Algorithmique Avancée

Antoine Genitrini

Antoine.Genitrini@Sorbonne-Universite.fr

Master Informatique 1

Moodle : UE MU4IN500

Année 2022-2023

Organisation (prévisionnelle)

- **Équipe pédagogique** :
  - **Cours** : A. Genitrini (mardi 10h45, amphi 45-A)
  - **TD** : P. Aubry (lundi matin), A. Genitrini (jeudi matin)
- **Planning** :
  - 1er cours le mardi 13 septembre
    TD : à partir du 19 septembre
- **Contrôle des Connaissances** (prévisionnel) :

  Session 1
  - Examen Réparti 1 (E1) : 7–10 Novembre 2022
  - Devoir de Programmation (D) : rendu Décembre 2022
  - Fin des enseignements le 16 Décembre 2022
  - Examen Réparti 2 (E2) : 3–13 Janvier 2023
  - (prévisionnel) Note de Session 1 = 0.3(E1) + 0.2(D) + 0.5(E2)

  Session 2
  - Examen Session 2 (SS) : Juin 2023
  - Note Session 2 = (SS)

Plan du cours

Objectifs : Complexité des algorithmes → Comparer, Optimiser

- **Compression**
  - Compresser du texte : méthode statistique
  - Compresser du texte : méthode par dictionnaire
  - Compresser des structures arborescentes
- **Structures Arborescentes : Files de priorité**
  - Files Binomiales et Files de Fibonacci
  - Coût amorti et Coût moyen
- **Structures Arborescentes pour la Recherche**
  - Arbres de Recherche équilibrés
  - Recherche externe
  - Tries et Arbres Digitaux
- **Méthodes de Hachage**
  - Hachage interne et externe
  - Hachage universel

Bibliographie

- T. Cormen, C. Leiserson, R. Rivest, C. Stein
  Introduction à l'algorithmique
- C. Froidevaux, M-C. Gaudel, M. Soria
  Types de données et algorithmes
- D. Beauquier, J. Berstel, P. Chrétienne
  Éléments d'algorithmique
- D. Knuth
  The art of computer programming (vol. 4)
- M. A. Weiss
  Data structures and algorithms analysis in C++
- R. Sedgewick, K. Wayne
  Algorithms
- S. S. Skiena
  The algorithm design manual

CHAPITRE 0
INTRODUCTION À LA COMPLEXITÉ

- Théorie de la complexité et classification de problèmes :
  - P : ce qui se calcule en temps polynomial $O(n^k)$
  - EXP : ce qui se calcule en temps exponentiel $O(2^n)$
  - NP : intermédiaire (P=NP ?)
- Problèmes exponentiels : optimisation combinatoire, systèmes cryptographiques, ...
- Problèmes polynômiaux : tri, recherche, géométrie, texte, arithmétique, ...

Affiner la classification, comparer les algorithmes, optimiser.

Analyse d'algorithmes

Algorithme $\mathcal{A}$ opère sur des données de $\mathcal{E}$ (mots, arbres, graphes)
taille des données : $\mathcal{E} \to \mathbb{N}$ (longueur mot, nombre sommet, ...)

Mesure de la complexité de $\mathcal{A}$ $\quad \tau_{\mathcal{A}} : \mathcal{E} \to \mathbb{N}$

*place mémoire, nombre d'**opérations fondamentales** effectuées*

Analyse de la complexité sur les données de taille n $\quad \tau_{\mathcal{A}} : \mathcal{E}_n \to \mathbb{N}$
pour comparer les méthodes de résolutions
Ex : multiplication de matrices, tri, recherche, ...

- complexité dans le meilleur des cas (minimale) : $\min\{\tau_{\mathcal{A}}(e); e \in \mathcal{E}_n\}$
- complexité dans le pire cas (maximale) : $\max\{\tau_{\mathcal{A}}(e); e \in \mathcal{E}_n\}$

temps réel, systèmes embarqués

- complexité en moyenne (uniforme) : $\frac{1}{|\mathcal{E}_n|}\sum_{e\in\mathcal{E}_n} \tau_{\mathcal{A}}(e)$

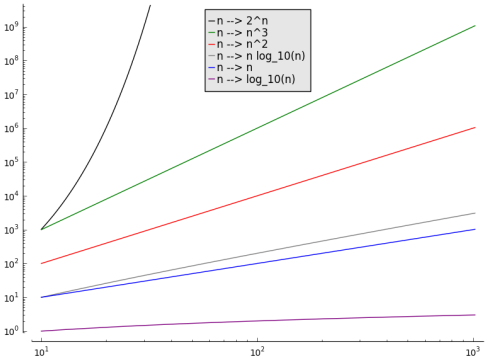
cas typique → probabilité des données

Analyse amortie : coût d'une suite d'opérations

Ordre de grandeur asymptotique

f et $g : \mathbb{N} \mapsto \mathbb{R}^+$

$f = o(g) \iff \lim_{n\to\infty} f(n)/g(n) = 0$
$f = O(g) \iff \exists \alpha \in \mathbb{R}^{+*} \mid \forall n_0 > n, f(n) \leq \alpha \cdot g(n)$
$f = \Theta(g) \iff f = O(g)$ et $g = O(f)$



Comparaisons d'ordres de grandeur

Machine faisant de l'ordre de 10^9 opérations/secondes :

	$n = 10$	$n = 10^3$	$n = 10^6$
$\log n$	$<<$ sec	$<<$ sec	$<<$ sec
$10 \log n$	$<<$ sec	$<<$ sec	$<<$ sec
$100 \log n$	$<<$ sec	$<<$ sec	$<<$ sec
n	$<<$ sec	$<<$ sec	$<<$ sec
$10 \cdot n$	$<<$ sec	$<<$ sec	$<<$ sec
$100 \cdot n$	$<<$ sec	$<<$ sec	$<<$ sec
n^2	$<<$ sec	$<<$ sec	Θ min
$10 \cdot n^2$	$<<$ sec	$<<$ sec	Θ heure
$100 \cdot n^2$	$<<$ sec	$<<$ sec	Θ jour
n^3	$<<$ sec	Θ sec	Θ année
$10 \cdot n^3$	$<<$ sec	Θ sec	∞
$100 \cdot n^3$	$<<$ sec	Θ min	∞
2^n	$<<$ sec	∞	∞
$10 \cdot 2^n$	$<<$ sec	∞	∞
$100 \cdot 2^n$	$<<$ sec	∞	∞

CHAPITRE 1

COMPRESSION

- Compression - Théorie de l'Information
- Compression de texte : méthode statistique
  - Algorithme de Huffman
  - Huffman adaptatif
- Compression de texte : méthode par dictionnaire
  - Méthode de Lempel-Ziv
  - Algorithme de Lempel-Ziv-Welch
- Compression de structures arborescentes
  - Principes
  - Application aux Diagrammes de Décision Binaires (BDD)

Généralités (1)

- Compression =
  Codage $C : A^* \mapsto \{0,1\}^*$
  + Modélisation (statistique ou dictionnaire)
- Abréger les redondances, réduire la taille
  (2 à 3 fois pour textes, plusieurs centaines de fois pour images)
- Motivations : Archivage et Transmission
- Algorithmes pour réduire la place occupée, sans perdre trop de temps
- Compression conservative (sans perte d'information) :
  fonction de compression réversible (injective)
  $\rightarrow$ compression de textes, de données scientifiques,
  de code compilé de programmes

$$\mathtt{T} \xrightarrow{Compression} C(\mathtt{T}) \xRightarrow{Transmission} C(\mathtt{T}) \xrightarrow{Decompression} \mathtt{T}$$

Généralités (2)

- Compression non conservative
  - transformation (inversible) des données
  - quantification (perte d'information)
  - compression conservative

$$\mathtt{T} \xrightarrow{Transf.} \tilde{\mathtt{T}} \xrightarrow{Perte} \hat{\mathtt{T}} \xrightarrow{Compr.} C(\hat{\mathtt{T}}) \xrightarrow{Decomp.} \hat{\mathtt{T}} \xrightarrow{Transf.Inv.} \mathtt{T}'$$

- Applications : approximation acceptable
  - compression de données analogiques (images, son)
  - transmission (télécopie), archivage (doubleur de disque)

Entropie – Quantité d'information

$\lceil log_2 n \rceil$ bits pour coder n symboles différents.

Théorie de l'information :
une source S produit n symboles différents $a_1, \ldots, a_n$ avec probabilité $p_1, \ldots, p_n$ (et $\sum p_i = 1$).

- Quantité d'information QI ou Entropie du symbole a_i = Nombre de bits pour coder a_i. Si a_i a probabilité p_i, alors QI = $log_2(1/p_i)$
  (aléatoires uniformes $\rightarrow$ QI = $\log_2 n$)
- Entropie de S = Valeur moyenne de la quantité d'information des symboles :

$$H = \sum_{i=1}^{n} p_i \log_2 \frac{1}{p_i} = -\sum_{i=1}^{n} p_i \log_2 p_i.$$

  Entropie max quand tous symboles équiprobables (source aléatoire uniforme).
  Plus entropie grande, plus grande est information transmise.

Théorèmes de Shannon :
Soit une source S, d'entropie H. a source S émet un message de longueur N.

- le codage d'un message aléatoire est toujours $> HN$.
- quand $N \rightarrow \infty$, il existe un codage dont la longueur $\rightarrow HN$.

Exemple : tirage dans une urne

On dispose d'une urne contenant des boules de 4 couleurs : rouge, bleue, jaune et verte. On souhaite encoder (le plus efficacement possible en longueur de code) la suite de couleurs qui sont tirées.

- L'urne contient une boule de chaque couleur.
  Aucun tirage privilégié $\rightarrow$ entropie maximale $\log_2 4 = 2$.
  Encodage : rouge $\rightarrow$ 00, bleue $\rightarrow$ 01, jaune $\rightarrow$ 10 et verte $\rightarrow$ 11
  Donc l'information contenue dans chaque tirage correspond bien à 2 bits.

