

Visualisation d'arbres de grandes tailles¹

Présentation de PSTL

Érika Baëna
erika.baena@etu.upmc.fr

Diana Malabard
diana.malabard@etu.upmc.fr

Université Pierre et Marie Curie
Encadrant : Antoine Genitrini

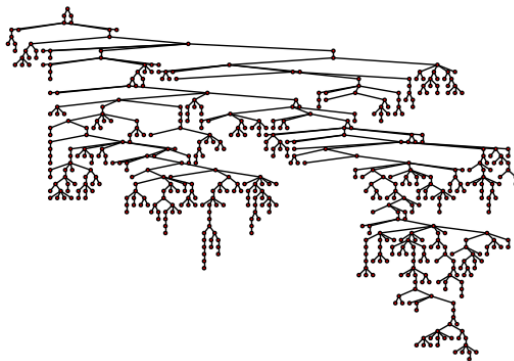
16 mai 2014

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

Plan de la présentation

- 1 Introduction
- 2 Contraintes d'affichage
- 3 Implémentation
- 4 Étude de performances
- 5 Comparaison des rendus
- 6 Conclusion

Pourquoi ?



- Pourquoi des arbres ? Structure primordiale en informatique.
- Pourquoi afficher des arbres de grande taille ? Pour observer des tendances.
- Pourquoi une nouvelle application ? Les arbres de recherche posent problème.

Objectif

Objectif du projet

Affichage élégant et efficace de tout type d'arbre

Objectif

Objectif du projet

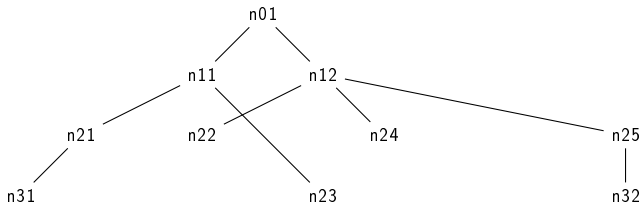
Affichage élégant et efficace de tout type d'arbre

Problèmes

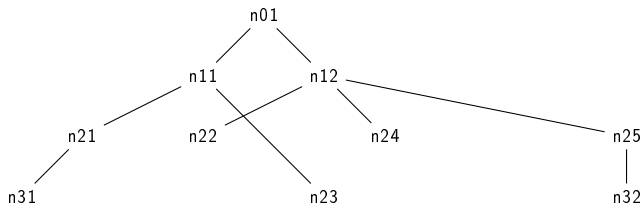
- Qu'est-ce qu'un affichage élégant ?
- Comment optimiser le calcul de la mise en page ?

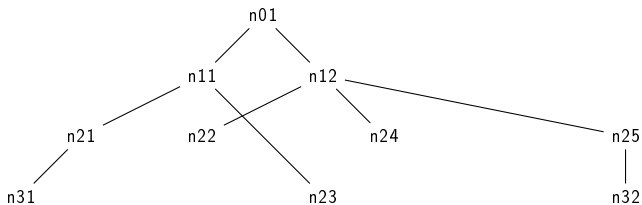
Plan de la présentation

- 1 Introduction
- 2 Contraintes d'affichage
- 3 Implémentation
- 4 Étude de performances
- 5 Comparaison des rendus
- 6 Conclusion



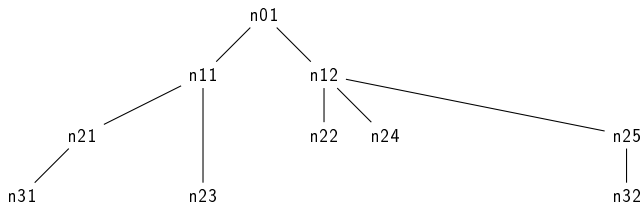
Que faire pour améliorer la visualisation de cet arbre ?

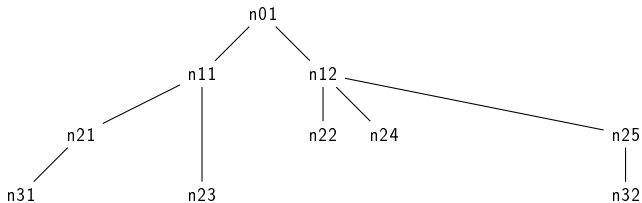




Contrainte 1

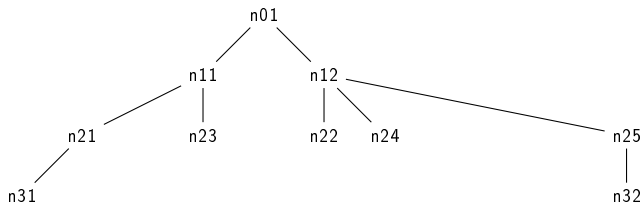
Les arêtes de l'arbre ne doivent pas s'intersecter.

