# *Algorithmique 1*

*Cours de Stéphan THOMASSE*

*Notes et figures LaTeX par A. Mazoyer*

*L3*

ÉCOLE
NORMALE
SUPÉRIEURE
DE LYON

# Table des matières

# Chapitre -1

# *Introduction*

**Définition -1.1**
L'*algorithmique*, c'est résoudre des problèmes efficacement.
**"borne sup"**

**Définition -1.2**
La *complexité*, c'est montrer que cela n'est pas possible.
**"borne inf"**

## -1.1 Problèmes

Pour spécifier un problème, il nous faut plusieurs données :

- Nom du problème
- Entrée
- Sortie
- Taille de l'entrée (l'entrée est en " " de "blabla" e.g un entier $n$ est en $\log_2 n$)

**Exemple (Voyageur de commerce construction)**

- Entrée : graphe $G = (V, E)$ et longueur $l : E \to \mathbb{N}$
- Sortie : Un cycle qui passe une fois exactement par chaque sommet[1] et minimal pour $l$ s'il existe et **FAUX** sinon

[1] On appellera ça une *tournée*

**Exemple (Voyageur de commerce optimisation)**

- Entrée : graphe $G = (V, E)$ et longueur $l : E \to \mathbb{N}$
- Sortie : La valeur min ou $\infty$ si elle n'existe pas

**Exemple (Voyageur de commerce décision)**

- Entrée : graphe $G = (V, E)$, longueur $l : E \to \mathbb{N}$ et seuil $t$
- Sortie : **VRAI** s'il existe une tournée de longueur $l' \leq t$, **FAUX** sinon

On note $Pb_1 \leq Pb_2$ si "$Pb_1$ se réduit polynomialement à $Pb_2$.
Clairement : $VdC_{dec} \leq VdC_{optim} \leq VdC_{construction}$

**Proposition -1.3**

$VdC_{optim} \leq VdC_{dec}$

**Démonstration**

Recherche dichotomique sur $\left\{0, \cdots, \sum_{e \in E} l(E)\right\}$ en utilisant $VdC_{dec} \longrightarrow$ on trouve OPT en $O\left(\log_2\left(\sum l(E)\right)\right)$ □

**Proposition -1.4**

$VdC_{construction} \leq VdC_{optim}$

**Démonstration**

1. □

**Remarques :**

▷ Les problèmes construction, optimisation, décision sont généralement équivalents.

**Une exception :**

COLLISION SOMME TIROIR

- Entrée : $S$ un ensemble de $n$ entiers $\leq 0$, $x_1, \cdot, x_n$ tel que $\sum_{i=1}^{n} x_i < 2^n$
- Sortie : **VRAI** s'il existe 2 ensembles $X \neq Y \subseteq S \mid \sum X = \sum Y$

Problème de décision : algo en $O(1)$, retourner **VRAI**

**Idée**
Par le principe des tiroirs, deux tels ensembles existent forcément car il existe $2^n$ sous ensembles possibles et moins de $2^n$ sommes possibles

**Définition -1.5 (Problème)**

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

**Définition -1.6 (Langage)**

Un *langage* est une partie de $A^*$ l'ensemble des mots.

Ainsi Pb ↔ langage.

**Un seul type de problème** : étant donné un langage $L$

- Entrée : tout $M \in A^*$
- Sortie : **VRAI** si $M \in L$

## -1.2 Résoudre efficacement

**Modèle de calcul** : automate ⇝ machine de Turing

Couplage parfait biparti

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

Sous-matrice de permutation

**Astuce**
Il est souvent judicieux de réfléchir également aux problèmes équivalents lors de la résolution d'un problème

- Entrée : matrice carrée $M$ de 0 et 1
- Sortie : **VRAI** ou **FAUX** s'il existe $\sigma \in \mathbb{N} \mid \forall i, M_{i,\sigma(i)} = 1$

Premier

- Entrée : $n$ entier
- Sortie : **VRAI** si $n$ est premier

Somme

- Entrée : ensemble d'entiers $S$, $t$ entier
- Sortie : **VRAI** s'il existe $X \subseteq S \mid \sum X = t$

Post

- Entrée : 7 dominos, chacun avec 2 mots (un en haut et un en bas)
- Sortie : **VRAI** si on peut trouver une suite de domino (avec répétition) où le mot concaténé du haut est égal au mot concaténé du bas

**Difficulté :**
CPB → (facilement) dans **P**
Premier → **P**
Somme → **NP** complet
Post → indécidable

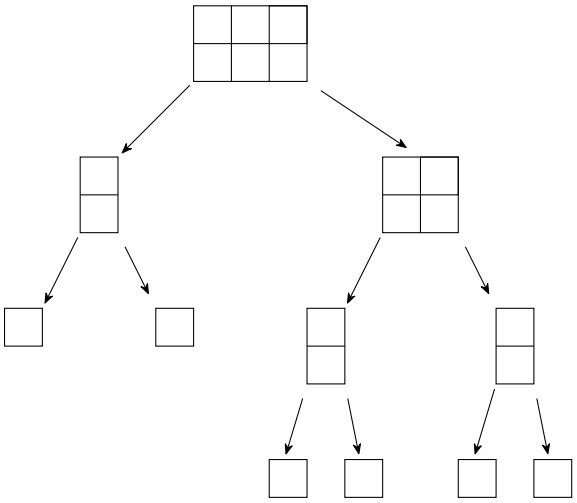
# Chapitre 0

## *Introduction : la science des arbres*

**Toy Pb**
Un *toy pb* est une version plus petite d'un problème. Il est souvent judicieux de commencer sur un toy pb.

Combien de découpe pour une tablette de chocolat $6 \times 4$ ?

**Bonnes pratiques :** Toy Pb



- Arbre binaire dont le nombre de feuilles est exactement le nombre de carré de chocolats
- Le nombre de noeuds correspond au nombre de découpes

**Proposition 0.1**
Le nombre de feuille d'un arbre binaire est égal au nombre de noeuds internes + 1

**Bijection** : chaque noeud interne emprunte les chemins D.G*

# Chapitre 1

# *Paradigme : diviser pour régner*

## 1.1 Nombre de multiplications

### 1.1.1 Exponentiation

- Entrée : $x, n > 0$
- Sortie : $x^n$

**Algorithme 1** Exponentiation rapide

**si** $n = 1$ **alors**
  **retourner** $x$
**sinon**
  **si** $n$ pair **alors**
    **retourner** $\left(x^{\frac{n}{2}}\right)^2$
  **sinon**
    **retourner** $\left(x^{\frac{n}{2}}\right)^2 \cdot x$
  **fin si**
**fin si**

**Complexité** : $\Theta(\log_2(n))$ multiplications

**A lire**
Logarithme discret

### 1.1.2 Multiplication d'entiers

Kolmogoroff : $\Theta(n^2)$ avec $n$ le nombre de chiffres des deux nombres

Karatsuba a une autre idée :

**Polynômes** : $A(X) = a_n X^n + \cdots + a_1 X + a_0$, $B(X) = b_n X^n + \cdots + b_1 X + b_0$

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

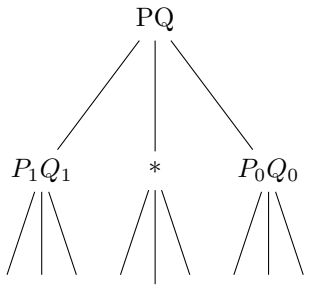
**Toy PB** : $(aX + b)(cX + d) = acX^2 + (ad + bc)X + bd = acX^2 + (ac + bd - (a - b)(c - d))X + bd \rightarrow 3$ multiplications

Maintenant comment calculer $P(X).Q(X)$ deux polynômes de degré $n$ ?

Division : $\begin{cases} P(X) = P_1(X).X^{n/2} + P_0(X) \\ Q(X) = Q_1(X).X^{n/2} + Q_0(X) \end{cases}$

$$PQ = P_1Q_1.X^n + (P_1Q_1 + P_0Q_0 - \underbrace{(P_1 - P_0)(Q_1 - Q_0)}_{*})X^{n/2} + P_0Q_0$$

Quel est le coût en multiplication ?




On a 1 multiplication par noeuds, on a 3 branchements à chaque niveau.
Au total le nombre de noeuds est $O(3^{\log_2 n}) = O(2^{\log_2 3 \log_2 n}) = n^{\log_2 3}$, donc on peut effectuer $n^{1.38}$ multiplications.

C'est la même idée utilisée par Strassen qui remarque que pour multiplier deux matrices $2 \times 2$, seuls 7 produits sont nécessaires.
Par conséquent on peut calculer le produit de deux matrices $n \times n$ en $O(7^{\log_2 n}) = O(n^{\log_2 7})$

**Question ouverte :** quelle est la meilleure complexité possible pour le produit matriciel ?

On note $\omega$ la meilleure complexité du produit matriciel[1]. Actuellement $\omega \simeq 2.38$.

Et pour le produit ? On peut atteindre $o(n \log_2 n)$.

Pour le produit de polynômes, on peut connaître les valeurs du produit en connaissant $2n+1$ valeurs différentes des deux polynômes, ce qui est linéaire en $n$, et on peut ensuite récupérer le polynôme produit des deux à l'aide de l'interpolation de Lagrange. Cependant obtenir ces $2n+1$ valeurs est coûteux.

[1] i.e ∃ un algo en $C$ [illegible] mais pas en

**FFT**
La FFT propose une manière de s'affranchir de cette limitation, mais la méthode est compliquée.

**Remarque :** Strassen interprète l'opération produit comme un tenseur[2].
Idée de Strassen : construire un tenseur qui représente le produit de matrices.

[2] tenseur = matrice 3D

*Illustration avec Karatsuba :*
$(a_1X + a_0)(b_1X + b_0) = c_1X + c_0$ dont on considère le tenseur $K = (p_{ijk})2 \times 2 \times 3$ où $p_{ijk} = 1$ si le produit $a_ib_j$ est un terme de $c_k$.

| | $b_1$ | $b_0$ |
|---|---|---|
| $a_1$ | 1 | 0 |
| $a_0$ | 0 | 0 |

$c_2$

| | $b_1$ | $b_0$ |
|---|---|---|
| $a_1$ | 0 | 1 |
| $a_0$ | 1 | 0 |

$c_1$

| | $b_1$ | $b_0$ |
|---|---|---|
| $a_1$ | 0 | 0 |
| $a_0$ | 0 | 1 |

$c_0$

Etant donné 3 vecteurs $u = (u_1, \cdots, u_l), v = (v_1, \cdots, v_m), w = (w_1, \cdots, w_n)$, on construit le tenseur $l \times m \times n$ $T$ : $u \otimes v \otimes w = (t_{ijk})$ où $t_{ijk} = u_1v_jw_k \to T$ est un tenseur de rang 1.

**Définition 1.1 (Rang d'un tenseur)**

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

**Remarques :**

▷ Si une suite de matrices converge vers une matrice $M \in M_n$ de rang plein $n$, alors $\exists n_0 \in \mathbb{N}, \forall n \leqslant n_0, M_i$ a rang plein. Or $M$ rang plein si et seulement si $\det M \neq 0$. Ensemble des matrices de rang plein est la préimage par det (continue) de $\mathbb{R} \setminus \{0\}$, c'est donc un ouvert, d'où la conclusion.

▷ Cas général : $\lim(\operatorname{rg} M_i) \leqslant \operatorname{rg} M$

▷ Dans les tenseurs, on peut avoir $\lim(\mathrm{rg}) < \mathrm{rg}(\lim)$

$$\begin{pmatrix}1 & 0\\0 & 0\end{pmatrix}\begin{pmatrix}0 & 1\\1 & \varepsilon\end{pmatrix} \xrightarrow[\varepsilon\to 0]{} \begin{pmatrix}1 & 0\\0 & 0\end{pmatrix}\begin{pmatrix}0 & 1\\1 & 0\end{pmatrix}$$

**Théorème 1.2 (Strassen)**

Le rang du tenseur de la multiplication de matrices $2 \times 2$ est $\leqslant 7$.

