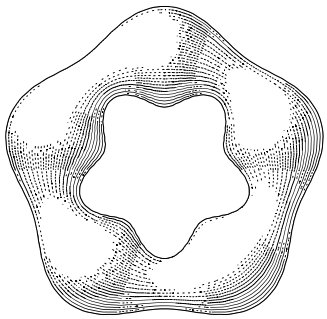
résultat (suite)



En espérant que cet outil pourra être utile, nous vous souhaitons beaucoup de belles figures.

Maxime Chupin

COMPOSER L'ARBRE DE HUFFMAN AVEC METAPOST ET METAOBJ

Pour des raisons pédagogiques, je souhaitais dessiner de beaux arbres (au sens informatique du terme) [28], plus particulièrement ceux issus de l'application de l'algorithme de Huffman (voir section « Algorithme de Huffman » page 36).

Pour dessiner des arbres, il existe de nombreux outils, tous très puissants. Par exemple, le package pst-tree est sans doute très adapté, en tout cas, sa belle documentation fournit un grand nombre d'exemples le laissant penser. De la même façon, fournit une syntaxe assez lisible et accessible pour représenter des arbres.

Cependant, étant utilisateur de METAPOST, je suis allé voir ce qui était disponible. Mon utilisation quasiment exclusive de LuaLaTeX rend l'utilisation de METAPOST pour produire des dessins extrêmement pratique grâce au package luamplib (voir mon article *LuaLaTeX et MetaPost avec luamplib, une introduction, Cahiers GUTenberg, numéro 58* [29]). De plus, je trouve le langage METAPOST bien plus adapté à la production graphique qu'un langage s'inscrivant dans la philosophie de TeX.

Du côté de METAPOST donc, il y a par son créateur même, John Hobby, le package boxes [30] permettant de dessiner et de relier des boîtes (rectangulaires ou circulaires). Il fournit quatre types de déclarations : créer des boîtes avec un nom, donner des relations géométriques entre ces boîtes, placer des boîtes à un endroit précis et les lier par les traits et des flèches. Je ne vais pas expliquer ce package plus en détail et je vous invite à consulter sa documentation disponible au format PDF `mpboxes.pdf` [31] (sous TeX Live, par exemple avec `texdoc mpboxes`), ou sa description dans le *The LaTeX Graphics Companion*. Nous illustrons simplement son utilisation dans l'exemple suivant.

28. À ce propos, le premier volume de *The Art of Computer Programming* en regorge, et je serais très curieux de voir ce qu'a produit Donald E. Knuth pour automatiser ces dessins !
29. `https://publications.gutenberg-asso.fr/cahiers/article/view/34`
30. Dont le code source se trouve, pour des raisons qui me sont inconnues, dans la partie « obsolète » du CTAN : `https://ctan.org/tex-archive/obsolete/graphics/metapost/base/texmf/metapost/base` et n'est pas trouvable quand on recherche `boxes`.
31. `https://tug.org/docs/metapost/mpboxes.pdf`

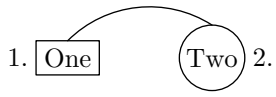
Exemple 19 : Package mptrees

code

```
input boxes;

beginfig(1);
boxjoin(one.e=two.w-(30,0));
boxit.one("One");
circleit.two("Two");
% on indique que l'est de la boîte one doit
% être séparé du vecteur (30,0) de
% l'ouest de la boîte two
% on dessine les boîtes
drawboxed(one,two);
label.lft("1.",one.w);
label.rt("2.",two.e);
draw one.n{dir 45} ..two.n;
endfig;
```

résultat



On dispose aussi du remarquable package mptrees, du francophone Olivier Péault. La syntaxe est très agréable, et le package semble fournir presque tous les éléments pour réaliser mon objectif. À l'époque de l'écriture du code présenté dans cet article, mptrees ne permettait pas de personnaliser l'aspect des feuilles et des nœuds (racines) des arbres. La dernière mise à jour corrige ceci, et il y a même un arbre de Huffman dans la documentation désormais. Vous pouvez voir ci-dessous un exemple de construction d'un arbre avec mptrees, et constater l'efficacité et la simplicité du code.

Exemple 20 : Package mptrees

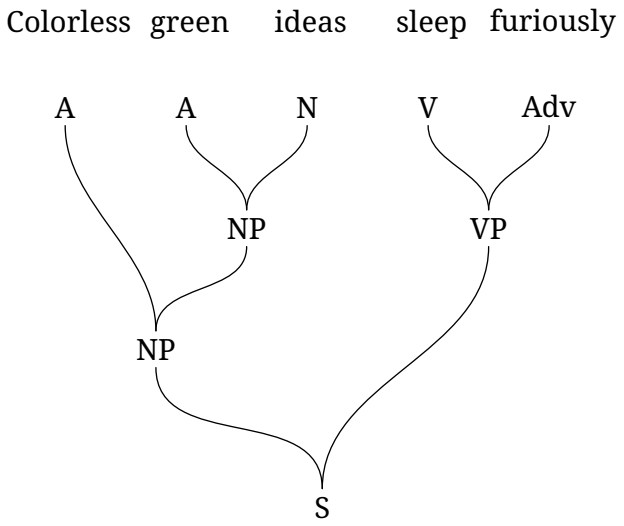
code

```
input mptrees;
beginfig(29)
u:=0.4cm;
branchtype:="curve";
dirlabel:=90;
abscoord:=true;
endlabelspace:=0.5cm;
draw startlabel("S");
draw tree[1][1]((-5.5u,4u),(5.5u,8u))("NP","","VP","");
draw tree[2][1]((-8.5u,12u),(-2.5u,8u))("A","","NP","");
draw tree[2][2]((3.5u,12u),(7.5u,12u))("V","","Adv","");
draw tree[3][2]((-4.5u,12u),(-0.5u,12u))("A","","N","");
draw endlabel[3][1]("Colorless");
draw endlabel[4][1]("green");
draw endlabel[4][2]("ideas");
draw endlabel[3][3]("sleep");
```

