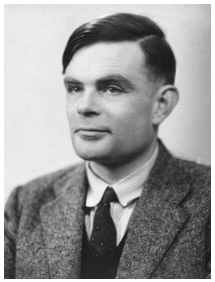
Du calcul efficace à la vérification efficace, et vice versa

Adam Shimi

20 mars 2019

Plan

Etat des lieux : la calculabilité


Modèle universel :
machine de Turing

Problèmes calculables

Problèmes incalculables

La calculabilité suffit-elle pour étudier les algorithmes ?

Problème de l'omniscience logique

Axiomes de l'arithmétique de Peano

- $0 \in \mathbb{N}$.
- $=$ est une relation d'équivalence sur $\mathbb{N}$.
- $\forall n \in \mathbb{N} : S(n) \in \mathbb{N}$.
- $\forall n, m \in \mathbb{N} : S(n) = S(m) \iff n = m$.
- $\forall n \in \mathbb{N} : (S(n) \neq 0)$.
- $[\phi(0) \wedge (\forall n \in \mathbb{N} : \phi(n) \implies \phi(S(n)))] \implies \forall n \in \mathbb{N} : \phi(n)$.

Problème de l'omniscience logique

Axiomes de l'arithmétique de Peano

- $0 \in \mathbb{N}$.
- $=$ est une relation d'équivalence sur $\mathbb{N}$.
- $\forall n \in \mathbb{N} : S(n) \in \mathbb{N}$.
- $\forall n, m \in \mathbb{N} : S(n) = S(m) \iff n = m$.
- $\forall n \in \mathbb{N} : (S(n) \neq 0)$.
- $[\phi(0) \wedge (\forall n \in \mathbb{N} : \phi(n) \implies \phi(S(n)))] \implies \forall n \in \mathbb{N} : \phi(n)$.

Maintenant, vous connaissez tous les théorèmes de la théorie des nombres...

Complexité : le temps

À la recherche d'une ressource à mesurer

Beaucoup de possibilités :

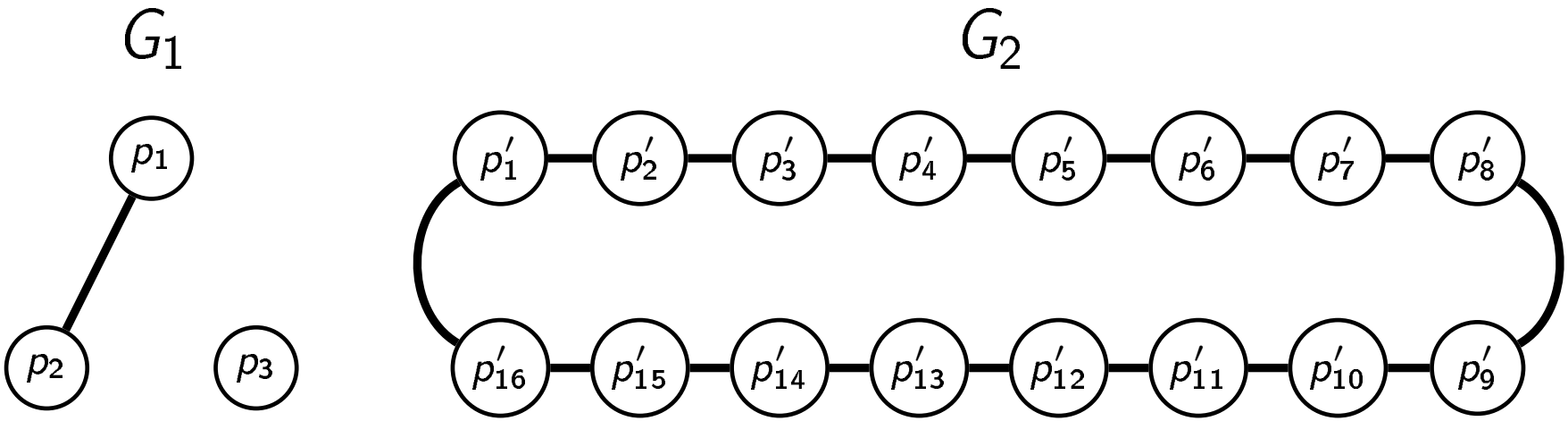
- Le temps de calcul
- La mémoire utilisée
- L'énergie consommée
- L'aléatoire nécessaire

Notre premier choix : le temps

On définit le temps de calcul comme **le nombre de pas de la machine de Turing utilisée**.

