Visualisation d'arbres de grandes tailles [1]
Rapport de PSTL

Érika Baëna
erika.baena@etu.upmc.fr

Diana Malabard
diana.malabard@etu.upmc.fr

Antoine Genitrini (encadrant)
antoine.genitrini@lip6.fr

Université Pierre et Marie Curie
15 mai 2014

1. Disponible sur: `https://github.com/BErika/PSTL_TreeDisplay`

Table des matières

Résumé

Des outils existent actuellement pour représenter des arbres de grande taille de façon efficace. Citons par exemple GraphViz. L'inconvénient d'un tel outil est qu'il ne prend pas en compte l'ordre des fils de manière triviale. Ceci pose problème lorsque l'on souhaite représenter des arbres dont l'ordre des fils est primordial : les arbres de recherche par exemple.

Ce projet consiste à fournir une alternative à Graphviz, afin de pouvoir visualiser n'importe quel type d'arbre, toujours de manière efficace, mais en conservant l'ordre des fils. Ce problème possède plusieurs problématiques. Tout d'abord, nous voulons que l'affichage d'un arbre soit faite de manière élégante. Ensuite, il faut que le calcul de la mise en page de l'arbre soit rapide.

Pour ce faire, nous avons donc dû étudier les algorithmes déjà existants pour la mise en page élégante des arbres de grande taille. Sachant ces algorithmes, nous avons conçu un algorithme permettant cette mise en page. Enfin, nous avons implémenté cet algorithme, de telle façon qu'il puisse être utilisé avec différentes sorties. Nous avons ici choisi de considérer trois sorties possibles : Tikz, pour pouvoir intégrer le code à un document LaTeX ; Asymptote, une alternative à TikZ ; et NetworkX, pour pouvoir générer automatiquement un pdf ou une image de l'arbre que l'on pourra ultérieurement insérer dans n'importe quel document.

Chapitre 1

État des lieux

Les articles que nous avons étudiés concernaient avant tout la visualisation des arbres binaires de grande taille sans prendre en compte l'ordre des fils. Il en ressort néanmoins plusieurs principes :

1. Les arêtes de l'arbre ne doivent pas s'intersecter.
2. Les nœuds de même profondeur doivent être dessinés sur la même ligne horizontale.
3. Les arbres doivent être dessinés de la manière la plus compacte possible.
4. Un nœud parent doit être centré par rapport à ses fils.
5. Un sous-arbre doit être dessiné de la même façon, peu importe où il est placé dans l'arbre.
6. Les nœuds fils d'un nœud père doivent être espacés de manière homogène.

Un des enjeux soulevés par l'ensemble de ces principes est que les nœuds ne doivent pas se chevaucher, tout comme les arbres en eux-même.

Il existe plusieurs algorithmes permettant de dessiner des arbres de grande taille, mais tous ne respectent pas tous les principes décrits ci-dessus.

Par exemple, l'algorithme le plus simple, l'algorithme de Knuth [2], ne respecte que les deux premiers principes. Cet algorithme décrit déjà une idée de "slot disponible". Mais ce sont Charles Wetherell et Alfred Shannon [6], en 1979, qui introduiront l'utilisation d'un tableau qui associera à chaque profondeur le prochain slot disponible. Ils arrivent ainsi à respecter tous les principes, à l'exception des principes 4 et 5. Ils introduisent malgré tout également l'idée de parcourir l'arbre de bas en haut, plutôt que l'inverse, ceci afin de centrer facilement un père selon ses fils.

Le principal problème est alors : Comment respecter tous ces principes et traiter le chevauchement d'arbres sans perdre en complexité ? C'est l'algorithme The Mods and the Rockers qui répondra à cette question. Au lieu de reparcourir les sous-arbres pour les décaler afin d'éviter tout chevauchement, on raisonne en deux passes de l'arbre. Lors de la première passe, on mémorise un *modifier* qui indiquera le décallage qui devra être appliqué sur chaque sous-arbre lors de la deuxième passe.

Il existe enfin des algorithmes dont le but est surtout d'optimiser les concepts des algorithmes existants. Citons le concept de contour d'arbre, qui permet de ne parcourir que le contour et donc de ne pas rentrer au cœur de l'arbre. Ceci est un gain conséquent puisque nous traitons ici des arbres de grande taille, et donc potentiellement très larges.

Nous allons donc voir à présent comment nous nous sommes inspirées de ces algorithmes afin de généraliser les concepts aux arbres n-aires, et en conservant l'ordre des fils.

Chapitre 2

Étude préliminaire

Le but de cette étude préliminaire est de trouver un outil adapté à la représentation d'arbres de grande taille. Étudions les performances de TikZ, Asymptote et NetworkX pour la génération d'un cas particulier d'arbres : les chaînes.

2.1 TikZ

TikZ est un package LaTeX permettant la création de graphiques.

On va utiliser le code Python suivant pour générer du code TikZ décrivant un arbre linéaire d'une taille passée en paramètre.

```
import sys

nbIte = int(sys.argv[1])
if (sys.argv[2] == "true"):
  labels = True
else:
  labels = False
i = 0

fileName = "testTikz%d" % (nbIte,)

if (labels):
  fileName += "A.tex"
else:
  fileName += "B.tex"

fichierTest = open (fileName, "w")

if (labels):
  fichierTest.write("\\node (a%d) at (0,%d) {$%d$};\n" % (i, i, i,))
  i += 1
  while i<nbIte:
    fichierTest.write ("\\node (a%d) at (0,%d) {$%d$};\n" % (i, i, i,))
    fichierTest.write ("\\draw (a%d) -- (a%d);\n" % (i-1, i,))
    i += 1
else:
  i += 1
  while i<nbIte:
    fichierTest.write ("\\draw (0,%d) -- (0,%d);\n" % (i-1, i,))
    i += 1

fichierTest.close()
```

2.1.1 500 nœuds

On utilise le code précédent pour générer un arbre linéaire de taille 500. On insère le code obtenu dans un fichier LaTeX pour voir le résultat. Le fichier LaTeX compile et le résultat est aux figures A.1 et A.2.

2.1.2 10000 nœuds

On utilise maintenant le même code mais pour avoir un arbre de 10000 nœuds. Le fichier LaTeX ne compile plus. On obtient l'erreur `dimension too large` à la ligne :

```
\node (a576) at (0,576) {$576$};
```

Si l'arbre est trop grand, essayons de réduire sa taille : on diminue l'échelle, on diminue la distance entre deux points et on supprime les labels. L'erreur persiste au même endroit. TikZ limite notre arbre à 575 nœuds à la verticale. Peut-être la limite serait-elle différente si les nœuds était répartis autrement.

2.2 Asymptote

Asymptote est un langage de description de dessins vectoriels. Un package permet de le compiler dans un fichier LaTeX mais le code asymptote peut aussi être autonome.

