— Algorithmique —

Cours donné par Stéphane THOMASSÉ (Bureau dans l'aile Sud–Ouest).

Qu'est ce qu'il y aura à l'exam ? (MCC = Modalité de Contrôle de Connaissances) :

$$2 \text{ devoirs} + 1 \text{ partiel} + 1 \text{ exam.}$$

***Office hour(s)* :** Lundi 15h30 – 16h30

I. Pré-introduction.

Qu'est ce que l'algorithmique ? C'est résoudre des problèmes efficacement. L'algorithmique, c'est le côté *clair* de l'informatique. On manipule des « bornes sup ».

Le côté *sombre*, c'est la complexité, l'étude des classes de complexité (comme **P** et **NP** par exemple). Ici, on cherche à montrer que résoudre efficacement n'est pas possible. On manipule des « bornes inf ».

Dans ce cours, on se place à la bordure entre ces deux côtés.

I.1. Problèmes.

Après les soutenances de stage de l'année dernière, le constat est simple : il est important de respecter une structure, où l'on donne
- nom du problème,
- entrée du problème et la taille de l'entrée,
- sortie du problème.

Par exemple, pour un entier n, la taille est de $\log_2 n$ en le représentant en binaire.

La sortie, elle peut être de trois types :

- un mot sur un alphabet $\mathscr{A}$, c'est un problème de ***construction*** ;
- un entier, c'est un problème d'***optimisation*** ;
- un booléen, c'est un problème de ***décision***.

Ces trois types ne sont pas les mêmes. Pour cela, on étudie le problème du « Voyageur de commerce » (noté *TSP* par la suite). Le but de ce problème est de se déplacer en un certain nombre de villes le plus rapidement possible.

En version constructif :

TSP_Constructif :
Entrée. Un graphe $G = (V, E)$ et une fonction $\ell : E \to \mathbb{N}$ la longueur d'une arête.
Sortie. Un cycle qui passe une fois exactement par chaque sommet en minimisant la longueur totale, et Faux sinon.

En version optimisation :

TSP_Optimisation :
Entrée. Un graphe $G = (V, E)$ et une fonction $\ell : E \to \mathbb{N}$ la longueur d'une arête.
Sortie. La longueur minimale d'un cycle qui passe une fois exactement par chaque sommet, et ∞ si un tel cycle n'existe pas.

En version décision :

TSP_Décision :
Entrée. Un graphe $G = (V, E)$, une fonction $\ell : E \to \mathbb{N}$ la longueur d'une arête et un entier t.
Sortie. Vrai s'il un cycle qui passe une fois exactement par chaque sommet de longueur inférieure ou égale à t (et Faux sinon).

On note $\mathrm{Pb}_1 \leqslant \mathrm{Pb}_2$ si Pb_1 se réduit polynomialement à Pb_2. Clairement, on a la chaîne de réductions :

TSP_Décision $\leqslant$ TSP_Optimisation $\leqslant$ TSP_Construction.

Proposition

On a TSP_Optimisation $\leqslant$ TSP_Décision.

Preuve. On réalise une recherche dichotomique sur l'ensemble $\{0, \ldots, \sum_{e \in E} \ell(e)\}$ en utilisant le problème TSP_Décision. On trouve la longueur minimale en $\mathrm{O}\big(\log_2\big(\sum_{e \in E} \ell(e)\big)\big)$ étapes. □

Proposition

On a TSP_Construction $\leqslant$ TSP_Optimisation.

Preuve. On construit l'algorithme ci-dessous.

```
L ← TSP_Optimisation (G, ℓ)
Si L = ∞ alors
    Retourner Faux
Sinon
    Pour toute arête e ∈ E faire
        Si TSP_Optimisation (G \ e, ℓ) = L alors
            G ← G \ e
    Retourner E(G)
```

Remarquons qu'à la fin, il ne peut rester que les arêtes d'un cycle optimal.

En sortie, nous savons qu'il reste une solution C de longueur L, et qu'il n'existe qu'un seul cycle. □

Les trois versions sont équivalentes polyomialement. **On va donc se restreindre au problème de décision.** En effet, les problèmes de construction, d'optimisation et de décision sont généralement équivalents polynomialement.

Une exception notable : le problème « Collision Somme Tiroirs » (abrégé par « CST »).

CST_DÉCISION :
Entrée. $S = \{x_1, \ldots, x_n\}$ un ensemble de n entiers strictement positifs tel que $\sum_{i=1}^n x_i < 2^n$.
Sortie. Vrai s'il existe deux ensembles $X \neq Y \subseteq S$ tels que $\sum_{x \in X} x = \sum_{y \in Y} y$.

En effet, le problème a une solution en $\mathrm{O}(1)$:

Retourner Vrai

La validité de cet algorithme est assurée par deux faits :
- il existe 2^n sous-ensembles possibles ;
- le nombre de sommes possibles est inférieur à 2^n.

Ces types de problèmes sont basés sur : le lemme des tiroirs, la parité, les points fixes.

Définition

Un problème peut-être assimilé à l'ensemble des mots en entrée qui admettent Vrai en sortie.

Un langage L est une partie de $\mathscr{A}^\star$, l'ensemble des mots sur l'alphabet $\mathscr{A}$.

Il y a une équivalence entre

$$\text{Problème} \longleftrightarrow \text{Langage}.$$

Il n'y a qu'un seul type de problème :

L-DÉCISION :
Entrée. un mot $M \in \mathscr{A}^\star$
Sortie. Vrai si $M \in L$.

I.2. Résoudre efficacement.

Il apparaît une question de *modèle de calcul*. Le premier modèle est l'*automate* auquel on rajoute de la mémoire pour obtenir une machine de Turing. Cependant, dans la machine de Turing, il y a une distinction entre déterministe et non déterministe. Ceci crée les classe de complexité **P** et **NP** : ceci correspond aux machines qui s'exécutent en un nombre polynomial d'étapes.

On étudie quatre exemples types de problèmes de décisions :

CPB :
Entrée. Un graphe $G = (V, E)$ biparti
Sortie. Vrai s'il existe un couplage parfait

Quelques définitions :

Biparti. il existe une partition A, B de V telle que toute arête est entre A et B ;

Couplage. ensemble d'arêtes deux à deux disjointes ;

Parfait. couvre tous les sommets.

Le problème CouplageParfaitBiparti (qu'on notera CPB) est équivalent au problème SousMatriceDePermutation (qu'on notera SSMP).

SSMP :
Entrée. Une matrice $n \times n$ à coefficients dans $\{0, 1\}$.
Sortie. Vrai s'il existe une sous-matrice de permutation :

$$\exists \sigma \in \mathfrak{S}_n, \forall i \in [\![1, n]\!], m_{i,\sigma(i)} = 1$$

L'équivalence vient de l'analyse d'un couplage dans la matrice d'adjacence du graphe. Cela correspond à une permutation.

D'autres problèmes intéressants.

Premier :
Entrée. Un entier n
Sortie. Vrai si n est premier.

Somme : | **Entrée.** Ensemble d'entiers S et un entier t
Sortie. Vrai s'il existe $X \subseteq S$ tel que $\sum X = t$.

Post : | **Entrée.** Sept dominos, chacun avec deux mots binaires :

$$\underbrace{\boxed{\begin{array}{c}\texttt{100}\\ \hline \texttt{001}\end{array}}}_{D_1}, \ldots, \underbrace{\boxed{\begin{array}{c}\texttt{110}\\ \hline \texttt{111}\end{array}}}_{D_7}$$

Sortie. Vrai si on peut trouver une suite de dominos avec répétition où le mot du haut est égal au mot du bas.

Par exemple, la suite ci-dessous est une solution de Post :

$$\boxed{\begin{array}{c}\texttt{0}\\ \hline \texttt{00}\end{array}}\boxed{\begin{array}{c}\texttt{00}\\ \hline \texttt{0}\end{array}}.$$

Certains de ces problèmes sont difficiles, mais ils ne sont pas tous *aussi* difficiles.

- le problème CPB est (facilement) dans **P** ;
- le problème Premier est dans **P** ;
- le problème Somme est **NP**-complet ;
- le problème Post est indécidable.

II. Introduction : la science des arbres.

On se place dans une situation, un *toy problem*. On veut faire un fondant au chocolat, et on essaie de découper une tablette au chocolat. Comment de découpes pour une tablette de chocolat 6×4 ?

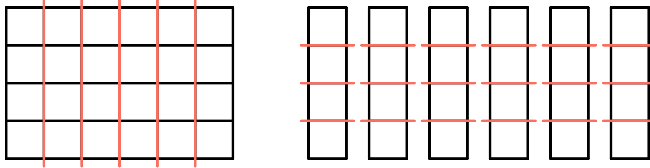


Figure 1 – Une tablette de chocolat

On a 23 découpes, qu'on commence ligne par ligne, ou colonne par colonne :

$$5 + 3 \times 6 \quad \text{ou} \quad 3 + 4 \times 5.$$

Une bonne pratique est de se placer dans une instance simple.

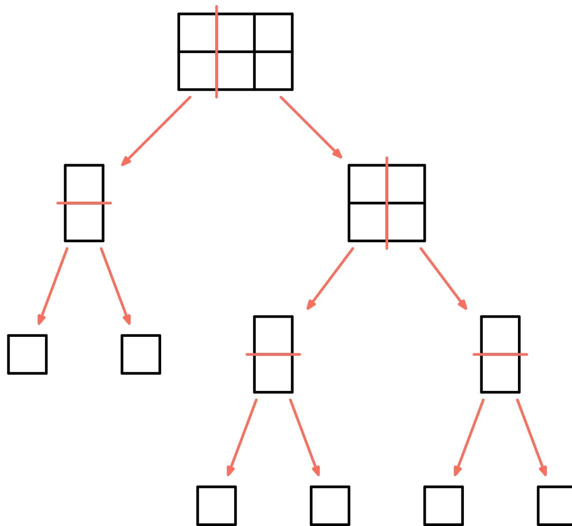


Figure 2 – Arbre de découpe

On représente la situation par un arbre de découpe :
- un nœud interne correspond à une découpe ;
- une feuille correspond à un carré 1×1 de chocolat ;
- l'arbre est un arbre binaire.

Le nombre de feuilles est donc exactement égale au nombre de carrés de chocolat.

Proposition

Le nombre de feuilles dans un arbre binaire, c'est un de plus que le nombre de nœuds internes.

On peut le démontrer par induction, mais pour comprendre l'intuition, le mieux, c'est une bijection. Une bijection serait par exemple : d'un nœud interne, on emprunte le chemin $D \cdot G^\star$ sous forme d'expression régulière. Il ne manque qu'un seul nœud, celui tout à gauche.

III. Paradigmes : *diviser pour régner*.

III.1. Nombre de multiplications.

III.1.a. L'exponentiation.

Exponentiation : **Entrée.** Un réel x et un entier $n > 0$
Sortie. Le calcul de x^n

On **ne fait pas** un calcul en $\mathrm{O}(n)$. On fait un algorithme diviser pour régner, l'*exponentiation rapide*.

Si $n = 1$ **alors**
  Retourner x
Sinon si n *est pair* **alors**
  Retourner $\left(x^{n/2}\right)^2$
Sinon
  Retourner $\left(x^{n/2}\right)^2 \cdot x$

On a une complexité en $\Theta(\log n)$ multiplications : il s'agit
- du nombre de bits dans l'expression de n moins 1 ;
- du nombre de « 1 » dans l'expression binaire de n moins 1.

Par exemple, pour 5, on a trois multiplications : $x^5 = (x^2)^2 \cdot x$. Ceci correspond bien : $5 \underset{2}{=}$ 101 et le nombre de multiplications est donné par la formule $3 - 1 + 2 - 1$.

Sujet intéressant à étudier : le *logarithme discret*.

III.1.b. Karatsuba & Strassen.

III.1.b.i. Algorithme de Karatsuba.

La multiplication d'entiers appris à l'école est en $\Theta(n^2)$ en nombre de multiplications.

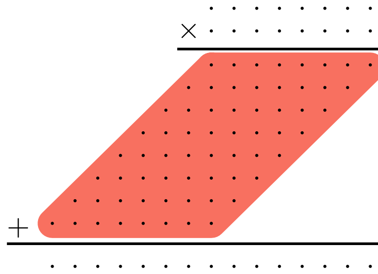


Figure 3 – Algorithme de multiplication vu à l'école

Dans un cours, KOLMOGOROFF pense que l'algorithme qu'on utilise depuis plusieurs millénaires est optimale. Mais, KARATSUBA lui donne tord.

Avant d'attaquer la multiplication d'entiers, on s'intéresse aux polynômes. Étant donné deux polynômes $A(X) = a_n X^n + \cdots + a_1 X + a_0$ et $B(X) = b_n X^n + \cdots + b_1 X + b_0$.

Comment calculer $C(X) = A(X) \times B(X)$?

On s'intéresse à un problème plus simple pour commencer :

$$\begin{aligned}(aX+b)\times(cX+d) &= acX^2 + (ad+bc)X + bd\\ &= \boxed{ac}\,X^2 + \Big(\boxed{ac} + \boxed{bd} + \boxed{(a-b)(c-d)}\Big)X + \boxed{bd}.\end{aligned}$$

On remarque qu'on peut faire ce calcul en n'utilisant que *trois* multiplications. Comment utiliser ce résultat pour généraliser ?

Il faut que le problème s'y prête : on verra des problèmes où une stratégie diviser pour régner ne s'y prête pas. En effet, il faut qu'on puisse couper une instance ce qui fonctionne bien avec des polynômes.