Subtilités sur le temps de calcul (1/3)

Prenons le problème de connectivité.



La connectivité de G_1 prend moins de temps à vérifier que de simplement "lire" G_2.

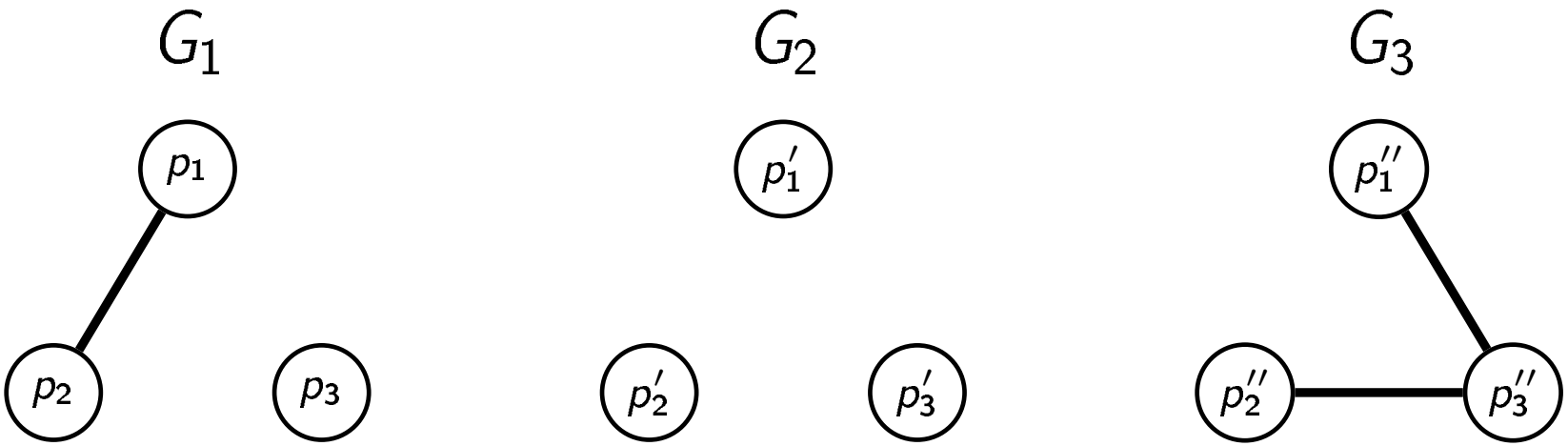
Fonction de la taille

La taille de l'entrée dicte le nombre de pas minimum

$\Longrightarrow$ **mesure du temps = fonction de la taille de l'entrée.**

Subtilités sur le temps de calcul (2/3)

Toujours le problème de connectivité.