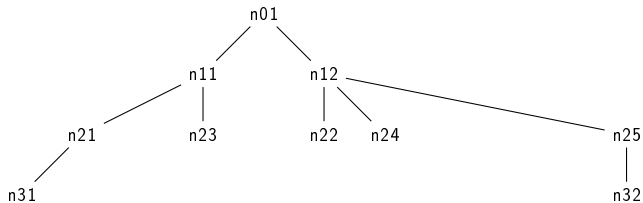




Contrainte 2

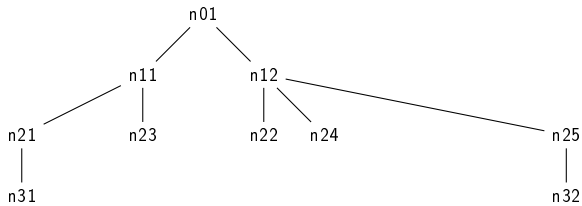
Les nœuds de même profondeur doivent être dessinés sur la même ligne horizontale.

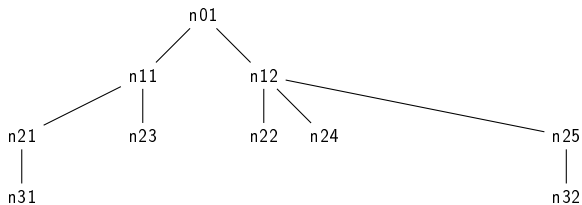




Contrainte 3

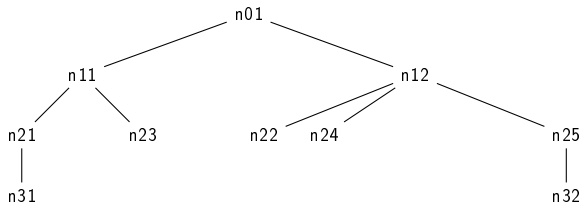
Un sous-arbre doit être dessiné de la même façon, peu importe où il est placé dans l'arbre.

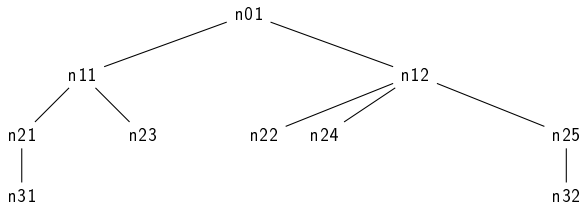




Contrainte 4

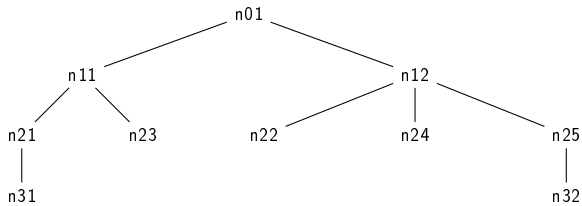
Un nœud parent doit être centré par rapport à ses fils.

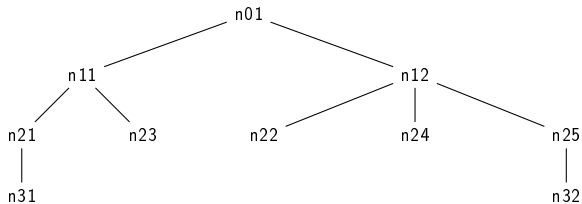




Contrainte 5

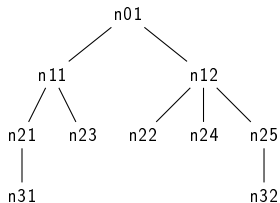
Les nœuds fils d'un nœud père doivent être espacés de manière homogène.





Contrainte 6

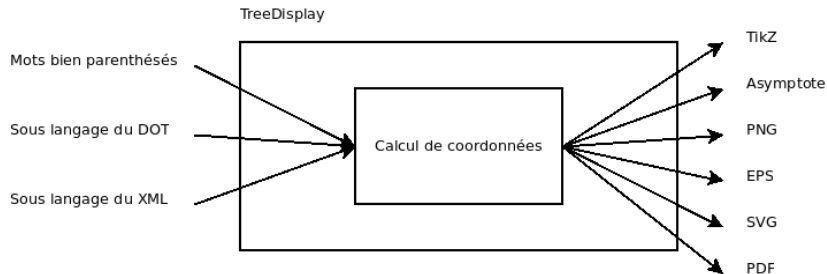
Les arbres doivent être dessinés de la manière la plus compacte possible.



Référence d'un arbre élégant.