Soient $P(X)$ et $Q(X)$ deux polynômes de degré n (pair). On pose

$$P(X) = P_1(X)X^{\frac{n}{2}} + P_0(X),$$
$$Q(X) = Q_1(X)X^{\frac{n}{2}} + Q_0(X).$$

Ainsi, pour calculer $P \times Q$, on utilise :

$$P \times Q = \boxed{P_1Q_1}\, X^n + \Big(\boxed{P_1Q_1} + \boxed{P_0Q_0} + \boxed{(P_1 - P_0)(Q_1 - Q_0)}\Big)X^{\frac{n}{2}} + \boxed{P_0Q_0}.$$

Quel est le coût en multiplications ? Pour cela, on peut utiliser le *master theorem* (vu plus tard) ou dessiner l'arbre d'appels.

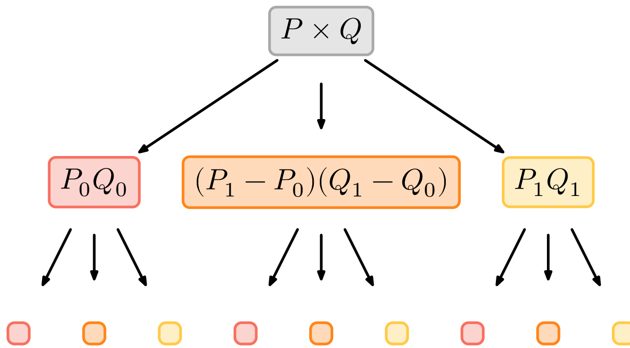


Figure 4 – Arbre d'appels pour la multiplication de polynômes

Dans l'arbre d'appels, il y a une multiplication par nœud. L'arbre d'appel, vu qu'on divise par deux le degré du polynôme à chaque étape, a $\log_2 n$ niveaux.

Au total, le nombre de nœuds est $\mathrm{O}\big(3^{\log_2 n}\big) = \mathrm{O}\big(n^{\log_2 3}\big)$.

On effectue donc $\mathrm{O}\big(n^{\log_2 3}\big)$ multiplications.

III.1.b.ii. Algorithme de Strassen.

Strassen utilise la même idée que Karatsuba pour le calcul matriciel. Il remarque que, pour multiplier de matrices 2×2, seules *sept* produits sont nécessaires. En procédant au calcul de matrice par blocs, il peut calculer le produit de deux matrices $n \times n$. Par conséquent, pour calcule deux matrices $n \times n$ en

$$\mathrm{O}\big(7^{\log_2 n}\big) = \mathrm{O}\big(n^{\log_2 7}\big) = \mathrm{O}\big(n^{2.80\,\cdots}\big).$$

On peut pousser encore plus loin l'argument. On note ω la meilleure complexité de produit de matrice (*i.e.* il existe un algorithme de complexité $\mathrm{O}(n^\omega)$, mais pas en $\mathrm{O}(n^{\omega-\varepsilon})$, quel que soit $\varepsilon > 0$). On se situe $\omega \in [2, 3]$. Actuellement, on a $\omega \leqslant 2{,}38$.

III.1.b.iii. Multiplication d'entiers en $n \log n$.

On travaille dans deux mondes : le monde des coefficients polynômiaux, et le monde des évaluations.

Au lieu de calculer $A \times B = C$ directement, on évalue les polynômes à $2n + 1$ valeurs différentes $v_1, \ldots, v_{2n+1}$. On réalise le produit des valeurs évaluées, que l'on peut interpréter pour obtenir les coefficients de C (opération linéaire).

Sujet intéressent à regarder : *FFT* (*Fast Fourier Transform*).

III.1.b.iv. Algorithme de Strassen (suite).

Strassen interprète l'opération produit comme un tenseur. Son idée est la suivante : il faut construire un tenseur (une matrice nD, en l'occurrence ici 3D) qui représente le produit de matrices.

On peut l'illustrer avec Karatsuba : on calcule

$$(a_1 X + a_0)(b_1 X + b_0) = c_2 X^2 + c_1 X + c_0.$$

On construit le tenseur $\boldsymbol{K} = \big(p_{i,j,k}\big)$ de dimension $2 \times 2 \times 3$, où l'on a $p_{i,j,k} = 1$ si le produit $a_i b_j$ est un terme de c_k.

Le tenseur s'écrit donc :

c_0:

	b_1	b_0
a_0	1	0
a_1	0	0

c_1:

	b_1	b_0
a_0	0	1
a_1	1	0

c_2:

	b_1	b_0
a_0	0	0
a_1	0	1

Étant donnés trois vecteurs $\vec{\boldsymbol{U}} = (u_1 \quad \dots \quad u_\ell)$, $\vec{\boldsymbol{V}} = (v_1 \quad \dots \quad v_m)$ et $\overrightarrow{\boldsymbol{W}} = (w_1 \quad \dots \quad w_n)$, on construit le tenseur $\boldsymbol{T}$ de dimension $\ell \times m \times n$ en posant $\boldsymbol{T} = \vec{\boldsymbol{U}} \otimes \vec{\boldsymbol{V}} \otimes \overrightarrow{\boldsymbol{W}} = \big(t_{i,j,k}\big)$, où $t_{i,j,k} = U_i V_j W_k$, idem que pour les matrices de rang 1. Le tenseur $\boldsymbol{T}$ est de rang 1.

Définition

Le rang d'un tenseur $\boldsymbol{A}$ est le plus petit nombre de tenseurs de rang 1 dont la somme vaut $\boldsymbol{A}$.

Théorème (Strassen)

Le rang du tenseur de la multiplication de matrices 2×2 est inférieur ou égal à 7.

Par exemple, pour le tenseur de multiplication de polynômes $\boldsymbol{K}$, il est de rang 3. En effet, c'est la somme des trois tenseurs :

(1)

1	0
0	0

1	0
0	0

0	0
0	0

(2)

0	0
0	0

0	0
0	1

0	0
0	1

(3)

0	0
0	0

−1	1
1	−1

0	0
0	0

Sujet intéressant à regarder : *Alpha Tensor* dans *Nature*.

Quelques remarques sur les tenseurs :

- calculer le rang d'une matrice, c'est un problème dans **P**, calculer le rang d'un tenseur, c'est un problème **NP**-dur ;
- si $(\boldsymbol{M}_i)_{i \in \mathbb{N}}$ converge vers $\boldsymbol{M}$ une matrice $n \times n$ de rang plein n, alors la suite est ultimement a rang plein[1] ;
- dans le cas général des matrices, la limite du rang d'une suite de matrice est supérieur ou égal au rang de la limite ;
- mais, avec les tenseurs, il est possible d'avoir une limite du rang strictement inférieur au rang de la limite[2] ;
- les rangs dans $\mathbb{C}$ et dans $\mathbb{R}$ d'un tenseur réel peuvent être différents ;
- pour discuter de tenseurs, parler avec Pascal KOIRAN.

III.1.c. Plus courts chemins.

On s'intéresse au problème ALLPATHSSHORTESTPATHS, abrégé par APSP :

APSP :
Entrée. Une matrice $\boldsymbol{D}$ de taille $n \times n$ à valeurs dans $\mathbb{N} \cup \{+\infty\}$.
Sortie. La matrice des plus courtes distances.

Par exemple, dans le graphe ci-dessous, on note $\boldsymbol{D}$ sa représentation matricielle. On donne $\boldsymbol{M}$ la matrice solution du problème APSP.

[1] On dit qu'une matrice a un rang plein si, et seulement si, $\det M \neq 0$. L'ensemble des matrices de rang plein est la pré-image par det (continue) de $\mathbb{R} \setminus \{0\}$.

[2] Un exemple :

$$\underbrace{\left[\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & 1 \\ 1 & \varepsilon \end{pmatrix}\right]}_{\text{tenseur de rang 2}} \underset{\varepsilon \to 0}{\longrightarrow} \underbrace{\left[\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}\right]}_{\text{tenseur de rang 3}}$$

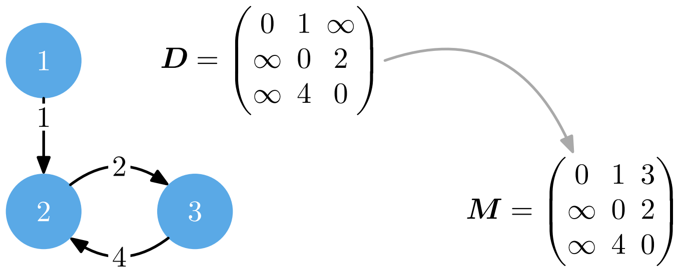


Figure 5 – Exemple de graphe et solution de APSP

Calculons le « carré » de la matrice $\boldsymbol{D}$. On a $\boldsymbol{D}^2 = \left(d_{i,j}^{(2)}\right)$, où

$$d_{i,j}^{(2)} = \min_k (d_{i,k} + d_{k,j})$$

On obtient les plus courts chemins de longueur inférieur à 2.

La matrice cherchée cherchée peut être obtenir $\lceil \log_2 n \rceil$ itération du carrés, d'où la complexité en $\mathrm{O}(n^3 \log_2 n)$.

Si on peut franchir la barrière du n^3 pour le produit *tropical*, et on obtient un algorithme en $\mathrm{O}(n^{3-\varepsilon})$ pour APSP.

Quelle est la complexité du calcul en algèbre (en nombre d'opérations min, $\times$, $+$) ? Elle est conjecturée d'être cubique.

III.2. Nombre de comparaisons.

On s'intéresse au tri d'un tableau $\boldsymbol{T}[1..n]$ d'entiers.

III.2.a. Calcul du minimum d'un tableau.

On peut calcule le minimum d'un tableau en $n-1$ comparaisons de la forme : « $\boldsymbol{T}[i] < \boldsymbol{T}[j]$ ».

L'approche diviser pour régner permet de calculer le minimum en $n-1$ comparaisons avec une technique de *tournoi*.

Un algorithme classique de calcul de minimum par tournois est celui ci-dessous.

Algorithme Minimum($\boldsymbol{T}, n$)

Créer un tableau $\boldsymbol{R}[1..2n-1]$
Copier $\boldsymbol{T}$ sur $\boldsymbol{R}[n..2n-1]$

Pour *i allant de $n-1$ à 1* **faire**
 $\boldsymbol{R}[i] \leftarrow \min(\boldsymbol{R}[2i], \boldsymbol{R}[2i+1])$

Retourner $\boldsymbol{R}[1]$

Par exemple, on trie le tableau $[3, 2, 6, 4, 1, 5]$. Pour cela, on crée l'arbre tournoi représenté dans la figure ci-après. Cet arbre est créé à partir du tableau ci-dessous :

$$[1, 1, 2, 4, 1, 3, 2, 6, 4, 1, 5].$$

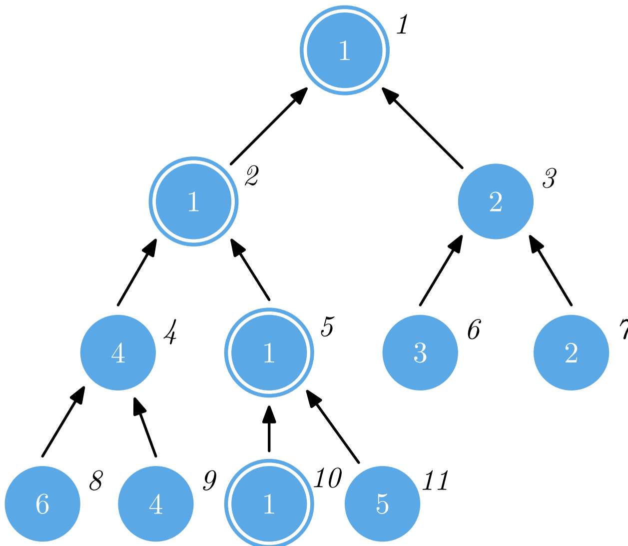


Figure 6 – Arbre tournoi pour le calcul du minimum de $[6, 4, 1, 5, 3, 2]$

III.2.b. | Algorithme de tri fusion.

On divise le tableau $\boldsymbol{T}[1..n]$ en deux sous-tableaux de taille $\lfloor n/2 \rfloor$ et $\lceil n/2 \rceil$. On appelle récursivement l'algorithme de tri, et on *fusionne*.

Dans le pire cas (un tableau alternant grande valeur et petite valeur), la fusion implique $n - 1$ comparaison.

Au final, l'algorithme a une complexité en $\mathrm{O}(n \log n)$. En effet, le « vrai » coût pire cas vérifie

- $c(1) = 0$
- $c(n) = c(\lfloor \frac{n}{2} \rfloor) + c(\lceil \frac{n}{2} \rceil) + n - 1$.

On peut montrer que $c(n) = \sum_{i=1}^{n} \lceil \log_2 i \rceil$.

Proposition

Le meilleur algorithme de tri dans le pire des cas effectue $\lceil \log_2(n!) \rceil = \left\lceil \sum_{i=1}^{n} \log_2 i \right\rceil$ comparaisons.

Preuve. On construit l'arbre ci-dessous. Dans les nœuds de l'arbre, on indique le nombre de permutations vérifiant les conditions (les questions). Ces permutations, on les appelles « solutions ».

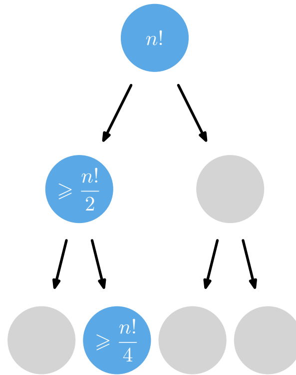


Figure 7 – Arbre d'appels pour la multiplication de polynômes

Une comparaison de la forme $\boldsymbol{T}[i] < \boldsymbol{T}[j]$ coupe l'espace des solutions en deux. Comme on se place dans le pire cas, on choisit la branche avec le nombre de solutions la plus élevée.

On peut montrer que, à l'étape k, il reste $n!/2^k$ solutions possibles. On ne peut conclure que lorsque le nombre de solutions est inférieur ou égale à 1.