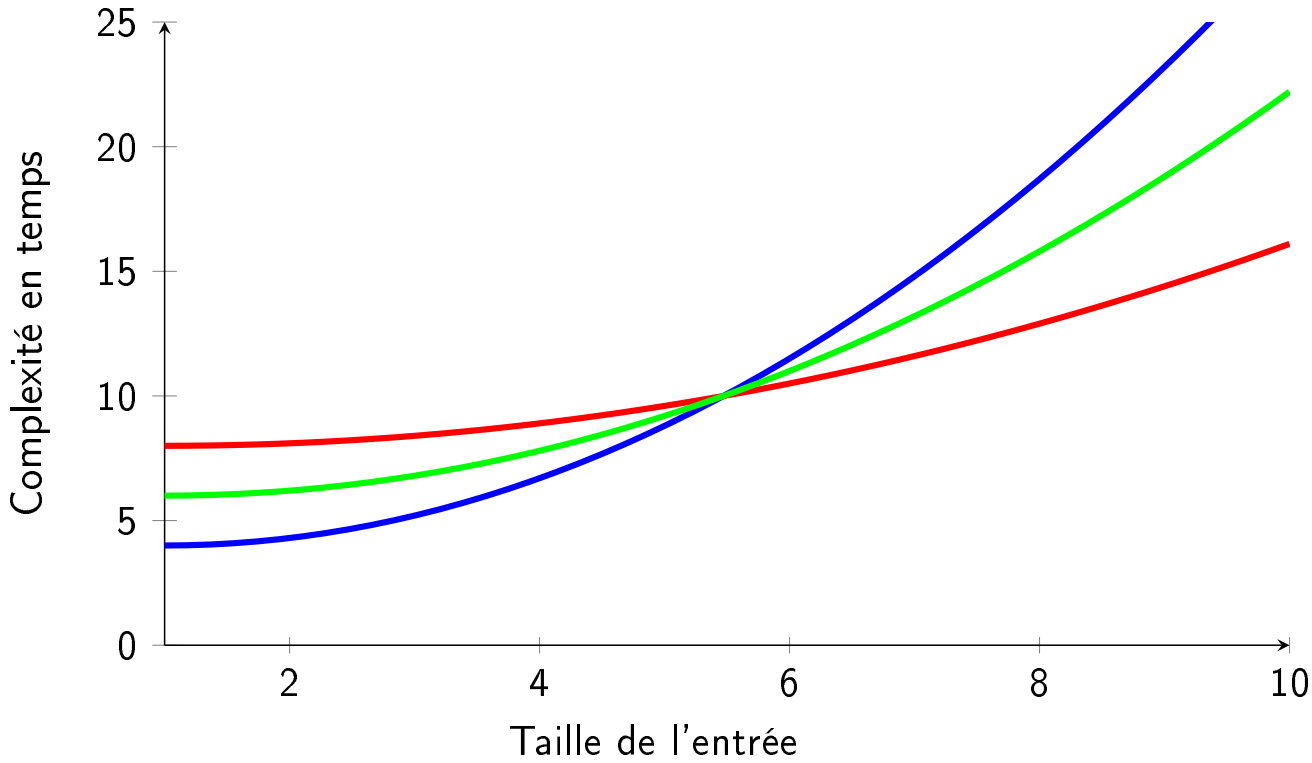
Lequel de G_1, G_2 ou G_3 est le plus "probable", le plus "courant" ?

Pire temps possible

Difficile de savoir quelle entrée de taille n est plus probable

$\Longrightarrow$ **mesure de temps = fonction de la taille de l'entrée vers le pire temps de calcul pour une instance de cette taille.**

Subtilités sur le temps de calcul (3/3)



Notation asymptotique

Pas moyen de comparer "exactement" les complexités en temps

$\Longrightarrow$ **On compare le comportement quand** $n \to \infty$.

Subtilités sur le temps de calcul (Résumé)

Fonction de la taille

La taille de l'entrée dicte le nombre de pas minimum

$\Longrightarrow$ **temps = fonction de la taille de l'entrée.**

Pire temps possible

Difficile de savoir quelle entrée de taille n est plus probable

$\Longrightarrow$ **temps = pire temps de calcul possible.**

Notation asymptotique

On compare le comportement des fonctions quand $n \to \infty$.

- $f(n) = O(g(n)) \iff \begin{cases} \exists M \in \mathbb{R}^{+*}, \exists n_0 \in \mathbb{N}^* : \\ \forall n \geq n_0 : f(n) \leq M g(n) \end{cases}$
- $f(n) = o(g(n)) \iff \begin{cases} \forall \epsilon \in \mathbb{R}^{+*}, \exists n_0 \in \mathbb{N}^* : \\ \forall n \geq n_0 : f(n) \leq \epsilon g(n) \end{cases}$

Choix de problèmes

Existence, recherche et optimisation

Soit G un graphe. On peut formuler plusieurs questions sur G :

- **(Problème de décision)** G est-il k-coloriable ? (coloriable avec k couleurs sans que les voisins partagent leur couleur)
- **(Problème de recherche)** Trouver un k-coloriage de G.
- **(Problème d'optimisation)** Trouver un coloriage pour le plus petit k tel que G est k-coloriable.

Dans ce cours, nous nous limiterons aux problèmes de décision.

Pourquoi privilégier la décision ?

- La plupart des résultats portent dessus.
- Plus simple à étudier mathématiquement.
- Dans de nombreux cas, un problème de recherche se réduit au problème de décision correspondant (pareil pour optimisation).

Classes de complexité : $\mathcal{DTIME}$

Détail préliminaire : constructibilité en temps

Une fonction $f : \mathbb{N} \mapsto \mathbb{N}$ est **constructible en temps** $\triangleq \exists M$ une machine de Turing qui calcule $f(n)$ en temps $O(f(n))$.
Exemple : n^2 est constructible en temps, mais pas $log(n)$.
On se limitera aux fonctions constructibles en temps pour les bornes supérieures.