code (suite)

```
17  draw endlabel[3][4]("furiously");
18  endfig;
```

résultat



METAOBJ

En ouvrant le *The LaTeX Graphics Companion* [32], j'ai redécouvert METAOBJ, un package METAPOST qui permet justement de dessiner des arbres.

Ce package m'avait toujours intrigué. En effet, il semble permettre de faire beaucoup de choses mais semble aussi très abstrait. Grossièrement, il étend (largement) boxes et ajoute une couche orientée objet où l'on construit et manipule des objets très variés, à commencer par des standards tels que des boîtes, des conteneurs de boîtes, des matrices de boîtes, des objets récursifs, des arbres, etc. Il permet aussi de connecter ces objets. Cependant, cela ne s'arrête pas là puisqu'il offre la possibilité de construire ses propres classes d'objets, aussi compliquées que l'on souhaite ou presque.

Les outils proposés par ce package, s'ils peuvent permettre d'approcher les mécanismes des `\node` de Ti*k*Z, sont en réalité bien plus puissants.

METAOBJ repose sur d'importants principes, notamment ceux utilisés par boxes et PStricks, qui sont les suivants [33] :

— un objet doit avoir une structure avec une forme et potentiellement un contenu ;
— les objets doivent être créés avec un constructeur [34] ;
— il doit être possible de définir des objets « flotants » et de fixer leurs positions aussi simplement qu'avec le package boxes ;
— il doit être possible de transformer les objets, par exemple avec des rotations ;

32. Ce livre est extrêmement utile et plus particulièrement pour METAPOST dont les extensions sont, à mon sens, assez mal répertoriées et documentées.
33. Traduction du contenu du *The LaTeX Graphics Companion*.
34. Un constructeur est, en programmation orientée objet, une fonction particulière appelée lors de l'instanciation. Elle permet d'allouer la mémoire nécessaire à l'objet et d'initialiser ses attributs.

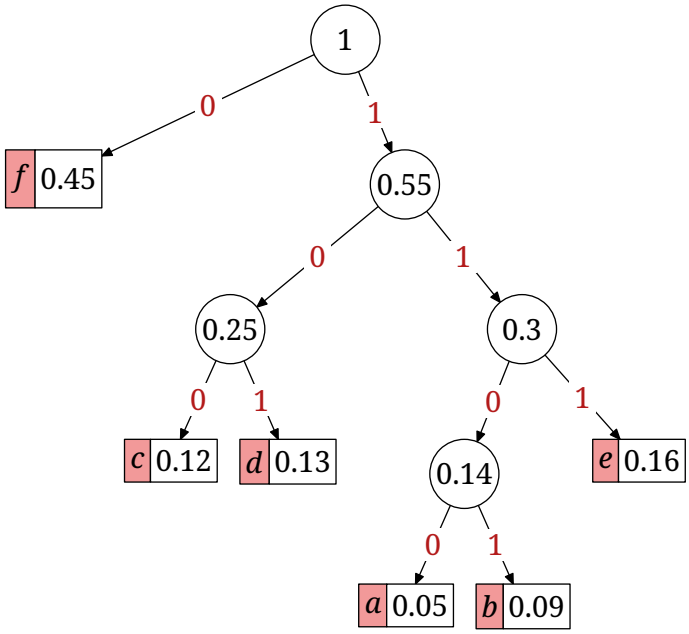
— il doit exister un mécanisme simple qui permet des spécifications par défaut qui peuvent être modifiées ;
— il doit être possible de construire des objets constitués d'autres objets, par exemple, mettre un carré dans un cercle.

En plus de respecter ces principes, METAOBJ permet de définir des nouvelles classes d'objets.

L'arbre à dessiner

Nous n'allons pas faire ici de présentation générale de METAOBJ qui mériterait beaucoup plus que ces quelques lignes. Nous allons, à travers un exemple de construction d'un arbre bien particulier, montrer comment on peut se servir de METAOBJ. Finalement, nous n'allons ici nous servir que de la partie de ce package consacrée aux arbres, mais rappelons qu'il peut faire bien plus !