[[FAIRE FIGURE TENSEURS KARATSUBA]]

**A lire**
Alpha Tensor dans Nature

**Mauvaises nouvelles :**

- Calculer le rang d'un tenseur est NP-difficile
- Le rang dans $\mathbb{C}$ et dans $\mathbb{R}$ d'un tenseur peut différer

On a besoin de pouvoir construire les solutions.

**Solution**
Un algorithme résolvant le problème est Floyd-Warshall

All Paths Shortest Paths

- Entrée : Matrice $n \times n$ de valeurs $\in \mathbb{N} \cup \{+\infty\}$ $D = (d_{ij})$ de distances
- Sortie : Matrice des plus courts distances

Lorsqu'on prend le carré de $D$, on obtient la matrice des plus courts chemins de longueur inférieure ou égale à 2. La matrice cherchée peut donc être calculée en $\lceil \log_2 n \rceil$ itérations de carré → $O(n^3 \log_2 n)$.
Si on peut franchir la barrière $n^3$ pour le produit tropical on aurait un algorithme en $O(n^{3-\varepsilon})$ pour APSP. [3]

[3] Complexité du calcul en algèbre min,+ ? → conjecturé comme cubique

## 1.2 Nombre de comparaisons

On se donne à présent un tableau $T[1, \cdots, n]$ d'entiers.

### 1.2.1 Calcul du minimum d'un tableau

On peut calculer le minimum d'un tableau en $n-1$ comparaisons "$T[i] < T[j]$".

L'approche "diviser pour régner" permet de calculer le minimum en $n-1$ comparaisons avec une technique de tournoi.
Un algo classique :

```
Minimum(T, n)
1 Créer R[1..2n-1]
2 Copier T sur R[n..2n-1]
3 Pour i de n-1 à 1 :
4   R[i] <- min(R[2i], R[2i+1])
5 Retourner R[1]
```

### 1.2.2 Tri fusion

La complexité du tri fusion est en $O(n \log n)$[4].

[4] $\log n$ niveaux de coût $n$

Le VRAI coût dans le pire cas est :
$C(1) = 0$
$C(n) = C(\lfloor n/2 \rfloor) + C(\lfloor \lceil n/2 \rceil \rfloor) + n - 1$
On montre que $C(n) = \sum_{i=1}^{n} \lceil \log_2 i \rceil$

**Proposition 1.3**

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

**Démonstration**

Le nombre de tri possible avant la première question[5] est $n!$. Chaque question coupe l'espace des solutions en 2, et l'une des branches a une taille $\geqslant \frac{n!}{2}$. A l'étape $k$, il reste au moins $\frac{n!}{2^k}$ solutions possibles. On ne peut conclure que lorsque le nombre de solutions est $\leqslant 1$. On ne peut pas conclure avant que $\frac{n!}{2^k} \leqslant 1 \Rightarrow n! \leqslant 2^k \Rightarrow \lceil \log_2 n! \rceil \leqslant k$. □

[5] $T[i] < T[j]$

### 1.2.3 Médiane

$T$ tableau de $n$ valeurs, et on cherche la valeur médiane ($T_{triee}[\lfloor n/2 \rfloor]$)).

Peut-on le faire en $O(n)$ ?

**Oui !** C'est l'algorithme de BFPRT.

**Généralisation :** on veut calculer la k-ième valeur de $T_{triee}$.

**Algorithme 2** Median

```
Diviser T en n/5 blocs de taille 5
Dans chaque bloc, extraire le médian → m_1, ..., m_{n/5}
Calculer m ← Median(m_1, ..., m_{n/5}, n/10)
Partitionner T avec m, qui est alors en position l
si k = l alors
    retourner m
sinon
    si k < l alors
        retourner Median(T[1..l-1], k)
    sinon
        retourner Median(T[l+1..n], k-l)
    fin si
fin si
```

**Remarques :**

▷ Algo randomisé : remplacer les lignes 1 à 3 par "Tirer $m$ au hasard dans $T$. Presque sûrement, le temps est linéaire.

▷ Analyse plus facile : tirer $\sqrt{n}$ valeurs de $T$ au hasard, puis brute force le médian

**Chernoff :**

$\forall \varepsilon > 0, \exists c > 1, \exists n_0$ tels que $\forall m > n_0$, la probabilité que $\|M \cap P\| \notin [(\frac{1}{3} - \varepsilon)\sqrt{n}, (\frac{1}{3} + \varepsilon)\sqrt{n}]$ est au plus $\frac{1}{c\sqrt{n}}$.
"La probabilité d'échec est exponentiellement faible en la taille de l'échantillon ($\sqrt{n}$)" → $m$ est un pivot

## 1.3 Master theorem

# Chapitre 2

# *Algorithmes gloutons*

## 2.1 Introduction

**Stratégie :** con construit une solution pas à pas, en ne remettant pas en cauqse les choix précédents.

### 2.1.1 Exemple : flots

- Entrée : Graphe orienté $D = (V, A)$, source $s$ et terminal $t$
- Sortie : un ensemble maximum de chemins de $s$ à $t$ avec arcs disjoints

Ici l'algorithme glouton ne marche pas du tout.

**Algorithme 3** Recherche gloutonne de $st$−chemins

```
S ← ∅
tant que il existe un st−chemin P faire
    S ← S ∪ {P}
    A ← A ∖ A(P)
fin tant que
```

Cependant on peut l'améliorer pour qu'il marche :

**Algorithme 4** Recherche gloutonne améliorée de $st$−chemins

```
tant que il existe un st−chemin P faire
    Inverser les arcs de P
fin tant que
Effacer les arcs inversés un nombre pair de fois, et inverser les autres
S ← ∅
tant que il existe un st−chemin P faire
    S ← S ∪ {P}
    A ← A ∖ A(P)
fin tant que
```

**Définition 2.1**
Une *st−coupe* est une partition $S, T$ de $V$ telle que $s \in S$ et $t \in T$.
On définit la *valeur de la coupe* comme le nombre d'arcs de $S$ à $T$.

**Remarque :** La valeur de la coupe est supérieure ou égale au nombre de $st$−chemins disjoints.

On va maintenant prouver la validité de l'algorithme.

1. Une itération de la première boucle fait décroître la valeur de **toutes** les coupes de 1. Le nombre d'itérations est égal à la valeur minimale d'une coupe (que l'on note $k$).
2. Après l'exécution de la ligne 4, $d^+(s) = d^-(t) = k$ et $\forall v \notin \{s,t\}, d^+(v) = d^-(v)$.
3. Une marche gloutonne partant de $s$ arrive forcément en $t$, donc je calcul d'un chemin est glouton. Lorsqu'on enlève après une itération de la ligne 8, les conditions de Kirschoff sont conservées et le degré sortant de $s$ décroît exactement de 1, on va donc avoir $k$ itérations et $k$ chemins disjoints.

**A vérifier**
C'est l'argument de Kirschoff.

**Théorème 2.2**
$\forall \varepsilon > 0$, il existe une algorithme pour flot maximum en $O((n+m)^{1+\varepsilon})$ avec $m$ le nombre de noeuds et $m$ le nombre d'arcs.

**Histoire**
Ce théorème a été prouvé il y a environ 3 ans, et est très très complexe.

### 2.1.2 Exemple : codage binaire

On a un mot $M \in A^*$ et on veut le transformer en un mot binaire en remplaçant chaque lettre par un mot de $0, 1$. Idéalement, on veut pouvoir retrouver $M$ et compresser le mot obtenu.
Pn veut un code préfixe optimal pour chaque lettre $x$ dans $M$, ayant $m_x$ occurrences, on veut associer un mot $\varphi(x) \in \{0,1\}^*$ tel que :

- préfixe : si $x \neq y$ alors $\varphi(x)$ n'est pas préfixe de $\varphi(y)$
- optimal : $\varphi(x)$ a longueur minimale

Un algorithme glouton résolvant ce problème de manière optimale est l'algorithme de Huffman.

**Limites de la compression :**

Une borne inférieure pour la longueur d'un codage binaire d'un mot $M$ avec occurrences de lettres $\{m_x, x \in A\}$ est $\log_2 \left( \dfrac{l!}{\prod_{x \in A} m_x!} \right)$ avec $l = |M|$.

Et asymptotiquement ?

$M$ de longueur $l$ et fréquences des lettres $f = \{f_x = \dfrac{m_x}{l} : x \in A\}$. Le nombre d'anagrammes de $M$ est $\sim 2^{H(f) \cdot l}$ où $H(f) = \sum_{x \in A} -f_x \log_2 f_x$.
Ici $H(f)$ représente le "coefficient d'allongement" de $M$ (borne inf) si on le code en binaire, c'est l'*entropie de Shannon*.
"La masse des mots binaires est concentrée sur l'équiréparti"[1]

[1] j'ai pas compris ce qu'il a voulu dire

## 2.2 Arbre couvrant de poids minimal

### 2.2.1 Recherche d'un arbre couvrant

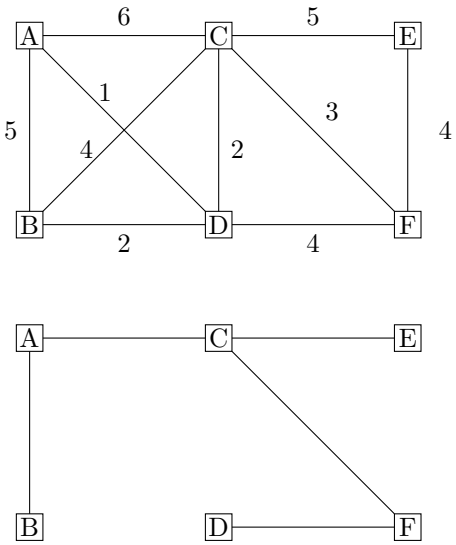



Figure 1: Le sous-graphe est un arbre couvrant du graphe.

**Définition 2.3**

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

**Proposition 2.4**

Tout graphe connexe possède un arbre couvrant.

**Démonstration (a. par induction)**

Considérer un $uv$–chemin $P$ glouton[2]. Tous les voisins de $v$ sont dans $P$ donc $G \setminus v$ est connexe. Il existe donc $A'$ couvrant $G \setminus v$ et donc $A' \cup \{vw\}$ arbre couvrant de $G$, avec $w$ un voisin de $v$. □

[2] C'est à dire que tant qu'on peut rajouter des arêtes on le fait

**Démonstration (b.)**

Considérer un ensemble $A$ connexe minimal. □

**Démonstration (c.)**

Considérer un ensemble $A$ acyclique maximal. □

**Remarque :** Le nombre d'arêtes de $A$ est $n - 1$, où $n = |V|$.

**Calcul effectif :**

- On peut utiliser un arbre de parcours : $O(n + m)$
- 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[3] : comment décider si une arête $xy$ reçue forme un cycle ou pas ?

[3] La meilleure structure est l'union-find

**Complexité :**

Pessimiste : $O(n \cdot m)$

Pire des cas : $O(m + n^2)$

En ajoutant l'union par rang, on obtient une complexité en $O(m + n \log n)$.

