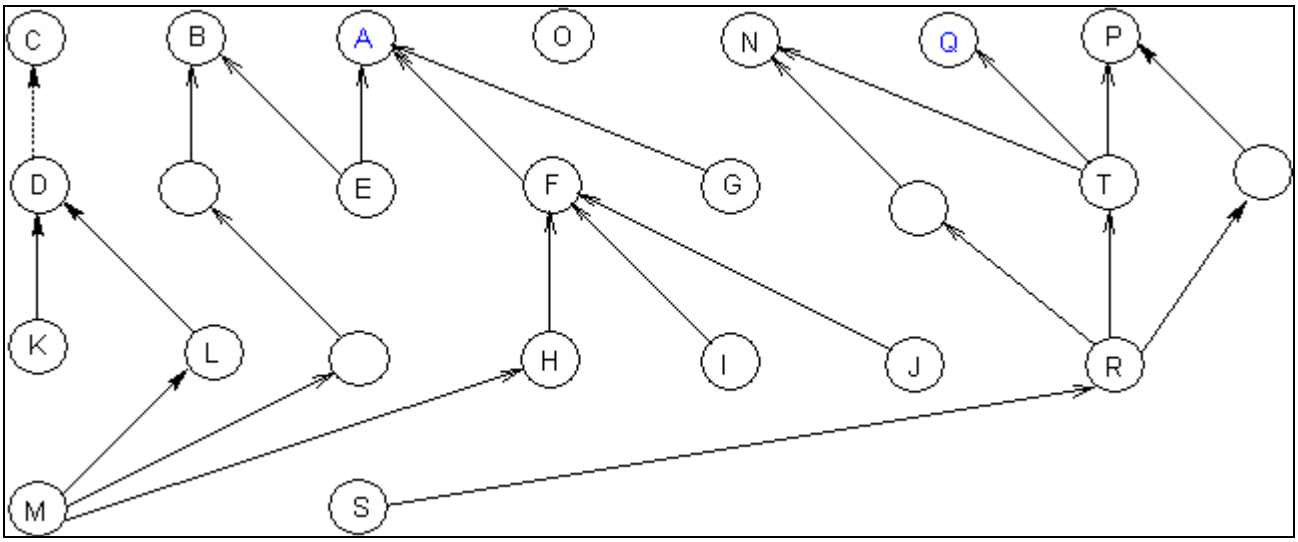

Figure 5 : Minimisations des croisements. Ce graphe obtenu contient 0 croisement. On remarque le réarrangement des sommets sur chaque niveau (par permutation).

4.3.3 Étape 3. La position horizontale des sommets est déterminée en considérant les trois éléments de lisibilité : les arêtes de grande envergure sont dessinées droites (si possible), une disposition rapprochée des sommets connectés les uns aux autres et une disposition (layout) balancée des lignes de et vers un sommet.

Une fois l'ordre de tous les sommets a été établi, on peut commencer l'étape de **déplacement des sommets horizontalement** pour assurer, si c'est possible, un balancement de sommets ainsi qu'une courte distance entre les sommets liés. Pour chaque sommet, on lui associe une priorité. Il s'agit de la connectivité de chaque sommet (nombre de liens). Pour les sommets artificiels, on leurs associe la plus grande priorité soit par exemple la somme de toutes les priorités des autres sommets ou la **priorité** la plus hautes des sommets réels dans un niveau plus 1 (la priorité maximale d'un niveau en question). Ainsi, le sommet le plus prioritaire sera déplacé en premier et ainsi de suite. Lorsqu'on déplace un sommet, on déplace tous les sommets qui ont une plus faible priorité dans le sens de son déplacement. On répète la procédure jusqu'à ce qu'un certain nombre de tour de boucle soit atteint sans modification des positions. Il arrive souvent qu'un sommet, ou plus, soit déplacé entre gauche et droite (+ ou – un index) constamment, et c'est pour ça que nous avons ajouté aussi une limite supérieure sur le nombre de boucles générales. Finalement, on procède à la normalisation des indices. En effet, il pourrait

avoir des indices négatifs ou très éloignés de 0. L'indice le plus petit est mis à 0, et tous les autres indices des autres sommets, sont modifiés en conséquence.

