Tristan MOLIN
Florian FICHANT
Martin LOCQUEVILLE

UQAC
Université du Québec
à Chicoutimi

Université du Québec à Chicoutimi

8INF914 - Visualisation Analytique

Etude des algorithmes de dessin d'arbres
Travail de session
Rapport

23 juin 2017

Table des matières

1 Introduction

L'objectif de notre projet est de parvenir à implémenter en *Python*, au sein de l'outil d'étude de graphe *Tulip*, un algorithme de dessin d'arbre, le plus optimal possible. Pour ce faire, nous avons recueilli et analysé les travaux de recherche dans quatre articles.

Ces articles avaient également comme problématique l'optimisation de dessin de graphe. Chacun d'eux cherchant à optimiser l'implémentation d'un algorithme de dessin d'arbre en prenant en compte les travaux de leurs prédécesseurs (Nous avons pris soin de noter les références de ces articles en fin de rapport).

Suite à l'étude de ces articles, nous allons premièrement implémenter les algorithmes résultant des travaux des auteurs de nos quatre articles de référence. C'est après cela que nous allons écrire de manière optimisée et implémenter notre solution algorithmique en *Python*, en utilisant la bibliothèque *Tulip*.

Pour finir, nous allons étudier plus en détail nos algorithmes à l'aide de jeux de données, permettant de voir ainsi s'ils répondent bien à leur objectif, s'ils peuvent éventuellement être améliorés au niveau du temps d'exécution, de l'écriture du code, etc.

2 Étude préliminaire des articles

Ayant relevé une continuité chronologique, dans la mesure où les auteurs reprennent les travaux des précédents articles, notre étude des articles gardera cet ordre chronologique.

Avant d'entrer dans l'étude des algorithmes, il est important de définir les caractéristiques principales de ces graphes. Un arbre, tel que défini dans notre premier article [1], est un graphe plan dans lequel aucune arête ne se croise, chaque nœud du graphe comprend un unique prédécesseur (à l'exception du nœud racine : *Root*), un nœud ne peut se trouver plus proche du *Root* que ses nœuds parents, le dernier nœud d'une branche (nœud qui n'a pas de fils) est appelé feuille. La dernière caractéristique de base d'un arbre est l'alignement de ses nœuds, les nœuds d'une même hauteur doivent être sur la même ligne de sorte que toutes les hauteurs soient sur des lignes parallèles (cette dernière propriété est noté dans l'article [1] : *Esthétique 1*).

Maintenant que notre arbre de base est défini, nous pouvons nous intéresser aux problématiques des articles concernant l'implémentation d'algorithmes (voir ci-dessous nos propriétés de base appliquées à un arbre).

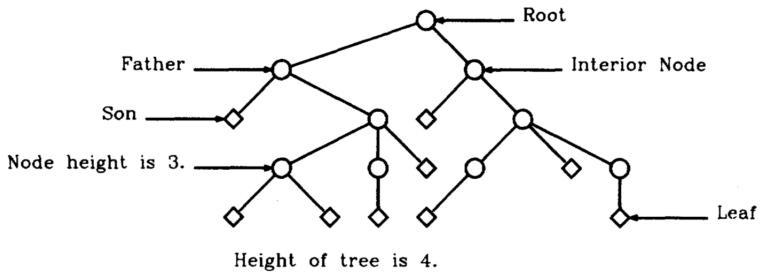

FIGURE 1 – Exemple d'un arbre d'après nos premières propriétés. [1]

2.1 *Tidy Drawings of Trees*

Les auteurs de l'article se sont penchés sur l'optimisation de dessin d'arbre. Dans un soucis d'espace sur lequel représenter les graphes, ils ont posé la problématique suivante (*Limite Physique 1*) : la largeur de l'arbre doit être la plus petite possible (la hauteur est fixée par l'arbre). Pour rappel, la hauteur d'un nœud correspond au nombre de branches entre lui et le *Root* (cf : Figure 1 page 3).

2.1.1 A Naive Tree Drawer

Le premier algorithme (*Algorithme 1*) de dessin d'arbre décrit par cet article est *l'algorithme d'arbre naïf*. Il a comme objectif de répondre à la *Limite Physique 1* en optimisant l'espace du graphe.