L'arbre à produire est celui-ci :



Pour dessiner des arbres, METAOBJ fournit un constructeur : `newTree` dont la syntaxe générique est la suivante :

`newTree.`⟨`name`⟩`(`⟨`root`⟩`)(`⟨$\texttt{leaf}_1$⟩`,`⟨$\texttt{leaf}_2$⟩`,...,`⟨$\texttt{leaf}_n$⟩`)` ⟨`options`⟩

Le ⟨`name`⟩ sera le nom de notre objet *arbre*. La racine (⟨`root`⟩) et les descendants (⟨$\texttt{leaf}_i$⟩ [35]) peuvent être n'importe quels objets qui ont une interface standard [36]. Les ⟨`options`⟩ sont sous la forme de chaînes de caractères (donc entre *doubles quotes*). La forme générale des options est `option(valeur)` qui approche, à la mode METAPOST, le fonctionnement du classique `clé=valeur` en LaTeX. Nous renvoyons à la documentation de METAOBJ pour la liste des différentes options (qui sont nombreuses, et permettent de paramétrer très finement les arbres produits). Nous verrons dans les exemples suivants comment on fixe les positions des objets, mais indiquons tout de suite la fonction permettant de tracer l'objet en question.

`drawObj(`⟨`name`⟩`)`

35. *leaf* signifie « feuille ».
36. Pour METAOBJ

Arbre élémentaire

Regardons maintenant comment on peut construire un des arbres les plus simples, c'est-à-dire un nœud et deux feuilles. Pour cela, nous allons définir trois *conteneurs*. Les conteneurs peuvent être des simples boîtes rectangulaires, des cercles, des polygones, des matrices de conteneurs, des conteneurs de conteneurs, etc. Les possibilités sont impressionnantes. Pour se fixer les idées et en restant dans des éléments simples, l'exemple suivant montre un petit échantillon.

Exemple 21 : Des conteneurs

code

```
input metaobj

beginfig(0);
newEllipse.a(btex un texte etex);
a.c=origin;
drawObj(a);
newBox.b(btex un autre etex);
b.c=a.c+(2cm,0);
drawObj(b);
picture pc;
pc:=image(draw (0,0)--(0.5cm,0.5cm));
newPolygon.c(pc,8);
c.c=b.c+(2cm,0);
drawObj(c);
endfig;
```

résultat

Les constructeurs de conteneurs présentés ici s'appellent en suivant les syntaxes génériques suivantes :

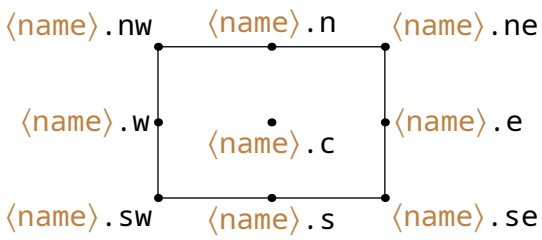
— `newBox.⟨name⟩(⟨content⟩) ⟨options⟩`

— `newEllipse.⟨name⟩(⟨content⟩) ⟨options⟩`

— `newPoylgon.⟨name⟩(⟨content⟩,⟨n⟩) ⟨options⟩`

Tous les conteneurs standards de METAOBJ prennent un argument (`⟨content⟩`) qui est une `picture` ou un objet METAOBJ et fournissent un cadre (boîte, cercle, polygone, etc.). Nous renvoyons à la documentation de METAOBJ pour les détails des options accessibles pour chaque conteneur (et elles sont nombreuses !).

On notera ici que pour fixer le centre de notre objet `⟨name⟩`, on a utilisé la syntaxe `⟨name⟩.c=...`. On peut aussi fixer l'est (`.e`), le nord (`.n`), etc., comme l'illustre la figure ci-dessous.



Ainsi, pour représenter un arbre élémentaire composé d'un nœud et de deux feuilles, il nous suffit de faire :

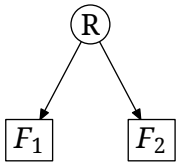
Exemple 22 : Simple arbre binaire

code

```
input metaobj;

beginfig(0);
newCircle.racine(btex R etex);
newBox.feuille1(btex $F_1$ etex);
newBox.feuille2(btex $F_2$ etex);
newTree.arbre(racine)(feuille1,feuille2);
arbre.c=origin;
drawObj(arbre);
endfig;
```

résultat



Première tentative

Avec cet élément de base, on peut donc, sans trop de difficulté, produire un arbre proche de celui souhaité.

