Calculabilité et Complexité (11X008) - Printemps 2026

12. Compléments et ouverture

Enseignant: Arnaud Casteigts

Assistants: M. De Francesco, M. Marseloo
Moniteurs: E. Bussod, N. Beghdadi

Dans ce dernier cours, nous évoquons plusieurs sujets d'ouverture en complexité algorithmique. On y parlera notamment d'auto-réductibilité, de PSPACE-complets, de la hiérarchie polynomiale, de preuves interactives, de preuves à divulgation nulle de connaissance et d'ordinateur quantique. On terminera par une application pratique du problème SAT.

12.1 Cook-Levin (complément)

Rappels des raisons pour lesquelles ce théorème implique que SAT est NP-difficile. En fait, on peut voir la construction de la semaine dernière comme une réduction de tout langage de NP vers SAT : par définition, si un langage L est dans NP, alors il existe une machine M non-déterministe qui le décide en temps polynomial. Pour n'importe quelle entrée w, en utilisant la description de cette machine, on peut construire en temps polynomial une formule ϕ de taille polynomiale en $|w|$ qui est satisfaisable si et seulement si $M(w)$ accepte. Cette formule encode la question de l'existence d'une branche acceptante dans l'exécution de $M(w)$. Il en résulte que si on peut décider SAT en temps polynomial, alors on peut décider n'importe quel langage de NP en temps polynomial : SAT est NP-difficile.

12.2 Auto-reductibilité

Pour rappel, on existe différents types de problèmes. Par exemple pour Clique :

- Décision : Étant donné (G, k), existe-t-il une clique de taille k dans G ?
- Recherche : Étant donné (G, k), trouver une clique de taille k dans G.
- Optimisation : Étant donné G, trouver une clique de taille maximum dans G.
- Dénombrement : Étant donné G et k, combien y a-t-il de cliques de taille k dans G.

On dit qu'un problème est auto-réductible si l'accès à un oracle pour la version "décision" permet de résoudre aussi la version "recherche" en temps polynomial. Un oracle pour un langage L est une machine fictive capable de décider L instantanément.

Exemple : Clique est auto-réductible. Supposons que l'on a accès à un oracle pour ce langage : pour tout (G, k), on peut savoir immédiatement si G admet une clique de taille k

(autrement dit, décider si (G, k) est une instance positive). On peut l'utiliser comme suit pour trouver une clique de taille k :

1. Demander à l'oracle si (G, k) est une instance positive et rejeter si ce n'est pas le cas.
2. Pour chaque sommet v de G, on demande à l'oracle si $(G \setminus \{v\}, k)$ est une instance positive. Si la réponse est oui, cela signifie qu'il existe une clique de taille k qui ne contient pas v. On enlève donc ce sommet et on continue.
3. Lorsqu'on a terminé, le graphe restant est une clique de taille k.

Cette procédure va faire (au plus) un appel à l'oracle pour chaque sommet et nous permettre in fine d'identifier les sommets qui font partie de la clique. On peut donc trouver une clique rapidement en faisant $O(n)$ appels à l'oracle pour un graphe à n sommets.

Il en va de même pour la version d'optimisation : on peut commencer par poser la question à l'oracle pour différentes valeurs de k avec le graphe G entier, afin de trouver la plus grande valeur k qui fonctionne. Cela coûte $O(\log n)$ appels (en dichotomisant parmi les valeurs possibles de $k \in [1, n]$). Une fois qu'on connaît cette valeur, on utilise la même procédure que précédemment.

En fait, tous les problèmes de NP sont auto-réductibles dans ce sens. Cela a été observé dès le début de la théorie [1] et a fortement justifié le fait de se concentrer sur l'étude des problèmes de décision.

Les versions de "dénombrement" des problèmes sont typiquement plus difficiles. En général, ces versions ne se réduisent pas en temps polynomial à la version décisionnelle. (L'inverse fonctionne par contre, c'est évident, si vous connaissez le nombre de cliques de taille k, vous en déduisez immédiatement l'existence.)

Les versions "dénombrement" des problèmes de NP sont dans une classe de complexité qui s'appelle #P (prononcer sharp P).

12.3 PSPACE-complétude

Pour rappel, un langage est dans PSPACE s'il peut être décidé en utilisant une quantité polynomiale d'espace (quel que soit le temps). Nous avons déjà évoqué que NP $\subseteq$ PSPACE. On peut montrer cela de deux manières :

1. NP $\subseteq$ NPSPACE (même argument que P $\subseteq$ PSPACE), suivi du théorème de Savitch, qui implique que NPSPACE = PSPACE.
2. On peut aussi le montrer par le fait qu'on peut énumérer tous les certificats possibles en espace polynomial et vérifier à chaque fois. Si un certificat existe, on va le trouver.

1. Karp, R. M. (1972). "Reducibility Among Combinatorial Problems." Complexity of Computer Computations, pp. 85-103.

De la même manière que certains problèmes sont NP-complet, on peut définir une notion de problème PSPACE-difficile et PSPACE-complet. Un problème est PSPACE-difficile si n'importe quel problème de PSPACE peut s'y réduire *en temps polynomial*. Il est PSPACE-complet s'il est PSPACE-difficile et dans PSPACE. De nombreux jeux combinatoires sont PSPACE-complets. Par exemple, le problème de décider si un joueur a une stratégie gagnante au jeu *Reversi/Othello* généralisé en dimension $n \times n$ est PSPACE-complet.