À ce stade de l'algorithme, on calcule les coordonnées réelles à partir des indices relatifs déterminés à l'étape précédente. On procède également à compacter les sommets du graphe en considérant chaque index comme un axe dans lequel, les nœuds ayant le même indice seront dans le même axe. Par conséquent, la largeur de chaque axe sera égale à la largeur de l'entité la plus large de cet axe plus une constante d'espacement choisie par défaut. Voir la Figure 6.

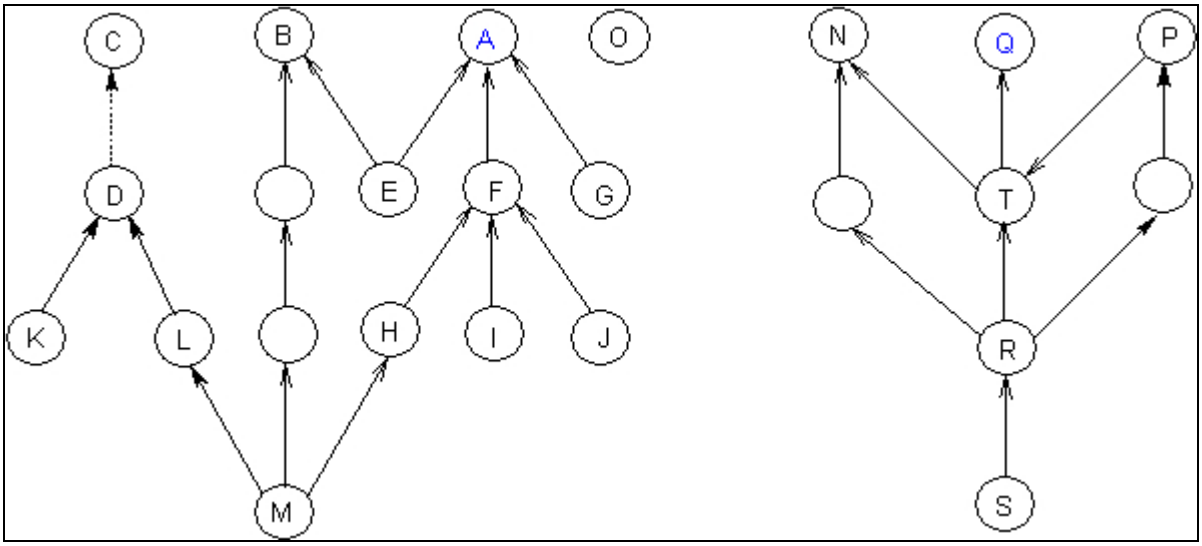

Figure 6 : Dispositions horizontales des sommets. On a conservé l'ordre obtenu pendant la phase des minimisations des croisements. On déplace les sommets sur l'axe horizontal pour obtenir des sommets balancés et une disposition rapprochée des sommets inter liés.

4.3.4 Étape 4. Une image bidimensionnelle est générée à partir de la hiérarchie propre où les nœuds et les arêtes artificiels sont éliminés et les arêtes de longues envergures correspondantes sont régénérées.

Finalement, à partir des données déjà calculées, on procède à la détermination des nœuds artificiels dans chacune des arêtes. On détermine aussi les canaux et les diviseurs (Splitter) pour chaque lien de généralisation. On construit les entités et éléments et on les retourne pour êtres dessinés. Voir la Figure 7.