Il prend en entrée un arbre, tel que nous l'avons défini plus haut, ainsi que la hauteur de cette arbre et retourne en sortie le même arbre dans lequel la position des nœuds a changé pour faire en sorte que l'arbre ait la largeur la plus étroite possible.

Le fonctionnement de cet algorithme comprend une variable qui compte la prochaine coordonnée en x de libre, pour y placer le prochain nœud. Pour ce qui est de la coordonnée en y, elle correspond à la hauteur du nœud dans l'arbre qui a été donnée en paramètre d'entrée de l'algorithme, de cette manière, on respecte toujours l'*Esthétique 1.*

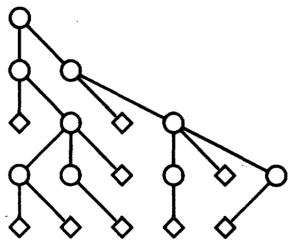

FIGURE 2 – Notre premier arbre modifié par l'*Algorithme 1.* [1]

2.1.2 Binary Tree Drawings

L'algorithme que nous venons de voir est efficace, mais un problème se présente lorsque nous souhaitons faire apparaitre des labels sur les nœuds. Nous en venons maintenant à un algorithme qui nous permet de prendre en compte l'affichage de label sur notre graphe. Il a été conçu pour les arbres binaires (arbres avec deux fils maximum pour un nœud).

Les auteurs de l'article [1] font appel à une nouvelle propriété *Esthétique 2* à respecter pour ces arbres binaires : les fils de gauche doivent être placés à gauche de leur parent.

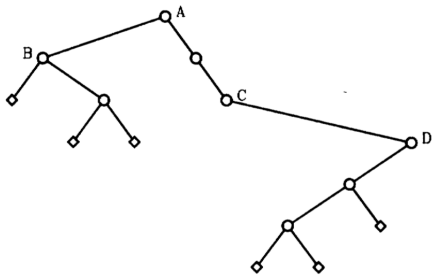

FIGURE 3 – Arbre binaire dessiné par l'*Algorithme de Knuth.* [1]

L'arbre binaire donné par ce nouvel algorithme respecte bien les contraintes de l'*Esthétique 1* ainsi que de l'*Esthétique 2*, mais notre *Limite physique 1* n'est pas respectée. L'algorithme fait en sorte que lorsqu'un nœud occupe une colonne, nul autre nœud ne peut occuper cette colonne. Nous obtenons donc une largeur proportionelle au nombre total de nœuds dans l'arbre.

Il est alors nécessaire d'utiliser un algorithme différent pour que les conditions de la *Limite Physique* soient respectées.

2.1.3 Drawings Satisfying the Physical Limit

Nous avons vu que l'*Algorithme 1* et l'*Algorithme de Knuth* satisfaisaient chacun des contraintes *Esthétique*. L'idée est ici de fusionner les deux algorithmes pour obtenir un algorithme qui respecte à la fois l'*Esthétique 1*, l'*Esthétique 2* et la *Limite Physique*. L'*Algorithme 3* qui résulte de cette fusion donne des meilleurs arbres que l'*Algorithme 2*, mais ne respecte pas la *Limite Physique* dans toutes les situations.

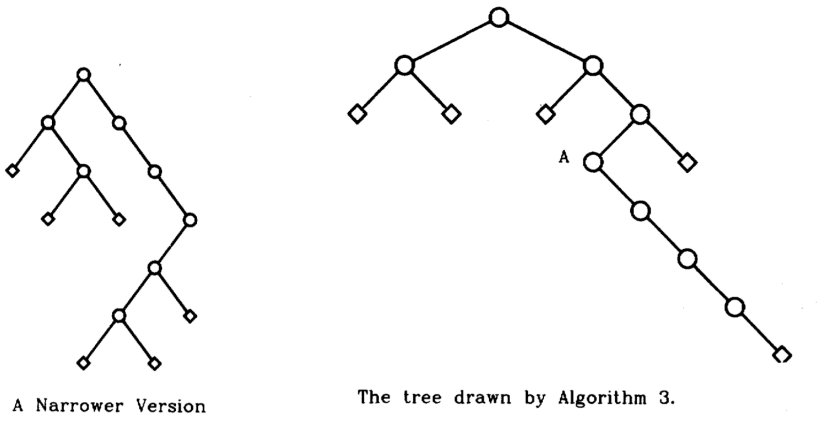

FIGURE 4 – À gauche l'arbre binaire vu précédemment à la Figure 3 ; à droite un autre arbre en sortie du même algorithme. [1]

L'arbre binaire que nous avons vu à la suite de l'*algorithme de Knuth* est à présent optimal en largeur. Il respecte les contraintes de la *Limite Physique*. Nous constatons que l'arbre qui est à droite, par contre, pourrait être plus optimisé. Il résulte de ce contre-exemple que l'*Algorithme 3* doit être modifié afin d'obtenir de meilleurs résultats.

