# Visualisation de Graphes Séries-Parallèles
# Rapport de PSTL

GRUENPETER Morane, RETAIL Tanguy

22/05/2016

Encadrants : DIEN Matthieu et GENITRINI Antoine

# Table des matières

**1 Introduction** **3**

**2 État des lieux** **4**
2.1 Technologies existantes . . . . . 4
2.2 Visualisation d'un arbre (d'après TreeDisplay) . . . . . 4

**3 Les Graphes Séries-Parallèles (GSP)** **7**
3.1 Définitions . . . . . 7
3.2 Reconnaissance . . . . . 9

**4 Visualisation de Graphes Séries-Parallèles** **11**
4.1 Contraintes d'affichage . . . . . 11
4.2 Algorithme . . . . . 12
4.2.1 Etape 1 : Trouver la profondeur du graphe et de chaque nœud . . . . . 13
4.2.2 Etape 2 : Initialiser les coordonnées de chaque nœud et identifier les diamants . . . . . 14
4.2.3 Etape 3 : Appliquer les décalages des sous-graphes . . 17
4.2.4 Etape 4 : appliquer les décalages des diamants . . . . 18

**5 Implémentation dans le cadre du PSTL** **19**
5.1 Structure de données et représentation . . . . . 19
5.2 Formats d'entrée . . . . . 20
5.3 Formats de sortie . . . . . 20
5.4 Interface Graphique . . . . . 20

**6 Conclusion** **22**

**7 Bibliographie** **22**

# 1 Introduction

Les récents progrès de la génération aléatoire permettent de nombreuses applications, notamment en algorithmique expérimentale (simulations), en physique, en bio-informatique ou encore au niveau du génie logiciel. Les générateurs de structures combinatoires nous permettent de générer uniformément des graphes (par exemple des arbres ou des graphes séries-parallèles) de plusieurs millions de nœuds en à peine quelques secondes. Habituellement, ces graphes sont stockés dans des fichiers textes structurés (dot, xml, ...). On veut naturellement obtenir une visualisation graphique de tels objets.

Il existe plusieurs outils permettant de visualiser des graphes quelconques en utilisant des algorithmes souvent coûteux, le plus connu étant graphviz. Dans le cas des arbres, il existe un algorithme de visualisation en temps linéaire qui a été implémenté dans deux outils, l'un en Python lors d'un précédent PSTL [1] et l'autre en Ocaml. Nous aimerions adapter cet algorithme à une autre classe de graphes que sont les graphes séries-parallèles.

Nous commencerons par faire un état des lieux des technologies existantes et des technologies dont nous nous sommes inspirées. Par la suite, nous verrons ce qu'est un graphe série-parallèle et comment il est possible de le reconnaître. Nous rentrerons alors dans le vif du sujet qui est la mise en oeuvre d'un algorithme linéaire permettant le calcul des coordonnées. Ce document intervenant dans un travail étudiant du master STL de l'UPMC, nous présenterons l'implémentation Java réalisée dans le cadre du projet avant de passer à la conclusion et aux suites envisageables.

# 2 État des lieux

## 2.1 Technologies existantes

Graphviz est un outil open source performant d'affichage de graphe. Graphiz, appelé aussi Graph Visualization Software, a été initialement développé par AT&T Labs Research. Cet outil prend en entrée des fichiers textuels en langage DOT afin de fournir en sortie un fichier de visualisation en SVG, PDF ou Postscript. Cependant, Graphviz a quelques défauts concernant le choix de l'ordre d'affichage, il ne conserve normalement pas l'ordre des fils mis en entrée. Une option permet en fait de conserver l'ordre, au détriment de la complexité. L'algorithme utilisé sur Graphviz est divisé en quatre passes, en utilisant un tri-topologique. Les passes regroupent les actions suivantes : procédure préalable pour éliminer les cycles, classement par ordre topologique, ordonnancement et positionnement [2].

Nous allons nous inspirer de l'outil Graphviz, de l'implémtation de Treedisplay et utiliser les fichiers DOT en entrée. Le langage DOT est un langage de description de graphe. Il permet la description de graphe orienté et non-orienté de manière lisible et facile à interpréter. Nous avons utilisé un parseur DOT existant dans notre programme afin de lire en entrée la description du graphe en texte simple.