- L'urne contient 4 boules rouges, 2 bleues, 1 jaune et 1 verte.

$$H = -\frac{4}{8}\log_2\left(\frac{4}{8}\right) - \frac{2}{8}\log_2\left(\frac{2}{8}\right) - \frac{1}{8}\log_2\left(\frac{1}{8}\right) - \frac{1}{8}\log_2\left(\frac{1}{8}\right)$$
$$= \frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \frac{3}{8} = \frac{7}{4}.$$

  Encodage : rouge $\rightarrow$ 0, bleue $\rightarrow$ 10, jaune $\rightarrow$ 110 et verte $\rightarrow$ 111
  Information nécessaire pour l'encodage : 1 bit 1 fois sur 2 ; 2 bits 1 fois sur 4 ; 3 bits 1 fois sur 4.

Codage des caractères

Chaque caractère est codé par une suite de N bits (appelée *symbole*).
Code = ensemble des couples (caractère, symbole).

- Code de Baudot (téléscripteurs) sur 5 bits. 2 caractères d'échappement permettent de doubler le nombre de symboles codés (Letters shift : caractères suivants doivent être interprétés comme des lettres et Figures shift : chiffres ou symboles spéciaux)
- Code ASCII (American Standard for Communication and Information Interchange).
  Code de base sur 7 bits
  ISO Latin sur 8 bits sur-ensemble de ASCII (contient en particulier les accents).
- BCD (Binary Coded Decimal) sur 6 bits : lettres, chiffres, ponctuation
- EBCDIC (Extended Binary Coded Decimal Interchange Code)
  BCD étendu à 8 bits pour contrôler les communications
- Unicode : codage de tout caractère de tout système d'écriture + propriétés sémantiques (composition, affichage, ...) $\rightarrow$ interopérabilité
  Définition du *jeu de caractères* : caractère $\rightarrow$ nom + identifiant (point-code)
  Codage : point-code $\rightarrow$ code informatique UTF (Universal Character Set) 8, 16 ou 32 bits.
  UTF-8 compatible ASCII mais code de longueur variable (1 $\rightarrow$ 4 octets).
  UTF-16 et UTF32 ordre de codage (Big-Endian ou Little-Endian)

Compression conservative

T texte sur un alphabet A, et codage $C : T \mapsto C(T)$

- Encodage des répétitions
- Méthodes statistiques – $C : A \mapsto \{0,1\}^*$
- Codage arithmétique – C : *Texte* $\mapsto \mathbb{R}$
- Méthodes avec dictionnaire – $C : A^* \mapsto \{0,1\}^*$

Souvent les progiciels utilisent plusieurs méthodes à la fois.

Encodage des répétitions

- **Principe** : remplacer suite de n symboles $a \ldots a$ par $a\#n$
- **Algorithmes** : Run Length Encoding (RLE) et variantes
- **Formats bitmap** : n&b, BMP (format natif Windows)

Méthodes statistiques

- **Principe** :
  symboles fréquents codés sur petit nombre de bits
- **Algorithmes** : Shannon–Fano (1948), Huffman (1950),
  Huffman dynamique (Gallager 1978) : algorithme adaptatif

Codage arithmétique : texte codé par un nombre réel,
L'algorithme produit un code pour un texte entier
(et non pour un symbole).
Meilleur que les autres en gain de place, mais peu rapide.

Méthodes avec dictionnaire

- **Principe** : code un groupe de symboles par son index dans un dictionnaire
  (statique ou adaptatif)
- **Algorithmes** : Lempel-Ziv77, Lempel-Ziv78, Lempel-Ziv-Welch
  - **compression de fichiers** COMPRESS Unix+ (LZW)
  - **formats d'image** GIF (Graphic Interchange Format) et
    TIFF (Tagged Image File Format)
  - **archivage** LHarc (LZ77+HD), free, PKZIP

Compression Statistique

1. Algorithme de Huffman statique
   - Principes et Définitions
   - Construction de l'arbre de Huffman
   - Optimalité du code
   - Compression : algorithme et complexité
   - Décompression : algorithme et complexité
2. Huffman Adaptatif
   - Principes et Définitions
   - Modification de l'arbre de Huffman adaptatif
   - Compresseur et décompresseur

Compression statistique – Huffman

Symbole fréquent $\rightarrow$ code court ; Symbole rare $\rightarrow$ code long

Au mieux :
coder un symbole avec le nombre de bits d'information qu'il contient $\log_2(1/p_i)$,
(mais ici nombre de bits est un *entier*).

Connaître les fréquences :

- Table de probabilité universelle
- Table de probabilité statique pour chaque texte à compresser : meilleure
  compression mais surcoût :
  calcul des fréquences + transmission de la table $\longrightarrow$ Algorithme de Huffman
- Fréquences calculées au fur et à mesure :
  codage d'un symbole évolue $\longrightarrow$ Algorithme de Huffman adaptatif
  Adaptativité inversible (fonction de décompression)

Code préfixe

Huffman : code préfixe de longueur variable.

Alphabet $A = \{a_1, \ldots, a_n\}$. Codage $C : A \mapsto \{0,1\}^+$.
P ensemble de mots de $\{0,1\}^+$, codages des caractères de A.

Définition :
P code préfixe $\stackrel{def}{=}$ aucun mot de P n'est préfixe propre d'un mot de P :

$$\forall w_1 \in P, \quad \text{si } w_1 w_2 \in P \text{ alors } w_2 = \epsilon.$$

Propriété :
Tout code préfixe est décodable de façon unique.

Aucun code d'un caractère n'est préfixe propre d'un code d'un autre caractère :
$C(xy) = C(x)C(y)$.

Arbre de type 2-trie

Définition : *arbre de type 2-trie* $\stackrel{def}{=}$ arbre binaire t.q. chaque feuille contient le codage de son chemin à partir de la racine (0 pour lien gauche, 1 pour droit).

Propriété : P est un code préfixe ssi il est constitué des codes des feuilles d'un arbre de type 2-trie.

Définition : Un code préfixe $C : A \mapsto \{0,1\}^+$ est dit *complet* ssi, lorsque u est un mot du code et $x < u$, alors les mots $x0$ et $x1$ sont soit des mots du code, soit des préfixes de mots du code.

Propriété : Un code préfixe complet est associé à un arbre de type 2-trie *complet*.

Remarques : Pour tout $a \in A$, $C(a)$ est un mot sur $\{0,1\}^+$, de longueur $L(a)$.
$L(a)$ est égal à la profondeur de la feuille codant a dans l'arbre de type 2-trie.

Code minimal

Texte $T \in A^*$. Codage $C : T \mapsto C(T) = T_C$.

Définitions :
Fréquence $f(a_i) = \#a_i$ dans T. *Taille* de T_C : $||T_C|| = \sum_{a_i \in A} f(a_i)|C(a_i)|$.

C code min. pour T ssi $||T_C|| = \min_\gamma\{||T_\gamma||, \gamma : A \to \{0,1\}^+\}$

Proposition :
C minimal pour T
$\Longrightarrow f(a_{i_1}) \leq f(a_{i_2}) \leq \ldots \leq f(a_{i_n})$ et $|C(a_{i_1})| \geq |C(a_{i_2})| \geq \ldots \geq |C(a_{i_n})|$.

Preuve : s'il existe a et b t.q. $f(a) < f(b)$ et $C(a) < C(b)$,
alors soit C' t.q. $C(x) = C'(x)$ pour tout $x \neq a, b$, et $C'(a) = C(b)$, et $C'(b) = C(a)$.
$||T_C|| - ||T_{C'}|| = f(a)C(a) + f(b)C(b) - (f(a)C(b) + f(b)C(a)) = (f(a) - f(b)) * (C(a) - C(b)) > 0$
Contradiction avec C minimal pour T.

Mais la réciproque n'est pas vraie. (ex : fréquences (2,2,3,3) : peigne ou équilibré)

Construction de l'arbre de Huffman

Exemple : $T = saperlipopette$, $f(p) = 3, f(e) = 3, f(t) = 2, f(s) = 1,$
$f(a) = 1, f(r) = 1, f(l) = 1, f(i) = 1, f(o) = 1$

Construction d'un arbre de type 2-trie

- Peigne par fréquences décroissantes
  $||T_C|| = 53$
- Utiliser les branchements
  $||T_C|| = 42$

Principe de construction d'un arbre de Huffman

- Unir 2 sous-arbres de poids minimal
- 2 degrés de liberté : choix des 2 sous-arbres + Gauche/Droite

Arbre de Huffman : Algorithme

ENTRÉE : Fréquences $(f_1, \ldots, f_n)$ des lettres (a_i) d'un texte T.

SORTIE : Arbre de type 2-trie donnant un code préfixe minimal pour T.

1. Créer, pour chaque lettre a_i, un arbre (réduit à une feuille) qui porte comme poids la fréquence f_i
2. Itérer le processus suivant :
   - Choisir 2 arbres G et D de poids minimal
   - Créer un nouvel arbre R, ayant pour sous-arbre gauche (resp. droit) G (resp. D) et lui affecter comme poids la somme des poids de G et D.
3. Arrêter lorsqu'il ne reste plus qu'un seul arbre : c'est un arbre de Huffman *(non unique)*

Construction de l'arbre de Huffman

Création arbre de Huffman contrôlée par une *file de priorité* de type tas.

Mais un tas, de quoi s'agit-il ?

