Chapitre 6
Programmation dynamique

Matthieu Willems
Département d'informatique
UQÀM

Notes de cours basées sur le matériel pédagogique fourni par Louise Laforest

Rappel : Principe de diviser pour régner

- Diviser :

Décomposer le problème principal en sous-problèmes

- Régner :

Résoudre individuellement chacun des sous-problèmes

- Combiner :

Fusionner l'ensemble des résultats obtenus pour chacun des sous-problèmes afin de déterminer le résultat du problème initial

La solution au problème original s'obtient de façon descendante (*top down*).

Problème

- Si des sous-problèmes se chevauchent, la complexité temporelle peut devenir exponentielle.

- Exemples :
  - Les nombres de Fibonacci
  - Les coefficients binomiaux

Principe de la programmation dynamique

- Diviser :

On établit un schéma récursif comme dans la méthode « diviser pour régner ».

- Régner :

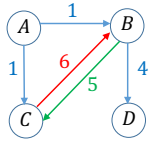
Les sous-problèmes sont résolus de façon ascendante (*bottom up*) sans récursivité.

- Des **tables** sont utilisées pour stocker les solutions des sous-problèmes.
- Pour les problèmes d'**optimisation**, les solutions des sous-problèmes doivent être contenues dans la solution du problème.

Principe d'optimalité

- Le **principe d'optimalité** s'applique lorsque la solution optimale d'un problème contient toujours les solutions optimales des sous-problèmes.

- Contre-exemple : Trouver le plus long chemin sans cycle entre deux sommets d'un graphe orienté.



Le chemin le plus long, sans cycle, de A à D est $A \to C \to B \to D$ de longueur 11. Le sous-chemin de la solution $A \to C$ a une longueur de 1 alors que le chemin le plus long de A à C est de longueur 6.

Coefficients binomiaux

- Définition : Soit $0 \leq k \leq n$ deux entiers, on note $\binom{n}{k}$ le nombre de sous-ensembles de cardinal k dans un ensemble de cardinal n.
- Exemples :
  - $\binom{n}{0} = 1$ pour tout entier $n \geq 0$
  - $\binom{n}{1} = n$ pour tout entier $n \geq 1$
  - $\binom{10}{3} = \frac{10 \cdot 9 \cdot 8}{3 \cdot 2 \cdot 1} = 120$
- Formule : $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ pour deux entiers $0 \leq k \leq n$
- Exemple : $\binom{50}{2} = \frac{50!}{2!(48)!} = 1225$
- Remarque :

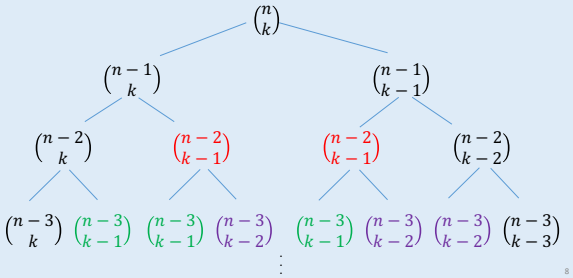
$50! = 30414093201713378043612608166064768844377641568960512000000000000$

Approche récursive

- $\binom{n}{k} = \begin{cases} 1 & \text{si } k = 0 \text{ ou } k = n \\ \binom{n-1}{k} + \binom{n-1}{k-1} & \text{si } n > k \geq 1 \end{cases}$
- Triangle de Pascal :

k / n	0	1	2	3	4	5	6
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
4	1	4	6	4	1		
5	1	5	10	10	5	1	
6	1	6	15	20	15	6	1

Dans une approche purement récursive, des valeurs sont calculées plusieurs fois



Approche dynamique

fonction binomial(n, k) **retourne** entier
- entrées :
  - $0 \leq k \leq n$ deux entiers
- sortie :
  - $B[n][k] = \binom{n}{k}$: un entier

début
1 $B[0][0] \longleftarrow 1$; $B[n][0] \longleftarrow 1$; $B[n][n] \longleftarrow 1$
2 **pour** $i \longleftarrow 1$ **haut** $n - 1$ **faire**
3 $B[i][0] \longleftarrow 1$; $B[i][i] \longleftarrow 1$
4 **pour** $j \longleftarrow 1$ **haut** $i - 1$ **faire**
5 $B[i][j] \longleftarrow B[i-1][j-1] + B[i-1][j]$
6 **fin pour**
7 **fin pour**
8 **pour** $j \longleftarrow 1$ **haut** $\min(n-1, k)$ **faire**
9 $B[n][j] \longleftarrow B[n-1][j-1] + B[n-1][j]$
10 **fin pour**
11 **retourner** $B[n][k]$
fin binomial

Analyse dans tous les cas

Complexité temporelle :
$T(n) = \Theta(n^2)$ (la taille du triangle de Pascal).

Complexité spatiale :
$\Theta(n^2)$ mais on peut facilement rendre cette complexité linéaire.

Multiplication chaînée de matrices

- Formule pour multiplier deux matrices $A_{m\times n}$ et $B_{n\times p}$:

$$C_{m\times p} = A_{m\times n}\times B_{n\times p} \qquad c_{i,j} = \sum_{k=1}^{n} a_{i,k}b_{k,j}$$

- Nombre de multiplications scalaires : mnp
- Si on veut multiplier trois matrices ou plus, comment choisir un parenthésage qui minimise le nombre de multiplications (le produit matriciel étant associatif mais pas commutatif).

Exemple

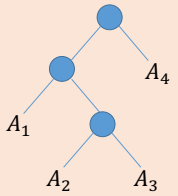
- $A_{5\times 7}\times B_{7\times 9}\times C_{9\times 25}$
  - $(A_{5\times 7}B_{7\times 9})C_{9\times 25} = D_{5\times 9}\times C_{9\times 25}$: $5\times 7\times 9 + 5\times 9\times 25 = 1440$ multiplications
  - $A_{5\times 7}(B_{7\times 9}C_{9\times 25}) = A_{5\times 7}\times E_{7\times 25}$: $7\times 9\times 25 + 5\times 7\times 25 = 2450$ multiplications
- Problème général : comment doit-on placer les parenthèses pour minimiser le nombre de multiplications scalaires dans un produit matriciel $A_1A_2 \dots A_n$?

Algorithme naïf

Générer tous les parenthésages possibles et prendre celui qui donne le minimum de multiplications scalaires.

Exemple avec 4 matrices :

1. $((A_1A_2)A_3)A_4$
2. $(A_1(A_2A_3))A_4$
3. $(A_1A_2)(A_3A_4)$
4. $A_1((A_2A_3)A_4)$
5. $A_1(A_2(A_3A_4))$



$(A_1(A_2A_3))A_4$

$A_{1(13\times 5)}A_{2(5\times 89)}A_{3(89\times 3)}A_{4(3\times 34)}$

Arbre	Parenthésage	Nombre de multiplications
	$((A_1A_2)A_3)A_4$	A_1A_2 : 13×5×89 = 5785 $(A_1A_2)A_3$: 5785 + 13×89×3 = 9256 $((A_1A_2)A_3)A_4$: 9256 + 13×3×34 = **10582**
	$(A_1(A_2A_3))A_4$	A_2A_3 : 5×89×3 = 1335 $A_1(A_2A_3)$: 1335 + 13×5×3 = 1530 $(A_1(A_2A_3))A_4$: 1530 + 13×3×34 = **2856**
	$(A_1A_2)(A_3A_4)$	A_1A_2 : 13×5×89 = 5785 A_3A_4 : 85×3×34 = 9078 $(A_1A_2)(A_3A_4)$ = 5785 + 9078 + 13×89×34 = **54201**
	$A_1((A_2A_3)A_4)$	A_2A_3 : 5×89×3 = 1335 $(A_2A_3)A_4$: 1335 + 5×3×34 = 1845 $A_1((A_2A_3)A_4)$: 1845 + 13×5×34 = **4055**
	$A_1(A_2(A_3A_4))$	A_3A_4 : 89×3×34 = 9078 $A_2(A_3A_4)$: 9078 + 5×89×34 = 24208 $A_1(A_2(A_3A_4))$: 24208 + 13×5×34 = **26418**

Nombres de parenthésages

- L'arbre binaire qui représente le parenthésage des n matrices contient n feuilles.
- Le nombre de parenthésages possibles de n matrices est égal au nombre d'arbres binaires de n feuilles.
- $C(n)$: nombres de Catalan : nombre d'arbres binaires de $n+1$ feuilles
- $C_0 = 1$, $C_1 = 1$ et on a la relation de récurrence suivante pour tout $n \geq 1$:

$$C_n = \sum_{i=0}^{n-1} C(i)C(n-1-i)$$

On peut démontrer par récurrence (exercice) que pour tout entier $n \geq 1$,
$C(n) = \frac{1}{n+1}\binom{2n}{n} = \Theta\left(\frac{4^n}{n^{\frac{3}{2}}}\right)$. Cet équivalent peut être déduit de la formule de Stirling.

n	0	1	2	3	4	5	6	7	8	9	14
$C(n)$	1	1	2	5	14	42	132	429	1430	4862	2674440

Approche « diviser pour régner »

$2(n-1)$ appels récursifs pour chaque position de la frontière Fr : position de la parenthèse qui « coupe en deux » le produit

$((A_1A_2)A_3)((A_4A_5)A_6)$ $\quad Fr = 3$
$((A_1A_2)A_3)(A_4(A_5A_6))$ $\quad Fr = 3$
$((A_1A_2)(A_3A_4))(A_5A_6)$ $\quad Fr = 4$
$(A_1A_2)\left(A_3(A_4(A_5A_6))\right)$ $\quad Fr = 2$
$A_1\left(A_2((A_3A_4)(A_5A_6))\right)$ $\quad Fr = 1$

Approche dynamique

Les solutions des sous-problèmes seront placées dans une table M où :
- $\mathrm{m}_{i,j}$: nombre minimal de multiplications pour $A_iA_{i+1}\dots A_j$, $1 \leq i < j \leq n$
- $\mathrm{m}_{i,i} = 0$ pour $1 \leq i \leq n$
- $\mathrm{m}_{i,j} = 0$ pour $1 \leq j < i \leq n$

Notations : La matrice A_i est de taille $p[i-1]\times p[i]$.

Calcul de $\mathrm{m}_{i,j}$ pour $1 \leq i < j \leq n$

$$A_iA_{i+1}\dots A_j = (\underbrace{A_iA_{i+1}\dots A_k}_{\mathrm{m}_{i,k}})(\underbrace{A_{k+1}A_{k+2}\dots A_j}_{\mathrm{m}_{k+1,j}}) \quad i \leq k < j$$

- $A_iA_{i+1}\dots A_k$ est de dimension $p[i-1]\times p[k]$
- $A_{k+1}A_{k+2}\dots A_j$ est de dimension $p[k]\times p[j]$
- Le calcul de $(A_iA_{i+1}\dots A_k)(A_{k+1}A_{k+2}\dots A_j)$ nécessite $p[i-1]p[k]p[j]$ multiplications scalaires.

$$\mathrm{m}_{i,j} = \min_{i\leq k<j}\left(\mathrm{m}_{i,k} + \mathrm{m}_{k+1,j} + p[i-1]p[k]p[j]\right)$$

Représentation

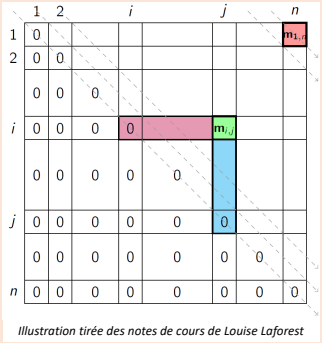


Illustration tirée des notes de cours de Louise Laforest

$c - l = n - 1$

$c - l = n - 2$

$$m_{i,j} = \min_{i \le k < j} \left(m_{i,k} + m_{k+1,j} + p[i-1]p[k]p[j] \right)$$

c : numéro de la colonne
l : numéro de la ligne

$c - l = 2$

$c - l = 1$

$c - l = 0$

Principes de l'algorithme

- Construire la table M diagonale par diagonale.
- Commencer par la plus longue diagonale.
- Éléments de la diagonale s : solutions optimales pour la suite de matrices consécutives $A_i A_{i+1} \dots A_j$ où $j - i = s$.

procédure trouverParenthésageOptimal($n; p;$ m; $frontiere$)

- entrées :
  - n : entier positif
  - p : vecteur d'entiers positifs indicé de 0 à n
- sorties :
  - m : matrice d'entiers positifs $n \times n$
  - $frontiere$: matrice d'entiers positifs $n \times n$


```
début
1     pour i ⟵ 1 haut n faire
2           m[i ; i ] ⟵ 0
3     fin pour
4     pour s ⟵ 1 haut n − 1 faire        s + 1 correspond au nombre de matrices à multiplier = j − i + 1
5           pour i ⟵ 1 haut n − s faire
6                 j ⟵ i + s              calcul de m[i; j] correspondant au produit des s + 1 matrices : A_i A_{i+1} ... A_j
7                 minimum ⟵ +∞
8                 pour k ⟵ i haut j − 1 faire
9                       temp ⟵ m[i; k] + m[k + 1; j] + p[i − 1]p[k]p[j ]
10                      si temp < minimum alors
11                            minimum ⟵ temp
12                            frontiereTemp ⟵ k
13                      fin si
14                fin pour
15                m[i; j] ⟵ minimum
16                frontiere[i; j] ⟵ frontiereTemp
17          fin pour
18    fin pour
fin trouverParenthésageOptimal
```


$$A_{1(13\times5)} A_{2(5\times89)} A_{3(89\times3)} A_{4(3\times34)}$$

Matrice m

$$\begin{pmatrix} 0 & 5785 & 1530 & 2856 \\ & 0 & 1335 & 1845 \\ & & 0 & 9078 \\ & & & 0 \end{pmatrix}$$

Matrice $frontiere$

$$\begin{pmatrix} & 1 & 1 & 3 \\ & & 2 & 3 \\ & & & 3 \end{pmatrix}$$

Analyse dans tous les cas

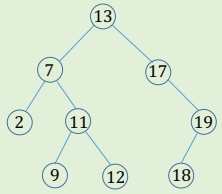
$T(n)$: nombres de comparaisons effectuées à la ligne 10.

$$T(n) = \sum_{s=1}^{n-1} \sum_{i=1}^{n-s} \sum_{k=i}^{j-1} 1 = \sum_{s=1}^{n-1} \sum_{i=1}^{n-s} (j - i) = \sum_{s=1}^{n-1} \sum_{i=1}^{n-s} s = \sum_{s=1}^{n-1} s(n-s)$$

$$T(n) = n \sum_{s=1}^{n-1} s - \sum_{s=1}^{n-1} s^2 = n \frac{(n-1)n}{2} - \frac{(n-1)n(2(n-1)+1)}{6} = \frac{1}{6} n^3 - \frac{1}{6} n$$

Arbre binaire de recherche

Définition : Un arbre binaire de recherche est un arbre binaire dans lequel chaque nœud est associé à une clé, tel que la clé de chaque nœud du sous-arbre gauche soit inférieure (ou égale) à celle du nœud considéré, et tel que la clé de chaque nœud du sous-arbre droit soit supérieure (ou égale) à celle-ci.



Arbre binaire de recherche optimal

Cas dynamique : On construit l'arbre au fur et à mesure de l'insertion des données. On peut utiliser un arbre AVL ou un arbre rouge-noir. La probabilité de recherche de chacune des clés est supposée équiprobable.

Cas statique : On connaît toutes les clés à insérer ainsi que leur probabilité de recherche (exemple : mots réservés en Java). On construit l'arbre en tenant compte des probabilités de recherche.