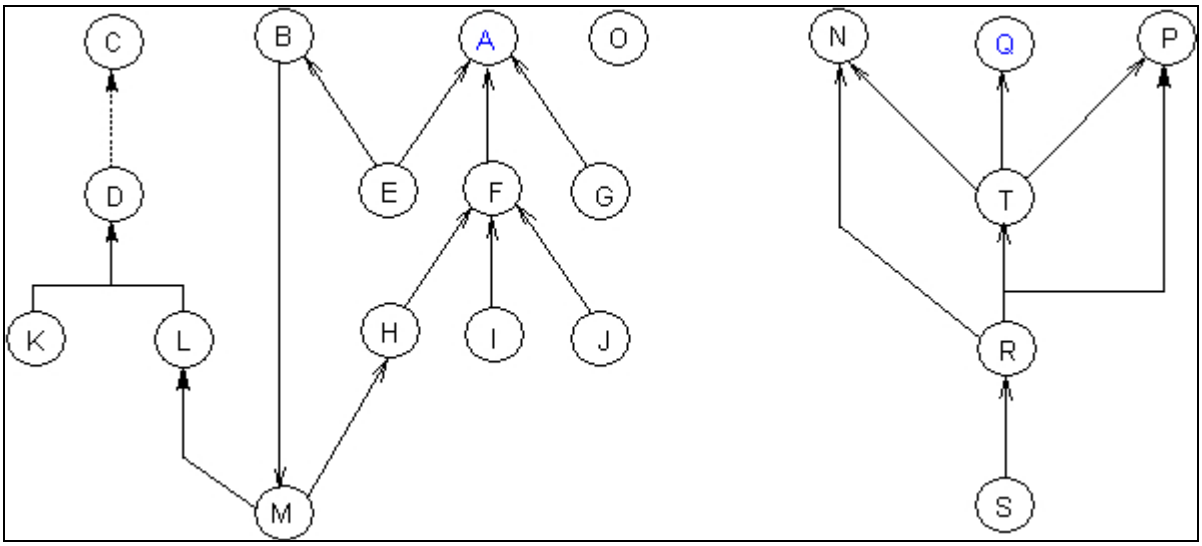

Figure 7 : Dessins du graphe. On supprime les nœuds artificiels pour régénérer les arêtes de grande envergure. Certaines arêtes sont dessinées d'une façon particulière pour exprimer la nature du lien différent qu'elles représentent (D-K, D-L et P-R). On retrouve également le sens réel des arêtes inversées pour le besoin de l'algorithme.

5. Difficultés et choix alternatifs.

5.1 Problèmes techniques divers.

Les postes Windows comporte un logiciel anti-virus qui assure que le programme ne démarrera pas avant au mois 10 minutes.

La plate-forme Eclipse n'était pas installée sur les postes du laboratoire. L'espace disque réseau était limité aussi. Une fois ces deux problèmes résolus, le démarrage du programme **Ptidej** ne prenaient pas moins de 10 minutes au moins à cause de l'utilisation de toutes les ressources système par le logiciel anti-virus et par la vérification systématique de tous les fichiers de l'espace de travail. C'est ce logiciel qui à chaque fois scan le programme et ralenti le système. L'installation d'Eclipse sous Windows sur le disque C rend l'exécution du programme normal.

Pour corriger cette situation, et avant de découvrir la cause de ce ralentissement anormale, je me suis décidé à travailler sur mon ordinateur portable personnel (Mac Book Pro). Comme il n'y avait pas de version d'Eclipse pour Mac OS X intel x86 Platform, j'ai patché la version qui roule sur PowerPC pour pouvoir travailler. Finalement, il fallait paramétrer correctement les chemins (paths) pour les différents projets dans Eclipse.