On ne peut pas conclure avant :

$$\frac{n!}{2^k} \leqslant 1 \implies n! \leqslant 2^k \implies \lceil \log_2 n! \rceil \leqslant k.$$

□

III.2.c. Calcul de médiane.

Soit $\boldsymbol{T}$ un tableau de n valeurs, et on cherche la valeur médiane $\boldsymbol{T}_{\text{triée}}[\lfloor n/2 \rfloor]$. On peut le faire en $\mathrm{O}(n)$ avec l'algorithme nommé BFPRT.[3]

Commençons par généraliser : on veut calculer la k-ième valeur du tableau $\boldsymbol{T}_{\text{triée}}$.

Première idée : un algorithme randomisé. Oui, mais non. On préfèrerai un algorithme non probabiliste.

Deuxième idée : forcer la chance (l'algorithme sera écrit proprement la prochaine fois)

Algorithme BFPRT

Entrée. $\boldsymbol{T}[i..j]$ de taille n et $k \in [\![1, n]\!]$.
Sortie. Le k-ième élément de $\boldsymbol{T}_{\text{trié}}$

On divise $\boldsymbol{T}$ en $n/5$ blocs de taille 5.
On extrait $m_1, \ldots, m_{n/5}$ les éléments médiants de chaque bloc.
On construit $\boldsymbol{M} = \left[m_1, \ldots, m_{n/5}\right]$.
On calcule le $(n/10)$-ième élément de $\boldsymbol{M}[1..n/5]$).
On partitionne $\boldsymbol{T}$ avec m : $\boldsymbol{T} = [\underbrace{t_1, \ldots, t_{\ell-1}}_{\leqslant m}, \underbrace{t_\ell}_{=m}, \underbrace{t_{\ell+1}, \ldots, t_n}_{\geqslant m}]$.

Si $\ell = k$ **alors**
  Retourner m
Sinon si $\ell > k$ **alors**
  Retourner le k-ième élément de $\boldsymbol{T}[1..\ell - 1]$ $(\star)$
Sinon
  Retourner le $(k - \ell)$-ième élément de $\boldsymbol{T}[\ell + 1..n]$ $(\star\star)$

[3] Blum-Floyd-Pratt-Rivest-Tarjan

Pour trouver la complexité, il faut se demander ce quelle taille ont les tableaux aux appels ($\star$) et ($\star\star$).

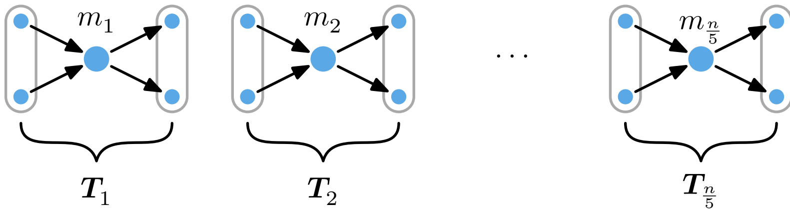


Figure 8 – Les éléments médians des $\frac{n}{5}$ sous-tableaux de 5 éléments

La moitié des médianes $m_1, \ldots, m_{\frac{n}{5}}$ est inférieure ou égale à la médiane m. Et, dans chaque sous-tableau, trois éléments sur les cinq sont donc assurés d'être inférieurs strictement à m. Ceci assure donc que $\frac{7}{10}n$ éléments sont supérieurs ou égaux à n ($\frac{7}{10} = 1 - (\frac{1}{2} \cdot \frac{3}{5})$). On élimine $\frac{3}{10}n$ éléments à chaque étape.

Dans les appels ($\star$) et ($\star\star$), on n'a donc que 70 % des éléments du tableau dans l'appel récursif.

Pour un nœud interne dans l'arbre des appels récursif, le coût est en $\mathrm{O}(n)$. La fonction de complexité $C(n)$ vérifie :

$$\begin{aligned} C(n) &\leqslant C\left(\frac{n}{5}\right) + C\left(\frac{7n}{10}\right) + cn \\ &\leqslant C\left(\frac{9}{10}n\right) + cn, \end{aligned}$$

par « super additivité » de la fonction C.[4] D'où,

$$C(n) \leqslant cn + \left(\frac{9}{10}\right)cn + \left(\frac{9}{10}\right)^2 cn + \cdots \leqslant 10cn.$$

[4]L'hypothèse de la super additivité est faite dans un raisonnement par analyse–synthèse. On vérifie aisément cette hypothèse après.

Remarque (Algorithme Las Vegas)

L'algorithme randomisé pour le calcul du k-ième plus petit élément peut se retrouver assez simplement en supprimant la division en 5 blocs. Il est presque sûrement en temps linéaire.

Algorithme BFPRT Randomisé

Entrée. $\boldsymbol{T}[i..j]$ de taille n et $k \in [\![1, n]\!]$.
Sortie. Le k-ième élément de $\boldsymbol{T}_{\text{trié}}$

On tire m un élément de $\boldsymbol{T}$ au hasard.
On partitionne $\boldsymbol{T}$ avec m : $\boldsymbol{T} = [\underbrace{t_1, \ldots, t_{\ell-1}}_{\leqslant m}, \underbrace{t_\ell}_{=m}, \underbrace{t_{\ell+1}, \ldots, t_n}_{\geqslant m}]$.

Si $\ell = k$ **alors**
  Retourner m
Sinon si $\ell > k$ **alors**
  Retourner le k-ième élément de $\boldsymbol{T}[1..\ell - 1]$
Sinon
  Retourner le $(k - \ell)$-ième élément de $\boldsymbol{T}[\ell + 1..n]$

Remarque (Algorithme Monte-Carlo)

Avec un algorithme randomisé alternatif, on tire $\sqrt{n}$ valeurs du tableau $\boldsymbol{T}$ au hasard, et on calcule la « vraie » médiane m. Il est donc important de se demander avec quelle probabilité le choix de m est le bon.

Théorème (Chernoff – application à notre algorithme)

Quel que soit $\varepsilon > 0$, il existe une constante $c > 1$ et un rang n_0 tels que, au delà de $n \geqslant n_0$,

$$\mathrm{P}\big(M \notin [(^1/_3 - \varepsilon)\sqrt{n}, (^1/_3 + \varepsilon)\sqrt{n}]\big) \leqslant \frac{1}{c^{\sqrt{n}}},$$

où l'on a noté P l'ensemble des valeurs $\leqslant m$ tirées aléatoirement.

« La probabilité d'échec du choix de valeurs est exponentiellement plus faible en la taille de l'échantillon. »

Pourquoi ne pas faire avec des blocs de 3 dans l'algorithme BFPTR ? Et bien, on calcule

$$C(n) \leqslant C\left(\frac{n}{3}\right) + C\left(\left(1 - \frac{1}{6} - \frac{1}{6}\right)n\right) + cn \leqslant \underbrace{C\left(\frac{n}{3}\right) + C\left(\frac{2}{3}n\right)}_{\leqslant C(n)} + cn.$$

Une complexité linéaire n'est donc pas assurée...

Théorème (Théorème Maître/*Master Theorem*)

Lorsqu'on a une relation de récurrence sur $T(n)$ (par exemple, le nombre de multiplications/comparaisons) de la forme

$$T(n) = aT\left(\frac{n}{b}\right) + \Theta\big(n^c \log^d n\big),$$

alors, en posant $\omega = \log_b a$, on a

$$T(n) = \begin{cases} \Theta(n^c \log^d n) & \text{si } \omega < c \\ \Theta(n^c \log^{d+1} n) & \text{si } \omega = c \\ \Theta(n^\omega) & \text{si } \omega > c. \end{cases}$$

IV. Algorithmes gloutons.

IV.1. Introduction.

Avec un algorithme glouton, la stratégie est la suivante.

> « On construit une solution pas à pas, en ne *remettant pas en cause* les choix précédents. »

Exemple

On construit $S \subseteq \{0, \dots, n\} = [n] \cup \{0\}$ de cardinal maximal sans suite arithmétique de longueur trois $(x,\ x+a,\ x+2a)$. Par exemple, en commençant à zéro, on peut construire

$$S = \{0, 1, 3, 4, 9, \dots\}.$$

On laisse en exercice la caractérisation de l'ensemble.

Exemple

Flots :

Entrée.
- Un graphe orienté $D = (V, A)$,
- Une source s et un terminal t ;

Sortie. Un ensemble S maximum de $(s-t)$-chemins avec des arcs disjoints.

On part d'un algorithme glouton.

Algorithme glouton

$S \leftarrow \emptyset$

Tant que *il existe un* $(s-t)$*-chemin* P **faire**
- $S \leftarrow S \cup \{P\}$
- $A \leftarrow A \setminus \mathrm{arcs}(P)$

Ce glouton n'est pas optimal. Certaines arêtes le gène pour atteindre le flot optimal.

Et, on ajoute deux lignes *mystérieuses*. Ces deux lignes permettent de filtrer les arêtes qui gênent l'algorithme glouton original.

Le fonctionnement de ces deux lignes apparaîtra plus clairement dans l'exemple après.

Algorithme Floyd-Fulkerson

Tant que *il existe un* $(s-t)$*-chemin* P **faire**
- Inverser les arcs de P

Effacer les arcs inversés un nombre pair de fois,
et inverser les autres. △ $(\star)$

$S \leftarrow \emptyset$

Tant que *il existe un* $(s-t)$*-chemin* P **faire**
- $S \leftarrow S \cup \{P\}$
- $A \leftarrow A \setminus \text{arcs}(P)$

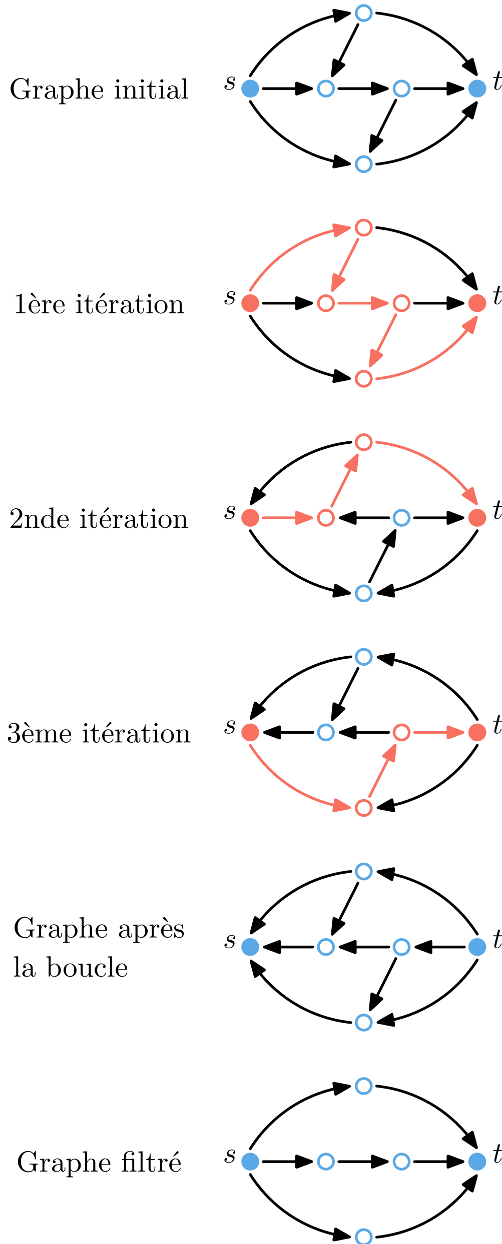


Figure 9 – Exécution de l'algorithme de Ford–Fulkerson

Justifions la validité de l'algorithme.

Commençons par la notion de coupe. Une $(s-t)$-coupe est une partition (S, T) de V telle que $s \in S$ et $t \in T$. La *valeur* d'une coupe est le nombre d'arcs de S à T. Remarquons que la valeur d'une $(s-t)$-coupe est un majorant du nombre de chemins disjoints.

Une itération de la première boucle « **Tant que** » fait décroître la valeur de *toutes* les coupe de 1. Le nombre d'itérations (dans la première boucle), qu'on notera k, est la valeur minimale d'une coupe. En effet, une fois que l'on atteint une valeur de 0 pour toutes les coupes, il n'y a donc plus d'arcs de S à T.

Pour reconstruire les partitions S et T, il suffit de trouver les sommets reliés à s.

Après l'exécution de la ligne $(\star)$, le degré sortant (noté $d^+(s) = k$) de s est k et le degré entrant (noté $d^-(s) = k$) de t est est k. On a la propriété que tous les sommets $v \notin \{s, t\}$ ont le même degré entrant et sortant :

$$\forall v \notin \{s, t\}, d^+(v) = d^-(v),$$

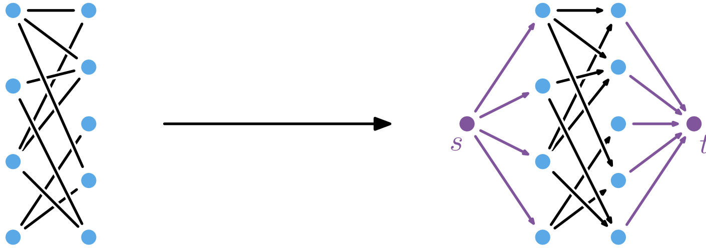
on appelle cela la condition de Kirchhoff.

Une marche gloutonne partant de s arrive forcément en t. On en déduit que le calcul d'un chemin est glouton.

Après une itération de la seconde boucle « **Tant que** », la condition de Kirchhoff est conservée, et le degré sortant de s décroît de 1 exactement. On a donc k chemins dans A.

Exemple

On peut appliquer le problème de flot aux calcul d'un couplage maximal dans le cas d'un graphe biparti. Pour cela, il suffit d'ajouter deux sommets s et t.



Problème de couplage Problème de flots

Ainsi, le calcul du nombre maximal de $(s - t)$-chemins disjoints correspond exactement au calcul du couplage maximal.