Efficacité d'un tas

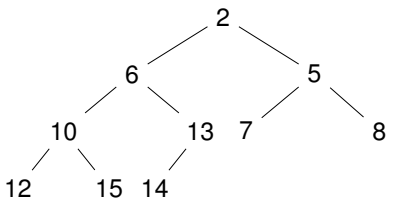
Nombre de comparaisons dans le pire des cas :

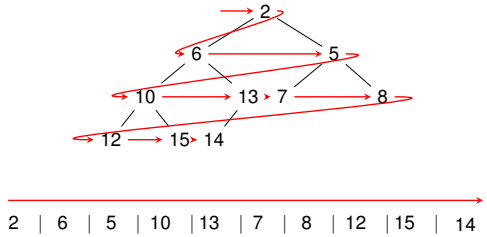
	Liste triée	Tas
Supp Min (1 élt parmi n)	$O(1)$	$O(\log n)$
Ajout (1 élt parmi n)	$O(n)$	$O(\log n)$
Construction (n élts)	$O(n^2)$	$O(n)$
Union (n élts et m élts)	$O(n+m)$	$O(n+m)$

Tas

Définition :
Un tas min est un arbre binaire étiqueté de façon croissante, dont toutes les feuilles sont situées au plus sur deux niveaux, les feuilles du niveau le plus bas étant positionnées le plus à gauche possible.

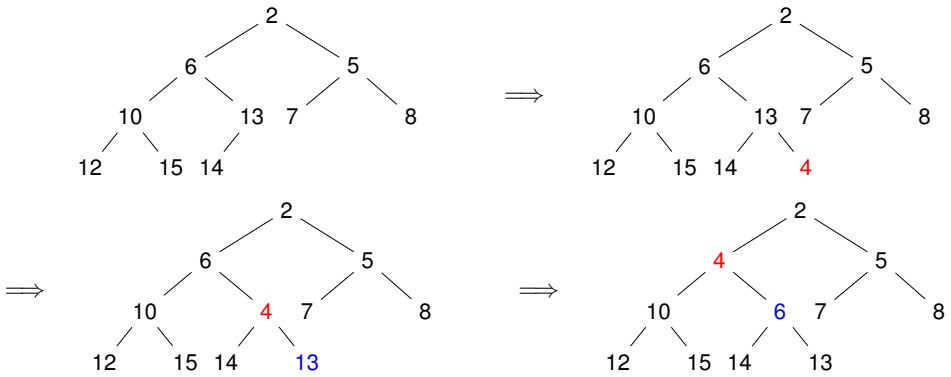
Exemple d'un tas *min* :





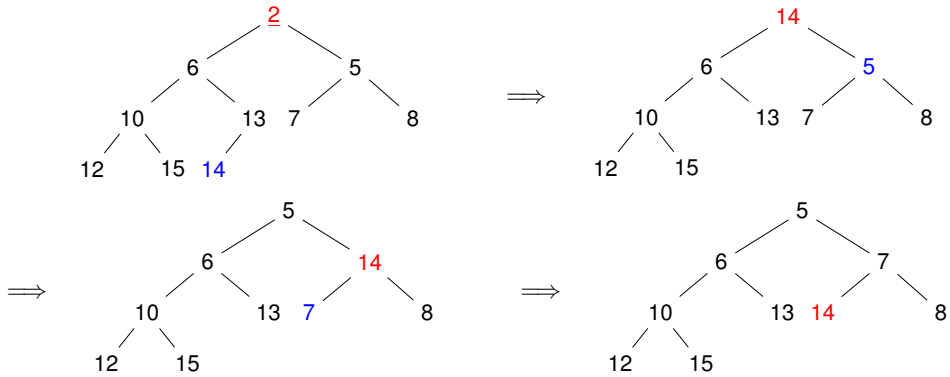
Insertion dans un tas

Exemple d'un tas *min* dans lequel on veut insérer la valeur 4 :



Extraction du min dans un tas

Exemple d'un tas *min* dans lequel on veut extraire le minimum :



Construction de l'arbre de Huffman

Création arbre de Huffman contrôlée par une *file de priorité* TAS

- éléments : arbres avec poids $(T, f(T))$, de type ABValué
- clés : poids attachés aux arbres
  (fréquences des lettres pour les feuilles et poids cumulés pour les arbres construits)

```
def Huffman(A, F):
        """ Alphabet * Frequences -> ArbreHuffman
            Hypothese : A = [a_1, a_2, ... a_n]
            Hypothese : F = [f_1, f_2, ... f_n]
            Renvoie l'arbre de Huffman associe a F."""

        FP = construireTas((a_1,f_1), ..., (a_n, f_n))
        for i in 1..len(F)-1:
                (G, FP1) = extraireMinTas(FP)
                (D, FP2) = extraireMinTas(FP1)
                R = ABvalue(G, D, pds(G)+pds(D))
                FP = ajouterTas(FP2, R)

        return extraireMinTas(FP)
```


Complexité de la construction de l'arbre de Hufffman

1. Construire le tas initial : $O(n \log n)$ *(on pourrait faire en $O(n)$)*
2. $(n-1)$ itérations :
   à chaque itération
   - extraire 2 fois l'arbre de poids min et réorganiser le tas : $O(\log n)$
   - fabriquer un nouvel arbre à partir des 2 précédents : $O(1)$
   - rajouter ce nouvel arbre au tas : $O(\log n)$

La construction de l'arbre de Huffman résultant est donc en $O(n \log n)$ comparaisons.

Taille du texte compressé : $||T_h|| = W(H) = \sum_1^n f_i L_i$,
où L_i est la profondeur de la feuille a_i dans l'arbre de Huffman
(nombre de bits dans le codage de a_i).

Preuve de l'optimalité du code de Huffman 1/2

Le code préfixe complet d'un arbre de Huffman est minimal i.e. meilleur que celui de tout autre arbre de type 2-trie de $A \to \{0,1\}^+$

Preuve : Par récurrence sur le nombre n de lettres de A
– pour $n = 2$, chaque lettre codée par 1 bit donc $||T_h|| =$# caractères de T
– (Hyp. de Réc.) : Supposons que pour tout alphabet de $(n-1)$ lettres, le code d'un arbre de Huffman est minimal.
– pour $n > 2$, l'algo "choisit" x et y de poids min, puis on définit un nœud z de poids $f(z) = f(x) + f(y)$ et applique récursivement la construction à $\tilde{T}$ sur $A - \{x,y\} \cup \{z\}$ qui contient $n-1$ lettres.
L'arbre $\tilde{H}$ construit pour $\tilde{T}$ produit un code minimal (Hyp. de Réc.), et on montre qu'il en est de même pour l'arbre H construit pour T. Par construction :
$W(H) = W(\tilde{H}) - f(z)L(z) + f(x)L(x) + f(y)L(y) = W(\tilde{H}) + f(x) + f(y)$.

Fin de preuve sur page suivante.

Preuve de l'optimalité du code de Huffman 2/2

Le code préfixe complet d'un arbre de Huffman est minimal i.e. meilleur que celui de tout autre arbre de type 2-trie de $A \to \{0,1\}^+$

Démonstration par l'absurde :
On suppose qu'il existe H' avec $W(H') < W(H)$, et trouvons $\tilde{H}'$ tel que $W(\tilde{H}') < W(\tilde{H})$, ce qui contredira $\tilde{H}$ minimal.
– Si x et y ont même parent ds H', on déduit $W(H') = W(\tilde{H}') + f(x) + f(y)$.
Or $W(H') < W(H)$ donc $W(H') - f(x) - f(y) < W(H) - f(x) - f(y)$ et finalement $W(\tilde{H}') < W(\tilde{H})$.
– Sinon on construit un arbre $H'' \mid x$ et y même parent et $W(H'') \leq W(H')$, et on sera ramené au cas précédent.
Construction de H'' : soient b et c, 2 feuilles ayant le même parent dans H', à prof. max., i.e. $L(b) = L(c) \geq L(f), \forall f$ feuille de H'.
H' est minimal et $f(x) \leq f(y) \leq f(b) \leq f(c)$ à symétrie près $x \leftrightarrow y$ et $b \leftrightarrow c$. On construit H'' en échangeant (x et b), et (y et c). On a donc
$W(H'') = W(H') - [(f(b) - f(x)).(L(b) - L(x))] - [(f(c) - f(y)).(L(c) - L(y))] \leq W(H')$ car les deux termes retirés sont positifs.

Algorithme de compression

ENTRÉE : Texte T

SORTIE : Texte compressé TC et codage de l'arbre de Huffman

1. Parcourir T pour compter le nombre d'occurrences de chaque caractère
2. Construire l'arbre de Huffman à partir des fréquences des caractères
3. Calculer la table des codes à partir de l'arbre de Huffman
4. Compresser T en concaténant les codes de chaque caractère $\to$ TC
5. Coder l'arbre de Huffman de façon compacte.

Analyse

Texte T : N caractères sur alphabet de n lettres,
Codage initial d'un caractère sur k bits

1. Parcours de T : $O(N)$. Stockage des fréquences : $O(n)$,
2. Construction de l'arbre H en $O(n \log n)$,
3. Construction table des codes : $O(n)$.
Stockage : $kn + \sum L_i$ bits,
4. Parcours de T : $O(N)$. Taille de TC : $\sum f_i L_i$,
5. Codage de H : Temps $O(n)$.
Taille de l'arbre codé sur $(2n - 1 + kn)$ bits (transparent suivant).

Codage de Dyck-ABCIF

Garder table des codes qui mémorise tous les chemins de l'arbre de Huffman :
$kn + \sum L_i$ bits

Mieux : Coder l'arbre de Huffman par un mot de Dyck enrichi de longueur $(2n-1)$
$\rightarrow (2n-1+kn)$ bits.

Arbre de Huffman = arbre binaire complet (n feuilles et $n-1$ nœuds internes)

Encodage de l'Arbre Binaire Complet avec Information aux Feuilles (ABCIF) via un mot de Dyck enrichi :
Parcours préfixe

- nœud interne $\longrightarrow$ 0
- feuille $\longrightarrow$ 1 ; de plus chaque 1 est suivi de k bits (code initial du caractère)

0 0 1 - 1 - 0 0 1 - 0 1 - 1 - 0 0 1 - 1 - 0 1 - 1 -

Exemple : 001p1e001t01s1a001r1l01i1o
Décodage : reconstruire l'arbre : 0 = nœuds interne et 1 = feuille,
et après chaque 1 lire k bits = codage initial d'un caractère.