Problème : Trouver l'arbre binaire de recherche qui minimise le nombre moyen de comparaisons lors d'une recherche fructueuse dans le cas statique pour les clés $x_1 < x_2 < \cdots < x_n$.

Notations

$C_{1,n}$: Nombre moyen de comparaisons pour une recherche fructueuse pour les clés $x_1 < x_2 < \cdots < x_n$

d_i : Profondeur (niveau) de x_i dans l'arbre

p_i : Probabilité de chercher x_i

On a :

$$C_{1,n} = \sum_{i=1}^{n} p_i \, (d_i + 1) \qquad \sum_{i=1}^{n} p_i = 1$$

Le problème revient à trouver les valeurs d_i telles que $C_{1,n}$ est minimal.

Exemples

a < b < c							
$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$	2	2	1,67	2	2
0,25	0,3	0,45	1,8	1,85	1,7	2,05	2,2
0,35	0,2	0,45	1,9	1,75	1,8	1,85	2,1

Cas équiprobable

$p_i = \frac{1}{n}$ pour tout entier $1 \le i \le n$. On a alors :

$$C_{1,n} = \sum_{i=1}^{n} \frac{1}{n}(d_i + 1) = \frac{1}{n}\sum_{i=1}^{n}(d_i + 1) = 1 + \frac{1}{n}\sum_{i=1}^{n} d_i$$

- On peut démontrer que l'arbre optimal est alors l'arbre de hauteur minimale. Cette hauteur minimale est égale à $p = \lfloor \lg(n) \rfloor$, c'est-à-dire à l'entier p, tel que
$$2^{p-1} \le n < 2^p.$$
- Tous les niveaux sont alors remplis sauf possiblement le dernier.
- Le nombre de nœuds au niveau $k \le p$ dans cet arbre est 2^k. La valeur de d_i pour ces nœuds est 2^k.

$$\sum_{k=0}^{p-1} k2^k \le \sum_{i=1}^{n} d_i \le \sum_{k=0}^{p} k2^k$$

Calcul exact des deux sommes

$$\sum_{k=0}^{p} x^k = \frac{x^{p+1} - 1}{x - 1}$$

On dérive des deux côtés :

$$\sum_{k=0}^{p} kx^{k-1} = \frac{(p+1)x^p(x-1) - (x^{p+1} - 1)}{(x-1)^2} = \frac{px^{p+1} - (p+1)x^p + 1}{(x-1)^2}$$

On multiplie par x des deux côtés :

$$\sum_{k=0}^{p} kx^k = x\frac{px^{p+1} - (p+1)x^p + 1}{(x-1)^2}$$

On remplace x par 2 :

$$\sum_{k=0}^{p} k2^k = 2(p2^{p+1} - (p+1)2^p + 1)$$

$$2\big((p-1)2^p - p2^{p-1} + 1\big) \le \sum_{i=1}^{n} d_i \le 2(p2^{p+1} - (p+1)2^p + 1)$$

$$2\big((p-1)2^p - p2^{p-1}\big) \le \sum_{i=1}^{n} d_i \le 2(p2^{p+1})$$

$$2^p\,(p-2) \le \sum_{i=1}^{n} d_i \le p2^{p+2}$$

Comme $p = \lfloor \lg(n) \rfloor$, on obtient :

$$n(\lfloor \lg(n) \rfloor - 2) \le \sum_{i=1}^{n} d_i \le 8n\lfloor \lg(n) \rfloor$$

On divise par n :

$$\lfloor \lg(n) \rfloor - 2 \le \frac{1}{n}\sum_{i=1}^{n} d_i \le 8\lfloor \lg(n) \rfloor$$

Cas non équiprobable

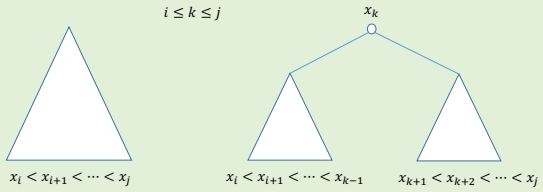
- **Algorithme 1** :

Générer tous les arbres possibles et prendre celui qui donne la valeur minimale pour $C_{1,n}$.

- **Analyse** : Pour chacun des $\frac{1}{n+1}\binom{2n}{n}$ arbres possibles, on calcule $C_{1,n}$, ce qui nécessite $\Theta(n)$ opérations. On aura donc une complexité temporelle totale de $T(n) = \Theta\left(\frac{4^n}{\sqrt{n}}\right)$.

Algorithme 2 : approche dynamique

Schéma récursif : On note $C_{i,j}$ le nombre moyen de comparaisons pour une recherche fructueuse pour les clés $x_i < x_{i+1} < \cdots < x_j$.



$$C_{i,j} = \sum_{l=i}^{j} p_l\,(d_l + 1) = \sum_{l=i}^{k-1} p_l\,(d_l + 1) + p_k(d_k + 1) + \sum_{l=k+1}^{j} p_l\,(d_l + 1)$$

$$C_{i,j} = \sum_{l=i}^{k-1} (p_l d_l + p_l) + p_k + \sum_{l=k+1}^{j} (p_l d_l + p_l)$$

$$C_{i,j} = \sum_{l=i}^{k-1} p_l\,(d_l^* + 1) + \sum_{l=i}^{k-1} p_l + p_k + \sum_{l=k+1}^{j} p_l\,(d_l^* + 1) + \sum_{l=k+1}^{j} p_l$$

$$C_{i,j} = C_{i,k-1} + C_{k+1,j} + \sum_{l=i}^{j} p_l$$

Formule récursive pour $C_{i,j}$

$$C_{i,j} = \begin{cases} p_i & \text{si } i = j \\ \sum_{l=i}^{j} p_l + \min_{i \le k \le j}\{C_{i,k-1} + C_{k+1,j}\} & \text{si } i < j \end{cases}$$

Représentation

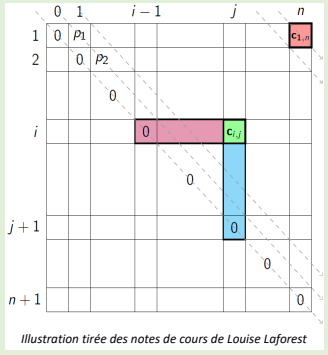


Illustration tirée des notes de cours de Louise Laforest

$c - l = n - 1$
$c - l = n - 2$

$$c_{i,j} = \sum_{l=i}^{j} p_l + \min_{i \le k \le j}\{c_{i,k-1} + c_{k+1,j}\}$$

c : numéro de la colonne
l : numéro de la ligne

$c - l = 1$
$c - l = 0$
$c - l = -1$

Algorithme 2

procédure trouverArbreOptimal$(p, r, C, racine)$

- entrées :
  - p : vecteur de probabilités indicé de 1 à n
  - $r = \{r_{i,j}\}_{1 \le i \le n}$: matrice telle que $r_{i,j} = \sum_{l=i}^{j} p_l$
- sorties :
  - $C = \{c_{i,j}\}_{1 \le i \le n+1}$: matrice donnant les espérances minimales des temps de recherche
  - $racine$: matrice permettant de trouver l'arbre optimal

début

1 **pour** $i \longleftarrow 1$ **haut** n **faire**
2 $\quad c_{i,i} \longleftarrow p_i$
3 $\quad c_{i,i-1} \longleftarrow 0$
4 $\quad racine_{i,i} \longleftarrow i$
5 **fin pour**
6 $c_{n+1,n} \longleftarrow 0$

```
7     pour longueur ⟵ 2 haut n faire                    longueur = c − l
8         pour i ⟵ 1 haut n − (longueur − 1) faire
9             j ⟵ i + (longueur − 1)
10            min ⟵ +∞
11            pour k ⟵ i haut j faire
12                temp ⟵ c_{i,k−1} + c_{k+1,j}
13                si temp < min alors
14                    min ⟵ temp
15                    rac ⟵ k
16                fin si
17            fin pour
18            c_{i,j} ⟵ min + r_{i,j}
19            racine_{i,j} ⟵ rac
20        fin pour
21    fin pour
fin trouverArbreOptimal
```


Analyse dans tous les cas

$T(n)$: nombres de comparaisons effectuées à la ligne 13.

$$T(n) = \sum_{s=2}^{n} \sum_{i=1}^{n-s+1} \sum_{k=i}^{j} 1 = \sum_{s=2}^{n} \sum_{i=1}^{n-s+1} (j - i + 1) = \sum_{s=2}^{n} \sum_{i=1}^{n-s+1} s = \sum_{s=2}^{n} s(n - s + 1)$$

$T(n) = \frac{1}{6}n^3 + \frac{1}{2}n^2 - \frac{2}{3}n$ (même type de calcul qu'à la page 23).

Sous-suite commune

- Alphabet Σ (sigma) : Ensemble de symboles possibles.
- Deux suites

$X = (x_1, x_2, \dots, x_m),\ m \geq 0,\ x_i \in \Sigma,\ 1 \leq i \leq m$

$Y = (y_1, y_2, \dots, y_n),\ n \geq 0,\ y_i \in \Sigma,\ 1 \leq i \leq n$

Problème : Trouver une sous-suite commune à X et Y de longueur maximale :

$Z = (z_1, z_2, \dots, z_k),\ 0 \leq k \leq \min(m, n)$, où

$z_1 = x_{i_1} = y_{j_1}, z_2 = x_{i_2} = y_{j_2}, \dots, z_k = x_{i_k} = y_{j_k}$

$1 \leq i_1 < i_2 < \cdots < i_k \leq m$

$1 \leq j_1 < j_2 < \cdots < j_k \leq n$

Exemple

- $X = (A, B, C, B, D, A, B)$ et $Y = (B, D, C, A, B, A)$.
- Z possibles :

$()$

$(A), (B), (C), (D)$

$(A, A), (A, B), (B, A), (B, B), (B, C), (B, D), (C, A), (C, B), (D, A), (D, B)$

$(A, B, A), (B, A, B), (B, B, A), (B, C, A), (B, C, B), (B, D, A)$

$(B, D, B), (C, A, B), (C, B, A), (D, A, B)$

$(B, C, A, B), (B, C, B, A), (B, D, A, B)$

Notations

- Préfixes de longueur k de $X, 1 \leq k \leq m$: $X_k = (x_1, x_2, \dots, x_k)$
- $X_m = X$
- $X_0 = ()$ suite vide
- Exemple : $X = (A, B, C, B, D, A, B)$

$X_0 = ()$

$X_3 = (A, B, C)$

$X_6 = (A, B, C, B, D, A)$

Propriétés de la SCLM

Soient $X = (x_1, x_2, \dots, x_m),\ m \geq 0$ et $Y = (y_1, y_2, \dots, y_n),\ n \geq 0$, et soit $Z = (z_1, z_2, \dots, z_k)$ une SCLM.

si $x_m = y_n$ **alors**
  $z_k = x_m = y_n$ et Z_{k-1} est une SCLM de X_{m-1} et Y_{n-1}
sinon
  si $z_k \neq x_m$ **alors**
    Z est une SCLM de X_{m-1} et Y
  fin si
  si $z_k \neq y_n$ **alors**
    Z est une SCLM de X et Y_{n-1}
  fin si
fin si

Schéma récursif de la solution

$L_{i,j}$ longueur maximale de la SCLM pour les suites X_i et Y_j.

$$L_{i,j} = \begin{cases} 0 & \text{si } i = 0 \text{ ou } j = 0 \\ 1 + L_{i-1,j-1} & \text{si } i > 0 \text{ et } j > 0 \text{ et } x_i = y_j \\ \max(L_{i-1,j}; L_{i,j-1}) & \text{si } i > 0 \text{ et } j > 0 \text{ et } x_i \neq y_j \end{cases}$$

Solution pour X et Y : $L_{m,n}$

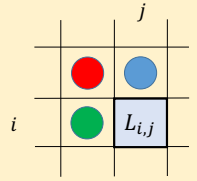
Algorithme 1

fonction longueurMax(x, y, i, j) **retourne** entier
- entrées :
  - x : suite de symboles numérotés de 1 à m
  - y : suite de symboles numérotés de 1 à n
  - i un entier entre 0 et m
  - j un entier entre 0 et n

début
1 **si** $i = 0$ **ou** $j = 0$ **alors**
2 $res \longleftarrow 0$
3 **sinon si** $x_i = y_j$ **alors**
4 $res \longleftarrow 1 +$ longueurMax$(x, y, i - 1, j - 1)$
5 **sinon**
6 $res \longleftarrow$ max(longueurMax $(x, y, i - 1, j)$;longueurMax $(x, y, i, j - 1)$)
7 **fin si**
8 **retourner** res
fin longueurMax

Algorithme 2 : approche dynamique

$$L_{i,j} = \begin{cases} 0 & \text{si } i = 0 \text{ ou } j = 0 \\ 1 + L_{i-1,j-1} & \text{si } i > 0 \text{ et } j > 0 \text{ et } x_i = y_j \\ \max(L_{i-1,j}; L_{i,j-1}) & \text{si } i > 0 \text{ et } j > 0 \text{ et } x_i \neq y_j \end{cases}$$



fonction TrouverLongueurMax(x, y) **retourne** entier
- entrées :
  - x : suite de symboles numérotés de 1 à m
  - y : suite de symboles numérotés de 1 à n

début
```
pour i ⟵ 1 haut m faire
        L[i ; 0 ] ⟵ 0
fin pour
pour j ⟵ 1 haut n faire
        L[0 ; j ] ⟵ 0
fin pour
pour i ⟵ 1 haut m faire
        pour j ⟵ 1 haut n faire
                si x_i = y_j alors
                        L[i ; j ] ⟵ L[i − 1; j − 1 ] + 1
                sinon si L[i − 1; j ] ≥ L[i ; j − 1 ] alors
                        L[i ; j ] ⟵ L[i − 1; j ]
                sinon
                        L[i ; j ] ⟵ L[i ; j − 1 ]
                fin si
        fin pour
fin pour
retourner L[m ; n ]
```
fin TrouverLongueurMax

fonction TrouverLongueurMaxPlus(x, y)
- entrées :
  - x : suite de symboles numérotés de 1 à m
  - y : suite de symboles numérotés de 1 à n
- sorties :
  - L : matrice des longueurs maximales
  - V : matrice des directions maximales

début
```
pour i ⟵ 1 haut m faire
        L[i ; 0 ] ⟵ 0 ; V[i ; 0 ] ⟵ ouest
fin pour
pour j ⟵ 1 haut n faire
        L[0 ; j ] ⟵ 0 ; V[0 ; j ] ⟵ ouest
fin pour
pour i ⟵ 1 haut m faire
        pour j ⟵ 1 haut n faire
                si x_i = y_j alors
                        L[i ; j ] ⟵ L[i − 1; j − 1 ] + 1; V[i ; j ] ⟵ nord − ouest
                sinon si L[i − 1; j ] ≥ L[i ; j − 1 ] alors
                        L[i ; j ] ⟵ L[i − 1; j ]; V[i ; j ] ⟵ nord
                sinon
                        L[i ; j ] ⟵ L[i ; j − 1 ]; V[i ; j ] ⟵ ouest
                fin si
        fin pour
fin pour
```
fin TrouverLongueurMaxPlus

fonction obtenirSCLM(V, x, i, j) **retourne** suite
- entrées :
  - V : matrice des directions
  - x : suite de symboles numérotés de 1 à m
  - i : position dans la suite de symboles
  - j : position dans la suite de symboles

début
```
si i = 0 ou j = 0 faire
        res ⟵ suite vide
sinon si V[i, j ] = nord − ouest alors
        res ⟵ obtenirSCLM(V, x, i − 1, j − 1 ) & x_i
sinon si V[i, j ] = nord alors
        res ⟵ obtenirSCLM(V, x, i − 1, j )
sinon
        res ⟵ obtenirSCLM(V, x, i, j − 1 )
fin si
retourner res
```
fin obtenirSCLM