Définition

Un langage $L \subseteq \{0,1\}^*$ appartient à la classe de complexité $\mathcal{DTIME}(f(n)) \triangleq \exists M$ une machine de Turing :

- $\forall x \in \{0,1\}^* : M$ prend $O(f(n))$ pas pour décider x.
- $\forall x \in L : M$ accepte x.
- $\forall x \notin L : M$ rejette x.

Théorème de hiérarchie temporelle

Hiérarchie temporelle

Soit f, g constructibles en temps avec $f(n)log(f(n)) = o(g(n))$.
Alors $\mathcal{DTIME}(f(n)) \subsetneq \mathcal{DTIME}(g(n))$.

- Il faut $c_M log(t)t$ pas pour simuler avec une machine universelle U une machine M sur une entrée x, où c_M est une constante dépendant de M.
- On peut construire grâce à cette idée un langage décidable en temps $O(g(n))$ mais pas en temps $O(f(n))$: les $(\langle M \rangle, x)$ tels que U rejette ou n'a pas fini après avoir simulé $M((\langle M \rangle, x))$ pendant $g(n)$ pas (de simulation).
- S'il existait M décidant ce langage en $O(f(n))$, alors pour x suffisament grand, $g(n) > c_M.c.log(f(n)).f(n)$, et $M((\langle M \rangle, x)) = 1$ ssi M rejettait $(\langle M \rangle, x)$, et donc ssi $M((\langle M \rangle, x)) = 0$. Contradiction.

Plan

Intuitions derrière l'idée de calcul efficace

Qu'est-ce que l'on attend d'un calcul "efficace" ?

- Meilleur que la recherche exhaustive en force brute.
- Composer deux calculs efficaces conserve l'efficacité.
- Aussi indépendant que possible du modèle de calcul.
- Capture les problèmes que l'on sait résoudre efficacement en pratique.

La classe $\mathcal{P}$

Définition

La classe de complexité $\mathcal{P} \triangleq \bigcup_{c \in \mathbb{N}} \mathcal{DTIME}(n^c)$

Problèmes intéressants dans $\mathcal{P}$

- Vérification de multiplication de matrices
- Test de primalité
- Existence d'un chemin entre deux nœuds d'un graphe
- Evaluation d'un circuit logique à partir des entrées.

La classe $\mathcal{P}$ capture-t-elle les calculs efficaces ?

Oui

+ Force brute est exponentielle, donc $\mathcal{P}$ ne contient pas les problèmes qui n'ont qu'une solution par force brute.
+ $\mathcal{P}$ est fermée par composition.
+ Toutes les réductions d'un modèle de calcul à un autre sont en temps polynomial, $\mathcal{P}$ est indépendante du modèle de calcul.
+ Les problèmes connus dans $\mathcal{P}$ ont un algorithme efficace en pratique.

Non

– Pire temps n'est pas forcément représentatif des cas pratiques.
– Ne prend pas en compte les constantes ni la valeur de la puissance de n.

Plan

De l'intérêt des preuves

Que demander d'une telle procédure de vérification ?

- Ne jamais accepter une mauvaise solution.
- Pouvoir être convaincu d'accepter une solution correcte.
- Qu'elle soit efficace, c'est-à-dire polynomiale.

La classe $\mathcal{NP}$

Définition

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

- $\forall x \in L, \exists y \in \{0,1\}^{p(|x|)}$: 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**.

Problèmes intéressants dans $\mathcal{NP}$

- Satisfiabilité d'une formule propositionnelle
- k-coloriabilité d'un graphe
- Problème du sac-à-dos

Définition alternative de $\mathcal{NP}$

Définition alternative

Un language $L \subseteq \{0,1\}^*$ appartient à la classe de complexité $\mathcal{NP} \triangleq \exists M$ une machine de Turing **non-déterministe** polynomiale :

- $\forall x \in L : M$ accepte x.
- $\forall x \notin L : M$ rejette x.

Equivalence entre les deux définitions

- $\exists$ une machine non-déterministe $\implies$ le certificat note les choix non-déterministes successifs qui font accepter.
- $\exists$ une machine de vérification $\implies$ on construit une machine non-déterministe qui fait autant de choix booléens que la taille du certificat, puis utilise cette suite de choix comme certificat.