Algorithme de décompression

ENTRÉE : Texte compressé TC, codage de l'arbre de Huffman

SORTIE : Texte décompressé T

1. Reconstruire l'arbre de Huffman $O(n)$
2. Parcourir TC en suivant l'arbre de Huffman $O(||TC||)$
   - suite de bits : de la racine à une feuille de l'arbre
     $\longrightarrow$ produit un caractère de T
   - et l'on poursuit le parcours de TC en recommençant à la racine de l'arbre

Statique $\longrightarrow$ Dynamique

Inconvénients de la compression de Huffman statique

- double parcours du texte (fréquences + compression)
- mémorisation du codage (arbre de Huffman)

Version dynamique – adaptative

- l'arbre de Huffman évolue au fur et à mesure de la lecture du texte et du traitement (compression ou décompression) des caractères.
- l'algorithme de compression (*compresseur*) et l'algorithme de décompression (*décompresseur*) modifient l'arbre de la même façon.
- à un instant donné du traitement les 2 algorithmes utilisent les mêmes codes – mais ces codes changent au cours du temps.

Principe Huffman dynamique

Compresseur Connaît les codes fixes (k bits) des caractères,

- Initialement, toutes fréquences nulles, et arbreH = #
- A la lecture de x dans T,
  - si prem. occ., retourne *code# dans H + code fixe*,
    et ajoute x à H
  - sinon retourne *code (variable) de x dans H*

  augmente de 1 la fréquence de x et modifie éventuellement H

Décompresseur Connaît les codes fixes (k bits) des caractères

- Initialement lit k bits, et ajoute caractère à arbre "vide" (H =#)
- Puis décode les bits du texte compressé sur $H \rightarrow$ feuille
  - si feuille=#, lit les k bits suivants de TC et ajoute le x à H
  - sinon retourne le caractère x correspondant à la feuille

  augmente de 1 la fréquence de x et modifie éventuellement l'arbre de Huffman de la même manière que le compresseur.

Arbre de Huffman adaptatif

Définition : Numérotation hiérarchique GDBH GaucheDroiteBasHaut

Définition : Un *arbre de Huffman adaptatif (AHA)* est un arbre de Huffman tel que le parcours hiérarchique GDBH $(x_1, \ldots, x_{2n-1})$ donne la suite des poids en ordre croissant $pds(x_1) \leq \ldots \leq pds(x_{2n-1})$

Propriété P : *Soit H un AHA et φ une feuille, de numéro hiérarchique x_{i_0}, dont le chemin à la racine est $\Gamma_\varphi = x_{i_0}, x_{i_1}, \ldots, x_{i_k}$ ($i_k = 2n-1$). Les nœuds du chemin Γ_φ sont dits incrémentables ssi $pds(x_{i_j}) < pds(x_{i_j+1})$, pour $0 \leq j \leq k-1$.*
Remarque : le sommet x_{i_j+1} se situe après x_{i_j} dans le parcours GDBH (en particulier x_{i_j+1} n'appartient pas nécessairement à Γ_φ).

Proposition : *soit H un AHA, φ une feuille de H et Γ_φ son chemin à la racine. Si Γ_φ vérifie la **Propriété P**, alors l'arbre résultant de l'incrémentation de φ est encore un AHA.*

Principe de modification

Modification après lecture d'un caractère de T correspondant à la feuille φ, de chemin Γ_φ à la racine de l'arbre de Huffman

- si Γ_φ vérifie **P**, incrémenter les poids sur ce chemin, sans modifier l'arbre,
- sinon transformer l'arbre pour qu'il vérifie **P**, puis incrémenter les poids sur le (nouveau) chemin de φ à la racine.

Traitement de l'arbre

- soit m le premier sommet de Γ_φ qui ne vérifie pas **P**, et soit b tel que $pds(x_m) = pds(x_{m+1}) = \ldots = pds(x_b)$ et $pds(x_b) < pds(x_{b+1})$
  (b est en *fin de bloc* de m ; on dispose d'une fonction `finBloc(H,m)` qui renvoie le nœud b)
- échanger sous-arbres $A1$ et $A2$ de racines numérotées x_m et x_b
  Rem : On n'échange jamais un nœud avec un de ses ancêtres
- recommencer le même traitement sur le nouveau chemin de la racine de $A1$ à la racine de l'arbre de Huffman

Algorithme de Modification

Feuille spéciale # de fréquence 0, et dont la position varie au cours du temps

```
def Modification(H, s):
    """ AHA * Symbole -> AHA
        Hypothese : H != #
        Renvoie l'arbre de Huffman avec s incremente."""

    if H == #:
        H devient arbre avec un noeud interne (pds 1)
            avec enfant gauche #
            et enfant droit s (pds 1)
        return H

    elif s not in H:
        Q = parent(feuille #)
        Remplacer noeud(#) par noeud (pds 1)
            avec enfant gauche #
            et enfant droit s (pds 1)

    else:
        Q = feuille(s)
        if enfants(parent(Q)) == {Q, #} and parent(Q) == finBloc(H,Q):
            Ajouter 1 au poids de Q
            Q = parent(Q)

    return Traitement(H, Q)
```


Algorithme de Traitement

```
def Traitement(H, Q):
    """ AHA * noeud  -> AHA """

    Q, Q_i1, ..., Q_ik = Gamma_Q le chemin direct de Q a la racine
    x_i0, ..., x_ik = les num des noeuds de Gamma_Q
    if tous les noeuds de Gamma_Q sont incrementables:
         Ajouter 1 a chaque pds sur le chemin de Gamma_Q
         return H
    else:
        m = premier indice de Gamma_Q tel que pds(x_m) == pds(x_(m+1))
        b = finBloc(H,m)
        Ajouter 1 a chaque pds du chemin de Q a Q_m
        Echanger dans H les sous-arbres enracines en Q_m et Q_b
        return Traitement(H, pere(Q_m))
```

Rappels :
- tous les nœuds de *Gamma_Q* sont incrémentables
ssi $\forall j \in \{0, \ldots, k-1\}$ tel que *x_ij* $\in$ *Gamma_Q* et *x_(ij + 1)*,
son successeur dans GDBH, vérifient $pds(x\_ij) < pds(x\_(ij+1))$
- *x_(m + 1)* est le successeur de *x_m* dans GDBH et b est tel que
x_m, x_(m + 1), ..., x_b sont des nœuds successifs dans GDBH, de même poids et
$pds(x\_b) < pds(x\_(b+1))$

Complexité $O(n)$ (hauteur maximale de H)

Algorithme de compression

ENTRÉE : Texte T

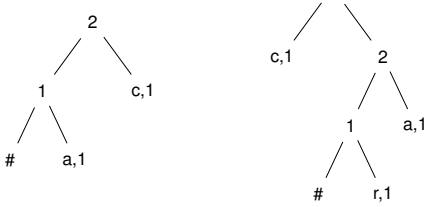
SORTIE : Texte compressé TC

1. H = feuille spéciale #
2. Soit s le symbole suivant de T
   - Si $s \in H$, alors transmettre le code de s dans H
     Sinon transmettre le code de # dans H puis le code initial de s
   - Modifier H : `H=Modification(H, s)`
3. Recommencer l'étape 2 tant que T n'est pas vide

