Complexité

Philippe Quéinnec

http://queinnec.perso.enseeiht.fr/Ens/calc.html

1er février 2021

Plan

(certains transparents sont issus du cours d'Adam Shimi 2019)

Ressources

Ce cours est inspiré des cours suivants :

- *Introduction à la calculabilité*, Pierre Wolper, Dunod, 2006
- *Langages formels, Calculabilité et Complexité*, Olivier Carton, éditions Vuibert, 2014
- *Calculabilité et complexité*, Anca Muscholl, 2018
  `http://www.labri.fr/perso/anca/MC.html`
- *Complexité algorithmique*, Sylvain Perifel, Ellipses, 2014
  `https://www.irif.fr/~sperifel/livre_complexite.html`
- *Fondements de l'informatique – Logique, modèles, et calculs*, Olivier Bournez, 2013–2020
  `https://www.enseignement.polytechnique.fr/informatique/INF412`
- *Computational Complexity : A Modern Approach*, Sanjeev Arora and Boaz Barak, Cambridge University Press, 2009
  (draft sur `http://theory.cs.princeton.edu/complexity/`)
- *Mathematics and Computation*, Avi Wigderson, 2019
  `https://www.math.ias.edu/avi/book`

Première partie

Introduction

Multiplier efficacement 577×423

- Par additions successives
  - $577 + 577 + 577 + 577 + \ldots$: nécessite 422 additions
  - multiplier deux nombres de n chiffres nécessite $O(n \cdot 10^n)$ additions
- par l'algorithme standard
  - $$\begin{array}{cccccc} & & & 5 & 7 & 7 \\ & & \times & 4 & 2 & 3 \\ \hline & & 1 & 7 & 3 & 1 \\ & 1 & 1 & 5 & 4 & \\ 2 & 3 & 0 & 8 & & \\ \hline 2 & 4 & 4 & 0 & 7 & 1 \end{array}$$: 3 additions et 3 multiplications élémentaires
  - multiplier deux nombres de n chiffres nécessite $O(n^2)$ opérations
- Par transformée de Fourier FFT (1971)
  $O(n \cdot \log n \cdot \log\log n)$ opérations (pour $n > 100000$)
- Algorithme de Harvey et van der Hoeven (2019)
  $O(n \cdot \log n)$ opérations (pour $n > 2^{1729^{12}}$)
- Peut-on faire mieux ? (on ne pense pas)

($O(f(n))$ signifie "dominé par $f(n)$", précisé plus loin)

Trier efficacement n entiers par comparaison 2 à 2

- *bogosort* : temps asymptotique $O(n!)$, espace $O(n)$
- Tri à bulles : temps $O(n^2)$, espace $O(1)$
- Tri par insertion : temps $O(n^2)$, espace $O(1)$,
- Tri rapide : temps en moyenne $O(n \log n)$, temps en pire cas $O(n^2)$, espace $O(\log n)$
- Tri rapide avec pivot aléatoire : espérance probabiliste $O(n \log n)$
- Tri fusion : temps $O(n \log n)$, espace $O(n)$
- *heap sort* : temps $O(n \log n)$, espace $O(1)$
- Peut-on faire mieux ? (on sait que non par comparaison)

Pigeonhole sort ou *counting sort*

- temps $O(n + k)$, espace $O(k)$ (où k est l'espace des clefs)

La question : efficacité et difficulté d'un problème

Problème vs algorithme

Difficulté d'un problème = efficacité du meilleur algorithme.

Quelles ressources mesurer ?

- le temps de calcul
- la mémoire utilisée
- l'énergie consommée
- l'aléatoire nécessaire
- la communication nécessaire

Pour quelle machine, quelle langage, quelle entrée ?

- Machine, langage : sans importance (équiv. à un polynôme près)
- Fonction de la taille de l'entrée, $\to \infty$
- Entrée : pire cas

Calculer vs vérifier (Cole, 1903)

- Trouver si 147573952589676412927 est premier est dur.
- Vérifier
  $2^{67} - 1 = 147573952589676412927 = 193707721 \times 761838257287$
  est facile (Cole, 1903).

$$\begin{array}{ccc} \text{Calculer} & & \text{Vérifier} \\ \text{en temps} & \stackrel{?}{=} & \text{en temps} \\ \text{polynomial} & & \text{polynomial} \end{array}$$

$$\mathbf{P} \stackrel{?}{=} \mathbf{NP}$$

(Projet de loi de programmation de la recherche pour les années 2021 à 2030, texte élaboré par la commission mixte paritaire, Assemblée nationale n°3533 & Sénat n°177, novembre 2020, p82 : trois *grandes questions ouvertes de la science* sont mises en avant, dont celle-ci)

Résumé

- **P** = classe des problèmes solubles en temps polynomial de la taille de l'entrée
- **NP** = classe des problèmes vérifiables en temps polynomial
- **NP**-complet = problèmes vérifiables en temps polynomial, pas solubles en temps polynomial (sans doute)
- **EXPTIME** = problèmes solubles en temps exponentiel
- **LSPACE** = problèmes nécessitant moins que $\log n$ espace
- **PSPACE** = problèmes nécessitant un espace polynomial
- **EXPSPACE** = problèmes nécessitant un espace exponentiel

Modèle de calcul considéré : les machines de Turing

Mesure de la complexité

Mesurer la complexité = ressources utilisées par une machine de Turing :

- Temps : nombre de pas nécessaire pour arriver à la solution
- Mémoire : nombre de cases nécessaires, en plus du mot en entrée
- Énergie : énergie = O(temps), toute transition de même coût
- Aléatoire : source externe de bits aléatoires

Asymptotiquement pour des entrées de taille $\to \infty$, et dans le pire cas.

(On verra que le modèle de calcul est sans importance)

Notations : Grand-O

Grand-O (*Big-Oh*) $f = O(g)$

f est dominée par g, f croît moins vite que g :

$$f(n) = O(g(n)) \triangleq \exists c \in \mathbb{R}, \exists n_0 \in \mathbb{N} : \forall n \geq n_0 : f(n) \leq c \cdot g(n)$$

On écrit $f = O(g)$ ou $f(n) = O(g(n))$.
L'usage de $=$ est abusif mais installé, il vaudrait mieux écrire $f \in O(g)$.

Petit-o $f = o(g)$

f croît beaucoup moins vite que g :

$$\begin{aligned} f(n) = o(g(n)) &\triangleq \forall \epsilon \in \mathbb{R}^{+*}, \exists n_0 \in \mathbb{N} : \forall n \geq n_0 : f(n) \leq \epsilon \cdot g(n) \\ &\triangleq \lim_{x \to \infty} \frac{f(n)}{g(n)} = 0 \end{aligned}$$

$n \log n = O(n^2), \quad 10n^2 + 24n = O(n^2), \quad n^3 = o(3^n)$

Deuxième partie

Complexité en temps

Plan

Importance de la complexité

Processeur effectuant 1 million d'instructions par seconde.

	Complexité						
taille	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 *s*	< 1 *s*	< 1 *s*	< 1 *s*	< 1 *s*	< 1 *s*	4 *s*
$n = 30$	< 1 *s*	< 1 *s*	< 1 *s*	< 1 *s*	< 1 *s*	18 *min*	10^{25} *ans*
$n = 50$	< 1 *s*	< 1 *s*	< 1 *s*	< 1 *s*	11 *min*	36 *ans*	∞
$n = 100$	< 1 *s*	< 1 *s*	< 1 *s*	< 1 *s*	$12,9$ *ans*	10^{17} *ans*	∞
$n = 1000$	< 1 *s*	< 1 *s*	1 *s*	18 *min*	∞	∞	∞
$n = 10000$	< 1 *s*	< 1 *s*	2 *min*	12 *jours*	∞	∞	∞
$n = 10^5$	< 1 *s*	2 *s*	3 *h*	32 *ans*	∞	∞	∞
$n = 10^6$	1 *s*	20 *s*	12 *jours*	31710 *ans*	∞	∞	∞

($\infty \geq 10^{25}$ans)

(Extrait de *Algorithm Design*, Kleinberg and Tardos, 2005)

Problème de décision

Problème de décision

Un problème de décision sur un alphabet Σ est un ensemble de mots sur Σ, ou alternativement, une fonction $f : \Sigma^* \to \{0, 1\}$

Problème de recherche

Un problème de recherche sur un alphabet Σ est une fonction $f : \Sigma^* \to \Sigma^*$

Exemple : soit G un graphe :

- Problème de décision : G est-il k-colorable ? (colorable avec k couleurs sans que les voisins partagent leur couleur)
- Problème de recherche / calcul : trouver une k-coloration de G.
- Problème d'optimisation : trouver une coloration pour le plus petit k tel que G est k-colorable.

Problème de recherche $\rightarrow$ problème de décision

On peut souvent transformer un problème de recherche en problème de décision de difficulté équivalente (mais pas toujours. . .)

- « Trouver le plus petit diviseur de N » $\rightarrow$ « Soit N et k, existe-t-il un diviseur de N qui soit inférieur k ? »
  Effectuer une recherche dichotomique $< N/2$? si oui, $< N/4$? Si oui, $< N/8$, si non $< 3N/8$ etc $\Rightarrow$ facteur $O(\log N)$
- « Calculer $f(x, y)$ » $\rightarrow$ « le i-ième bit de $f(x, y)$ est-il 1 ? »
  nombre de bits à calculer indépendant des entrées $\Rightarrow$ facteur$O(1)$
  nombre de bits à calculer $O(g)$ des entrées $\Rightarrow$ facteur $O(g)$

Plan

DTIME

DTIME

La classe $\mathrm{DTime}(f(n))$ est la classe des problèmes de décision décidé en $O(f(n))$ pas : un langage L appartient à la classe de complexité $\mathrm{DTime}(f(n)) \triangleq \exists M$ une machine de Turing :

- $\forall x \in L$: M accepte x.
- $\forall x \notin L$: M rejette x.
- $\forall x$: M prend $O(f(|x|))$ pas pour terminer.

(machine de Turing basique : déterministe, mono-ruban, mono-tête)

Fonction constructible en temps

Constructible en temps

Une fonction $f : \mathbb{N} \to \mathbb{N}$ est constructible en temps si $f(n) \geq n$ et s'il existe une machine de Turing qui calcule $f(n)$ en $O(f(n))$ pas.