Lors de notre travail, nous avons mis en ligne notre projet sur le site Github [3]. Vous êtes libre d'utiliser, de modifier ou de diffuser nos fichiers, en citant vos sources.

## 2.2 Visualisation d'un arbre (d'après TreeDisplay)

Les algorithmes desquels Treedisplay est inspiré sont :

— Knuth (1971)
— Whetherell et Shanon (1979)
— Walker (1990) [4, 5]

C'est l'algorithme de Walker qui permettra d'affiner l'algorithme sans perdre en complexité avec un raisonnement en deux passes sur l'arbre. La première passe pour mémoriser un décalage et la deuxième passe pour propager le décalage sur chaque sous-arbre[1].

Nous nous sommes basés sur l'algorithme TreeDisplay en raison de ses performances et du fait que les graphes séries-parallèles ont des caractéristiques communes avec les arbres, puisque ceux-ci sont une sous-classe des graphes série-parallèles.

**Explications et fonctionnement**

L'algorithme est divisé en deux passes importantes sans aller-retour en profondeur.
La première passe a pour but :

— déterminer l'ordonnée d'un nœud
— si possible centrer un nœud par rapport à ses enfants, sinon retenir le décalage.

L'ordonnée d'un nœud dans un arbre est triviale, c'est simplement sa profondeur. L'algorithme calcule en premier les abscisses des enfants d'un nœud afin de pouvoir le placer.
On cherche donc à placer un nœud entre ses enfants, trois cas peuvent survenir :

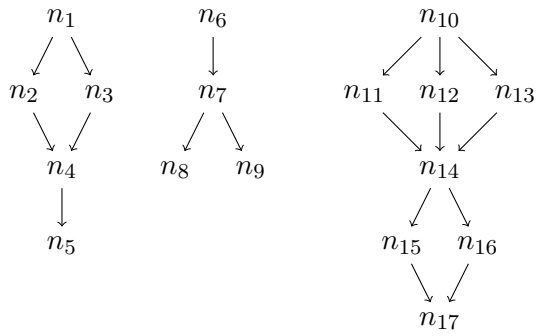
1. La place du nœud centré au milieu de ses enfants est libre. Dans ce cas l'abscisse est la moyenne entre les abscisses de son fils le plus à gauche et de son enfant le plus à droite.
2. La place du nœud n'est pas disponible, un nœud occupe déjà la place. Dans ce cas, le nœud est positionné à droite du dernier nœud sur son ordonnée. On retient la différence entre la place obtenue et la place souhaitée, qu'on appelle décalage.
3. Le nœud est une feuille, et n'a donc pas d'enfant. Sa place est à droite du dernier nœud sur son ordonnée.

La deuxième passe applique donc chaque décalage précédemment calculé au sous-arbre, afin de recentrer le sous-arbre sous son parent.
Chaque algorithme effectue un simple parcours en profondeur. La complexité est donc linéaire.
A la fin, l'algorithme nous propose un arbre dont :

- Les arêtes de l'arbre ne s'intersectent pas.
- Les nœuds de même profondeur sont dessinés sur le même axe horizontal.
- L'ordre des enfants est respecté
- Un nœud parent est centré vis à vis de ses enfants.
- Un sous-arbre est dessiné de la même manière, peu importe sa place dans l'arbre.
- Les nœuds enfants d'un nœud parent sont espacés de manière homogène.

En réalité, nous avons remarqué que cette dernière contrainte sur l'espacement homogène des nœuds n'est pas vérifiée.