Théorème

Quel que soit $\varepsilon > 0$, il existe un algorithme de calcul du flot maximum en $\mathrm{O}((n+m)^{1+\varepsilon})$ (complexité quasi-linéaire).

IV.2. Codage binaire.

On a un mot $M \in \mathscr{A}^\star$ et on veut le transformer en mot binaire en replaçant chaque lettre par un mot de $\{\mathtt{0}, \mathtt{1}\}$. Idéalement, on veut pouvoir retrouver M et compresser le mot binaire.

On veut un code préfixe optimal, c'est-à-dire, pour chaque lettre x dans M, ayant m_x occurrences, on veut associer un mot $\varphi(x) \in \{\mathtt{0}, \mathtt{1}\}^\star$ tel que

- *préfixe* : pour deux lettres $x \neq y$, $\varphi(x)$ n'est pas préfixe de $\varphi(y)$;
- *optimal* : $\varphi(M)$ est de longueur minimal, *i.e.*, $\sum_{x \in \mathscr{A}} m_x \ |\varphi(x)|$ est minimal parmi les codes préfixes.

Exemple

On considère le mot $M \in \{\mathtt{a}, \ldots, \mathtt{g}\}^\star$ avec les occurrences :

Lettre	$\mathtt{a}$	$\mathtt{b}$	$\mathtt{c}$	$\mathtt{d}$	$\mathtt{e}$	$\mathtt{f}$	$\mathtt{g}$
Nb. occ.	28	7	10	6	1	4	8

Une solution possible est le code ASCII. Le mot M a une longueur totale de $7 \cdot (28 + 7 + 10 + \cdots + 8) = 7 \cdot 64$.

Une autre idée est d'utiliser des codes binaires de longueur 3. Ceci donne une longueur de l'encodage de M valant $3 \cdot 64$.

Pourrait-on utiliser un code plus court pour $\varphi(x)$? Oui, c'est possible. Mais pour cela, il faut changer de perspective.

Qu'est ce qu'un code préfixe ? Le codage binaire (l'ensemble des $\varphi(x)$ pour $x \in \mathscr{A}$) correspond a des feuilles dans un arbre binaire.

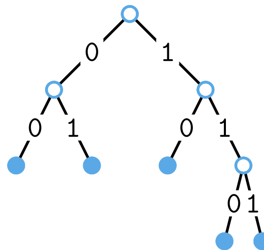


Figure 11 – Représentation en arbre d'un code préfixe binaire

Ainsi, l'image de φ est $\{\mathtt{111}, \mathtt{110}, \mathtt{01}, \mathtt{00}, \mathtt{10}\}$, c'est l'ensemble des chemins partant de la racine jusqu'aux feuilles. Comme le code cherché est optimal, il n'y a pas de nœud interne de degré 1.

Le problème CodagePréfixeBinaireOptimal, noté CPBO, est défini comme

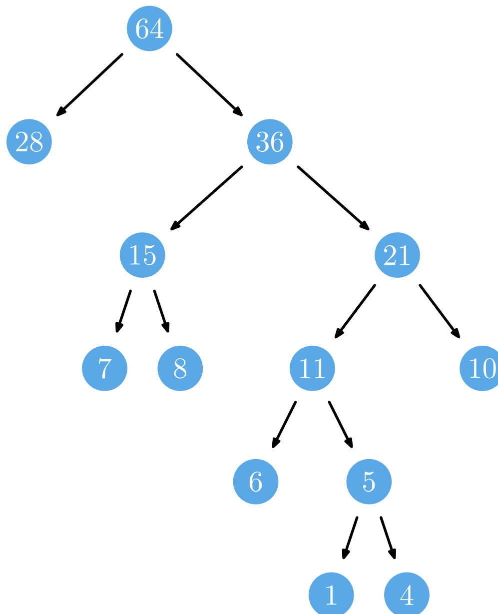
CPBO : **Entrée.** L'ensemble $E = \{m_x \mid x \in \mathscr{A}\}$ d'entiers
Sortie. L'arbre A_φ où φ est le codage optimal

Pour résoudre ce problème, on utilise l'algorithme de Huffman. C'est un algorithme glouton.

Huffman

Tant que card $E \geqslant 2$ **faire**
  Calculer m_x et m_y les deux plus petites valeurs dans E.
  $E \leftarrow (E \setminus \{m_x, m_y\}) \cup \{m_x + m_y\}$
Retourner l'arbre A_φ.[5]

On code l'algorithme de Huffman avec un *tas*. En effet, sans tas, on a une complexité en $\mathrm{O}(n^2)$ alors qu'avec un tas, on a une complexité en $\mathrm{O}(n \log n)$.



[5] Avec l'exemple, ça devrait être plus clair sur ce qu'il se passe ici.

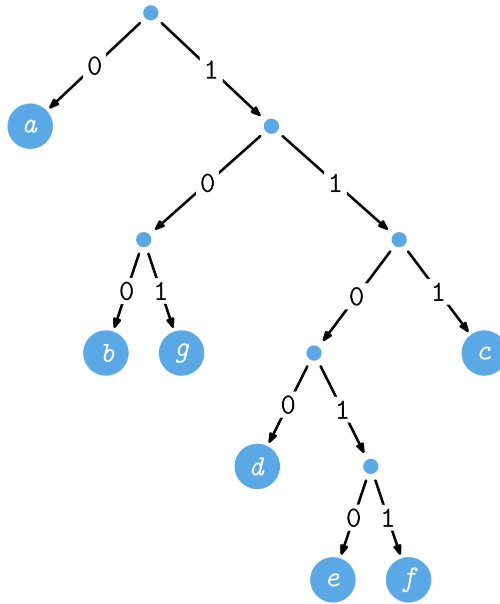


Figure 13 – Arbre de Huffman pour l'exemple précédent

Pour justifier la validité, on utilise un argument classique. On construit une solution avec l'algorithme A_{HUF} et une solution optimale A_{OPT}. On va rapprocher A_{OPT} de A_{HUF}.

On procède par induction sur la taille de l'arbre.

- Dans le cas de base, il n'y a qu'un nœud. L'arbre optimal est donc l'arbre de Huffman.
- Dans le cas général, supposons inverser deux éléments x et z (en rouge dans la figure ci-dessous). Alors, l'arbre n'est plus optimal. Or, par construction, l'arbre de Huffman vérifie cette propriété.

  De plus, en remplaçant le nœud u par une feuille de symbole « $x + y$ » (en un seul caractère de l'alphabet), alors on obtient un arbre de taille plus petite. On peut donc appliquer l'hypothèse de récurrence pour en déduire que l'arbre optimal peut être construit à l'aide de l'algorithme de Huffman.

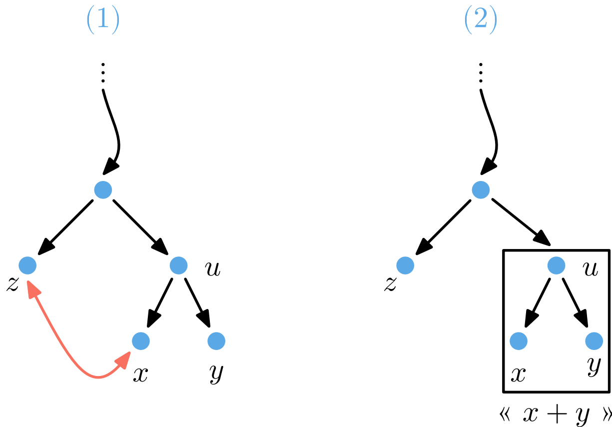


Figure 14 – Démonstration de l'optimalité de l'arbre de Huffman
(1) Inversion de deux éléments dans l'arbre optimal ;
(2) Fusion de deux éléments dans l'arbre optimal.

Quelles sont les limites de la compression ? Une borne inférieure pour la longueur d'un codage binaire d'un mot M de longueur ℓ avec pour occurrences $(m_x)_{x \in \mathscr{A}}$ est

$$\log_2 \underbrace{\left(\frac{\ell!}{\prod_{x \in \mathscr{A}} m_x!} \right)}_{\substack{\text{Nombre d'anagrammes} \\ \text{de } M}}$$

Et asymptotiquement ? On définit les fréquences des lettres

$$f = \left(f_x := \frac{m_x}{\ell} \;\middle|\; x \in \mathscr{A} \right).$$

Le nombre d'anagrammes de M est donc $\sim 2^{\mathrm{H}(f)\ell}$ (à un terme polynômial près), où $\mathrm{H}(f) = \sum_{x \in \mathscr{A}} -f_x \log_2 f_x$ est le « coefficient d'allongement » de M (vu comme une borne inférieure) : pour passer d'un mot à son encodé, la longueur est multiplié par $\mathrm{H}(f)$. On nomme $\mathrm{H}(f)$ l'*entropie de Shannon*.

Avec les fréquences $f = (\frac{1}{2}, \frac{1}{2})$, on a $\mathrm{H}(f) = 1$. « La masse des mots binaires est concentré sur équiréparti. »

Avec un mot binaire de fréquences $f = (0.49, 0.51)$, on a $\mathrm{H}(f) < 1$. En effet, le nombre de mots binaires avec plus de 0.49 fois « `0` » est équivalent à $2^{\mathrm{H}(f)\ell}$. Ainsi, si on tire un mot binaire aléatoire, la probabilité que le nombre de `0` est inférieur à 0.49ℓ est équivalente à

$$\frac{2^{\mathrm{H}(f)\ell}}{2^\ell} = \frac{1}{2^{(1-\mathrm{H}(f))\ell}},$$

on retrouve le théorème de Chernoff.

IV.3. Arbre couvrant de poids minimum.

IV.3.a. Arbre couvrant.

Définition

Soit $G = (V, E)$ un graphe connexe. On appelle *arbre couvrant* un ensemble $A \subseteq E$ tel que A est connexe et acyclique.

Proposition

Tout graphe connexe possède un arbre couvrant.

Preuve. On se propose plusieurs preuves.

(1) *Par induction.* On considère un $(u - v)$-chemin P glouton. Par construction, tous les voisins de v sont dans P. Ainsi, $G \setminus \{v\}$ est un graphe connexe. En effet, si $G \setminus \{v\}$ n'est pas connexe, on a donc une partition en composantes connexes, et alors P est totalement inclus dans une composante connexe. Mais, G ne serait donc pas connexe, ce qui est absurde.

    Ainsi, par hypothèse d'indiction, il existe un arbre A' couvrant $G \setminus \{v\}$. On pose $A = A' \cup \{vw\}$, où w est un voisin de v. L'ensemble A est un arbre couvrant de G.

(2) On considère un ensemble A connexe minimal.

(3) On considère un ensemble A acyclique maximal.

□

Remarquons que le nombre d'arêtes de A, un arbre couvrant, est $n - 1$ où $n = |V|$.

Comment *calculer efficacement* un arbre couvrant ?

- On peut utiliser un arbre de parcours, ceci aurait une complexité en $\mathrm{O}(n+m)$.
- Dans le cas d'un algorithme *online*, où les arêtes sont reçues une à une, et la décision doit se faire à chaque arête, on doit utiliser une structure adaptée.

Comment décider si une arête xy reçue forme un cycle ou pas ? Comment mettre à jour les composantes déjà obtenues ?

Pour cela, on utilise une structure *union-find* (*unir & trouver*) :
- $C(x)$ désigne le représentant de la composante de x ;
- $\mathrm{comp}(x)$ désigne l'ensemble des sommets de la composante de x, lorsque x est le représentant.

Arbre couvrant

```
A ← ∅

Pour tout v ∈ V faire
    C(v) ← v
    comp(v) ← v)

Pour tout xy ∈ E faire
    Si c(x) ≠ c(y) alors
        A ← A ∪ {xy}
        comp(C(x)) ← comp(C(x)) ∪ comp(C(y))
        C ← comp(C(y))
        Pour tout z ∈ C faire
            C(z) ← C(x)

Retourner A
```

La validité de l'algorithme est assurée par la preuve (3) précédente.

La complexité de cet algorithme est, en pire cas, $\mathrm{O}(m+n^2)$. Ce pire cas arrive, par exemple, sur le graphe :
- avec pour sommets $\{1,2,3,4,5,6\}$,

- et pour arêtes $\{21, 32, 43, 54, 65\}$.

Ceci renomme $\binom{n}{2} = \mathrm{O}(n^2)$ sommets.

Est-ce que le cas moyen est en $\mathrm{O}(m + n^2)$? Qu'en est-il si les arêtes de E sont tirées aléatoirement ?

L'intuition est que, pour une étape intermédiaire, les composantes sont globalement de même taille. Mais, on remarque que, dans la pratique, il y a une « composante géante ». On essaie d'optimiser l'algorithme pour éviter de recopier toute la composante géante dans une plus petite composante.

Arbre couvrant, plus optimisé

$A \leftarrow \emptyset$

Pour tout $v \in V$ **faire**
- $C(v) \leftarrow v$
- $\mathrm{comp}(v) \leftarrow v)$

Pour tout $xy \in E$ **faire**
- **Si** $c(x) \neq c(y)$ **alors**
  - **Si** $|\mathrm{comp}(C(x))| < |\mathrm{comp}(C(y))|$ **alors**
    - Échanger x et y.
  - $A \leftarrow A \cup \{xy\}$
  - $\mathrm{comp}(C(x)) \leftarrow \mathrm{comp}(C(x)) \cup \mathrm{comp}(C(y))$
  - $C \leftarrow \mathrm{comp}(C(y))$
  - **Pour tout** $z \in C$ **faire**
    - $C(z) \leftarrow C(x)$

Retourner A

La complexité devient $\mathrm{O}(m + n \log n)$! Ceci vient du fait qu'un sommet z dans la boucle « **Pour tout** » intérieure n'est renommé

qu'au plus $\log_2 n$ fois. Ainsi, la taille de sa composante double à chaque fois.