& : concaténation

Exemple

Matrice L calculée **ligne par ligne**

X \ Y	0	a 1	x 2	y 3	b 4	s 5	u 6	c 7	d 8
0	0	0	0	0	0	0	0	0	0
x 1	0	0	1	1	1	1	1	1	1
a 2	0	1	1	1	1	1	1	1	1
b 3	0	1	1	1	2	2	2	2	2
z 4	0	1	1	1	2	2	2	2	2
c 5	0	1	1	1	2	2	2	3	3
u 6	0	1	1	1	2	2	3	3	3
s 7	0	1	1	1	2	3	3	3	3
d 8	0	1	1	1	2	3	3	3	4
x 9	0	1	2	2	2	3	3	3	4
i 10	0	1	2	2	2	3	3	3	4

0 ⟵ 1 ↑ 2 ↖

Matrice V calculée **ligne par ligne**

X \ Y	0	a 1	x 2	y 3	b 4	s 5	u 6	c 7	d 8
0	0	0	0	0	0	0	0	0	0
x 1	0	1	2	0	0	0	0	0	0
a 2	0	2	1	1	1	1	1	1	1
b 3	0	1	1	1	2	0	0	0	0
z 4	0	1	1	1	1	1	1	1	1
c 5	0	1	1	1	1	1	1	2	0
u 6	0	1	1	1	1	1	2	1	1
s 7	0	1	1	1	1	2	1	1	1
d 8	0	1	1	1	1	1	1	1	2
x 9	0	1	2	0	1	1	1	1	1
i 10	0	1	1	1	1	1	1	1	1

0 ⟵ 1 ↑ 2 ↖

SCLM déterminée en **remontant depuis le coin inférieur droit**

X \ Y	0	a 1	x 2	y 3	b 4	s 5	u 6	c 7	d 8
0	**0**	**0**	0	0	0	0	0	0	0
x 1	0	1	**2**	**0**	0	0	0	0	0
a 2	0	2	1	**1**	1	1	1	1	1
b 3	0	1	1	1	**2**	**0**	**0**	0	0
z 4	0	1	1	1	1	1	**1**	1	1
c 5	0	1	1	1	1	1	1	**2**	0
u 6	0	1	1	1	1	1	2	**1**	1
s 7	0	1	1	1	1	2	1	**1**	1
d 8	0	1	1	1	1	1	1	1	**2**
x 9	0	1	2	0	1	1	1	1	**1**
i 10	0	1	1	1	1	1	1	1	**1**

Distance d'édition (Distance de Levenshtein)

- <u>Définition</u> : Nombre d'opérations élémentaires requises pour transformer la première suite X en la deuxième Y

- <u>Opérations élémentaires</u> :
  - $Del(a)$: suppression (« délétion » en biologie) d'une lettre de X à une position donnée
  - $Ins(a)$: insertion d'une lettre de Y dans X à une position donnée
  - $Sub(a, b)$ substitution d'une lettre de X à une position donnée par une lettre de Y

- <u>Coût des opérations élémentaires</u> :
  - $Del(a) = Ins(a) = 1$
  - $Sub(a, b) = \delta_{a \neq b} = \begin{cases} 0 \text{ si } a = b \\ 1 \text{ si } a \neq b \end{cases}$ (symbole de Kronecker)

Exemple

"$plusieurs$" ⟶ "$malheur$"

X / Y	p	l	u	s	i	e	u	r	s
	m	a	l	h		e	u	r	
Opération	S	I	S	S	D	D	S	S	D
Coût	1	1	0	1	1	1	0	0	1

S = Sub I = Ins D = Del

Schéma récursif de la solution

- Alphabet : Σ
- Σ^* : toutes les suites possibles formées avec les symboles de Σ
- Suite vide : $\varepsilon \in \Sigma^*$
- $a \in \Sigma, b \in \Sigma, U \in \Sigma^*, V \in \Sigma^*$
- $Lev(U, V)$: distance minimale d'édition de U et V

Formule pour $d_{i,j}$

$$d_{i,j} = \begin{cases} 0 & \text{si } i = 0 \text{ et } j = 0 \\ d_{i-1,0} + Del(x_i) & \text{si } i > 0 \text{ et } j = 0 \\ d_{0,j-1} + Ins(y_j) & \text{si } i = 0 \text{ et } j > 0 \\ \min \begin{cases} d_{i-1,j-1} + Sub(x_i, y_j) \\ d_{i-1,j} + Del(x_i) \\ d_{i,j-1} + Ins(y_j) \end{cases} & \text{sinon} \end{cases}$$

fonction distanceEdition(x, y) **retourne** entier

- entrées :
  - x : suite de symboles numérotés de 1 à m
  - y : suite de symboles numérotés de 1 à n
- sorties :
  - D: matrice des distances
  - V : matrice des directions optimales

début

1 $D[0\,;0\,] \longleftarrow 0$
2 **pour** $i \longleftarrow 1$ **haut** m **faire**
3 $D[i\,;0\,] \longleftarrow D[i-1\,;0\,] + Del(x_i)$
4 **fin pour**
5 **pour** $j \longleftarrow 1$ **haut** n **faire**
6 $D[0\,;j\,] \longleftarrow D[0\,;j-1\,] + Ins(y_j)$
7 **pour** $i \longleftarrow 1$ **haut** m **faire**
8 $D[i\,;j\,] \longleftarrow \min \begin{pmatrix} D[i-1\,;j-1\,] + sub(x_i, y_j) \\ D[i-1\,;j\,] + Del(x_i) \\ D[i\,;j-1\,] + Ins(y_j) \end{pmatrix}$
9 $V[i\,;j\,] \longleftarrow S, T, D$ ou I selon le minimum
10 **fin pour**
11 **fin pour**
12 **retourner** $D[m\,;n\,]$

fin distanceEdition

Exemple

"$plusieurs$" ⟶ "$malheur$"

X / Y	p	l	u	s	i	e	u	r	s	
	m	a	l	h		e	u	r		
Opération	S	I	T	S	D	D	T	T	T	D
Coût	1	1	0	1	1	1	0	0	0	1

- S = $Sub(a, b)$ avec $a \neq b$ (coût 1)
- T = $Sub(a, a)$ (coût 0)
- I = Ins (coût 1)
- D = Del (coût 1)

Exemple

Matrice D calculée **ligne par ligne**

X \ Y		m 1	a 2	l 3	h 4	e 5	u 6	r 7
	0	1	2	3	4	5	6	7
p 1	1	1	2	3	4	5	6	7
l 2	2	2	2	2	3	4	5	6
u 3	3	3	3	3	3	4	4	5
s 4	4	4	4	4	4	4	5	5
i 5	5	5	5	5	5	5	5	6
e 6	6	6	6	6	6	5	6	6
u 7	7	7	7	7	7	6	5	6
r 8	8	8	8	8	8	7	6	5
s 9	9	9	9	9	9	8	7	6

I ⟵ D ↑ T, S ↖

Matrice V calculée **ligne par ligne**

X \ Y	m 1	a 2	l 3	h 4	e 5	u 6	r 7
p 1	S	I	I	I	I	I	I
l 2	D	S	T	I	I	I	I
u 3	D	D	D	S	I	T	I
s 4	D	D	D	D	S	I	S
i 5	D	D	D	D	D	S	I
e 6	D	D	D	D	T	I	S
u 7	D	D	D	D	D	T	I
r 8	D	D	D	D	D	D	T
s 9	D	D	D	D	D	D	D

I ⟵ D ↑ T, S ↖

Alignement déterminé en **remontant depuis le coin inférieur droit**

X \ Y	m 1	a 2	l 3	h 4	e 5	u 6	r 7
p 1	S	I	I	I	I	I	I
l 2	D	S	T	I	I	I	I
u 3	D	D	D	S	I	T	I
s 4	D	D	D	D	S	I	S
i 5	D	D	D	D	D	S	I
e 6	D	D	D	D	T	I	S
u 7	D	D	D	D	D	T	I
r 8	D	D	D	D	D	D	T
s 9	D	D	D	D	D	D	D

Chapitre 7
Algorithmes gloutons

Matthieu Willems
Département d'informatique
UQÀM

Notes de cours basées sur le matériel pédagogique fourni par Louise Laforest

Principaux éléments d'un algorithme utilisant la stratégie gloutonne (ou vorace, ou greedy en anglais)

- Un **ensemble de candidats** dans lequel sera puisée la solution.
- Une fonction de **sélection**, qui permet de choisir le meilleur candidat à ajouter à la solution.
- Une fonction de **faisabilité**, qui vérifie si un candidat peut faire partie d'une solution.
- Une fonction **objective**, qui associe une valeur à une solution totale ou partielle.
- Une fonction **solution**, qui nous permet de déterminer lorsque nous sommes en présence d'une solution complète.

Caractéristiques d'un algorithme utilisant l'approche vorace

- À chaque étape, il choisit l'élément **le plus prometteur**.
- Il ne revient **jamais en arrière**.
- S'il **rejette** un candidat, il **ne le considérera plus** par la suite.
- Il ne donne **pas toujours de solution**.
- S'il donne une solution, celle-ci n'est **pas nécessairement optimale**.

Algorithme générique

fonction vorace(C) **retourne** ensemble
- entrée :
  - C : ensemble de candidats

début
1 $S \longleftarrow \emptyset$
2 **tant que** pas de solution **et** $C \neq \emptyset$ **faire**
3 $x \longleftarrow$ élément de C maximisant sélect(x) — sélect(x) : valeur du choix de x
4 $C \longleftarrow C - \{x\}$
5 **si** réalisable($S \cup \{x\}$) **alors** — réalisable(E) : vrai si l'ensemble E peut faire partie d'une solution
6 $S \longleftarrow S \cup \{x\}$
7 **fin si**
8 **fin tant que**
9 **si** solution **alors**
10 $res \longleftarrow S$
11 **sinon**
12 $res \longleftarrow \emptyset$
13 **fin si**
14 **retourner** res
fin vorace

Rendu de monnaie

- Problème : Faire la monnaie d'un montant d'argent avec un nombre minimal de pièces
- Valeur des pièces : $p_1 < p_2 < \cdots < p_n, p_i > 0, 1 \leq i \leq n$
- $S \geq 0$: Montant à remettre
- Trouver les valeurs entières x_i, $1 \leq i \leq n$, qui minimisent le nombre total de pièces : $\sum_{i=1}^{n} x_i$ sous la contrainte $S = \sum_{i=1}^{n} x_i\, p_i$
- Exemple :
  - Pièces : 1 ¢, 5 ¢, 10 ¢, 25 ¢, 1 \$ et 2 \$.
  - Montant : 1,47 \$
  - Solution : $1 * 1\,\$ + 1 * 25\,¢ + 2 * 10\,¢ + 2 * 1\,¢$
  - Valeur des x_i : $x_1 = 2, x_2 = 0, x_3 = 2, x_4 = 1, x_5 = 1, x_6 = 0$

Stratégie gloutonne

- Utiliser la valeur maximale possible pour x_n.
- Utiliser la valeur maximale possible pour x_{n-1}
- …

Remarque : On suppose qu'on dispose d'un nombre « infini » de pièces de chaque montant.

Schéma récursif :

$$Monnaie(S, p, n) = \begin{cases} \emptyset & \text{si } S = 0 \\ Monnaie(S \bmod p_n, p, n-1) \cup \left\{\left(n, \left\lfloor \frac{S}{p_n} \right\rfloor\right)\right\} & \text{sinon} \end{cases}$$

Version itérative

fonction Monnaie(S, p, n) **retourne** ensemble de couples
- entrées :
  - S : montant à remettre
  - p : tableau **ordonné** des valeurs des pièces indicé de 1 à n
  - n : nombre de valeurs différentes des pièces
- sortie :
  - (i, x_i): couples tels que x_i est le nombre de pièces de valeur p_i

début
1 $E \longleftarrow \emptyset$
2 $i \longleftarrow n$
3 **tant que** $i > 0$ **faire**
4 $x \longleftarrow \left\lfloor \frac{S}{p_i} \right\rfloor$
5 $S \longleftarrow S \bmod p_i$
6 $E \longleftarrow E \cup \{(i, x)\}$
7 $i \longleftarrow i - 1$
8 **fin tant que**
9 **si** $S \neq 0$ **alors**
10 $E \longleftarrow \emptyset$
11 **fin si**
12 **retourner** E
fin Monnaie

$T(n) = \Theta(n)$

Exemples

p	S	Solution gloutonne
$(1, 5, 10, 25, 100, 200)$	147	(6,0), (5,1), (4,1), (3,2), (2,0), (1,2) : 6 pièces
$(1, 5, 11, 25)$	15	(4,0), (3,1), (2,0), (1,4) : 5 pièces Solution optimale : (2,3) : 3 pièces
$(2, 4, 5)$	16	(3,3), (2,0), (1,0) Pas de solution trouvée. Solution : (2,4)

Remarques

- L'approche gloutonne ne trouve pas toujours la solution optimale, et même parfois ne trouve pas de solutions, même s'il en existe au moins une.
- On appelle canonique un système de pièces (et billets) avec lequel l'approche gloutonne donne toujours la solution optimale.
- La plupart des systèmes sont canoniques. Contre-exemple : le système anglais avant la réforme de 1971.
- Il n'existe pas de caractérisations mathématiques des systèmes canoniques.

Sac alpin (Havresac)

Soient les entiers positifs ou nuls pour $1 \leq i \leq n$:

- p_i : poids des objets de type i
- v_i : valeur des objets de type i (exemple : valeur nutritive)
- x_i : nombre d'objets de type i dans le sac

Problème : maximiser $\sum_{i=1}^{n} x_i v_i$ sous la contrainte $\sum_{i=1}^{n} x_i p_i \leq P_{max}$.

On veut donc maximiser la valeur du sac avec une contrainte sur son poids total.

On numérote les objets dans l'ordre croissant de leurs « valeurs relatives » : $\frac{v_1}{p_1} \leq \frac{v_2}{p_2} \leq \cdots \leq \frac{v_n}{p_n}$

Stratégie gloutonne

- Choisir le plus grand nombre d'objets de type n, i.e., la plus grande valeur de x_n, qui satisfait la contrainte de poids maximal.
- Choisir le plus grand nombre d'objets de type $n-1$, i.e., la plus grande valeur de x_{n-1}, qui satisfait la contrainte de poids maximal.
- …

Remarque : On suppose qu'on dispose d'un nombre « infini » d'objets de chaque type.

Boucle principale de l'algorithme

1 **pour** $i \longleftarrow n$ **bas** 1 **faire**

2 $x_i \longleftarrow \left\lfloor \frac{P_{max}}{p_i} \right\rfloor$

3 $P_{max} \longleftarrow P_{max} - x_i p_i$

4 **fin pour**

Complexité temporelle : $T(n) = \Theta(n)$ ou $T(n) = \Theta(n \lg(n))$, si on inclut le tri des objets.

Remarques

- L'algorithme fournit toujours une solution car la contrainte est $\leq P_{max}$.
- On peut montrer que l'algorithme ne fournit pas toujours la solution optimale mais « s'en approche ».