Complexité $O(nN)$

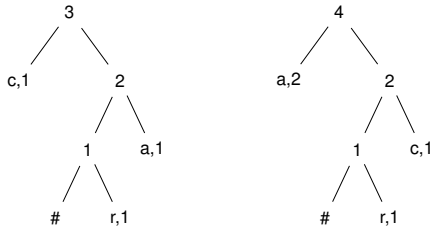
Huffman dynamique par l'exemple

carambarbcm **ca**rambarbcm



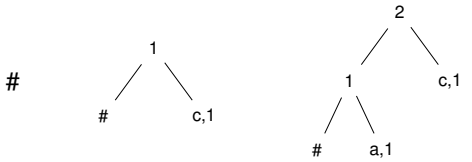
Huffman dynamique par l'exemple

carambarbcm carambarbcm **car**ambarbcm



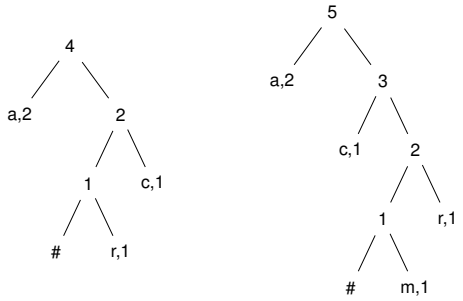
Huffman dynamique par l'exemple

carambarbcm **c**arambarbcm **ca**rambarbcm



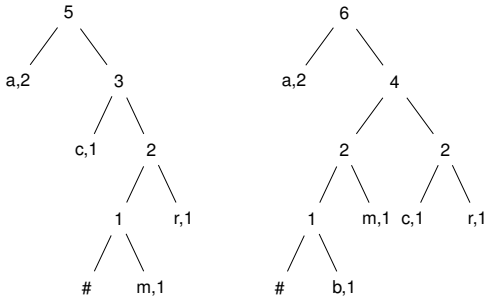
Huffman dynamique par l'exemple

carambarbcm **cara**mbarbcm



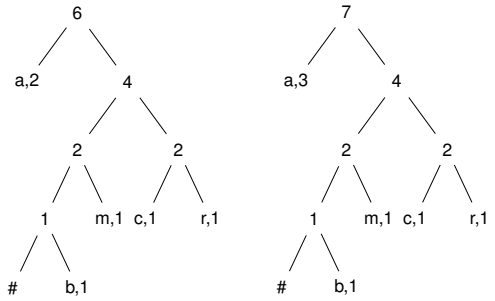
Huffman dynamique par l'exemple

carambarbcm **caram**barbcm



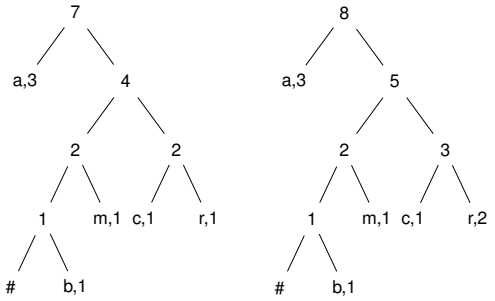
Huffman dynamique par l'exemple

carambarbcm **caramb**arbcm



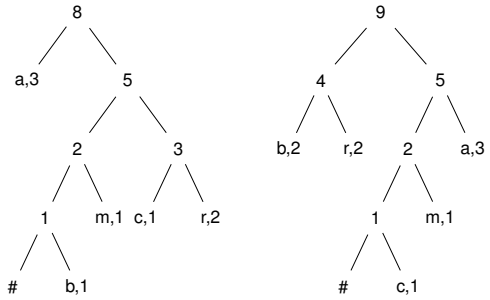
Huffman dynamique par l'exemple

carambarbcm **caramba**rbcm



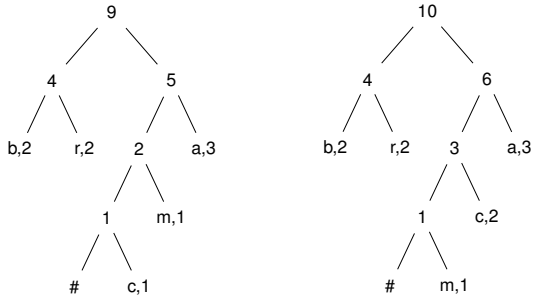
Huffman dynamique par l'exemple

carambarbcm **carambar**bcm



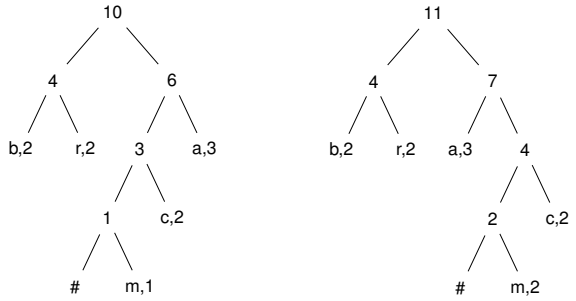
Huffman dynamique par l'exemple

carambarbcm **carambarb**cm



Huffman dynamique par l'exemple

carambarbcm **carambarbc**m



Algorithme de décompression

ENTRÉE : Texte compressé TC (et connaît codes fixes des caractères)

SORTIE : Texte décompressé T

1. H = #, décoder k bits $\rightarrow$ s, et `H=Modification(H,s)`
2. Lire les bits de TC en suivant H à partir de sa racine $\rightarrow$ feuille
   - Si c'est la feuille # alors lire les k bits suivant $\rightarrow$ s
     Sinon $\rightarrow$ caractère s de la feuille
   - `H=Modification(H,s)`
3. Recommencer l'étape 2 tant que TC n'est pas vide

Complexité $O(nN)$

Conclusions sur les algorithmes de Huffman

- Huffman Dynamique ne calcule pas les fréquences
- Huffman Dynamique “à la volée” :
  la décompression peut commencer avant que compression terminée
- Codage Huffman Statique minimal pour $T \leftrightarrow (f_i)$, mais doit transmettre l'arbre
  $\Rightarrow$ augmente taille de l'encodage
- Résultats expérimentaux sur données homogènes (complexité temporelle) :
  Huffman Dynamique légèrement meilleur que Huffman Statique pour petits fichiers, et bien meilleur pour gros fichiers.

Algorithmique Avancée

Antoine Genitrini

Antoine.Genitrini@Sorbonne-Universite.fr

Master Informatique 1

Moodle : UE MU4IN500

Année 2022-2023

Compression par Dictionnaires

- Principes de la compression par dictionnaire
- Dictionnaire Statique
- Dictionnaire Adaptatif
  - LZ77-LZSS
  - LZW : phase stationnaire et phase adaptative

Compression avec dictionnaire

Principe : groupe de symboles codé par index dans dictionnaire $C : A^* \to \{0,1\}^*$

1. Dictionnaire statique
   Dictionnaire de référence, partagé par compresseur et décompresseur
2. Dictionnaire adaptatif
   Le dictionnaire est “défini” par le fichier à compresser : apprentissage dynamique des répétitions
   Adaptativité inversible (fonction de décompression)

   J. Ziv and A. Lempel *"A universal algorithm for sequential data compression"* IEEE Transactions on Information Theory May 1977

Dictionnaire statique

- Codes postaux (corpus particulier – méthodes ad hoc)
- Dictionnaire de référence : Petit Robert junior illustré 2017
  Pages $< 2^{11}$, 2 Colonnes, Mots par colonnes $< 2^4$
  Études à l'Université $\rightarrow 406/1/10//1/1/1//613/1/1//1135/2/4$
  21 caractères (21*8=168 bits) compressés en $4 * (11 + 1 + 4) = 64$ bits
- Dictionnaire par taille de mot : en français taille des mots entre 1 et 25
  On connaît distribution des mots selon leur taille (9000 mots de 6 lettres)
  9000 à comparer aux $26^6 > 100$ millions ($\sim 2^{30}$) de possibilités
  $\rightarrow$ codage sur 14 bits par mot (+ 5 bits pour taille) au lieu de 30 bits *

Limitations et inconvénients : dictionnaire figé ; rendre compte des conjugaisons diverses, exceptions ....

Dictionnaire adaptatif LZ77-LZSS

LZ77-LZSS : Fenêtre fixe ou coulissante sur le fichier
Ex : AIDE-TOI-LE-CIEL-T-AIDERA 25car sur 8 bits = 200 bits
1A1I1D1E1-1T1O1I1-1L0(3,2)1C1I1E1L0(4,2)1-0(0,4)1R1A
20 bits + 17car sur 8 bits + 3couples(*pos*, *long*) sur 5+2 bits = 177 bits
- fenêtre de taille $2^k \Rightarrow k$ bits pour *pos* ;
- *long* sur *n* bits $\Rightarrow$ on peut compresser séquences de 2^n car.

Difficultés :

- gestion et représentation de la fenêtre coulissante
- compression : rechercher, en toutes positions, la plus longue séquence de la fenêtre pour coder la suite du texte. Décompression pas de pb !
- complexité : *tailleFen* $\times$ *longMax* si fenêtre structurée séquentiellement
- complexité : $\log_2(\textit{tailleFen}) \times \textit{longMax}$ si fenêtre structurée en arbre (suffixe). On peut alors doubler taille fenêtre sans augmenter trop le temps de recherche.

LZ78- LZW

LZ78- LZW : Dictionnaire = Ensemble (Hachage-Arbre de type 2-trie) de toutes les séquences déjà rencontrées (potentiellement illimité).
Méthodes adaptatives : phase transitoire (gagne en compression par l'apprentissage) puis stationnaire (ne sert à rien de continuer à adapter).

Algorithme en 2 phases :

1. Phase adaptative : construit un dictionnaire tout en commençant le codage
2. Phase stationnaire : le dictionnaire n'évolue plus, on l'utilise pour poursuivre le codage

Le compresseur et le décompresseur construisent le même dictionnaire !
Dictionnaire = table associative,
La fonction de hachage $h : A^+ \rightarrow [0, \ldots, m-1]$ *est connue par le compresseur et le décompresseur.*

Phase stationnaire

Définition : un ensemble *F* de mots est dit *préfixiel* ssi lorsque $f \in F$, alors tous les préfixes de *f* sont dans *F*.
Le dictionnaire de LZW est un ensemble préfixiel *F* de séquences du texte *T* à coder. Chaque $f \in F$ est codé par son index (numéro) dans la table.

En phase stationnaire (Le dictionnaire est connu des 2 cotés)

- La compression consiste à déterminer le plus long préfixe de *T* qui est dans le dictionnaire *F* et transmettre son index dans *F*,
  et recommencer sur la suite de *T*.
- Pour la décompression il suffit de remplacer chaque index reçu par son contenu dans la table.

  Exemple : T = ...ababcbababaaaaaaabbabaabca

Exemple

index	*f*	*(parent, lettre)*
1	a	
2	b	
3	c	
4	ab	1b
5	ba	2a
6	abc	4c
7	cb	3b
8	bab	5b
9	baba	8a
10	aa	1a
11	aaa	10a

T= ... ababcbababaaaaaaabbabaabca

Structures de données

$T =$	...	ab	abc	baba	ba	aaa	aaa	b	baba	abc	a
$C(T) =$		4	6	9	5	11	11	2	9	6	1

Fonction de hachage (collisions résolues) :
$h(a) = 1, h(b) = 2, h(c) = 3, h(ab) = 4, h(ba) = 5, h(abc) = 6,$
$h(cb) = 7, h(bab) = 8, h(baba) = 9, h(aa) = 10, h(aaa) = 11.$

1. À la compression,
   - pour reconnaître le plus long préfixe de T dans F : **Arbre de type 2-trie**,
   - stocker dans la table la valeur du parent (index) et la dernière lettre (gain de place)
2. À la décompression
   - la table est suffisante (pas besoin d'arbre digital)
   - décoder le contenu (parent, lettre)

Phase adaptative

Au départ la table F contient les lettres de A.

1. Le compresseur
   - détermine le plus long préfixe f de T dans F
   - transmet l'index de f dans F
   - ajoute dans F la séquence fx (où x est le caractère suivant dans T)
   - recommence à lire T à partir de x
2. Le décompresseur, lit le premier numéro et renvoie la lettre, puis
   - lit un numéro i et consulte la table à cet index
   - s'il y a un contenu, le décode en w_i, puis ajoute à la table (par hachage) une entrée formée de w_{i-1} (ou son index), suivi de la première lettre de w_i
   - sinon ajoute à cet endroit w_{i-1} (ou son index), suivi de **sa** première lettre. Cas de **chevauchement** : $w_{i-1} = bw$ et $w_i = bwb$.