Remarque

- La gestion des composantes peut se faire en $O(n \log^\star n)$ et même encore mieux (△ *structure Union-Find*).
- Codez l'algorithme ! Et observez l'apparition de la composante géante (△ *Erdős–Rényi*). Commenter et décommenter la condition sur la taille des composantes (passage de n^2 à $n \log n$).

IV.3.b. Arbre couvrant de poids minimal.

On considère le problème Arbre Couvrant de Poids Minimal (abrégé en ACPM) :

ACPM :
Entrée. Un graphe $G = (V, E)$ et une fonction de pondération $w : E \to \mathbb{N}$
Sortie. Un arbre couvrant A avec $w(A)$ minimal.

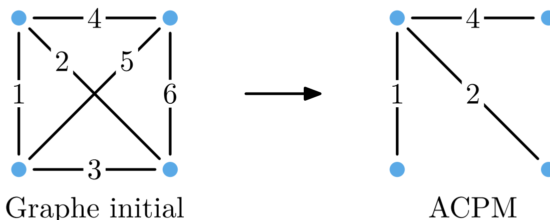


Figure 15 – Un exemple d'arbre couvrant de poids minimal

On résout ce problème à l'aide de l'algorithme de Kruskal.

Algorithme de Kruskal

On trie les arêtes de E par ordre croissant de pondération.
On calcule un arbre couvrant avec l'algorithme précédent.

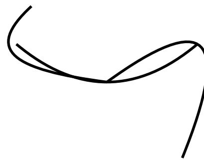
Justifions de la validité de l'algorithme. Soient A et A' deux arbres couvrants de G.

Proposition

Quel que soit l'arête $e \in A \setminus A'$, il existe $e' \in A' \setminus A$ tel que l'ensemble $(A \setminus \{e\}) \cup \{e'\}$ soit un arbre couvrant. C'est un échange de e par e'.

Preuve. On ajoute l'arête e à A'. Ceci, comme A' était un arbre couvrant, implique qu'on crée un cycle $P \cup \{e\}$, avec $P \subseteq A'$.

(La figure sera terminée plus tard . . .)



Soient V_1 et V_2 les deux composantes connexes de $A \setminus \{e\}$. Par le cycle P, on sait qu'il existe une arête e' de P qui va d'un sommet dans V_1 à un sommet dans V_2.

Ainsi, $A_1 = (A \setminus \{e\}) \cup \{e'\}$ est un arbre, et il est couvrant par construction. □

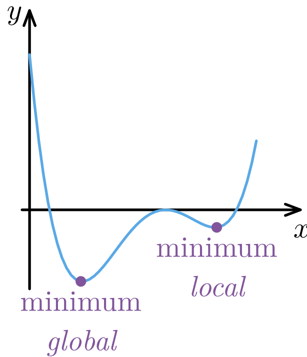
Ainsi, $A_1 = (A \setminus \{e\}) \cup \{e'\}$ et $A_1' = (A \setminus \{e'\}) \cup \{e\}$ sont deux arbres couvrants. Une façon de le voir, c'est que A_1 et A_1' sont plus proches que A et A' ne l'étaient.

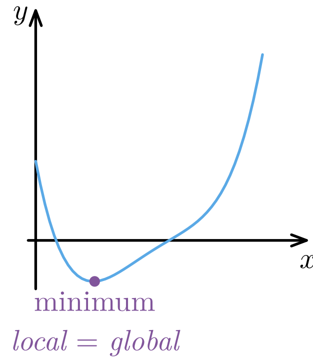
En effet, on peut poser $\operatorname{dist}(A, A') = \frac{A \triangle A'}{2} = d$ et remarquer que l'on a $\operatorname{dist}(A_1, A') = d - 1$.

Ce que l'on construit est une *géodésique* dans un certain espace.

Quel est l'espace ambiant ? C'est l'ensemble des arbres couvrants. Deux arbres sont voisins s'ils diffèrent de deux arêtes. On veut minimiser w dans cet espace.

Dans le cas continu, la globalité d'un minimum local est assurée par la *convexité* de la fonction considérée.





Fonction non convexe — Fonction convexe

Figure 17 – Globalité (ou non) d'un minimum local

Proposition

Si l'arbre A_{LOC} est un minimum local[6] alors c'est un minimum global.

Preuve. Considérons A_{GLO} le minimum sur tous les arbres de w. Si $A_{\text{GLO}} \neq A_{\text{LOC}}$, alors il existe $e \in A_{\text{LOC}}$ et $e' \in A_{\text{GLO}}$ tels que

$$A_1 = (A_{\text{LOC}} \setminus \{e\}) \cup \{e'\} \qquad \text{et} \qquad A_2 = (A_{\text{GLO}} \setminus \{e'\}) \cup \{e\}$$

soient des arbres.

Par propriétés de minimum local, $w(A_{\text{LOC}}) \leqslant w(A_1)$, d'où $w(e') \geqslant w(e)$. D'où, $w(A_2) \leqslant w(A_{\text{GLO}})$, on en déduit que A_2 est aussi un minimum global. Mais, $d(A_{\text{GLO}}, A_{\text{LOC}})$ a diminuée. De proche en proche, on obtient que $w(A_{\text{LOC}}) = w(A_{\text{GLO}})$. □

L'idée est la même que pour l'algorithme de Huffman : pour démontrer l'optimalité de la solution, on *rapproche* la solution optimale au résultat de l'algorithme.

Proposition

L'algorithme de Kruskal retourne un minimum local.

[6] Quel que soit A' voisin de A ($d(A, A') = 1$), on a $w(A') \geqslant w(A)$.

Preuve. Supposons que A_{KRU} soit retourné et que $A' = (A_{\text{KRU}} \setminus \{e\}) \cup \{e'\}$ soit un arbre voisin de A_{KRU}. C'est qu'au moment du choix possible de e', l'arête a été rejetée.

Ses extrémités étaient à cette étape dans la même composante connexe. Le chemin P était déjà dans A, par le choix glouton (et le pré-traitement du tri des arêtes), on a $w(e) \leqslant w(e')$.

Ceci permet d'en déduire que $w(A_{\text{KRU}}) \leqslant w(A')$. □

On en déduit de la validité de l'algorithme de Kruskal.

IV.4. Algorithme GLOUTON.

Soit $H = (V, S)$ un *hypergraphe*, *i.e.* $S \subseteq 2^V$ où S peut être vu comme l'ensemble des solutions admissibles d'un problème. On suppose S donné sous forme d'oracle : qui peut répondre en $O(1)$ à une question « $X \in S$? » lorsque $X \subseteq V$.

Maximisation :
Entrée. $H = (V, S)$ un oracle et $w : V \to \mathbb{N}$
Sortie. Un sous ensemble $X \in S$ tel que $w(X)$ est maximal.

Cette définition est très (trop) générale !

L'algorithme GLOUTON

$A \leftarrow \emptyset$

Trier V par ordre décroissant de w.

Pour tout $v \in V$ **faire**
  Si $A \cup \{v\} \in S$ **alors**
    $A \leftarrow A \cup \{v\}$

Retourner A

- Pour l'algorithme de Kruskal, si V est l'ensemble des arêtes d'un graphe et $S \subseteq 2^V$ est l'ensemble des parties acycliques, alors l'algorithme GLOUTON retourne bien un arbre de poids maximal.

Exemple (Un autre exemple)

Considérons une matrice $\boldsymbol{M}$ de taille $n \times m$ sur un corps $\mathbb{K}$. On pose V l'ensemble des vecterus colonnes, et

$$S = \{X \subseteq V \mid X \text{ est une famille libre}\}.$$

Alors, quelle que soit la pondération $w : V \to \mathbb{N}$, l'algorithme GLOUTON retourne une famille libre maximisant w.

On pourra démontrer cela en imitant la preuve de validité de l'algorithme Kruskal.

Dans l'exemple, l'algorithme Kruskal est un cas particulier : du graphe $G = (V, E)$ avec $n = |V|$ et $m = |E|$, on associe une matrice d'incidence $\boldsymbol{I}_G = \left(i_{v,e}\right)_{v \in V, e \in E}$ où

$$i_{v,e} = \begin{cases} 0 & \text{si } v \notin e \\ 1 & \text{si } v \in e. \end{cases}$$

Par exemple, pour le graphe ci-dessus on associe la matrice

$$\begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \end{pmatrix},$$

appelée *matrice d'incidence.*

Alors, on a l'équivalence ci-dessous.

Proposition

On a $S \subseteq E$ est acyclique si, et seulement si, S (vu comme un ensemble de colonnes) est libre dans $\mathbb{F}_2$.

Preuve.

- « $\Longleftarrow$ ». Par contraposée, supposons S non acyclique. Il existe donc un cycle
$$x_1x_2, x_2x_3, \dots, x_kx_1.$$
Mais, ceci implique que la somme des colonnes $x_1x_2, \ \dots, x_k, x_1$ vaut 0. D'où, la famille S n'est pas libre.
- « $\Longrightarrow$ ». Par induction sur le cardinal de S. Si S est acyclique c'est donc une forêt. Alors il existe un arbre A (le cas S vide est trivial) et donc une feuille x dans l'arbre acyclique A. On applique l'hypothèse d'induction sur $A \setminus \{x\}$.

□

En fait, on peut même généraliser à un problème qui n'est pas ACPM.

Proposition

L'algorithme GLOUTON retourne toujours une base de poids minimal lorsque w est une fonction de poids sur les colonnes d'une matrice.

Preuve. Il suffit de démontrer que, pour toutes bases $\mathscr{B} \neq \mathscr{B}'$, et pour tout $x \in \mathscr{B} \setminus \mathscr{B}'$, il existe $x' \in \mathscr{B}' \setminus \mathscr{B}$ tel que $(\mathscr{B} \cup \{x'\}) \setminus \{x\}$ et $(\mathscr{B}' \cup \{x\}) \setminus \{x'\}$ soient deux bases. □

Remarque

On peut réaliser une même construction sur $\mathbb{R}$, **mais** on représente chaque arête par une arête orientée.

Remarque (Digression sur la simple connexité)

Le graphe G est connexe si, et seulement si, $\operatorname{rg}(\boldsymbol{I}_G) = n - 1$.

IV.4.a. Matroïdes.

Définition

On dit que $H = (V, S)$ est un matroïde si :
(1) $S \neq \emptyset$; « non-vacuité »
(2) $\forall X \in S, \forall Y \subseteq X, Y \in S$; « clôture »
(3) $\forall X, Y \in S, |Y| > |X| \implies \exists y \in Y \setminus X, X \cup \{y\} \in S$. « échange »

Par exemple, le point (3) est vérifié par les familles libres.

Théorème

L'algorithme GLOUTON sur H retourne une solution optimal (quel que soit w) si, et seulement si, H est un matroïde.

Ce théorème est génial. Pourquoi ? Il lie la « structure » d'un ensemble (algébrique) à un « modèle de calcul » (combinatoire).

Un résultat qui a ce même lien est l'équivalence entre automates finis (modèle de calcul) et expressions régulières (structure).

Exemple

Voici quelques exemples de matroïdes :
(1) le *matroïde graphique* : les acycliques dans un graphe ;
(2) le *matroïde vectoriel* : les familles livres dans une matrice ;
(3) le *matroïde uniforme* : pour k fixé,

$$S = \{X \subseteq V \mid |X| \leqslant k\}.$$

(4) le *matroïde de partition* :

$$V = V_1 \uplus V_2 \uplus \cdots \uplus V_\ell$$
$$S = \{X \subseteq V \mid |X \cap V_i| \leqslant t_i, \forall i \in [\![1, \ell]\!]\}.$$

Remarque

- Certains matroïdes sont *représentables* comme matroïdes vectoriels mais il en existe des *non-représentables*.
- Une étape de plus, l'intersection : le problème

> **Entrée.** Deux matroïdes S_1 et S_2 sur V et un poids $w : V \to \mathbb{N}$
> **Sortie.** Un sous-ensemble X tel que $X \in S_1 \cap S_2$ et $w(X)$ maximal

est résolvable en temps polynomial.

Exemple

On considère un graphe biparti. Le problème du couplage maximal (aussi appelé Couplage Parfait Biparti) peut se résoudre en temps polynomial. En effet, $X \subseteq E$ s'exprime sous forme d'intersection de deux matroïdes de partitions sur A et B.

Et si on ajoute une autre intersection ? Le problème est **NP**-dur.

Exemple

On se donne un graphe biparti G' comme construit dans la figure ci-dessus. On se donne plusieurs contraintes :

(1) le degré est inférieur à 2 à gauche ;
(2) le degré est inférieur à 2 à droite ;
(3) l'ensemble des arêtes est acyclique.

Le couplage maximal du graphe G' est un chemin de longueur maximal qui passe par tous les points. On se réduit donc au problème *Travelling Sales Person* (TSP).

V. NP-complétude.

V.1. La classe NP, définition intuitive.

Un problème est dans **NP** s'il existe un algorithme polyomial $\mathscr{A}$ et une constante d telle que X est vrai *ssi* il existe un certificat C de taille au plus $|X|^d$ tel que $\mathscr{A}(X, C)$ est vrai.

En pratique : on sait facilement vérifier qu'une instance est vrai si on peut en fournir la preuve (en taille polynomial). La preuve, c'est la donnée de la « solution ».

C'est en général très facile.

Par exemple, le problème

SOMME :
Entrée. S et $n_1, \ldots, n_\ell$.
Sortie. VRAI s'il existe $I \subseteq [\ell]$ telle que $\sum_{i \in I} n_i = S$.

est dans **NP** car il suffit de donner I comme certificat (ici l'algorithme A est juste la somme et le teste d'égalité à S).

Le problème

CPB :
Entrée. Un graphe $G = (V, E)$ biparti
Sortie. VRAI s'il existe un couplage parfait

est dans **NP**. En effet, il suffit de donner le couplage comme certificat.