Exemple 23 : Première tentative

code

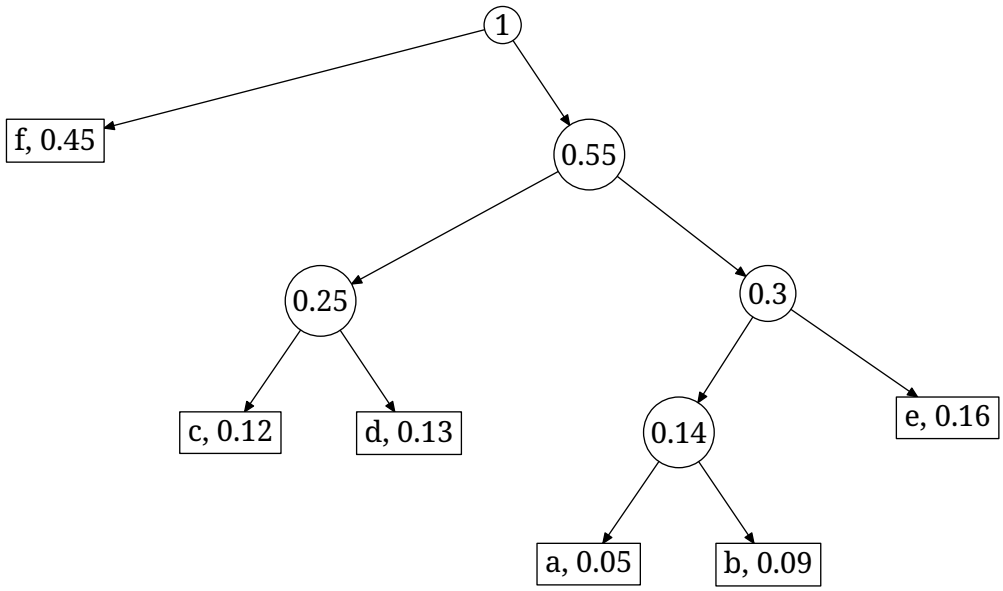
```
input metaobj;

beginfig(0);
% premier sous arbre
newCircle.noeud1(btex 0.14 etex);
newBox.a(btex a, 0.05 etex);
newBox.b(btex b, 0.09 etex);
newTree.arbre1(noeud1)(a,b);
% deuxième sous arbre
newCircle.noeud2(btex 0.3 etex);
newBox.e(btex e, 0.16 etex);
newTree.arbre2(noeud2)(arbre1,e);
% troisième sous arbre
```

code (suite)

```
newCircle.noeud3(btex 0.25 etex);
newBox.c(btex c, 0.12 etex);
newBox.d(btex d, 0.13 etex);
newTree.arbre3(noeud3)(c,d);
% quatrième sous arbre
newCircle.noeud4(btex 0.55 etex);
newTree.arbre4(noeud4)(arbre3,arbre2);
% cinquième sous arbre
newCircle.root(btex 1 etex);
newBox.f(btex f, 0.45 etex);
newTree.arbre5(root)(f,arbre4);
% placement
arbre5.c=origin;
drawObj(arbre5);
endfig;
```

résultat



Quelques améliorations

On peut constater quelques problèmes d'ordre esthétique dans l'arbre précédent : il serait plus joli avec des nœuds internes de même dimension et le style des feuilles est trop simple.

Contrairement à ce qui se fait avec Ti*k*Z, METAOBJ ne permet pas de définir des styles. Cependant, la création de nouveaux objets à partir d'objets standards est très simple : il suffit de définir une macro qui s'appuie sur le constructeur standard.

Pour régler le problème des nœuds internes de taille différente, nous allons construire le nœud comme un cercle vide auquel nous ajouterons un *label* en son centre. La macro définissant l'objet *Node* est alors la suivante.

Exemple 24 : Style de *node*

```
% pour tracer les boîtes vides
show_empty_boxes:=true;
% definition de la taille d'un nœud
_node_size:=13pt;
% style d'un nœud interne (non feuille) de l'arbre
vardef newHuffmanNode@#(expr v) text options=
    newCircle.@#("") "circmargin(_node_size)" options;
    ObjLabel.Obj(scantokens(str @#))(textext(v));
enddef;
```

Pour comprendre cette macro, il faut connaître un peu METAPOST. Le `@#` permet de récupérer ce qui se trouve après le point dans l'appel à la macro. Par exemple, dans `newHuffmanNode.petit("grand")`, `@#` vaut `petit` et `v` vaut la chaîne `"grand"`. On permet aussi de transférer les options au constructeur standard de METAOBJ `newCircle`. Il faut aussi savoir que `str` transforme des *tokens* en chaînes de caractères, l'opérateur `&` concatène des chaînes de caractères, et enfin `scantokens` transforme une chaîne de caractères en *tokens*.

On voit apparaître dans cette macro une autre fonctionnalité de METAOBJ : l'annotation d'objet avec la commande générique suivante :

`ObjLabel.Obj(`⟨objectName⟩`)(`⟨content⟩`)` ⟨options⟩

Grâce aux options, on peut paramétrer beaucoup d'éléments de l'annotation. Ici, nous nous servons de sa version la plus simple, c'est-à-dire mettre un texte au centre d'un objet.

Avec la macro `newHuffmanNode`, on définit un nouveau constructeur en essayant de respecter la philosophie de METAOBJ qui nous produit un nœud de taille fixe (qu'il faudra sans doute adapter suivant les exemples rencontrés) avec un texte en son milieu.

Attaquons-nous maintenant au style des feuilles de l'arbre de Huffman. Ici, nous allons passer par LaTeX avec une macro construisant l'assemblage des deux boîtes, une pour le symbole, l'autre pour sa valeur [37].

Exemple 25

```
\definecolor{mypink}{rgb}{0.95,0.6,0.6}
\fboxsep=2pt
\newcommand\contenu[2]{\hbox to#1{\hss\strut#2\hss}}
\newcommand\feuille[2]{%
  \mbox{\fcolorbox{black}{mypink}{\contenu{5mm}{#1}}%
    \hskip-\fboxrule\fbox{\contenu{1cm}{#2}}}%
}
```

Grâce à ce code, on obtient la feuille comme suit (avec LaTeX pour l'instant) :

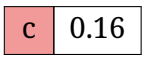
37. Nous remercions ici Denis Roegel pour nous avoir suggéré cette voie de présentation, ainsi que pour nous avoir fourni le code.

