Leçon 915 : Classes de complexité. Exemples

Julie Parreaux

2018 - 2019

Références pour la leçon

[1] Carton, *Langages formels, calculabilité et complexité.*
[2] Cormen, *Algorithmique.*
[3] Floyd et Biegel, *Le langage des machines.*
[4] Kleinberg et Tardos, *Algorithm design.*
[5] Papadimitriou, *Comptutational complexity.*
[6] Sipser, *Introduction to the Theory of Computation.*

Développements de la leçon

Le problème PSA est NP-complet

Théorème de Savitch

Plan de la leçon

Motivation

Défense

Une première classification des problèmes en informatique est sa décidabilité ou non. Cependant, cette classification est grossière et une question naturelle a été de classifier les problème décidable en fonction de leur dureté : l'accessibilité dans un graphe et Presburger ce n'est pas tout à fait le même problème.

Les machines de Turing sont un outils permettant de réaliser cette classification : elles permettent de définir la complexité d'un problème. Nous allons donc étudier les classes de complexité en temps et en espace (les familles de problèmes qui ont des complexités équivalentes) et leurs hiérarchies. Remarquons que nous nous intéressons à la complexité d'un problème et non d'un algorithme et même si les deux notions sont très liées, l'étude précise d'une implémentation ne nous intéresse pas (on souhaite uniquement mettre le problème dans une case).

Ce qu'en dit le jury

Le jury attend que le candidat aborde à la fois la complexité en temps et en espace. Il faut naturellement exhiber des exemples de problèmes appartenant aux classes de complexité introduites, et montrer les relations d'inclusion existantes entre ces classes, en abordant le caractère strict ou non de ces inclusions. Le jury s'attend à ce que les notions de réduction polynomiale, de problème complet pour une classe, de robustesse d'une classe vis à vis des modèles de calcul soient abordées.

Se focaliser sur la décidabilité dans cette leçon serait hors sujet.

Introduction

Motivation : Classer les problème décidable en fonction de leur difficulté (la difficulté est de définir la difficulté et la notion de problème plus dur qu'un autre).
Cadre : Problème décidable
Pré-requis : Machine de Turing (déterministe, non déterministe, à plusieurs rubans (elles ne servent que pour NL (hors programme), si on en parle pas, on peut tout faire avec un seul ruban)), exécution, configurations

1 Classes de complexité

Pour définir les classes de complexité, nous devons définir les problèmes de décision et surtout soulever le problème du codage des entiers. Ensuite, nous définissons la complexité pour notre modèle de calcul qui sert d'échelon : les machine de Turing (déterministe, non déterministe, à plusieurs ruban).

1.1 Problèmes de décisions

Un des points subtile dans la théorie de la complexité est le codage des entrées de notre problème. En effet, selon le codage choisi, la complexité de notre problème peut être différente (ou la preuve plus ou moins difficile).

— *Définition* : Problème de décision
— *Problème* : Codage des entrées
— *Exemple* : Primes et UnaryPrimes

1.2 Complexité d'une machines de Turing

La complexité d'une machine de Turing est sa consommation d'une ressource donnée lors de son exécution. Cette ressource est temporelle ou spatiale. Comme on ne se concentre que sur des problèmes décidables, on peut supposer que les exécutions de notre machine sont finies. On utilise la taille des configurations : il nous faut faire attention dans le cadre d'une machine à plusieurs rubans.

— *Définition* : Soient $\mathcal{M}$ une machine de Turing, w une entrée et C_w l'ensemble des suites de configurations des exécutions possibles sur w. La complexité d'une exécution c en temps et en espace est $t(c) = n$ et $s(c) = \max_{0 \leq i \leq n} |c_i|$.

— *Remarque* : Dans le cas d'une machine de Turing à plusieurs rubans, la taille des configurations ne prend pas en compte les rubans lecture ou en écriture seul dont la tête de lecture se déplace qu'à droite (on ne compte que les rubans qui ont un rôle de mémoire).