**Remarques :**

▷ La gestion des composantes peut se faire en $O(n \cdot \log^* n)$ et même mieux à l'aide d'un union-find

**Algorithme 5** Arbre couvrant

$A \leftarrow \varnothing$
pour tout $v \in V$, $c(v) \leftarrow v$
pour tout $v \in V$, $comp(v) \leftarrow \{v\}$
**pour** tout arête $xy \in E$ **faire**
    **si** $c(x) \neq c(y)$ **alors**
        $A \leftarrow A \cup \{xy\}$
        $comp(c(x)) \leftarrow comp(c(x)) \cup comp(c(y))$
        $C \leftarrow comp(c(y))$
        **pour** tout $z \in C$ **faire**
            $c(z) \leftarrow c(x)$
        **fin pour**
    **fin si**
**fin pour**

▷ Coder l'algorithme :

- Observer l'apparition de la composante géante (Erdos-Renyi)
- Commenter et décommenter l'union par rang

### 2.2.2 Arbre couvrant de poids minimal

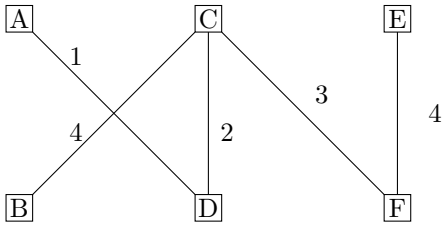



Figure 2: L'arbre couvrant de poids minimal du graphe de la figure 1.

ACPM

- Entrée : $G = (V, E)$ et $\omega : E \to \mathbb{N}$
- Sortie : $A$ arbre couvrant tel que $\omega(A)$ minimal

**Algorithme 6** Kruskal

Trier les arêtes de $E$ par ordre croissant de $\omega$
Appliquer l'algorithme classique d'arbre couvrant

**Proposition 2.5**

L'algorithme de Kruskal appliqué à un graphe non-orienté connexe $G = (S, A)$ renvoie un arbre couvrant $T = (S, B)$ de poids minimal.

**Démonstration**

Montrons tout d'abord que $T$ est bien un arbre couvrant. On a $T = (S, B)$ donc $T$ est couvrant. On a pour invariant que le sous-graphe en construction est acyclique. Si on s'arrête en atteignant $|S| - 1$ étapes, on a un sous-graphe acyclique à $|S| - 1$ arêtes qui est donc connexe. Il s'agit d'un arbre.

Montrons que ceci est toujours le cas. Supposons que l'on a considéré toutes les arêtes et que le sous-graphe n'est pas connexe. Il comporte au moins deux composantes connexes. Comme $G$ est connexe et ne comporte qu'une seule composante connexe, deux de ces composantes connexes sont reliés dans $G$ pour une arête $\{x, y\}$.

Or, cette arête $\{x, y\}$ a été envisagée dans l'algorithme de Kruskal. A ce moment, elle ne reliait pas les composantes connexes qui contenaient $x$ et $y$, sinon, ce serait toujours le cas. Elle a donc due être ajoutée. Absurde.

Montrons maintenant que la propriété suivante est un invariant en notant $\underline{T} = (S, \underline{B})$ le sous-graphe en cours de construction :

$\underline{T} = (S, \underline{B})$ est un sous-arbre d'un graphe $T^* = (S, B^*)$ de poids minimal de $G$.

**Initialisation :** Avant le premier tour de boucle, on a $\underline{T} = (S, \varnothing)$. Il existe donc un arbre couvrant $T^*$ de poids minimal (car l'ensemble des arbres couvrants est fini non vide) et $\underline{T}$ est bien sous-graphe de $T^*$.

**Hérédité :** Supposons la propriété vraie au début d'un tour de boucle. On a donc $\underline{T} = (S, \underline{B})$ sous-graphe d'un arbre couvrant $\underline{T^*} = (S, \underline{B^*})$ de poids minimal de $G$. Dans un passage dans la boucle, distinguons trois cas :

1. L'arête $\{x, y\}$ n'est pas ajoutée à $\underline{T}$, on a $\overline{T} = \underline{T}$. Et la propriété reste vraie avec $\overline{T^*} = \underline{T^*}$.

2. L'arête $\{x, y\} \in \underline{B^*}$, dans ce cas, $\overline{T} = (S, \underline{B} \cup \{\{x, y\}\})$. On a encore $\overline{T}$ sous graphe de $\overline{T^*} = \underline{T^*}$ arbre couvrant de poids minimal.

3. Si $\{x, y\} \in A \backslash B^*$, on a $\overline{T} = (S, \underline{B} \cup \{\{x, y\}\})$ qui n'est pas un sous-graphe de $\underline{T^*}$. Le sous-graphe $T^* + \{x, y\}$ comporte un cycle (contenant $\{x, y\}$). Comme $x$ et $y$ ne sont pas dans une même composante connexe de $T$, le chemin de $x$ à $y$ dans $\underline{T^*}$ qui utilise le reste du cycle, ne peut pas être en entier dans $\underline{T}$. Donc il existe une arête $a \in \underline{B*} \backslash \underline{B}$ sur ce cycle. On ne peut pas encore avoir rencontré $a$, sinon on aurait ajouté $a$. En effet, comme $\underline{T}$ est sous-graphe de $T^*$, il ne peut pas y avoir d'autre chemin reliant dans $\underline{T}$ les extrémités de $a$ (sinon cela crée un cycle dans $\underline{T^*}$).

   Comme on considère les arêtes dans l'ordre croissant, on a donc $p(\{x, y\}) \leq p(a)$. On considère $\overline{T*} = \underline{T^*} - a + \{x, y\}$ qui est encore un arbre couvrant. $p(\overline{T^*}) = p(\underline{T^*}) - p(a) + p(\{x, y\}) \leq p(\underline{T^*})$. Donc $\overline{T^*}$ est aussi de poids minimal (et $p(\overline{T^*}) = p(\underline{T^*})$).

L'invariant est conservée $\overline{T}$ est un sous-graphe de $\overline{T^*}$ qui est un arbre couvrant de poids minimal.

**Correction :** A la fin de la boucle $T = (S, B)$ est un arbre couvrant, sous-graphe d'un arbre couvrant de poids minimal, donc égal à cet arbre de poids minimal. □

**Définition 2.6**

L'*espace ambiant* est l'ensemble de tous les arbres couvrants.
Deux arbres sont voisins s'ils diffèrent de deux arêtes.

## 2.3 Algorithmes gloutons

Soit $H = (S, V)$ 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, c'est-à-dire qu'il peut répondre en $O(1)$ à une question "$X \in S$" lorsque $X \subseteq V$.

MAXIMISATION

- Entrée : $H = (S, V)$ (oracle), $\omega : V \to \mathbb{N}$
- Sortie : $X \in S$ tel que $\omega(X)$ maximal

**Algorithme 7** L'algorithme GLOUTON

```
A ← ∅
Trier V par ordre décroissant de ω
pour tout v ∈ V faire
    si A ∪ {V} ∈ S alors
        A ← A ∪ {v}
    fin si
fin pour
retourner A
```

**Remarque :** Si $S \subseteq 2^E$ est l'ensemble des parties cycliques de $S$, alors GLOUTON retourne bien un arbre de poids maximal.

**Exemple (autre exemple)**

Considérer une matrice $M$ de taille $n \times m$ sur un corps $\mathbb{K}$.
$V = \{\text{vecteurs colonnes}\}$, $S = \{X \subseteq V \mid X \text{ famille libre}\}$
$\forall \omega : V \to \mathbb{N}$, GLOUTON retourne une famille libre maximisant $\omega$

**Exemple (Kruskal)**

Si $V$ est l'ensemble des arêtes d'un graphe et $S$ représente les acycliques, alors GLOUTON retourne acyclique max.

**Lemme 2.7**

Un ensemble de sommets est acyclique si et seulement si les colonnes formées par ces sommets dans la matrice de ... est libre

**Démonstration**

$\boxed{\Leftarrow}$ non acyclique $\Rightarrow$ il existe un cycle $x_1x_2, \cdots, x_{n-1}x_n$ et la somme des colonnes vaut 0

$\boxed{\Rightarrow}$ acyclique A $\Rightarrow$ il existe une feuille $x$ de l'acyclique $\Rightarrow$ on applique l'induction sur $A \setminus x$
□

**Lemme 2.8**

GLOUTON retourne toujours une base de poids maximal lorsque $\omega$ est une fonction de poids sur les colonnes d'une matrice.

**Démonstration**

Suffit de démontrer que pour toutes deux bases d'un EV $B \neq B'$, $\forall x \in B \setminus B'$, $\exists x' \in B' \setminus B$, $(B \cup x') \setminus x$ et $(B' \cup x) \setminus x'$ sont deux bases. □

## 2.4 Matroïdes

**Définition 2.9 (Matroïde)**

$H = (V, S)$ est un *matroïde* si

1. $S \neq \varnothing$ [4]
2. $\forall X \in S,\ \forall Y \subseteq X,\ Y \in S$ [5]
3. $\forall X, Y \in S,\ |Y| > |X| \Rightarrow \exists y \in Y \setminus X,\ X \cup \{y\} \in S$ [6]

[4] Non vacuité

[5] Clôture

[6] Échange

**Exemple**

3. est vérifié par les familles libres (c'est le théorème de la base incomplète)

**Théorème 2.10**

L'algorithme glouton sur $H$ retourne OPT $\forall \omega$ si et seulement si $H$ est un matroïde.

**Exemple (de matroïdes)**

1. Matroïde graphique : les acycliques dans un graphe
2. Matroïde vectoriel : les familles libres dans une matrice
3. Matroïde uniforme : on se fixe $k$, on pose $S = \{X \subseteq V \mid |X| \leqslant k\}$
4. Matroïde de partition : on se donne $V V_1 \sqcup \cdots \sqcup V_l$ et $S = \{X \subseteq V \mid |X \cap V_i| \leqslant 1 \forall i \in [[1, l]]\}$

**Remarques :**

▷ Certains matroïdes sont représentables comme matroïdes vectoriels, mais il en existe des non représentables.

▷ Une étape de t : intersection, $S_1$ et $S_2$ deux matrices sur $V$, $\omega : V \to \mathbb{N}$
Trouver $X$ de $\omega(X)$ max tel que $X \in (S_1 \cap S_2)$ → il existe un algo polynomial.

**Exemple**

Couplage max : $X \subseteq E$ s'exprime sous forme d'intersection de deux matroïdes de partition sur $A$ et $B$.

# Chapitre 3

# *Génération aléatoire uniforme*

**But :**

- Tester des hypothèses
- Génération d'inputs (benchmarks)

## 3.1 Générateurs de nombres aléatoires dans $[[1, N]] := [N]$

**Littérature**
Voir littérature pour des informations sur les générateurs de nombres aléatoires

**Éviter rand() % k**
Biais vers les petites valeurs

## 3.2 Permutation aléatoire uniforme

Utilisé : preprocess sur QuickSort.

**Algorithme 8** Permut

**Entrée :** $T = [1, \cdots, n]$
1: **pour** $i$ de $n$ à 2 **faire**
2: $j \leftarrow$ nb aléatoire dans $[i]$
3: échanger $T[i] \leftrightarrow T[j]$
4: **fin pour**

Coder, évaluer la distribution des longueurs des cycles, la probabilité que la permutation soit un dérangement ($T[i] \neq i \forall i$)

## 3.3 Matrices $O-1$ et graphes aléatoires

On peut tirer chaque arête d'un graphe avec probabilité $p$.
Modèle $\mathcal{G}(n,p)$ où $G \in \mathcal{G}(n,p)$ si et seulement si $G$ a $n$ sommets et proba $p$ d'arêtes.

! Tous les graphes aléatoires se ressemblent.

**Exemple (Illustration)**

- Loi 0 – 1 Pour toute formule $F$ du premier ordre, $\lim_{n\to\infty} \text{proba}(G \vdash F : G \in \mathcal{G}(n,p)) = 0$ ou 1

  C'est cependant faux au second ordre.