Exemple

i	1	2	3	4	5
p_i	40	50	8	10	30
v_i	40	60	12	20	66
$\frac{v_i}{p_i}$	1	1,2	1,5	2	2,2

On prend $P_{max} = 106$.
Solution gloutonne : $x_5 = 3, x_4 = 1$, Valeur totale : 218, Poids total : 100
Solution optimale : $x_5 = 3, x_3 = 2$, Valeur totale : 222, Poids total : 106

Choix d'activités

- Une ressource (par exemple une salle de conférences)
- n activités (tâches)
- d_i : début de la tâche i
- f_i : fin de la tâche i

$1 \leq i \leq n$

Problème : Trouver le maximum de tâches pouvant être traitées par la ressource.

- La ressource ne peut traiter qu'une tâche à la fois.
- Une tâche ne peut être interrompue.

Définitions

- Les tâches i et j sont **compatibles** si et seulement si $[d_i; f_i[\cap [d_j; f_j[= \emptyset$.
  Sinon, elles sont **incompatibles**.
- **Ensemble stable** : Sous-ensemble de tâches ne contenant pas de tâches incompatibles.
- Problème : Trouver un ensemble stable de cardinalité maximale.

Algorithme 1

- Générer tous les sous-ensembles possibles de l'ensemble $\{1,2, \ldots, n\}$
- Pour chacun, vérifier s'il est stable
- Retourner celui de cardinalité maximale
- Complexité temporelle : $\Omega(2^n)$

Stratégie gloutonne

Parmi les tâches qui peuvent être ajoutées (compatibles) choisir celle qui se termine **le plus tôt**.

Algorithme

fonction choixActivites(d, f) **retourne** ensemble

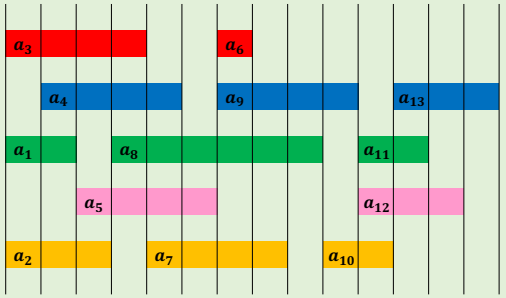
- entrées :
  - d : tableau des débuts d'activités, indicé de 1 à n
  - f : tableau **ordonné** des fins d'activités, indicé de 1 à n
  - n : nombre des activités
- sortie :
  - A : ensemble stable de cardinalité maximale

début

```
1   A ⟵ {1}
2   finTacheCourante ⟵ f[1]
3   pour i ⟵ 2 haut n faire
4       si d[i] ≥ finTacheCourante alors      Tâche i compatible avec celles de A ?
5           A ⟵ A ∪ {i}
6           finTacheCourante ⟵ f[i]
7       fin si
8   fin pour
9   retourner A
```

fin choixActivites

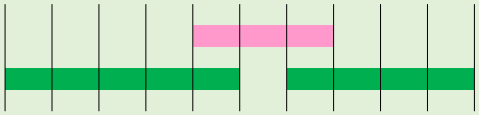
Exemple



Une solution optimale : $a_1\ a_5\ a_6\ a_{10}\ a_{13}$

Remarques

- On peut démontrer que l'approche gloutonne donne une solution optimale.
- La complexité temporelle est $T(n) = \Theta(n)$ ou $T(n) = \Theta(n\lg(n))$, si on inclut le tri des tâches.
- Une autre approche gloutonne qui consisterait a choisir les tâches les plus courtes d'abord ne donne pas toujours une solution optimale.



Codes de Huffman

Codage de caractères (symboles)

- Codes de longueur fixe :
  - ASCII (8 bits)
  - Unicode (16 bits ou plus)
- Codes de longueur variable :
  - code morse
  - codes préfixes

Dans un code préfixe, le code d'un caractère n'est jamais le préfixe d'un autre caractère.

Problème : Trouver un codage qui minimise la longueur moyenne des codes pour un ensemble de n symboles compte tenu de la fréquence d'apparition des symboles : Minimiser $\mu = \sum_{i=1}^{n} l_i f_i$ où f_i est la fréquence de l_i la longueur du code du symbole i.

- Exemples :

	a	b	c	d	e	f	μ	
f_i	0,45	0,13	0,12	0,16	0,09	0,05		
Code 1	000	001	010	011	100	101	3	
Code 2	0	11	10	100	111	01	1,8	✗
Code 3	0	101	100	111	1101	1100	2,24	√

- Codage :

	cadef	longueur
Code 1	010 000 011 100 101	15
Code 2	10 0 100 111 01	11
Code 3	100 0 111 1101 1100	15

- Décodage :

	Texte codé	Texte décodé
Code 1	011100010101	011100010101
Code 2	1001111001	100111100 1 ou 1001111001
Code 3	11111011001100	11111011001100

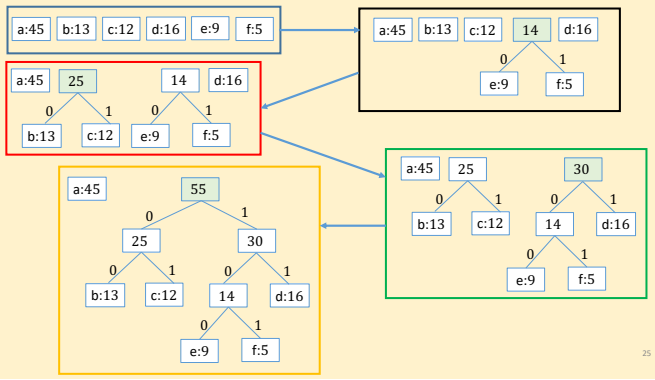
Approche gloutonne (Huffmann)

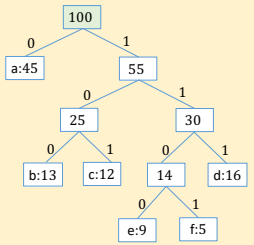
- On construit un arbre (de bas en haut) en regroupant au fur et à mesure les caractères les moins fréquents. On trouve le codage de chaque feuille en parcourant l'arbre depuis la racine jusqu'à cette feuille.

- Propriétés :
  - Le codage de Huffmann est un codage **préfixe**
  - C'est aussi un codage (préfixe) **optimal**
  - Complexité temporelle : $T(n) = \Theta(n\lg(n))$

- Exemple :

a	b	c	d	e	f
45	13	12	16	9	5





	a	b	c	d	e	f	μ
f_i	0,45	0,13	0,12	0,16	0,09	0,05	
Code de Huffman	0	100	101	111	1100	1101	2,24

Ordonnancement de tâches avec pénalités et échéances

- $E = \{1,2,\dots,n\}$ un ensemble de tâches unitaires (durée $= 1$)
- d_i : échéance de la tâche i
- w_i : pénalité si la tâche i se termine après l'échéance
- Une tâche est **en retard** si elle se termine après l'échéance, sinon, elle est **en avance**.
- Problème : Dans quel ordre effectuer les tâches pour minimiser la somme des pénalités des tâches **en retard** (ou maximiser la somme des pénalités des tâches **en avance**).

Notations

- $F \subseteq E$: ensemble de tâches
- $N_t(F) = \text{card}\{i | i \in F \text{ et } d_i \leq t\}$: nombre de tâches de F, dont l'échéance est $\leq t$ (tâches en avance)
- $w(F) = \sum_{i \in F} w_i$: le poids de F
- F est dit **indépendant** si et seulement si $N_t(F) \leq t$, pour tout $1 \leq t \leq n$.

Exemples

Exemple 1 : $E = \{1,2,3,4,5,6,7\}$

i	1	2	3	4	5	6	7
d_i	1	1	2	3	2	1	4
$N_i(E)$	3	5	6	7	7	7	7

L'ensemble E n'est pas indépendant, car par exemple $N_1(E) = 3 > 1$.

i	6 2 5 1 3 4 7						
d_i	1	2	3	4	5	6	7

Exemples

Exemple 2 : $E = \{1,2,3,4,5,6\}$

i	1	2	3	4	5	6
d_i	1	7	4	2	3	5
$N_i(E)$	1	2	3	4	5	5

L'ensemble E est indépendant, car $N_i(E) \leq i$, pour tout $1 \leq i \leq 6$.

i	1	4	5	3	6		2
d_i	1	2	3	4	5	6	7

Exemples

Exemple 3 : $E = \{1,2,3,4\}$

i	1	2	3	4
d_i	7	7	7	7
$N_i(E)$	0	0	0	0

L'ensemble E est indépendant, car $N_i(E) \leq i$, pour tout $1 \leq i \leq 4$.

i							4 3 2 1
d_i	1	2	3	4	5	6	7

Propriétés

- Si E est indépendant, alors $F \subseteq E$ l'est aussi.
- L'ensemble F est indépendant si et seulement s'il existe un ordonnancement de F, tel que toute ses tâches sont en avance.
- Trouver un ordonnancement optimal pour E : Trouver un sous-ensemble indépendant F de E de poids maximal.

Algorithme 1

- Générer tous les 2^n sous-ensembles possibles de l'ensemble $E = \{1,2,\dots,n\}$
- Prendre l'un de ceux de cardinalité maximale et de poids maximal, parmi ceux qui sont indépendants

Algorithme glouton

Étape 1 – Trouver un sous-ensemble indépendant de poids maximal

début

1 Trier (numéroter) les tâches en ordre décroissant de pénalité
2 $F \longleftarrow \emptyset$
3 **pour** $i \longleftarrow 1$ **haut** n **faire**
4 **si** estIndependant$(F \cup \{i\})$ **alors**
5 $F \longleftarrow F \cup \{i\}$
6 **fin si**
7 **fin pour**
8 **retourner** F

fin

Algorithme glouton

Étape 2 – Trouver l'ordonnancement

- Ordonnancer les tâches de F par ordre croissant d'échéance
- Placer à la suite les éléments de $E - F$ dans n'importe quel ordre

Exemple 1

$E = \{1,2,3,4,5,6,7\}$

i	1	2	3	4	5	6	7
d_i	4	2	4	3	1	4	6
w_i	70	60	50	40	30	20	10

i				1			
d_i	1	2	3	4	5	6	7
$N_i(F)$	0	0	0	1	1	1	1

Exemple 1

$E = \{1,2,3,4,5,6,7\}$

i	1	2	3	4	5	6	7
d_i	4	2	4	3	1	4	6
w_i	70	60	50	40	30	20	10

i		2			1		
d_i	1	2	3	4	5	6	7
$N_i(F)$	0	1	1	2	2	2	2

Exemple 1

$E = \{1,2,3,4,5,6,7\}$

i	1	2	3	4	5	6	7
d_i	4	2	4	3	1	4	6
w_i	70	60	50	40	30	20	10

i		2	3		1		
d_i	1	2	3	4	5	6	7
$N_i(F)$	0	1	1	3	3	3	3

Exemple 1

$E = \{1,2,3,4,5,6,7\}$

i	1	2	3	4	5	6	7
d_i	4	2	4	3	1	4	6
w_i	70	60	50	40	30	20	10

i		2	3	4	1		
d_i	1	2	3	4	5	6	7
$N_i(F)$	0	1	2	4	4	4	4

Exemple 1

$E = \{1,2,3,4,5,6,7\}$

i	1	2	3	4	5	6	7
d_i	4	2	4	3	1	4	6
w_i	70	60	50	40	30	20	10

i	5	2	3	4	1		
d_i	1	2	3	4	5	6	7
$N_i(F)$	0	1	2	4	4	4	4
$N_i(G)$	1	2	3	5	5	5	5

Exemple 1

$E = \{1,2,3,4,5,6,7\}$

i	1	2	3	4	5	6	7
d_i	4	2	4	3	1	4	6
w_i	70	60	50	40	30	20	10

i	5	2	6	3	4	1		
d_i	1	2	3	4	5	6	7	
$N_i(F)$	0	1	2	4	4	4	4	
$N_i(G)$	1	2	3	6	6	6	6	

Exemple 1

$E = \{1,2,3,4,5,6,7\}$

i	1	2	3	4	5	6	7
d_i	4	2	4	3	1	4	6
w_i	70	60	50	40	30	20	10

i	5	2	6	3	4	1	7	
d_i	1	2	3	4	5	6	7	
$N_i(F)$	0	1	2	4	4	5	5	

Ordonnancement : **2 4 1 3 7** 5 6 de pénalité 50.

Exemple 2

$E = \{1,2,3,4,5,6,7\}$

i	1	2	3	4	5	6	7
d_i	1	1	2	3	2	1	4
w_i	70	60	50	40	30	20	10

i	2	5	1	3	4	7		
d_i	1	2	3	4	5	6	7	

Ordonnancement : **1 3 4 7** 2 6 5 de pénalité 110.

Exemple 3

$E = \{1,2,3,4,5,6,7\}$

i	1	2	3	4	5	6	7
d_i	1	7	4	2	3	5	7
w_i	70	60	50	40	30	20	10

i	1	4	5	3	6		2	7
d_i	1	2	3	4	5	6	7	

Ordonnancement : **1 4 5 3 6 2 7** de pénalité 0.

Remarques

- On peut démontrer que l'approche gloutonne donne une solution optimale.
- La complexité temporelle est $T(n) = O(n^2)$.

Arbre de recouvrement minimal ARM

- Soit $G = (S, E)$ un graphe non orienté, pondéré, connexe
- Fonction de poids $w : S \times S \rightarrow \mathbb{R}$
- Soit $G' = (S, E')$ connexe, sans cycle (i.e., un arbre), tel que $E' \subseteq E$
- L'arbre G' est un ARM si $\sum_{(u,v)\in E'} w_{u,v}$ est minimale.
- Remarque : $|E'| = |S| - 1 = n - 1$ (car G est un arbre)

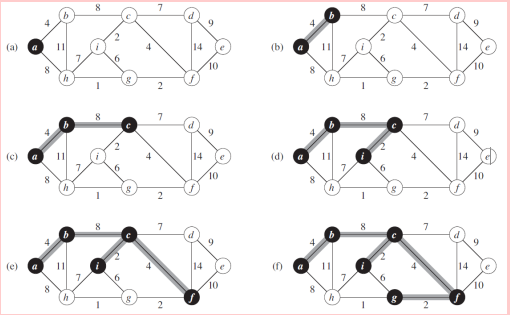
Algorithme naïf

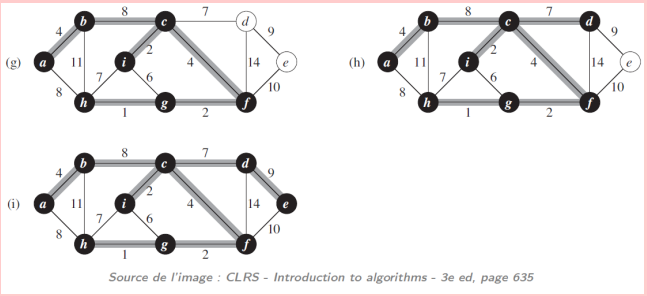
Pour tous les ensembles E' tels que $E' \subseteq E$ de cardinalité $n - 1$, vérifier si $G' = (S, E')$ est connexe et sans cycle. Prendre celui de poids minimal.