— *Définition* : Soient $\mathcal{M}$ une machine de Turing, w une entrée et C_w l'ensemble des suites de configurations des exécutions possibles sur w. La complexité pour une entrée est alors $t(w) = \max_{c \in C_w} t(c)$ et $s(w) = \max_{c \in C_w} s(c)$.

— *Définition* : Soient $\mathcal{M}$ une machine de Turing, w une entrée et C_w l'ensemble des suites de configurations des exécutions possibles sur w. La complexité d'une machine est alors $t : n = \max_{|w|=n} t(w)$ et $s : n = \max_{|w|=n} s(w)$ (fonction en fonction de la taille de l'entrée).

— *Proposition* (Lien entre espace et temps). Soit $\mathcal{M}$ une machine de Turing, il existe $K \in \mathbb{R}^+$ tel que $s(n) \leq \max(t(n), n)$ et $t(n) \leq 2^{Ks(n)}$.

— *Théorème* : Accélération temporelle et spatiale

1.3 Classes de complexité

Nous allons maintenant définir la complexité d'un problème de décision et les classes de complexités usuelles. On choisit des machines de Turing à une seule bande. Grâce au théorème d'accélération, on sait que les constance ne sont pas nécessaire dans l'étude de la complexité.

— *Définition* : Famille de problème en espace et en temps

— *Définition* : Classes de complexité usuelles

— *Remarque* : Implication de l'encodage (exemple Primes et UnaryPrimes)

— *Exemple* : Logique dans chacune des classes au programme (on commence à poser l'idée que la logique est une bonne porte d'entrée pour résoudre des problèmes difficile).

— *Proposition* : Hiérarchie

— *Définition* : Fonction constructible

— *Proposition* : Rupture dans la hiérarchie

— *Corollaire* : P$\neq$ EXPTIME et PSPACE$\neq$ EXPSPACE

2 Complexité en temps : P vs NP

Nous allons nous intéresser à la complexité en temps et notamment aux classes P et NP. Nous allons étudier leur robustesse en fonction du modèle. Puis nous allons étudier la complétude et son impacte sur la question ouverte $P = NP$ ou non. Enfin, nous allons regarder au-delà avec la classe EXPTIME.

2.1 Robustesse des classes temporelles

Les classes temporelles sont robustes à l'ajout ou à la suppression de ruban. Cependant, elles ne le sont pas au passage au non-déterministe. Cela implique que nous ne savons toujours pas si P égale ou non NP.

— *Proposition* : Toute machine à $k \geq 2$ bandes $\mathcal{M}$ est équivalente à une machine à une bande $\mathcal{M}'$ telle que $t_{\mathcal{M}'} = O\left((t_{\mathcal{M}})^2\right)$.

— *Interprétation* : L'utilisation d'une machine à plusieurs bandes n'influe pas sur la classe de complexité (on peut donc utiliser plusieurs bandes sans soucis).

— *Proposition* : Toute machine non-déterministe $\mathcal{M}$ est équivalente à une machine déterministe $\mathcal{M}'$ telle que $t_{\mathcal{M}'} = 2^{O(t_{\mathcal{M}})}$.

— *Interprétation* : Les classes P et NP ou EXPTIME et NEXPTIME ne sont à priori pas équivalent : on prend une exponentielle en passant de l'une à l'autre (on ne peut donc pas déterminiser facilement sans impacter la complexité une machine non-déterministe).

2.2 Notion de NP-complétude et conséquences théoriques

Afin d'étudier les spécificités de la classe NP, nous allons définir une famille de problème qui sont plus durs que tous les autres de cette classe. On définit alors la notion de complétude. Dans le cas de ces classes, nous devons définir la notion de réduction polynomiale.

— *Définition* : Réduction polynomiale

— *Définition* : Problème NP-dure et NP-complet

— *Théorème* : Cook

— *Application* : Montrer que d'autres problèmes sont NP-complet

— *Exemple* : Le problème PSA est NP-complet DEV

— *Exemples* : 3-coloration, set-cover, ...