Le problème

PREMIER :
Entrée. Un entier n
Sortie. VRAI si n est premier.

est dans **NP**, mais c'est bien moins évident. Pour ce problème, trouver un certificat de taille polynomial est bien plus complexe

(la représentation de n est en $\log_2(n)$ donc un $O(n)$ est en réalité *exponentiel* en la taille de n).

Pour trouver un certificat, on commence par analyser le *théorème de Lucas* : un entier n est premier si, et seulement s'il existe $a \in \mathbb{N}$ tel que

$$a^{n-1} \equiv 1 \pmod n \quad \text{et} \quad \forall q \text{ premier divisant } n-1, a^{(n-1)/q} \not\equiv 1 \pmod n$$

(Justification plus tard . . .)

Remarque

La classe **NP** est biaisée vers Vrai. Mais, on aurait pu la biaiser vers Faux et on obtiendrai la classe **co-NP**.

Par exemple, le problème

SAT :
Entrée. Une formule φ en forme normale conjonctive de variables $x_1, \ldots, x_n$.
Sortie. Vrai si on peut affecter les x_i afin de rentre φ vraie (***satisfaire*** φ)

est clairement dans **NP** car il suffit de donner la solution. **Mais mais mais**, il semble difficile d'obtenir un certificat de Faux de taille polynomiale (et même de taille $2^{o(n)}$).

V.2. *Les deux (vraies) définitions de* NP.

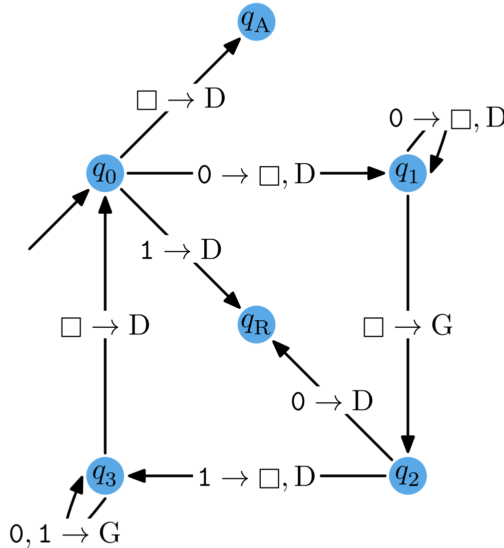
On commence par les automates finis. Ils sont puissants, mais restent assez limités : $\{0^n 1^n \mid n \in \mathbb{N}\}$ n'est pas rationnel. Turing propose de donner un ruban mémoire à l'automate où il pourra lire, écrire et déplacer une tête de lecture.

Au début du calcul, un mot $M \in \{0,1\}^\star$ est inscrit sur le ruban. Le reste du ruban est rempli de caractères vides « $\square$ ». Et, au cours des changements d'états, si l'automate arrive dans q_A un état

acceptant, alors M est accepté ; s'il arrive dans q_R (état rejetant) alors M est rejeté.

Exemple

Pour le langage $\{0^n 1^n \mid n \in \mathbb{N}\}$, on construit la machine de Turing $\mathscr{M}$:



Ainsi, $\mathscr{M}$ décide L, qu'on note $\mathscr{L}(\mathscr{M}) = L$.

Définition

On dit que $\mathscr{M}$ *décide* L si :
- tous les calculs sur toutes les entrées terminent ;
- M *est accepté* par $\mathscr{M}$ si et seulement si $M \in L$.

Si tous les calculs terminent en temps polynomial alors $\mathscr{M}$ est une machine de Turing polynomiale. Un langage L est dans la classe **P** s'il existe une machine de Turing polynomiale qui décide L.

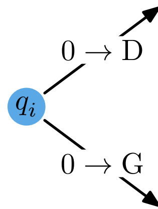
Dans les définitions suivantes, on simplifie avec $\mathscr{A} = \{\mathtt{0}, \mathtt{1}\}$ comme alphabet.

Définition (version 1)

On a $L \in \mathbf{NP}$ s'il existe un polynôme p et un langage $L' \in \mathbf{P}$ tels que :

$$L = \left\{ M \in \mathscr{A}^\star \;\middle|\; \exists c \in \mathscr{A}^\star, \begin{bmatrix} |c| \leqslant p(|M|) \\ M \cdot C \in L' \end{bmatrix} \right\}.$$

On ajoute le non déterministe aux machines de Turing : on autorise d'avoir deux transitions comme dans la figure ci-dessous.



On exige toujours que tous les calculs terminent. Un mot M est *accepté* par une machine de Turing non déterministe s'il existe un chemin vers q_{A}.

De plus, si les chemins terminent en un nombre polynomial d'étapes en la taille de l'entrée, on parle de machine de Turing non déterministe polynomiale.

Définition (version 2)

On a $L \in \mathbf{NP}$ s'il existe une machine de Turing non déterministe polynomiale qui décide L.

Théorème

Les deux définitions sont équivalentes.

Preuve (Esquisse de preuve . . .).

- On utilise le non déterminisme pour générer un certificat et on teste ce certificat sur la machine de vérification.
- Le certificat correspond à une suite de choix qui mène à q_{A}.

□

Remarque

On remarque que $L \in \textbf{co-NP}$ si et seulement si $\mathscr{A}^\star \setminus L \in \textbf{NP}$. Une classe intéressante est $\textbf{co-NP} \cap \textbf{NP}$.[7]

V.3. Réductions.

Définition

Un langage L est dit polynomialement réductible à un langage L' s'il existe une fonction f calculable en temps polynomial de $\mathscr{A}^\star \to \mathscr{A}^\star$, tel que

$$M \in L \text{ si, et seulement si } f(M) \in L'.$$

C'est aussi appelé une *many-to-one reduction* ou une *Karp reduction*.

Intuitivement, un problème P se réduit à un problème Q s'il existe un algorithme polynomial qui transforme les instances de P en instances de Q en respectant Vrai/Faux.

[7] Aller voir la photo de Jack Edmonds sur Wikipedia.

Remarque

Si $L' \in \mathbf{P}$ alors $L \in \mathbf{P}$.
Si $L' \in \mathbf{NP}$ alors $L \in \mathbf{NP}$.

Intuitivement, Q est *au moins aussi difficile* que P.

On note $L \leqslant L'$ et cela correspond à un ordre partiel de difficultés des langages (au détail de l'anti-symétrie, par exemple tous les problèmes de $\mathbf{P}$ sont équivalents).[8]

On peut aussi définir les réductions Turing, où l'on peut utiliser plus d'une fois un oracle sur L. Maison ne verra que des réductions Karp dans ce cours.

Théorème (Cook-Levin)

Il existe $L' \in \mathbf{NP}$ tel que, pour tout $L \in \mathbf{NP}$, on a $L \leqslant L'$.

En particulier, Cook a montré que SAT convient pour L', un tel langage L' est dit ***NP**-complet*.

L'intérêt est :
- si vous n'arrivez pas à trouver un algorithme polynomial pour un problème P_L alors essayez de montrer qu'il est **NP**-complet ;
- pour ce faire, on cherche à réduire SAT en L en transformant les instantes.

V.4. Quelques réductions classiques.

△ *Liste des 21 problèmes **NP**-complets de Karp.*

[8]La composition est assurée en composant les deux algorithmes polynomiaux.

3SAT : **Entrée.** Une formule φ sous forme normale conjonctive dont toutes les clauses ont 3 littéraux.
Sortie. VRAI si φ est satisfiable.

Proposition

3-SAT est **NP**-complet.

Preuve. On montre que SAT $\leqslant$ 3-SAT. On se donne une formule $F = C_1 \wedge \cdots \wedge C_m$ de SAT en variables $x_1, \ldots, x_n$. On veut transformer F en une formule équivalente de 3-SAT.

On suppose $C_1 = \ell_1 \vee \cdots \vee \ell_t$ et on va exprimer C_1 comme une conjonction de taille inférieure à 3.

- Si $t \leqslant 3$, on ne change pas C_1 ;
- Si $t > 3$, on va créer $t-3$ nouvelles variables $y_1, \ldots, y_{t-3}$ et on remplace la clause C_1 par :

$$C_1' := (\ell_1 \vee \ell_2 \vee y_1) \wedge (\neg y_1 \vee y_2 \vee \ell_3) \wedge (\neg y_2 \vee y_3 \vee \ell_4) \wedge \cdots \wedge (\neg y_{t-4} \vee y_{t-3} \vee \ell_{t-1}) \wedge (\neg y_{t-3} \vee y_{t-1} \vee \ell_t).$$

On applique la même transformation pour tous les C_i (à chaque fois avec des nouvelles variables). On obtient finalement une formule de 3-SAT notée F'.

De plus F est satisfiable si, et seulement si F' est satisfiable. En effet, on ne peut pas satisfaire C_1' avec uniquement les y_i, car on aurait $y_1 =$ VRAI et $y_{t-3} =$ FAUX et donc il existerait j tel que $y_j =$ VRAI mais $y_{j+1} =$ FAUX. Or, pour satisfaire $\ell_{j+2} =$ VRAI. On a donc prouvé la réciproque. Le sens direct se réalise bien plus simplement. □

CLIQUE : **Entrée.** Un graphe $G = (V, E)$ et un entier k
Sortie. VRAI si G a une clique[9] de taille k

Proposition

CLIQUE est **NP**-complet.

Preuve.

[9] *i.e.* possède k sommets reliés deux à deux

(1) On vérifie que CLIQUE est dans **NP** : le certificat, c'est la clique de taille k.
(2) On réduit un problème bien connu (ici, SAT) **NP**-complet à CLIQUE.

Montrons que 3-SAT $\leqslant$ CLIQUE. Soit $\varphi = C_1 \wedge C_2 \wedge \cdots \wedge C_n$ où chaque C_i est composé de trois littéraux x_i ou $\neg x_i$. On forme un graphe G_φ sur $3n$ sommets, un pour chaque littéral dans chaque clause. Les arêtes de G_φ relient les littéraux $\ell_i \ell_j$ de clauses différentes et vérifiant $\ell_i \neq \ell_j$.

Il est équivalent que : G_φ a une clique de taille n et que φ est satisfiable.

- « $\Longrightarrow$ » Si φ est satisfiable, il existe une affectation avec au moins un littéral vrai dans chaque clause, et

□

SOMME : **Entrée.** Des entiers $s_1, \ldots, s_m$ et S
Sortie. VRAI s'il existe $I \subseteq [n]$ tel que $\sum_{i \in I} s_i = S$

La taille du codage est $\sum \lceil \log s_i \rceil + \lceil \log s \rceil$.

Proposition

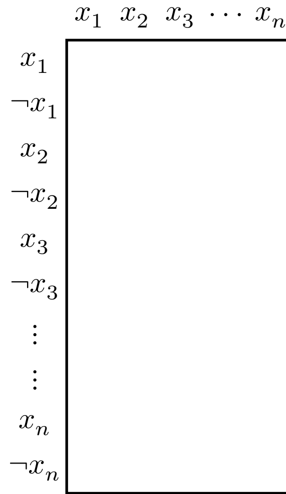
SOMME est **NP**-complet.

Preuve.
(1) On vérifie que SOMME est dans **NP** (on donne la solution).
(2) Montrons que 3-SAT $\leqslant$ SOMME. On va construire une fonction f des instances de 3-SAT vers les instances de SOMME telle que I est VRAI *ssi* $f(I)$ est VRAI. On se donne $F = C_1 \wedge C_2 \wedge \cdots \wedge C_m$ en les variables $x_1, \ldots, x_n$ et on lui associe des entiers. Le « truc », c'est d'utiliser des représentations de nombres en base 10 mais qui n'utilisent que des zéros et des uns. Ainsi, il n'y a pas de retenue et c'est un codage combinatoire.

On forme $2n + 2m$ entiers dont les chiffres sont des zéros et des uns à l'aide de la matrice $\boldsymbol{M}$ (les lignes sont des entiers). On crée **LC** la matrice d'incidence clause/littéraux : $\mathbf{LC}_{\ell,c} = 1$ si le littéral ℓ est da,s la clause c et 0 sinon. On se donne $S = \underbrace{1\ldots1}_{n}\underbrace{3\ldots3}_{m}$.

La formule F est satisfiable si, et seulement si, il existe $I \subseteq [2n + 2m]$ tel que $\sum_{i \in I} s_i = S$. En effet, si F satisfiable, il suffit de sélectionner les littéraux positifs, la somme donne bien S_1 (la somme des littéraux). Et, comme chaque clause est satisfaite par les m premiers chiffres, on a donc une valeur de 1, 2 ou 3, il suffit de compléter pour avoir 3.

□



SAC-À-DOS :

Entrée.
- un ensemble de couples $(p_i, v_i)_{i\in[n]}$ entiers ;
- volume total V ;
- prix à atteindre P.

Sortie. VRAI s'il existe $I \subseteq [n]$ tel que
- $\sum_{i\in I} v_i \leqslant V$;
- $\sum_{p\in I} p_i \geqslant P$.

Remarque

Ce problème est dans **P** si codé en unaire (*i.e.* la taille d'une instance est en $\sum p_i + \sum v_i + V + P$) mais il est **NP**-complet s'il est codé en binaire.

Proposition

SAC-À-DOS (codé en binaire) est **NP**-complet.

Preuve. On montre SOMME $\leqslant$ SAC-À-DOS.

À une instance de SOMME $(s_1, \ldots, s_n, S)$, on construit l'instance de SAC-À-DOS en posant :

- $p_i = s_i$;
- $v_i = v_i$;
- $V = S$;
- $P = S$.

On a immédiatement l'équivalence des instances vraies. □

STABLE : **Entrée.** G un graphe et k un entier
Sortie. Vrai s'il existe k sommets deux-à-deux non reliés

Proposition

Le problème STABLE est **NP**-complet car CLIQUE $\leqslant$ STABLE. En effet, il suffit de remplacer les arêtes par des non-arêtes (et vice versa).