12.4 PH (hiérarchie polynomiale)

Décider s'il existe une stratégie gagnante dans la plus part des jeux combinatoires revient à résoudre une question du type :

Existe-t-il un coup pour le joueur 1 tel que *pour tous* les coups du joueur 2, il *existe* un coup pour le joueur 1 tel que *pour tous* les coups du joueur 2, ... le joueur 1 gagne.

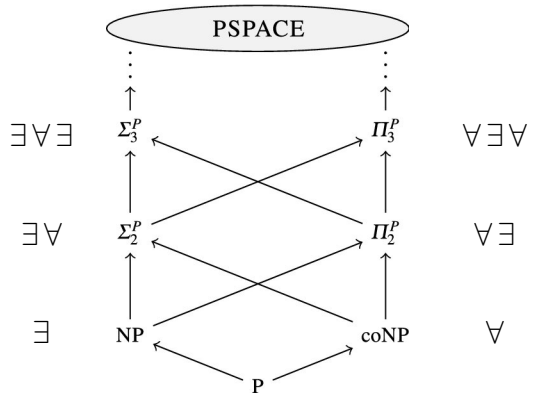
La question est de la forme $\exists x, \forall y, \exists z, \ldots prop(x, y, z, \ldots)$. Intuitivement, les problèmes PSPACE-complets correspondent aux problèmes de cette forme, où le nombre de quantifieurs n'est pas constant : il dépend de n (polynomialement).

En contraste, les problèmes de NP correspondent au fait d'avoir un seul quantifieur $\exists$ et les problèmes de co-NP correspondent au fait d'avoir un seul quantifier $\forall$.

NP : $\exists x, prop(x)$. (il existe une clique de taille k)

coNP : $\forall x, \neg prop(x)$. (aucune clique n'est de taille k)

Entre ces classes et PSPACE-complet, on peut imaginer une infinité de niveau dans lesquels ont limite le nombre de quantifieurs utilisés.



L'union de toutes ces classes intermediaires est appelée PH (*polynomial hierarchy*). Autrement dit, PH contient tous les langages que l'on peut formuler comme une alternance d'un nombre constant de quantifieurs, par exemple : le joueur 1 peut-il gagner en trois coups ?

Un autre exemple de problème naturel est la minimisation de circuit logique. Etant donné un circuit C, *existe*-t-il un circuit plus petit tel que *pour toute* entrée, la valeur de sortie est identique à celle de C ?

Au niveau des relations entre classes, cela donne :

$$\mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PH} \subseteq \mathsf{PSPACE}$$

Rappelez-vous qu'on n'arrive même pas à prouver que $\mathsf{P} \neq \mathsf{PSPACE}$, c'est dingue !

12.5 Preuves interactives (et probabilistes)

Nous avons déjà évoqué la difficulté de prouver que deux graphes *ne sont pas* isomorphes : on ne sait pas si Graph Isomorphism est dans co-NP, c'est une question ouverte. Il existe cependant des preuves *probabilistes* et *interactives* de cela : supposons qu'un être super-puissant (Merlin) veuille convaincre une personne normale (Arthur) que deux graphes ne sont pas isomorphes. On peut concevoir la preuve interactive suivante :

- Arthur mélange G_1 et G_2.
- Merlin trouve G_1
- on recommence k fois.

Pour $k = 1$, il y a une chance sur deux pour que ce soit de la chance, mais à mesure qu'on répète l'expérience, le fait que Merlin réussisse à trouver G_1 à chaque fois finira par convaincre Arthur que les deux graphes sont différents. Concrètement, après k répétitions, la probabilité que les graphes soient différents est de $1 - 1/2^k$. Supposons qu'Arthur veuille être sûr à 99% que les graphes sont différent, il suffira de répéter l'expérience 7 fois.

La classe des problèmes qui admettent des preuves interactives avec un nombre polynomial d'interaction permettant à Arthur d'être sûr avec probabilité d'au moins une constante $p > 0.5$ est appelé IP (interactive proofs). Cela est suffisant, car en répétant cela, on peut amplifier la certitude d'Arthur de manière exponentielle (c.f. formule ci-dessus) en conservant un nombre polynomial de répétition. On peut donc dire qu'Arthur sera *rapidement* convaincu.

Un résultat surprenant est le théorème de Shamir (1992), qui montre que $\mathsf{IP} = \mathsf{PSPACE}$.