— *Corollaire* : Si il existe $p \in$ NP-complet et $p \in$ P alors P = NP.

— *Corollaire* : Si P $\neq$ NP alors il existe $p \in$ NP tel que $p$ne soit pas complet.

2.3 Conséquence de la NP-complétude en pratique

Un problème NP-complet peut se traiter en pratique. On peut alors jouer : le temps, l'expressivité ou l'exactitude (si on a un problème d'optimisation).

— *Méthode* : Backtracking ou Branch and bound

— *Exemple* : DPLL

— *Méthode* : Approximation

— *Exemple* : Approximation du problème du voyageur de commerce

— *Méthode* : Restriction des entrées

— *Remarque* : Baisse de l'expressivité du problème

— *Exemple* : Logique de Horn

2.4 Au delà de la NP-complétude : la classe EXPTIME

Les classes P et NP ne sont pas les seules classes temporelles (même si elles sont très intéressantes). Les classes EXPTIME et NEXPTIME contiennent des problèmes dont la résolution n'est pas aisée car ce sont des tours d'exponentielle. Notons qu'il existe des problèmes encore plus difficile : les problèmes non-élémentaires : qui sont EXPTIME pour tout k de tour d'exponentielle. La généralisation de Presburer est un tel problème.

— *Exemple* (admis) : Presburger
— *Proposition* : Si EXPTIME $\neq$ NEXPTIME alors P $\neq$ NP (argument de padding)
— *Définition* : Problèmes non élémentaires
— *Exemple* : Presburger et généralité

3 Complexité en espace : *PSPACE*

Nous allons nous intéresser à la complexité en espace et notamment aux classes PSPACE et NPSPACE. Nous allons étudier leur robustesse en fonction du modèle. Puis nous allons étudier la complétude. Enfin, nous étudierons l'impacte de la complexité en espace dans l'étude de la classe P.

3.1 Robustesse des classes spatiales

Dans le cas de la complexité spatiale, les classes sont robustes si on déterminise la machine (théorème de Savitch). Cependant, elles ne le sont pas nécessairement face aux multibandes comme nous le verrons plus tard (même si ça ne change rien pour PSPACE).

— *Théorème* : Savitch DEV
— *Corollaire* : PSPACE = NPSPACE
— *Corollaire* : EXPSPACE = NEXPSPACE

3.2 Notion de PSPACE-complétude et conséquences pratiques

La complétude dans PSPACE se défini à l'aide d'une réduction polynomiale. On peut alors définir des solveurs qui viennent résoudre l'ensemble des problèmes PSPACE.

— *Définition* : Problème PSPACE-dur et PSPACE-complet.
— *Proposition* : Le problème QBF est PSPACE-complet.
— *Application* : Montrer que les problèmes sont PSPACE-complet.
— *Exemple* : Le problème d'universalité algébrique est PSPACE-complet.
— *Remarque* : De manière analogue aux SAT-solver, on produit des QBF-solver.

3.3 Exploration de la classe P

La complexité spatiale permet d'explorer la classe P est définissant une complexité log-space, la réduction qui va avec et les sous-classes qu'on en déduit. Nous définissons alors la classe NL, la NL-complétude et la P-complétude avec la réduction log-space.

— *Définition* : Classes NL et L
— *Exemple* : Accessibilité dans un graphe est dans NL
— *Définition* : Log-space réduction

— *Définition* : Problème NL-dur et NL-complet
— *Proposition* : Le problème d'accessibilité dans un graphe est NL -complet
— *Application* : Montrer que d'autre problème le sont.
— *Exemple* : Le problème 2-SAT est NL-complet
— *Remarque* : Théorème de Savitch
— *Définition* : Problème P-dur et P-complet
— *Proposition* : Le problème Horn est P-complet

4 Hiérarchie

Nous pouvons conclure avec le dessin de la hiérarchie telle que nous la connaissons aujourd'hui.