- La restriction $f(n) \geq n$ est pour pouvoir lire l'entrée n. On pourrait aussi dire "en $O(n + f(n))$ pas".
- Toutes les fonctions usuelles pour la complexité (log, $+$, $\times$, exp...) sont constructibles en temps.
- Dans la suite, on ne considère que des fonctions constructibles en temps.

Concept pas vraiment important, mais il se passe des choses bizarres si l'on considère des classes de complexité de fonctions non constructibles en temps : elles apportent du temps gratuit.

Propriétés de DTIME

fermeture

Pour f constructible en temps, $\mathrm{DTIME}(f)$ est fermé par union, intersection et complémentaire :
Pour L_1, L_2 problèmes de décision dans $\mathrm{DTIME}(f)$,

- $L_1 \cup L_2 \in \mathrm{DTIME}(f)$
- $L_1 \cap L_2 \in \mathrm{DTIME}(f)$
- $\overline{L_1} \in \mathrm{DTIME}(f)$

- $\cup$: exécuter M_1 qui reconnaît L_1, si échec exécuter M_2 qui reconnaît L_2. Temps pire = somme des temps + constante pour le "si".
- $\cap$: exécuter M_1 qui reconnaît L_1, si ok exécuter M_2 qui reconnaît L_2. Temps pire = somme des temps + constante pour le "si".
- $\overline{L_1}$: inverser la décision.

Hiérarchie temporelle

Ordre partiel

Pour f, g constructibles en temps, $f = O(g) \Rightarrow \mathrm{DTIME}(f) \subseteq \mathrm{DTIME}(g)$

Ordre strict

Pour f, g constructibles en temps, $f = o(g) \Rightarrow \mathrm{DTIME}(f) \subsetneq \mathrm{DTIME}(g)$

Bizarrement, preuve pas triviale par diagonalisation, similaire à l'indécidabilité de l'arrêt.

La classe **P** = les calculs efficaces

Classe de complexité **P**

P = classe des problèmes décidables en temps polynomial :

$$\mathbf{P} \triangleq \bigcup_{c \in \mathbb{N}} \mathrm{DTime}(n^c)$$

Pourquoi polynomial pour les calculs efficaces ?

- Classe fermée par addition, multiplication et composition : résoudre un problème par agrégation d'autres problèmes conserve l'efficacité.
- Les polynômes ne croissent pas trop vite. La force brute est exponentielle et n'est pas dans **P**.
- Le modèle de calcul est sans importance : tous équivalents à un polynôme près *(à suivre)*.
- La représentation des données est sans importance *(à suivre)*.
- *Loi des petits nombres* : en pratique, on rencontre plutôt des problèmes en $O(n^2)$ ou $O(n^3)$ qu'en $O(n^{100})$.

Exemples de problèmes dans **P**

- Vérification de multiplication de matrices
- Test de primalité (2004)
  (naïf en $O(2^{\log n})$, algorithme probabiliste polynomial 1976)
- Existence d'un chemin entre deux nœuds d'un graphe
- Test de planarité d'un graphe
- Graphe eulériens : existence d'un cycle passant par chaque arc exactement une fois
- Programmation linéaire (1984)
- Factorisation de polynôme dans $\mathbb{Q}$ (1982)
- Un Rubik's cube arbitrairement coloré est-il soluble ? (2015)
- Évaluation d'un circuit logique à partir des entrées

Limites de **P**

- Ne distingue pas n^{50} de n^2.
- Les constantes sont ignorées : ne distingue pas $1000n^2 + 100000$ et n^2.
- Considère le pire cas. Le cas moyen serait-il plus utile ?
- Ne considère que des solutions exactes. Qu'en est-il des solutions approximatives ?
- Précision des calculs : tous les modèles de calcul Church-Turing équivalents sont *discrets*. Qu'en est-il d'un modèle de calcul dans $\mathbb{R}$ et non pas dans les flottants informatiques ? *(non réalisable)*
- Les MT sont déterministes. Qu'en est-il si on a accès à de l'aléatoire ? *(classe* **BPP***)*
- Qu'en est-il d'un ordinateur quantique ? *(classe* **BQP***)*

Le codage des données est (presque) sans importance

Passer d'une représentation des données à une autre est polynomial (sauf codage catastrophique : base 1) $\Rightarrow$ un algorithme efficace pour une représentation reste efficace pour une autre (commencer par convertir les données).

Exemple : représentation d'un graphe de n nœuds :

- Matrice d'adjacence : matrice de n^2 cases
- Listes d'adjacence : n listes d'au plus n nœuds
- Ensemble d'arêtes : ensemble d'au plus n^2 couples

Théorème d'accélération : plusieurs transitions en une

Accélération linéaire

Pour toute constante $\epsilon > 0$, si un langage L est reconnu par une machine M en $O(t(n))$, alors il existe une machine M' reconnaissant L en $O((1+\epsilon)n + \epsilon t(n))$.

Principe : agrandir l'alphabet pour faire plusieurs transitions de M en une seule dans M'.
Soit Σ l'alphabet de M et $c = \lceil 6/\epsilon \rceil$ le facteur d'accélération. On construit M' sur $\Sigma \cup \Sigma^c$: une case de M' contient c symboles et une transition de M' effectue c transitions de M.
M' a besoin de déterminer au plus c cases de M à partir de sa case courante, donc a besoin de lire la case courante et ses voisines à droite et à gauche, puis simuler c transitions de M en une, et écrire le résultat $\rightarrow$ 6 transitions de M'.

Temps = conversion de Σ vers $\Sigma^c = n + \lceil \frac{n}{c} \rceil = O((1+\epsilon)n)$
$+$ exécution de $M' = \lceil \frac{6}{c} \rceil \cdot t(n) = O(\epsilon t(n))$

Réduction de l'alphabet

Réduction de l'alphabet

Soit une machine de Turing M sur un alphabet Σ, alors il existe une machine de Turing M' définie sur l'alphabet $\Sigma' = \{0, 1\}$ telle que (avec $\varphi : \Sigma^* \to \Sigma'^*$ une bijection de codage) :

- $M(x) = \varphi^{-1}(M'(\varphi(x)))$
- Si M s'arrête sur l'entrée x en t pas de calcul, alors M' s'arrête sur l'entrée $\varphi(x)$ en moins de $(6 \cdot t \cdot \log|\Sigma|)$ pas ;
- Si M s'arrête sur l'entrée x en utilisant un espace s, alors M' s'arrête sur l'entrée $\varphi(x)$ en espace $\leqslant (2 \cdot s \cdot \log|\Sigma|)$.

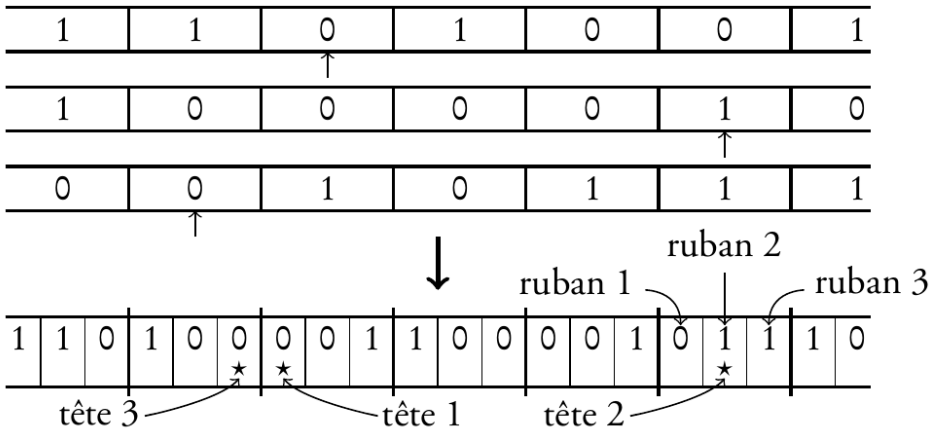
(même idée que pour l'accélération)

Réduction du nombre de rubans

Réduction de k rubans à un seul

Une machine de Turing à k rubans fonctionnant en temps t peut être simulée par une machine de Turing à un ruban et fonctionnant en temps $O(t^2)$.

Étape 1 : un seul ruban, k têtes : alterner les symboles de chaque ruban



Étape 2 : une seule tête, faire des aller-retours pour récupérer les k symboles et simuler la transition.

L'utilisation du parallélisme ne change pas la classe de complexité.

(dessin de Sylvain Perifel)

Le modèle de calcul est sans importance

Le modèle de calcul (de la famille Church-Turing) est sans importance.

Un problème de la classe **P** possède un algorithme efficace (polynomial) dans n'importe quel modèle ou langage : un algorithme efficace en CAML est un algorithme efficace en JAVA, ou un algorithme efficace en C.

Machines non déterministes

On sait qu'une machine non déterministe M peut être convertie en une machine déterministe en explorant en largeur l'arbre du calcul non déterministe. cf cours calculabilité

Déterminisation

Si une machine non déterministe a besoin d'au plus $t(n)$ transitions pour décider une entrée n, alors la machine déterminisée a besoin de $O(2^{t(n)})$ transitions.

La déterminisation change la classe de complexité : polynomial $\rightarrow$ exponentiel

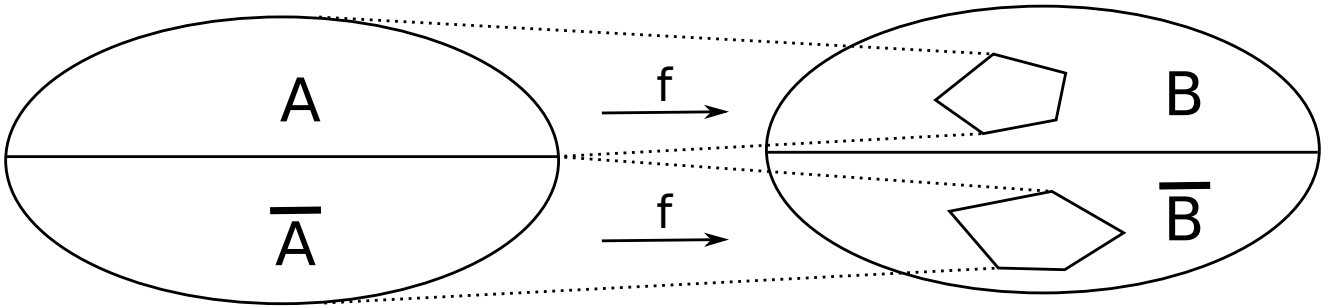
À venir, classe **NP**

Réduction de problèmes

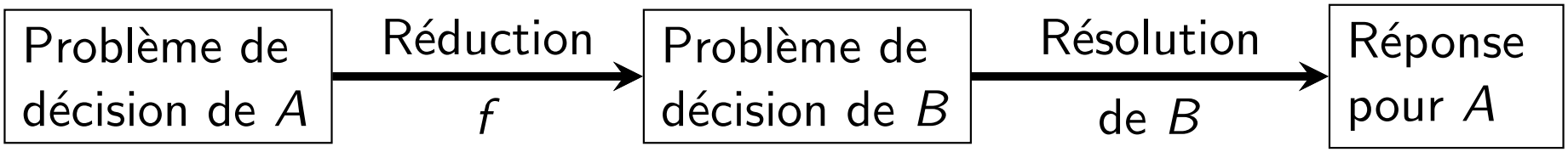
Réduction $A \leqslant B$

(Rappel) Soit A et B deux problèmes de décision. Une réduction de A vers B est une fonction calculable f telle que $x \in A \Leftrightarrow f(x) \in B$.