COUVERTURE : **Entrée.** G un graphe et k un entier
Sortie. Vrai s'il existe k sommets qui intersectent toutes les arêtes

Proposition

COUVERTURE (aussi appelé VERTEX-SET) est **NP**-complet.

Preuve. Par réduction de STABLE, noter que G a un stable de taille $n - k$ si et seulement si G a une couverture de taille k (il suffit de prendre $C = V \setminus S$ où S est un stable). □

FEEDBACK-VERTEX-SET : **Entrée.** G un graphe et k un entier
Sortie. Vrai s'il existe un ensemble X d'au plus k sommets tel que $G \setminus X$ est acyclique.

Proposition

FVS est **NP**-complet.

Preuve.

□

On a la chaîne de réduction :

$$\textsc{sat} \leqslant \textsc{3-sat} \leqslant \textsc{clique} \leqslant \textsc{stable} \leqslant \textsc{couverture} \leqslant \textsc{fvs}.$$

COUPLAGE-3D :

Entrée. Un tenseur $\boldsymbol{T} = (t_{i,j,k})$ de taille $n \times n \times n$ de $\{0,1\}$

Sortie. Vrai s'il existe un couplage 3D, *i.e.* la donnée de deux permutations σ et τ du groupe symétrique $\mathfrak{S}_n$ vérifiant que l'on ait $t_{i,\sigma(i),\tau(i)} = 1$ pour $i \in [\![1,n]\!]$.

C'est équivalent à trouver un ensemble d'entrées « 1 » tel que toutes les $3n$ tranches contiennent exactement une de ces entrées.

Une autre façon de voir un couplage est la suivante.
- On considère trois ensembles H, L et C disjoints de taille n.
- On ajoute un triangle (i,j,k) avec $i \in H$ et $j \in L$ et $k \in C$ pour chaque $t_{i,j,k} = 1$.
- Il existe un couplage 3D *ssi* il existe n triangles qui couvrent l'ensemble $V := H \cup L \cup C$.

Proposition

COUPLAGE-3D est **NP**-complet.

Preuve. On montre 3-SAT $\leqslant$ COUPLAGE-3D. On se donne $F = C_1 \wedge C_2 \wedge \cdots \wedge C_m$ une formule de 3-SAT en variables $x_1, \ldots, x_n$. On transforme F en une instance de couverture par triangle (la variante équivalente de COUPLAGE-3D décrite ci-avant).

Chaque variable x_i donne un gadget.

Les $2m$ pointes sont les seuls éléments partagés avec les autres gadgets. Il n'y a que deux possibilités de faire une couverture par triangle, et ceci correspond à choisir x_i ou $\mid x_i$.

Puis, pour chaque clause, comme par exemple $C_i = \ell_1 \vee \ell_2 \vee \ell_3$, on crée deux sommets $a \in C$ et $b \in L$ (la « base » de C_i) et on relie a et b à ℓ_1, ℓ_2, ℓ_3 (à la i-ème pointe des gadgets) puis on relie a et b.

Si F est satisfaite, on peut couvrir par des triangles : toutes les bases de clauses et couvrir toutes les bases de variables pour le littéral satisfait.

Mais, il reste des pointes isolées. Précisément, il en reste $mn - m$ (on a mn libérés par les gadgets variables et m capturés par les clauses).

On ajoute $mn - m$ gadgets « nettoyants » construit par : on crée deux sommets a et b reliés entre eux, et on les relie tous les deux à toutes les pointes.

Ceci permet d'assurer l'équivalence. □

MAX-CUT :

Entrée. Un graphe $G = (V, E)$ et un entier k.
Sortie. VRAI s'il existe une coupe $V = A \cup B$ tel que le nombre d'arêtes entre A et B (noté $e(A, B)$) est au moins k.

Proposition

MAX-CUT est **NP**-complet.

Preuve. On montre NAE-3-SAT[10] $\leqslant$ MAX-CUT. Soit F une formule $C_1 \wedge \cdots \wedge C_m$.

On construit le multigraphe G_F en deux étapes :
- on ajoute tous les littéraux x_i et $\neg x_i$ pour $i \in [\![1, n]\!]$;
- pour la variable x_i, on ajoute N arêtes entre x_i et $\neg x_i$;
- on relie les littéraux d'une même clause.

On a l'équivalence : F est NAE-3-SAT satisfiable *ssi* on peut trouver une coupe de G_F avec au moins $nN - 2m$ arêtes, en supposant que $N > 2m$. (« Il y a plus intérêt à couper les arêtes d'une clause que ceux entre x_i et $\neg x_i$ »). □

CIRC.-HAM.-ORIENTÉ :

Entrée. Un graphe orienté
Sortie. VRAI s'il existe un cycle qui passe par tous les sommets

Proposition

CIRC.-HAM.-ORIENTÉ est **NP**-complet.

[10] Le problème NAE-3-SAT, « *not all equal* » 3-SAT, est le problème qui, à une formule, est vrai s'il existe une valuation qui value différemment les littéraux de chaque clause (valuation pas toute égale dans une clause).

Preuve. C'est une réduction classique depuis 3-SAT. □

V.5. Aparté : NP ∩ co-NP.

V.5.a. Quelques exemples.

Les problèmes de la classe **NP ∩ co-NP** sont les problèmes de décisions admettant un certificat de réponse Vrai **et** de réponse Faux polynomiaux à vérifier. On appelle cela les problèmes **bien caractérisés**.

V.5.a.i. Le problème PREMIER.

- C'est un problème de **co-NP** : le certificat de réponse Faux est un diviseur de n.
- C'est un problème de **NP** (▷ Pratt en '75).
- **Finalement**, c'est un problème de **P** (Agraval, Kayal et Saxena en 2002).

V.5.a.ii. Le problème CouplageParfait.

CouplageParfait :
- **Entrée.** Un graphe G
- **Sortie.** Vrai s'il existe un couplage qui couvre tout les sommets.

Dans le cas d'un graphe biparti, l'algorithme polynomial (algorithme « hongrois ») résout ce problème (∼ '30).

Le problème est simplement dans **NP** : il suffit de donner la solution en certificat. Mais, pour l'appartenance à **co-NP**, c'est plus subtil.

Théorème (Tutte en '46)

Pour tout ensemble X de sommets, le nombre de composantes connexes de taille impaire de $G \setminus X$ (noté $\mathrm{ci}(X)$) vérifie $\mathrm{ci}(X) \leqslant |X|$.

Par ce théorème, X est bien un certificat pour l'appartenance de CouplageParfait à **co-NP**.

Finalement, c'est un problème de **P** (Edmonds en '66).

V.5.a.iii. Le problème SystèmeInéquations.

SystèmeInéquations :

Entrée. m inéquations linéaires de la forme

$$\sum_{j=1}^{n} a_{i,j} x_j \leqslant b_i$$

pour $i = 1..m$.

Sortie. Vrai s'il existe une solution.

Ici, la taille de l'entrée, c'est $\sum_{i,j} \lceil \log a_{i,j} \rceil + \sum_i \lceil \log b_i \rceil$.

C'est un problème de **NP** : s'il existe une solution, elle peut être exprimée par un sous-sytème d'égalité et elles s'expriment (Krammer) comme un rapport de déterminants (qui est polynomial en la taille de l'entier).

C'est aussi un problème de **co-NP** (Furkas en 1902).

Finalement, c'est un problème de **P** à l'aide de l'algorithme de l'ellipsoïde (Khachyan en '75).

V.5.a.iv. Le problème Décomposition Nombre Premiers.

Le problème DécompositionNombrePremiers (abrégé en DNP) dans sa version problème de décision, est le problème ci-dessous.

DNP :

Entrée. Deux entiers n et K.

Sortie. Vrai si n admet un diviseur inférieur ou égal à K.

C'est un problème de **NP**, il suffit de fournir un diviseur inférieur ou égal à K.

C'est un problème de **co-NP**, il suffit de fournir la décomposition en produit de facteurs premiers (Pratt en '75).

À partir de ce problème de décision, on peut construire la décomposition d'un nombre en produit de facteurs premiers.

Point négatif la sécurité informatique est basée sur un problème de **NP** $\cap$ **co-NP** (sans parler de l'algorithmique quantique).

Point positif il y a plein de travail en cryptographie post-quantique.

V.5.b. Dualité.

Les certificats de **co-NP** sont basés sur un problème dit *dual*. L'exemple type est MAXFLOT qui est exactement MINCUT.

V.5.b.i. Le problème SYSTÈMEÉQUATIONS.

SYSTÈMEÉQUATION :
Entrée. Un système $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$.
Sortie. VRAI s'il existe une solution.

Pour le certificat **co-NP**, il suffit de fournir un $\boldsymbol{y}$ tel que $\boldsymbol{A}^\top \boldsymbol{y}$ et $\boldsymbol{b}^\top \boldsymbol{y} \neq 0$. En effet, cela est équivalent à une combinaison linéaire des lignes du système qui donne $0 = 1$.

V.5.b.ii. Le problème SYSTÈMEINÉQUATIONS.

Pour le certificat **co-NP**, on utilise le lemme ci-dessous.

Lemme (Farkas en 1902)

Le système $\boldsymbol{A}\boldsymbol{x} \leqslant \boldsymbol{b}$ n'a pas de solutions si, et seulement si, il existe une combinaison linéaire positive (ou nulle) d'inéquations qui donne $0 \leqslant -1$.

V.5.b.iii. Le problème SYSTÈMEPOLYNOMIAUX.

Sys.Poly :

Entrée. $P_1, \dots, P_m$ des polynômes en variables $x_1, \dots, x_n$ à coefficients entiers.

Sortie. Vrai s'il existe $\boldsymbol{x}^\star \in \mathbb{C}^n$ tel que $P_i(\boldsymbol{x}^\star)$ quel que soit $i = 1..m$.

Un cas particulier de ce problème, c'est SystèmeÉquations car il est équivalent à $\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b} = 0$.

En degré 2, on peut aisément coder 3-sat. Soit $F = C_1 \wedge C_2 \wedge \cdots \wedge C_m$ une formule sous 3-CNF en variables $x_1, \dots, x_n$ et on considère le système suivant :

$$\begin{cases} P_1 & : x_1^2 - x_1 = 0 \\ P_2 & : x_2^2 - x_2 = 0 \\ \vdots & \quad \vdots \\ P_n & : x_n^2 - x_n = 0 \\ Q_1 & \\ \vdots & \\ Q_m & \end{cases}$$

où, par exemple, on code la clause $C_1 = x_1 \vee \neg x_2 \vee x_3$ par le polynôme

$$Q_1 : x_1 + (1 - x_2) + x_3 + \boxed{y_{1,1} + y_{1,2}} = 3,$$

où l'on ajoute, pour chaque clause C_i, deux variables $y_{i,1}$ et $y_{i,2}$ vérifiant $y_{i,1}^2 - y_{i,1} = 0$ et $y_{i,2}^2 - y_{i,2} = 0$ (variables *dummy*).

Théorème (Hilbert's Nullstellensatz en 1893)

Le système $\{P_i(\boldsymbol{x}) = 0 \mid i = 1..m\}$ n'a pas de solution dans $\mathbb{C}$ si, et seulement si, il existe $Q_1, \dots, Q_m \in \mathbb{C}[x_1, \dots, x_n]$ tels que

$$\sum_{j=1}^{m} P_i Q_i = 1$$

(et donc on récupère la contradiction $0 = 1$.)

Ce **n**'est **pas** un certificat **co-NP** car un polynôme Q_i peut avoir une complexité en espace exponentielle.

VI. Programmation dynamique.

L'idée est de définir l'optimalité d'un problème à l'aide d'un nombre polynômial de sous-problèmes.

VI.1. *Plus longue suite commute.*

On considère le problème

PLSC :
- **Entrée.** Un mot A de longueur n et B de longueur m sur l'alphabet $\{\texttt{0}, \texttt{1}\}$
- **Sortie.** Un mot C le plus long possible qui est un sous-mot de A et de B

On rappelle que sous-mot signifie ici sous-suite : `001` est un sous-mot de `0101`.

On s'intéresse aux sous-problèmes. On note $m_{i,j}$ la longueur du plus long sous-mot commun de $A[1..i]$ et $B[1..j]$. On cherche $m_{n,m}$.

On procède « par récurrence ».
- pour initialiser, on a $m_{i,j} = 0$ si i ou j vaut 0 ;
- pour l'hérédité, on a :
  (1) si $A[i] \neq B[j]$ alors $m_{i,j} = \max\big(m_{i-1,j}, m_{i,j-1}\big)$;

(2) si $A[i] = B[j]$ alors $m_{i,j} = 1 + m_{i-1,j-1}$.

On peut coder en *bottom up* (avec deux boucles). Ou, on peut aussi utiliser une technique *top down*, mais il faut éviter les appels multiples sur les sous-problèmes en *mémoïsant* les résultats déjà calculés (par exemple avec un tableau **memo** de taille $n \times m$ initialisé à « -1 »).

PLSC-mémoïsé

Si $i = 0$ *ou* $j = 0$ **alors**
  Retourner 0
Sinon si memo$[i, j] = -1$ **alors**
  Si $A[i] \neq B[j]$ **alors**
    memo$[i, j] = \max\begin{pmatrix}\text{PLSC-mémoïsé}(A,i-1,B,j)\\ \text{PLSC-mémoïsé}(A,i,B,j-1)\end{pmatrix}$
  Sinon
    memo$[i, j] = 1 + \text{PLSC-mémoïsé}(A, i-1, B, j-1)$
Retourner memo$[i, j]$

L'intérêt de la mémoïsation, c'est que ça peut être plus efficace (par exemple $A = B$ est en temps linéaire).

Complexité. Le temps de calcul de chaque sous-problèmes est en $\mathrm{O}(1)$. Il y a $\mathrm{O}(nm)$ sous problèmes. La complexité est donc en $\mathrm{O}(nm)$.