Exemple 26

code

```
\feuille{c}{0.16}
```

résultat



Si nous utilisons LuaLaTeX pour produire nos figures avec METAPOST, il suffira de mettre ce code LaTeX dans notre préambule de document. Si nous utilisons uniqement METAPOST, alors on pourra utiliser le latexmp pour faire appel à LaTeX dans METAPOST et on utilisera les mécanismes de personnalisation de préambule décrits dans la documentation de celui-ci pour y ajouter ces macros.

Une fois cette macro LaTeX créée, nous pourrons définir le style de feuille par la macro METAPOST suivante.

Exemple 27 : Style de *leaf*

```
% style d'une feuille symbole et de sa valeur
vardef newHuffmanLeaf@#(expr v) text options=
    % @# est le symbole considéré (ou la chaîne)
    % v est la valeur associée
    save str_leaf;
    % construction de la chaîne \feuille{<symbole>}{<valeur>}
    string str_leaf;
    str_leaf:="\feuille{"&str @# &"}{"&v&"}";
    newBox.scantokens(str @# & "char")(textext(str_leaf)) "framed(false)",
    "dx(0pt)", "dy(0pt)" ,options;
enddef;
```

Ici, nous ne traçons pas le contour de la boîte (`"framed(false)"`) puisque c'est LaTeX qui s'en charge, et nous enlevons les marges autour de la boîte avec `"dx(0pt)"` et `"dy(0pt)"`.

On aurait pu employer une autre méthode (c'est ce qui est fait dans le package présenté plus bas). En effet, METAOBJ fournit des constructeurs qui permettent de mettre à l'intérieur d'autres conteneurs. Ainsi, sans coder la chose pour notre arbre de Huffman, nous allons ici montrer un exemple d'agencement de boîtes horizontalement avec le constructeur `newHBox` et son option `"hbsep(0pt)"` qui permet de coller horizontalement les boîtes à l'intérieur de conteneurs.

Exemple 28 : Conteneur de conteneurs

code

```
input metaobj
color _huffmanLeaf;
_huffmanLeaf:=(0.95,0.6,0.6);
beginfig(0);
newBox.symb(btex \vphantom{Bq}$c$ etex) "filled(true)","
```

code (suite)

```
   fillcolor(_huffmanLeaf)";
newBox.valeur(btex \vphantom{Bq}0.15 etex) ;
newHBox.feuille(symb,valeur)  "hbsep(0pt)";
feuille.c=origin;
drawObj(feuille);
endfig;
```

résultat

On notera la présence de `\vphantom{Bq}` permettant d'ajouter une boîte verticale de largeur nulle et de hauteur celle contenant ici le *B* et le *q*. Ainsi, les deux boîtes ont la même hauteur et s'alignent bien.

Enfin, pour finir de peaufiner notre arbre, nous allons créer un constructeur très simple pour les arbres binaires élémentaires, et annoter leurs liens avec la valeur des *bits* de codage, 0 ou 1.

Exemple 29 : Style d'arbre binaire élémentaire

```
% couleur pour les bits
color _huffmanBit;
_huffmanBit:=(0.7,0.1,0.1);
% style de l'arbre binaire de Huffman
vardef newHuffmanBinTree@#(suffix theroot)(text subtrees
   ) text options=
    % un simple arbre
    newTree.@#(theroot)(subtrees) "Dalign(top)" , "hbsep
   (0.3cm)";
    % et on met 0 et 1 sur ses deux connections
    ObjLabel.Obj(@#)(textext("0"))
    "labpathid(1)", "laberase(true)", "labcolor(
   _huffmanBit)";
    ObjLabel.Obj(@#)(textext("1"))
    "labpathid(2)", "laberase(true)", "labcolor(
   _huffmanBit)";
enddef;
```

Si on met tout cela dans un fichier `huffloc.mp`, pour produire l'exemple voulu, on pourra écrire le code suivant :