— Conclusion sur la hiérarchie des classes

Ouverture

On utilise les machines de Turing pour effectuer l'échelle de notre classification. Cependant, il existe d'autre modèle de calcul nous permettant d'étendre, de compléter, de raffiner ou de changer de points de vu sur cette classification.

Machines de Turing alternantes En fonction de leur degrés d'alternance, elles définissent la hiérarchie polynomiale qui étend les classes P, NP et $co - NP$.

Circuits Les classes AC_i (classe de problèmes résolus à l'aide d'un circuit de taille polynomiale (nombre de porte) et de profondeur $O\left(\log^i n\right)$) de la hiérarchie AC comme $AC_0 \subset NL$ dont on sait l'inclusion stricte dans P.

Machines de Turing à oracle Permet de classer les problèmes dans EXPTIME et même dans les problèmes indécidables.

Logique La logique permet de définir des classes de complexité : c'est ce que nous appelons la complexité descriptive. Par exemple la classe NP correspond à la classe des problèmes exprimables en la logique du second ordre existentielle, c'est-à-dire la logique du second ordre où on interdit de quantifier universellement sur les prédicats et fonctions.

Quelques notions importantes

Réductions

Pour montrer qu'un problème apparaît dans une certaine classe de complexité (et même sa dureté) ou qu'il est indécidable, nous utilisons une technique de preuve : la réduction. Pour appliquer le principe de réduction il nous faut connaître un premier problème possédant les propriétés que l'on souhaite montrer sur le deuxième.

Nous présentons ici le principe de la réduction dans sa généralité puis nous verrons comment le spécialiser pour en faire ce que nous souhaitons.

Définition. Une réduction d'un problème A à un problème B (Figure 1) est une fonction tr calculable telle que pour tout w instance de A, w est une instance positive de A si et seulement si $tr(w)$ est une instance positive de B. On note $A \leq B$.

On dit que A se réduit à B s'il existe une réduction de A à B (intuitivement, A est plus facile que B).

Remarque : En fonction des propriétés sur la fonction tr, on obtient différentes réductions qui vont nous permettre de spécialiser la réduction au résultat que nous souhaitons montrer.

Théorème (Principe de la réduction)**.** *Si A se réduit à B, alors si P est une propriété sur B alors P est une propriété sur A (dans notre cas, P peut être l'appartenance à une classe de complexité ou être le caractère indécidable d'un problème, ...).*

Démonstration. On raisonne par l'absurde et grâce à la fonction de traduction, on obtient une contradiction. □

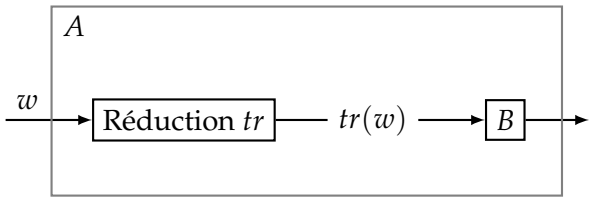

FIGURE 1 – Schéma du principe de réduction du problème A au problème B.

Nous allons donner quelques réductions de A à B et leurs propriétés. On note C une classe de complexité telle que $P \subseteq C$.

Réduction	Propriétés	Conséquences
Calculable	tr est calculable par une machine de Turing (variante de MT : équivalence)	A indécidable $\Rightarrow$ B indécidable B décidable $\Rightarrow$ A décidable
Temps polynomial	tr est calculable par une machine de Turing déterministe en temps polynomial	A est C-dur $\Rightarrow$ B est C-dur $(C \neq P)$ $B \in C \Rightarrow A \in C$
Espace logarithmique	tr est calculable par une machine de Turing déterministe en espace logarithmique (la MT a trois rubans)	A est D-dur $\Rightarrow$ B est D-dur $B \in D \Rightarrow A \in D$ $(D \neq P)$ avec $D \in \{L, NL, co-NL, P\}$

Remarque : La réduction en espace logarithmique est une réduction très spéciale car une machine qui travail en espace logarithmique a au moins deux rubans (il ne faut pas que l'entrée rentre dans la calcul). Mais pour la réduction, la sortie n'est pas non plus dans la borne de la mémoire utilisé (notons qu'elle est polynomiale (car une réduction en espace logarithmique s'exécute en temps polynomial)), il nous faut donc un troisième ruban. La machine de Turing effectuant la réduction a donc trois rubans : un ruban contenant l'entrée en lecture seule et en une seule passe ; un ruban de travail logarithmique et un ruban de sortie polynomial en écriture seule et en une seule passe.