La bibliothèque graphique présente dans **Ptidej** n'était pas conçue pour afficher des arêtes de grande envergure composées de plusieurs nœuds et arêtes artificiels. J'ai dû modifier les classes des dessins des lignes dans le paquetage **AWT** et **SWT**. J'ai ajouté aussi les classes **PlainDoubleSquareLine** et **DottedDoubleSquareLine** pour afficher les liens de généralisation par des lignes composées de segments soit verticales ou horizontales. J'ai également modifié les classes **Implementation** et **Specialisation** ainsi que **Plaintriangle** et **Dottedtriangle** pour que le triangle de la flèche suive la déviation (inclinaison) de la ligne dans le cas où la cible et l'origine ne se situeraient pas sur le même axe vertical.

5.2 Prendre et comprendre le code d'une autre personne.

La taille des projets, les différents niveaux de généralisation ainsi que le manque des commentaires. Heureusement, que Yann était présent pour m'expliquer toutes les classes que j'ai utilisées puisque toutes les classes que j'ai utilisées étaient codées par lui.

5.3 Recherche des détails non clarifiés dans l'article de Sugiyama.

Une grande partie de mon projet a consisté à la recherche des détails sur les heuristiques présentées très sommairement dans l'article de Sugiyama et la majorité des études ultérieures menées sur le sujet ne fond que reprendre ce que Sugiyama présentait. En effet, dans l'article de Sugiyama, l'auteur présente le pseudo code très abstraitement : il détaille la minimisation des croisements entre deux niveaux par la méthode de barycentres, il parle ensuite de la possibilité de généraliser la procédure sur n niveaux. Ce pendant, il laisse beaucoup de détails non clarifiés. Par exemple, le critère d'arrêt et l'inversion de l'ordre des sommets de la solution partielle pour se donner la chance de sortir d'une éventuelle solution dégénéré plus facilement.

Pour l'heuristique de la disposition horizontale des sommets, aucun pseudo code n'est fourni. Seul un schéma explicatif de la technique ainsi qu'un exemple très simple ont été présentés. De plus, l'auteur affirme que plus de détails se trouvent ailleurs [4]. Mais il s'agit d'un rapport de recherche non disponible. En suivant les procédures habituelles, j'ai demandé de commander l'article. Après 3 semaines, d'attente, j'ai finalement reçu le rapport de recherche tellement attendu. À ma grande surprise et déception, ce rapport contenait quasiment toutes les données présentées dans l'article original avec un nouveau diagramme. Il s'agit d'un diagramme de flow de la méthode de disposition (layout) par priorité présenté aux pages 20 et 21.

Cependant, pendant les trois semaines d'attente, j'ai continué ma recherche jusqu'à ce que je tombe sur une autre référence [5]. Il s'agit de la thèse d'un étudiant en doctorat à l'université McGill qui s'intéressait aux algorithmes de mise en pages des graphes. C'est là que j'ai trouvé un pseudo code plus clair et détaillé de cette heuristique présentée aux pages 33 à 43. J'ai également pu trouver d'autres erreurs dans l'heuristique de minimisation des croisements basée uniquement sur l'article de Sugiyama. Les algorithmes présents dans la référence [5] se basaient sur une structure de données différente de celle déjà développé. Par conséquent, j'ai réécrit les deux heuristiques en essayant de suivre au maximum les algorithmes et principes présentés dans les deux références citée plus haut [1] et [5].

5.4 Difficulté d'implémenter un test pour les heuristiques.

Finalement, la plus grande difficulté rencontrée pendant l'écriture du code source était l'absence de test automatisé pour chaque étape que j'ai implémentée. Ce point sera proposé

dans les études futures possibles pour étendre mon projet. Il est évident donc, que je développais une nouvelle mise en pages des diagrammes UML et il est très difficile de faire un test puisque le critère d'échec ou de réussite d'un tel test n'est pas clairement identifiable.

5.5 Choix raisonnable du nombre de sommets et des arêtes du graphe.

Voir la figure 9 en Annexe.