On va utiliser le code Python suivant pour générer du code Asymptote décrivant un arbre linéaire d'une taille passée en paramètre.

```
import sys

nbIte = int(sys.argv[1])
if (sys.argv[2] == "true"):
  labels = True
else:
  labels = False

i = 0

fileName = "testAsymptote%d" % (nbIte,)
if (labels):
  fileName += "A.tex"
else:
  fileName += "B.tex"

fichierTest = open (fileName, "w")
fichierTest.write("\\begin{asy}\n")
fichierTest.write("size(20cm,20cm);\n")

if (labels):
  fichierTest.write("label(\"a%d\", (0, %d), E);\n" % (i, i,))

i += 1

while i<nbIte:
  if (labels):
    fichierTest.write ("label(\"a%d\", (0, %d), E);\n" % (i, i,))
  fichierTest.write ("draw((0, %d) -- (0, %d));\n" % ((i-1), i,))
  i += 1

fichierTest.write("\\end{asy}\n")

fichierTest.close()
```

2.2.1 500 nœuds

On commence doucement en générant un arbre de 500 nœuds. On insère le code obtenu dans un fichier LaTeX comme précédemment. Le fichier compile et le résultat obtenu est celui des figures A.3 et A.4.

2.2.2 10000 nœuds

On recommence en mettant la barre à 10000 nœuds. Le fichier compile sans problème et le résultat est celui des figures A.5 et A.6.

2.3 NetworkX accompagné de Matplotlib

NetworkX est une bibliothèque Python pour l'étude des graphes, conçue pour fonctionner sur des grands graphes.

Matplotlib est aussi une bibliothèque Python mais qui permet quant à elle de générer une image 2D dans différents formats de sortie possible comme par exemple un png, un pdf ou un svg.

On va utiliser le code Python suivant pour générer du code NetworkX décrivant un arbre linéaire d'une taille passée en paramètre.

```
import matplotlib.pyplot as plt
import networkx as nx
import sys

nbIte = int(sys.argv[1])
longueur_arete = float(sys.argv[2])
if (sys.argv[3] == "true"):
  labels = True
else:
  labels = False
filename="testNetworkx%d" % (nbIte,)
if (labels):
  filename += "A.png"
else:
  filename += "B.png"
G = nx.path_graph(nbIte)
pos={x: (5, x*longueur_arete) for x in G.nodes()}
if (labels):
  labels={x: "label" for x in G.nodes()}
  nx.draw(G, pos, labels=labels, node_size=5, with_labels=True)
else:
  nx.draw(G, pos, node_size=5, with_labels=False)
plt.savefig(filename)
```

2.3.1 500 nœuds

De même que précédemment, on génère d'abord un arbre de 500 nœuds. On choisi le format de sortie png. Le résultat est visible aux figures A.7 et A.8.

2.3.2 10000 nœuds

Passons maintenant à 10000 nœuds. Le résultat est aux figures A.9 et A.10.

2.4 Conclusion

TikZ permet une représentation claire d'un arbre avec ses labels. En effet, même avec 500 nœuds, les labels sont lisibles si on zoome suffisamment. Cependant, une limite a rapidement été atteinte. TikZ serait préférable pour la représentation de petits arbres avec (ou sans !) labels.

Asymptote permet de représenter de grands arbres. Son point faible est la représentation des labels. Cependant, les labels sont compilés avec LaTeX ce qui peut permettre d'avoir des labels un peu plus évolués qu'une chaîne de caractères, une fois la taille des labels maîtrisée.

NetworkX et Matplotlib permettent plusieurs formats de sortie différents. Cela pourrait être utile aux non-utilisateurs de LaTeX. Cependant, l'affichage des labels n'est pas non plus très optimal.

Notons tout de même que cette étude préliminaire ne prend pas en compte le temps de calcul des coordonnées, ce qui est le cœur de notre projet et qui est expliqué dans le chapitre suivant.

Chapitre 3

Choix d'implémentation

3.1 Fonctionnement général

L'utilisateur fourni un fichier et précise son type. Il peut aussi choisir d'afficher des labels sur les nœuds ainsi que le format de sortie souhaité. Par défaut, l'application génère un png sans labels. L'application fonctionne ensuite en trois étapes :

1. Parsing du fichier d'entrée selon le type indiqué pour obtenir une représentation d'arbre selon notre structure interne.
2. Calcul des coordonnées de chaque nœud.
3. Génération d'une image selon le type de sortie choisie.

```
erika@erika-K53SD:~/Documents/Cours/M1S2/PSTL/Implementation$ python3 treeDisplay.py -h
usage: treeDisplay.py [-h] [-L] [-O {tikz,asy,png,pdf,eps}] [-N NAME]
                      {str,arb,xml,dot} src

positional arguments:
  {str,arb,xml,dot}     The type of the given file.
  src                   The file which contains a tree description.

optional arguments:
  -h, --help            show this help message and exit
  -L, --labels          Print labels on the output tree, depreciate with
                        NetworkX and Asymptote
  -O {tikz,asy,png,pdf,eps}, --output {tikz,asy,png,pdf,eps}
                        The type of the output file
  -N NAME, --name NAME  The name of the output file
erika@erika-K53SD:~/Documents/Cours/M1S2/PSTL/Implementation$
```

3.2 Plus en détails

3.2.1 Structure d'arbre

Nous avons une structure d'arbre n-aire classique avec une liste pour les enfants ainsi que des attributs pour le calcul de coordonnées et la génération finale. En particulier, les attributs `width` et `height` représente respectivement la plus grande abscisse et la plus grande ordonnée rencontrée dans le sous-arbre. Au niveau de la racine, cela représente la largeur et la hauteur du graphe.

```
class Tree(object):
  "Local representation of tree"

  def __init__(self, x = -1, depth=0, label="", children=None, offset = 0, isRoot
    = False):
    self.x = x
    self.y = depth
    self.label = label
    self.offset = offset
    self.height = None
    self.width = None
    if children is None:
      self.children = list()
    else:
      self.children = children
```

3.2.2 Fonctionnement des algorithmes de parsing

Mots bien parenthésés

Le parser de mots bien parenthésés (cf. B.1) respecte la grammaire suivante :

```
ARBRE ::= '(' LABEL NOEUDS ')'
NOEUDS ::= ARBRE NOEUDS | ϵ
LABEL ::= [a-zA-Z1-9]* | ϵ
```

Compte-tenu du fait qu'on veut pouvoir travailler sur des arbres de grande taille, le parser bufferise le fichier d'entrée pour ne pas saturer la mémoire en chargeant tout le fichier dans une variable.

Dot

Le parser dot (cf. B.2) parse une sous-partie du langage dot, à savoir :

```
DOT ::= STRICT GRAPH ID '' SEQINST ''
STRICT ::= strict | ϵ
GRAPH ::= digraph | graph
SEQINST ::= INST ';' SEQINST | ϵ
INST ::= ID '[' label = "LABEL" ']' | ID LINK ID
LINK ::= -- | ->
ID ::= [0-9]*
LABEL ::= [a-zA-Z1-9]* | ϵ
```

XML

Le parser XML (cf. B.3)respecte la grammaire ci-dessous. Comme beaucoup de langages de programmation, Python a une librairie dédiée au XML. Nous l'avons utilisée.

```
XML ::= <?xml version="1.0"?><tree> NOEUDS </tree>
NOEUDS ::= NOEUD NOEUDS | ϵ
NOEUD ::= <node type=TAG id=ID> NOEUDS </noeud> | <leaf type=TAG id=ID />
TAG ::= " [a-zA-Z1-9]* "
ID ::= [0-9]*
```

Remarques

On veut pouvoir utiliser les fichiers générés par `Arbogen` [4] qui est un générateur d'arbre. Il nous permet de générer des arbres de grandes tailles pour nos tests. C'est en analysant les fichiers fournis par ce générateur que nous avons établi les différences d'interprétations entre `LABEL`, `ID` et `TAG`.