Analyse :

- Nombre d'ensembles $E' \subseteq E$ de cardinalité $n - 1$: $\binom{|E|}{n-1}$
- $n - 1 \leq |E| \leq \binom{n}{2}$
- Pire qu'exponentiel dans le pire cas

Algorithme de Prim





Source de l'image : CLRS - Introduction to algorithms - 3e ed, page 635

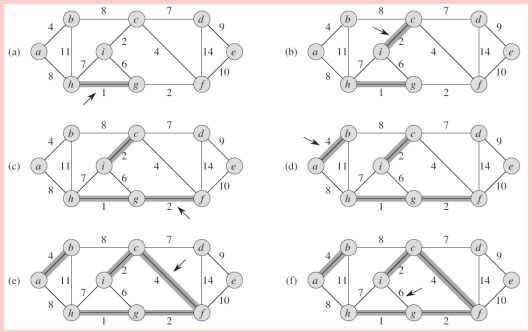
Utilisation d'une file de priorité

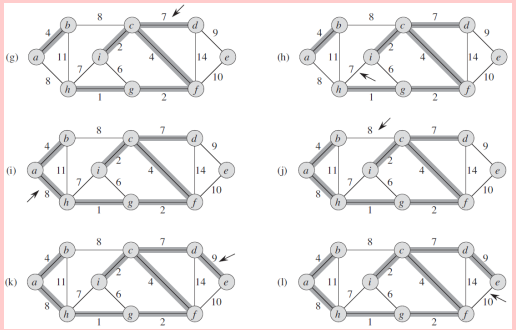
- insérer(F, x) ou $F \longleftarrow F \cup \{x\}$: insertion de l'élément x dans la file F.
- minimum(F) : retourne l'élément ayant la plus petite clé dans F.
- extraireMin(F) : enlève et retourne l'élément ayant la plus petite clé dans F.
- modifierClé(F, x, k) : modifier la clé de x par la nouvelle valeur k.

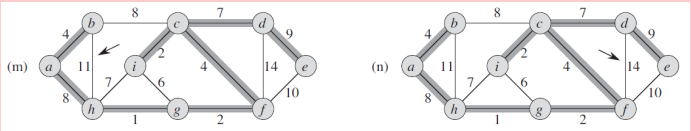
fonction Prim(G, w, r) **retourne** ensemble d'arêtes
- entrées :
  - G: graphe connexe non-orienté
  - w : fonction de poids sur les arêtes
  - r : sommet de départ

début		
1	$A \longleftarrow \emptyset$	
2	**pour** chaque $u \in G.$sommets **faire**	
3	$u.cle \longleftarrow \infty$; $u.parent \longleftarrow null$	
4	**fin pour**	
5	$r.cle \longleftarrow 0$; $file \longleftarrow G.$sommets	$\Theta(\|S\|)$
6	**tant que** $file \neq \emptyset$ **faire**	$\|S\|$ fois
7	$u \longleftarrow$ extraireMin$(file)$	$O(\lg(\|S\|))$
8	$A \longleftarrow A \cup \{(u, u.parent)\}$	
9	**pour** chaque $v \in u.$adjacents **faire**	$O(\|E\|)$
10	**si** $v \in file$ **et** $w(u, v) < v.cle$ **faire**	$O(1)$
11	$v.parent \longleftarrow u$; $v.cle \longleftarrow w(u, v)$	$O(\lg(\|S\|))$
12	**fin si**	
13	**fin pour**	
14	**fin tant que**	
15	**retourner** $A - \{(r, r.parent)\}$	
fin Prim		$O(\|E\|\lg(\|S\|))$

Algorithme de Kruskal







Source de l'image : CLRS - Introduction to algorithms - 3e ed, page 632-633

Principes de l'algorithme

- On maintient une forêt d'arbres sous-ensembles d'un ARM
- Utilisation de la structure ensemble disjoints

créerEnsemble(x) : crée un ensemble ne contenant que x (qui ne doit pas faire partie d'un autre ensemble)

union(x, y) : retourne l'ensemble résultant de l'union des deux ensembles contenant x et y

trouverEnsemble(x) : retourne l'ensemble contenant x

Exemple d'implémentation :

Éléments : $\{1,2, \dots, n\}$

Ensemble de sous-ensembles de $\{1,2, \dots, n\}$

On choisit un représentant par ensemble, exemple le minimum.

Tableau tab tel que $tab[i]$ représentant de i

fonction union(i, j)
    $i \longleftarrow$ trouverEnsemble(i)
    $j \longleftarrow$ trouverEnsemble(j)
    si $i < j$ **alors**
        $tab[j] \longleftarrow i$
    sinon
        $tab[i] \longleftarrow j$
    fin si

i	1	2	3	4	5	6	7	8	9	10
$tab[i]$	1	2	3	2	1	3	4	3	3	4

fonction trouverEnsemble(i)
    tant que $tab[i] \neq i$ **faire**
        $i \longleftarrow tab[i]$
    fin tant que
    retourner i

trouverEnsemble$(1) = 1$
trouverEnsemble$(4) = 2$
trouverEnsemble$(10) = 2$

i	1	2	3	4	5	6	7	8	9	10
$tab[i]$	1	2	3	2	1	3	4	3	3	4

union(1,2) union(6,10)

i	1	2	3	4	5	6	7	8	9	10
$tab[i]$	1	1	3	2	1	3	4	3	3	4

i	1	2	3	4	5	6	7	8	9	10
$tab[i]$	1	2	2	2	1	3	4	3	3	4

fonction Kruskal(G, w) **retourne** ensemble d'arêtes
- entrées :
  - G: graphe connexe non-orienté
  - w : fonction de poids sur les arêtes

début
1 $A \longleftarrow \emptyset$
2 **pour** chaque $v \in G.$sommets **faire**
3 créerEnsemble(v)
4 **fin pour**
5 Trier les arêtes de $G.$arêtes par ordre croissant de poids
6 **pour** chaque $(u, v) \in G.$arêtes en ordre croissant de poids **faire**
7 **si** u et v n'appartiennent au même ensemble **alors**
8 $A \longleftarrow A \cup \{(u, v)\}$
9 union(u, v)
10 **fin si**
11 **fin pour**
12 **retourner** A
fin Kruskal

Séparation et évaluation (*branch and bound*)

- Méthode utilisée pour des problèmes d'optimisation dans lesquels les candidats sont représentés sous la forme des feuilles d'un arbre.
- On stoppe la recherche (élagage) si les scores de tous les descendants d'un nœud sont supérieurs à celui d'une feuille (si on recherche un minimum).
- On commence avec un score initial.

Principes (pour la recherche d'un minimum)

- La méthode diminue l'espace des solutions.
- **Il faut être capable de minorer le score de tous les descendants d'un nœud.**
- On utilise un score optimal α, qui est mis à jour au fur et à mesure.
- Le choix du score de départ est important, d'où l'intérêt d'une bonne approximation initiale si possible.
- Si X est un nœud dont tous les descendants ont un score supérieur à α, il n'est pas nécessaire d'explorer les descendants de X.
- Nous sommes sûrs de trouver le minimum, mais nous ne contrôlons pas le temps d'exécution.
- Technique générale d'exploration d'un arbre de possibilités.

Problème de l'affectation de tâches

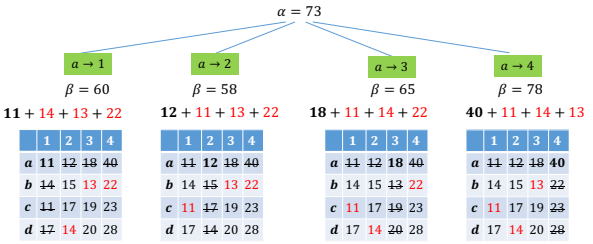
- On doit répartir n tâches en affectant chacune de ces tâches à un agent différent choisi parmi un ensemble de n agents.
- Pour chaque agent, on connaît le coût (par exemple le temps) de chacune des tâches.
- On veut minimiser la somme des coûts des n tâches.
- Nombre de possibilités : $n!$

Exemple pour $n = 4$

Tâche / Agent	1	2	3	4
a	11	12	18	40
b	14	15	13	22
c	11	17	19	23
d	17	14	20	28

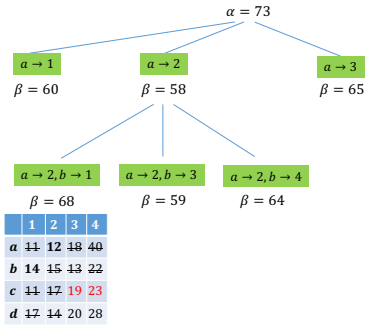
L'affectation en diagonale $a \to 1, b \to 2, c \to 3$ et $d \to 4$ donne un coût total de 73. On peut prendre par exemple ce score comme score initial.

Arbre des possibilités

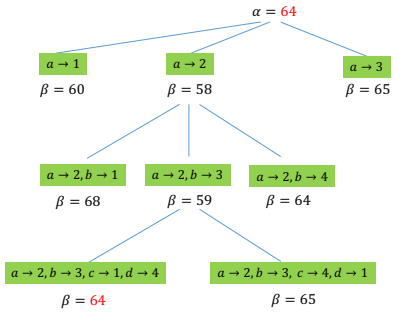


Pour calculer β, on ajoute le minimum possible pour chacune des tâches non affectées

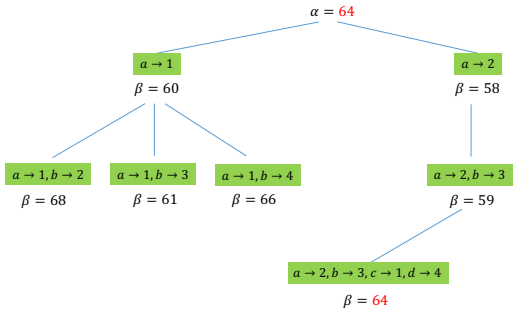
Le quatrième nœud est supprimé car $78 > 73$



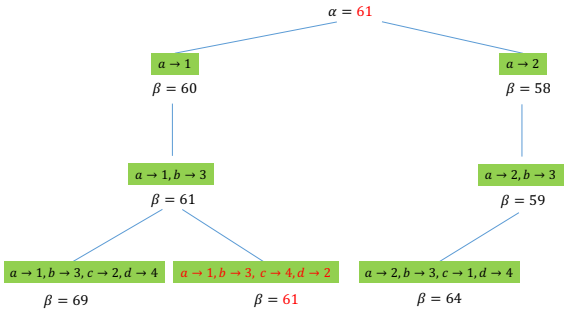
On remplace α par 64



Quatre nœuds sont supprimés



Deux nœuds sont supprimés après la mise à jour de α



Solution optimale

- $a \to 1, b \to 3,\ c \to 4, d \to 2$ de coût total 61.
- On a vérifié 4 feuilles au lieu de 24.
- Il existe un algorithme en temps polynomial (Kuhn-Munkres) pour résoudre ce problème.
- **On explore toujours le nœud avec le score β le plus petit, quitte à remonter dans l'arbre avant d'avoir atteint une feuille.**
- On peut utiliser une approche gloutonne pour trouver l'affectation initiale. Si par exemple on prend pour chaque agent dans l'ordre la tâche disponible qui coûte le moins, on obtient ici $a \to 1, b \to 3, c \to 2, d \to 4$ de coût total 69, ce qui ne change rien ici.

Chapitre 9
NP-complétude

Matthieu Willems
Département d'informatique
UQÀM

Notes de cours basées sur le matériel pédagogique fourni par Louise Laforest

Introduction

- **Algorithme polynomial** : $T(n) = O(n^k)$ pour $k \geq 0$.
- **Problèmes non résolvables**
  Exemple : Problème de l'arrêt d'une machine de Turing : Il n'existe pas de programme permettant de tester n'importe quel programme informatique écrit dans un langage suffisamment puissant, afin de vérifier dans tous les cas s'il s'arrêtera en un temps fini ou bouclera à jamais.
- **Problèmes résolvables mais pas en temps polynomial**, i.e. non $O(n^k)$.
  Ces problèmes sont qualifiés d'intraitables ou difficiles.
  Exemples : Commis voyageur, sac à dos, tours de Hanoï, etc.
- **Problèmes résolvables en temps polynomial**, i.e. $O(n^k)$.
  Ces problèmes sont qualifiés de traitables ou faciles.
  Exemples : Multiplication chainée de matrices, tri fusion, etc.

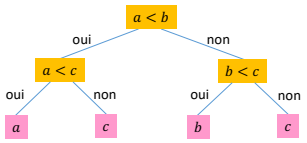
Bornes inférieures

- Borne inférieure pour un problème : Travail minimal requis pour solutionner le problème peu importe l'algorithme utilisé.
- Borne inférieure **serrée** : Il existe un algorithme dont la complexité temporelle **dans le pire cas** est cette borne.
- Borne inférieure **triviale** : combien d'éléments de l'entrée doivent être **traités** + combien d'éléments doivent être produits.

Exemples de bornes inférieures triviales

- Algorithme générant les $n!$ permutations de n premiers entiers : $\Omega(n!)$
- Fusionner deux listes triées de longueur n : $\Omega(n)$
- Évaluation d'un polynôme $p(n) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_0$: $\Omega(n)$
- Nombre d'essais pour deviner un nombre entre 1 et n que quelqu'un a choisi : $\Omega(\lg(n))$
- Multiplication de deux matrices carrées $A \times B$: $\Omega(n^2)$. On ne sait pas si c'est une borne borne serrée.
- Commis voyageur avec $\frac{n(n-1)}{2}$ distances qui produit un circuit de n villes : $\Omega(n^2)$. On n'a rien trouvé de polynomial.

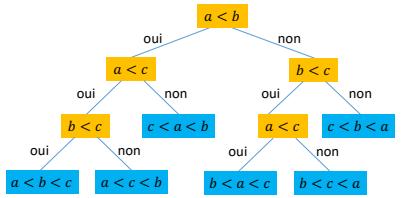
Trouver le minimum entre a, b et c



Arbre de décision

f = nombre de feuilles ≥ nombre de réponses possibles
h = hauteur de l'arbre = nombre maximal de comparaisons
$h \geq \lceil \lg(f) \rceil$; $h + 1 \leq f \leq 2^h$

Trier un tableau Tab tel que $Tab[1] = a, Tab[2] = b$ et $Tab[3] = c$



Arbre de décision

Algorithme de tri basé sur les comparaisons

- Tableau de taille n, $n!$ tableaux possibles.
- f = nombre de feuilles $\geq n!$
- h = hauteur de l'arbre = nombre maximal de comparaisons

- $h \geq \lceil \lg(f) \rceil \geq \lceil \lg(n!) \rceil$
- Formule de Stirling : $n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right)$
- $\lg(n!) = \lg\left(\sqrt{2\pi}\right) + \frac{1}{2}\lg(n) + n\lg(n) - n\lg(e) + \lg\left(1 + \Theta\left(\frac{1}{n}\right)\right)$
- $\lg(n!) = \Theta(n\lg(n))$
- $h = \Omega(n\lg(n))$

Réduction de problème

Solutionner un problème en le réduisant à un problème dont on connait la solution.

- Solutionner $dx + e = 0$: Utiliser la solution pour $ax^2 + bx + c = 0$ en posant $a = 0$, $b = d$ et $c = e$.
- $\text{PPCM}(a, b)$:
  - Trouver la factorisation en nombres premiers de a et b et en déduire $\text{PPCM}(a, b)$.
  - Plus efficace : $\text{PPCM}(a, b) = \frac{ab}{\text{PGCD}(a,b)}$ et utiliser l'algorithme d'Euclide pour trouver $\text{PGCD}(a, b) = O(\log_{10}(b))$ où $b = \min(a, b)$.
- Calcul du nombre de chemins de longueur k entre s_i et s_j dans un graphe : $a_{i,j}^k$. Utiliser l'exponentiation à la russe pour calculer A^k (A : matrice d'incidence).
- $\min(f(x)) = -\max(-f(x))$

Problèmes souvent utilisés pour la réduction de problème

Problème	Borne inférieure	Serrée ?
Trier	$\Omega(n\lg(n))$	oui (tri fusion par exemple)
Rechercher dans un tableau trié	$\Omega(\lg(n))$	oui (fouille binaire par exemple)
Problème de l'unicité	$\Omega(n\lg(n))$	oui
Multiplication d'entiers à n chiffres	$\Omega(n)$	non ($O(n^2)$, $O(n^{1,585})$)
Multiplication de matrices carrées	$\Omega(n^2)$	non ($O(n^3)$, $O(n^{2,808})$)

Trois classes de problèmes

- **P** : la solution se trouve en temps polynomial

- **NP** : problèmes vérifiables en temps polynomial (Polynomiaux Non déterministes)
  - Exemple : Circuit hamiltonien (passe une seule fois par tous les sommets d'un graphe). **Trouver** un circuit hamiltonien est **plus que polynomial**. **Vérifier** si un circuit donné est hamiltonien est **polynomial**.