L'*Algorithme 3* viole en effet la *Limite Physique* car il essaye d'introduire une propriété plus forte de l'*Esthétique 2* : les parents doivent être centrés par rapport à leurs fils. Cette *Esthétique 3* inclue les fils directs, mais aussi les fils des fils.

Face à ce problème, les auteurs de l'article [1] propose une version modifiée de l'*Algorithme WS*, où l'on privilégie la contrainte de *Limite Physique* plutôt que cette contrainte *Esthétique 3*.

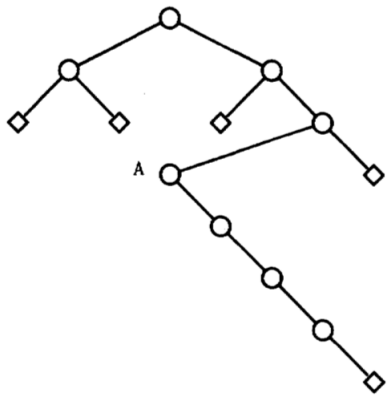

FIGURE 5 – Arbre de droite de la Figure 4 après l'*Algorithme WS modifié*. [1]

L'*Algorithme 3* présenté ne concerne que les arbres binaires mais peut aussi être adapté pour d'autres types d'arbres, en choisissant un moyen de placer les nœuds parents au-dessus de leurs enfants en tenant compte de leurs nombres ou d'autres spécificités de l'arbre.

2.2 *Tidier Drawings of Trees*

L'article [2] reprend les travaux que nous venons de voir. Nous retrouvons donc les trois différentes *Esthétiques* ainsi que la problématique commune de satisfaire à la fois les trois *Esthétiques* et la *Limite Physique*.

Les auteurs relèvent l'insuffisance de l'*Algorithme WS* pour réaliser des arbres de largeur optimale. Ils notent également la proposition de l'*Algorithme WS modifié* afin de la respecter au détriment de la contrainte *Esthétique 3*. Cependant, ils affirment que les graphes obtenus peuvent être à la fois plus étroits et meilleurs d'un point de vue esthétique (comme nous pouvons le voir à la figure 6).

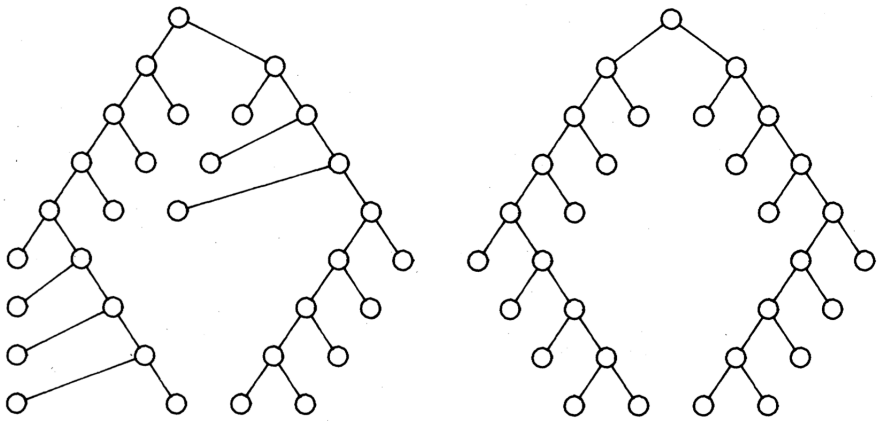

FIGURE 6 – À gauche l'arbre binaire avec l'*Algorithme WS modifié* ; à droite un arbre plus optimal. [2]

Pour tenter de parvenir à cette optimisation, une nouvelle contrainte *Esthétique 4* est posée : un arbre et son image miroir doivent être le reflet l'un de l'autre. De plus, un sous-arbre doit être dessiné de la même manière quelle que soit sa position au sein de l'arbre global.