Plan de la présentation

- 1 Introduction
- 2 Contraintes d'affichage
- 3 Implémentation
 - Fonctionnement général
 - Parsing
 - Calcul des coordonnées
- 4 Étude de performances
- 5 Comparaison des rendus
- 6 Conclusion



Mots bien parenthésés

Grammaire respectée

ARBRE ::= '(' LABEL NOEUDS ')'

NOEUDS ::= ARBRE NOEUDS | ϵ

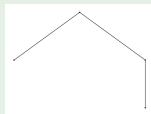
LABEL ::= [a-zA-Z1-9]* | ϵ

Exemple

Le parsing de

((()()))

donne l'arbre



Sous-langage de DOT I

Grammaire respectée

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

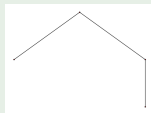

Sous-langage de DOT II

Exemple

Le parsing de

```
digraph {  
  1 [label=""];  
  2 [label=""];  
  3 [label=""];  
  4 [label=""];  
  1 -> 2;  
  2 -> 3;  
  3 -> 4;  
}
```

donne l'arbre



Sous-langage de XML I

Grammaire respectée

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

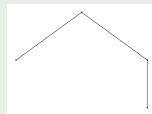
Sous-langage de XML II

Exemple

Le parsing de

```
<?xml version="1.0"?>
<tree>
  <node type="" id=1>
    <leaf type="" id=2 />
    <node type="" id=3>
      <leaf type="" id=4 />
    </node>
  </node>
</tree>
```

donne l'arbre



Structure de données

```
1 class Tree(object):  
    "Local representation of tree"  
3  
    def __init__(self, x = -1, depth=0, label="", children=None,  
        offset = 0, isRoot = False):  
5        self.x = x  
        self.y = depth  
7        self.label = label  
        self.offset = offset  
9        self.height = None  
        self.width = None  
11       if children is None:  
            self.children = list()  
13       else:  
            self.children = children
```

Idée générale

- Deux passes sur l'arbre
 - ① Centrer les pères sur les fils + gestion des collisions
 - ② Résolution des collisions
- Pourquoi? Complexité linéaire au lieu de quadratique

Algorithme 1

Passe 1

```
1  def setup (self, depth=0, nexts=None, offset=None):
2      if nexts is None:
3          nexts = defaultdict(lambda:0)
4          if offset is None:
5              offset = defaultdict(lambda:0)
6
7      # L'ordonnée est triviale, c'est la profondeur.
8      self.y = depth
9
10     # On calcule d'abord les coordonnées des enfants.
11     for c in self.children:
12         c.setup(depth+1, nexts, offset)
13
14     # On centre le noeud au milieu de ses enfants.
15     nbChildren = len(self.children)
16     if (nbChildren == 0):
17         place = nexts[depth]
18         self.x = place
19     else:
20         place = (self.children[0].x +
21                 self.children[nbChildren-1].x) / 2
22
23     # On calcule l'éventuel décalage engendré.
24     offset[depth] = max(offset[depth], nexts[depth]-place)
```

Algorithme II

Passe 1

```
25 # On applique le décalage de la profondeur.  
   if (nbChildren != 0):  
27     self.x = place + offset[depth]  
  
29 # On met à jour la prochaine place disponible à cette  
   profondeur.  
   nexts[depth] = self.x + 1  
31  
   # On mémorise le décalage à appliquer au sous-arbre lors de  
   la deuxième passe.  
33 self.offset = offset[depth]
```

Algorithme III

Passe 2

```
1  def addOffsets (self , offsum=0):  
    self.x = self.x + offsum  
3  offsum = offsum + self.offset  
  
5  self.height = self.y  
    self.width = self.x  
7  
    for c in self.children:  
9        c.addOffsets(offsum)  
        self.height = max (self.height , c.height)  
11       self.width = max (self.width , c.width)
```


Plan de la présentation

- 1 Introduction
- 2 Contraintes d'affichage
- 3 Implémentation
- 4 Étude de performances**
- 5 Comparaison des rendus
- 6 Conclusion

Protocole

- Une centaine d'arbres de taille de l'ordre de 10^0 à 10^5 générés aléatoirement
- Chaque instruction (parser, calcul des coordonnées, générateur) exécutée entre 5 et 10 fois
- Moyenne des temps d'exécution pour chaque instruction

Bilan général

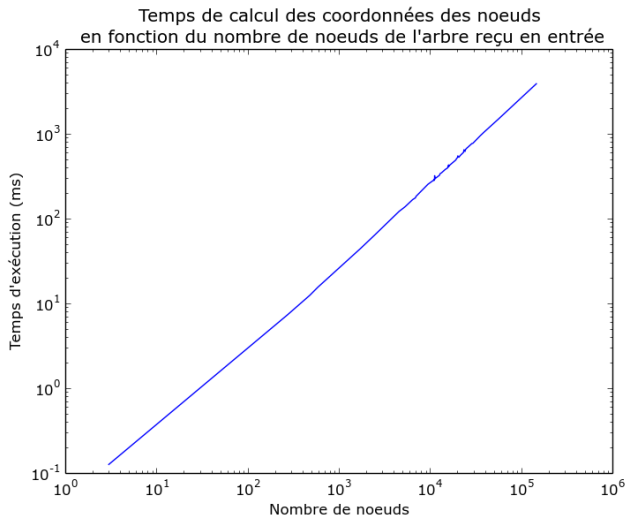
Meilleur cas (XML + Asymptote ou TikZ) :