- Si on tire chaque parie d'entiers $i, j \in \mathbb{N}$ avec proba 1/2, on obtient presque surement le même graphe dénombrable $R$ à isomorphisme près. La limite vaut 1 si et seulement si $R \vdash F$

## 3.4 Tirer un arbre aléatoire uniforme sur $[n]$

1er essai :

- Tirer une permutation aléatoire des arêtes possibles (il y en a $\binom{n}{2}$)
- Appliquer Kruskal

Or l'espérance de diamètre n'est pas $\sqrt{n}$

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

**Théorème 3.1**
Il y a $n^{n-2}$ arbres sur $[n]$.

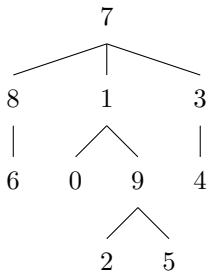
**Démonstration**
Codage de Prüfer qui réalise une bijection des arborescences enracinées sur $[n]$ avec les suites de longueur $n-1$ sur $[n]$. [1] □

[1] $n-1$ n'est pas une erreur, on travaille ici avec des arborescence et il reste un choix de sommet pour la racine

**Algorithme 9** Prüfer

**Entrée :** $A$ arborescence
1: $C \leftarrow \varepsilon$
2: **tant que** $|A| > 1$ **faire**
3: trier les feuilles par ordre croissant d'indice
4: **pour** toute feuille $F$ **faire**
5: $C \leftarrow C \bullet \text{parent}(F)$
6: **fin pour**
7: supprimer les feuilles de $A$
8: **fin tant que**
9: **retourner** $C$

**Exemple**



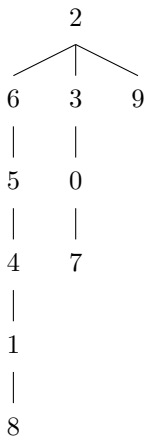

se code en 193987717

**Proposition 3.2**

Ce codage admet une unique réciproque.

**Exemple**

Le décodage de 012342562 est




Tirer un arbre aléatoire sur $[n]$ :

- Tirer une suite aléatoire de $[n]^{n-1}$
- Appliquer décodage de Prüfer
- Oublier la racine

Tirer un arbre couvrant aléatoire uniforme dans un graphe connexe : faire une marche aléatoire sur le graphe

**Proposition 3.3**

$G$ graphe connexe et $xy$ arête, alors $\mathcal{P}$ ($A$ arbre couvrant tiré uniformément contient $xy$) = résistance électrique entre $x$ et $y$ si toutes les arêtes ont une résistance de $1\Omega$.

Le nombre d'arbre couvrant dans un graphe connexe $G$ est égal au signe près au déterminant d'un mineur principal de la matrice laplacienne de $G$ (notée $L(G)$).
$L(G) = A(G) - D(G)$ avec :

- $A(G) :=$ matrice d'adjacence
- $D(G) :=$ diagonale où $d_{vv}$ est le degré de $v$

**Exemple (Illustration pour le graphe complet à $n$ éléments)**

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

Son déterminant vaut $n^{n-2}$

**Voir plus loin**
Cauchy-Binet

## 3.5 Cas du couplage

Comment tirer un couplage uniformément dans un graphe biparti ?

**Théorème 3.4 (Valient)**

Compter le nombre de couplages dans un graphe biparti est $\sharp\mathbf{P}$–complet[2].

[2] analogue de **NP**–complet pour la complexité de comptage de solution d'un problème

**Remarque :** Si on savait compter, on saurait tirer uniformément.

## 3.6 Instances typiques

Tirer uniformément une instance → instances typiques } instances en crypto

# Chapitre 4

# *NP -Complétude*

## 4.1 La classe NP , définition intuitive

Un problème de décision est dans **NP** s'il existe un algorithme polynomial $A \to \{V, F\}$ et une constante $d$ telle que $X$ **VRAI** si et seulement si il existe un certificat $C$ de taille au plus $|X|^d$ tel que $A(X, C)$ **VRAI**

En pratique, on sait facilement vérifier qu'une instance est vraie si on se fournit la preuve (de taille polynomiale).

C'est en général très facile.

- Premier $\in$ **NP** (Pratt 75)
- Somme
  - Entrée : $n_1, \cdots, n_l, S \in \mathbb{N}$
  - Sortie : **VRAI** si et seulement si $\exists I \subseteq [l],\ \sum_{i \in I} n_i = S$

  $\in$ **NP**

**Remarque :** **NP** est biaisé vers **VRAI** . On aurait pu biaiser vers **FAUX** , c'est la classe **coNP** .

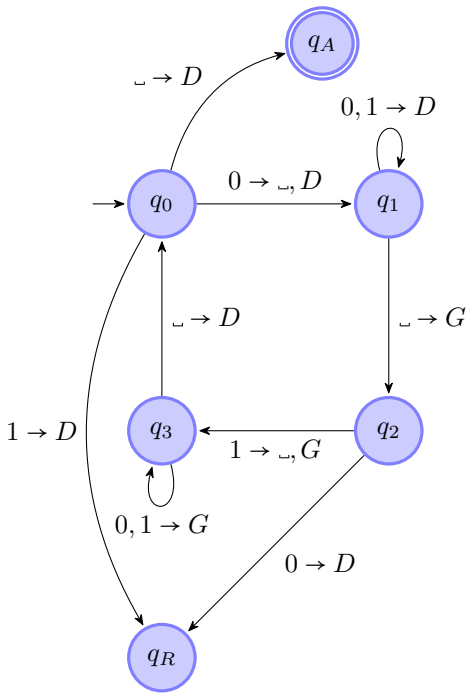
Par exemple le problème SAT

- Entrée : Une formule $F$ en forme normale conjonctive en les variables $x_1, \cdots, x_n$ booléennes
- Sortie : **VRAI** si il existe une valuation qui satisfait $F$

est clairement dans **NP** car il suffit de donner la solution, mais SAT n'est pas dans **coNP** (conjecture).

## 4.2 Les deux (vraies) définitions de NP

Automates très limités : $\{0^n 1^n, n \in \mathbb{N}\}$ n'est pas rationnel.
Turing propose de donner un ruban mémoire à l'automate où il pourra lire, écrire et déplacer la tête de lecture.




Machine de Turing $\mathcal{M}$ décidant $\{0^n 1^n, n \in \mathbb{N}\}$

Au début du calcul, un mot $m \in \{0,1\}^*$ est écrit sur le ruban, et au cours des changements d'états, si l'automate arrive dans $q_A$ (état acceptant), $m$ est accepté et s'il arrive dans $q_R$ (état rejetant), $m$ est rejeté.

$\mathcal{M}$ décide $L$ si :

- tous les calculs, sur toutes les traces, terminent
- $M$ accepté par $\mathcal{M}$ si et seulement si $M \in L$

Si tous les calculs terminent en temps polynomial, alors $\mathcal{M}$ est une MT polynomiale.
Un langage $L \in \mathbf{P}$ s'il existe une MT polynomiale qui décide $L$.

**Définition 4.1 (1)**

Un langage $L \in \mathbf{NP}$ s'il existe un polynôme $P$ et un langage $L' \in \mathbf{P}$ tels que $L = \{M \in \mathcal{A}^* \mid \exists c \in \mathcal{A}^*,\ |c| \leqslant P(|M|) \& M \cdot c \in L'\}$

**Ajout du non déterminisme :**

- On exige toujours que tous les calculs terminent
- Un mot $M$ est accepté par une MT non déterministe s'il existe un chemin vers $q_A$. De plus, si tous les chemins terminent en un nombre polynomial d'étapes en la taille de l'entrée, on parle de MTND polynomiale.

**Définition 4.2 (2)**

$L \in \mathbf{NP}$ s'il existe une MTND polynomiale qui décide $L$.

**Proposition 4.3**

Définition 1 $\Leftrightarrow$ Définition 2

**Proposition 4.4**

$L \in \mathbf{coNP}$ si $\mathcal{A}^* \setminus L \in \mathbf{NP}$

Une classe intéressante : $\mathbf{NP} \cap \mathbf{coNP}$

**A voir**

Photos sur Wikipédia de Jack Edmonds

## 4.3 Réduction

**Définition 4.5 (Réduction polynomiale)**

Un langage $L$ est polynomialement réductible à un langage $L'$ s'il existe une fonction $f$ calculable en temps polynomial de $\mathcal{A}^* \to \mathcal{A}^*$ tel que $M \in L \Leftrightarrow f(M) \in L'$, aussi appelée "many to one reduction" ou "Karp reduction".
On note $L \leqslant L'$ et cela correspond à un ordre partiel de difficulté des langages (au détail de l'antisymétrie).

**Intuitivement :** Un problème 1 se réduit à un problème 2 s'il existe un algorithme polynomiale qui transforme les instances de problème 1 en instances de problème 2 en respectant **VRAI** / **FAUX**

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

Intuitivement, problème 2 est au moins aussi difficile que problème 1.

**Définition 4.6 (Réduction de Turing)**

On peut utiliser plus d'une fois un oracle sur $L'$.

**Théorème 4.7 (Cook-Levin)**

$\exists L' \in \mathbf{NP}$ tel que $\forall L \in \mathbf{NP}$, on a $L \leqslant L'$.

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

**Intérêt :**

- Si vous n'arrivez pas à trouver un algorithme polynomiale pour un problème $L$, essayer de montrer qu'il est **NP**−complet
- Pour ce faire, on cherche à réduire $SAT$ en $L$ en transformant les instances

## 4.4 Réductions classiques

**A voir**
La liste des 21 problèmes de Karp

### 4.4.1 3-SAT

3-SAT

- Entrée : Formule FNC $F$ dont toutes les clauses ont 3 littéraux
- Sortie : **VRAI** si $F$ satisfaisable

**Proposition 4.8**

3-SAT est **NP**−complet

**Démonstration**
On montre que SAT $\leqslant$ 3-SAT
On se donne une formule $F = C_1 \wedge C_2 \wedge \cdots \wedge C_m$ de SAT en variables $x_1, \cdots, x_n$.
On veut transformer $F$ en une formule équivalente de 3–SAT.
On suppose $C_1 = l_1 \vee l_2 \vee \cdots \vee l_t$ et on veut exprimer $C_1$ comme conjonction de clauses de taille 3.

- Si $t \leqslant 3$ on ne change pas $C_1$
- Si $t \geqslant 3$, on va créer $t-3$ nouvelles variables $y_1, \cdots, y_{t-3}$ et on remplace $C_1$ par $(l_1 \vee l_2 \vee y_1) \wedge (\overline{y_1} \vee y_2 \vee l_3) \wedge (\overline{y_2} \vee y_3 \vee l_4) \wedge \cdots \wedge (\overline{y_{t-3}} \vee l_{t-1} \vee l_t)$

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

### 4.4.2 Clique

CLIQUE

- Entrée : Graphe $G = (V, E)$, entier $k$
- Sortie : **VRAI** si $G$ a une clique de taille $k$

**Proposition 4.9**
CLIQUE est **NP**–complet.

**Démonstration**
1. Vérifier que CLIQUE $\in$ **NP**
2. Réduire un problème bien connu **NP**–complet à CLIQUE

   Montrons que 3-SAT $\leqslant$ CLIQUE. Soit $F = C_1 \wedge \cdots \wedge C_m$ où chaque $C_i$ est composé de 3 littéraux $x_i$ ou $\overline{x_i}$. On forme un graphe $G_F$ sur $3m$ sommets, un pour chaque littéral dans chaque clause. Les arêtes de $G_F$ relient les littéraux $l_i, l_j$ de clauses $\neq$ et vérifiant $l_i \neq \overline{l_j}$

   Sont équivalents $G_F$ a une clique de taille $m$ et $F$ satisfaisable :

   - $\boxed{\Leftarrow}$ si $F$ est satisfaisable, il existe une affectation avec au moins un littéral **VRAI** par clause $\rightarrow$ choisir un seul littéral par clause donne une clique de taille $m$ de $G_F$
   - $\boxed{\Rightarrow}$ Toute clique de taille $m$ de $G_F$ intersecte un littéral par clause sans avoir $l_i$ et $\overline{l_i}$ $\rightarrow$ étendre cette pré-affectation en une affectation pour $F$.