Un `LABEL` est une étiquette quelconque sur un nœud, qui peut apparaître à différents endroits de l'arbre. Cela nous assure qu'un mot bien parenthésés ne peut représenter qu'un arbre car seul l'arborescence des parenthèses permet de définir les arcs entre nœuds.

Un `ID` est un identifiant unique d'un nœud. On remarque alors que la grammaire Dot peut représenter des arbres mais aussi plus généralement des graphes quelconques car les `ID` sont la seule façon de représenter les arcs entre nœuds.

Un `TAG` correspond en fait à une partie gauche dans la grammaire donnée en entrée à `Arbogen`. Il n'est pas donc unique dans l'arbre et est interprété dans le parser comme un label. Cependant, la grammaire XML comporte aussi des `ID`. Si en plus de considérer l'arborescence des balises XML, on considérait aussi les `ID`, on pourrait aussi représenter des graphes. Ce n'est pas le cas de notre parser XML car on s'intéresse aux arbres.

3.2.3 Fonctionnement de l'algorithme de calcul de coordonnées

On décide que la distance minimale entre deux nœuds est de 1 unité selon l'axe des abscisses et celui des ordonnées.

L'ordonnée d'un nœud est triviale : c'est sa profondeur.

L'abscisse d'un nœud est un peu plus complexe et demande donc plus de réflexion.

On veut qu'un père soit centré sur ses fils. On commence donc par placer ses fils s'il en a puis on fait la moyenne de l'abscisse de ses 2 fils extrémaux pour le centrer.

Si un nœud n'a pas de fils, on le place à 1 de son frère gauche. D'un point de vue de l'architecture, on a donc besoin d'une structure qui, à profondeur `p` mémorise la prochaine place disponible (ou au choix la dernière place utilisée).

Si un nœud a des fils, il se peut qu'en calculant son abscisse en fonction de ses fils on se retrouve avec un nœud qui est trop proche de son frère gauche comme dans l'exemple ci-dessous.

Considérons l'arbre suivant : `(root (n11) (n12) (n13 (n21) (n22) (n23)))`.

	root		n11		n12		n13		n21		n22		n23	
	x	y	x	y	x	y	x	y	x	y	x	y	x	y
Initialement	-1	0	-1	0	-1	0	-1	0	-1	0	-1	0	-1	0
Appel sur root	-1	**0**	-1	0	-1	0	-1	0	-1	0	-1	0	-1	0
Appel sur n11	-1	**0**	**0**	**1**	-1	0	-1	0	-1	0	-1	0	-1	0
Retour sur root	-1	**0**	**0**	**1**	-1	0	-1	0	-1	0	-1	0	-1	0
Appel sur n12	-1	**0**	**0**	**1**	**1**	**1**	-1	0	-1	0	-1	0	-1	0
Retour sur root	-1	**0**	**0**	**1**	**1**	**1**	-1	0	-1	0	-1	0	-1	0
Appel sur n13	-1	**0**	**0**	**1**	**1**	**1**	-1	0	-1	0	-1	0	-1	0
Appel sur n21	-1	**0**	**0**	**1**	**1**	**1**	-1	0	**0**	**2**	-1	0	-1	0
Retour sur n13	-1	**0**	**0**	**1**	**1**	**1**	-1	0	**0**	**2**	-1	0	-1	0
Appel sur n22	-1	**0**	**0**	**1**	**1**	**1**	-1	0	**0**	**2**	**1**	**2**	-1	0
Retour sur n13	-1	**0**	**0**	**1**	**1**	**1**	-1	0	**0**	**2**	**1**	**2**	-1	0
Appel sur n23	-1	**0**	**0**	**1**	**1**	**1**	-1	0	**0**	**2**	**1**	**2**	**2**	**2**
Retour sur n13	-1	**0**	**0**	**1**	**1**	**1**	-1	0	**0**	**2**	**1**	**2**	**2**	**2**

La position ensuite calculée pour `n13` est 1. Or il y a déjà `n12` à cet endroit. La première place disponible à profondeur 1 est 2. On va donc devoir effectuer un décalage de 1.

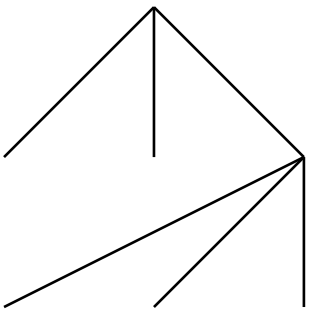


FIGURE 3.1 – Résultat de l'exemple à la fin de l'étape 1.

On compare la position calculée avec la première position disponible et on prend le maximum. Si la position calculée n'est pas celle retenue, le père n'est plus centré au milieu de ses fils. On mémorise donc le décalage effectué pour ce père pour ensuite l'appliquer à ses sous-arbres dans un second temps pour des raisons de complexité. En effet, si on faisait chaque décalage lorsqu'il se présentait, le décalage serait quadratique car il y a potentiellement un décalage pour chaque ancêtre d'un nœuds.

```
def setup (self, depth=0, nexts=None, offset=None):
  if nexts is None:
    nexts = defaultdict(lambda:0)
  if offset is None:
    offset = defaultdict(lambda:0)

  # L'ordonnée est triviale, c'est la profondeur.
  self.y = depth

  # On calcule d'abord les coordonnées des enfants.
  for c in self.children:
    c.setup(depth+1, nexts, offset)

  # On centre le noeud au milieu de ses enfants.
  nbChildren = len(self.children)
  if (nbChildren == 0):
    place = nexts[depth]
    self.x = place
  else:
    place = (self.children[0].x + self.children[nbChildren-1].x) / 2

  # On calcule l'éventuel décalage engendré.
  offset[depth] = max(offset[depth], nexts[depth]-place)

  # On applique le décalage de la profondeur.
  if (nbChildren != 0):
    self.x = place + offset[depth]

  # On met é jour la prochaine place disponible é cette profondeur.
  nexts[depth] = self.x +1
```

Il faut ensuite appliquer récursivement les décalages calculés lors de la première passe. On calcule à la même occasion la hauteur et la largeur de l'image pour pouvoir connaître le ratio de l'image lors de la génération.

```
def addOffsets (self, offsum=0):
  self.x = self.x + offsum
  offsum = offsum + self.offset

  self.height = self.y
  self.width = self.x

  for c in self.children:
    c.addOffsets(offsum)
    self.height = max (self.height, c.height)
    self.width = max (self.width, c.width)
```

Le résultat final est le suivant :

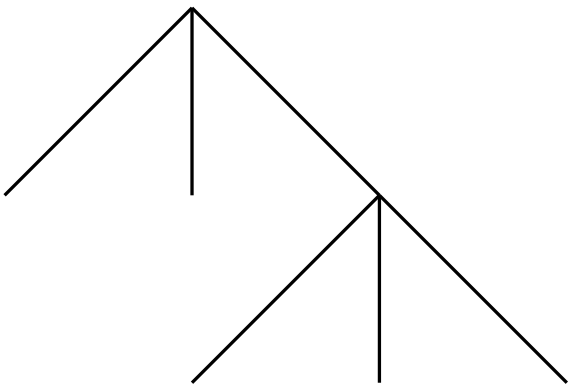


FIGURE 3.2 – Résultat de l'exemple à la fin de l'étape 2.