Exemple 30 : Améliorations

code

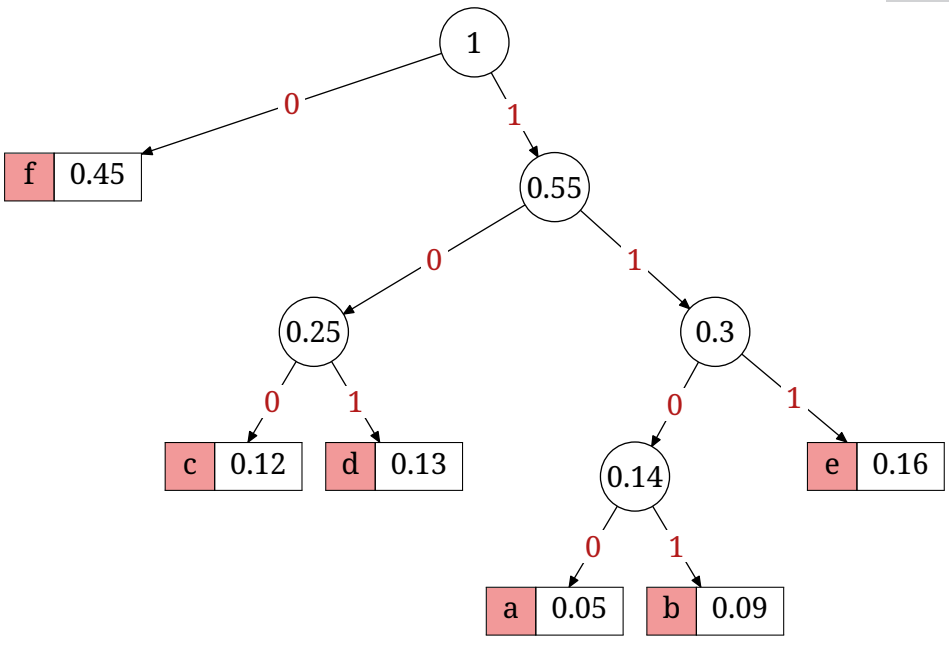
```
input metaobj;
input huffloc;

beginfig(0);
% premier sous arbre
newHuffmanNode.noeud1("0.14");
newHuffmanLeaf.a("0.05");
```

code (suite)

```
newHuffmanLeaf.b("0.09");
newHuffmanBinTree.arbre1(noeud1)(achar,bchar);
% deuxième sous arbre
newHuffmanNode.noeud2("0.3");
newHuffmanLeaf.e("0.16");
newHuffmanBinTree.arbre2(noeud2)(arbre1,echar);
% troisième sous arbre
newHuffmanNode.noeud3("0.25");
newHuffmanLeaf.c("0.12");
newHuffmanLeaf.d("0.13");
newHuffmanBinTree.arbre3(noeud3)(cchar,dchar);
% quatrième sous arbre
newHuffmanNode.noeud4("0.55");
newHuffmanBinTree.arbre4(noeud4)(arbre3,arbre2);
% cinquième sous arbre
newHuffmanNode.root("1");
newHuffmanLeaf.f("0.45");
newHuffmanBinTree.arbre5(root)(fchar,arbre4);
% placement
arbre5.c=origin;
drawObj(arbre5);
endfig;
```

résultat



Algorithme de Huffman

Nous avons dit plus haut que cet arbre est issu de l'application de l'algorithme de Huffman. Mais quel est cet algorithme et à quoi sert-il ?

L'algorithme de Huffman peut se comprendre sans trop de difficulté. On considère une suite de m symboles $(s_i)_{i\in\{1,\dots,m\}}$ (par exemple les caractères

ASCII), et une suite de m valeurs $(p_i)_{i\in\{1,\dots,m\}}$ qui sont des poids (par exemple la probabilité d'apparition du caractère dans un texte). On va chercher à construire l'arbre qui minimise la profondeur moyenne que l'on notera $\ell = \sum_{i=1}^{m} p_i n_i$ ou n_i est la profondeur de la feuille correspondant au symbole s_i. On présente ici l'algorithme dans le cas binaire où chaque nœud a au plus deux descendants.

Algorithme 1 : Algorithme de Huffman d'un code binaire de m symboles

Entrées : $(s_i)_{i\in\{1,\dots,m\}}$ symboles de distribution de poids $(p_i)_{i\in\{1,\dots,m\}}$.
Définir m nœuds actifs correspondants aux m symboles.
tant que *il reste strictement plus d'un nœud actif* **faire**
  Grouper, tant que fils d'un nœud nouvellement créé, les 2 nœuds actifs les moins probables.
  Marquer les 2 nœuds actifs ainsi choisis comme *non actifs* et le nœud nouvellement créé comme *actif*.
  Assigner au nœud nouvellement créé un poids égal à la somme des poids des 2 nœuds qui viennent d'être désactivés.

Comme l'illustrent les nombreux dessins de cet article, pour chaque nœud avec deux descendants, on attribuera les bits 0 et 1 à chacun des descendants. L'algorithme de Huffman permet d'obtenir un code binaire qui minimise la taille moyenne du codage de chaque symbole et, par là, réalise une compression. L'algorithme 1 décrit ce qu'on appelle le codage de Huffman. Il en est ainsi de la suite de symboles (a, b, c, d, e, f, g, h) de l'exemple 31 page suivante et des probabilités d'apparition associées $(0.04, 0.05, 0.06, 0.07, 0.1, 0.1, 0.18, 0.4)$ dans une chaîne de caractère à coder, par exemple :

$$hhehggdehaghbfgh$$

Cette chaîne contient donc 16 symboles, et si on code ces symboles (ici des caractères) sur un nombre fixe de bits (comme fait l'ASCII), on obtient le code suivant :

symbole	code
a	000
b	001
c	010
d	011
e	100
f	101
g	110
h	111

La taille de la chaîne $hhehggdehaghbfgh$ codée est alors de $16 \times 3 = 48$ bits. Si maintenant, on utilise le codage donné par l'arbre de l'exemple 31 page suivante :

symbole	code
a	11100
b	11101
c	1010
d	1011
e	1111
f	100
g	110
h	0

alors on obtient une taille de la chaîne codée en utilisant ce code est :

$$6_h \times 1 + 4_g \times 3 + 1_f \times 3 + 2_e \times 4 + 1_d \times 4 + 1_b \times 5 + 1_a \times 5 = 39 < 48,$$

où l'on a noté 6_h le fait que le h apparaisse 6 fois. Ainsi, notre chaîne codée avec Huffman est plus petite qu'avec un codage avec un nombre fixe de bits pour chaque symbole [38].