□

### 4.4.3 SOMME

SOMME

- Entrée : $s_1, \cdots, s_n, s$ entiers
- Sortie : **VRAI** s'il existe $I \subseteq [n]$ tel que $\sum_{i \in I} s_i = s$

**Théorème 4.10**
SOMME est **NP**–complet

**Attention**
La taille du codage est $\sum \log s_i + \log_s$

**Démonstration**

- SOMME $\in$ **NP**
- On va montrer que 3–SAT $\leqslant$ SOMME

  On construit une fonction $f$ des instances de 3–SAT vers les instances de SOMME telle que $I$ **VRAI** si et seulement si $f(I)$ **VRAI**

  On se donne $F = C_1 \wedge \cdots \wedge C_n$ en les variables $x_1, \cdots, x_n$ et on lui associe des entiers.

Demander reste de la preuve à qqn □

**Astuce**
Utiliser des représentations de nombres en base 10 mais que n'utilisant des 0 et des 1 → par de retenue et codage en binaire

### 4.4.4 Sac à dos

SAC A DOS

- Entrée :
  - Ensemble de couples $(p_i, v_i)$ entiers, $i \in [n]$
  - Volume total $v$
  - Prix à atteindre $p$
- Sortie : **VRAI** s'il existe $I \subseteq [n]$ tel que :
  - $\sum_{i \in I} v_i \leqslant v$
  - $\sum_{i \in I} p_i \geqslant p$

**Remarque :** Ce problème est dans **P** si codé $\sum p_i + \sum v_i + v + p$ et **NP** si codé en binaire.

**Démonstration**
SOMME $\leqslant$ SAC A DOS
Instance de somme $I : s_1, \cdots, s_n, s$

$$\begin{bmatrix} p_i & = s_i \ \forall i \in [n] \\ v_i & = s_i \\ v & = s \\ p & = s \end{bmatrix}$$

□

### 4.4.5 STABLE

STABLE

- Entrée : $G$ graphe, $k$ entier
- Sortie : **VRAI** s'il existe $k$ sommets 2 à 2 non reliés

STABLE est **NP** -complet car CLIQUE $\leqslant$ STABLE, il suffit de remplacer les arêtes par des non-arêtes et vice-versa. □

### 4.4.6 COUVERTURE (VERTEX COVER)

COUVERTURE

- Entrée : Graphe $G$, entier $k$
- Sortie : **VRAI** s'il existe $k$ sommets qui intersectent toutes les arêtes

**Proposition 4.11**
COUVERTURE est **NP** -complet.

**Démonstration**
Noter que $G$ a un stable de taille $n-k$ si et seulement si $G$ a une couverture de taille $k$. □

### 4.4.7 FEEDBACK VERTEX SET

FEEDBACK VERTEX SET

- Entrée : Graphe $G$, entier $k$
- Sortie : **VRAI** s'il existe un ensemble $X$ d'au plus $k$ sommets tels que $G \setminus X$ est acyclique

**Proposition 4.12**
FVS est **NP** -complet.

**Démonstration**
On veut transformer une instance de COUVERTURE en FVS.
En trichant (version multi-arête) : on double les arêtes (ce qui créé un cycle entre chaque sommets). □

### 4.4.8 COUPLAGE 3D

COUPLAGE 3D

- Entrée : Un tenseur de $\{0,1\} = (t_{ijk})$
- Sortie : **VRAI** s'il existe un couplage 3D, c'est la donnée de 2 permutation $\sigma$ et $\tau$ de $S_n$ vérifiant $t_{i,\sigma(i),\tau(i)} = 1 \ \forall i \in [n]$

Une autre façon de voir couplage 3D :
On considère 3 ensembles $H, L, C$ de taille $n$ et on ajoute un triangle $(i,j,k) \in H \times L \times C$ pour chaque $t_{ijk} = 1$.

**Proposition 4.13**
COUPLAGE 3D est **NP** -complet.

### 4.4.9 MAXCUT

MAXCUT

- Entrée : Graphe $G = (V,E)$, entier $k$
- Sortie : **VRAI** s'il existe une coupe $V = A \sqcup B$ tel que le nombre d'arêtes entre $A$ et $B$ (noté $e(A,B)$) est au moins $k$

**Proposition 4.14**
MAXCUT est **NP** -complet car NAE 3–SAT $\leqslant$ MAXCUT

**NAE 3-SAT**
Pareil que 3-SAT mais pas avec $V, V, V$

**Démonstration**
Soit $F$ une formule $C_1 \wedge \cdots \wedge C_m$ en les variables $x_1, \cdots, x_n$ et $C_1 = x_1 \vee \overline{x_2} \vee x_3$
On transforme en une instance MAXCUT avec le gadget suivant :
[[INSERER GADGET]]
On ajouter les triangles correspondants aux clauses
$F$ est NAE-3-SAT satisfiable si et seulement si MAXCUT$(G_F, nN + 2n)$ (on choisit $N > 2n$).
□

### 4.4.10 Circuit Hamiltonien orienté

Circuit Hamiltonien orienté

- Entrée : Graphe orienté
- Sortie : **VRAI** s'il existe un cycle qui passe par tous les sommets

**Proposition 4.15**

Ce problème est **NP** -complet.

**Démonstration**

On réduit depuis 3-SAT. On considère une formule $F$ en variables $x_1, \cdots, x_n$
[[INSERER IMAGE]]
La clause $C_1 = x_1 \vee \overline{x_2} \vee x_n$ est codée par 1 sommet. On définit les zones des $C_i$ comme écartés d'au moins 1 sommet. □

## 4.5 NP ∩ coNP

### 4.5.1 Exemples

Problème de décision qui admettent un certificat de réponse **VRAI** et de réponse **FAUX** polynomiaux à vérifier → problèmes bien caractérisés.

**Exemple** (PREMIER)

- ∈ **coNP** car certificat de réponse **FAUX** est un diviseur
- ∈ **NP** (Pratt 75)

**Exemple**

COUPLAGE PARFAIT

- Entrée : Graphe $G$
- Sortie : **VRAI** s'il existe un couplage qui couvre tous les sommets
- ∈ **NP** car il suffit de prendre la solution
- ∈ **coNP**

**Théorème 4.16 (Tutte '46)**

$G$ est un couplage parfait si et seulement si pour tout ensemble $X$ de sommets, le nombre de composantes connexes de taille $i$–impaire de $G \setminus X$ (noté $c_i(X)$) vérifie $c_i(X) \leqslant |X|$

Donc $X$ est bien certificat **coNP** pour couplage parfait.

**Exemple**

Système d'inéquation

- Entrée : $m$ inéquations linéaires $\sum_{j=1}^{n} a_{ij}x_j \leqslant b_i$
- Sortie : **VRAI** s'il existe une solution

**Taille entrée :** $\sum \log a_{ij} + \sum \log b_i$

- $\in$ **NP** s'il existe une solution, elle peut être exprimée par un sous-système d'égalité, alors s'exprime comme rapport de déterminant → polynomial en la taille de l'entrée.
- $\in$ **coNP** (Farkas 1902)

**Exemple**

Décomposition en facteurs premiers (décision)

- Entrée : entier $n$, entier $k$
- Sortie : **VRAI** si $n$ admet un diviseur $\leqslant k$
- $\in$ **NP**, il suffit de fournir un diviseur $\leqslant k$
- $\in$ **coNP**, il suffit de fournir la décomposition en produit de facteurs premiers (Pratt '75)

A partir de ce problème de décision, on peut construire la décomposition d'un nombre en produits de facteurs.

### 4.5.2 Dualité

Certificat **coNP** sont souvent basé sur un problème dit *dual*.
Exemple type : `MaxFlot = MinCut`

**Exemple**

Système d'équations linéaire

- Entrée : Système $Ax = B$
- Sortie : **VRAI** s'il existe une solution

Certificat **coNP** : $y$ tel que $A^T y = 0$ et $B^T y \neq 0$. Cela est équivalent à une combinaison linéaire des lignes des systèmes qui donne $0 = 1$.

**Exemple** (Système d'inéquations)

$\in$ **coNP** car :

> **Lemme 4.17 (Farkas 1902)**
>
> $Ax = B$ n'a pas de solutions si et seulement si il existe une combinaison linéaire positive d'inéquations qui donne $0 \leqslant -1$

### Exemple

Systèmes polynomiaux

- Entrée : $P_1, \cdots, P_n$ des polynômes en variables $x_1, \cdots, x_n$ à coefficients entiers
- Sortie : **VRAI** s'il existe $x^* \in \mathbb{C}^n$ tel que $P_i(x^*) = 0 \ \forall i \in [n]$

Cas particulier $Ax - B = 0$
En degré 2, on peut aisément coder 3–SAT. Soit $F = C_1 \wedge \cdots \wedge C_n$ une formule 3–SAT de variables $x_1, \cdots, x_n$ : $\begin{bmatrix} x_1^2 - x_1 = 0 & P_1 \\ \vdots & \vdots \\ x_n^2 - x_n = 0 & P_n \end{bmatrix}$
Equations pour les clauses $C_1 = x_1 \vee \overline{x_2} \vee x_3$

$$\begin{array}{lll} Q_1 & x_1 + (1 - x_2) + x_3 + y_{1,1} + y_{1,2} & = 3 \\ \vdots & \vdots & \\ Q_n & \cdots & = 3 \end{array}$$

On ajoute pour chaque clause $C_i$ deux variables $y_{i,1}, y_{i,2}$ vérifiant $\begin{cases} y_{i,1}^2 - y_{i,1} = 0 \\ y_{i,2}^2 - y_{i,2} = 0 \end{cases}$

**Théorème 4.18 (Hilbert's Nullstellensatz 1893)**

Le système $\{P_i(x) = 0, i \in [n]\}$ n'a pas de solution dans $\mathbb{C}$ si et seulement si $\exists Q_1, \cdots, Q_n \in \mathbb{C}[x_1, \cdots, x_n]$ tels que $\sum_{i=1}^{n} P_i Q_i = 1$

Ceci n'est **pas** un certificat **coNP** car un polynôme $Q_i$ peut avoir une complexité en espace exponentielle.

# Chapitre 5

# *Programmation dynamique*

**Idée :** définir l'optimalité d'un problème à l'aide d'un nombre polynomial de sous problèmes.

## 5.1 Plus longue suite commune

PLSC

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

**Sous-problèmes :** on note $m_{ij}$ le plus long sous-mot commun de $A[1..i]$ et $B[1..j]$. On cherche $m_{n,m}$

**Init :** $m_{0,j} = 0 \ \forall j \in [m]$, $m_{i,0} = 0 \ \forall i \in [n]$

**Réc :**

- Si $A[i] \neq B[j]$, $m_{i,j} = \max(m_{i-1,j}, m_{i,j-1})$
- Si $A[i] = B[j]$, $m_{i,j} = 1 + m_{i-1,j-1}$

On peut coder en bottom-up (avec 2 boucles) ou avec une technique top-down, mais il faut éviter les appels multiples sur les sous problèmes en mémoïsant les résultats déjà calculés.

## 5.2 Produit en chaîne de matrices

**Remarque :** Le produit (naïf) de $A$ de taille $a \times b$ avec $B$ de taille $b \times c$ se fait en $abc$ opérations.