3.2.4 Fonctionnement de la génération du code

TikZ

```
from tree import *

def toTikz (t, withLabels = False, fileName = "treeTikz.tex"):
  "Generate an output file which name is fileName with the coordonates of the tree
    t. Node labels are displayed if withLabels is True"
  f = open (fileName, "w")
  if (withLabels):
    toTikzRecWith (t, f)
  else:
    toTikzRecWithout (t, f)
  f.close()


def toTikzRecWith (t, f):
  f.write("\\node (a%d) at (%f, -%f) {%s};\n" % (id(t), t.x, t.y, t.label,))
  for c in t.children:
    toTikzRecWith (c, f)
    f.write("\\draw (a%d) -- (a%d);\n" % (id(t), id(c),))


def toTikzRecWithout (t, f):
  for c in t.children:
    toTikzRecWithout (c, f)
    f.write("\\draw (%f, -%f) -- (%f, -%f);\n" % (t.x, t.y, c.x, c.y,))
```

Asymptote

```
from tree import *

def toAsymptote (t, withLabels=False, fileName="treeAsy.tex"):
  f = open (fileName, "w")

  f.write("\\begin{asy}\n") #LaTeX file doesn't compile if asy environment is
    written in the LaTeX file... strange!
  f.write("size(20cm, 20cm);\n")
  if (withLabels): # Avoid to do the test at each iteration, even if
    toAsymptoteRecWith and toAsymptoteRecWithout are really similar
    toAsymptoteRecWith(t, f)
  else:
    toAsymptoteRecWithout (t, f)
  f.write("\\end{asy}\n")

  f.close()

def toAsymptoteRecWithout (t, f):
  for c in t.children:
    toAsymptoteRecWithout(c, f)
    f.write("draw((%f, -%f) -- (%f, -%f));\n" % (t.x, t.y, c.x, c.y,))

def toAsymptoteRecWith (t, f):
  f.write("label(\"%s\", (%f, -%f), E);\n" % (t.label, t.x, t.y,))
  for c in t.children:
    toAsymptoteRecWith(c, f)
    f.write("draw((%f, -%f) -- (%f, -%f));\n" % (t.x, t.y, c.x, c.y,))
```

Autre

```
from tree import *

import networkx as nx
import sys
import matplotlib.pyplot as plt

node_size=5

```

```
#Main function
def toNetworkX(treeToDraw, addLabels=False, outputfile="treeNetworkX",
    outputformat="png") :
  if (not(treeToDraw==None)) :
    print("Generating NetworkX...")
    (G, pos) = toNetworkXrec(treeToDraw)
    print("Printing...")
    plt.clf()
    plt.gca().invert_yaxis()
    nx.draw(G, pos, with_labels=False, node_size=node_size)
    # Draw the labels a little upper, so they don't overlap the nodes
    if (addLabels):
      node_labels = nx.get_node_attributes(G,'label')
      pos_labels=dict()
      for k, v in pos.items():
        pos_labels[k]=(v[0], v[1]-0.1)
      nx.draw_networkx_labels(G, pos_labels, labels=node_labels, with_labels=True,
    node_size=node_size)
    plt.savefig(outputfile+'.'+outputformat)
    G.clear()
  else :
    print("Tree must not be empty.")

#Aux rec function
def toNetworkXrec(treeToDraw, pos=None) :
  if (not(pos)) :
    pos=dict()
  #create a graph and add the root and its position
  G=nx.Graph()
  G.add_node(hex(id(treeToDraw)), label=treeToDraw.label)
  pos[hex(id(treeToDraw))] = (treeToDraw.x, treeToDraw.y)
  #print("noeud courant : ",treeToDraw.label)
  #print("est une feuille ? ",(not(treeToDraw.children)))
  if (not(treeToDraw.children)) :
    #the current node is a leaf -> return the graph and the associated positions
    return (G, pos)
  else :
    for t in treeToDraw.children :
      #the current tree is not a leaf ->
      #add one edge from the root to each son, make a graph
      #from each son, and combine all the results
      #G.add_edge(treeToDraw.label, t.label)
      G.add_edge(hex(id(treeToDraw)), hex(id(t)))
      (H, pos2) = toNetworkXrec(t, pos)
      for (node, data) in H.nodes_iter(data=True):
        G.add_node(node, **data)
      G.add_edges_from(H.edges())
      pos.update(pos2)
  return (G, pos)
```

3.3 Complexité

On suppose que la taille des labels est bornée. On note n le nombre de nœuds dans l'arbre. Dans ce cas :

1. Le parsing d'un fichier est en $O(n)$ car on parcours linéairement le fichier.
2. Le calcul des coordonnées est en $O(n)$ car on effectue 2 passes sur l'arbre et lors de chaque passe on visite une et une seule fois chaque nœud.
3. La génération du fichier de sortie est en $O(n)$ car on visite une unique fois chaque nœud pour écrire ses coordonnées dans le fichier de sortie.

On a donc une complexité générale en $O(n)$ où n est le nombre de nœud de l'arbre.

Notons que nous avons supposé que la taille des labels était bornée. Or nous n'avons aucune prise sur la taille des labels du fichier qui nous est passé en entrée. Dans ce cas, même si on borne la taille des labels pour l'affichage, la complexité est dominée par le parsing du fichier d'entrée car on doit lire tous les caractères du fichier.