Distinction entre calculer et vérifier

Calculer n'est pas exactement vérifier

- $\mathcal{P}$ se concentre sur la résolution d'un problème de décision, c'est-à-dire à calculer si l'entrée est une instance du langage à reconnaitre.
  Exemple : calculer si un graphe est connecté.
- $\mathcal{NP}$ parle de vérification, c'est-à-dire de la validation d'un certificat (une "preuve") montrant que l'entrée est bien une instance du langage à reconnaitre.
  Exemple : vérifier qu'un graphe est connecté si le certificat est un chemin reliant tous les nœuds.

$\mathcal{P} \subseteq \mathcal{NP}$: pour un problème dans P, on peut juste vérifier un certificat y pour x en calculant la solution pour x... sans regarder y.
Par contre, $\mathcal{NP} \subseteq \mathcal{P}$? est un problème ouvert. (voir la fin du cours)

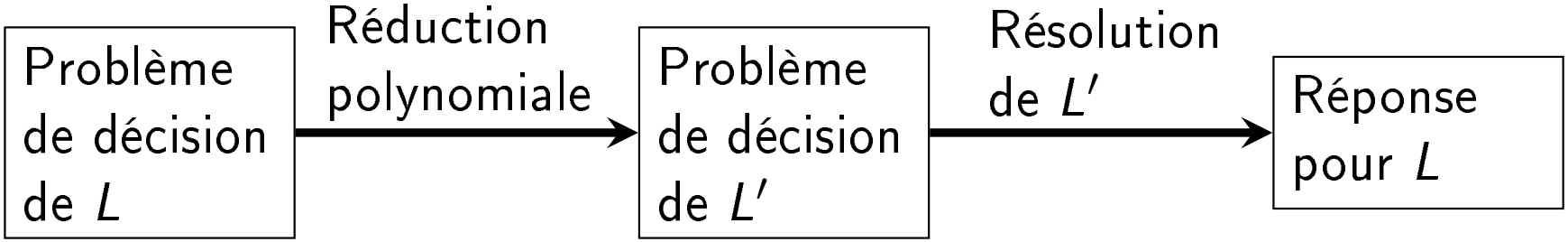
Réductions d'un problème à un autre

Rappel : fonction calculable

Une fonction calculable est une fonction $f : \{0,1\}^* \mapsto \{0,1\}^*$ telle que $\exists M$ une machine de Turing qui prend une entrée $x \in \{0,1\}^*$ et retourne $f(x)$.

Réduction polynomiale d'un problème de décision à un autre

Soit L et L' deux langage dans $\{0,1\}^*$. On dit que L est réductible en temps polynomial à L' s'il existe une fonction f calculable en temps polynomial telle que $\forall x \in \{0,1\}^* : x \in L \iff f(x) \in L'$.



$\mathcal{NP}$-complétude

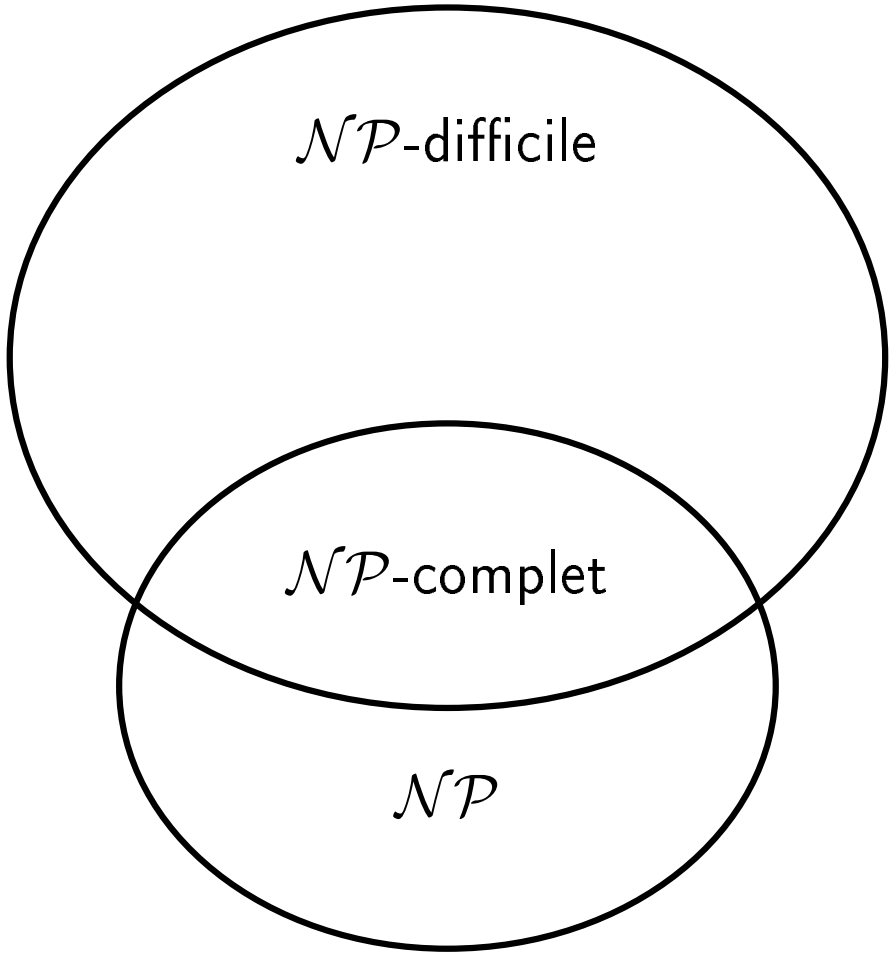
Soit L un problème de décision.