(A est plus facile que B, A se réduit à B)



Réduction de problèmes



Réduction et complexité

- Si $A \leqslant B$ et $B \in \mathbf{P}$, alors $A \in \mathbf{P}$
- Si $A \leqslant B$ et $A \notin \mathbf{P}$, alors $B \notin \mathbf{P}$

Temps exponentiel

Classes exponentielles

$$\textbf{ETIME} \triangleq \bigcup_{k\in\mathbb{N}} \mathrm{DTime}(2^{kn}) \qquad \textbf{EXPTIME} \triangleq \bigcup_{k\in\mathbb{N}} \mathrm{DTime}(2^{n^k})$$

Inclusion

$$\textbf{P} \subsetneq \textbf{ETIME} \subsetneq \textbf{EXPTIME}$$

(d'après la hiérarchie temporelle T21)

Exemple d'**EXPTIME** :

- Soit une machine de Turing M, $M(\epsilon)$ s'arrête-t-elle en $\leq k$ étapes ? La seule manière est d'exécuter M pendant k transitions. L'entrée k, codée en base 2, prend $\lceil \log_2 k \rceil$ bits, l'algorithme prend $O(2^{\log k})$ pas.
- Échecs, Go (généralisés) : une partie peut nécessiter un nombre de pas exponentiel de la taille du damier.
- Vérification d'une formule LTL : exponentiel en le nombre d'opérateurs temporels.

Bilan

- Définition de classes de complexité pour des problèmes de décision déterministes
- Classe **P** = les problèmes faciles, à résolution en temps polynomial
- Non importance du modèle de calcul

Plan

Problème de vérification

Que demander d'une procédure de vérification ?

- Ne jamais accepter une mauvaise solution
- Toujours accepter une solution correcte fournie avec un certificat
- Être efficace, c'est-à-dire polynomiale

Vérificateur

Un vérificateur pour un problème A est une machine M tel que
$A = \{w \mid M \text{ accepte } \langle w, u \rangle \text{ pour un certain mot } u\}$

u est le certificat qui permet la preuve que w est dans A.

Classe **NP**

Définition de **NP**

Un langage $L \subseteq \{0,1\}^*$ appartient à la classe de complexité **NP** $\triangleq$
$\exists M$ une machine de Turing polynomiale, $\exists p$ un polynôme :

- $\forall x \in L, \exists y \in \{0,1\}^{p(|x|)}$: M accepte $\langle x, y \rangle$.
- $\forall x \notin L, \forall y \in \{0,1\}^*$: M rejette $\langle x, y \rangle$.

Un y pour lequel M accepte est un certificat ou un témoin.
Observer que le certificat est de longueur polynomiale, car la machine ne peut lire en temps polynomial qu'un nombre polynomial de symboles.

(**NP** ne signifie pas Non Polynomial mais Non déterministe Polynomial)

Exemple de problèmes NP

- Satisfiabilité d'une formule propositionnelle
  certificat : valuation des symboles
- k-colorabilité d'un graphe
  certificat : l'affectation des couleurs
- Problème du sac à dos
  certificat : les objets retenus
- Factorisation : x a-t-il un diviseur premier dans l'intervalle $[a, b]$?
  certificat : le diviseur
- Programmation linéaire
  certificat : valuation des variables
- Existence d'un cycle hamiltonien (qui passe exactement une fois par tous les nœuds)
  certificat : un tel cycle

Temps non déterministe NTIME

NTIME

La classe $\mathrm{NTime}(f(n))$ est la classe des problèmes de décision décidé en $O(f(n))$ pas par une machine de Turing non déterministe : un langage L appartient à la classe de complexité $\mathrm{NTime}(f(n)) \triangleq \exists M$ une machine de Turing non déterministe :

- $\forall x \in L$: M accepte x.
- $\forall x \notin L$: M rejette x.
- $\forall x$: M prend $O(f(|x|))$ pas pour terminer

(toute branche de son arbre de calcul est majorée en cas de rejet)

Définition alternative de **NP**

NP : Non déterministe, polynomial

NP est la classe des problèmes décidables en temps polynomial par une machine non déterministe :

$$\mathbf{NP} \triangleq \bigcup_{c \in \mathbb{N}} \mathrm{NTime}(n^c)$$

Classes exponentielles

$$\mathbf{NETIME} \triangleq \bigcup_{k \in \mathbb{N}} \mathrm{NTime}(2^{kn}) \qquad \mathbf{NEXPTIME} \triangleq \bigcup_{k \in \mathbb{N}} \mathrm{NTime}(2^{n^k})$$

Fermeture

fermeture

Pour f constructible en temps, $\mathrm{NTime}(f)$ est fermé par union et intersection :

Pour L_1, L_2 problèmes de décision dans $\mathrm{NTime}(f)$,

- $L_1 \cup L_2 \in \mathrm{NTime}(f)$
- $L_1 \cap L_2 \in \mathrm{NTime}(f)$
- $\mathrm{NTime}(f)$ n'est pas fermé par complémentaire (pense-t-on)

Hiérarchie

Ordre strict

Pour f, g constructibles en temps et croissantes,
$f(n+1) = o(g(n)) \Rightarrow \mathrm{NTime}(f(n)) \subsetneq \mathrm{NTime}(g(n))$

Hiérarchie

- $\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{EXPTIME} \subseteq \mathbf{NEXPTIME}$
- $\mathbf{P} \subsetneq \mathbf{EXPTIME}$ $\quad$ $\mathbf{NP} \subsetneq \mathbf{NEXPTIME}$
- $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$
- $\mathbf{NP} \stackrel{?}{=} \mathbf{EXPTIME}$
- $\mathbf{EXPTIME} \stackrel{?}{=} \mathbf{NEXPTIME}$

(on pense que toutes les inclusions sont strictes)

$\mathbf{P} \stackrel{?}{=} \mathbf{NP}$

Un problème dont on peut vérifier efficacement la solution est-il soluble par un algorithme efficace ?
Un problème possédant un espace exponentiel de solutions potentielles a-t-il un algorithme polynomial qui permet de trouver une solution ?

- Prix de 1 million de US$ du Clay Mathematics Institute (*Millennium Prize Problems*)
- La majorité des chercheurs pensent que $\mathbf{P} \neq \mathbf{NP}$:
  - Intuitivement, vérifier semble plus facile que calculer (cf T8).
  - Depuis 70 ans, on cherche des algorithmes polynomiaux pour des centaines de problèmes **NP**-complets, sans succès.
    (Les problèmes **NP**-complets sont les problèmes les plus durs de la classe **NP**, à suivre)

Et si **P** = **NP** ?

- Programmes efficaces pour prouver tout théorème (une preuve est un certificat de taille polynomiale du théorème).
- Optimalité des solutions pour tous les problèmes sur les graphes.
- Inutilité de l'aléatoire : les algorithmes probabilistes ne seraient pas mieux que les algorithmes déterministes.
- Tout chiffrage (cryptage) serait efficacement déchiffrable $\Rightarrow$ plus de sécurité (*privacy*).

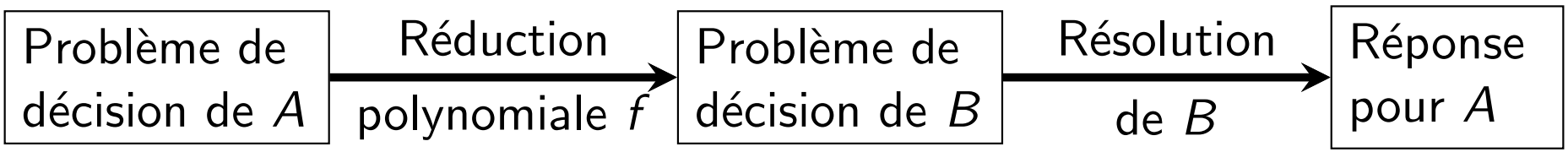
(côté pratique, si **P** = **NP** mais que le facteur pour passer de vérification à résolution est n^{beaucoup}, ça ne changera pas grand chose ; côté théorique, cela serait un bouleversement)

Réduction polynomiale de problèmes

Réduction polynomiale $A \leqslant_p B$

Rappel : soit A et B deux problèmes de décision. Une réduction polynomiale de A vers B est une fonction f calculable en temps polynomial telle que $x \in A \Leftrightarrow f(x) \in B$.

(A est plus facile que B, A se réduit à B)



La seule différence avec la réduction générale est la contrainte que f est calculable en temps polynomial.

NP-complétude

Problème NP-difficile (ou NP-dur)

Un problème de décision L est **NP**-difficile $\triangleq \forall L' \in \mathbf{NP} : L' \leqslant_p L$
(L' est réductible en temps polynomial à L).