12.6 Preuves à divulgation nulle de connaissance

Observez qu'avec l'interaction précédente, Arthur devient convaincu que les deux graphes sont différents, mais il n'a absolument rien appris sur les raisons de cette différence. Une

preuve qui permet de convaincre quelqu'un sans qu'il n'apprenne rien d'autre que la véracité de la proposition est appelé une preuve à divulgation nulle de connaissance (zero-knowledge proof). Ces preuves sont très utiles en cryptologie, et au delà. Récemment, Google Quantum AI a publié une telle preuve portant sur un circuit quantique très optimisé, montrant que les ordinateurs quantiques de première génération pourraient casser des clés de cryptographie à courbe elliptique en seulement 9 minutes. Leur preuve *peut être vérifiée*, sans apprendre le moindre détail sur les méthodes d'attaques utilisées, ce qui serait trop risqué.

La classe des langages qui admettent des preuves zero-knowledge est appelée ZK et elle contient notamment NP. Incroyable non ? Pour n'importe quel problème de NP, vous pouvez convaincre quelqu'un que la réponse est oui sans ne rien fuiter de la solution elle-même.

- Cette formule SAT est-elle satisfaisable ? Ce graphe a-t-il une clique de taille k ?...
- Oui
- OK, prouve le moi en me donnant la solution
- Non, j'ai pas envie
- OK, alors je ne te crois pas
- Si, regarde : *preuve interactive sans divulgation de connaissance*
- OK, tu m'as convaincu, et je n'ai rien appris d'autre que le fait que c'est vrai.

12.7 Ordinateurs quantiques

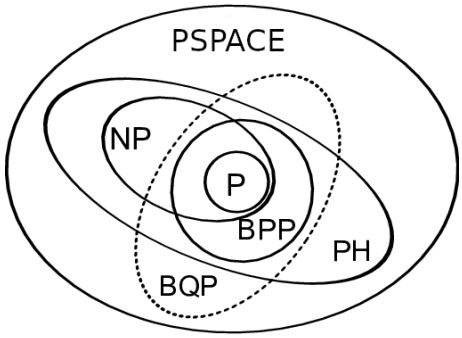
En 1994, Peter Shor a montré qu'un hypothétique ordinateur quantique pourrait factoriser de grande nombres rapidement. Concrètement, il a montré que le problème FACTORING est dans la classe BQP (Bounded-error quantum polynomial), alors que ce problème n'est pas connu pour être dans P.

Les ordinateurs quantiques étant fondamentalement probabilistes, la définition de BQP passe d'abord par celle de BPP :

- BPP (Bounded-error probabilistic polynomial) : Un langage L est dans BPP s'il existe une machine de Turing (un algorithme) utilisant des bits aléatoires qui décide si $w \in L$ avec une probabilité d'erreur bornée par une constante $p < 0.5$. (Ici aussi, on peut amplifier en répétant.)
- BQP : Même chose pour un ordinateur quantique.

Notons que les ordinateurs quantiques n'ont pas d'impact sur la calculabilité, seulement sur la complexité, car ils peuvent être simulés par des machines de Turing classiques. En fait, on est capable d'effectuer cette simulation avec un surcoût d'espace polynomial, donc BQP $\subseteq$ PSPACE. En revanche, on ne sait pas le faire en temps polynomial, donc présumément, BQP contient donc des problèmes qui ne sont pas dans P. En fait, on soupçonne que BQP contient des problèmes qui ne sont même pas dans NP et vice versa, que NP contient des

problèmes qui ne sont pas dans BQP (notamment les problèmes NP-complets). Au final, voici l'état des croyances actuelles :



Et l'on ne sait toujours pas si P = PSPACE ! Si c'est le cas, toutes ces classes sont égales à P.

12.8 Le côté positif de la force

Terminons sur une note positive. Nous avons montré que SAT permet d'*encoder* un grand nombre de problèmes (tous les problèmes de NP), ce qui explique que ce problème (de même que les autres problèmes NP-complets) est "difficile". En réalité, pour tous les problèmes NP-complets, de nombreuses instances issues du monde réel sont faciles à résoudre. D'ailleurs, SAT est très utilisé pour encoder ces problèmes et les résoudre en utilisant des solveurs SAT, qui sont des programmes hyper optimisés qui cherchent à satisfaire des formules SAT.

Un exemple : vous avez une grille Sudoku à résoudre. Nous pouvons utiliser des variables $x_{r,c,i}$ pour chaque ligne (raw), chaque colonne (column) et chaque valeur $i \in [1, 9]$. Par exemple, $x_{2,3,8}$ vaut vrai si la case de la deuxième ligne et troisième colonne vaut 8.

On peut alors encoder les contraintes :

- Chaque case a une valeur : $(x_{1,1,1} \vee \cdots \vee x_{1,1,9}) \wedge (x_{1,2,1} \vee \cdots \vee x_{1,1,9}) \wedge \ldots$.
- Chaque case n'a qu'une valeur : $(\overline{x_{1,1,1}} \vee \overline{x_{1,1,2}}) \wedge (\overline{x_{1,1,1}} \vee \overline{x_{1,1,3}}) \wedge \ldots$
- Chaque valeur apparaît dans chaque ligne.
- Chaque valeur apparaît dans chaque colonne.
- Chaque valeur apparaît dans chaque carré de 3×3.
- Les cases déjà remplies ont la variable correspondante qui est forcée.

Une fois cette grande formule créée, vous pouvez la donner au solveur SAT, qui vous trouvera une solution, en général, en une fraction de seconde.