Il est vrai que la réduction du nombre de croisements améliore nettement la clarté du dessin. Cependant, au-delà d'une certaine limite, il devient très difficile d'améliorer la clarté du graphe. Il serait donc justifié de choisir le nombre de sommet et des arêtes d'une façon raisonnable pour obtenir les meilleurs résultats. Et d'ailleurs, comme futurs travaux, on pourra déterminer le meilleur rapport nombre de sommets et d'arêtes donnant la meilleure clarté possible. On pourra aussi attribuer un pointage (score) sur la clarté du dessin de graphe. Comme repère, zéro croisement correspond au meilleur pointage!

6. Travaux Futurs.

Comme il s'agit d'une heuristique, on pourra ajouter des méthodes pour mieux chercher de nouvelles possibilités pendant la phase de ré-ordonnancement inversée de la phase de minimisation des croisements.

On pourra aussi implémenter des tests de convergences plus performants, pour provoquer la condition d'arrêt de l'algorithme. On cite par exemple la détection de l'apparition périodique d'une même matrice lors de la minimisation des croisements. Il y a aussi le cas de déplacement horizontal des sommets dont on peut détecter lorsqu'un sommet est déplacé constamment de gauche à droite et vice vers ça, ce qui permettra d'améliorer les conditions d'arrêts.

On remarque aussi que l'étape la plus lente de toutes les phases de l'algorithme est le calcul de nombre de croisement global des arêtes dans le graphe. En effet, Sugiyama présente une fonction de sommes imbriqués dont la complexité est dans l'ordre de n^4. Il serait possible d'améliorer le temps d'exécution en évitant de recalculer tous les croisements dans tous les niveaux mais plutôt de rectifier le nombre des croisements au fur et à mesure qu'un sommet est déplacé dans un niveau. Le ralentissement est dans de la fonction getCrossingNumber() dans la classe Matrix. En effet, cette fonction est d'une complexité [(n-1) + (n-2) + (n-3) + ... + 2 + 1] * [(m-1) + (m-2) + (m-3) + ... + 2 + 1] pour un tableau de taille n x m (4 boucles for () imbriqués sur un même tableau binaire en 2D) où n est le nombre de sommet dans un niveau et m le nombre de sommet dans le niveau suivant!

Il serait également intéressant de développer un jeu de test très varié et comportant un grand cas de variétés de dispositions possible.

Pour améliorer la clarté du dessin du graphe lorsque le nombre de croisement est grand (faible pointage), on pourra générer des mises en pages localisés à l'intérieur même du graphe. L'idée est d'afficher, lorsqu'on sélectionne une boîte, toutes les entités qui lui sont proches à un niveau donné en paramètre.

Quand on sélectionne une entité (une classe dans le diagramme), on met en évidence tous ces enfants et parents en dépriment des autres sur le même graphe! On pourra aussi limiter le nombre de niveau affiché en le modifiant dans les paramètres.

Il y a aussi le cas des liens entre deux nœuds se trouvant sur un même niveau pour qui on doit dessiner un lien courbé pour ne pas se superposer avec d'autres entités sur la même ligne!

On peut afficher une barre de progrès informant l'utilisateur que le programme est en train de faire du calcul utile. On pourra aussi permettre à l'utilisateur d'interrompre le calcul et même de modifier les paramètres utilisés.

Le niveau de débogage : un entier (selon le numéro entre 0 et 10) (pour chaque étape, on peut aussi afficher les messages de débogage) plus ou moins détaillés.

7. Conclusion.

L’implémentation de l’algorithme de Sugiyama dans Ptidej a donné au programme une façon claire et lisible de présenter les diagrammes de classes construits par rétro conception à partir du code source. Notre implémentation à été découpée selon les étapes proposées pas l’auteur original, Sugiyama [1]. On a donc généré une hiérarchie propre, minimiser les croisements, calculer les positions horizontales des sommets et finalement afficher une image du graphe d’un diagramme de classes.

Malgré les problèmes techniques rencontrés, nos objectifs ont été atteints par l’intégration de l’algorithme de Sugiyama dans le projet de mise en pages des diagrammes de classes UML. Ce pendant, plusieurs extensions et améliorations futures peuvent êtres accomplis pour rendre le programme plus robuste et performant.