Chapitre 4

Étude de performances

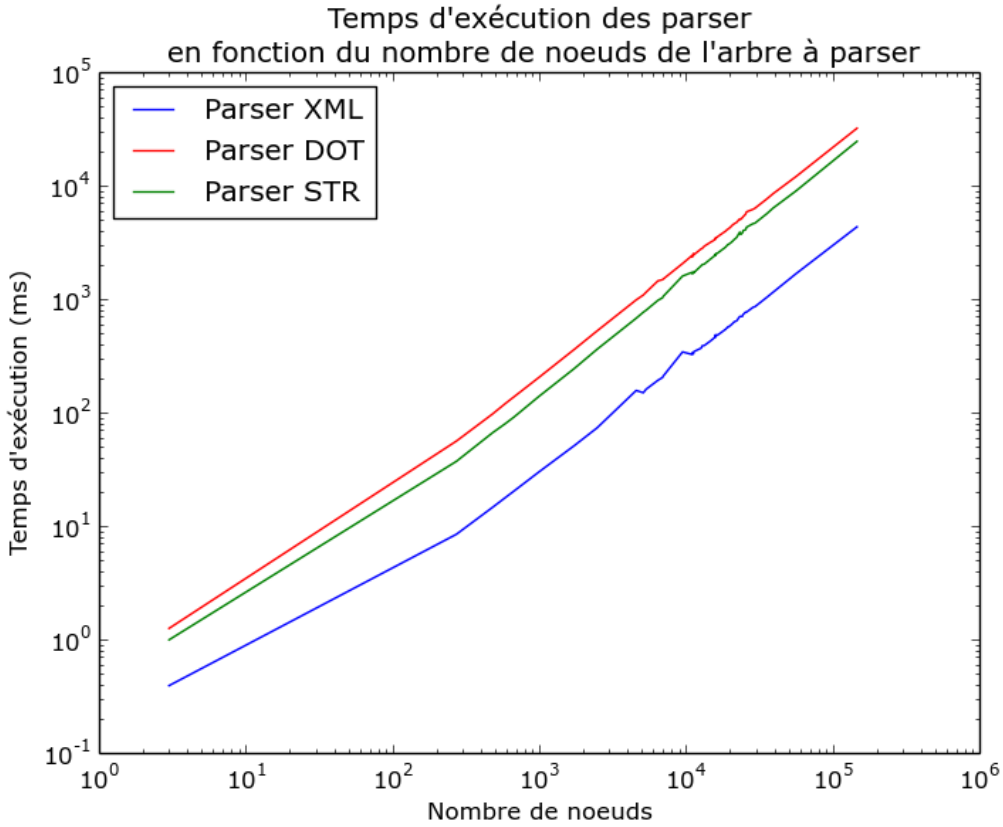
4.1 Protocole

Avant toute chose, il est important de préciser que les tests effectués ci-dessous ont été réalisés sans aucun label. Comme expliqué précédemment, nous pouvons considérer que si la taille des labels est constante, la complexité des algorithme ne dépendra que du nombre de nœuds compris dans l'arbre. C'est donc l'hypothèse que nous faisons ici.

Nous avons générer environ 2200 arbres de tailles croissantes en XML avec `Arbogen`. Sur ces 2200 arbres, nous en gardé aléatoirement 200 qui constituent notre jeu de tests. Chacun de ces arbres au format XML a été converti aux formats dot et str pour exécuter les parser sur les mêmes arbres. Ensuite, chacune des trois étapes de notre application - à savoir parsing, calcul de coordonnées et génération - a été exécuté 3 fois et la moyenne a été prise comme valeur de référence.

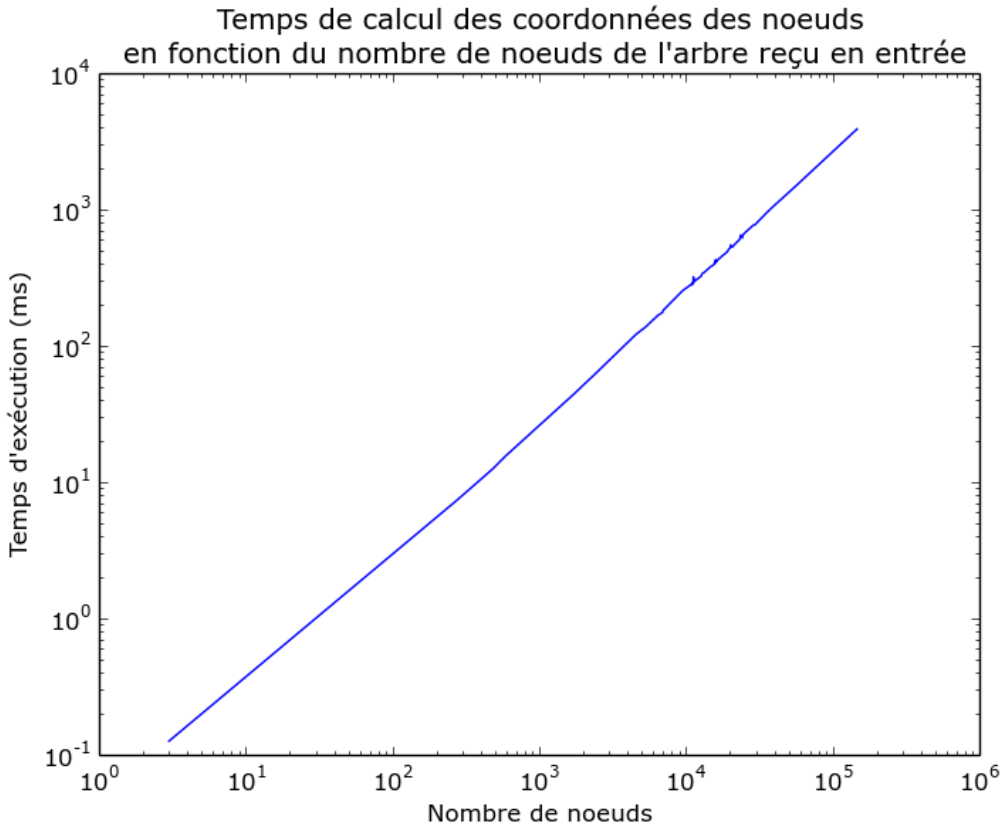
4.2 Résultats

4.2.1 Comparaison des temps d'exécution des différents parsers.



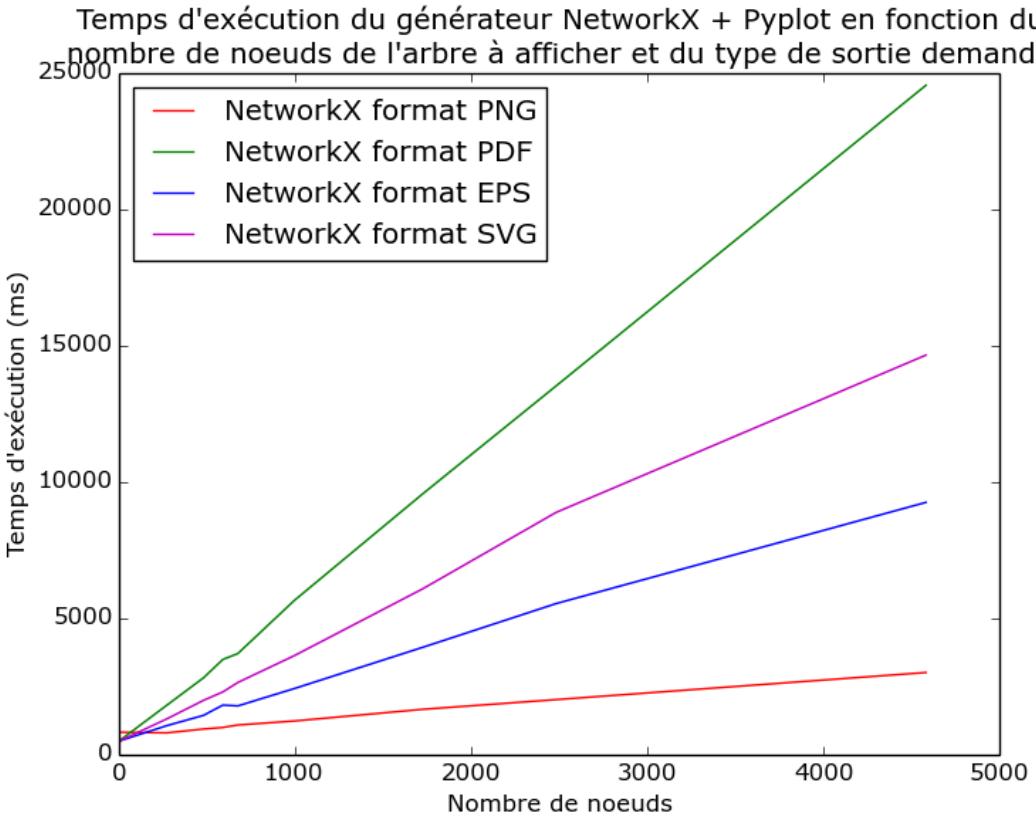
Comme on avait pu le montrer précédemment, on peut observer que les parser ont une complexité linéaire. Ce qu'il ressort également de cette comparaison, c'est que le parser XML est le plus efficace des trois. Cela est certainement du au fait que nous utilisons un module interne à Python pour traiter le fichier d'entrée. En effet, les modules internes des langages sont généralement bien optimisés car ils sont pensés pour le langage concerné. De plus, en à peine une dizaine de secondes, nous pouvons parser un arbre contenant un million de nœuds.