# 3 Les Graphes Séries-Parallèles (GSP)




## 3.1 Définitions

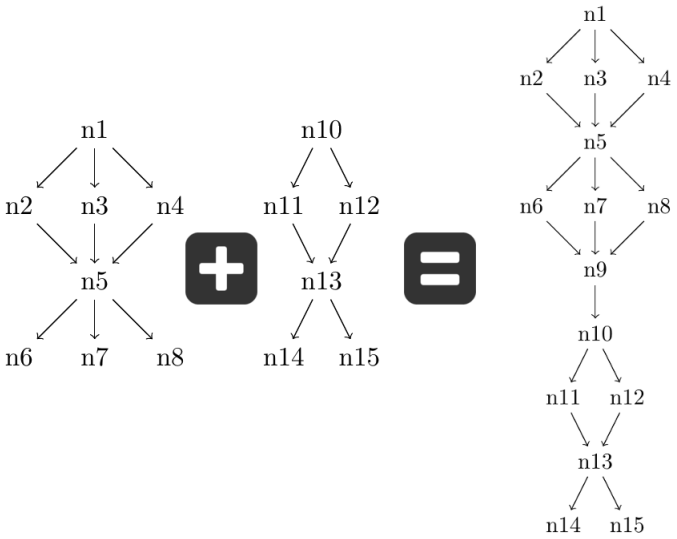
Les graphes séries-parallèles (GSP) sont des **graphes orientés** que nous définissons récursivement. Un GSP est soit :

— L'ensemble vide.
— Un nœud.
— Un GSP composé en parallèle avec un autre GSP.
— Un GSP composé en série avec un autre GSP.

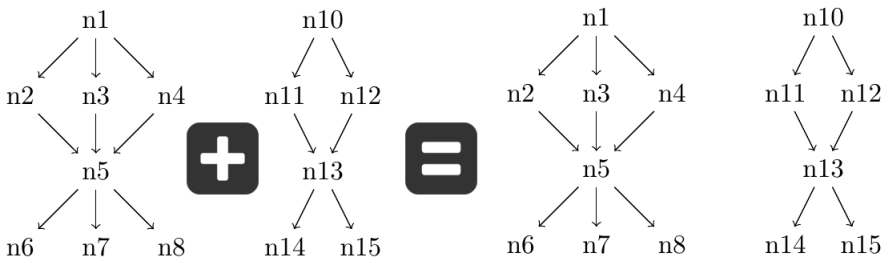
Un GSP peut avoir un ou plusieurs nœuds source, c'est à dire un nœud sans prédécesseurs (parents). Un GSP peut également avoir un ou plusieurs nœuds sans successeurs (enfants), dits nœuds puits. Si le graphe contient qu'un seul nœud il est considéré comme la source et le puits du graphe.

Dans le cadre de l'implémentation (détaillée en section 5) nous utilisons une liste d'adjacence pour représenter les GSP, avec une liste de sources et une liste de puits pour un graphe donné.

**Composition en série :**
Soient deux GSP $G_1$ et $G_2$ avec respectivement $n_1$ et $n_2$ sources, $m_1$ et $m_2$ puits.
Une composition en série est l'union telle que $G_3 = G_1 \otimes G_2$ où les sources de $G_3$ est l'ensemble $n_1$ et les puits de $G_3$ est l'ensemble $m_2$. Les ensembles $m_1$ et $n_2$ sont reliés avec un nœud intermédiaire $k$.




**Composition en parallèle :**
Soient deux GSP $G_1$ et $G_2$ avec respectivement $n_1$ et $n_2$ sources, $m_1$ et $m_2$ puits.
Une composition en série est l'union disjointe telle que $G_3 = G_1 \oplus G_2$ où les sources de $G_3$ est l'ensemble $n_1 \cup n_2$ et les puits de $G_3$ est l'ensemble $m_1 \cup m_2$




### 3.2 Reconnaissance

A la première découverte des graphes séries-parallèles nous avons remarqué leurs caractéristiques particulières. Ce qui nous a mené vers l'idée de reconnaissance des séries-parallèles avant la visualisation de ces graphes.
La reconnaissance des GSP est un sujet à part entière et cela ne fait pas partie de notre projet. En revanche, il est important de bien comprendre la nature de ces graphes afin de trouver l'affichage optimal. Pour ne pas permettre le rejet d'un graphe pendant l'application de l'algorithme de visualisation, une passe de reconnaissance pourrait être mis en place.
Dans cette section nous allons détailler les caractéristiques des GSP qui nous ont permis de les reconnaître et qui est la base pour l'algorithme de visualisation détaillé dans la section 4.