METAPOST est un langage suffisamment pratique et puissant pour que cet algorithme s'implémente assez simplement. C'est ce que nous avons fait en utilisant METAOBJ pour la représentation graphique de l'arbre. Nous n'allons pas ici présenter le code qui, sauf à fournir beaucoup d'explications, ne serait pas très lisible ; mais, une fois cela codé, et toujours en essayant de respecter la philosophie de METAOBJ avec la création d'un constructeur, on obtient le résultat présenté en exemple 31.

Cela a finalement donné lieu à la publication sur le CTAN du package huffman. Nous en illustrons quelques options dans les exemples suivants, et renvoyons à sa documentation pour plus de détails. Il s'agit certes un package très particulier, mais il est sans doute préférable que ce travail se retrouve partagé plutôt que de prendre la poussière numérique sur mon disque dur. Ensuite, peut-être que si cela intéresse, il pourra être enrichi [39], soit par moi, soit par d'autres, et c'est aussi la force du monde libre que de se stimuler, de s'entre-aider et de collaborer.

Exemple 31 : Un seul constructeur

code

```
verbatimtex \leavevmode etex;
input huffman;
string charList[];
numeric weight[];
beginfig(1);
charList[1]:="a"; weight[1]:=0.04;
charList[2]:="b"; weight[2]:=0.05;
charList[3]:="c"; weight[3]:=0.06;
charList[4]:="d"; weight[4]:=0.07;
charList[5]:="e"; weight[5]:=0.1;
```

38. Vous pourrez remarquer que le décodage ne pose pas de problème, car même si les codes sont de longueurs différentes, aucun code n'est le début d'un autre. On appelle ce type de code, un code *préfixe*.

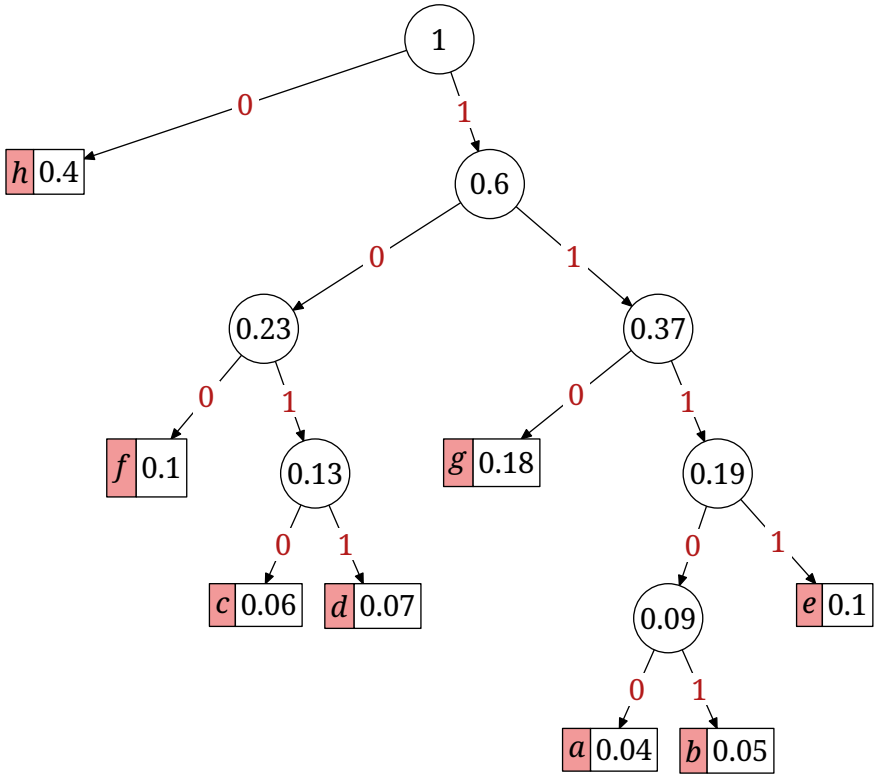
39. Et il y a de quoi faire : il faudrait par exemple coder l'algorithme de Huffman dans le cas où l'on n'a pas simplement un arbre binaire mais un arbre à q-aire (c'est-à-dire que chaque nœud a q descendants).

code (suite)

```
charList[6]:="f"; weight[6]:=0.1;
charList[7]:="g"; weight[7]:=0.18;
charList[8]:="h"; weight[8]:=0.4;

newBinHuffman.myHuff(8)(charList,weight);
myHuff.c=origin;
drawObj(myHuff);
endfig;
```

résultat



Ce package étant construit grâce à METAOBJ et le constructeur `newBinHuffman` étant une surcouche aux arbres de METAOBJ, on peut utiliser les options de METAOBJ concernant les arbres. Ainsi, on pourra simplement changer l'orientation de l'arbre et ajuster la séparation horizontale entre les étages de l'arbre (`"hsep()"`) et la distance verticale entre les descendants (`"vbsep()"`). On pourra aussi très simplement modifier le type de lien de descendance (`"edge()"`).