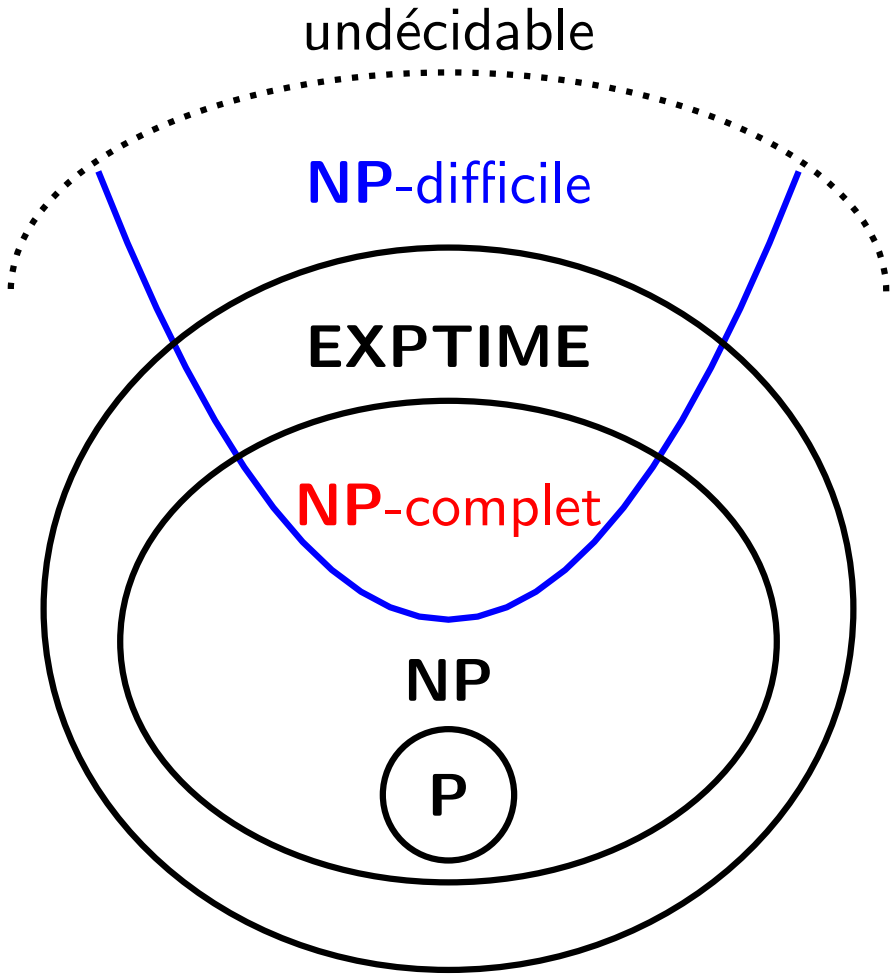
(intuition : au moins aussi difficile que n'importe quel problème dans **NP**)

Problème NP-complet

L est **NP**-complet $\triangleq$ L est **NP**-difficile et $L \in \mathbf{NP}$.

(intuition : un des problèmes les plus difficiles de **NP**)

NP-complétude



(sous l'hypothèse **P** $\neq$ **NP** et **NP** $\neq$ **EXPTIME**)

Intérêt des problèmes complets

Les problèmes complets capturent la classe tout entière :

- Si l'on prouve qu'un problème complet pour une classe $\mathcal{A}$ appartient à la classe $\mathcal{B}$, on obtient $\mathcal{A} \subseteq \mathcal{B}$.
  Si l'on trouve un problème **NP**-complet qui est dans **P**, alors **P** = **NP**.
- si on prouve que ce problème n'appartient pas à la classe $\mathcal{C}$, on obtient $\mathcal{A} \not\subset \mathcal{C}$.

Étudier une classe de complexité revient à étudier ses problèmes complets.

- Si on étudie une classe intéressante (comme **P** et **NP**), on va chercher leurs problèmes complets.
- Si on étudie un problème intéressant, on va regarder s'il existe une classe de complexité naturelle pour laquelle il est complet.

Remarques sur la **X**-complétude

- Un problème est **X**-complet si tous les problèmes de **X** se réduisent à lui. Pour **X** quelconque, rien ne dit qu'il existe des problèmes **X**-complets.
  (c'est bon pour **NP**-complet)
- Tous les problèmes **X**-complets sont équivalents (tous réductibles entre eux).
- Il existe des problèmes hors de **NP** qui ne sont pour autant pas **NP**-complets.

Un problème artificiel **NP**-complet

Acceptation en temps max

Le problème $A \triangleq \{\langle\langle N\rangle, x, 1^t\rangle \mid N(x)$ accepte en temps $\leq t\}$, où N est une machine de Turing non déterministe, x un mot d'entrée et t un entier (donné en base 1), est **NP**-complet.

1. $A \in \mathbf{NP}$: simuler la machine $N(x)$ par une machine de Turing universelle. Le certificat est un chemin de calcul qui conduit à l'acceptation de $N(x)$ en moins de t pas.
2. A est **NP**-difficile : soit un problème B dans **NP**. $B(n)$ est reconnu par une MT non déterministe M en temps $p(n)$. La fonction $f : w \to \langle\langle M\rangle, w, 1^{p(w)}\rangle$ est calculable en temps polynomial et vérifie $w \in B \Leftrightarrow f(w) \in A$. Donc $B \leqslant A$.

(problème artificiel : jeu sur les définitions)

Le problème SAT

Définition

SAT (ou satisfiabilité booléenne) est le langage formé par toutes les formules propositionnelles satisfiables, c'est-à-dire qui ont une valuation des variables telle que la formule est évaluée à vrai.

Pour simplifier et parce qu'il existe une transformation polynomiale, les formules considérées sont en forme normal conjonctive (CNF) :
$C_1 \wedge C_2 \wedge ... \wedge C_k$, où chaque C_i est de la forme
$x_{i_1} \vee \ldots \vee x_{i_q} \vee \neg x_{i_r} \vee \ldots \vee \neg x_{i_s}$.

Exemple :

$(\neg x1 \vee x2) \wedge (\neg x2 \vee x3) \wedge (\neg x3) \wedge (x1 \vee x3 \vee x4) \wedge (\neg x4 \vee x1) \quad \notin$ SAT

$(\neg x1 \vee x2) \wedge (\neg x2 \vee x3) \wedge (x1 \vee x3 \vee x4) \wedge (\neg x4 \vee x1) \quad \in$ SAT

Théorème de Cook-Levin

SAT est **NP**-complet.

Preuve : SAT est **NP**-complet

1. SAT $\in$ **NP** : compte tenu d'une "bonne" valuation des variables (le certificat), on vérifie la formule en temps polynomial de sa taille.
2. SAT est **NP**-difficile : soit un langage $L \in$ **NP**, donc il existe une MT non déterministe $M(x)$ qui décide $x \in L$. On construit (en temps polynomial de $|x|$) une formule CNF qui représente les calculs acceptants possibles de $M(x)$ (la formule est donc de taille polynomiale de $|x|$). Cette formule est satisfiable ssi M accepte x.

Preuve : SAT est **NP**-difficile (suite)

M nécessite au plus $T = |x|^c$ pas $\rightarrow$ $T+1$ configurations de taille $T+1$:

- $c_{t,i}$ est la valeur inscrite dans la i-ème case à l'étape t.
- $p_{t,j}$ vaut 1 si la tête de lecture est sur la case j à l'étape t, 0 sinon.
- $s_{t,k}$ vaut 1 si M est dans l'état k à l'étape t, et 0 sinon.

Formules :

- Un seul état à la fois : $\forall t, \forall k \neq k' : s_{t,k} \implies \neg s_{t,k'}$
- Une seule position de la tête à la fois : $\forall t, \forall j \neq j' : p_{t,j} \implies \neg p_{t,j'}$
- La seule case qui peut changer est celle où se trouve la tête : $\forall t > 0, \forall j : \neg p_{t,j} \implies (c_{t+1,j} = c_{t,j})$.
- État initial : $s_{0,0} \wedge p_{0,0} \wedge \forall k \leq |x| : c_{0,k} = x_k$
- Transition (état k, lecture b dans case j) $\rightarrow$ (état k', écriture b', mouvement j') :
  $s_{t,k} \wedge p_{t,j} \wedge (c_{t,j} = b) \implies s_{t+1,k'} \wedge p_{t+1,j'} \wedge (c_{t+1,j} = b')$.
- Le calcul accepte en temps polynomial : $\bigvee_{0<t\leq T} (s_{t,Halt} \wedge c_{t,0})$

Conséquences de la **NP**-complétude de SAT

- Si on réduit polynomialement SAT à un problème A, alors A est **NP**-difficile : A est au moins aussi difficile que SAT.
  $\rightarrow$ en supposant **P** $\neq$ **NP**, il n'existe pas d'algorithme polynomial pour A.
- Si un problème A se réduit polynomialement à SAT, alors A est dans **NP**.
  $\rightarrow$ on peut résoudre A en utilisant la réduction à SAT + un solveur SAT.
  Si A est un problème difficile, cela peut être plus facile et peut-être plus efficace que de chercher un algorithme direct.

Solveur SAT

SAT est **NP**-complet $\rightarrow$ tout algorithme sera exponentiel dans le pire cas (sauf si $\mathbf{P} = \mathbf{NP}$).
Pour autant, il existe des solveurs SAT extrêmement performants la majorité du temps :

- 1000 variables / 10 000 clauses en < 2 ms
- 100 000 variables / 200 000 clauses en quelques minutes

Mais pas de magie : il y aura toujours des formules de quelques dizaines ou centaines de variables qui nécessiteraient plusieurs milliards d'années.

3-SAT est **NP**-complet

3-SAT (où les clauses ont au plus trois variables) est **NP**-complet.

Preuve : SAT $\leqslant_p$ 3-SAT.
Réduction de SAT vers 3-SAT :
Transformer chaque clause $C = x_1 \vee ... \vee x_n$ qui contient $n > 3$ valeurs en la conjonction de deux clauses (où z est une nouvelle variable)
$C_1 = x_1 \vee ... \vee x_{n-2} \vee z$ et
$C_2 = \neg z \vee x_{n-1} \vee x_n$.
Répéter jusqu'à n'avoir plus que des clauses de taille 3.
La réduction est polynomiale.

(trivialement 3-SAT $\leqslant_p$ SAT)

Autres problèmes **NP**-complets : coloration de graphes

3-colorabilité de carte

Un graphe planaire est-il colorable avec trois couleurs ?
(colorable sans que deux régions adjacentes soient de la même couleur)

(théorème des 4 couleurs : tout graphe planaire est colorable avec 4 couleurs, prouvé en 1976 par ordinateur – énumération des cas –, pas de preuve connue sans ordinateur)

La 3-colorabilité est **NP**-complet, la 2-colorabilité est dans **P**.