Avec les définitions d'un GSP, les GSP sont composés de disjonction et de jonction. Nous allons appeler les nœuds de disjonction $fork$ et les nœuds de jonction $join$. Un couple [$fork$-$join$] prend la forme d'un couple de parenthèses dans un GSP.
Dans le cadre de l'algorithme de reconnaissance d'un GSP nous avons gardé seulement la détection de circuit pour le traitement préalable à la visualisation.

**Détection de circuit**

*Lemme* : un GSP ne contient pas de circuit.

*Preuve* : Un GSP minimal (celui qui contient seulement un nœud) ne contient pas de circuit. Un GSP composé par composition série ne contient un circuit si et seulement si l'un des deux graphes contient un circuit, mais le circuit ne sera pas créé par la composition série, puisque cette composition crée un chemin orienté entre deux GSP distincts et indépendants. Pareillement pour un GSP composé en parallèle, puisqu'aucun chemin n'est ajouté au graphe.

L'algorithme de détection de circuit, appelé *cycleDetection*, utilise un parcours en profondeur (DFS- depth first search) pendant lequel il associe une couleur de traitement au nœud. Un nœud non traité est en blanc, pendant le traitement d'un nœud, on change la couleur en gris et à la fin du traitement, on change la couleur en noir. Cet algorithme est populaire et sera donc omis.

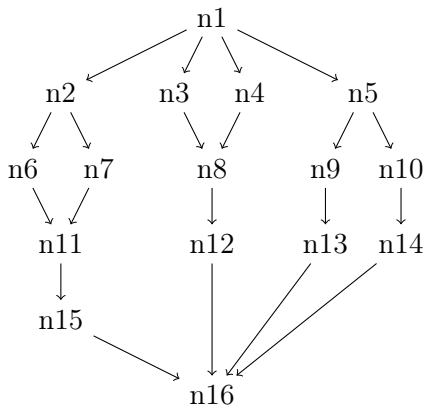
# 4 Visualisation de Graphes Séries-Parallèles

## 4.1 Contraintes d'affichage

Contrairement à un arbre, l'ordonnée d'un nœud n'est pas triviale puisque sa profondeur dans un chemin dépend du chemin emprunté. Ainsi, nous définissons la profondeur d'un nœud dans un GSP comme la profondeur maximale de ce nœud dans tous les chemins possibles. L'ordonnée d'un nœud sera donc sa profondeur maximale dans le graphe.
Tout comme dans un arbre, un nœud peut être parent de plusieurs nœuds (fork), et dans ce cas nous le centrons par rapport à ses enfants. Mais dans un graphe, un nœud peut également être fils de plusieurs parents (join). Selon les cas, nous déciderons de centrer ce nœud entre ces fils, ou d'aligner ce nœud avec un fork.
Si une composante connexe possède plusieurs sources, nous prenons la liberté d'insérer une nouvelle source parente de ces sources, afin de simplifier l'algorithme.
Ainsi, les contraintes d'affichages d'un GSP sont :

- Les arêtes du graphe ne s'intersectent pas.
- Les nœuds de même profondeur sont dessinés sur le même axe horizontal.
- Le graphe est dessiné de la manière la plus compacte possible.
- Un nœud fork est centré vis à vis de ses fils.
- Un join qui ferme exactement toutes les branches d'un fork est aligné vis-à-vis de celui-ci.
- Un join qui ne ferme pas toutes les branches d'un fork est centré entre ses parents.
- Un fork qui ne possède pas de join qui ferme uniquement ses branches, est centré vis-à-vis de ses fils.

Nous définissons par ailleurs la notion de diamant :
Un diamant complet est un couple join-fork dont le join ferme uniquement et exactement toutes les branches engendrées par le fork. Un diamant est dit join-incomplet lorsqu'il n'existe pas de join qui ferme uniquement et exactement les branches engendrées par un fork. De la même façon, un diamant sera dit fork-incomplet lorsqu'il n'existe pas de fork qui engendre uniquement et exactement les branches fermées par un join.

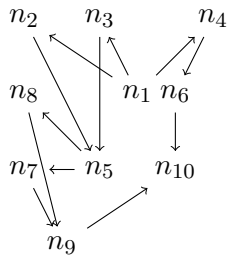



Sur cet exemple, nous avons :

— deux diamants complets qui sont les couples $\{n_2,n_{11}\}$ et $\{n_1, n_{16}\}$.
— un diamant join-incomplet $\{n_8\}$.
— un diamant fork-incomplet $\{n_5\}$.