Problème $\mathcal{NP}$-difficile

L est $\mathcal{NP}$-**difficile** $\triangleq$
$\forall L' \in \mathcal{NP}$: L' est réductible en temps polynomial à L.
Intuition : aussi difficile que n'importe quel problème dans $\mathcal{NP}$.

Problème $\mathcal{NP}$-complet

L est $\mathcal{NP}$-**complet** $\triangleq$
L est $\mathcal{NP}$-difficile et $L \in \mathcal{NP}$.
Intuition : L est parmi les plus difficiles de tous les problèmes dans $\mathcal{NP}$



Pourquoi s'intéresser aux problèmes complets ?

Ils capturent la classe tout entière

Si l'on prouve qu'un problème complet pour une classe $\mathcal{A}$ appartient à la classe $\mathcal{B}$, cela nous donne $\mathcal{A} \subseteq \mathcal{B}$.
Et 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 $\mathcal{P}$ et $\mathcal{NP}$), 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.

Le problème SAT

Définition

SAT (ou satisfiabilité booléenne) est le langage formé par toutes les formules propositionelles satisfiables, càd 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 : $C_1 \wedge C_2 \wedge ... \wedge C_k$, où chaque C_i est de la forme $x_{i_1} \vee ... \vee x_{i_q}$.

SAT est dans $\mathcal{NP}$

Il "suffit" de donner une valuation des variables, et l'on vérifie la valeur de la formule en temps polynomial dans la taille de la formule.

SAT est $\mathcal{NP}$-complet (Intuition)

On veut réduire tout problème dans $\mathcal{NP}$ à SAT.

Approche générale

Soit L un langage dans $\mathcal{NP}$ et $x \in \{0,1\}^*$.

- Il existe une machine de Turing polynomiale M qui accepte la paire x et un certificat y ssi $x \in L$. M est déterministe : avec x et y fixés, le calcul est complètement déterminé.
- On représente le calcul de $M(\langle x, y \rangle)$ comme une formule dépendant uniquement de y, avec une taille bornée par un polynome en $|x|$.
- On ajoute la contrainte que le calcul termine en acceptant en temps polynomial.

De cette façon, vérifier la satisfiabilité de cette formule revient à vérifier l'existence d'un certificat y pour x faisant accepter M.

SAT est $\mathcal{NP}$-complet (Détails 1/3)

Etat : Start

0	1	2	3	4	5	⋯
1	0	0	1	1	0	⋯

Exemples :

$x_{1,0} = 1; x_{1,5} = 0$

$p_{1,0} = 1; p_{1,3} = 0$

$s_{1,Start} = 1$

Variables

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

SAT est $\mathcal{NP}$-complet (Détails 1/3)

Etat : Start

0	1	2	3	4	5	⋯
1	0	0	1	1	0	⋯

Exemples :

$x_{1,0} = 1; x_{1,5} = 0$

$p_{1,0} = 1; p_{1,3} = 0$

$s_{1,Start} = 1$

Variables

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

SAT est $\mathcal{NP}$-complet (Détails 1/3)