Coloration de graphe

Un graphe arbitraire est-il colorable avec k couleurs ?
(colorable sans que deux nœuds adjacents soient de la même couleur)

La coloration à k couleurs est **NP**-complet pour $k > 2$.

3-colorabilité $\leqslant_p$ SAT

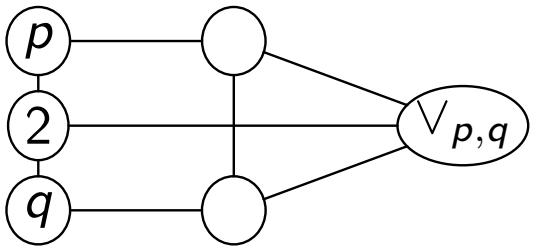
Soit un graphe $G = (V, E)$, construisons une formule φ_G telle que G est 3-colorable $\Leftrightarrow$ φ_G est satisfiable :

- Trois couleurs $1, 2, 3$
- Variables v_i pour $v \in V$ et $i \in \{1, 2, 3\}$
  (v_i est vraie si le sommet v est de couleur i)
- Chaque sommet est coloré par une et une seule couleur :
  $\bigwedge_{v \in V} ((v_1 \vee v_2 \vee v_3) \wedge \bigwedge_{i \neq j} (\neg v_i \vee \neg v_j))$
- Deux sommets voisins n'ont pas la même couleur :
  $\bigwedge_{(u,v) \in E} \bigwedge_{i \in \{1,2,3\}} (\neg u_i \vee \neg v_i)$

SAT $\leqslant_p$ 3-colorabilité

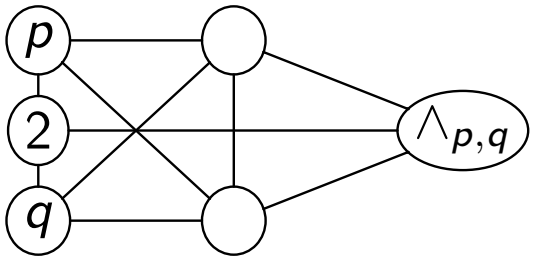
Soit une formule φ, construisons un graphe dont la 3-coloration indique la satisfiabilité de φ :

- Trois couleurs 0, 1 (= vrai), 2
- Trois sommets distingués 0,1,2 de couleur 0,1,2, reliés entre eux
- Pour chaque variable v_i, sommets v_i et $\neg v_i$, reliés entre eux et à 2
- $p \vee q$ codé par :



nœud $\vee_{p,q}$ colorable par 1 ssi p ou q sont colorés 1

- $p \wedge q$ codé par :



nœud $\wedge_{p,q}$ colorable par 1 ssi p et q sont colorés 1

- Sommet φ relié à 0 et 2 (à colorer en 1).

Autres problèmes **NP**-complets : graphes

cycle hamiltoniens

L'existence d'un cycle hamiltonien (qui passe exactement une fois par tous les nœuds) est **NP**-complet.

(l'existence d'un cycle eulérien, passant exactement une fois par chaque arc, est **P**)

Clique

Dans un graphe non orienté, une clique est un ensemble de sommet tous reliés deux à deux (le sous-graphe est complet).
Étant donné un graphe non orienté et un entier k, existe-t-il une clique de taille k ?
Le problème Clique est **NP**-complet (2006).

Équation diophantienne

Équation diophantienne

Équation polynomiale à une ou plusieurs inconnues, à coefficients entiers, et dont les solutions sont cherchées parmi les nombres entiers.
(dixième problème de Hilbert, 1900, prouvé indécidable en 1970)

2DIO : Équation diophantienne restreinte à deux inconnues

Quelles équations $Ax^2 + By + C = 0$ sont solubles par entiers positifs ?

(1975)

2DIO est **NP**-complet.

Autres problèmes **NP**-complets

Voyageur de commerce

Étant donné des villes, les distances les séparant et une limite, existe-t-il un cycle qui passe par toutes les villes sans dépasser la limite de distance ?

Somme

Étant donné des objets avec valeur, existe-t-il un sous-ensemble d'objets qui atteint une valeur donnée ?
Soit des entiers $v_1, \ldots, v_k > 0$ et un entier s ; existe-t-il $x_1, \ldots, x_k \in \{0, 1\}$ tel que $\Sigma_{i=1}^{k} x_i v_i = s$?

Sac à dos

Étant donné des objets avec valeur et poids, une limite de poids, existe-t-il un sous-ensemble d'objets qui maximise la valeur sans dépasser le coût ?
Soit des couples d'entiers positifs $(v_1, p_1), \ldots, (v_k, p_k)$ et un entier C, trouver $x_1, \ldots, x_k \in \{0, 1\}$ qui maximise $\Sigma_{i=1}^{k} x_i v_i$ en respectant $\Sigma_{i=1}^{k} x_i p_i \leq C$.

Réflexions sur la **NP**-complétude

De nombreuses disciplines (biologie, chimie, économie, neuroscience, électronique...) décrivent le comportement des systèmes par des *règles simples et locales*, analogues à des algorithmes.
Ces modèles sont souvent indécidables ou au mieux **NP**-complets (exemple : calcul de la configuration d'énergie minimale d'une protéine – *protein folding*) $\rightarrow$ leur résolution (quand elle est faisable) prend un temps exponentiel. Or la nature est *efficace* : un processus a lieu en temps borné et son observation nécessite un temps borné.

- Est-ce car le modèle **NP**-complet n'est qu'une approximation d'un modèle plus simple (polynomial) inconnu ?
- Est-ce car $\mathbf{P} = \mathbf{NP}$?
- Est-ce car les problèmes difficiles n'existent pas dans la nature ?

(Wigderson 2019)

Bilan sur la complexité en temps

- Définition de classes de complexité
- Distinction résolution d'un problème / vérification d'une solution
- Notion de réduction : passer d'un problème à un autre
- Classe **P** : les problèmes à résolution polynomiale
- Classe **NP** : les problèmes à vérification polynomiale
- Problèmes **NP**-complets : les problèmes **NP** les plus difficiles
- $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$

Troisième partie

Complexité en espace

Plan

Objectifs

- L'espace (la mémoire) est une ressource précieuse, autant que le temps (systèmes embarqués par exemple).
- Même si un algorithme est rapide, il faut que sa consommation en espace n'explose pas (mais espace $\leq$ temps : on ne peut pas manipuler plus de cases que de transitions effectuées).
- L'espace se comporte fondamentalement différemment du temps : il est réutilisable.
- Il n'y a pas de sous-classes intéressantes de **P** définies pour le temps, mais il y en a pour l'espace (espace constant, espace logarithmique).

Quelle mesure ?

Que mesure-t-on ?

On mesure uniquement l'espace utilisé par le calcul.

On ne compte pas la taille de l'entrée ni de la sortie, ni la taille de l'algorithme (constant).

Attention au risque de donner de l'espace "gratuit", par exemple en réutilisant l'espace pour l'entrée ou celui pour la sortie :

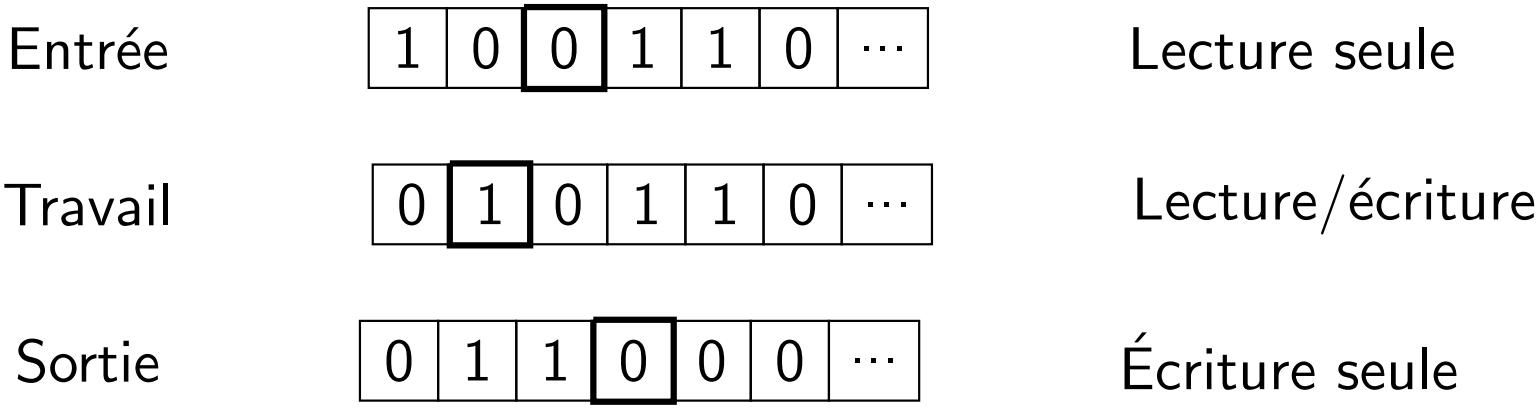
- L'espace d'entrée doit être uniquement en lecture
- L'espace de sortie doit être uniquement en écriture

Modèle standard

Machine de Turing à trois bandes

Les définitions et résultats sont indépendants du modèle, mais il est plus aisé d'utiliser une machine de Turing à trois bandes :

- Une bande d'entrée en lecture seule.
- Une bande de travail classique, en lecture et écriture.
- Une bande de sortie en écriture seule.



Réduction linéaire de l'espace

Réduction linéaire de l'espace

Pour toute constante $\epsilon > 0$, si un langage L est reconnu par une machine M en espace $s(n)$, alors il existe une machine M' reconnaissant L en $\epsilon \cdot s(n)$.