## 4.2 Algorithme

Pour faciliter la compréhension des algorithmes, nous présenterons l'évolution du graphe ci-dessous dont nous souhaitons améliorer l'affichage :

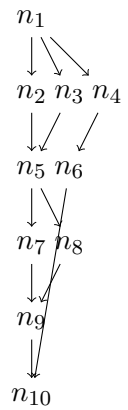


### 4.2.1 Etape 1 : Trouver la profondeur du graphe et de chaque nœud

**Algorithme 1 :** profondeurs(Node, int) **[**$\mathcal{O}(\mathcal{N})$**]**

**Données :** Node node ; int profondeur=0
**Résultat :** void
**si** $profondeur > node.y$ ***ou*** *première fois qu'on passe par ce nœud*
**alors**
  $node.y \leftarrow profondeur$;
  **si** $graphDepth < profondeur$ **alors**
    $graphDepth \leftarrow profondeur$;
  **fin**
  **pour chaque** *enfant de node* **faire**
    $profondeurs(enfant, profondeur + 1)$;
  **fin**
**fin**

Dans cette première étape, nous effectuons un parcours en profondeur dans le graphe. Chaque nœud pouvant être atteint par différents chemins, il est important de parcourir ces chemins. La profondeur d'un nœud sera donc le chemin le plus long menant à ce nœud, et correspondra donc à son ordonnée. On en profite pour déterminer la profondeur du graphe qui est le plus long chemin dans le graphe.
La profondeur du graphe nous permet d'allouer un tableau d'entier "prochain", qui nous sert à déterminer la prochaine place disponible sur chaque ligne du graphe.



### 4.2.2 Etape 2 : Initialiser les coordonnées de chaque nœud et identifier les diamants

Soit prochain[y] la prochaine place disponible à l'ordonnée y.

**Algorithme 2 :** initialisation(Node) **[**$\mathcal{O}(\mathcal{N})$**]**

```
Données : Node node, int prochainePlaceDispoDroite
Résultat : void
/* premiere partie                                                    */
pour chaque parent de node faire
    si parent n'a pas ete visite alors
        parentsOntEteVisites ← false;
        break;
    fin
fin
si parentsOntEteVisites = false et node a deja ete traite alors
    retourner ;
fin
si tous les parents ont ete visite alors
    node ← visite;
    majDiamantsIdentifies(node);
fin
/* seconde partie                                                     */
si node n'a pas encore ete espace alors
    prochain[node.y] ← max(prochain[node.y],
     prochainePlaceDispoDroite);
    node ← espace;
fin
pour chaque enfant de node faire
    initialisation(enfant, prochain[node.y]);
fin
/* troisieme partie                                                   */
si le nœud n'a pas ete traite alors
    si node n'a pas d'enfant alors
        placeSouhaite←prochaine place disponible sur la ligne;
    sinon
        placeSouhaite←au centre de ses fils le plus à gauche, et le plus
         à droite;
    fin
    node.x ← max(placeSouhaite, prochain[node.y]);
    next[node.y] ← max(prochain[node.y], node.x+1);
    node.decalage ← abs(node.x - placeSouhaite);
    node ← traite;
fin
```

En première partie de l'algorithme, avant l'appel récursif, nous effectuons un parcours en profondeur par tri topologique pour s'assurer d'avoir parcouru tous les chemins menant à ce nœud. Il est alors possible de déterminer le diamant auquel appartient ce nœud.

En seconde partie, la variable prochainePlaceDispoDroite nous permet de connaitre le noeud qui se situe le plus à droite parmi toutes les lignes horizontales lors de notre parcours avec tri topologique afin de placer le noeud à droite de cette valeur. Ainsi, nous évitons les croisements qui seraient dus à un sous-graphe trop large. Dans notre exemple, si $n_5$ possédaient énormément de fils, nous ne voudrions pas que les arcs depuis $n_5$ vers ses derniers fils intersectent l'arc $(n_6 \to n_{10})$.