Exemple

Initialement la table contient les lettres : $h(a) = 1, h(b) = 2, h(c) = 3$
$T = ababcbababaaaaaaa$

$C(T)$ = 1 2 4 3 5 8 1 10 11

Fonction de hachage (collisions résolues) :

$h(ab) = 4, h(ba) = 5, h(abc) = 6, h(cb) = 7,$

$h(bab) = 8, h(baba) = 9, h(aa) = 10, h(aaa) = 11$

- Compression
- Décompression

Statistique vs. Dictionnaire

- Dictionnaire code les redondances de séquences
- apprentissage des répétitions $\neq$ méthodes statistiques d'ordre supérieur.
- Exemple : fichier uniforme contenant 1000 fois le même caractère C (on suppose seules les 26 lettres majuscules sont codables -> 5 bits par lettre)
  - Huffman : C codé sur 1 bit $\Rightarrow$ 1000 bits,
  - LZW : À chaque étape ajoute un préfixe contenant un C de plus :
    $1 + \ldots + k = 1000 \Rightarrow k \sim 45$ adresses
    $\Rightarrow$ 45 * 7 bits $=$ 315 bits
    (il faut 7 bits par adresse car 26 lettres + 45 adresses)

Statistique vs. Dictionnaire

A l'inverse,
il existe des cas où la compression statistique est mieux adaptée que LZW :
Fichier d'octets $\rightarrow$ 8 fichiers binaires (le premier formé de tous les premiers bits, le deuxième formé de tous les deuxièmes bits,. . ..).
Compression des 8 fichiers par Huffman OK, mais pas par LZW, car en découpant en 8 "plans" on a pu casser des régularités.

Succession de deux compressions :

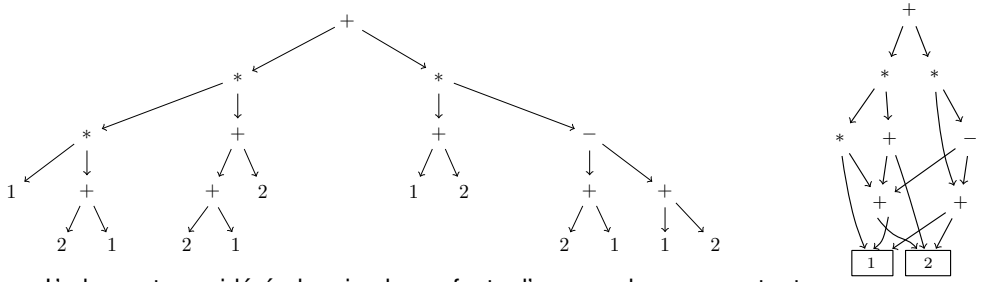
- Huffman suivi de LZW : améliore souvent la compression
- LZW suivi de Huffman : NON car le hachage de LZW "cache" les fréquences.

Compression Arborescente

Idée : les sous-arbres communs ne sont pas répétés, mais remplacés par un pointeur vers l'unique occurrence conservée.

En compilation ou en calcul symbolique, ce processus est

l'Élimination de sous-expression commune.



L'arbre est considéré plan, i.e. les enfants d'un nœud ne commutent pas.

Problème de l'isomorphisme d'arbres

On se place dans le contexte des
arbres plans enracinés d'arité quelconque.
Deux arbres T_1 et T_2 sont dits *isomorphes* s'ils sont identiques, c'est-à-dire : pour tout i, soient ν et μ les i-èmes nœuds de T_1 et T_2 via un parcours arbitraire, alors le contenu de ν et μ est identique et étant données les listes (ordonnées) des enfants de ν et de μ, chaque enfant de ν est isomorphe à son analogue pour μ.

Problème de l'isomorphisme d'arbres

Afin de réaliser le processus de compression de l'arbre, il faut être en mesure de reconnaître si deux (sous-)arbres sont isomorphes.

Détour combinatoire : mots de Lukasiewicz

On utilise un alphabet à deux caractères $\mathcal{L} = \{(,)\}$.
Un *mot de Lukasiewicz* est un mot vérifiant la grammaire suivante :

$$\mathcal{L} \quad = \quad \epsilon \quad | \quad (\,\mathcal{L}\,)\,\mathcal{L}.$$

Ainsi $\mathcal{L} \supset \{\epsilon,\ (),\ (()),\ ()(),\ ((())),\ (()()),\ (())(),\ ()(()),\ ()()(),\ \dots\}$.

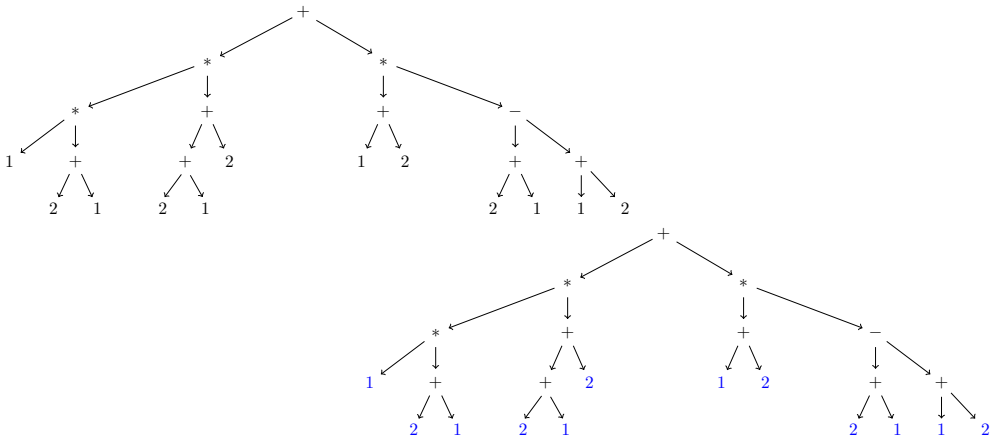
Lemme :

Les structures d'arbres enracinés plans d'arité quelconque sont en bijection avec les mots de Lukasiewicz, i.e. les mots bien parenthésés.

Théorème

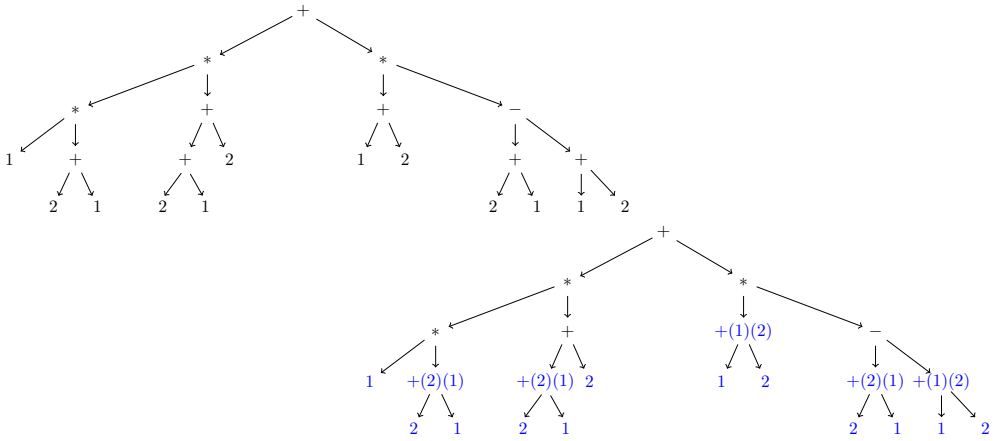
Deux structures d'arbres enracinés plans sont isomorphes si et seulement si ils sont associés au même mot de Lukasiewicz.

En ajoutant un peu d'information

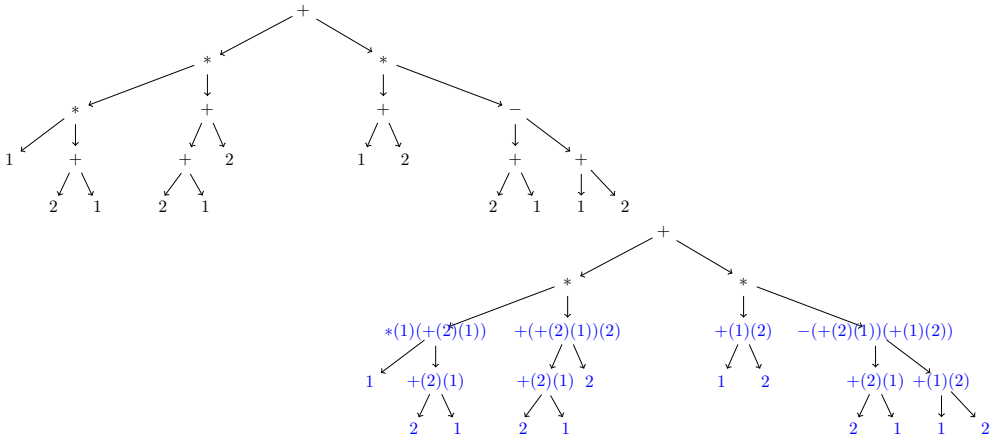


Rappel : les sous-arbres d'un nœud ne commutent pas.