Remarque

- L'espérance de la longueur plus longue suite commune de deux suites aléatoires de longueur n de $\texttt{0,1}$ est dans l'intervalle $[0.76n, 0.82n]$.
- Sous *SETH* (expliqué ci-après), il n'existe pas d'algorithme en $\mathrm{O}(n^{2-\varepsilon})$ quel que soit $\varepsilon > 0$.

  ETH (*Exponential Time Hypothesis*). Il n'existe pas d'algorithme pour 3-SAT en $\mathrm{O}\big(2^{\mathrm{o}(n)}\big)$.

  SETH (*Strong Exponential Time Hypothesis*). Pour tout $\varepsilon > 0$, il existe k un entier tel qu'il n'existe pas d'algorithme pour k-SAT en $\mathrm{O}\big(2^{(1+\varepsilon)\cdot n}\big)$.

VI.2. Produit en chaîne de matrices.

Remarque

Le produit (naïf) de $\boldsymbol{A}$ de taille $a \times b$ avec une matrice $\boldsymbol{B}$ de taille $b \times c$ se fait en abc opérations.

Exemple (L'ordre importe dans la complexité)

Soit $\boldsymbol{A}$ de taille 2×4, $\boldsymbol{B}$ de taille 4×6, et $\boldsymbol{C}$ de taille 6×8. Vaut-il mieux calculer $(\boldsymbol{AB})\boldsymbol{C}$ ou $\boldsymbol{A}(\boldsymbol{BC})$?

Dans le premier cas, on a $2 \times 4 \times 6 + 2 \times 6 \times 8 = 144$ opérations.

Dans le second cas, on a $4 \times 6 \times 8 + 2 \times 4 \times 8 = 256$ opérations.

Il faut donc bien mieux calculer $(\boldsymbol{AB})\boldsymbol{C}$.

PCM :

Entrée. Une suite de matrices $\boldsymbol{M}_1, \ldots, \boldsymbol{M}_n$ valide[11]

Sortie. Le produit effectué avec le moins d'opérations possibles.

On utilise la programmation dynamique avec le sous-problème : calculer le nombre $m_{i,j}$, défini comme le nombre minimal d'opérations pour calculer $\boldsymbol{M}_i \cdot \boldsymbol{M}_{i+1} \cdots \boldsymbol{M}_j$.

Pour l'initialisation, on a $m_{i,i} = 0$ (la matrice $\boldsymbol{M}_i$ est déjà calculée). Pour l'hérédité, on a :

$$m_{i,j} = \min_{i \leqslant k < j} \big(m_{i,k} + m_{k+1,j} + \ell_i \ell_{k+1} \ell_{j+1}\big),$$

où l'on note ℓ_i le nombre de lignes de $\boldsymbol{M}_i$, pour $i = 1..n$ et on pose ℓ_{n+1} le nombre de colonnes de $\boldsymbol{M}_n$.

Complexité. On a $\mathrm{O}(n^2)$ sous-problèmes, et chaque sous problème est en $\mathrm{O}(n)$, d'où l'algorithme est en $\mathrm{O}(n^3)$.

Il existe un algorithme en $\mathrm{O}(n \log n)$ (mais l'article original était « faux »). Ceci est revenu dans la littérature.

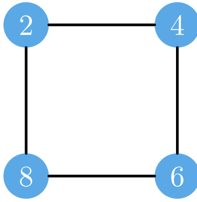
2019. ***Abstract :*** *[. . .] We present an alternative proof for the correctness of the first two algorithms and show that a third algorithm by Nimbark, Gohel, and Doshi (2011) is beyond repair.*

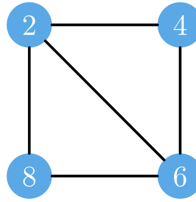
2021. ***Abstract :*** *[. . .] We believe that this exposition is simple enough for classroom use. [. . .]* ▷ Thong Le et Dan Gusfeld

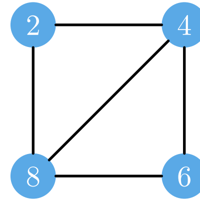
On va étudier la solution du second papier. Les parenthésages sont en bijection avec les triangulations de n-gones.

Avec l'exemple précédent,

[11]où l'on peut réaliser le produit en chaîne (les dimensions des matrices permettent le produit)







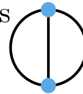
4-gone $\qquad$ $\omega(T) = 144$ $\qquad$ $\omega(T) = 256$

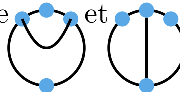
On procède par réduction : comment trianguler un n-gone pondéré en minimisant la somme des triangles (chacun étant le produit de ses éléments).

Une idée qui peut venir est de procéder une induction. On sait déjà qu'il existe toujours au moins 2 éléments isolés dans la triangulation (de degré 2). Ceci est vrai car le *dual* d'une triangulation est un arbre ; s'il a au moins deux éléments, il a au moins deux feuilles.

L'algorithme de calcul d'une triangulation est le suivant :

- si $v_1 v_2$ ou $v_1 v_3$ n'est pas une arête du n-gone

  appels sur les deux sous n-gones 

- retourner le minimum de  et

Annexe A. Génération aléatoire uniforme.

Le but est double :
- tester des hypothèses ;
- générer des *benchmarks*.

A.1. Nombre aléatoire dans $[N] = \{1, \ldots, N\}$.

Pas grand chose de nouveau dans cette partie (voir la littérature), il n'y a qu'une remarque.

Remarque

Éviter d'utiliser `rand() % N` ! En effet, on a un biais vers les petites valeurs.

A.2. Permutation aléatoire uniforme.

Cette génération peut être utiliser en *pre-process* de Quicksort par exemple.

Génération de permutation

Pour i **de** n **à** 2 **faire**
- $j \leftarrow$ un nombre aléatoire dans $[i]$
- Échanger $T[i]$ et $T[j]$

Codez cet algorithme. Puis, évaluez
- la distribution des longueurs des cycles ;
- la probabilité que la permutation est un dérangement (*i.e.* n'a aucun point fixe).

A.3. Matrice à coefficients $0/1$ et graphes aléatoires.

On peut tirer chaque arête d'un graphe avec une probabilité p. On définit le modèle $\mathscr{G}(n,p)$ par : $G \in \mathscr{G}(n,p)$ si G à n sommets avec une probabilité p d'avoir une arête.

Remarque

Tous les graphes aléatoires se ressemblent. En effet, ceci est une conséquence des théorèmes ci-après.

Théorème (*Loi 0-1*)

Pour toute formule F du premier ordre, on a

$$\lim_{n\to\infty} \mathrm{P}(G \vDash F \mid G \in \mathscr{G}(n,p)) = 0 \text{ ou } 1.$$

Ce théorème est faux pour une formule du second ordre.

Exemple

La formule

$$\forall x, y \in V, (xy \in E) \vee (\exists z \in V, xz \in E \wedge yz \in E)$$

est vrai si, et seulement si le diamètre d du graphe vérifie $d \leqslant 2$.

Si on tire chaque parie d'entiers $i, j \in \mathbb{N}$ avec une probabilité $1/2$, on obtient presque sûrement le même graphe dénombrable R *à isomorphisme près*. La limite (dans la loi 0-1) vaut 1 *ssi* $R \vDash F$.

A.4. *Arbre aléatoire uniforme sur $[N]$.*

A.4.a. Première tentative. Voici comment procéder :

- tirons une permutation aléatoire des arêtes possibles (il y en a $\binom{n}{2}$)
- appliquer l'algorithme de Kruskal.

L'espérance du diamètre n'est pas $\sqrt{n}$.

A.4.b. Deuxième tentative.

Combien y a-t-il d'arbres sur $[n]$?

- $n = 1$: il y en a $1 = 1^{-1}$;
- $n = 2$: il y en a $1 = 2^0$;
- $n = 3$: il y en a $3 = 3^1$;
- $n = 4$: il y en a $16 = 4^2$;
- $n = 5$: il y en a $125 = 5^3$;
- ...

Théorème

Il y a n^{n-2} arbres sur $\{1, \ldots, n\}$.

Preuve. On applique le *codage de Prüfer* qui réalise une bijection des arborescences[12] enracinés sur $\{1, \ldots, n\}$ avec les suites de longueur $n - 1$ sur $\{1, \ldots, n\}$. □

Codage de Prüfer

```
C ← ε
Tant que |A| > 1 faire
    Trier les feuilles par ordre croissant
    Pour toute feuille f faire
        C ← C · parent(f)
    Supprimer les feuilles de A
Retourner C
```

Ce codage admet une réciproque.

Exemple (Codage de Prüfer)

[12]Une arborescence est un arbre auquel on indique clairement la racine.

Exemple (Activité pratique)

- Prendre votre numéro de téléphone
- Supprimer le zéro initial

On obtient un mot de 9 lettres sur un alphabet $\{\texttt{0}, \dots, \texttt{9}\}$. C'est le codage de Prüfer d'une arborescence. **Trouver cet arbre.**

Pour tirer un nombre aléatoire sur $[N]$, on procède donc par :
- tirer une suite aléatoire de $[N]^{N-1}$;
- appliquer le décodage de Prüfer ;
- oublier la racine (passage d'une arborescence à un arbre).

Proposition

Si G est un graphe connexe et xy une arête, alors la probabilité

$$\mathrm{P}(\text{Arbre couvrant tiré uniformément contient } xy)$$

est égale à la résistance entre x et y si toutes les arêtes ont une résistance $1\,\Omega$.

Proposition

Le nombre d'arbres couvrants dans un graphe connexe G est égal, au signe près, au déterminant d'un mineur principal de la matrice Laplacienne de G, notée $L(G)$ où

$$L(G) = A(G)D(G)$$

où
- $A(G)$ est la matrice d'adjacence de G ($a_{uv} = 1$ si uv est une arête de G et 0 sinon) ;
- $D(G)$ est la matrice diagonale où d_{vv} est le degré de v.

Exemple

Pour le graphe complet K_n à n éléments, on a :

$$L(K_n) = \begin{pmatrix} -(n-1) & 1 & \cdots & 1 \\ 1 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & 1 \\ 1 & \cdots & 1 & -(n-1) \end{pmatrix}$$

et donc le déterminant est :

$$\det \begin{pmatrix} -(n-1) & 1 & \cdots & 1 \\ 1 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & 1 \\ 1 & \cdots & 1 & -(n-1) \end{pmatrix} = \det \begin{pmatrix} -1 & 0 & \cdots & 0 \\ 0 & -n & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & -n \end{pmatrix} = (-1)^{n-1} \cdot n^{n-2}.$$

On retrouve bien le nombre n^{n-2}.

A.4.c. Cas du couplage.

Comment tirer un couplage uniformément dans un graphe biparti ? C'est dur à résoudre exactement…

En effet, le théorème ci-dessous l'explique.

Théorème (Valient)

Compter le nombre de couplages dans un graphe biparti est un problème #**P**-complet.[13]

Si on savait compter efficacement, alors on saurait tirer uniformément. En effet, il suffit de compter le nombre de de couplages contenant une arête e puis de compter le nombre total de couplages. On peut calculer des probabilités donc tirer uniformément.

[13] Analogue de **NP**-complet pour la complexité de comptage de solution d'un problème.

— Sommaire —

I. Pré-introduction. 1
I.1. Problèmes. 1
I.2. Résoudre efficacement. 4

II. Introduction : la science des arbres. 6

III. Paradigmes : *diviser pour régner*. 8
III.1. Nombre de multiplications. 8
III.1.a. L'exponentiation. 8
III.1.b. Karatsuba & Strassen. 9
III.1.b.i. Algorithme de Karatsuba. 9
III.1.b.ii. Algorithme de Strassen. 11
III.1.b.iii. Multiplication d'entiers en $n \log n$. . 11
III.1.b.iv. Algorithme de Strassen (suite). 11
III.1.c. Plus courts chemins. 13
III.2. Nombre de comparaisons. 14
III.2.a. Calcul du minimum d'un tableau. 14
III.2.b. Algorithme de tri fusion. 15
III.2.c. Calcul de médiane. 17

IV. Algorithmes gloutons. 20
IV.1. Introduction. 21
IV.2. Codage binaire. 25
IV.3. Arbre couvrant de poids minimum. 30
IV.3.a. Arbre couvrant. 30
IV.3.b. Arbre couvrant de poids minimal. 33
IV.4. Algorithme GLOUTON. 36
IV.4.a. Matroïdes. 38

V. NP-complétude. 40
V.1. La classe **NP**, définition intuitive. 41
V.2. Les deux (vraies) définitions de **NP**. 42
V.3. Réductions. 45
V.4. Quelques réductions classiques. 46

V.5. Aparté : **NP** $\cap$ **co-NP**. 53
V.5.a. Quelques exemples. 53
V.5.a.i. Le problème PREMIER. 53
V.5.a.ii. Le problème CouplageParfait. 53
V.5.a.iii. Le problème SystèmeInéquations. 54
V.5.a.iv. Le problème Décomposition Nombre Premiers. 54
V.5.b. Dualité. 55
V.5.b.i. Le problème SystèmeÉquations. 55
V.5.b.ii. Le problème SystèmeInéquations. 55
V.5.b.iii. Le problème SystèmePolynomiaux. 55

VI. Programmation dynamique. 57
VI.1. Plus longue suite commute. 57
VI.2. Produit en chaîne de matrices. 59

Annexe A. Génération aléatoire uniforme. 62
A.1. Nombre aléatoire dans $[N] = \{1, \ldots, N\}$. 62
A.2. Permutation aléatoire uniforme. 62
A.3. Matrice à coefficients 0/1 et graphes aléatoires. 62
A.4. Arbre aléatoire uniforme sur $[N]$. 63
A.4.a. Première tentative. 63
A.4.b. Deuxième tentative. 64
A.4.c. Cas du couplage. 66