En troisieme partie, après l'appel récursif, lorsque le nœud est visité pour la première fois, nous essayons de le centrer vis-à-vis de ses enfants, tout comme dans le TreeDisplay. Si un nœud est déjà présent à cette place souhaitée, nous le plaçons à le dernière place disponible sur son ordonnée. Si la place souhaitée n'est pas retenue, il faut enregistrer la différence entre la place voulue et la place effectivement obtenue. Il faut également mettre à jour la dernière place disponible sur son ordonnée.

Remarquons que si les parents du nœuds n'ont pas tous été visités et que le nœud courant l'a déjà été, il est inutile de continuer l'appel récursif puisqu'aucun traitement ne sera alors effectué, augmentant inutilement la complexité de l'algorithme.

**Algorithme 3 :** majDiamantsIdentifies(Node) **[**$\mathcal{O}(\mathcal{N})$**]**

```
Données : Node node
Résultat : void
si node est un join alors
    tailledenode ← nombredeparentsdenode;
    tant que tailledenode > 0 faire
        si tailledenode = tailledudernierforkempile alors
            liste des diamants ← {node, dernier fork empile};
            break;
        sinon si tailledenode > tailledudernierforkempile alors
            tailleDeNode←tailleDeNode - taille du dernier fork empile
             + 1;
        sinon
            On rempile le fork avec la valeur tailleDuFork -
             tailleDeNode +1;
            liste des diamants ← {node, null};
            break;
        fin
    fin
fin
si node est un fork alors
    liste des forks ← node;
fin
```

Une partie déterminante de l'algorithme est de savoir identifier les diamants.

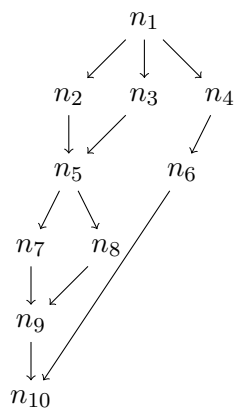
Dans l'algorithme Initialisation, nous effectuons un tri topologique. majDiamantsIdentifies est donc appelée uniquement lorsque tous les ancètres d'un nœud ont été visité, ce qui nous permet de nous assurer que nous connaissons bien tous les éléments à notre disposition afin de reconnaître les diamants.

Premier cas :

- C'est un join. Dans ce cas-là, nous essayons de l'associer au dernier fork présent dans la pile.
- Si ils ont la même valeur, ils sont associés. C'est un diamant complet.
- Si le join a une valeur plus grande, alors il a fermé une partie du fork. Le fork est un diamant fork-incomplet. Dans ce cas-là, nous mettons à jour la nouvelle valeur du join puis essayons de l'associer avec le fork suivant.
- Si jamais la valeur du join est inférieure à la valeur du fork, alors le join

ne ferme pas complètement le fork. C'est un diamant join-incomplet. Deuxième cas :

— Le nœud est un fork. Nous l'empilons simplement avec sa valeur de fork qui correspond à son nombre de fils.

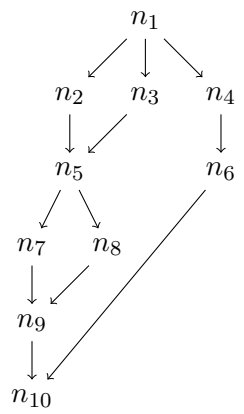



### 4.2.3 Etape 3 : Appliquer les décalages des sous-graphes

**Algorithme 4 :** appliquerDecalages(Node, float) **[$\mathcal{O}(\mathcal{N})$]**

**Données :** Node node ; float decalageCumule=0
**Résultat :** void
**si** *nœud n'a pas été visité* **alors**
  $node.x \leftarrow node.x + decalageCumule$;
  $decalageCumule \leftarrow decalageCumule + node.decalage$;
  $node.decalage \leftarrow 0$;
  node ← visité;
  **pour chaque** *enfant de node* **faire**
    $appliquerDecalages(enfant, decalageCumule)$;
  **fin**
**fin**

Le décalage est ici calculé pour les parents uniques qui n'ont pas pu se centrer entre leurs enfants par manque de place sur leur ligne. Le sous-graphe est donc décalé, ce sont les enfants qui viennent se positionner par rapport à leur parent.