4.2.2 Évolution du temps de calcul des coordonées en fonction du nombre de nœuds de l'arbre.

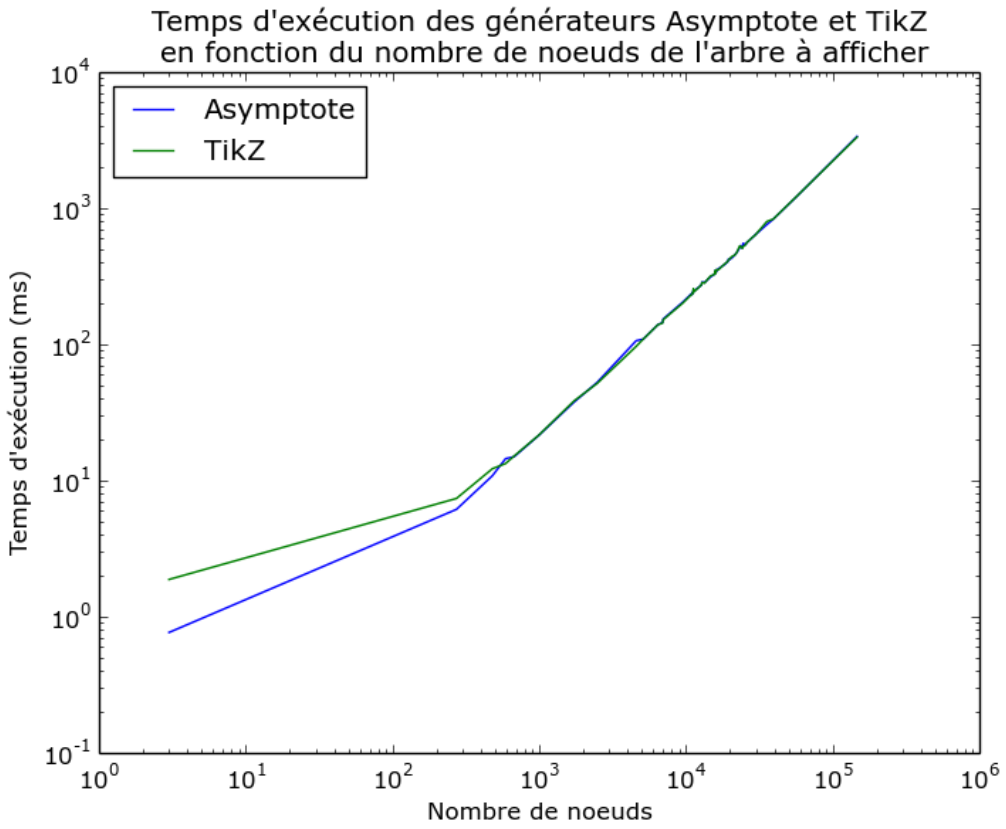


Les tests confirment que notre algorithme est d'une complexité linéaire en temps d'exécution. Nous pouvons également supposer, en observant l'évolution de la courbe, que pour un million de nœuds, nous n'aurions besoin que d'à peine une seconde pour effectuer le calcul de coordonnées, ce qui semble un résultat acceptable.

4.2.3 Comparaison des temps d'exécution des différents générateurs.

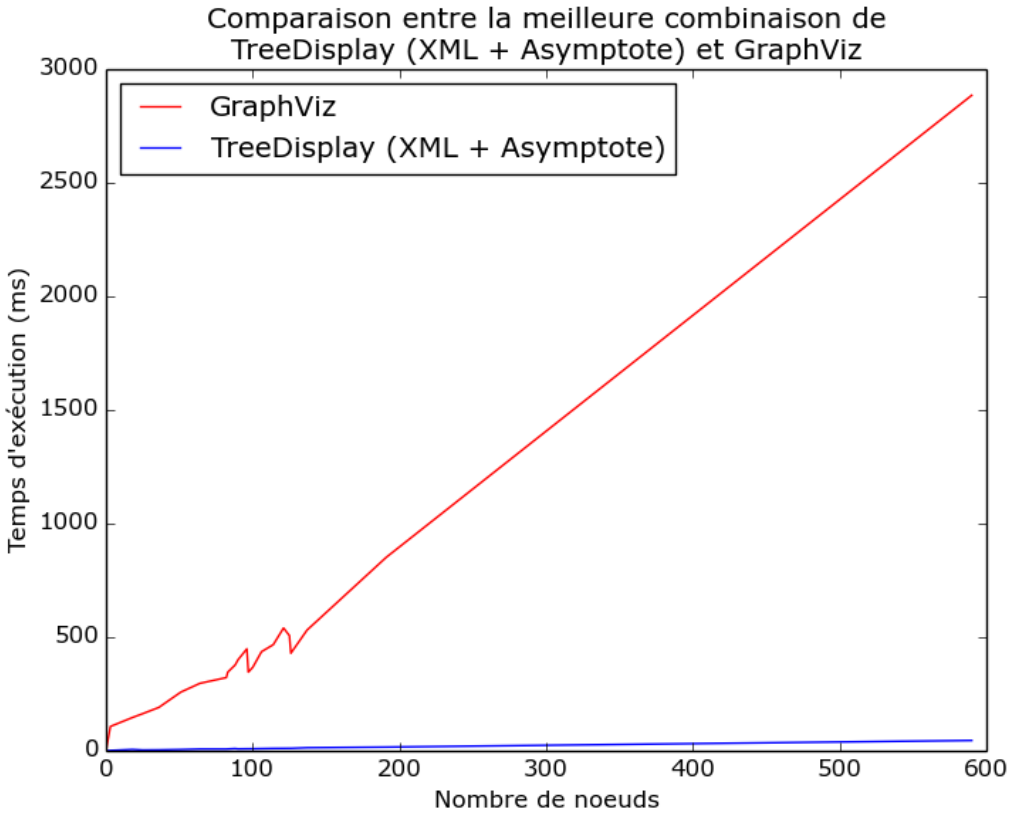


Nous comparons tout d'abord les différentes sorties gérées par NetworkX. Nous observons que nous sommes toujours en complexité linéaire, mais que le temps d'exécution est bien supérieur à ce qui a été observé pour les autres générateurs, avec plusieurs secondes pour seulement 5000 nœuds. Cependant, la complexité de ce générateur vient en grande partie du fait qu'il génère une image via MatplotLib. Ceci ralentit grandement l'exécution.