Il est clairement visible que l'arbre en sortie de l'*Algorithme WS modifié* vu à la Figure 6 ne respecte pas cette nouvelle contrainte. En la respectant, il devrait ressembler à l'arbre de droite que nous venons de voir sur la même figure.

L'*Algorithme WS modifié* ne peut en effet pas respecter cette contrainte car la forme d'un sous-arbre est modifié en fonction de la position des nodes qui lui sont extérieurs, on peut donc obtenir des sous-arbres à la base symétriques qui

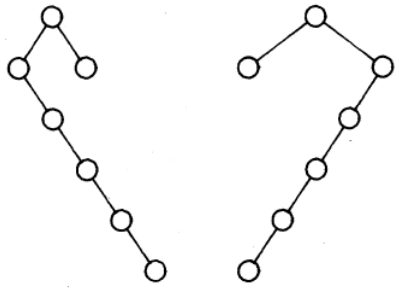

FIGURE 7 – Arbre et son miroir avec l'*Algorithme WS modifié*. [2]

sont dessinés assymétriquement.

Afin de respecter l'*Esthétique 4*, il est nécessaire de sacrifier de la largeur, mais les auteurs considèrent que l'*Esthétique 4* est prioritaire par rapport à la *Limite Physique* pour faciliter la perception humaine.

L'idée de l'algorithme proposé, l'*Algorithme TR*, est de générer les sous-arbres d'un même nœud de manière séparée, puis de les placer les plus proches possible. Ceci est d'abord réalisé par un parcours postfixe, en superposant les arbres et en les déplaçant jusqu'à ce que les arbres ne se chevauchent à aucun niveau de profondeur, avec une distance minimale incompressible définie. Une fois ces positions connues, il suffit de faire un parcours préfixe de l'arbre, en donnant des positions absolues aux nœuds.

Cet algorithme respecte à la fois les contraintes d'*Esthétique 1*, *Esthétique 2*, *Esthétique 3*, mais en plus l'*Esthétique 4* définie précédemment.

2.2.1 Generalization to m-ary trees and forests

L'*Algorithme TR* peut facilement être adapté pour des arbres n-aires, sans en affecter les performances, en effectuant n-1 opérations de séparation des sous-arbres. Pour les forêts, cela dépend de la représentation utilisée, mais reste possible et est détaillé dans un autre article.

2.3 *A Node-Positioning Algorithm for General Trees*

L'article [3] s'intéresse au cas général des arbres n-aires, en gardant comme objectif le respect des contraintes définies dans les précédents articles, sauf l'*Esthétique 2*, qui n'est pas pertinente dans les cas où les nœuds ont plus de deux fils. L'*Esthétique 4*, définie plus tôt, est d'ailleurs renforcée en imposant également que des petits sous-arbres ne soient pas arbitraitement positionnés lorsque proches d'autres sous-arbres plus grands. Ainsi, des petits sous-arbres aux extrêmes gauche et droite devraient être placés de manière adjacente à leurs sous-arbres voisins plus grands.

Il est admis qu'on ne cherche que la position en x des nœuds, la position en y étant déterminée par la hauteur dans l'arbre.

L'algorithme proposé réalise deux parcours de l'arbre. Premièrement, on parcourt l'arbre en assignant une valeur en x temporaire à chaque nœud ainsi qu'en remplissant un champ particulier qui servira à décaler ses noeuds au second parcours. Ce premier parcours est réalisé de la même manière que dans l'article [2], avec un parcours postfixe qui décale les sous-arbres des nœuds de manière à respecter à tous les niveaux de profondeur un écartement défini. On peut noter qu'il n'y a pas de mécanisme pour rapprocher les sous-arbres alors qu'on pourrait à une itération ne pas avoir à écarter les sous-arbres, dans le cas d'un sous-arbre gauche très écarté vers la gauche par exemple, mais que l'on ne rapproche pas pour autant, alors qu'il pourrait être très éloigné d'un grand sous-arbre droit. Cet agglutinement est d'ailleurs selon les auteurs le défaut des algorithmes précédents (articles [1], et [2]). La correction proposée est d'utiliser un modificateur de position pour déplacer les sous-arbres, et dans ce cas précis les rapprocher lorsque c'est possible.