- Parser $\sim 40\%$ du temps d'exécution
- Calcul des coordonnées $\sim 30\%$ du temps d'exécution
- Génération de la sortie $\sim 30\%$ du temps d'exécution

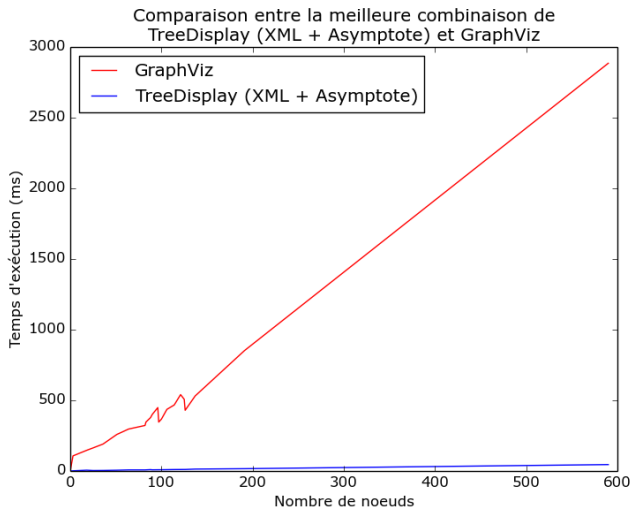
Pire cas (DOT + NetworkX) :

- Parser $\sim 10\%$ du temps d'exécution
- Calcul des coordonnées $\sim 1\%$ du temps d'exécution
- Génération de la sortie $\sim 89\%$ du temps d'exécution

Calcul des coordonnées



GraphViz vs TreeDisplay

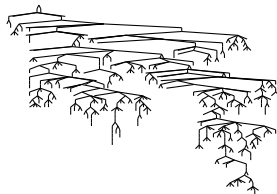


Plan de la présentation

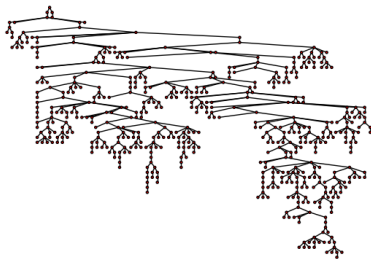
- 1 Introduction
- 2 Contraintes d'affichage
- 3 Implémentation
- 4 Étude de performances
- 5 **Comparaison des rendus**
 - Arbre de grande taille sans labels
 - Arbre de petite taille avec labels
- 6 Conclusion



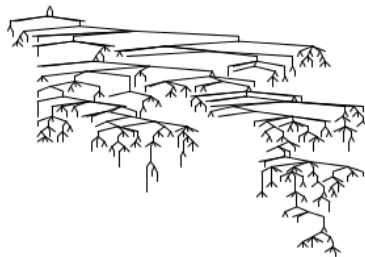
GraphViz



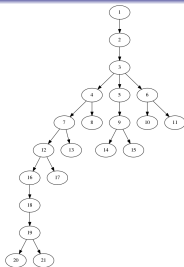
TikZ



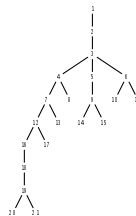
NetworkX



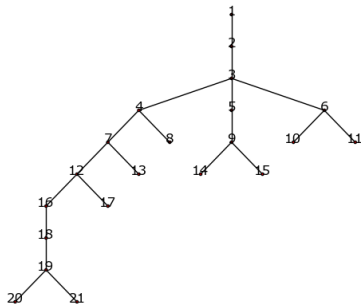
Asymptote



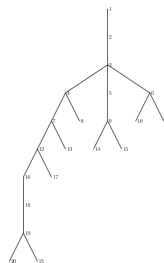
GraphViz



TikZ



NetworkX



Asymptote

Plan de la présentation

- 1 Introduction
- 2 Contraintes d'affichage
- 3 Implémentation
- 4 Étude de performances
- 5 Comparaison des rendus
- 6 Conclusion

Bilan

- Étude d'articles scientifiques
- Complexité linéaire
- Modules réutilisables

Pour la suite

- Extension aux DAG
- Optimisation mémoire
- Prise en charge du format ARB
- Ajout de critères de représentation
- Génération du profil des arbres trop grands