Visualisation d'arbres de grandes tailles
Rapport de PSTL

Érika Baëna
erika.baena@etu.upmc.fr

Diana Malabard
diana.malabard@etu.upmc.fr

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

7 mai 2014

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. Ceci pose problème lorsque l'on souhaite représenter des arbres dont l'ordre des fils et primordial : les arbres de recherche.
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 générer automatiquement un pdf ou intégrer le code à un document LaTeX ; Asymptote, une alternative à Tikz ; et NetworkX, pour pouvoir générer une image de l'arbre que l'on pourra ultérieurement insérer dans n'importe quel document.

Chapitre 1

État des lieux

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 LaTeXpermettant 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 LaTeXpour voir le résultat. Le fichier LaTeXcompile 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 LaTeXne 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 LaTeXmais 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 LaTeXcomme 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

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])
G = nx.path_graph(nbIte)
pos={x: (5, x*longueur_arete) for x in G.nodes()}
nx.draw(G, pos, node_size=5, with_labels=False)
plt.savefig("networkx_%d_nodes.png" % nbIte)
```

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

2.3.2 10000 nœuds

Passons maintenant à 10000 nœuds. Le résultat est aux figures

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 était 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 LaTeXce qui peut permettre d'avoir des labels un peu plus évolués qu'une chaine de caractères, une fois la taille des labels maitrisée.

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

```
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) respecte la grammaire suivante :

```
ARBRE : '(' ID NOEUDS ')'
NOEUDS : ARBRE NOEUDS | e
ID : [a-zA-Z1-9]* | e
```

Dot Le parser dot parse une sous-partie du langage dot, à savoir :

```
DOT : STRICT GRAPH ID '{' SEQINST '}'
STRICT : strict | e
GRAPH = diagraph | graph
SEQINST : INST SEQINST | e
INST : REF [label = "ID"] ';' | REF LINK REF
LINK : -- | ->
REF : [0-9]*
ID : [a-zA-Z1-9]* | e
```

XML Le parser XML utilise la librairie XML de Python. Il respecte la grammaire suivante :

```
XML : <?xml version="1.0"?><tree> NOEUDS <tree>
NOEUDS : NOEUD NOEUDS | e
NOEUD : <node type=TAG id=ID> NOEUDS </noeud> | <leaf type=TAG id=ID />
TAG : "Leaf" | "BinNode"
```

3.2.3 Fonctionnement de l'algorithme de calcul de coordonnées

On décide que la distance minimale entre deux nœuds est de 1 unité.

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

L'abcisse d'un nœud est un peu plus complexe et demande donc plus de réflexion. Pour un nœud donné, on commence par placer ses fils. On centre ensuite ce nœud au milieu de ses fils en faisant la moyenne de l'absisse de ses deux fils extrémaux. 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 père a des fils, on le centre au milieu de ses fils. Par ce calcul, on peut se retrouver avec un nœud qui est trop proche de son frère gauche [Mettre un exemple]. Pour cela, on compare la position calculée avec la première position disponible et on prend le max. 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-arbre dans un second temps.

1. Pour un nœud donné, on commence par placer ses fils.
2. On centre ensuite ce nœud au milieu de ses fils.
3. Si un nœud collisionne avec son frère gauche, on le décale vers la droite et on mémorise ce décalage car il faudra ensuite décaler ses sous-arbres. On applique ce décalage dans une seconde passe pour des raisons de complexité. En effet, si on faisait chaque décalage lorsqu'il se présentait, le décalage serait quadratique alors que dans le cas choisi, on est linéaire.

3.2.4 Fonctionnement de la génération du code

TikZ

Asymptote

Autre

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 $O(n)$.
2. Le calcul des coordonnées est en $O(n)$ (on effectue 2 passes sur l'arbre).
3. La génération du fichier de sortie est en $O(n)$.

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

Conclusion

4.1 Bilan

L'application `treeDisplay` permet de calculer les coordonnées des nœuds d'un arbre en temps linéaire par rapport à ce nombre de nœuds.

4.2 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.

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.

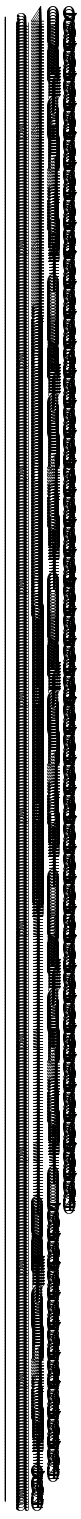

FIGURE A.3 – Arbre linéaire de taille 500 avec labels obtenu avec Asymptote.

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.

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,"éajout")
    (f, s, currentIndex) = skipSpaces(f, s, currentIndex)
    c = s[currentIndex]
    #print("c èaprs avoir éajout 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)
```

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

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

  # TO DO
  f= open(src)

  s = f.read()

  """
  (toto, i) = getWord (s, 0, '{')
  toto = re.sub(r'\s', "", toto)

  if (toto == "digraph"):
    arrow = "->"
  elif (toto == "graph"):
    arrow = "--"
  else:
    raise Exception ("Tree is ill-formed")
  """

  (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"
```

```
47      if re.compile('graph').search(toto):
          if (re.search(u'graph', toto).start() != 0):
49          raise Exception ("Dot syntaxe error")
          else:
51          k = re.search(u'graph', toto).end()
        else:
53        raise Exception ("Dot syntaxe error")
      elif (toto[k] == 'd'): #read "diagraph"
55      if re.compile('digraph').search(toto):
          if (re.search(u'digraph', toto).start() != 0):
57          raise Exception ("Dot syntaxe error")
          else:
59          k = re.search(u'digraph', toto).end()
        else:
61        raise Exception ("Dot syntaxe error")
      else:
63      raise Exception ("Dot syntaxe error")

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



71    i+= 1 #skip the {
      print (s[i])
73
      dico = {}
75    roots = set()
      children = set()
77
      try:
79      while (True):
          (seq, i) = getWord (s, i, ';')
81        seq = re.sub(r'\s', "", seq)
          j=0
83        if (s[i-1] == ']'):
            # A node should be matched
85
            (num, j) = getWord(seq, j, '[')
87          j += 1 # skip the '['

89          (etiquette, j) = getWord (seq, j, '=')
            j += 1 # skip the '='
91
            (label, j) = getWord (seq, j, ']')
93          j += 1 # skip the ']'

95          if (num == "" or etiquette != "label" or label[0] != '\"' or
        label[len(label)-1] != '\"'):
              raise Exception ("Tree is ill-formed")
97
            dico[num] = Tree (label=label[1:len(label)-1])
99          roots.add(dico[num])

101       else:
            # An arrow should be matched
103
            (start, j) = getWord (seq, j, '-')
105         j+=2 #skip -- or ->

107         end = seq[j:]

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

115         # If start not in children add to roots, add end to children
            #if t not in children:
117         # 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:]
      #print ("end:",seq)
      seq = re.sub(r'\s', "", seq)
      #(seq, i) = skipSpaces(s, i)
      #if (s[i] == '}'):
      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)
```

```
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')
```

```
32      if (l):
          return Tree(label=l, children=children)
34      else:
          return Tree(children=children)
```