Le second parcours permet d'attribuer la coordonnée en x définitive grâce à ce modificateur présent sur les nœuds.

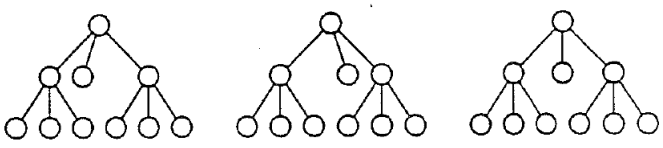

FIGURE 8 – Arbres montrant l'effet observé d'agglutinement à gauche, à droite, puis le résultat idéal. [3]

2.4 *Improving Walker's Algorithm to Run in Linear Time*

Dans l'article [4], l'auteur reprend les algorithmes de dessin de graphes, en rappelant la complexité linéaire de l'exécution des algorithmes *Algorithme WS*, *Algorithme TR*. Cependant, contrairement à ce qui est annoncé par Walker dans son article [3], la complexité de son algorithme est quadratique et non linéaire. Aussi, l'auteur propose une amélioration à l'*Algorithme de Walker* pour réduire sa complexité algorithmique et garder une exécution en temps linéaire en nombre de nœuds.

L'*Algorithme de Walker* permet en effet de régler un phénomène apparu avec l'*Algorithme TR*, qui est le placement à gauche de sous-arbres plus petits que leurs voisins, qui ne sont donc pas centrés alors que de la place est disponible.

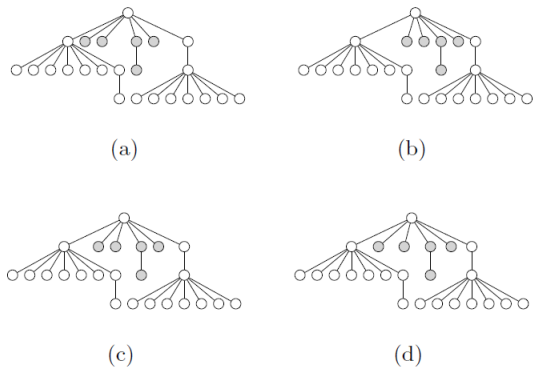

FIGURE 9 – Arbres dénotant l'effet de décalage à gauche (a), à droite (b), la moyenne des deux (c) et la version où les sous-arbres sont espacés équitablement (d). [4]

Certaines parties de l'*Algorithme de Walker* sont cependant non-linéaires, et concernent la procédure de répartition, qui s'occupe d'attribuer un décalage aux sous-arbres, mais ces parties sont présentées par l'auteur comme améliorables, soit par des idées simples déjà utilisées auparavant, soit par de nouvelles idées.

Pour améliorer l'*Algorithme de Walker*, plusieurs traitements sont présentés, comme l'ajout de nouvelles informations sur les nœuds pour accélérer la recherche des nœuds ancêtres afin d'appliquer le modificateur de position, ou en limitant le décalage de certains sous-arbres seulement en fin d'algorithme.

3 Implémentation des algorithmes

Suite à la lecture et à l'analyse des articles, nous avons voulu implémenter les algorithmes que nous venons de brièvement présenter. Il va sans dire que nous n'allions pas implémenter la totalité des algorithmes, mais seulement les deux qui nous semblent *clés* dans l'évolution chronologique de l'optimisation d'arbres, c'est à dire celui de *Walker* et son optimisation.

Implémenter l'algorithme de Walker en Python avec l'interface Tulip n'était pas chose simple car il nous a fallu comprendre les algorithmes présentés tantôt en *Pascal*, tantôt en *Pseudo-code*. Le fait que les articles dataient de quelques années (voir de quelques décennies pour certains) nous a posé problème. En effet, nous nous sommes interrogés sur la logique algorithmique du code de l'article [3], celui-ci présentant des suites d'instructions illogiques au premier abord. De plus, la clarté des paragraphes explicatifs laissant à désirer, la tâche de compréhension et retranscription de certaines fonctions n'a pas été facile. Nous avons également constaté que certaines fonctions (et non les plus triviales) n'étaient pas définies bien qu'utilisées dans les algorithmes.