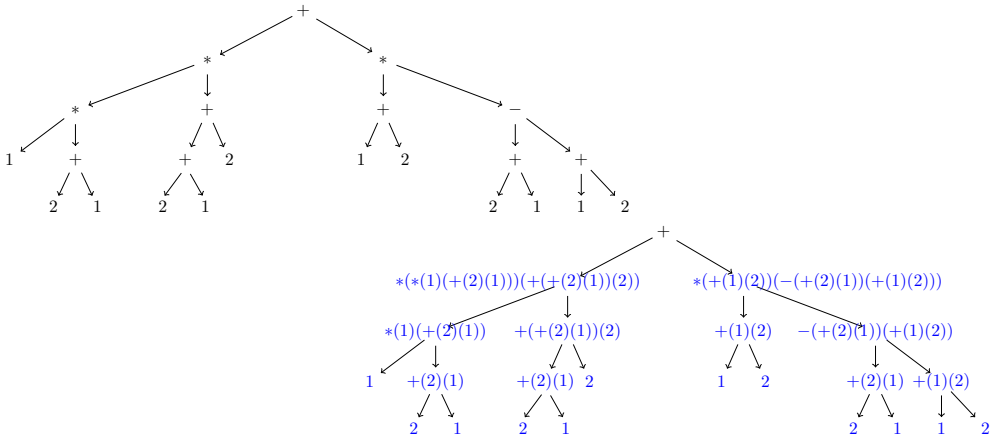
En ajoutant un peu d'information



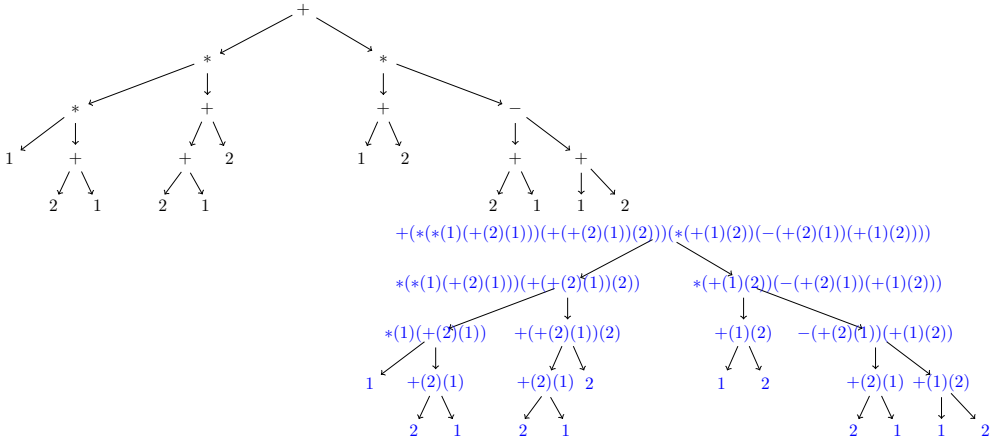
En ajoutant un peu d'information



En ajoutant un peu d'information



En ajoutant un peu d'information



Isomorphisme d'arbres

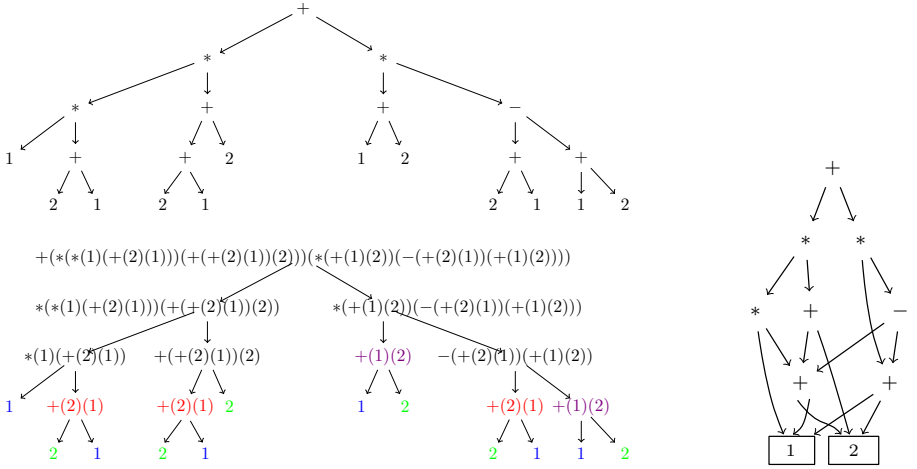
Théorème

Soient S et T deux arbres de même taille n.
L'algorithme pour tester si S et T sont isomorphes
a une complexité temporelle en $O(n)$.

Idées de la preuve :

- On construit les mots de Dyck associés à S et T en temps $O(n)$
  (les concaténations des mots des sous-arbres n'est pas faite, seule la concaténation à la racine est faite, et en supposant la taille des étiquette comme constante).
- Chacun des mots est de longueur $O(n)$.
- On teste l'égalité des mots de Dyck en $O(n)$.

Compression de l'exemple



Isomorphisme de sous-arbres

Théorème

Soit S et T deux arbres de tailles resp. s et n, vérifiant $s \leq n$.
L'algorithme de recherche des sous-arbres de T isomorphes à S a une complexité temporelle en $O(n)$.

Idées de la preuve :

- Pour être isomorphe à S, il faut avoir même hauteur et même taille que S.
- Soit k le nombre de sous-arbres de T de hauteurs et tailles identiques à celles de S.
  Ces sous-arbres ne partagent aucun nœud. Donc $k \cdot s \leq n$.
- Le cumul des complexités de tests pour chacun des k sous-arbres est $O(k \cdot s) = O(n)$.
- Attention à l'écriture en mémoire des mots de Lukasiewicz.

Compression d'arbres

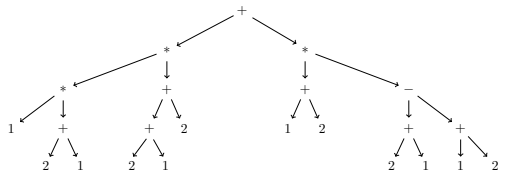
Approche naïve

Une approche naïve pour la compression d'un arbre T consiste à rechercher, pour chaque sous-arbre S de T, l'ensemble des sous-arbres de T isomorphes à S.
Cet algorithme a une complexité temporelle $O(n^2)$.

On applique pour chaque sous-arbre de T l'approche du transparent précédent.

Meilleure stratégie de compression

Soit T un arbre de taille n.



- à chaque valeur d'étiquette de feuille on associe un entier.

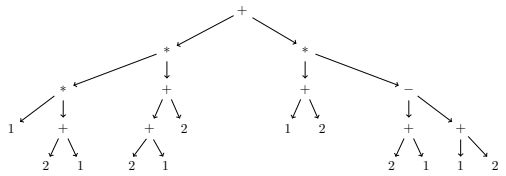
Via un parcours suffixe :

- chaque sous-structure reconnue est remplacée par sa valeur
- à chaque nouvelle sous-structure on associe une nouvelle valeur

Structure	Valeur
1	0
2	1
(+, 1, 0)	2
(*, 0, 2)	3
(+, 2, 1)	4
(*, 3, 4)	5
(+, 0, 1)	6
(-, 2, 6)	7
(*, 6, 7)	8
(+, 5, 8)	9

Meilleure stratégie de compression

Soit T un arbre de taille n.



- à chaque valeur d'étiquette de feuille on associe un entier.

Via un parcours suffixe :

- chaque sous-structure reconnue est remplacée par sa valeur
- à chaque nouvelle sous-structure on associe une nouvelle valeur

Structure	Valeur
1	0
2	1
(+, 1, 0)	2
(*, 0, 2)	3
(+, 2, 1)	4
(*, 3, 4)	5
(+, 0, 1)	6
(-, 2, 6)	7
(*, 6, 7)	8
(+, 5, 8)	9

Encodage des triplets

Définir l'association de chaque étiquette possible

Structure	Valeur
1	0
2	1
+	2
*	3
-	4

La première des composantes est inférieure à 5, les deux suivantes sont inférieures à n.
Le triplet (a, b, c) est encodé en $(a + b * 5 + c * 5 * n)$, ainsi

Structure	Encodage	Structure	Encodage
1	`0`	`(*, 3, 4)`	`478`
2	`1`	`(+, 0, 1)`	`117`
`(+, 1, 0)`	`2+1*5+0*5*23 = 7`	`(-, 2, 6)`	`704`
`(*, 0, 2)`	`3+0*5+2*5*23 = 233`	`(*, 6, 7)`	`838`
`(+, 2, 1)`	`127`	`(+, 5, 8)`	`947`

Reconnaissance des valeurs déjà vues

- À chaque création d'une nouvelle valeur on ajoute son encodage dans un ABR.
- Afin de savoir si une sous-structure a déjà été vue, on recherche son encodage dans l'ABR.

La structure compressée peut-être construite à la volée en ajoutant à la table d'association un pointeur dans la structure.

Théorème
Soit T un arbre de taille n.
L'algorithme de compression de T
a une complexité temporelle (espérée) en $O(n \log n)$.

Compression d'arbres

- approche naïve : $O(n^2)$ en temps,
  en chaque nœud on stocke un mot de Lukasiewicz :
  $O(n^2)$ en espace
- approche reconnaissance à la volée : $O(n \log n)$ en temps,
  une table d'association et un ABR : $O(n)$ en espace

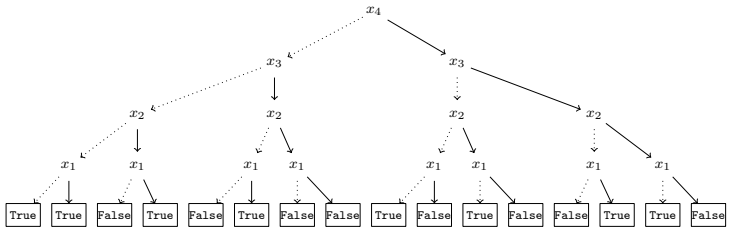
Approche par hachage

Une approche par hachage consiste à hacher une représentation de chaque sous-arbre (le mot de Lukasiewicz enrichi, ou l'encodage des triplets) de T. Puis, il suffit de savoir si une représentation est dans la table pour savoir si on a déjà vu le sous-arbre courant (complexité temporelle $O(1)$).
Cet algorithme a une complexité temporelle $O(n)$
mais nécessité de stocker d'une table de hachage
(en espace $O(n^2)$ en pratique).