Finalement, même si la réduction du nombre de croisement améliore nettement la clarté et visibilité d’un graphe, au-delà d’une certaine limite, lorsque le nombre d’arêtes et/ou des sommets devient grand, il n’est plus possible d’améliorer la visibilité puisqu’il devient très difficile de réduire les croisements.

On conseille alors l'utilisation de graphes creux ou peu denses avec un nombre raisonnable de sommets et de liens.

8. Divers.

8.1 Eclipse.

Plat forme de développement indispensable pour mon projet. En effet, au cours de mon projet, j'ai eu l'occasion d'apprendre l'utilisation d'éclipse. Cet outil de développement était très utile pour gestion des différents projets, paquetages et classes. De plus ça m'a permis d'interagir avec le **CVS** et ainsi être capable de fusion mon code avec celui des autres développeurs et avoir toujours une copie de mon espace de travail.

8.2 Remerciements.

Yann-Gaël Guéhéneuc était responsable de la supervision de mon projet. Je tiens personnellement à souligner d'une part le grand travail qu'il a consacré pour le développement et la mise sur pied de **Ptidej** et d'autre part sa présence et support inégalés tout au long du déroulement de mon projet. Bravo !

J'ai apprécié l'environnement de travail avec **Yann-Gaël** ainsi que tous les membres du laboratoire. J'ai eu la chance de connaître un grand spécialiste du génie logiciel et de profiter de ses conseils et de son expertise dans le domaine.

Je tiens à remercier **Sung Hui Park** pour ses précieux conseils et support avant, pendant et après le déroulement de mon projet ainsi que le Docteur **Stéphane Monnier** pour leurs supervisions du projet.

Je tiens à remercier le **Conseil de Recherche en Sciences Naturelles Et Génie du Canada** (CRSNG) pour leur bourse en recherche avec laquelle a été financé mon projet.

Je tiens également à remercier tous les membres du laboratoire **GEODES** pour le bon environnement de travail que j'ai trouvé : **David, Pierre, Karim, Guillaume, Salima, Elias, Youssef, Jihan, Naouel, Moustapha, Farida.**

9. Références.

[1]. K. Sugiyama, S. Tagawa, and M.Toda. Methods for Visual Understanding of Hierarchical Sysytem Structre. IEEE Transactions on Systems, Man and Cybernetics, 11(2) : 109-125, 1981.

[2]. Kusnadi, Craig Beebe, and Jo Dale Carothers. Digraph Visualisation Using a Neural Algorithm with a Heuristic Activation Scheme. IEEE Transactions on Systems, Man and Cybernetics-part B, 28(2) : 562-572, august 1998.

[3]. Dabo Sun and Kenny Wong. Departement of Computing Science, University of Alberta. On evaluating the Layout of UML class Diagrams for program Comprehension. IEEE Computer Society, Procedings of the 13th Intenational Workshop on Program Comprehension (IWPC 05).

[4]. K. Sugiyama, S. Tagawa, and M.Toda, « Effective representations of hierarchical structre ». International Institute for Advanced Study of Social Information Science, Fujisu Limited, Research Rep. No. 8, pp. 1-29, 1979.

[5]. Graph Layout for Domain-Special Modeling.
Denis Dubé.
Supervisor: Prof. Hans Vangheluwe
School of Computer Science McGill University, Montréal, Canada
A thesis submitted to the Faculty of Graduate Studies and Research in partial fullment of the requirements of the degree of Master of Science in Computer Science