Etat : Start

0	1	2	3	4	5	⋯
1	0	0	1	1	0	⋯

Exemples :

$x_{1,0} = 1; x_{1,5} = 0$

$p_{1,0} = 1; p_{1,3} = 0$

$s_{1,Start} = 1$

Variables

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

SAT est $\mathcal{NP}$-complet (Détails 1/3)

Etat : Start

0	1	2	3	4	5	⋯
1	0	0	1	1	0	⋯

Exemples :

$x_{1,0} = 1; x_{1,5} = 0$

$p_{1,0} = 1; p_{1,3} = 0$

$s_{1,Start} = 1$

Variables

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

SAT est $\mathcal{NP}$-complet (Détails 2/3)

Sous-formules

- Un seul état à la fois : $\forall t > 0, \forall k \neq k' : s_{t,k} \implies \neg s_{t,k'}$
- Une seule position de la tête à la fois :
  $\forall t > 0, \forall j \neq j' : p_{t,j} \implies \neg p_{t,j'}$
- La seule case qui peut changer est celle où se trouve la tête :
  $\forall t > 0, \forall j \in \mathbb{N} : \neg p_{t,j} \implies (x_{t+1,j} = x_{t,j})$.
- Si on lit b sur la case j dans l'état k, alors on écrit b', la tête bouge sur la case j' et on passe dans l'état k' :
  $s_{t,k} \wedge p_{t,j} \wedge (x_{t,j} = b) \implies s_{t+1,k'} \wedge p_{t+1,j'} \wedge (x_{t+1,j} = b')$.
- Le calcul accepte en temps polynomial :
  $$\bigvee_{0<t<poly(|x|)} (s_{t,Halt} \wedge x_{t,0})$$
- Et d'autres...

SAT est $\mathcal{NP}$-complet (Détails 3/3)

Pourquoi la formule est-elle de taille polynomiale ?

Comme $L \in \mathcal{NP}$, la machine M qui vérifie les certificats pour x a son pire temps de calcul borné par un polynome $T(n)$.
$\Longrightarrow$ Il suffit donc de considérer $T(|x|)$ cases et $T(|x|)$ étapes de calcul, donc un nombre polynomial de configurations.

- Il y a un nombre polynomial de variables par configuration, et donc un nombre polynomial de variables au total.
- Nous avons un nombre polynomial de sous-formules avec un nombre polynomial de variables, ce qui donne une formule générale de taille polynomiale.

On a donc construit une formule de taille polynomiale en $|x|$ (donc constructible en temps polynomial en $|x|$) telle que cette formule est satisfiable ssi il existe un certificat y qui fait accepter M.
Donc L est réductible à SAT en temps polynomial. QED.

Réduction de SAT à 3SAT

Définition

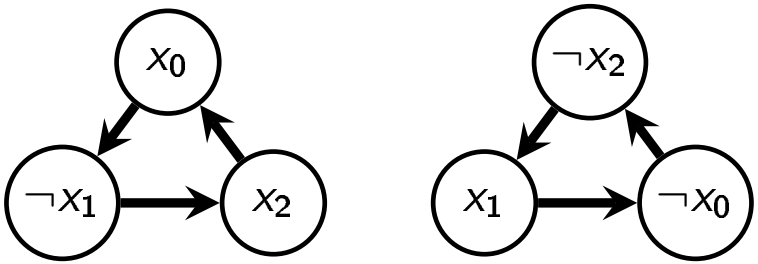
3SAT est le sous-langage de SAT formée par les formules propositionelles satisfiables en forme normale conjonctive $(C_1 \wedge ... \wedge C_k)$, où chaque clause est une disjonction d'au plus 3 variables.
Exemple : $(x_1 \vee \neg x_2 \vee x_4) \wedge (x_3 \vee x_2)$.

Réduire SAT à 3SAT

On transforme chaque clause $C = x_1 \vee ... \vee x_n$ qui contient $n > 3$ valeurs en la conjonction de deux clauses
$C_1 = (\neg(x_1 \vee ... \vee x_{n-2}) \implies z) = x_1 \vee ... \vee x_{n-2} \vee z$ et
$C_2 = (z \implies (x_{n-1} \vee x_n)) = \neg z \vee x_{n-1} \vee x_n$.
Répèter cette transformation sur C_1 jusqu'à n'avoir plus que des clauses de taille 3.