À cause de tous ces facteurs, il nous a fallu tenter de corriger ce qui semblait être des erreurs de la part de l'auteur du rapport, comme des *utilisations de variables non déclarées*, des *boucles qui ne sont jamais invoquées ou ne bouclent qu'une unique fois*, des *variables non utilisées*, des *affectations de valeur illégales*, le tout en bataillant contre les *plantages de Tulip* à chaque chargement du script.

3.1 Implémentation *Python* de l'*Algorithme de Walker* [3]

Notre tentative d'implémentation de l'algorithme de *Walker* en Python peut être trouvée dans la même archive que ce rapport, dans le fichier *walker.py*. Les dernières erreurs que nous n'avons pas réussi à corriger se situent dans la fonction *apportion*, qui est justement celle censée différencier l'algorithme de ceux des articles précédents. C'est cette même fonction qui, d'après le 4e article, fait exploser la complexité de l'algorithme, qui n'est alors plus linéaire.

C'est dans cette fonction que nous avons trouvé le plus d'erreurs logiques ou de syntaxe, et c'est donc cette fonction qui nous a demandé le plus d'expérimentation et le plus d'improvisation, pour en arriver à une version qui fonctionne sur certains arbres aléatoires et ne fonctionne pas sur d'autres. De plus, même si l'algorithme s'exécute sans erreur, certaines edges du graphe s'entrecroisent toujours, car les sous-arbres ne sont pas proprement espacés. L'erreur qui nous semble la plus flagrante a été signalée dans le code par un commentaire, et est présentée ci-dessous.

On peut voir dans la figure 10 que lorsque l'on sort de la première boucle while, si cette boucle a été appellée au moins une fois (ce qui a toujours été le

```
if MoveDistance > 0 then
    begin
        (* Count interior sibling subtrees in LeftSiblings*)
        TempPtr ← Node;
        LeftSiblings ← 0;
        while (TempPtr ≠ ϕ and
               TempPtr ≠ AncestorNeighbor) do
            begin
                LeftSiblings ← LeftSiblings + 1;
                TempPtr ← LEFTSIBLING(TempPtr);
            end;


    if TempPtr ≠ ϕ then
        (* Apply portions to appropriate leftsibling  *)
        (* subtrees.                                  *)
        begin
            Portion ← MoveDistance / LeftSiblings;
            TempPtr ← Node;
            while
                TempPtr = AncestorNeighbor do
                begin
                    PRELIM(TempPtr) ←
                        PRELIM(TempPtr) + MoveDistance;
                    MODIFIER(TempPtr) ←
                        MODIFIER(TempPtr) + MoveDistance;
                    MoveDistance ← MoveDistance -
                        Portion;
                    TempPtr ← LEFTSIBLING(TempPtr);
                end;
        end;
```

FIGURE 10 – Captures d'écran d'une des erreurs dans la logique de l'algorithme, extraite de la fonction *apportion*.

cas en pratique), on a soit TempPtr vide ou égal à AncestorNeighbor. Dans ce dernier cas, on rentre ensuite dans le bloc *if* suivant, où l'on réinitialise la valeur de TempPtr (ayant été modifiée par la boucle, sinon il y aurait division par 0 à la définition de Portion), avant de tenter de boucler sur la condition *TempPtr = AncestorNeighbor*, qui n'est donc jamais réalisable. Encore une fois, même si nous avions *Node = TempPtr = AncestorNeighbor*, il y aurait une erreur puisque la première boucle n'incrémenterait pas la valeur de LeftSiblings, ce qui provoquerait une division par 0.

3.2 Comparaisons des résultats

Sont présentés dans la figure 11 deux exemples d'arbres ayant été déssinés par notre implémentation de l'algorithme de Walker et par le plugin Walker propre à Tulip. Les images de gauche sont les résultats de notre propre implémentation, celles de droite montrent les résultats de l'algorithme natif.