> **Exemple**
>
> Soit $A$ de taille $2 \times 4$, $B$ de taille $4 \times 6$ et $C$ de taille $6 \times 8$.
> $(AB)C \to 2 \times 4 \times 5 + 2 \times 6 \times 8 = 144$
> $A(BC) \to 4 \times 6 \times 8 + 2 \times 4 \times 8 = 256$

PCM

- Entrée : Suite de matrices $M_1, \cdots, M_n$ valides

- Sortie : Le produit effectué avec le moins d'opérations possibles

On note $l_i$ le nombre de lignes de $M_i$ et $l_{n+1}$ le nombre de colonnes de $M_n$.
On utilise la prog dyn avec les sous-problèmes $m_{ij}$ := nombre minimal d'opérations pour calculer $M_i M_{i+1} \cdots M_j$

**Init :** $m_{ii} = 0$

**Réc :** $m_{ij} = \min_{i \leq k < j}(m_{i,k} + m_{k+1,j} + l_i \cdot l_{k+1} \cdot l_{j+1})$

**Complexité :**

- nombre de sous-problèmes : $O(n^2)$
- calcul d'un sous-problème en $O(n)$

$\rightarrow O(n^3)$
Il existe un algo en $O(n \log n)$ (mais un article "faux")

On peut cependant faire mieux que l'algorithme de programmation dynamique. En effet le problème est semblable à un problème de parenthésage qui est lui-même similaire au problème de triangulation d'un n-gone.

**Réduction :** Comment trianguler un n-gone pondéré en minimisant la somme des triangles (chacun étant le produit de ses éléments)

Idée : on cherche les deux plus petits $v_1, v_2$ et on les relie.
Fait : une solution optimale relie $v_1$ à $v_2$ $\rightarrow$ induction ?
Fait : il existe toujours au moins 2 éléments isolés dans la triangulation (de degré 2)

Essai : $v_1$ relié à $v_2$ et $v_3$ $\rightarrow$ passe avec induction

Algorithme :

- si $v_1v_2$ ou $v_1v_3$ n'est pas un arête du n-gone $\rightarrow$ appel sur les deux sous n-gones
- sinon, retourner min [[FAIRE FIGURE (25/11)]]

$\rightarrow$ cet algorithme est en $\Theta(n^2)$

## 5.3 Autres problèmes

1. Sac à dos
2. Carré de 1 dans une matrice $n \times n$
3. Ensemble indépendants max dans un arbre

### 5.3.1 Sac à dos

Sac à dos

- Entrée : Couple $(p_1, v_1, \cdots, p_n, v_n)$, volume total $V$
- Sortie : Ensemble de poids maximum

On suppose le problème codé en unaire (en binaire c'est NP-difficile de calculer une solution optimale)

On propose un algorithme en $\Theta(n \cdot v)$

Sous-problèmes :

- $F(i, w)$ ::= solution optimale en utilisant des objets $O_1, \cdots, O_i$ si on limite le volume à $w$
- $F(i+1, w) = \max(F(i, w), p_{i+1} + F(i, w - v_{i+1}))$

### 5.3.2 Carré de 1

Carré de 1

- Entrée : Matrice $n \times n$ de $0, 1$
- Sortie : Plus grand carré contigüe de 1 (pas contigüe est NP-complet)

Sous-problèmes :

- $S(i,j)$ ::= plus grand carré de coin nord $(i,j)$
- $S(i,j) = \begin{cases} 0 \text{ si } M_{i,j} = 0 \\ 1 + \min(S(i-1,j), S(i-1,j-1), S(i,j-1)) \end{cases}$

L'algorithme est en $\Theta(n^2)$

### 5.3.3 Indépendant de poids maximal dans un arbre

Indépendant de poids maximal dans un arbre

- Entrée : Arbre sur un ensemble de noeuds $V$, chaque noeud $v$ a un poids $p(v) \geqslant 0$
- Sortie : $I \subseteq V$, deux à deux non adjacents et tels que $p(I)$ est maximum

Algorithme de programmation dynamique :

1. Enraciner l'arbre
2. Pour chaque noeud $v$ on calcule
   - $a(v)$ ::= la valeur de poids max d'un indépendant de $B_v$ contenant $v$
   - $b(v)$ ::= la valeur de poids max d'un indépendant de $B_v$ ne contenant pas $v$
3. Retourner $\max(a(r), b(r))$

**Initialisation :** si $v$ feuille : $\begin{cases} a(v) := p(v) \\ b(v) := 0 \end{cases}$

**Induction :**

- $a(v) := p(v) + \sum_{w \text{ enfant de } v} b(w)$
- $b(v) := \sum_{w \text{ enfant de } v} \max(a(w), b(w))$

# Chapitre 6

# *Analyse amortie*

**Exemple (Tableau dynamique)**

- On initialise un tableau $T$ de taille 1
- On écrit un nouvel élément $\to T[1]$
- On écrit un nouvel élément, pas de place, on double la taille de $T$ et on réécrit $T[1], T[2]$
- On écrit un nouvel élément, pas de place, on double la taille de $T$ et ...
- On ajoute un 4e élément, il y a la place

**Problème :** Quel est le coût amorti des $C_i$ ?

$C_i = 1$ sauf si $i = 2^k + 1$ auquel cas $C_i = i$

$$\sum_{i=1}^{2^n+1} C_i = 2^n + 1 + \sum_{j=0}^{n} 2^j = 2^n + 1 + 2^{n+1} - 1 = 3 \cdot 2^n$$

On a donc un coût amorti d'environ $\dfrac{3 \cdot 2^n}{2^n + 1} \simeq 3$

**Acomptes :** Pourquoi le coût est-il de 3 par opération ?

Chaque élément paye 1 pour s'inscrire, puis 1 pour sa duplication suivante. Si $x = i$, on paye pour la duplication de $j = i - 2^{\lfloor \log_2 i \rfloor - 1}$

**Potentiel :** Trouver une fonction potentiel $\varphi_i$ telle que le coût amorti $\widehat{C_i} = C_i + (\varphi_i - \varphi_{i-1}) \leqslant 3$

On aura ainsi que $\sum_{i=1}^{k} \widehat{C_i} = \sum_{i=1}^{k} C_i + \varphi_k - \varphi_0$, d'où $C_k = 3k - \varphi_k + \varphi_0$

On choisit $\varphi_i = 2i - n$ avec $n$ la taille du tableau dans lequel $i$ est inséré

$\varphi_i \geqslant 0$ car un élément est toujours écrit dans la deuxième moitié du tableau.

Si $i \neq 2^k + 1$, $\widehat{C_i} = 1 + (2i - n) - (2(i-1) - n) = 3$

Si $i = 2^k + 1$, $\widehat{C_i} = 2^k + 1 + 2(2^k + 1) - 2^{k+1} - (2 \cdot 2^k - 2^k) = 3$

**A voir**

Codes sur $\{0,1\}^k$ : code de Gray et graphe de Debroign

# Chapitre 7

# *Approximation*

## 7.1 Introduction

On considère ic des problèmes d'optimisation où la solution doit maximiser ou minimiser une certaine fonction objectif.
On notera en général OPT la valeur optimale.

On dit qu'un problème est NP-dur ou NP-difficile si son problème de décision associé (e.g. OPT $\geqslant k$ ?) est NP-complet.

Les problèmes d'optimisation sont souvent difficiles (voyageur de commerce, sac à dos), on aimerait trouver une solution approchée.

**Définition 7.1 ($c$–approximation)**

On se fixe $c > 1$. On dira qu'un algorithme (polynomial) est une *$c$–approximation* si :

- Pour un problème de maximisation, il retourne une solution de valeur au moins $\frac{OPT}{c}$
- Pour un problème de minimisation, il retourne une solution de valeur au plus $c \cdot OPT$

**Définition 7.2 (classe APX)**

S'il existe une valeur $c > 1$ telle qu'il existe un algorithme polynomial de $c$–approximation, on dit que le problème est dans APX.

S'il existe une valeur $c > 1$ telle que l'existence d'un algorithme de $c$–approximation implique **P** = **NP** , alors le problème est APX-dur.

**Définition 7.3 (schéma d'approximation polynomial)**

On parle de *schéma d'approximation polynomial* si pour tout $\varepsilon > 0$, il existe un algorithme de $(1+\varepsilon)$–approximation polynomial, 3 cas possibles :

- PTAS : algorithme en $O(n^{f(\frac{1}{\varepsilon})})$
- Efficient PTAS : algorithme en $O(f(\frac{1}{\varepsilon}) \cdot n^c)$
- Fully PTAS : algorithme polynomial $(\frac{1}{\varepsilon}, n)$

## 7.2 Illustration sur le problème du sac à dos

$O = \{O_1, \cdots, O_n\}$, $V$ volume max
Chaque $O_i$ a un prix $p_i$ et un volume $v_i$

On veut $I \subseteq [n]$ tel que $\sum_{i \in I} v_i \leqslant V$ et qui maximise $\sum_{i \in I} p_i$

NP-dur en codage binaire, polynomial en unaire (programmation dynamique)

**Hypothèse :** on a $v_i \leqslant V \ \forall i \in [n]$

### 7.2.1 Algorithme de 2-approximation

On a intérêt à choisir les objets de meilleure qualité $q_i = \frac{p_i}{v_i}$ (pour simplifier on suppose que les $q_i$ sont $\neq$ deux à deux)

**Idée :** On trie les $O_i$ par ordre décroissant de qualité (on suppose que l'ordre donné est celui-là). On fait un algorithme glouton : $\overbrace{\underbrace{O_1 \cdots O_t}_{GLOUTON} \ O_{t+1}}^{\overline{GLOUTON}} \cdots O_n$ (triés) et on retourne le max entre $\{O_1, \cdots, O_t\}$ et $\{O_{t+1}\}$

**Terminologie**
On appelle ça la Programmation Linéaire en Nombres Entiers (PLNE)

Pour le voir (et cela permet "l'intuition" de la notion de qualité, on adopte un point de vue fractionnaire (nouveau paradigme))

On introduit des variables $x_1, \cdots, x_n$ qui représente la fraction de l'objet $O_i$ sélectionnée dans une solution.

Sac à dos en relaxation fractionnaire devient : maximiser $\sum_{i=1}^{n} p_i x_i$ sous $\sum_{i=1}^{n} v_i x_i \leqslant V$ (avec $0 \leqslant x_i \leqslant 1$)

ATTENTION : résoudre ce problème est polytime. En ajoutant des contraintes $x_i \in \{0, 1\}$ au lieu de $x_i \in [0, 1]$, le problème devient NP-dur, mais on montre que le problème de départ est contenu dans cette relaxation.

On note $OPT$ la solution optimale de sac à dos, $p^*$ la solution optimale de sac à dos fractionnaire, alors $p \leqslant p^*$

**Fait :** la solution $(x)$ optimale vérifie :

- $x_1^* = x_2^* = \cdots = x_k^* = 1$
- $x_{k+1}^* \in [0, 1]$
- $\forall i > k+1, \ x_i^* = 0$

$p \leqslant p^* \leqslant \sum_{i=1}^{k} p_i + p_{k+1}$

Montrons maintenant qu'on a bien une 2-approximation, on va faire deux hypothèses de simplification :

- tous les $v_i$ sont $\leqslant v$
- tous les $q_i$ sont différents deux à deux

**Lemme 7.4**
Soit $i < j$, si $x_j^* > 0$ alors $x_i^* = 1$

**Démonstration**
Sinon, on peut transférer une partie de $x_j^*$ sur $x_i^*$.
Supposons par l'absurde que $x_i^* > 0$ et $x_i^* < 1$, choisissons $\varepsilon_i \leqslant 1 - x_i^*$ et $\varepsilon_j \leqslant x_j^*$ tels que $v_i\varepsilon_i = v_j\varepsilon_j$.
On ajoute $\varepsilon_i$ à $x_i^*$, on enlève $\varepsilon_j$ à $x_j^*$, on trouve une nouvelle solutions (valide par construction des valeurs) de prix total $p = \varepsilon_i p_i - \varepsilon_j p_j = p + \varepsilon_i p_i - \dfrac{v_i\varepsilon_i}{v_j} p_j = p + \varepsilon_i v_i \left(\dfrac{p_i}{v_i} - \dfrac{p_j}{v_j}\right)$ → contradiction
□