Contrairement à NetworkX, nous générons ici un fichier qui doit être recompilé. Ceci explique certainement en partie le fait que ce générateur soit plus efficace que le générateur NetworkX. Cependant, nous restons toujours en complexité linéaire.

4.2.4 Comparaison des temps d'exécution entre GraphViz et la meilleur combinaison parser/générateur de TreeDisplay.



Nous comparons maintenant TreeDisplay dans sa meilleure combinaison possible de parser et de générateur de sortie, avec le visualiseur que nous avons cité en début de ce rapport : GraphViz. Il en ressort que TreeDisplay est bien plus puissant que GraphViz. Cependant, GraphViz gère des graphes de manière générale et fournit une image en sortie, et nous avons généré un fichier .tex à compiler. Ceci peut expliquer en partie la différence de performance.

Chapitre 5

Conclusion

5.1 Bilan

L'application `treeDisplay` permet de calculer les coordonnées des nœuds d'un arbre en temps linéaire par rapport à ce nombre de nœuds. Cependant, l'affichage qui en résulte n'est pas aussi élégant que GraphViz. Nous ne pouvons par exemple pas voir la structure de l'arbre en détail. Mais puisque nous travaillons majoritairement avec des arbres de grande taille, les détails nous intéressent peu. On veut avant tout observer des tendances.

Nous avons également observé que la partie de l'application qui prend le plus de temps à l'exécution est le parsing et la génération de sortie. La clé de la complexité de cette application était donc de lire les fichiers d'entrée et de générer les fichiers de sortie de manière optimale, de façon à utiliser le moins de ressources possible.

5.2 Pour la suite

La structure choisie permet aussi de représenter des graphes. On peut donc envisager par la suite d'implémenter un algorithme de calcul de coordonnées pour les graphes. Les modules de parsing et de génération sont pleinement réutilisables.

La complexité en mémoire est actuellement égale à celle en temps. Cependant, on utilisant une table de hachage pour les sous-arbres, on peut devenir sous-linéaire. L'idée consiste à calculer des coordonnées relatives et lorsqu'un arbre comporte deux (ou plus) sous-arbres identiques. Le sous-arbre en question n'est sauvegarder en mémoire qu'une seule fois et la deuxième fois on ne sauvegarde qu'une référence. On calcule ensuite les coordonnées absolues lors de la génération.

L'algorithme de calcul de coordonnées implémenté respecte un certain nombre de contraintes que nous avons appelés *principes*. On peut étudier un autre algorithme qui va respecter d'autres contraintes comme par exemple la minimisation de la largeur.

Annexe A

Images obtenues lors de l'étude préliminaire

FIGURE A.1 – Arbre linéaire de taille 500 avec labels obtenu avec TikZ.

FIGURE A.2 – Arbre linéaire de taille 500 sans labels obtenu avec TikZ.

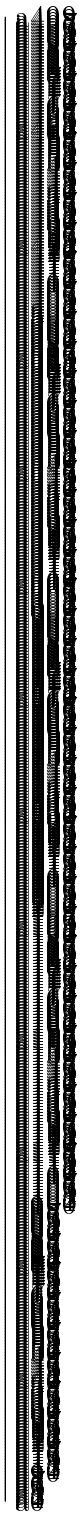


FIGURE A.3 – Arbre linéaire de taille 500 avec labels obtenu avec Asymptote.

FIGURE A.4 – Arbre linéaire de taille 500 sans labels obtenu avec Asymptote.

FIGURE A.5 – Arbre linéaire de taille 10000 avec labels obtenu avec Asymptote.

FIGURE A.6 – Arbre linéaire de taille 10000 sans labels obtenu avec Asymptote.

FIGURE A.7 – Arbre linéaire de taille 500 avec labels obtenu avec NetworkX et Maptplotlib.

FIGURE A.8 – Arbre linéaire de taille 500 sans labels obtenu avec NetworkX et Maptplotlib.

FIGURE A.9 – Arbre linéaire de taille 10000 avec labels obtenu avec NetworkX et Maptplotlib.

FIGURE A.10 – Arbre linéaire de taille 10000 sans labels obtenu avec NetworkX et Maptplotlib.

Annexe B

Extraits de l'implémentation

```
from tree import *
import re

nbCharAlire = 1024

def strParser (src):
  "Parses a string file into the intern tree structure"

  f=open(src, 'r')
  s=f.read(nbCharAlire)
  #(T, f) = parse(f, f.read(1))
  T=parse(f, s)[0]
  f.close()

  return T

def parse (f, s, currentIndex=0):

  isWhiteSpace=re.compile("[\s]")

  (f, s, currentIndex) = skipSpaces(f, s, currentIndex)

  if s[currentIndex]!='(':
    raise Exception ("The tree is ill-formed : A tree should start with a '('")

  (f, s, currentIndex) = incrementIndex(f, s, currentIndex)
  size = len(s)
  c = s[currentIndex]
  label=""

  (f, s, currentIndex) = skipSpaces(f, s, currentIndex)
  if (s==[]):
    raise Exception ("The tree is ill-formed : End of file reached")
  c = s[currentIndex]

  while (s!=[] and c!='(' and c != ')'):
    #print("currentIndex=",currentIndex)
    if (not isWhiteSpace.match(c)):
      label = label + c
    (f, s, currentIndex) = incrementIndex(f, s, currentIndex)
    c = s[currentIndex]

  (f, s, currentIndex) = skipSpaces(f, s, currentIndex)
  c = s[currentIndex]

  listChildren = []
  while (s!=[] and c == '('):
    (child, s, currentIndex) = parse (f, s, currentIndex)
    listChildren.append(child)
    #print("fils ",label,"ajouté")
    (f, s, currentIndex) = skipSpaces(f, s, currentIndex)
    c = s[currentIndex]
    #print("c aprés avoir ajouté le fils",label,": ",c)

```

```
  (f, s, currentIndex) = skipSpaces(f, s, currentIndex)
  c = s[currentIndex]
  if (c == ')'):
    (f, s, currentIndex) = incrementIndex(f, s, currentIndex)
    return (Tree(label = label, children=listChildren), s, currentIndex)
  else:
    raise Exception ("The tree is ill-formed : The tree",label,"should end with a
    ')'")

def incrementIndex(f, s, currentIndex):
  currentIndex+=1
  l=len(s)
  if (currentIndex>=1):
    #print("fin du buffer -> on re-remplit")
    s=f.read(nbCharAlire)
    currentIndex=0
  if (s==[]):
    raise Exception ("The tree is ill-formed : End of file reached")
  #print("currentIndex dans incrementIndex :",currentIndex)
  return (f, s, currentIndex)

def skipSpaces(f, s, currentIndex) :
  isWhiteSpace=re.compile("[\s]")
  while (s!=[] and isWhiteSpace.match(s[currentIndex])):
    (f, s, currentIndex) = incrementIndex(f, s, currentIndex)
  return (f, s, currentIndex)
```