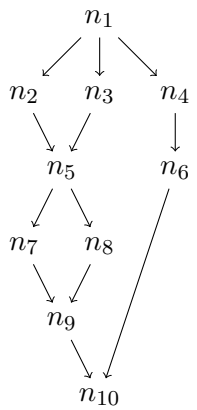

### 4.2.4 Etape 4 : appliquer les décalages des diamants

**Algorithme 5 :** applyDiadmondsOffsets(Node, float) **[$\mathcal{O}(\mathcal{N})$]**

```
Données : Node node ; float offsum=0
Résultat : void
si nœud n'a pas été visité alors
    si nœud est join dans un diamant alors
        si ce diamant est complet alors
            decalage ← forkDiamant.x - joinDiamant.x;
            si decalage != 0 alors
                offsum ← decalage;
            fin
        sinon
            /* distance entre la position a atteindre et la
               position actuelle avec le decalage cumule    */
            offsum ← offsum + ABS(centreDesParents.x/2 -
              (node.x+offsum));
        fin
        node.x ← node.x + offsum;
        node ← visité;
        pour chaque enfant de node faire
            applyDiadmondsOffsets(enfant, offsum);
        fin
    fin
fin
```

On ne s'occupe pas des diamants join-incomplets, ils ont déjà centrés

entre leurs fils à l'étape précédente.
La passe vérifie donc si le diamant est complet ou bien fork-incomplet.
Si le diamant est complet, on applique donc le décalage nécessaire au join et à l'éventuel sous-graphe afin de les aligner vis-à-vis du fork.
Si le diamant est fork-incomplet, on centre le join et l'éventuel sous-graphe entre les parents du join.




**Complexité totale :**
Chaque passe effectue un unique parcours en profondeur dont la complexité est linéaire dans le nombre de nœuds. Le nombre de passes étant constante, il est donc évident que la complexité totale de notre algorithme est linéaire.

# 5 Implémentation dans le cadre du PSTL

Nous avons choisi d'implémenter les algorithmes en Java, en ajoutant à cette implémentation une interface graphique afin de voir rapidement le résultat des algorithmes. En premier lieu nous avons traduit l'algorithme de Treedisplay [1], qui a été implémenté en Python. Avec cette première implémentation nous avons remarqué quels sont les points problématiques à traiter avec notre algorithme. Puis, nous avons créé la structure de donnée GSP, détaillée ci-après.

## 5.1 Structure de données et représentation

Nous avons choisi de représenter le graphe avec une liste d'adjacence. Un GSP possède une liste de nœuds sources et une liste de nœuds puits, qui appartiennent à la même composante connexe. Il contient aussi un pointeur vers une autre composante connexe du GSP. Les nœuds possèdent quatre

champs {id, valeur, x, y} ainsi que la liste de leurs parents et de leurs enfants. Nous avons choisi de vous présenter la structure de données utilisée dans le language Java ci-dessous :

```
public class Node<T> extends Point.Float {
        public final String id;
        public T value;
        private List<Node<?>> parents;
        private List<Node<?>> children;
        //Les coordonnees (x,y) sont heritees.
}

public class Graph implements Iterable<Graph> {
        public Graph voisin; //Composante en parallele
        List<Node<?>> sources;
        List<Node<?>> sinks;
}
```

### 5.2 Formats d'entrée

Le parseur de fichiers [6] parse les fichiers directement en Graphe GSP avec la bibliothèque *Java-based Parser for Graphviz Documents*, nous permettant d'utiliser les informations ainsi obtenues afin de construire notre graphe.

### 5.3 Formats de sortie

La génération d'image (PNG, JPEG, BMP) est possible depuis l'interface graphique que nous proposons.

La génération pour TIKZ est une sortie en texte qui peut être intégrée dans un fichier TEX. Ce texte contient la représentation texte du graphe avec ses coordonnées, calculées depuis le logiciel. Il est également possible d'exporter avec Tikz depuis l'interface graphique.