Principe : similaire à l'accélération temporelle : agrandir l'alphabet de la bande de travail pour avoir besoin de moins de cases. Par exemple en passant de $\{0, 1\}$ à $\{00, 01, 10, 11\}$, on divise par deux le nombre de cases.
(c'est pour cela qu'on va plutôt compter le nombre de bits, et/ou utiliser uniquement l'alphabet $\{0, 1\}$)

DSPACE

DSPACE

La classe $\mathrm{DSPACE}(s(n))$ est la classe des problèmes de décision décidé en espace $O(s(n))$: un langage L appartient à la classe de complexité $\mathrm{DSPACE}(s(n)) \triangleq \exists M$ une machine de Turing déterministe (à trois bandes) :

- $\forall x \in L : M$ accepte x.
- $\forall x \notin L : M$ rejette x.
- $\forall x : M$ écrit $O(s(|x|))$ cases

(Comme pour le temps, il faut que s soit constructible en espace : le calcul de $s(n)$ nécessite un espace $O(s(n))$, sinon s apporterait un espace gratuit)

(Contrairement au temps, on pourrait affaiblir le cas $\forall x \notin L$ en M rejette x ou M boucle mais il devient alors délicat de comparer les classes de complexité temporelles et spatiales)

Propriétés de DSPACE

fermeture

Pour s constructible en espace, $\mathrm{DSPACE}(s(n))$ est fermé par union, intersection et complémentaire : pour L_1, L_2 problèmes de décision dans $\mathrm{DSPACE}(s(n))$,

- $L_1 \cup L_2 \in \mathrm{DSPACE}(s(n))$
- $L_1 \cap L_2 \in \mathrm{DSPACE}(s(n))$
- $\overline{L_1} \in \mathrm{DSPACE}(s(n))$

Preuve comme pour DTIME : on utilise le max des espaces utilisés par chaque problème (en le réutilisant) :

- $\cup$: exécuter M_1 qui reconnaît L_1, si échec exécuter M_2 qui reconnaît L_2.
- $\cap$: exécuter M_1 qui reconnaît L_1, si ok exécuter M_2 qui reconnaît L_2.
- $\overline{L_1}$: inverser la décision.

Hiérarchie spatiale

Ordre partiel

Pour f, g constructibles en espace,
$f = O(g) \Rightarrow \text{DSPACE}(f) \subseteq \text{DSPACE}(g)$

Ordre strict

Pour f, g constructibles en espace,
$f = o(g) \Rightarrow \text{DSPACE}(f) \subsetneq \text{DSPACE}(g)$

Démonstration analogue au cas temporel.

NSPACE

NSPACE

La classe $\mathrm{NSPACE}(s(n))$ est la classe des problèmes de décision vérifié en $O(s(n))$ espace : un langage L appartient à la classe de complexité $\mathrm{NSPACE}(s(n)) \triangleq \exists M$ une machine de Turing *non déterministe* (à trois bandes) :

- $\forall x \in L : M$ accepte x.
- $\forall x \notin L : M$ rejette x.
- $\forall x : M$ écrit $O(s(|x|))$ cases

Graphe de configuration

Définition

Soit une machine de Turing M avec une entrée x,

- Une configuration est son état, le contenu de la bande de travail et les positions des têtes d'entrée et de travail.
- Le graphe de configuration est le graphe constitué des configuration avec des arêtes orientées reliant une configuration à une suivante selon la fonction de transition, et partant des configurations initiales.

Inutile de s'occuper de la bande de sortie, en écriture seule, et qui dans un problème de décision se réduit à un bit écrit à la dernière transition.
(si M est déterministe, le graphe est linéaire)

Propriétés du graphe de configuration

Nombre de nœuds

Pour $s(n)$ tel que $s(n) \geq \log n$, le graphe de configuration d'une machine dans $\mathrm{NSPACE}(s(n))$ a $2^{O(s(n))}$ nœuds.

- Le nombre de configurations est $\leq Q.B.T$, où Q est le nombre d'états, B est le nombre de contenus possibles et T est le nombre de positions possibles des têtes.
- Comme M est en espace $O(s(n))$, $\exists c$ telle que le nombre de cases utilisées est $\leq c \cdot s(n)$ et le nombre de contenus possibles du ruban de travail est $\leq 2^{c \cdot s(n)}$.
- La tête de travail est sur l'une des $c \cdot s(n)$ cases, son codage prend $\log(c \cdot s(n))$ bits.
- La tête d'entrée est sur une des n cases, son codage prend $\log n$ bits.
- Le produit total donne $|Q| \cdot \log(c \cdot s(n)) \cdot \log n \cdot 2^{c \cdot s(n)}$. Comme $\log(c \cdot s(n)) \cdot \log n = o(2^{s(n)})$, cqfd.

Lien temps espace

Pour f constructible en temps et espace,
$\text{DTIME}(f(n)) \subseteq \text{DSPACE}(f(n)) \subseteq \text{NSPACE}(f(n)) \subseteq \text{DTIME}(2^{O(f(n))})$

- $\text{DTIME}(f(n)) \subseteq \text{DSPACE}(f(n))$: Une machine ne peut écrire qu'une case par transition. En $f(n)$ transitions, elle ne peut utiliser qu'au plus $f(n)$ cases.
- $\text{DSPACE}(f(n)) \subseteq \text{NSPACE}(f(n))$: une MT déterministe est un cas particulier de MT non déterministe.
- $\text{NSPACE}(f(n)) \subseteq \text{DTIME}(2^{O(f(n))})$: Si une MT s'arrête, elle ne peut pas passer deux fois par la même configuration. Il y a $2^{O(f(n))}$ configurations, donc la machine s'arrête en moins de $2^{O(f(n))}$ transitions.

Théorème de Savitch

Théorème de Savitch

Soit une fonction s telle que $s(n) \geq \log n$ pour n assez grand. Une machine non déterministe qui fonctionne en espace $O(s(n))$ est équivalente à une machine de Turing déterministe en espace $O(s(n)^2)$:

$$\text{NSPACE}(s(n)) \subseteq \text{DSPACE}(s(n)^2)$$

Conséquence : pour l'espace, le non-déterminisme n'est intéressant que pour des classes de complexité $< O(\text{polynôme}(n))$.

Preuve du théorème de Savitch

Soit une machine M dans $\mathrm{NSPACE}(s(n))$, on simule $M(x)$ par l'algorithme $\mathrm{ACCESS}(C, C', t)$ qui renvoie vrai si M peut passer de C à C' en $< t'$ transitions :

```
ACCESS(C, C', t) ≜
  if t ≤ 1 then return C = C'
  else
    for all C'' de taille ≤ s(|x|) do
       if ACCESS(C, C'', ⌈t/2⌉) and ACCESS(C'', C', ⌊t/2⌋) return true;
    end for
    return false
```

M accepte x en espace $s(n)$, donc en temps $2^{O(s(n))}$:
$\exists c : \mathrm{ACCESS}(C_0, C_f, 2^{c \cdot s(n)})$.
Espace utilisé = taille pile $\times$ espace d'un appel
Pile des appels récursifs = $c \cdot s(n)$ (dichotomie)
Espace d'un appel = $(C, C', C'', t) = 3 * s(n) + c \cdot s(n)$ (t codé en binaire)
Total = $O(s(n)^2)$

La classe **1SPACE**

Classe **1SPACE**

1SPACE = problèmes décidables en espace constant :

$$\textbf{1SPACE} \triangleq \text{DSPACE}(1)$$

Exemples :

- Parité d'un entier
- Incrémenter un entier
- Nombre pair de 1 dans un mot de $\{0,1\}^*$
- And/Or bit-à-bit
- *n*-ième nombre de Fibonacci

Classes LSPACE, NLSPACE

Classe LSPACE

LSPACE = problèmes décidables en espace logarithmique :

$$\textbf{LSPACE} \triangleq \mathrm{DSPACE}(\log(n))$$

- Additionner, multiplier deux entiers
- Comparer deux entiers, trier un ensemble d'entier
- Connexité d'un graphe non orienté (2004)

Classe NLSPACE

NLSPACE = problèmes décidables non déterministiquement en espace logarithmique :

$$\textbf{NLSPACE} \triangleq \mathrm{NSPACE}(\log(n))$$

NLSPACE

Définition alternative de **NLSPACE**

Un langage $L \subseteq \{0,1\}^*$ appartient à la classe **NLSPACE** $\triangleq$
$\exists M$ une machine de Turing $\exists p$ un polynôme :

- $\forall x \in L, \exists y \in \{0,1\}^{p(|x|)}$: M accepte $\langle x, y \rangle$.
- $\forall x \notin L, \forall y \in \{0,1\}^*$: M rejette $\langle x, y \rangle$.
- M utilise au plus $O(\log(n))$ cases.

Le certificat y doit être sur une bande à usage unique : la tête ne peut pas aller vers la gauche (sinon, on offrirait gratuitement un espace polynomial : le certificat)

Problème st-CONN

Problème st-CONN

Soit un graphe orienté G et deux nœuds s et t, existe-t-il un chemin de s à t ?

st-CONN est **NLSPACE**-complet.

La difficulté est dans la définition de la réduction en espace logarithmique

Étrangement, st-UCONN, défini identiquement pour un graphe non orienté, est dans **LSPACE** (2005).

PSPACE, EXPSPACE

Classes PSPACE, EXPSPACE

PSPACE = problèmes décidables en espace polynomial :

$$\textbf{PSPACE} \triangleq \bigcup_{c \in \mathbb{N}} \mathrm{DSPACE}(n^c)$$

EXPSPACE = problèmes décidables en espace exponentiel :

$$\textbf{EXPSPACE} \triangleq \bigcup_{c \in \mathbb{N}} \mathrm{DSPACE}(2^{n^c})$$

$$\textbf{NPSPACE} \triangleq \bigcup_{c \in \mathbb{N}} \mathrm{NSPACE}(n^c) = \textbf{PSPACE}$$

$$\textbf{NEXPSPACE} = \textbf{EXPSPACE}$$

(d'après le théorème de Savitch)

Problème QBF

Problème QBF (ou QSAT)

QBF = ensemble des formules booléennes quantifiées closes qui sont vraies

(closes = sans variable libre)
exemple : $\forall x_1, \exists x_2, x_1 \vee x_2$

QBF est **PSPACE**-complet.

(SAT est le cas particulier de QBF avec que des $\exists$)

QBF est dans **PSPACE**

$QBF(\varphi) \triangleq$

si $\varphi = \exists x : \psi$ alors $QBF(\psi[x \leftarrow 0]) \vee QBF(\psi[x \leftarrow 1])$

sinon si $\varphi = \forall x : \psi$ alors $QBF(\psi[x \leftarrow 0]) \wedge QBF(\psi[x \leftarrow 1])$

sinon évaluer φ (φ n'a plus de quantificateur)

Soit une formule de taille m avec n variables. L'algorithme n'a besoin que d'une seule copie de φ et une pile de taille proportionnelle au nombre de variables $\rightarrow$ espace $O(nm)$.