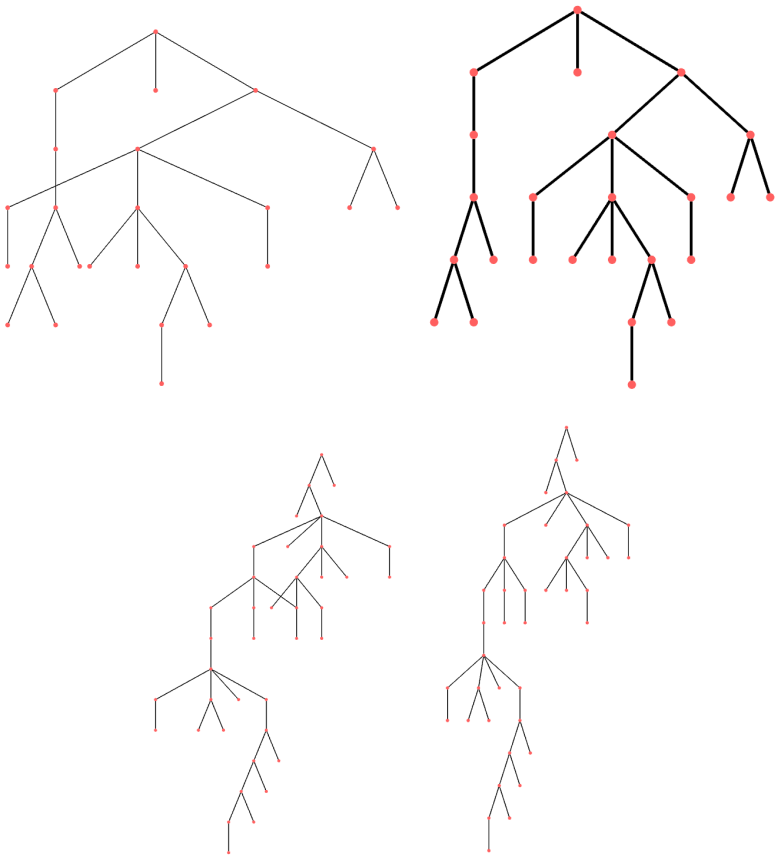

FIGURE 11 – À gauche l'arbre binaire avec *notre implémentation* ; à droite l'implémentation Tulip native.

3.3 Implémentation *Python* de l'*Algorithme de Walker* amélioré [4]

N'ayant pas réussi à implémenter complètement l'algorithme de *Walker*, il nous a été impossible de l'utiliser pour programmer l'amélioration présentée dans le 4ième article sans repartir de 0, ni de la comparer ensuite à l'algorithme de Walker amélioré déjà implémenté nativement dans Tulip.

Toute étude comparative empirique de ces deux algorithmes est donc malheureusement impossible dans l'état actuel de notre implémentation, et le restera tant que des tests plus élaborés ne seront pas réalisés pour tenter de corriger les nombreuses erreurs présentes dans le document.

4 Conclusion

Les quatre algorithmes montrent une évolution temporelle intéressante, chaque algorithme reprenant le précédent en l'améliorant. L'*Algorithme WS* est un algorithme basique, permettant de définir des critères *Esthétiques* à respecter et propose deux algorithmes qui ne remplissent pas tous les critères en même temps. L'*Algorithme TR* reprend l'*Algorithme WS* en ajoutant une contrainte de symétrie des sous-arbres qui n'était pas respectée à cause de l'assignation des coordonnées de gauche à droite et qui dépendait de la position de nœuds extérieurs aux sous-arbres. L'*Algorithme de Walker* reprend l'*Algorithme TR* en généralisant aux arbres n-aires et en corrigeant un phénomène d'agglutinement qui pouvait se produire lorsqu'un grand sous-arbre était ajoutée à droite d'un plus petit. L'*Algorithme de Walker amélioré* reprend l' *Algorithme de Walker* en diminuant sa complexité algorithmique pour qu'elle redevienne linéaire, comme les précédents algorithmes. Cependant, malgré nos efforts, l'impossibilité de reproduire dans un environnement de test l'algorithme de Walker nous a empêché de pousser plus loin notre étude.

Références

[1] C. Wetherell and A. Shannon. Tidy Drawings of Trees. *IEEE transactions on Software Engineering* SE-5, 5 (septembre 1979) p514-520.

[2] E. Reingold and J. Tilford. Tidier Drawings of Trees. *IEEE Transactions on Software Engineering* SE-7, 2 (mars 1981) p223-228.

[3] J. Walker II. A Node-Positioning Algorithm for General Trees. *Software – Practice and Experience* SE-20, 2 (1990) p685–705.

[4] C. Buchheim, M. Jünger and S. Leipert. Improving Walker's Algorithm to Run in Linear Time. Technical Report zaik2002-431, ZAIK, Universität zu Köln, (2002) p344-352.