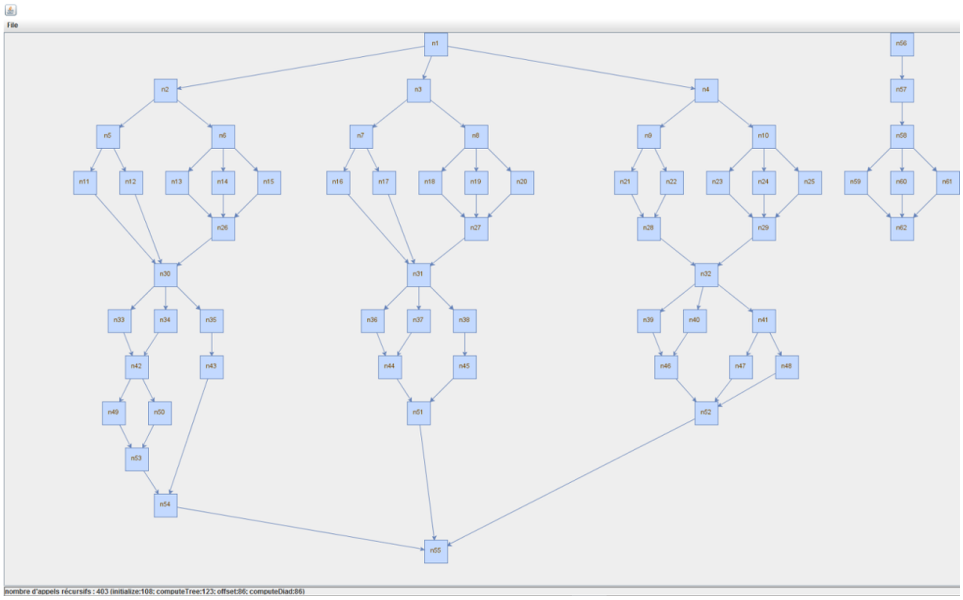
### 5.4 Interface Graphique

L'interface graphique permet d"apprécier le résultat rapidement. Nous avons utilisé Java Swing, la bibliothèque Java pour les interfaces graphiques, et MxGraph, une bibliothèque Java facilitant l'affichage de graphe.

Dans le menu File, il est possible de choisir un fichier DOT afin d'en afficher le graphe série-parallèle. Ceci dit, aucune vérification n'est effectuée

afin de vérifier si le fichier est bien formé, ou si le contenu représente bien un graphe série-parallèle. Dans ce même menu, il est possible d'exporter le graphe affiché en format image ou dans un fichier Tikz.

En pied de fenêtre, une barre de status communique le nombre de passes qui ont été nécessaires afin de générer les coordonnées des nœuds du graphe.



# 6 Conclusion

L'algorithme respecte donc les contraintes fixées afin d'obtenir une visualisation facilement interprétable par un humain aussi bien pour un arbre que pour un graphe série-parallèle. Bien que ce fût parfois laborieux, nous avons réussi à garder la complexité en temps linéaire et avons donc pleinement répondu au cahier des charges qu'il nous avait été donné de respecter.

La lecture des fichiers en entrée fût grandement simplifiée grâce aux outils existants, mais nous regrettons de n'avoir eu le temps de permettre une reconnaissance de graphes séries-parallèles entièrement fonctionnelle. Cette fonctionnalité pourrait donc être une suite envisageable dans le cadre d'un PSTL. Il serait encore une fois pertinent d'essayer d'adapter l'algorithme à une classe de graphes similaire et plus large que la classe des graphes séries-parallèles.

# 7 Bibliographie

## Références


[1] Erika Baena et Diana Malabard. *Visualisation d'arbre de grande taille.* PSTL-UPMC, Encadrant : Antoine Genitrini, 2014.

[2] Emden R. Gansner, Eleftherios Koutsofios, and Stephen North. Drawing graphs with dot, 2015.

[3] Gruenpeter Morane and Retail Tanguy. Dessiner un graphe série-parallèle (https ://github.com/fyrthis/pstl), 2016.

[4] J. Walker II. A node-positioning algorithm for general trees. *Software – Practice and Experience*, 20(7)(10) :685–705, 1990.

[5] Michael J¨unger Christoph Buchheim and Sebastian Leipert. Improving walker's algorithm to run in linear time. *Universit¨at zu K¨oln, Institut f¨ur Informatik.*

[6] Alexander Merz. Jpgd manual, 2006.