On peut vérifier en temps polynomial si une solution donnée est correcte mais qu'en est-il de trouver une solution ?

$$\mathbf{P} \subseteq \mathbf{NP}$$

- **NP-Complets**

Classe des problèmes NP-Complets (statut inconnu)

- Un problème NP-Complet :
  - appartient à NP
  - et est aussi difficile que n'importe quel problème de NP
  - aucun algorithme polynomial trouvé à ce jour
  - aucune preuve de l'absence d'un algorithme polynomial
- Question appelée P ≠ NP (posée en 1971)
- S'il existe un problème NP-Complet résoluble en temps polynomial, alors tous les problèmes NP-Complets (et donc aussi tous les problèmes NP) seront résolubles en temps polynomial.

Exemples

Problème polynomial	Problème NP-Complet
• Plus courts chemins à origine unique dans un graphe orienté (exemple : Algorithme de Dijkstra) $O(\|S\|\|A\|)$	• Plus longs chemins à origine unique dans un graphe orienté • Existence d'un chemin de longueur au moins k
• Tournée eulérienne (passe une seule fois par chaque arc) $O(\|A\|)$	• Circuit hamiltonien (passe une seule fois par chaque sommet)
• Satisfaisabilité $2-\text{FNC}$ $(2-\text{SAT})$	• Satisfaisabilité $3-\text{FNC}$ $(3-\text{SAT})$

Remarques

- La NP-Complétude s'applique aux problèmes de décision mais pas directement aux problèmes d'optimisation.
- On peut transformer un problème d'optimisation en un de décision :

Exemple :
  - *Optimisation* : PLUS-COURT-CHEMIN Trouver le chemin utilisant le moins d'arêtes qui relie u à v dans le graphe G.
  - *Décision* : CHEMIN Existe-t-il un chemin de u à v dans le graphe G qui utilise au plus k arêtes.

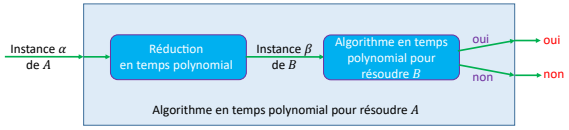
- Un problème de décision est plus facile (pas plus difficile) que le problème d'optimisation correspondant.
- Fournir la preuve qu'un problème de décision est difficile prouve que le problème d'optimisation qui lui est associé est aussi difficile.
- Ce principe s'applique aussi lorsqu'on utilise un problème de décision pour résoudre un autre problème de décision.

Réductibilité

On peut ramener un problème A à un problème B si on peut transformer toute instance α du problème A en une instance β équivalente du problème B.

Une telle transformation doit vérifier les conditions suivantes :

- La transformation prend un temps polynomial
- Réponse de l'algorithme A exécuté sur α ⇔ Réponse de l'algorithme B exécuté sur β



Exemple

Problème A : CHEMIN : Existe-t-il un chemin de u à v dans le graphe G qui utilise au plus k arêtes ?

Problème B : Autre problème de décision dans P dont on connait la solution

Instance de CHEMIN : $\alpha = (G, u, v, k)$

Réduction : Transformer α pour en faire une instance β de B en temps polynomial.

Problèmes de la satisfaisabilité (SAT)

Définition : Un littéral est une expression booléenne qui est soit une formule atomique (V, F ou une variable booléenne) soit la négation d'une formule atomique.

Exemples : $V, F, x, \neg y$

Rappels :

Conjonction : $x \wedge y$

x \ y	F	V
F	F	F
V	F	V

Négation : $\neg x$

x	F	V
$\neg x$	V	F

Disjonction : $x \vee y$

x \ y	F	V
F	F	V
V	V	V

Forme normale conjonctive

- Définition : Une expression booléenne ϕ est en **forme normale conjonctive** (FNC, formule de Krom lorsque $k = 2$) si ϕ est de la forme $\wedge_{j=1}^{m} C_j$, où chaque **clause** C_j est une disjonction de littéraux.
- Exemples :
  - $(\neg x \vee y \vee z) \wedge (x \vee w) \wedge (x \vee y \vee v \vee \neg w)$
  - $(\neg x_1 \vee x_2) \wedge (x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_3 \vee x_4 \vee x_5 \vee x_6) \wedge (x_4 \vee \neg x_5)$
- Contre-exemples :
  - $\neg\,(x \vee y)$
  - $\big((x \wedge v) \vee (y \wedge z)\big) \wedge w$
- Toute expression booléenne est logiquement équivalente à une forme normale conjonctive :
  - $\neg\,(x \vee y) \equiv \neg x \wedge \neg y$
  - $\big((x \wedge v) \vee (y \wedge z)\big) \wedge w \equiv (x \vee y) \wedge (x \vee z) \wedge (v \vee y) \wedge (v \vee z) \wedge w$

Problèmes k-SAT, $k \geq 2$

- Soit ϕ une expression booléenne en FNC, où chaque clause contient $k \geq 2$ littéraux (avec un nombre indéterminé de variables), existe-t-il une affectation des variables qui rende ϕ vraie ?
- Exemplaires du problème :
  - $\boldsymbol{k = 2}$: $P\big((x \vee y) \wedge (x \vee \neg y) \wedge (\neg x \vee y) \wedge (\neg x \vee \neg y)\big) = faux$, car aucune affectation des variables rend ϕ vraie.
  - $\boldsymbol{k = 3}$: $P\big((\neg z \vee \neg y \vee x) \wedge (\neg z \vee y \vee x) \wedge (\neg z \vee y \vee \neg x)\big) = vrai$, car, par exemple, l'affectation $x = V, y = V, z = V$ rend ϕ vraie.
- Remarques :
  - Il existe des algorithmes polynomiaux pour résoudre le problème 2-SAT.
  - Les problèmes k-SAT, $k \geq 3$ sont NP-complets.

Solution polynomiale pour 2-SAT

Rappels : $x \vee y \equiv \neg x \rightarrow y \equiv \neg y \rightarrow x$

Implication : $x \rightarrow y$

x \ y	F	V
F	V	V
V	F	V

Équivalence : $x \leftrightarrow y$

x \ y	F	V
F	V	F
V	F	V

Algorithme linéaire pour résoudre 2-SAT :

On construit un graphe avec les arcs $(\neg a, b)$ **et** $(\neg b, a)$ pour chaque clause $a \vee b$. Une instance du problème est satisfaisable si aucune variable et sa négation n'appartiennent à la même *composante fortement connexe*.

Composante fortement connexe : il existe un chemin entre chaque paire de sommets.

Problèmes abstraits

Définition : Un problème abstrait Q est une relation binaire $R \subseteq I \times S$ où
- I : ensemble des instances de Q
- S : ensemble des solutions

Pour un problème de décision, $S = \{0,1\}$ où $1 =$ oui et $0 =$ non.

Exemple : Si $i = (G, u, v, k)$ est une instance de CHEMIN, alors
- CHEMIN$(i) = 1$ s'il existe un chemin de longueur k entre u et v dans G
- CHEMIN$(i) = 0$ sinon

Encodages

- Définition : Un encodage d'un ensemble S d'objets abstraits est une fonction $e : S \rightarrow \{0,1\}^*$

$\{0,1\}^*$: ensemble de toutes les chaînes binaires (y compris la chaîne vide)

Exemples :
- $S = \mathbb{N}$, $e(S) = \{0, 1, 10, 11, 100, 101, 110, 111, 1000, 1001, \dots\}$
- S : caractères unicode, $e(A) = 1000001$

Tout peut être représenté par des chaines binaires : polygones, graphes, polynomes, fonctions, programmes, etc.

L'efficacité de la résolution d'un problème ne doit pas dépendre de l'encodage choisi.

Définitions formelles I

- Un algorithme qui résout un **problème de décision abstrait** prend en entrée un encodage d'une instance du problème.
- Un problème est dit **concret** si l'ensemble de ses instances sont des chaines binaires.
- Un algorithme résout un **problème concret** en temps $O(T(n))$, si sur une instance i de longueur $n = |i|$ (nombre de caractères binaires) il produit la solution en temps $O(T(n))$.
- Un **problème concret** est résoluble en temps polynomial s'il existe un algorithme de complexité temporelle $O(n^k)$ pour une certaine constante k.
- Classe de complexité P : ensemble des **problèmes de décision concrets** résolubles en temps polynomial.

Définitions formelles II

- Définition : Une fonction $f: \{0,1\}^* \rightarrow \{0,1\}^*$est calculable en temps polynomial s'il existe un algorithme A à temps polynomial qui produit $f(x)$ où $x \in \{0,1\}^*$, une entrée de A.

- Définition : Pour un certain ensemble I d'instances, deux encodages e_1 et e_2 sont reliés polynomialement s'il existe deux fonctions calculables en temps polynomial f_{12} et f_{21} telles que :

$$\forall i \in I, f_{12}(e_1(i)) = e_2(i) \text{ et } f_{21}(e_2(i)) = e_1(i).$$

Théorie des langages formels I

- Alphabet Σ : ensemble fini de symboles (exemple $\Sigma = \{0,1\}$)

- Σ^* : Ensemble de toutes les chaines possibles avec des symboles de Σ y compris la chaine vide ε. (exemple : $\{\varepsilon, 0, 1, 10, 11, 100, 101, 110, \dots\}$)
- Langage L: $L \subseteq \Sigma^*$

- L'ensemble des instances d'un problème de décision concret Q est l'ensemble $\{0,1\}^*$ et Q est un langage L tel que $L = \{x \in \{0,1\}^* : Q(x) = 1\}$.

Théorie des langages formels II

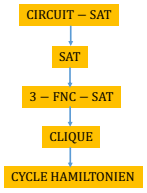
- Un algorithme A **accepte** une chaine $x \in \{0,1\}^*$ si l'algorithme produit le résultat $A(x) = 1$.
- Un algorithme A **rejette** une chaine $x \in \{0,1\}^*$ si l'algorithme produit le résultat $A(x) = 0$.
- Langage accepté par A : $L = \{x \in \{0,1\}^* : A(x) = 1\}$
- Le langage L est **décidé en temps polynomial** par un algorithme A s'il est accepté par A et que $\forall x \in \{0,1\}^*, |x| = n$, l'algorithme décide correctement si $x \in L$ en temps $O(n^k)$ pour une constante k.
- $P = \{L \subseteq \{0,1\}^* : \exists A$ qui décide L en temps polynomial$\}$
- Exemple : EST-PREMIER(n)

$L = \{10, 11, 101, 111, 1011, 1101, 10001, \dots\}$

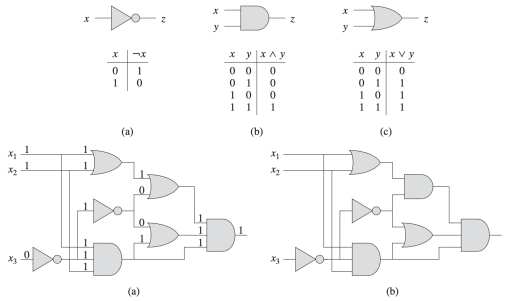
Problèmes NP-Complets

Théorème de Cook (1971) : Le problème SAT est NP-Complet.

Suite de réductions :



CIRCUIT – SAT : Satisfaisabilité de circuit

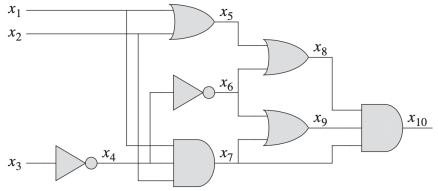


Source de l'image : CLRS - Introduction to algorithms - 3e ed, pages 1071-1072

Énoncé du problème

- Existe-t-il une affectation des variables d'entrée du circuit qui donne 1 en sortie ?
- Algorithme naïf en $\Omega(2^k)$: vérifier toutes les 2^k possibilités pour les k variables.
- On peut montrer que ce problème est NP-Complet.

Réduction de $\mathbf{CIRCUIT-SAT}$ vers $\mathbf{SAT}$



Source de l'image : CLRS - Introduction to algorithms - 3e ed, page 1081

$$\phi = x_{10} \wedge (x_4 \leftrightarrow \neg x_3) \wedge (x_4 \leftrightarrow (x_1 \vee x_2)) \wedge (x_6 \leftrightarrow \neg x_4) \wedge (x_7 \leftrightarrow (x_1 \wedge x_2 \wedge x_4)) \wedge (x_8 \leftrightarrow (x_5 \vee x_6)) \wedge (x_9 \leftrightarrow (x_6 \vee x_7)) \wedge (x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9))$$

Réduction de $\mathbf{SAT}$ vers $\mathbf{3-FNC-SAT}$

$3-\mathrm{FNC}-\mathrm{SAT}$: Satisfaisabilité booléenne en FNC. Il s'agit de déterminer si une formule booléenne est satisfaisable.

- n variables booléennes
- m connecteurs logiques
- des parenthèses

On veut transformer une formule booléenne quelconque ϕ en une formule booléenne ϕ''' qui vérifie les deux propriétés suivantes :

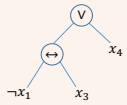
- ϕ''' est sous la forme $3-\mathrm{FNC}$
- La satisfaisabilité de ϕ''' est équivalente à celle de ϕ.