QBF est **PSPACE**-difficile (1/2)

Pour tout $L \in \mathbf{PSPACE}$, montrons $L \leqslant_p QBF$. Soit une machine M qui décide L en espace polynomial $O(s(n))$ et soit $x \in \{0,1\}^*$.

- On considère le graphe de configuration de $M(x)$.
- On a vu qu'il existe un polynôme $q(n) = c \cdot s(n)$ tel que le graphe de $M(x)$ a au plus $2^{q(n)}$ nœuds.
- On va construire en temps et espace polynomial une formule QBF ψ^x qui capture l'existence d'un chemin de longueur $\leq 2^{q(n)}$ entre le nœud de départ et celui d'acceptation.

QBF est **PSPACE**-difficile (2/2)

$\psi_i(s, e)$ est l'existence d'un chemin de taille $\leq 2^i$ entre s et e.

- $\psi_0(s, e) \triangleq (s = e) \vee (s \rightsquigarrow e)$
- $\psi_{i+1}(s, e) \triangleq \exists m : \psi_i(s, m) \wedge \psi_i(m, e)$ mais cela conduit à une formule de taille exponentielle (double à chaque pas).

  $\psi_{i+1}(s, e) \triangleq$

  $\exists m, \forall u, v, ((u = s \wedge v = m) \vee (u = m \wedge v = e)) \implies \psi_i(u, v)$
- Acceptation $\psi^x = \exists c$ acceptante, $\psi_{q(n)}(c_0, c)$

Taille de ψ^x :

- $|\psi_0|$: énumérer les configurations pour vérifier l'existence d'une transition = un compteur en base 2 = $O(q(n))$
- $|\psi_{i+1}| \leq p(n) + |\psi_i|$: la première partie de la formule récursive utilise un nombre constant de configurations en espace polynomial, donc est en taille polynomiale ($< p(n)$).
- $|\psi^x| = O(q(n) \times p(n) + r(n))$

QBF = jeux à deux joueurs

Trouver une stratégie gagnante est PSPACE-complet

On considère un jeu (comme le Go généralisé) avec un nombre polynomial de mouvements et une description polynomiale des positions et des mouvements, alors l'ensemble des positions avec une stratégie gagnante est **PSPACE**-complet.

On réduit QBF à un jeu généralisé entre

- un joueur E qui décide la valeur des variables quantifiées par $\exists$.
- un joueur A qui décide la valeur des variables quantifiées par $\forall$.

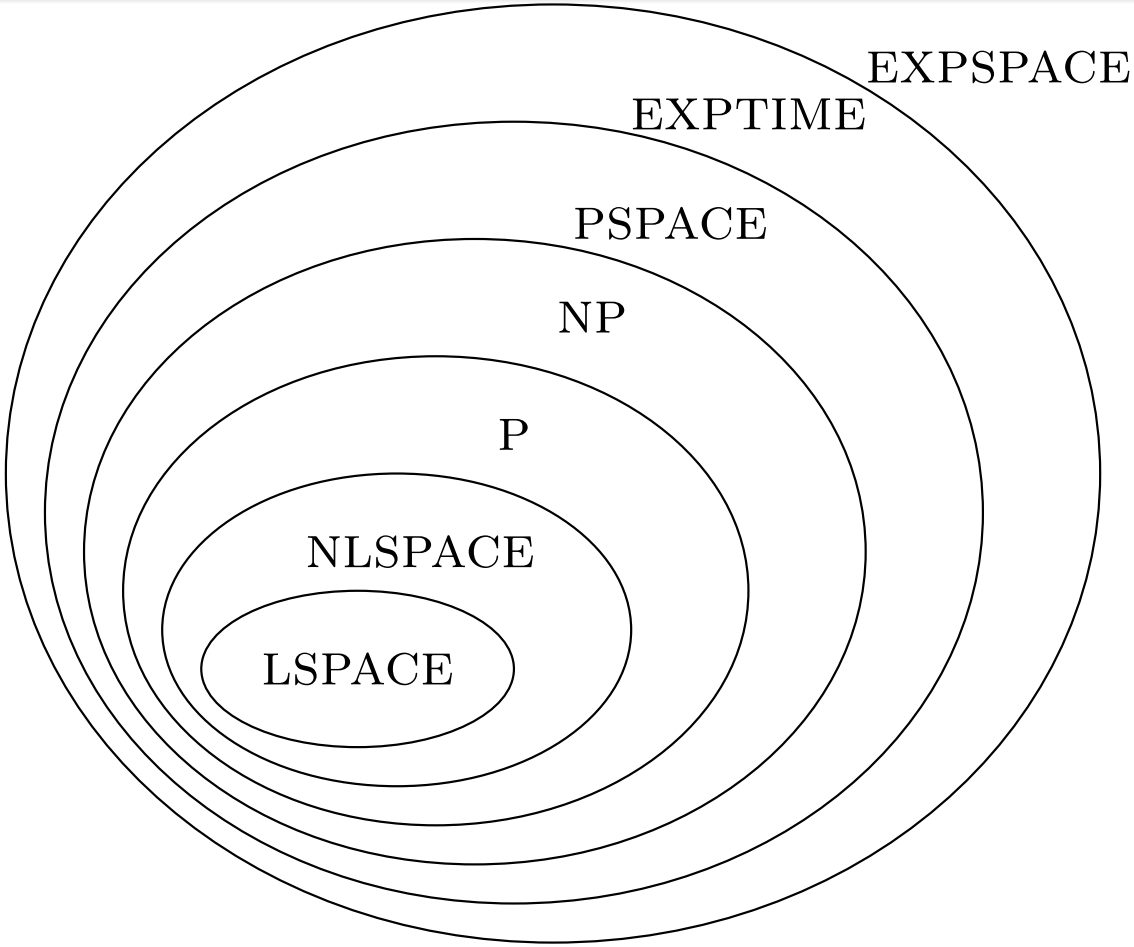
$\exists c_0, \forall c_1, \exists c_2, \forall c_3, \ldots, \exists c_n, (c_0 \to c_1) \wedge (c_1 \to c_2) \wedge \ldots \wedge c_n$ acceptant

Dans ce jeu, la valeur de la formule est l'existence ou non d'une stratégie gagnante pour le joueur existentiel.

Bilan

1SPACE $\subseteq$ LSPACE $\subseteq$ NLSPACE $\subseteq$ P $\subseteq$ NP $\subseteq$ PSPACE $\subseteq$ EXPTIME $\subseteq$ EXPSPACE

- **NLSPACE $\subsetneq$ P**
- **P $\subsetneq$ EXPTIME**
- **PSPACE $\subsetneq$ EXPSPACE**
- On pense que toutes les inclusions sont strictes
- Le non-déterminisme n'apporte rien pour l'espace à partir de **PSPACE**



Quatrième partie

Complexité probabiliste

Plan

Apport du calcul probabiliste

Intuition

- Un algorithme avec la possibilité de faire des choix probabilistes, comme s'il lançait des pièces.
- Un compromis entre déterminisme et non-déterminisme : il ne faut pas forcément toujours accepter, mais il faut accepter "suffisamment souvent".
- Moyen de ne pas surdéterminer un algorithme, en faisant des choix aléatoires plutôt que déterministes.

Deux formulations de l'aléa

En ligne et hors ligne

- En ligne : le lancer de pièces. Lorsqu'il y a un choix aléatoire à faire, on "lance une pièce", c'est-à-dire qu'on tire un bit aléatoire.
  Comme une machine non déterministe aux choix probabilistes.
- Hors ligne : une liste initiale de bits aléatoires. Les choix sont fait en lisant un nouveau bit de cette liste.
  Comme une machine déterministe qui vérifie un certificat généré aléatoirement.

Équivalence

La même intuition que pour **NP** permet de passer d'une formulation à l'autre : la liste initiale peut encoder tous les lancers de pièces, et les lancers de pièces peuvent simuler une lecture dans la liste initiale.

Las Vegas et Monte Carlo

Deux variations d'algorithmes probabilistes

- Algorithme de Las Vegas : algorithme qui renvoie toujours la réponse correcte, mais dont le temps pris pour le calcul est une variable aléatoire.
- Algorithme de Monte Carlo : algorithme qui finit en temps déterministe (en fonction de la taille), mais qui peut retourner des erreurs.

Puisque l'on étudie des classes de complexité, on veut des garanties sur le temps pris par l'algorithme et on se limite donc aux algorithmes de Monte Carlo.

Types d'erreurs

Double erreur et simple erreur

Une caractéristique des calculs probabilistes est la possibilité d'erreur :

- Double erreur : Il peut y avoir une erreur en acceptant ou en rejetant : l'algorithme peut accepter une mauvaise instance, et en rejeter une bonne.
- Simple erreur : Il ne peut y avoir une erreur qu'en acceptation (faux positifs) ou qu'en rejet (faux négatifs).

Digression : complexité lissée (*smoothed complexity*)

De par notre définition classique de la complexité, il est possible qu'un algorithme soit efficace partout sauf pour certaines instances très spécifiques. Faut-il pour autant dire qu'il n'est pas efficace ?

Utiliser l'aléa pour négliger les cas pathologiques

La complexité lissée essaye de résoudre ce problème : le temps mesuré pour une entrée devient l'espérance du temps quand l'entrée est perturbée aléatoirement (par une distribution gaussienne).
On prend ensuite le maximum de ces temps pour chaque taille d'entrée.

Si les entrées qui prennent du temps sont pathologiques (très spécifiques), alors la perturbation fera baisser énormément la complexité.

Machine de Turing probabiliste

Machine de Turing probabiliste

Une machine de Turing probabiliste est une machine de Turing avec deux fonctions de transitions. À chaque pas du calcul, l'une des fonctions est utilisée de manière équiprobable. La machine fonctionne en temps $O(s(n))$ si elle s'arrête en temps $\leq s(|x|)$ pour toute entrée x et quels que soient les choix pris.

(équivalent à une MT non déterministe avec, à chaque pas, deux transitions possibles, équiprobables)

L'arbre d'exécution d'une MT probabiliste est un arbre binaire, la probabilité d'une branche de longueur l est 2^{-l}.

- Dans une MT non déterministe, pour une entrée x, on se demande s'il *existe un chemin* qui conduit à l'acceptation.
- Dans une MT probabiliste, pour une entrée x, on se demande si une *fraction majoritaire* des chemins équiprobables conduisent à l'acceptation.