On a alors $p(\overline{GLOUTON}) \geqslant p^* \geqslant p$ donc $p(GLOUTON) + p(O_{t+1}) \geqslant p$ or l'un de $p(GLOUTON)$ et $p(O_{t+1})$ est $\geqslant \frac{p}{2}$

On a donc bien une 2-approximation

### 7.2.2 PTAS

On vient, pour tout $\varepsilon > 0$, construire en temps polynomial une solution de prix au moins $(1-\varepsilon)p$

**Principe de dualité :** On se base sur l'objectif aussi bien que sur les contraintes (ici le prix)

**Lemme 7.5**
Si l'élément $O_{t+1}$ a un prix $p_{t+1} \leqslant \varepsilon p$ alors $GLOUTON$ a $prix \geqslant (1-\varepsilon)p$

**Démonstration**
Le nombre d'objets $O_i$ de prix $> \varepsilon p$ est au plus de $\frac{1}{\varepsilon}$ dans OPT. Notons $O' \subseteq O$ l'ensemble des $O_i$ de prix $> \varepsilon p$

On a alors l'algorithme GUESS & GLOUTON (cf. dessous).
**Complexité :** en $O(n^{\frac{1}{\varepsilon}})\cdot$ complexité(GLOUTON)

**Problème :** on ne connait pas $p$.

**Solutions :**

- On calculer $p'$, le prix de glouton $p' \geqslant \frac{p}{2}$
- Au lieu du $O'$ précédent, on considère tous les $O_i$ de prix $> \varepsilon p'$. On considère alors tous les sous ensembles de taille $\leqslant \dfrac{2}{\varepsilon}$

→ inefficace mais ouvre la voie □

**Algorithme 10** Algo GUESS & GLOUTON

**pour** tout sous ensemble $X \subseteq O'$ de taille $\leqslant \dfrac{1}{\varepsilon}$ **faire**
  Mettre $X$ dans une solution $S_X$
  Compléter $S_X$ de manière gloutonne sur $O \setminus O'$
**fin pour**
**retourner** le meilleur $S_X$

### 7.2.3 FPTAS

Reprenons l'algorithme de programmation dynamique de SAD, mais en se basant sur les prix plutôt que sur les valeurs.

Soit $S(i,p) :=$ volume minimal d'une solution de prix $p$ incluse dans $\{O_1, \cdots, O_i\}$ ($-\infty$ si pas de solution)

**Conditions initiales :**

- $\forall i \in [n],\ S(i,0) = 0$
- si $p < 0,\ S(0,p) = -\infty$

**Induction :** $S(i,p) = \min(S(i-1,p), S(i-1,p-p_i)+v_i)$ → retourner le max $p$ tel que $S(n,p) \leqslant v$

**Complexité :** $O\left(n \cdot \left(\sum_{i=1}^{n} p_i\right)\right)$

**Idée :** comme on cherche une solution approchée, on a (peut-être) pas besoin de toutes la précision des $p_i$

Considérons $\alpha > 0$ petit tel que l'algorithme de programmation dynamique appliqué aux valeurs $p'_i = \lfloor \alpha p_i \rfloor$ retourne une solution OPT' telle que $p(OPT') \geqslant (1-\varepsilon)p$ → quelle contraintes sur $\alpha$ ?

**Remarque :** $\alpha p_i - 1 \leqslant p'_i \leqslant \alpha p_i$ donc $p(OPT') = \sum_{i \in OPT'} p_i \leqslant \frac{1}{\alpha} \sum_{i \in OPT'} p'_i \geqslant \frac{1}{\alpha} \sum_{i \in OPT} p'_i \geqslant \frac{1}{\alpha} \sum_{i \in OPT} \alpha p_i - 1 \geqslant p - \frac{|OPT|}{\alpha} \geqslant p - \frac{n}{\alpha}$

Pour avoir une solution $\geqslant (1-\varepsilon)p$, il faut que $\frac{n}{\alpha} \leqslant \varepsilon p$. Il suffit de choisir $p'$ au lieu de p → on pose $\alpha = \frac{n}{\varepsilon p'}$

**Remarque :** On peut aussi choisir $p_{\max}$ au lieu de $p'$ où $p_{\max} = \max p_i$

La complexité est alors en $O\left(n \cdot \sum_{i=1}^{n} p'_i\right) = O\left(\frac{n^3}{\varepsilon}\right)$

En effet $\sum_{i=1}^{n} p'_i \leqslant \alpha_i \sum_{i=1}^{n} p_i = \frac{n}{\varepsilon p} \sum_{i=1}^{n} p_i = \frac{n}{\varepsilon} \sum_{i=1}^{n} \frac{p_i}{p'} = O\left(\frac{n^2}{\varepsilon}\right)$

Si $p'$ est une fraction positive de $\sum p_i$ on a $O(\frac{n^2}{\varepsilon})$

## 7.3 Transversaux de poids minimal (aka Min Hitting Set)

K-TRANSVERSAL

- Entrée : $E \subseteq \binom{[n]}{k}$ ensemble des parties de taille $k$ et $\omega : [n] \to \mathbb{R}^+$
- Sortie : $T \subseteq [n]$ tel que $T \cap e \neq \varnothing$ pour tout $e \in E$ et $\omega(T)$ minimum

> **Exemple**
> $k = 2$ et $\omega = 1$ : c'est le problème de minimisation vertex cover → OPT a valeur 2

### 7.3.1 Cas de poids égal à 1

Lorsque $\omega = 1$, on a un algorithme de $k$−approximation trivial :

1. construire (GLOUTON) des éléments de $E$ deux à deux disjoints $e_1, \cdots, e_t$ tels que $\forall e \in E, \exists i,\ e \cap e_i \neq \varnothing$
2. retourner $\bigcup_{i=1}^{t} e_i =: S$

**Remarque :** On a forcément $|OPT| \leqslant t$ (car il doit toucher tous les $e_i$, et la solution $S$ vérifie que $|S| = t \cdot k \leqslant k\,|OPT|$) → $k$−approximation

> **Théorème 7.6**
> A moins que **P** = **NP**, il n'existe pas de $(\sqrt{2}-\varepsilon)$−approximation pour vertex cover.

**Théorème 7.7**
A moins que la "Unique Gamma Conjecture" soit fausse, il n'existe pas de $(2-\varepsilon)$–approximation pour vertex cover.

Comment $k$–approximer avec $\omega$ quelconque ?

### 7.3.2 Arrondi en relaxation fractionnaire

On considère la relaxation $(p)$ :

- $x_i$ := fraction de l'élément $i$ choisi dans $T$
- Minimiser $\sum_{i=1}^{n} \omega_i x_i$
- Sous les contraintes $\sum_{i \in e} x_i \geqslant 1 \ \forall e \in E$ et $x_i \geqslant 0 \ \forall i \in [n]$

Notons $OPT^* = (x_i^*)$ une solution optimale.

**Remarque :** La valeur de $\omega(OPT)$ est $\geqslant \sum_{i=1}^{n} \omega_i x_i^*$ car OPT est un élément du domaine de $(p)$ (en posant $x_i = 1$ si $i \in OPT$, 0 sinon)

On calculer une solution $(x_i^*)$.
On construit une solution 0/1 $(OPT')$ comme suit :

- $x'_i = 1$ si $x_i^* \geqslant \frac{1}{k}$
- $x'_i = 0$ sinon

**Fait :** $\omega(OPT') \leqslant k\omega(OPT^*) \leqslant k\omega(OPT)$ donc $OPT'$ est une $k$–approximation

On a deux problèmes :

- on doit résoudre $(p)$
- si $k = 10$, la taille de $E$ est (peut-être) $\Theta(n^10)$. Pourrait-on faire un algorithme de $k$–approximation avec seulement un oracle pour $E$ ?

Oracle

- Entrée : $T$ (transversal potentiel)
- Sortie : **VRAI** si $T$ est transversal, **FAUX** et $e \in E$ tel que $e \cap T = \varnothing$ sinon

### 7.3.3 $k$–approximation avec arrondi

A présent, $E$ est non visible : on dispose juste d'un oracle, qui recevant en entrée un (possible) $T$ retourne :

- **VRAI** si $T$ transversal
- **FAUX et** $e \in E \mid e \cap T = \varnothing$

**Remarques :**

▷ A la terminaison, $T$ est bien un transversal

**Algorithme 11** Primal Dual

1: Initialiser $T = \varnothing$ et $y : E \to \mathbb{N}$ telle que $y = 0$
2: **tant que** $\exists e \in E$ tel que $e \cap T = \varnothing$ **faire**
3: Augmenter $y(e)$ jusqu'à atteindre pour un certain $i \in e$ l'égalité $\sum_{e \ni i} y(e) = w(i)$
4: $T \leftarrow T \cup \{i\}$
5: **fin tant que**
6: **retourner** $T$

▷ Calculons à présent $\omega(T) = \sum_{i \in T} \omega(i) = \sum_{i \in T} \sum_{e \ni i} y(e) = \sum_{e \in E} |e \cap T| \cdot y(e) \leqslant k \cdot \sum_{e \in E} y(e)$

On montre maintenant que si $T_{opt}$ est la solution optimale de $k-$transversal, alors $\sum_{e \in E} y(e) \leqslant \omega(T_{opt})$ ; on aura alors $w(T) \leqslant K \cdot \omega(T_{opt})$ par conséquent $T$ est une $k-$approximation de $k-$transversal

$\omega(T_{opt}) = \sum_{i \in T_{opt}} \omega(i) \geqslant \sum_{i \in T_{opt}} \sum_{e \ni i} y(e) = \sum_{e \in E} |T_{opt} \cap e| \cdot y(e) \geqslant \sum_{e \in E} y(e)$

## 7.4 Compléments

Dans l'algorithme de Christofides, il est utilisé le fait que calculer un couplage de poids max (dans un graphe quelconque) est dans **P** [1]. On aurait juste besoin de pouvoir approximer → est-il possible d'approximer le couplage de taille maximum dans un graphe arbitraire ?

[1] Algorithme d'Edmond

OUI avec :

### 7.4.1 Paradigme : Recherche locale

**But :** Obtenir un couplage de taille $\frac{4}{5}OPT$

On part d'une solution :

- couplage vide $C$
- Tant que $\exists$ couplage $C'$ qui diffère de $C$ sur au plus 7 arêtes, et tel que $|C'| > |C|$
- $C \leftarrow C'$

**Fait :** à la fin, $C$ est une $\frac{4}{5}-$approximation de $OPT$
Noter que $OPT \cup C$ est une union de cycles et de chemins
Si on avait $|C| < \frac{4}{5}|OPT|$, il existe un cycle ou un chemin où cette inégalité est vérifiée

- Cycle : impossible car alternant
- Chemin : possible seulement s'il alterne $(OPT, C, OPT, C, \cdots)$

  Pour que $|C| < \frac{4}{5}|OPT|$, on doit avoir que la longueur du chemin est au plus 7, échanger les arêtes de $C$ sur ce chemin pour les arêtes de $OPT$ fournit une meilleure solution.

### 7.4.2 Autres algorithmes de recherche locale

- MaxCut 2−approx (TD)
- Descente de Gradient [2]
- Algorithme du simplexe [3]
- Prochain TD : résoudre HopField

[2] M1

[3] M1

### 7.4.3 Paradigme : Relaxation fractionnaire

- Penser à la relaxation fractionnaire → une fois faite : dualiser.

**Dualiser :**
Partant d'un problème $P$, on écrit un programme linéaire appelé primal, $(P)$ : maximiser $C^T \cdot x$ sous $A \cdot x \leqslant b$ et $x \geqslant 0$

**Théorème 7.8 (Dualité)**

$(P)$ a une solution optimale de valeur $v$ si et seulement si $(D)$ [4] a une solution optimale de valeur $v$ où $(D)$ : minimiser $b^T y$ sous $A^T y \geqslant c$ et $y \geqslant 0$

[4] Dual

Les problèmes viennent pas deux, le primal et le dual.