Proposition. *Une réduction en espace logarithmique est une réduction en temps polynomial.*

Le théorème de Cook

Le théorème de Cook est un théorème historique car c'est le premier résultat de NP-complétude. Ce résultat n'applique donc pas une réduction polynomial à partir d'un problème NP-dur mais il explique quel est la réduction polynomiale à partir d'un problème NP quelconque.

Quelques rappels sur la NP-complétude [Papadimitriou ?]

Définition. La classe P est l'ensemble des problèmes de décision (vu comme un langage) décidé par une machine de Turing (ou un algorithme) déterministe en temps polynomial en la taille de l'entrée.

Remarque : Une définition alternative existe : elle fait apparaître la robustesse de cette classe. En effet, P est la réunion (sur $\mathbb{N}$) de tous les problèmes (langages) qui sont décidés par une machine de Turing déterministe en temps $O(n^k)$.

Définition. La classe NP est l'ensemble des problèmes de décision (vu comme un langage) décidé par une machine de Turing (ou un algorithme) non-déterministe en temps polynomial en la taille de l'entrée.

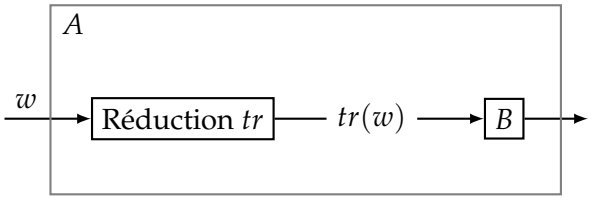

FIGURE 2 – Schéma du principe de réduction polynomiale du problème A au problème B.

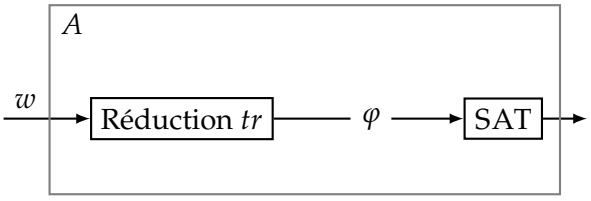

FIGURE 3 – Schéma du principe de réduction polynomiale dans le théorème de Cook d'un problème A dans NP au problème SAT.

Définition. Un vérifieur pour un problème A est une machine de Turing $\mathcal{M}$ déterministe telle que w est une instance positive de A si et seulement s'il existe c tel que $\mathcal{M}$ accepte (w, c). Un tel c est appelé certificat.

Proposition (Définition alternative de NP [6, p.294])**.** *$A \in NP$ si et seulement si il existe un vérifieur en temps polynomial de A.*

Idée de la démonstration. L'idée est de montrer comment convertir un vérifieur en une machine de Turing non-déterministe et réciproquement.

Algorithm 1 Algorithme convertissant un vérifieur en une machine de Turing non-déterministe.

```
1: procedure M(w)
2:     Choisir c de longueur f(|w|)
3:     if V(w, c) accepte then
4:         Accepter
5:     else
6:         Rejeter
7:     end if
8: end procedure
```

Algorithm 2 Algorithme convertissant un vérifieur en une machine de Turing non-déterministe.

```
1: procedure V(w, c)
2:     Simuler l'exécution M(w) en prenant les choix conseillés par c.
3:     if l'exécution accepte then
4:         Accepter
5:     else
6:         Rejeter
7:     end if
8: end procedure
```

□

Remarque : On a $P \subseteq NP$ et si $NP - C \cap P \neq \emptyset$ alors $P = NP$ où $NP - C$ est l'ensemble des problèmes NP-complets.