1ère étape : construction d'un arbre d'analyse

$$\phi = \big((x_1 \to x_2) \vee \neg\big((\neg x_1 \leftrightarrow x_3) \vee x_4\big)\big) \wedge \neg x_2$$

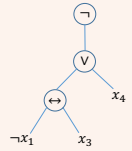
$$\phi = \big((x_1 \to x_2) \vee \neg\big((\neg x_1 \leftrightarrow x_3) \vee x_4\big)\big) \wedge \neg x_2$$

1ère étape : construction d'un arbre d'analyse



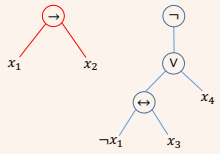
$$\phi = \big((x_1 \to x_2) \vee \neg\big((\neg x_1 \leftrightarrow x_3) \vee x_4\big)\big) \wedge \neg x_2$$

1ère étape : construction d'un arbre d'analyse



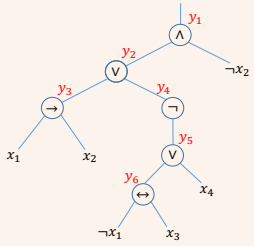
$$\phi = \big((x_1 \to x_2) \vee \neg\big((\neg x_1 \leftrightarrow x_3) \vee x_4\big)\big) \wedge \neg x_2$$

1ère étape : construction d'un arbre d'analyse



$$\phi = \big((x_1 \to x_2) \vee \neg\big((\neg x_1 \leftrightarrow x_3) \vee x_4\big)\big) \wedge \neg x_2$$

1ère étape : construction d'un arbre d'analyse

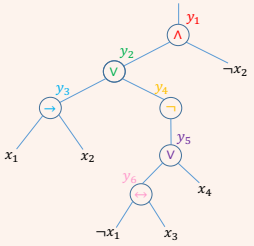


Chaque feuille correspond à une variable d'origine (ou à sa négation)

On ajoute des variables y_i aux nœuds internes

$$\phi = \big((x_1 \to x_2) \vee \neg\big((\neg x_1 \leftrightarrow x_3) \vee x_4\big)\big) \wedge \neg x_2$$

1ère étape : construction d'un arbre d'analyse



On construit une formule équivalente ϕ' qui est une conjonction de y_1 et d'équivalences (une par nœud interne) :

$$\phi' = y_1 \wedge (y_1 \leftrightarrow (y_2 \wedge \neg x_2)) \wedge (y_2 \leftrightarrow (y_3 \vee y_4)) \wedge (y_3 \leftrightarrow (x_1 \to x_2)) \wedge (y_4 \leftrightarrow \neg y_5) \wedge (y_5 \leftrightarrow (y_6 \vee x_4)) \wedge (y_6 \leftrightarrow (\neg x_1 \leftrightarrow x_3))$$

$$\phi' = \phi'_1 \wedge \phi'_2 \wedge \phi'_3 \wedge \phi'_4 \wedge \phi'_5 \wedge \phi'_6 \wedge \phi'_7$$

$$\phi = \big((x_1 \to x_2) \vee \neg\big((\neg x_1 \leftrightarrow x_3) \vee x_4\big)\big) \wedge \neg x_2$$

Remarques

- La transformation de ϕ en ϕ' introduit au plus 1 variable supplémentaire et 1 clause par connecteur logique.
- On peut rajouter des parenthèses si nécessaire. Par exemple, on a le choix entre $a \vee b \vee c = a \vee (b \vee c) = (a \vee b) \vee c$.
- La formule ϕ' est une **conjonction** de clauses impliquant **au plus trois variables**.

$$\phi' = y_1 \wedge (y_1 \leftrightarrow (y_2 \wedge \neg x_2)) \wedge (y_2 \leftrightarrow (y_3 \vee y_4)) \wedge (y_3 \leftrightarrow (x_1 \rightarrow x_2)) \wedge (y_4 \leftrightarrow \neg y_5) \wedge (y_5 \leftrightarrow (y_6 \vee x_4)) \wedge (y_6 \leftrightarrow (\neg x_1 \leftrightarrow x_3))$$

$$\phi' = \phi'_1 \wedge \phi'_2 \wedge \phi'_3 \wedge \phi'_4 \wedge \phi'_5 \wedge \phi'_6 \wedge \phi'_7$$

2e étape : on transforme chaque ϕ'_i en une forme normale conjonctive ϕ''_i

On construit par exemple la table de vérité de $\phi'_2 = y_1 \leftrightarrow (y_2 \wedge \neg x_2)$

y_1	y_2	x_2	$y_1 \leftrightarrow (y_2 \wedge \neg x_2)$
1	1	1	0
1	1	0	1
1	0	1	0
1	0	0	0
0	1	1	1
0	1	0	0
0	0	1	1
0	0	0	1

2e étape : on transforme chaque ϕ'_i en une forme normale conjonctive ϕ''_i

On construit par exemple la table de vérité de $\phi'_2 = y_1 \leftrightarrow (y_2 \wedge \neg x_2)$

y_1	y_2	x_2	$y_1 \leftrightarrow (y_2 \wedge \neg x_2)$
1	1	1	0
1	1	0	1
1	0	1	0
1	0	0	0
0	1	1	1
0	1	0	0
0	0	1	1
0	0	0	1

On en déduit la négation de ϕ'_2 sous forme normale disjonctive

$\neg\phi'_2 = (y_1 \wedge y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge \neg x_2) \vee (\neg y_1 \wedge y_2 \wedge \neg x_2)$

2e étape : on transforme chaque ϕ'_i en une forme normale conjonctive ϕ''_i

On construit par exemple la table de vérité de $\phi'_2 = y_1 \leftrightarrow (y_2 \wedge \neg x_2)$

y_1	y_2	x_2	$y_1 \leftrightarrow (y_2 \wedge \neg x_2)$
1	1	1	0
1	1	0	1
1	0	1	0
1	0	0	0
0	1	1	1
0	1	0	0
0	0	1	1
0	0	0	1

On en déduit la négation de ϕ'_2 sous forme normale disjonctive

$\neg\phi'_2 = (y_1 \wedge y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge \neg x_2) \vee (\neg y_1 \wedge y_2 \wedge \neg x_2)$

On en déduit ϕ''_2 sous forme normale conjonctive

$\phi''_2 = (\neg y_1 \vee \neg y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee x_2) \wedge (y_1 \vee \neg y_2 \vee x_2)$

Remarques

- La transformation de ϕ' en ϕ'' introduit au plus 8 clauses pour chaque clause de ϕ', puisque la table de vérité contient au plus $2^3 = 8$ lignes.
- Vous ferez les autres transformations pendant la séance d'exercices.
- Il nous reste à « ajouter des variables » aux clauses de ϕ'' qui en contiennent seulement 1 ou 2.

3e étape : transformation des clauses ϕ''_i qui contiennent seulement 1 ou 2 variables.

Si la clause contient seulement une variable : $\phi''_i = x$, on rajoute 2 variables a et b et on remplace ϕ''_i par :

$\phi'''_i = (x \vee a \vee b) \wedge (x \vee \neg a \vee b) \wedge (x \vee a \vee \neg b) \wedge (x \vee \neg a \vee \neg b)$

Si la clause contient seulement deux variables : $\phi''_i = x \vee y$, on rajoute 1 variable a et on remplace ϕ''_i par :

$\phi'''_i = (x \vee y \vee a) \wedge (x \vee y \vee \neg a)$

Exemple :

$\phi'''_1 = (y_1 \vee z_1 \vee z_2) \wedge (y_1 \vee \neg z_1 \vee z_2) \wedge (y_1 \vee z_1 \vee \neg z_2) \wedge (y_1 \vee \neg z_1 \vee \neg z_2)$

Remarques

- La transformation de ϕ'' en ϕ''' utilise 3 variables et au plus 4 clauses pour chaque clause de ϕ''.
- La taille de ϕ''' est donc polynomiale par rapport à la taille de ϕ.
- Chacune des transformations peut être effectuée en un temps polynomial.
- Puisque SAT est NP-Complet, alors $3 - \text{FNC} - \text{SAT}$ l'est aussi.

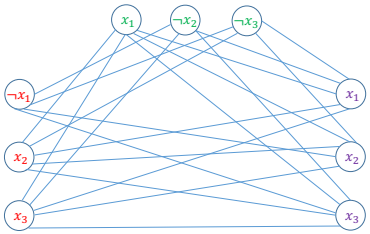
CLIQUE

- Une **clique** dans un graphe non-orienté $G = (S, A)$ est un sous-ensemble $S' \subseteq S$ de sommets dont chaque paire est reliée par une arête de A, i.e. c'est un sous-graphe complet de G (chaque sommet est connecté à tous les autres sommets).
- Problème CLIQUE : Trouver la clique de taille maximale dans un graphe.
- On peut réduire $3 - \text{FNC} - \text{SAT}$ vers CLIQUE. Puisque $3 - \text{FNC} - \text{SAT}$ est NP-Complet, alors CLIQUE l'est aussi.

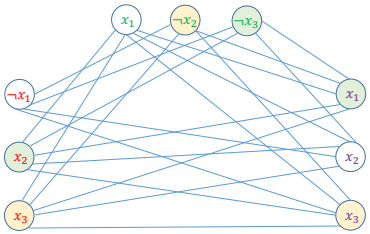
Réduction de $3 - \text{FNC} - \text{SAT}$ vers CLIQUE

- Soit $\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_k$ une formule booléenne en FNC.
- $C_r = l_1^r \vee l_2^r \vee l_3^r$
- On construit $G = (S, A)$ (se fait en temps polynomial) :
  - Pour chaque C_r on place 3 sommets v_1^r, v_2^r et v_3^r dans S
  - On met un arc entre v_i^r et v_j^s si :
    - $r \neq s$
    - l_i^r n'est pas la négation de l_j^s

Exemple : $\phi = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$



Exemple : $\phi = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$



Une assignation qui satisfait ϕ :
$x_1 = 0$ ou $1, x_2 = 0, x_3 = 1$

Clique : sommets jaunes

Une autre assignation qui satisfait ϕ :
$x_1 = 1, x_2 = 1, x_3 = 0$

Clique : sommets verts

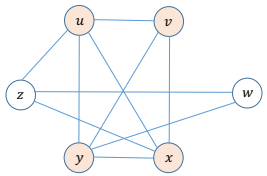
COUVERTURE-SOMMETS

- Une couverture de sommets d'un graphe non-orienté $G = (S, A)$ est un sous-ensemble $S' \subseteq S$ tel que pour tout $(u, v) \in A$ alors $u \in S'$ ou (inclusif) $v \in S'$ (chaque arête a au moins une de ses extrémités dans S').

- Problème COUVERTURE-SOMMETS : trouver l'ensemble S' de cardinalité minimale.

Réduction de CLIQUE vers COUVERTURE-SOMMETS

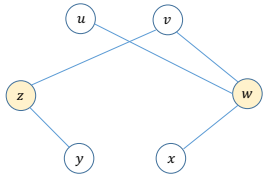
(a) $G = (S, A)$



(a) Clique
$S' = \{u, v, y, x\}$

(b) Complémentaire de G : $\bar{G} = (S, \bar{A})$ où

$\bar{A} = \{(u, v) | u \in S, v \in S, u \neq v, (u, v) \notin A\}$



(b) Couverture de sommets
$S - S' = \{z, w\}$

INF5130 Algorithmique

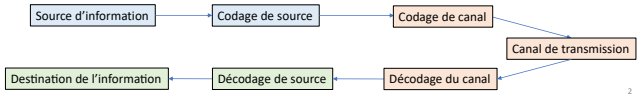
Chapitre 11
Introduction à la théorie de l'information

Matthieu Willems
Département d'informatique
UQÀM

Notes de cours basées sur le matériel pédagogique fourni par Louise Laforest et Marie-Jean Meurs

Introduction

- Seconde moitié du XX^{e} siècle : développement de systèmes où l'information transmise est codée sous forme numérique.
- Transmission simultanée d'informations de natures très différentes : données, signaux audio, TV, etc.
- Fondements théoriques : Claude Shannon, articles sur les *Théories mathématiques des communications* (1948)
- Problèmes mathématiques :
  - Comment convertir de façon optimale des signaux analogiques en signaux numériques ?
  - Comment transmettre sans erreur un message numérique sur une voie de transmission bruyante ?



Le codage de source

[Problème 1.] Comment convertir de façon optimale des signaux analogiques en signaux numériques ?

- On considère les ensembles [source + codage de source] et [décodage de source + destinataire].
- Ils sont reliés par une voie de transmission fictive non bruyante : l'ensemble [codage de canal + voie de transmission + décodage de canal].
- La source d'information engendre l'information sous forme numérique.

Le codage de source – Cas d'une source numérique

- Établir une correspondance bijective entre les éléments de l'alphabet de source A et les mots-codes constitués à l'aide de l'alphabet de code B.
  - Si B a seulement deux symboles $(0,1)$: codage binaire
  - Si B a q éléments : codage q-aire.
- Exemple : le code morse utilise un alphabet ternaire $\mathrm{B} = \{.,-,\text{blanc}\}$

Le codage de source – Cas d'une source numérique

- Le codage étant bijectif et le canal non bruité, le décodage se fait sans ambiguïté.
- Objectif dans le cas d'une source numérique : réduire au maximum la représentation de l'information
  → la réduction (compression) de données dépend du débit de symboles nécessaires à la représentation complète du signal de source.
- Ce débit est lié à un paramètre de la source : **l'entropie** (de Shannon).

Le codage de source – Cas d'une source analogique

- Source de type analogique : le signal engendré ne peut pas être représenté par une suite finie de symboles d'un alphabet fini.
- Conversion analogique vers numérique : alphabet de source A infini *versus* alphabet de code infini.
- ⇒ perte d'information, *i.e.*, « distorsion » entre le signal source et le signal reconstruit.

Le codage de canal

[Problème 2.] Comment transmettre sans erreur un message numérique sur une voie de transmission bruyante ?

- Opérations de codage et décodage de canal, la voie de transmission étant considérée comme bruyante.
- Ce codage/décodage a pour objectif d'annuler les effets du bruit présent sur la voie de transmission qui causerait des erreurs de décodage à la réception ⇔ réaliser une voie de transmission fictive non bruyante.
- Théorème faisant apparaître la **capacité** d'une voie de transmission (Shannon) : si le débit de symboles à l'entrée du codeur de voie de transmission est inférieur à la capacité de la voie, on peut trouver un codage permettant de reconstruire la séquence sans erreur.

L'information et sa mesure

Qu'est-ce qui fait que certains messages apportent plus d'information que d'autres ? → **Quantifier l'information !**

Propriétés :

1. La quantité d'information d'un événement est lié au degré d'incertitude de cet événement.
2. Plus l'événement est incertain, plus il contient d'informations.
3. Si deux événements sont indépendants, l'information qu'ils contiennent ensemble est égale à la somme des informations propres à chacun pris séparément.

L'information et sa mesure

La mesure d'information d'un événement α de probabilité d'occurrence $p(\alpha)$ est notée :

$$I(\alpha) = -\log(p(\alpha))$$

- En base 2, l'unité d'information est le bit (choix par défaut dans ces notes). On écrira souvent $I(\alpha) = h(\alpha) = -\lg(p(\alpha))$
- En base e, l'unité d'information est le nat.

Exemple : Un candidat a 50% de chance de réussir un test. Annoncer sa réussite apporte donc une information de 1 bit.

Information relative à une variable aléatoire- Entropie

Soit X une variable aléatoire à valeurs dans $\{a_1, \dots, a_k, \dots, a_m\}$ avec les probabilités $\{p_X(a_1), \dots, p_X(a_k), \dots, p_X(a_m)\}$.
Si X vaut x, la quantité d'information fournie est $I(x) = -\lg\big(p(x)\big)$

L'espérance mathématique de $I(X)$ est appelée l'**entropie** de la variable aléatoire X :

$$H(X) = -\sum_{k=1}^{m} p_X(a_k) \lg\big(p_X(a_k)\big) = -\sum_{x} p(x) \lg\big(p(x)\big)$$

Remarque : $I(x)$ mesure l'*incertitude* a priori sur le fait que X puisse prendre la valeur x. Dès que cette valeur est réalisée, l'incertitude devient *information*.

Information mutuelle

Soit (X, Y) un couple de variables aléatoires **non indépendantes** à valeurs dans $\{a_k, k = 1, 2 \dots\} \times \{b_j, j = 1, 2 \dots\}$ caractérisé par une distribution de probabilités $p_{XY}(a_k, b_j)$.

- L'incertitude initiale (a priori, *i.e.*, en l'absence de l'information $y = b_j$) de l'événement $x = a_k$ est $-\lg\big(p_X(a_k)\big)$.
- L'incertitude finale (a posteriori, *i.e.*, quand on sait que $y = b_j$) de l'événement $x = a_k$ est $-\lg\Big(p_{X|Y}\big(a_k|b_j\big)\Big)$.