Application pour les Fonctions Booléennes

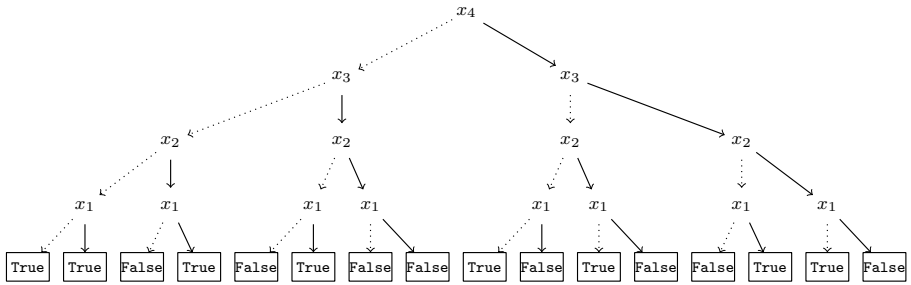
Une fonction Booléenne f sur k variables est une application de $\{\texttt{True}, \texttt{False}\}^k$ vers $\{\texttt{True}, \texttt{False}\}$.

Un arbre de décision binaire est une structure de données permettant d'évaluer une fonction f via un stratégie de type *diviser pour régner*.



L'arête en pointillés correspond à l'évaluation à `False` (variable source de l'arête) et l'arête solide correspond à l'évaluation à `True`.

Table de vérité et fonctions Booléennes



Autre représentation : la table de vérité.
Formalisme basé sur l'arbre de décision et décomposition récursive : La première moitié de la table correspond à l'affectation de la première variable (ici x_4) à `False` et la deuxième moitié à `True`.
On obtient ainsi

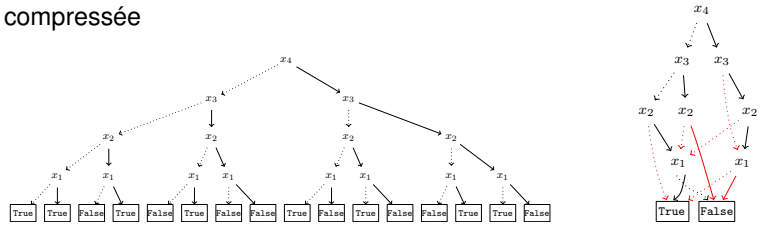
| T | T | F | T | F | T | F | F | T | F | T | F | F | T | T | F |

Utilisation des structures de décision

Les ROBDD (reduced ordered binary decision diagrams) sont issus de la compression des arbres de décision binaires (R. Bryant 1986).
Particulièrement utiles dans les contextes :

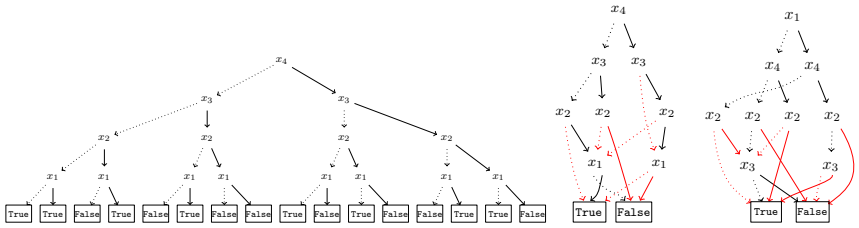
- conception assistée par ordinateur pour générer des circuits
- vérification formelle, model-checking
- optimisation combinatoire

L'un des avantages : les opérations sont effectuées directement sur la structure compressée



Ordre de variables

Un ordre pour les variables étant fixé, chaque fonction Booléenne admet un unique arbre de décision et donc un unique ROBDD.



Deux ordres distincts sur les variables peut induire des ROBDD de taille différente pour une même fonction.

Compression de l'arbre de décision

Via un parcours suffixe, soit ν le nœud actuellement visité :

- Si ν pointe deux fois vers le même enfant, alors ν disparaît et son parent pointe une unique fois vers cet enfant ;
- Sinon, si la sous-structure enracinée en ν (avec ses deux enfants déjà compressés) existe déjà dans la structure, alors le parent de ν pointe vers la première occurrence de la sous-structure et ν disparaît.

Remarque : un ROBDD est un DAG (graphe dirigé acyclique) vérifiant un certain nombre de contraintes.

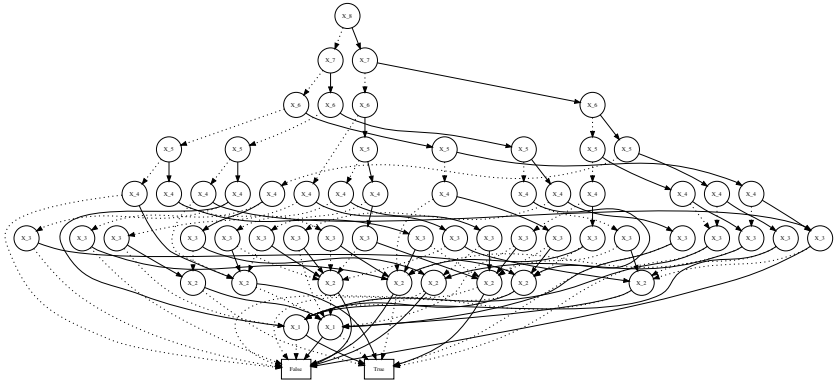
ROBDD optimal

Une fonction Booléenne f étant fixée, trouver l'ordre de variables tel que la taille du ROBDD de f soit la plus petite possible est un problème NP-difficile.

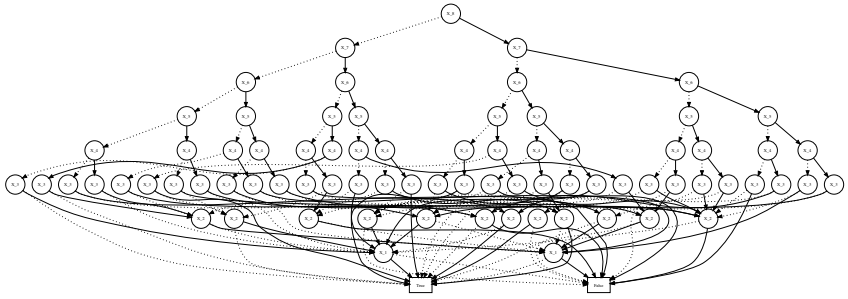
Alors que la manipulation à la main semble gérable, sur les petits arbres de décision, la compression devient très rapidement non gérable à la main...

ROBDD exemple : 8 variables et taille 62

Taille arbre de décision : $1 + 2 + \cdots + 2^7 + 2^8 = 2^9 - 1 = 511$
Taille table de vérité : $2^8 = 256$



ROBDD exemple : 8 variables et taille 77



Synthèse de ROBDD

Soient f et f' deux fonctions sur k variables et B et B' leurs ROBDD.
But : Construire le ROBDD de g, combinaison de f et f'.
Par exemple $g = f \wedge f'$ ou $g = f$ `xor` f'.

C'est la méthode la plus utilisée pour construire les ROBDD de fonctions complexes.

Construction du ROBDD de g en 3 étapes :

- Fusion
- Simplification
- Réduction

L'étape de réduction consiste en une compression, mais est plus efficace que la compression de l'arbre de décision de g.

Fusion de ROBDD

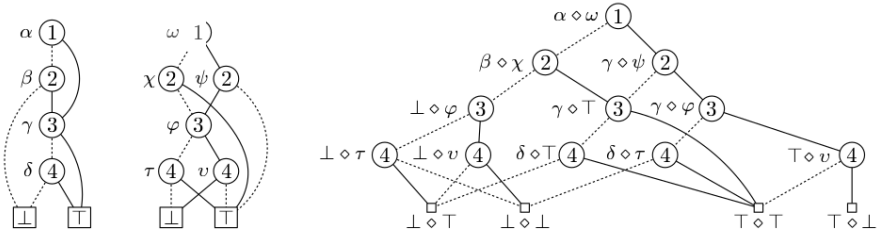
Soient B et B' les ROBDD de f et f'. Pour α, α' des nœuds resp. de B et B', on note $\alpha = (v, \ell, h)$ avec ℓ le pointeur lorsque la variable v est évaluée à `False` et h pour l'évaluation à `True`. De même pour $\alpha' = (v', \ell', h')$.

Si α et α' ne sont pas des puits du DAG, on a le DAG fusionné composé des nœuds :

$$\alpha \diamondsuit \alpha' = \begin{cases} (v, \ell \diamondsuit \ell', h \diamondsuit h') & \text{si } v = v' \\ (v, \ell \diamondsuit \alpha', h \diamondsuit \alpha') & \text{si } v < v' \\ (v', \alpha \diamondsuit \ell', \alpha \diamondsuit h') & \text{si } v > v'. \end{cases}$$

cf. D. Knuth : The art of computer science, vol. 4
Attention, ici les variables sont ordonnées dans le sens inverse.

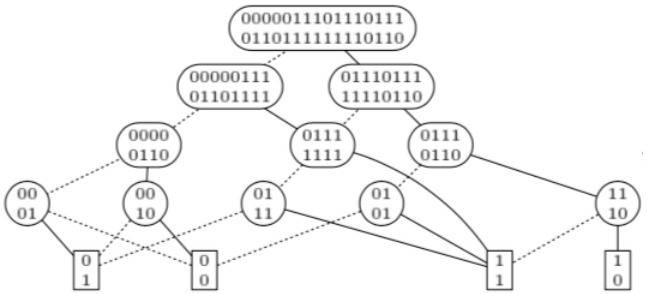
Exemple de fusion



cf. D. Knuth : The art of computer science, vol. 4

Simplification et Compression

Calcul en chaque nœud $a \diamondsuit b$ des tables de vérité de a et b
(dans ce nouveau DAG, par concaténations successives) :



Simplification via l'opérateur Booléen $\diamondsuit$ combinant f et f' ($\wedge$, `xor`, ...).
Puis compression des nœuds identiques.