Exemple 32 : Un seul constructeur

code

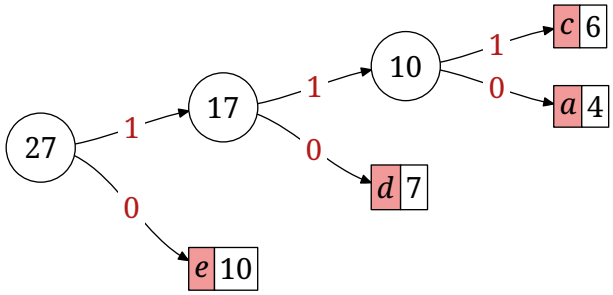
```
verbatimtex \leavevmode etex;
input huffman;
string charList[];
numeric weight[];
beginfig(1);
charList[4]:="a"; weight[4]:=4;
charList[3]:="c"; weight[3]:=6;
```

code (suite)

```
charList[1]:="d"; weight[1]:=7;
charList[2]:="e"; weight[2]:=10;

newBinHuffman.myHuff(4)(charList,weight)
"treemode(R)","hsep(1.5cm)","vbsep(0.5cm)", "edge(
   nccurve)",
"angleA(0)","angleB(0)" ;
myHuff.c=origin;
drawObj(myHuff);
endfig;
```

résultat



Des options propres au package huffman sont aussi disponibles. Sans en faire une présentation exhaustive, on peut en illustrer quelques-unes avec de la production de l'arbre suivant, beaucoup plus sobre.

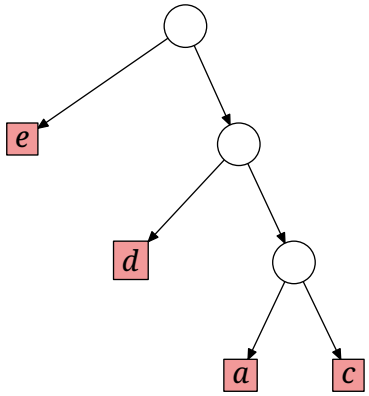
Exemple 33 : Un seul constructeur

code

```
verbatimtex \leavevmode etex;
input huffman;
show_bits:=false;
show_node_values:=false;
show_leaf_values:=false;
set_node_size(8pt);
string charList[];
numeric weight[];
beginfig(1);
charList[4]:="a"; weight[4]:=4;
charList[3]:="c"; weight[3]:=6;
charList[1]:="d"; weight[1]:=7;
charList[2]:="e"; weight[2]:=10;

newBinHuffman.myHuff(4)(charList,weight) "hbsep(1cm)";
myHuff.c=origin;
drawObj(myHuff);
endfig;
```

résultat (suite)



Pour aller plus loin

Si les sujets de cet article vous ont intéressés, voici quelques références pour approfondir le sujet.

Sur l'algorithme de Huffman : on pourra consulter la page Wikipédia dédiée [40]. On pourra aussi lire en anglais la publication de David A. Huffman de 1951 [41], ou [2, section 2.3] de Donald E. KNUTH.

Sur METAPOST : on pourra consulter les publications dans les *Cahiers* GUTenberg [42], notamment [3] de Fabrice POPINEAU, [4] de Yves SOULET ou enfin [5] de Denis ROEGEL. Pour voir METAPOST en action, on pourra aussi consulter le site web de l'association GUTenberg : `https//metapost.gutenberg-asso.fr`.

Sur METAOBJ : on pourra consulter sa très riche documentation [43] ainsi que la section dédiée de *The LaTeX Graphics Companion*.

Références

[1] Michel GOOSSENS et al. *The LaTeX Graphics Companion*. 2e édition. Upper Saddle River, NJ : Addison Wesley, 2 août 2007. 976 p. ISBN : 978-0-321-50892-8.

[2] Donald E. KNUTH. *The Art of Computer Programming : Volume 1 : Fundamental Algorithms*. 3e édition. Reading, Mass : Addison Wesley, 7 juill. 1997. 672 p. ISBN : 978-0-201-89683-1.

[3] Fabrice POPINEAU. « MetaPost pratique ». In : *Cahiers GUTenberg* 41 (2001), p. 167-175. URL : `http://www.numdam.org/item/CG_2001___41_167_0/`.

[4] Yves SOULET. « MetaPost raconté aux piétons ». In : *Cahiers GUTenberg* 52-53 (2009), p. 5-117. URL : `http://www.numdam.org/item/CG_2009___52-53_5_`.

[5] Denis ROEGEL. « MetaPost l'intelligence graphique ». In : *Cahiers GUTenberg* 41 (2001), p. 5-16. URL : `http://www.numdam.org/item/CG_2001___41_5_0/`.

Maxime Chupin

40. `https://fr.wikipedia.org/wiki/Codage_de_Huffman`
41. `https://www.ias.ac.in/article/fulltext/reso/011/02/0091-0099`
42. `http://www.numdam.org/search/Metapost-q&eprint%3DFalse/`
43. `https://ctan.org/pkg/metaobj`