Listing B.1 – ../Src/strParser.py

```
from tree import *
from pyparsing import Word, alphas
import re

def dotParser (src):
  "Parses a dot file into the intern tree structure"

  f= open(src)

  s = f.read()

  (toto, i) = getWord (s, 0, '{') # [strict] (graph | diagraph) [ID]

  k=0

  # [strict]
  while (toto[k] == " "):
    k += 1

  if (toto[k] == 's'): #read "strict"
    if re.compile('strict').search(toto):
      if (re.search(u'strict', toto).start() != 0):
        raise Exception ("Dot syntaxe error")
      else:
        k = re.search(u'strict', toto).end()
    else:
      raise Exception ("Dot syntaxe error")

  # (graph|digraph)
  while (toto[k] == " "):
    k += 1

  if (toto[k] == 'g'): #read "graph"
    if re.compile('graph').search(toto):
      if (re.search(u'graph', toto).start() != 0):
        raise Exception ("Dot syntaxe error")
      else:
        k = re.search(u'graph', toto).end()
    else:
      raise Exception ("Dot syntaxe error")
  elif (toto[k] == 'd'): #read "diagraph"
    if re.compile('digraph').search(toto):
      if (re.search(u'digraph', toto).start() != 0):
        raise Exception ("Dot syntaxe error")
```

```
      else:
        k = re.search(u'digraph', toto).end()
    else:
      raise Exception ("Dot syntaxe error")
  else:
    raise Exception ("Dot syntaxe error")

  # [ID]
  while (k<i and toto[k] == " "):
    k += 1



  i+= 1 #skip the {
  #print (s[i])

  dico = {}
  roots = set()
  children = set()

  try:
    while (True):
      (seq, i) = getWord (s, i, ';')
      seq = re.sub(r'\s', "", seq)
      j=0
      if (s[i-1] == ']'):
        # A node should be matched

        (num, j) = getWord(seq, j, '[')
        j += 1 # skip the '['

        (etiquette, j) = getWord (seq, j, '=')
        j += 1 # skip the '='

        (label, j) = getWord (seq, j, ']')
        j += 1 # skip the ']'

        if (num == "" or etiquette != "label" or label[0] != '\"' or
    label[len(label)-1] != '\"'):
          raise Exception ("Tree is ill-formed")

        dico[num] = Tree (label=label[1:len(label)-1])
        roots.add(dico[num])

      else:
        # An arrow should be matched

        (start, j) = getWord (seq, j, '-')
        j+=2 #skip -- or ->

        end = seq[j:]

        # Check if start and end are in dico
        t = dico[start]
        c = dico[end]
        if (t == None or c == None):
          raise Exception ("Tree is ill-formed")

        # If start not in children add to roots, add end to children
        #if t not in children:
        # roots.add (t)

        if c in roots:
          roots.remove (c)

        children.add (c)
        #print (t, c, t.children, c.children)

        #Add child to node
        t.children.append(c)

      i+=2 # skip the ; and go to the following item
  except IndexError:
```

```
    seq = s[i:]
    seq = re.sub(r'\s', "", seq)
    if (seq == "}"):
      # end of tree

      #verify that there is only one root and found it for returning
      if len(roots) != 1:
        raise Exception ("Tree is ill-formed")

      f.close()
      for x in roots:
        return x
      #return roots.get()

  #Exception indice out of array should have been raised
  f.close()
  raise Exception ("Tree is ill-formed")

def getWord (s, start, end):
  "Return a substring of s that starts at indice start and ends with caracter end
    excluded. Return also the indice of caracter end."
  res = ""
  j = start
  while (s[j] != end):
    res = res + s[j]
    j+=1
  return (res, j)
```

Listing B.2 – ../Src/dotParser.py

```
import xml.etree.ElementTree as etree
from tree import *

def xmlParser (src):
  "Parses a xml file into the intern tree structure"

  tree = etree.parse(src)
  root = tree.getroot()[0] #getRoot returns the node with the "tree" tag,
                #we want the "node" or "leaf" inside that node

  return parse(root)

# Given an xml tree parsed by ElementTree, returns the corresponding Tree object
def parse(xmltree):

  if(xmltree.tag=='leaf'):
    # Case 1 : the tree is a leaf
    # Try to get the id
    l=xmltree.get('id')
    if (l):
      return Tree(label=l)
    else:
      return Tree()
  else:
    # Case 2 : the tree is a node with children
    # Get the children
    children=[]
    for child in xmltree:
      children.append(parse(child))
    # Try to get the id of the node
    l=xmltree.get('id')
    if (l):
      return Tree(label=l, children=children)
    else:
      return Tree(children=children)
```

Listing B.3 – ../Src/xmlParser.py

Bibliographie

[1] Christoph Buchheim, Michael Jünger et Sebastian Leipert : Improving walker's algorithm to run in linear time. *In Graph Drawing*, 2002. http://dirk.jivas.de/papers/buchheim02improving.pdf.

[2] D. E. Knuth : Optimum binary search trees. *Acta Informatica*, 1(1):14–25, 1971. http://wenku.baidu.com/view/a4feec5e312b3169a451a42f.

[3] Kim Marriott et Peter J. Stuckey : Np-completeness of minimal width unordered tree layout. *Journal of Graph Algorithms ans Applications*, 8(3):295–312, 2004. http://www.emis.de/journals/JGAA/accepted/2004/MarriottStuckey2004.8.3.pdf.

[4] Frédéric Peschanski : Arbogen, 2014. https://github.com/fredokun/arbogen.

[5] Edward M. Reingold et John S. Tilford : Tidier drawing of trees. *IEEE Transactions on Software Enginering*, 7(2):223–227, mars 1981. http://scholar.google.fr/scholar_url?hl=fr&q=http://emr.cs.iit.edu/~reingold/tidier-drawings.pdf&sa=X&scisig=AAGBfm26weQoxIBL-KjosMM1a-_bR1Isnw&oi=scholarr&ei=QsZsU_j9OOnX0QWL24HwAw&ved=0CC8QgAMoADAA.

[6] Charles Wetherell et Alfred Shannon : Tidy drawing of trees. *IEEE Transactions on Software Enginering*, 5(5):514–519, septembre 1979. http://scholar.google.fr/scholar_url?hl=fr&q=http://poincare.matf.bg.ac.rs/~tijana/geometrija/seminarski/tree_drawing.pdf&sa=X&scisig=AAGBfm34LGKfBgU7qNXhH-xoqu4zpmbfdA&oi=scholarr&ei=QsZsU_j9OOnX0QWL24HwAw&ved=0CDAQgAMoATAA.