Définition. Une réduction polynomiale (Figure 2) de A vers B est une fonction tr telle que

— tr est calculable en temps polynomial ;

— w est une instance positive de A si et seulement si $tr(w)$ est une instance positive de B.

On note alors $A \preccurlyeq B$.

Théorème. *Si le problème A se réduit au problème B, alors :*

— *si $B \in P$ alors $A \in P$;*

— *si $A \in NP$ alors $B \in NP$.*

Définition. Un problème A est dit NP-dur si pour tout problème $B \in NP$, il existe une réduction polynomiale de B à A. Si, de plus, $A \in NP$, alors A est dit NP-complet

Le théorème en lui-même Nous allons maintenant énoncé et donner l'idée de la preuve du théorème de Cook (qui fut le premier à exhiber un problème NP-complet).

Définition. On définit le problème SAT pour les formules de la logique propositionnelle.

Problème : SAT
entrée une formule ϕ de la logique propositionnelle
sortie Oui si ϕ est satisfiable ; non sinon

Théorème. *Le problème* SAT *est NP-complet.*

Idée de la démonstration. Montrons que e problème SAT est NP-complet.

SAT $\in NP$ Choisir une valuation pour chacune des variables de la formule (choix non-déterministe) et vérifier si c'est une valuation. On accepte lorsqu'on a trouvé une valuation et sinon on rejette.

SAT est NP-dur Comme c'est la première démonstration qu'un problème est NP-complet, il nous faut utiliser la définition. On prend donc un problème NP B et on va le réduire au problème SAT (Figure 3). Par définition, il existe une machine de Turing non-déterministe $\mathcal{M}$ qui décide B en temps polynomial. On va alors coder l'exécution de cette machine dans des formules de la logique propositionnelle. Nous énonçons les propriétés que nous souhaitons coder sans en donner la traduction en formule de la logique propositionnelle.

1. La machine de Turing $\mathcal{M}$ et dans un état à tout instant t.
2. La machine de Turing $\mathcal{M}$ n'est jamais dans deux états à la fois.
3. Le curseur est positionné quelque part à tout instant t.
4. Le curseur n'est jamais à deux positions différentes.
5. À tout instant, toute case du ruban contient une lettre.
6. Une case du ruban ne contient pas plus d'une lettre.
7. À tout instant, on tire une transition pour aller vers l'instant $t + 1$.
8. On ne tire jamais plus d'une transition.
9. À l'instant 0, le ruban contient l'instance donnée à la machine.
10. À l'instant 0, la machine est dans l'état initial q_0 et son curseur est à la position 1.
11. La machine atteint son état d'acceptation.
12. On ne change pas le contenu du ruban, si le curseur n'y ait pas.
13. On ne tire qu'une transition de son état courant lisant la lettre sur le ruban.
14. On change s'état lorsqu'on applique une transition (le bon état).
15. On écrit la nouvelle valeur du ruban sur la case i.
16. Le curseur change de position lors d'une transition.

La formule que l'on cherche est la conjonction de toutes ses formules, ce qui prouve le théorème de Cook. □

Application La première application de ce théorème est une technique de preuve plus agréable pour montrer la NP-dureté. En effet, maintenant que nous avons un premier problème NP-complet, nous allons pourvoir réduire les autres à celui et par transitivité de la relation est réductible à, on prouvera la NP-dureté de ce problème. Depuis on en a trouvé des nombreux.

Références

[1] O. Carton. *Langages formels, calculabilité et complexité*. Vuibert, 2008.

[2] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Algorithmique, 3ème édition*. Dunod, 2010.

[3] R. Floyd and R. Biegel. *Le langage des machines*. International Thomson Publishing, 1994.

[4] J. Kleinberg and E. Tardos. *Algorithm Design*. Pearson international Edition, 2006.

[5] C.H. Papadimitriou. *Computational complexity*. Pearson, 1993.

[6] M. Sipser. *Introduction to the Theory of Computation*. Cengage learning, 1133187811.