Tri rapide probabiliste

Tri rapide déterministe

$tridet(x) \triangleq$ si $|x| = 1$ alors retourner x
sinon $T_1 \leftarrow \{x_i \mid x_i < x_1\}$, $T_2 \leftarrow \{x_i \mid x_i > x_1\}$
retourner $[tridet(T_1), x_1, tridet(T_2)]$

Complexité pire cas $= O(n^2)$ (si x est déjà trié)

Tri rapide probabiliste

$triproba(x) \triangleq$ si $|x| = 1$ alors retourner x
sinon $p \leftarrow$ au hasard parmi $1, ..|x|$
$T_1 \leftarrow \{x_i \mid x_i < x_p\}$, $T_2 \leftarrow \{x_i \mid x_i > x_p\}$
retourner $[triproba(T_1), x_p, triproba(T_2)]$

- Temps moyen d'exécution $= O(n \log n)$ sur toute entrée
- Encore mieux : tirer trois pivots et utiliser le x_p médian

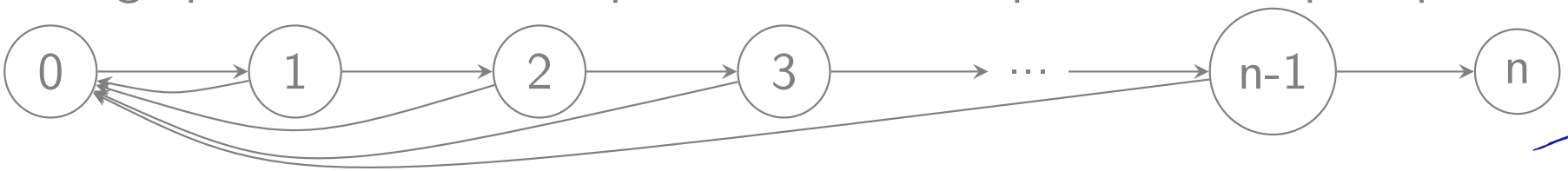
Sortir d'un labyrinthe

Problème st-UCONN

Soit un graphe non orienté G et deux nœuds s et t, trouver un chemin de s à t.

- Parcours en profondeur de G : temps et espace polynomiaux
- Parcours aléatoire : choisir aléatoirement la prochaine arête, sans mémoire : temps polynomial, espace logarithmique (numéro du nœud courant). En $O(n^3)$ pas, il y a une forte probabilité d'avoir parcouru toutes les arêtes (1979).
- En fait, il existe un algorithme déterministe polynomial en espace logarithmique (2005).

Si le graphe est orienté, un parcours aléatoire prend un temps exponentiel



La classe **BPP**

Bounded error Probabilistic Polynomial time

Classe **BPP**

Un langage $L \subseteq \{0,1\}^*$ appartient à la classe de complexité **BPP** $\triangleq \exists M$ une machine de Turing probabiliste et un polynôme $s(n)$ tel que pour tout mot $x \in \{0,1\}^*$:

- $M(x)$ s'arrête en au plus $s|x|$ transitions quels que soient les choix aléatoires
- $x \in L \implies Pr[M(x) \text{ accepte}] \geq 2/3$
- $x \notin L \implies Pr[M(x) \text{ rejette}] \geq 2/3$

Pourquoi $2/3$? Toute constante $> 1/2$ est équivalente (à suivre).

La classe **BPP**, en terme de certificat

Classe **BPP**

Un langage $L \subseteq \{0,1\}^*$ appartient à la classe de complexité **BPP** $\triangleq \exists M$ une machine de Turing polynomiale, $\exists p$ un polynôme :

- $\forall x \in L : Pr_{y \in \{0,1\}^{p(|x|)}}[M(\langle x, y \rangle) \text{ accepte}] \geq 2/3$.
- $\forall x \notin L : Pr_{y \in \{0,1\}^{p(|x|)}}[M(\langle x, y \rangle) \text{ rejette}] \geq 2/3$.

Le certificat y est la suite des choix aléatoires.
$Pr_{x \in E}[A] \triangleq$ probabilité de A selon la distribution uniforme sur E. On prend $\frac{1}{p(|x|)}$ pour y, traduisant que tous les choix sont équiprobables.

La classe **RP**

Randomized Probabilistic Polynomial time

Classe **RP**

Un langage $L \subseteq \{0,1\}^*$ appartient à la classe de complexité **RP** $\triangleq \exists M$ une machine de Turing probabiliste et un polynôme $s(n)$ tel que pour tout mot $x \in \{0,1\}^*$:

- $M(x)$ s'arrête en au plus $s|x|$ transitions quels que soient les choix aléatoires
- $x \in L \implies Pr[M(x) \text{ accepte}] \geq 2/3$
- $x \notin L \implies Pr[M(x) \text{ rejette}] = 1$

La machine ne se trompe que par faux négatifs (rejet erroné d'un mot dans L).
Pourquoi 2/3 ? Pour faire comme **BPP** : toute constante > 0 est équivalente !

Réduction de l'erreur pour **RP**

Réduction de l'erreur

Pour tout langage $A \in \mathbf{RP}$ et pour toute constante c, $\exists M$ une machine de Turing polynomiale probabiliste tel que pour tout mot $x \in \{0,1\}^*$:

- $x \in L \implies Pr[M(x) \text{ accepte}] \geq 1 - 2^{-c}$
- $x \notin L \implies Pr[M(x) \text{ rejette}] = 1$

Pour réduire l'erreur à $\leq 2^{-c}$, il suffit de recommencer l'algorithme c fois. Si l'une des exécutions accepte, alors $x \in A$, sinon on décrète que $x \notin A$. La probabilité de se tromper est $(1/3)^c$, qui est $\leq 2^{-c}$.

On peut réduire la probabilité d'erreur de manière exponentielle : peu de répétitions suffisent à avoir une probabilité d'erreur négligeable.

Réduction de l'erreur pour **BPP**

Réduction de l'erreur

Pour tout langage $A \in \mathbf{BPP}$ et pour toute constante c, $\exists M$ une machine de Turing polynomiale probabiliste tel que pour tout mot $x \in \{0,1\}^*$:

- $x \in L \implies Pr[M(x) \text{ accepte}] \geq 1 - 2^{-c}$
- $x \notin L \implies Pr[M(x) \text{ rejette}] \geq 1 - 2^{-c}$

Même idée : répéter plusieurs fois et garder la réponse majoritaire.
(le calcul exact fait intervenir un peu de théorie des probabilités (bornes de Chernoff). Toute valeur $> 1/2$ fonctionne pour définir **BPP**)

Test de primalité dans **BPP**

Primalité de N

Choisir A tel que $1 \leq A < N$.
Si $pgcd(N, A) > 1$ ou $(\frac{N}{A}) \neq A^{(N-1)/2} \pmod N$ alors conclure composite, sinon conclure premier.

- N premier $\Rightarrow$ l'algorithme conclut correctement
- N composite $\Rightarrow$ l'algorithme se trompe avec une probabilité $1/2$.

On améliore la probabilité par répétition du test.
(Il existe plusieurs algorithmes probabilistes de primalité, peu coûteux en général. Un résultat inattendu de 2002 a montré que le test de primalité est dans **P**, avec un algorithme polynomial déterministe mais peu performant)

Égalité de polynômes

PIT *Polynomial Identity Testing*

Soit F un corps fini (par exemple les entiers modulo 5). Soit $p, q : F^n \mapsto F$ deux polynômes de degré au plus d.
PIT $\triangleq \forall x_1, ..., x_n \in F : p(x_1, ..., x_n) = q(x_1, ..., x_n)$.

(Version simplifiée de PIT : l'énonce formel est sur les entiers non bornés, l'algorithme choisit une valeur m et tous les calculs sont modulo m)

Idée de l'algorithme

Tester la valeur de $p(n) - q(n)$ en un point choisi aléatoirement :

- Si le résultat est non nul, on est sûr de devoir rejeter.
- Sinon, il y a une probabilité $\frac{d}{|F|}$ d'être tombé sur une racine du polynôme, puisqu'il y en a au plus d.

Problème à solution dans **BPP** pour lequel on ne connaît pas d'algorithme déterministe polynomial.

Hiérarchie

$$\mathbf{P} \subseteq \mathbf{RP} \subseteq \mathbf{BPP} \subseteq \mathbf{PSPACE}$$
$$\mathbf{RP} \subseteq \mathbf{NP}$$

- $\mathbf{P} \subseteq \mathbf{RP}$: $\mathbf{P}$ = pas d'erreur
- $\mathbf{RP} \subseteq \mathbf{BPP}$: $\mathbf{RP}$ = moins d'erreur
- $\mathbf{BPP} \subseteq \mathbf{PSPACE}$: d'après la simulation d'une MT non déterministe : évaluer la MT selon tous les chemins possibles et évaluer la probabilité que le mot soit accepté
- $\mathbf{RP} \subseteq \mathbf{NP}$: Dans $\mathbf{RP}$, $x \in L \Rightarrow \exists y, M(\langle x, y \rangle)$ accepte et $x \notin L \Rightarrow \forall y, M(\langle x, y \rangle)$ rejette, ce qui est la caractérisation existentielle de $\mathbf{NP}$ en terme de certificat.
- On ne sait pas la relation entre $\mathbf{BPP}$ et $\mathbf{NP}$ (dans aucun sens).

On soupçonne $\mathbf{RP} = \mathbf{BPP} = \mathbf{P}$.
$\rightarrow$ les algorithmes probabilistes ne permettraient pas de résoudre plus de problèmes que les algorithmes déterministes, mais ils peuvent être plus rapides d'un facteur arbitraire (avec une probabilité d'erreur).

Bilan

- Classe **BPP** = problèmes à résolution probabiliste en temps polynomial
- Classe **RP** = sous-classe de **BPP** pour les problèmes sans faux positifs
- La répétition améliore drastiquement la probabilité de succès
- Beaucoup d'interrogations subsistent (dont le lien entre **BPP** et **NP** et si **BPP** = **P**)
- Même si **BPP** = **P** (l'aléatoire ne permettrait pas de résoudre plus de problème que le déterminisme), nous avons des algorithmes probabilistes qui sont beaucoup plus rapides que les algorithmes déterministes connus (quand il existe), au pris d'une petite probabilité d'erreur.

Cinquième partie

Conclusion

Conclusion

Bilan

TODO