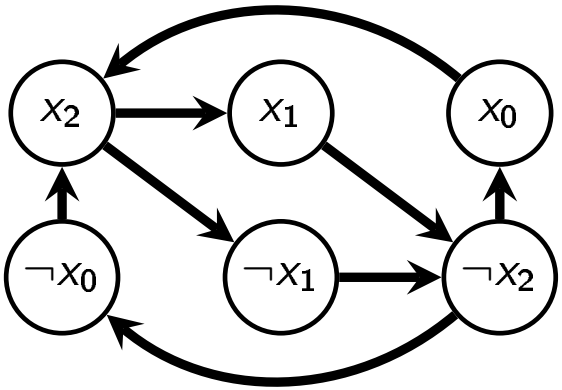
Par contre, 2SAT est dans $\mathcal{P}$

$(x_0 \vee \neg x_2) \wedge (x_1 \vee x_2) \wedge (\neg x_0 \vee \neg x_1)$



Satisfiable

$(x_0 \vee x_2) \wedge (\neg x_0 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_2)$



Non-satisfiable

On utilise $(x \vee y)$

$\iff (\neg x \implies y)$

$\iff (\neg y \implies x)$

Algorithme 1 : Algorithme Polynomial pour 2SAT

Construire graphe des implications G.

si $\exists x$: *G contient un cycle passant par x et $\neg x$.* **alors**

  Retourner **Faux**.

fin

Retourner **Vrai**.

Plus de problèmes NP-complets

Liste de problèmes NP-complets

- **(Voyageur de Commerce)** Etant 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 ? Applications : planning, logistique, séquençage ADN,...
- **(Sac à dos)** Etant donné des objets avec valeur et coût, une limite de coût et un objectif, existe-t-il un sous-ensemble d'objets qui atteint l'objectif sans dépasser le coût ? Applications : investissements, chargement de containers,...
- **(Coloriage d'un graphe)** (pour $k \geq 3$ couleurs) Etant donné un graphe, existe-t-il un coloriage de ses nœuds avec k couleurs tel que personne ne partage la couleur de ses voisins ? Applications : ordonnancement, allocation de fréquences,...

Plan

$\mathcal{P} = \mathcal{NP}$ ou $\mathcal{P} \neq \mathcal{NP}$?

Le problème du millénaire

La question la plus fondamentale de l'informatique est : $\mathcal{P} = \mathcal{NP}$ ou $\mathcal{P} \neq \mathcal{NP}$.
En reformulant, est-ce que tous les problèmes dont on peut vérifier efficacement la solution sont résolvables par un algorithme efficace ?

La majorité des chercheurs en théorie de la complexité croient que $P \neq NP$. Parmi les raisons avancées :

- Depuis les années 50, beaucoup, beaucoup de gens ont cherché des algorithmes polynomiaux pour les milliers de problèmes NP-complets, et personne n'a rien trouvé.
- Intuitivement, vérifier semble demander bien moins d'efforts que de trouver une solution. Un peu comme la différence entre apprécier un film et le créer.

Que faire devant un problème $\mathcal{NP}$-complet ?

- **(Force Brute)** Si l'instance est petite.
- **(Cas Spéciaux)** Peut-être que tous les cas qui vous intéressent sont contenus dans un sous-problème solvable efficacement. Par exemple 2SAT.
- **(Approximation)** Il existe de nombreux algorithmes d'approximations pour les problèmes NP-complets. Dans certains cas, ils sont même très précis et efficaces, comme ceux pour le voyageur de commerce.
- **(Heuristiques)** Beaucoup d'astuces permettent en pratique d'éviter les cas les plus problématiques.

Ressources

Pour ceux qui veulent approfondir, ou voir des preuves détaillées :

- *Automata, Computability and Complexity*, Cours 6.045 au MIT, Scott Aaronson, 2016 `http://stellar.mit.edu/S/course/6/sp16/6.045/materials.html`
- *Computational Complexity : A Modern Approach*, Sanjeev Arora and Boaz Barak, Cambridge University Press, 2009 (version préliminaire : `http://theory.cs.princeton.edu/complexity/`) Disponible à la BU
- *Computational Complexity : A Conceptual Perspective*, Oded Goldreich, Cambridge University Press, 2008, Disponible à la BU
- *Complexité algorithmique*, Sylvain Perifel, Ellipses, 2014 `https://www.irif.fr/~sperifel/complexite.pdf`, Disponible à la BU