10. Annexe.

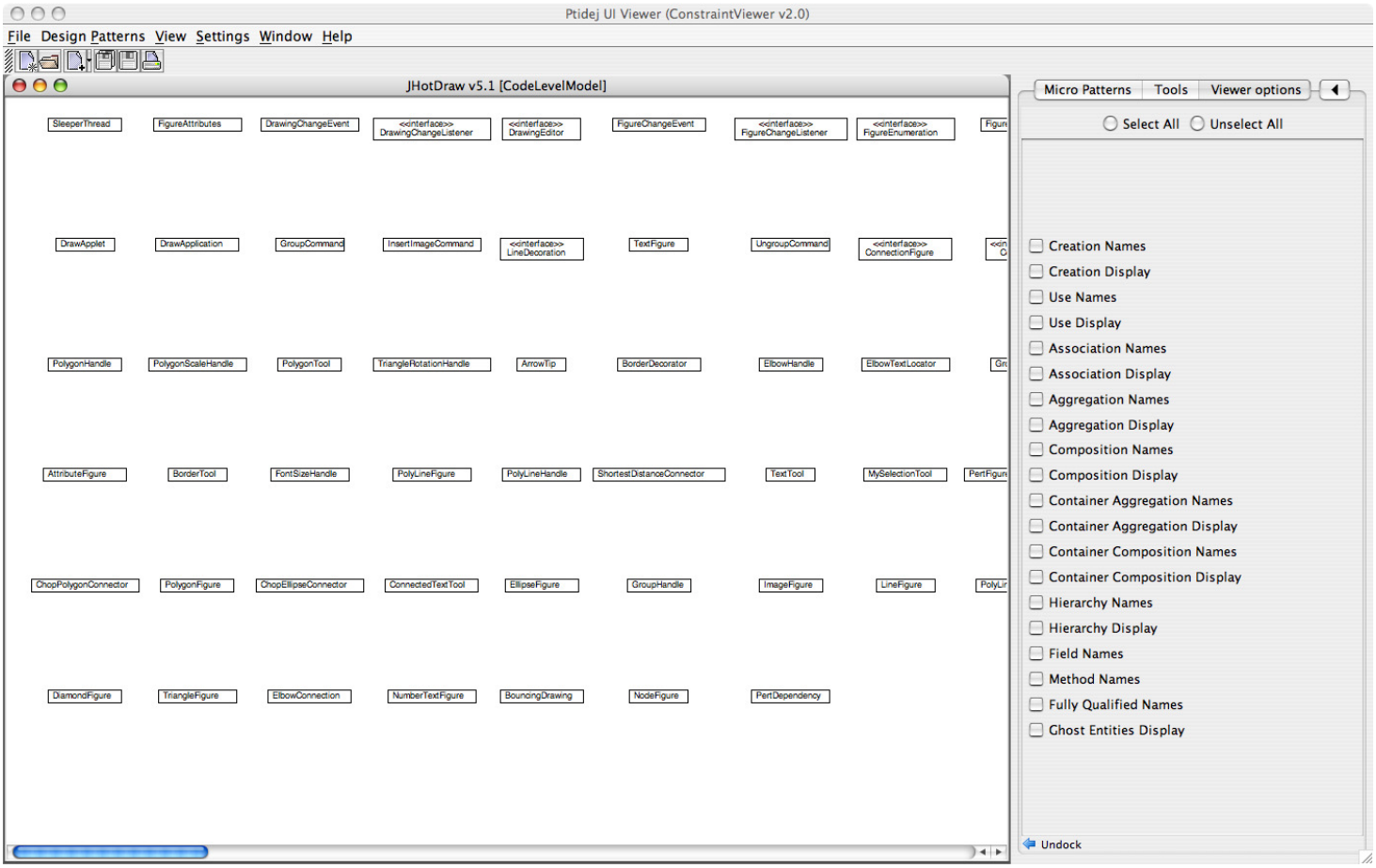

Figure 8 : Pas de liens sélectionnés.