**Exemple (couplage dans un graphe)**

$x$ est une distribution de poids sur les arêtes. La relaxation de couplage est de trouver une telle distribution qui vérifie que le poids des arêtes incidentes à un sommet est au plus 1.

$(P)$ : maximiser $1 \cdot x$ sous $A \cdot x \leqslant 1$ et $x \geqslant 0$
où $A$ est la matrice d'incidence, en colonne les arêtes, et en ligne les sommets.

$(D)$ : minimiser $1 \cdot y$ sous $A^T \cdot y \geqslant 1$ et $y \geqslant 0$
→ $y$ est une distribution de poids sur les sommets telle que pour chaque arête $uv$ on a $y(u) + y(v) \geqslant 1$

C'est la relaxation fractionnaire de vertex cover

# Chapitre 8

# *Paradigme : Algorithmes randomisés*

## 8.1 Calcul d'un couplage parfait

Soit $(A, B)$ un graphe biparti. Décider s'il existe un couplage parfait est dans **P**.

On veut un algorithme qui :

- s'il retourne **VRAI** : il existe un couplage parfait
- s'il retourne **FAUX** , avec probabilité 1/2, il n'existe pas de couplage parfait

Soit $M$ la matrice de $A, B$, $m_{ab} = \begin{cases} 1 \text{ si } a - b \\ 0 \text{ sinon} \end{cases}$

On peut supposer $A = B = [n]$

Un couplage parfait est équivalent à l'existence d'une permutation $\sigma$ vérifiant $\forall i \in [n]$, on a $m_{i,\sigma(i)} = 1$. C'est équivalent à $\exists$ permutation $\sigma$ telle que $\prod_{i \in [n]} m_{i,\sigma(i)} = 1$

**Rappel :** $\det(M) = \sum_{\sigma \in S_n} (-1)^{\varepsilon(\sigma)} \cdot \prod_{i \in [n]} m_{i,\sigma(i)}$

$Perm(M) = \sum_{\sigma} \prod_{i \in [n]} m_{i,\sigma(i)}$ est le *permutant* de $M$ et calcule le nombre de couplages parfaits

Valiant a montré que $Perm$ est $\sharp$**P**−complet → dur à calculer

**Idée :**

- On calculer $\det(M)$ et si $\neq 0$ alors retourner **VRAI**
- En revanche, on peut avoir $\det(M) = 0$ par simplification de termes

**Algorithme 12** Tutte

1: Remplacer chaque arête par une valeur aléatoire entre 1 et $2n \to M$
2: **retourner VRAI** si $\det(M) \neq 0$, **FAUX** sinon

Avec probabilité 1/2, s'il existe un couplage parfait alors $\det(M) \neq 0$

**Remarques :**

▷ Cela marche pour les graphes généraux : étant donné un graphe $G$, poser $M$ comme sa matrice d'adjacence

Pour $i, j$ : remplacer $m_{i,j}$ par $C$, $m_{j,i}$ par $-C$ avec $C$ tiré au hasard entre 1 et $2n$

Même conclusion

▷ Ce qui sous-tend Tutte :

Si $P$ est un polynôme multivarié $\in \mathbb{R}[x_1, \cdots, x_n]$ de degré $d$ et non identiquement nul, alors si on tire les valeurs des $x_i$ aléatoirement dans $\{1, \cdots, 2d\}$ alors avec probabilité $\geqslant 1/2$ l'évaluation est $\neq 0 \rightarrow$ application directe du déterminant

# Chapitre 9

# *Algorithmes exponentiels exacts*

**But :** résoudre des problèmes **NP** -complet pour des petites instances, trouver $c$ le plus petit possible pour une complexité de $c^n$

**Notation**
$O^*(c^n) = O(c^n \cdot \text{poly}(n))$

## 9.1 Meet in the middle

Somme

- Entrée : $T = \{t_1, \cdots, t_n\}, S$ entier
- Sortie : $I \subseteq [n] \mid \sum_{i \in I} t_i = S$ ou **FAUX**

**Brute force** : $2^n$

Peut-on faire $\sqrt{2^n} = 2^{n/2}$ ?

- Partitionner $T$ en $T_1$ et $T_2$ de taille $\frac{n}{2}$
- Calculer toutes les sommes partielles de $T_1$ et $T_2 \to P_1$ et $P_2$ (coût $O(2^{n/2})$)
- Trier $P_1 = \{x_1, \cdots, x_p\}$ et $P_2 = \{y_1, \cdots, y_q\}$ (à faire lors de la construction)
- On fait une recherche par approche double pointeur : $a \leftarrow 1$ et $b \leftarrow q$ au début et on itère si $x_a + y_b < S$ alors $a \leftarrow a + 1$ sinon $b \leftarrow b - 1$ jusqu'à trouver ou non $S$.

$\to$ la complexité finale est $O^*(\sqrt{2^n})$

## 9.2 3-SAT avec affectation partielle

3-SAT

- Entrée : formule 3-SAT $F$ en variable $x_1, \cdots, x_n$ avec des clauses à 1, 2 ou 3 littéraux
- Sortie : affectation satisfaisant $F$ si une existe, **FAUX** sinon

On va construire une affectation partielle et à chaque étape :

- $R_1$ : si une clause de $F$ possède un littéraux $V$, on la supprime
- $R_2$ : si une clause ne possède que des littéraux $F$, on retourne **FAUX**

- $R_3$ : si une clause possède un littéral $F$, on supprime ce littéral
- $R_4$ : si une clause possède 1 littéral, on affecte $V$

`Alg1`$(F, a)$ :
Si $\exists$ clause $(l_1 \vee l_2)$ dans $F$ : retourner `Alg1`$(F, a \cup \{l_1 = V, l_2 = V\}) \vee$ `Alg1`$(F, a \cup \{l_1 = V, l_2 = F\}) \vee$ `Alg1`$(F, a \cup \{l_1 = F, l_2 = V\})$
Sinon $(l_1 \vee l_2 \vee l_3)$ dans $F$ : retourner $\vee$ des 7 possibilités

Complexité du premier return : $T(n) = 3T(n-2)$
Complexité du second return : $T(n) = 7T(n-3)$

**Hyp :** $T(n) = c^n$, alors $c^n = 3c^{n-2} \to c^2 = 3 \rightsquigarrow c \simeq 1.73$ et $c^n = 7c^{n-3} \rightsquigarrow c \simeq 1.91$

Donc dans le pire cas $O^*(1.91^n)$

Mais on peut faire mieux :

`Alg2`$(F, a)$
si $(l_1 \vee l_2) \in F$ : ...
si $(l_1 \vee l_2 \vee l_3) \in F$ : retourner `Alg2`$(F, a \cup \{l_1 = V\}) \vee$ `Alg2`$(F, a \cup \{l_1 = F, l_2 = V\}) \vee$ `Alg2`$(F, a \cup \{l_1 = F, l_2 = F, l_3 = V\}$

$T(n) = T(n-1) + T(n-2) + T(n-3)$ donc $c^n = c^{n-1} + c^{n-2} + c^{n-3}$ donc $c^3 = c^2 + c + 1$ et $O^*(1.84^n)$

Pour continuer : Monien-Speckenmeyer

1. affectation $a$ "autarcique" si toutes les clauses qui intersectent $a$ sont positives
2. si $a$ pas autarcique, une clause au moins est réduite et donc devient de taille 2 → branchement sur les clauses de taille 2
   $c^n = c^{n-1} + c^{n-2}$ donne $O^*(1.618^n)$

## 9.3 3-SAT RECHERCHE LOCALE

**Notation**
$a \oplus (l_i = F)$ : on modifie $a$ en posant $l_i = F$

**But :** on part d'une affectation $a$ totale et la modifier
`HAM`$(F, a, h)$
si $a$ satisfait $F$ : retourner **VRAI** (ou $a$)
si $h = 0$ : retourner **FAUX**
sinon $\exists (l_1 \vee l_2 \vee l_3)$ non satisfaite dans $F$ : retourner `HAM`$(F, a \oplus (l_1 = V), h-1) \vee$ `HAM`$(F, a \oplus (l_1 = F, l_2 = V), h-1) \vee$ `HAM`$(F, a \oplus (l_1 = F, l_2 = F, l_3 = V), h-1)$

**Remarque :** `HAM`$(F, a =$ `TRUE`$, h = n)$ va résoudre 3-SAT sur $F$.
La distance de Hamming entre $a$ et une (possible) affectation qui satisfait $F$ décroît de 1 dans l'une des branches → correct et $O^*(3^n)$

On peut aussi `HAM`$(F, a =$ `TRUE`$, h = n/2) \vee$ `HAM`$(F, a =$ `FALSE`$, h = n/2)$ et donc on a une complexité $O^*(3^{n/2}) = O^*(1.73^n)$

Généraliser : $s$ points qui couvrent avec leur $t$-voisinages tout cube de dimension $n$ ("code correcteur like") → $O^*(s \cdot 3^t)$

Essai : $t = \frac{n}{4}$ : combien faut-il de points sur l'hypercube de dimension $n$ pour que leur voisinage à distance $\frac{n}{4}$ couvre tout ?

borne inf : (volume) au moins $2^n$ [1] donc la borne inf est $\frac{2^n}{2^{H(1/4)\cdot n}} = 2^{(1-H(1/4)\cdot n)}$

- bonne nouvelle, il suffit de choisir $2^{(1-H(1/4)\cdot n)} \cdot \text{poly}(n)$ points aléatoires et $s = O^*(2^{(1-H(1/4)\cdot n)})$ pour un branchement de $3^{n/4}$

$H(p) = -p \log_2 p - (1-p)\log_2(1-p)$ donc $H(1/4) = 2 - \frac{3}{4}\log_2 3$

**complexité :** $2^{(1-H(1/4)\cdot n)} \cdot 3^{n/4} = O^*(1.5^n)$

[1] "taille du voisinage" $\simeq 2^{H(\frac{1}{4})\cdot n} \simeq O^*(\binom{n}{\frac{n}{4}}) \simeq 1 + n + \binom{n}{2} + \binom{n}{3} + \cdots + \binom{n}{\frac{n}{4}}$

## 9.4 Nombre chromatique d'un graphe

Soit $G = (V, E)$ un graphe, le nombre chromatique de $G$ noté $\chi(G)$ est le plus petit entier $k$ tel que $G$ soit $k$-coloriable, ou de manière équivalente le nombre minimum d'ensembles indépendants qui couvrent $V$.

**Problème :** comment décider (par exemple) si $\chi(G) \leqslant \sqrt{n}$ ? → brute force en $O^*(\sqrt{n}^n)$

**Théorème 9.1 (Björklund-Husfield-Koiristo (2006))**

On peut calculer $\chi(G)$ en $O^*(2^n)$

$c_k(G)$ = nombre de couverture de $V(G)$ par $I_1, \cdots I_k$ qui sont des indépendants de $G$.
Pour $X \subseteq V$, on note $s(X)$ = nombre d'indépendants disjoints de $X$

**Théorème 9.2**

$$c_k(G) = \sum_{X \subseteq V} (-1)^{|X|} \cdot s(X)^k$$

**Démonstration**

contribution d'un terme fixé dans $c_k(G)$ $I_1, \cdots, I_k$
→ si $I_1 \cup \cdots \cup I_k = V$ la contribution du terme est uniquement pour $X = \varnothing$ : contribution $= 1$
→ si $I_1 \cup \cdots \cup I_k = V \setminus Y$, la contribution est $= 0$. Il existe autant de parties $X \subseteq Y$ paires que impaires □

calcul de $s(X)$ en programmation dynamique :

- $s(V) = 1$
- $\forall X, \forall v \notin X, s(X) = S(X \cup \{v\}) + S(X \cup \{v\} \cup N(v))$ où $N(v)$ est le voisinage de $v$

On peut calculer $s(X)$ en $O^*(2^n)$

# Index