La différence d'incertitude entre les deux situations est donc :

$$I_{X|Y}(a_k, b_j) = -\lg\big(p_X(a_k)\big) - \Big(-\lg\Big(p_{X|Y}\big(a_k|b_j\big)\Big)\Big) = \lg\left(\frac{p_{X|Y}(a_k|b_j)}{p_X(a_k)}\right)$$

Information mutuelle

On a :

$$I_{Y|X}(b_j, a_k) = \lg\left(\frac{p_{Y|X}(b_j|a_k)}{p_Y(b_j)}\right) = I_{X|Y}(a_k, b_j)$$

On appelle ce nombre **information mutuelle** entre les événements $x = a_k$ et $y = b_j$ que l'on note

$$I(x, y) = \lg\left(\frac{p(x|y)}{p(x)}\right).$$

L'espérance de $I(x, y)$ sur toutes les valeurs possibles de x et y est (l'espérance de) l'information mutuelle entre X et Y :

$$I(X, Y) = \sum_{x}\sum_{y} I(x, y)p(x, y)$$

Remarque : $I(X, Y)$ caractérise conjointement le couple de variables aléatoires (X, Y) tandis que $I(x, y)$ caractérise une réalisation particulière (x, y).

Information mutuelle et conditionnelle

On peut également définir **l'information propre conditionnelle** de l'événement $x = a_k$ sachant que l'événement $y = b_j$ est réalisé :

$$I_{X|Y}(a_k|b_j) = \lg\left(\frac{1}{p_{X|Y}(a_k|b_j)}\right)$$

que l'on note :

$$I(x|y) = \lg\left(\frac{1}{p(x|y)}\right)$$

→ information nécessaire à un observateur pour déterminer complètement l'événement $x = a_k$ s'il sait déjà que $y = b_j$.

C'est donc l'incertitude propre à l'événement $x = a_k$ sachant que $y = b_j$.

Information mutuelle et conditionnelle

On peut alors calculer l'**entropie conditionnelle** :

$$H(X|Y) = \sum_{x}\sum_{y} I(x|y)p(x, y)$$

On déduit de $I(x, y) = I(x) - I\,(x|y)$ et de l'équation précédente que :

$$I\,(X, Y) = H\,(X) - H\,(X\,|Y)$$

Remarque : L'espérance de l'information fournie sur X par la réalisation de Y (information mutuelle entre X et Y) est égale à l'espérance de l'incertitude a priori sur X moins l'incertitude restante (a posteriori) sur X après observation de Y.

Source discrète sans mémoire

Définition

Une source est dite **discrète et sans mémoire** si :

- elle utilise un alphabet fini ;
- les symboles de source sont engendrés selon une distribution de probabilités fixée ;
- les symboles successifs sont statistiquement indépendants.

Remarque : On peut démontrer que la source discrète sans mémoire d'entropie H maximale est celle pour laquelle les m symboles de l'alphabet sont équiprobables.

Définition du codage de source

Soit A un alphabet de source fini et B un alphabet de code également fini. Le **codage de source** est une règle qui permet d'assigner à chaque symbole de source (de A) un mot-code constitué à l'aide des symboles de l'alphabet de code B.

On peut donc définir un code de source comme l'ensemble des mots-codes $\{K(a_1), \dots, K(a_m)\}$ où :

- $A = \{a_1, \dots, a_m\}$ est l'alphabet de source ($|A| = m$);
- K est la fonction faisant correspondre un mot-code à chaque symbole de source.

Remarque : Si B est binaire, le codage est dit binaire.

Code décodable sans ambiguïté, code séparable

Un code est **décodable sans ambiguïté** si, pour toute séquence de symboles de source de longueur finie, la séquence de symboles de code qui lui correspond est différente de toute autre séquence de symboles de code correspondant à toute autre séquence de source.

Cette propriété étant difficile a vérifier en pratique, on s'intéresse à une classe de code plus restrictive qui satisfait à la condition de non ambiguité : la classe des **codes séparables**.

Définition : Un code est dit **séparable** si aucun mot-code n'est le préfixe d'un autre mot-code.

Remarque : Les codes séparables sont décodables sans ambiguité et ont la propriété de permettre le décodage mot-code par mot-code, sans observer les mots-codes futurs (décodage en ligne ou instantané).

Codage de Shannon-Fano

Le codage de Shannon-Fano fournit des codes séparables efficaces. La méthode nécessite de connaître la distribution de probabilités d'émission des symboles de source a priori.
Procédure d'encodage :

1. Classer les symboles de source dans l'ordre décroissant de leur probabilité d'émission ;
2. Diviser l'ensemble des symboles de source en deux sous-ensembles qui doivent avoir (presque) la même probabilité totale d'émission (somme sur tous les symbole du sous-ensemble). Attribuer un "0" comme premier bit de chaque mot-code des membres du premier sous-ensemble et un "1" comme premier bit de chaque mot-code des membres du second sous-ensemble ;
3. Itérer le processus sur chaque sous-ensemble jusqu'à ce que les deux sous-ensembles obtenus ne possèdent plus qu'un seul symbole.

Codage de Shannon-Fano - **Exemple**

Soit une source définie par ses symboles et leur distribution de probabilités :

Symbole	1	2	3	4	5	6	7
Probabilité	0,4	0,1	0,1	0,1	0,1	0,1	0,1

Première division :

Symboles	1 et 2	3, 4, 5, 6 et 7
Probabilité	0,5	0,5
Code	0	1

Deuxième division :

Symboles	1	2
Probabilité	0,4	0,1
Code	00	01

Troisième division :

Symboles	3, 4 et 5	6 et 7
Probabilité	0,3	0,2
Code	10	11

Quatrième division :

Symboles	3 et 4	5
Probabilité	0,2	0,1
Code	100	101

Codage de Shannon-Fano - **Exemple**

Cinquième division :

Symboles	3	4
Probabilité	0,1	0,1
Code	1000	1001

Sixième division :

Symboles	6	7
Probabilité	0,1	0,1
Code	110	111

Codage final (un des codes possibles) :

Symbole	1	2	3	4	5	6	7
Code	00	01	1000	1001	101	110	111

Code optimal

Soit une source d'entropie H dont on code les mots émis avec K mots-codes de longueur l_k, chaque mot ayant la probabilité $p(k)$.

- La longueur moyenne (espérance mathématique) des mots-codes est :

$$\mu = \sum_{k=1}^{K} p(k) l_k$$

- Théorème fondamental du codage de source :
  Pour toute source discrète sans mémoire d'entropie H, on peut trouver un code séparable q-aire décodable sans ambiguité tel que :

$$\frac{H}{\lg(q)} \leq \mu \leq \frac{H}{\lg(q)} + 1$$

Code optimal – Efficacité d'un code

- L'**efficacité** d'un code est $\frac{H}{\mu \lg(q)}$.
- Un code séparable est **optimal** pour toutes les distributions de probabilités s'il n'existe aucun autre codage symbole par symbole qui possède une meilleure efficacité.
- Le codage binaire de Huffman fournit des codes séparables optimaux.

Dans l'exemple précédent (Shannon-Fano), on a :

- $\mu = 2{,}7$ bits et
- $H = 2{,}5219$ bits.
- Le code de cet exemple a donc une efficacité de 93,4%.

Calculs :

$$\mu = 2 * 0{,}4 + (2 + 4 + 4 + 3 + 3 + 3) * 0{,}1$$
$$H = -(0{,}4 * \lg(0{,}4) + 6 * 0{,}1 * \lg(0{,}1))$$

Codage de Huffmann
Vérifications à faire pendant la séance d'exercices

Codage final avec les mêmes probabilités (un des codes possibles) :

Symbole	1	2	3	4	5	6	7
Code	0	100	101	1100	1101	1110	1111

Efficacité : 97%

Le codage de canal

- On suppose ici que la source fournit une séquence de symboles binaires.
- Le **codage de canal** désigne les traitements effectués sur le signal pour lui permettre d'être transmis sur un canal imparfait.
- Parmi les traitements, on s'intéresse au codage permettant la **détection et la correction d'erreurs** à la réception. Ces traitements impliquent l'ajout d'information – **redondance** – qui permettra de déterminer la présence d'erreurs de transmission et la correction de ces erreurs.

Stratégies utilisées :

- détection d'erreur et retransmission (Automatic Repeat Request ARQ)
- détection et correction sans retransmission (Forward Error Correcting FEC)

Remarque : Dans ce chapitre, on s'intéressera seulement à la stratégie FEC.

Détection et correction d'erreurs

On distingue deux types de codes de détection et correction d'erreurs :

- Les **codes en blocs** (n, k) : les mots de k symboles binaires (blocs) à coder sont remplacés par des mots-codes de n symboles binaires $(n > k)$ composés des k symboles binaires et de $n - k$ symboles (redondance) calculés à partir de l'information du mot à coder. Le codage d'un bloc ne dépend pas des blocs précédents.
- Les **codes convolutifs** : des mots-codes de n symboles binaires sont associés aux mots de k symboles binaires mais les mots-codes dépendent du mot à coder **et** des mots qui le précèdent.

Remarque : Un codeur convolutif est une machine à états finis.

Distance de Hamming

- La **distance de Hamming** entre deux mots de n symboles binaires est le nombre de positions où ces mots ont des symboles différents.
- Si deux mots sont représentés par des vecteurs u et v en dimension n, alors :

$$d_H(u, v) = |\{i : u_i \neq v_i, 0 \leq i < n\}|$$

- La **distance minimale de Hamming** d'un code C est la distance minimale entre toutes les paires de mots-codes de C :

$$d_{\min}(C) = \min_{(u,v) \in C^2, u \neq v} d_H(u, v)$$

Remarque : La distance minimale de Hamming du code C est aussi appelée *distance du code* C.

Exemples

Un code correcteur d'erreurs très simple est le code **binaire de répétition** de longueur 3 :

- On répète 3 fois chaque bit d'information : 0 est codé 000 et 1 est codé 111.
- La distance de Hamming entre les mots-codes 000 et 111 est 3.
- La distance minimale de Hamming est aussi 3 (le code n'a que les deux mots-codes 000 et 111).

Détection et correction d'erreurs- Les codes en blocs

- La loi de codage définit la correspondance bijective entre les 2^k mots (de k symboles binaires) à coder et les 2^k mots-codes (de n symboles binaires, $n > k$) choisis parmi les 2^n mots de n symboles possibles. On notera $[\boldsymbol{n}, \boldsymbol{k}, \boldsymbol{d_{\min}}]$ un code en blocs de *longueur* n qui code k bits (de *dimension* k) et dont la distance est $d_{\min}$.
- Les $2^n - 2^k$ autres mots de n symboles non retenus comme mots-codes ne sont **jamais** utilisés.
- Les 2^k mots-codes retenus doivent être aussi différents les uns des autres que possible pour augmenter l'efficacité du code.
- Le *taux* ou *rendement* du code est $\frac{k}{n}$ ($\frac{1}{3}$ dans notre exemple précédent).

Capacité de détection et de correction d'un code

- La capacité de détection d'un code $[n, k, d_{\min}]$ est : $d_{\min} - 1$
- La capacité de correction d'un code $[n, k, d_{\min}]$ est : $\left\lfloor \frac{d_{\min}-1}{2} \right\rfloor$
- Dans le cas d'un code binaire de répétition de longueur 3, on a donc :
  - Le nombre d'erreurs détectables est : $d_{\min} - 1 = 2$
  - Le nombre d'erreurs pouvant être corrigées est : $\left\lfloor \frac{d_{\min}-1}{2} \right\rfloor = 1$.

Exemple de codage en blocs

Un récepteur reçoit le mot $u = (u_1, \dots, u_n)$.

Pour détecter la présence d'erreurs :

- On examine la redondance **calculée par le récepteur** sur les k premiers symboles, $r = f(u_1, \dots, u_k)$.
- Si r n'est pas égale à la redondance reçue $(u_{k+1}, \dots, u_n)$, le mot u n'appartient pas au code et on a donc **détecté une erreur**.
- On appelle syndrome : $s = r - (u_{k+1}, \dots, u_n) \bmod 2$.
  Si le syndrome est nul, on n'a pas d'erreur de transmission.

Principes

- Si le syndrome n'est pas composé que de 0, cela signifie qu'il y a eu une ou plusieurs erreurs de transmission.
- Si le syndrome est constitué uniquement de 0, cela signifie qu'il n'y a eu aucune erreur de transmission ou qu'il y en a eu un nombre strictement supérieur à la capacité de **détection**.
- Si le syndrome n'est pas composé que de 0, il faut chercher s'il existe un mot-code à une distance du mot reçu inférieure ou égale à la capacité de **correction**. Si un tel mot existe, il est unique et c'est le mot envoyé (sauf s'il y a eu strictement plus d'erreurs que la capacité de **correction**). Si tous les mots-codes sont à une distance du mot reçu strictement supérieure à la capacité de **correction**, on ne peut pas corriger les erreurs.

Exemple de codage en blocs

Soit $x = (x_1, x_2, x_3, x_4)$ un mot à transmettre auquel on ajoute 3 symboles y_1, y_2 et y_3 pour former un mot-code.
La fonction f suivante permet de calculer les symboles de contrôle (addition modulo 2) :

$$\begin{aligned} y_1 &= x_2 + x_3 + x_4 \\ y_2 &= x_1 + x_3 + x_4 \\ y_3 &= x_1 + x_2 + x_4 \end{aligned}$$

D'où :

- l'information $x = (x_1, x_2, x_3, x_4)$.
- la redondance $y = f(x) = (y_1, y_2, y_3)$.
- le mot-code $c = (x_1, x_2, x_3, x_4, y_1, y_2, y_3)$.

Rappel : Si a et b sont des bits alors $a + b \bmod 2 = a - b \bmod 2 = a \oplus b$

Questions (à resoudre pendant la séance d'exercices)

1. Quel est l'ensemble des mots-codes ?
2. Déterminer la longueur, la dimension et la distance du code.
3. Calculer le taux, la capacité de détection et la capacité de correction du code.
4. On reçoit le mot 1000110. Une erreur de transmission s'est-elle produite ?
5. Quelle est la valeur du syndrome ?
6. Peut-on corriger l'erreur ?

Codes linéaires

Un code $[n, k, d_{\min}]$ est appelé **code linéaire** s'il existe une matrice M de dimension $k \times n$ permettant d'obtenir les mots-codes par produit matriciel à partir des mots de source x, *i.e.* :

$$C = \{y, y = x.M, x \in \{0,1\}^k\}$$

Le code de Hamming $[7,4,3]$ précédent est un code linéaire.

Trouver (**réponse pendant la séance d'exercices**) la matrice de ce code.

Exercice 1

Soit $x = (x_1, x_2, x_3)$ un mot à transmettre auquel on ajoute 2 symboles y_1 et y_2 pour former un mot-code. La fonction f suivante permet de calculer les symboles de contrôle (addition modulo 2) :

$$\begin{aligned} y_1 &= x_1 \\ y_2 &= x_2 + x_3 \end{aligned}$$

1. Quel est l'ensemble des mots-codes ?
2. Déterminer la longueur, la dimension et la distance du code.
3. Calculer le taux, la capacité de détection et la capacité de correction du code.
4. On reçoit le mot 10110. Une erreur de transmission s'est-elle produite ?
5. Quelle est la valeur du syndrome ?
6. Peut-on corriger l'erreur ?

Exercice 2

Soit une source définie par ses symboles et leur distribution de probabilités :

Symbole	*a*	*b*	*c*	*d*	*e*	*f*	*g*
Probabilité	0,45	0,2	0,15	0,1	0,05	0,03	0,02

1. Déterminer un codage de longueur fixe et calculer son efficacité.
2. Déterminer un codage de Shannon-Fano et calculer son efficacité.
3. Déterminer un codage de Huffmann et calculer son efficacité.