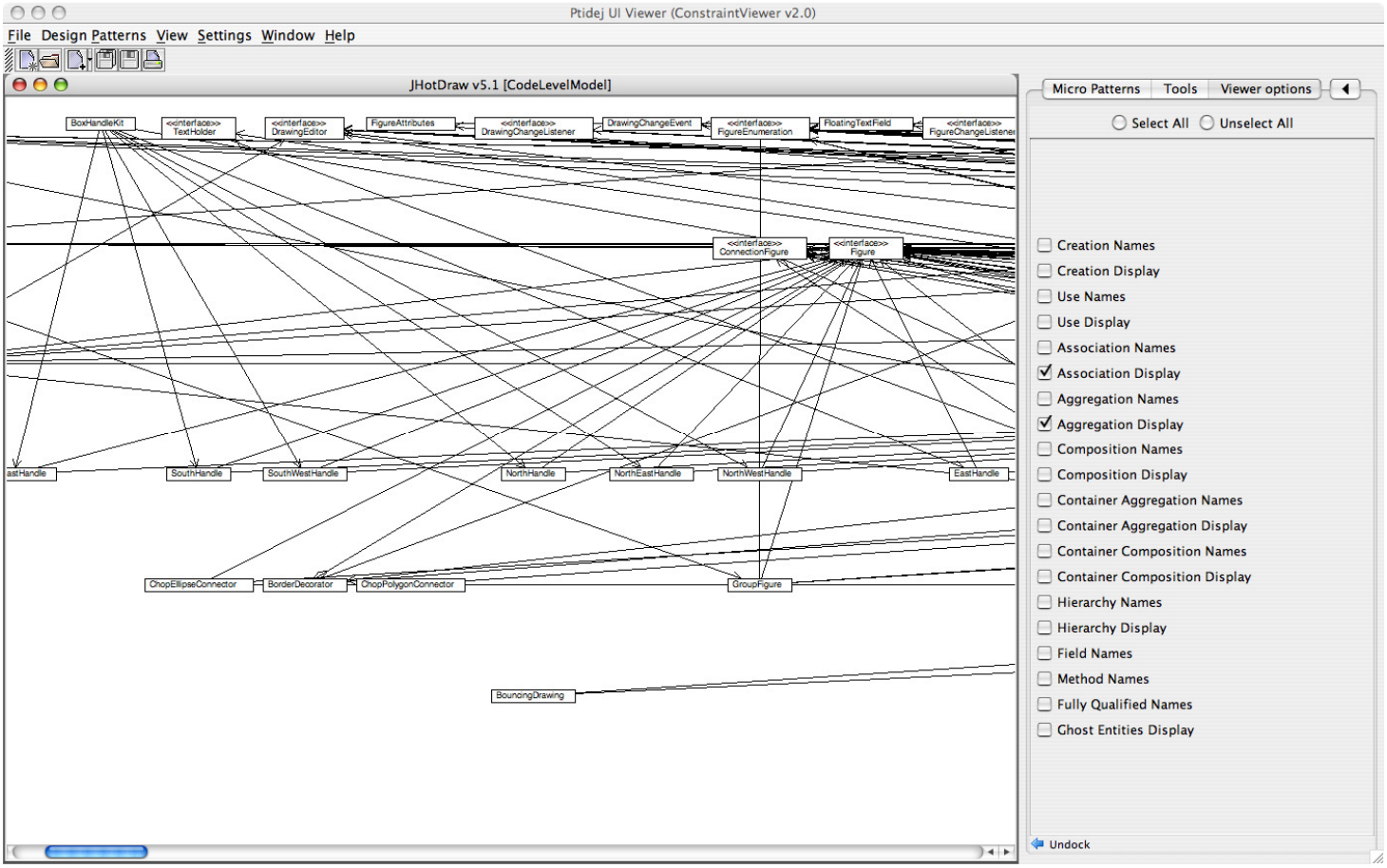

Figure 9 : Une portion d'un diagramme de classes. On remarque donc, que si le nombre d'arêtes devient très important, on atteint les limites de réductions des croisements escomptés.