Eléments de Complexité

Christian Attiogbé
Faculté des sciences – Université de Nantes
Christian.Attiogbe@irin.univ-nantes.fr

Complexité des problèmes

Motivations

☞ Sensibiliser et donner des rudiments de la *NP-complétude*, Rudiments indispensables pour l'appréhension de problèmes informatiques et la bonne conception des algorithmes.

☞ Il y a des problèmes pour lesquels il n'existe pas d'algorithmes efficaces.

- Identifier, face à un cahier de charges, qu'on se trouve confronté à de tels cas de problèmes,
- Ne pas chercher un algorithme efficace mais une d'approximation.

☞ Certains problèmes réels ne semblent pas plus difficiles que la recherche dans un graphe ou le tri,
- en réalité, ils sont *NP-complets* (ou de la classe des *NP-complets*),

- le **concepteur qui n'est pas familier à cette classe de problèmes court alors le risque de perte de temps considérable**.

Algorithmes efficaces

☞ **Hiérarchisation des fonctions, liée à leur rapidité de croissance.**

Une fonction exponentielle croît beaucoup plus vite que n'importe quelle fonction polynomiale.

Un algorithme dont la complexité croît exponentiellement avec la taille des données d'entrée ?

$\rightarrow$ **n'est pas utilisable dans la pratique.**

Efficacité

Définitions

Un algorithme est dit polynomial si sa complexité est $O(n^p)$ pour un certain entier p.

Les algorithmes polynomiaux sont dits **efficaces** ou **rapides**.

L'*efficacité* dans ce cadre est un concept purement mathématique.

Exemples

L'algorithme de multiplication de matrices (nxn) a un coût de n^3 pour des données de taille n.

C'est une complexité polynomiale. L'algorithme est efficace.

L'algorithme d'énumération des parties d'un ensemble de n éléments a un coût de (2^n) pour des données de taille n.

C'est d'une complexité exponentielle. L'algorithme n'est pas efficace.

Traitables - Intraitables

n \ f(n)	n	n^2	n^3	2^n	3^n
10	0,00001s	0,0001s	0,001s	0,001s	0,59s
20	0,00002s	0,0004s	0,008s	1s	58mn
40	0,00004s	0,0016s	0,064s	1,7mn	3855**siècles**
60	0,00006s	0,0036s	0,216s	366 **siècles**	$1,3.10^{13}$**siècles**

☞ **Traitables** : Temps polynomial

☞ **Intraitables** : Temps (super)exponentiel

Décidables et indécidable

☞ L'ensemble des problèmes est partitionné en deux classes.
– Les **indécidables** : classe des problèmes pour lesquels on ne trouvera jamais d'algorithmes de résolution,

– Les **décidables** : classe des problèmes pour lesquels il existe une solution algorithmique.

☞ Résoudre un problème par l'algorithmique ⇒ risque :
– de ne **trouver qu'une solution algorithmique inefficace**,
– de **ne pas trouver de solution algorithmique**.

Décidables et indécidable : conséquences

Il est alors important de vouloir répondre à la question :
"Existe-t-il des problèmes qui n'ont pas de solution algorithmique ?"

Pour répondre à une telle question,
une **formalisation** ou à défaut
une **formulation précise de la notion de problème est nécessaire**.

Formalisation : Problèmes et Langages

Problème : Définition

Un problème (abstrait) Q est une relation binaire sur un ensemble $\mathcal{I}$ d'instances d'un problème et un ensemble $\mathcal{S}$ de solutions.

$$Q \ : \mathcal{I} \ \leftrightarrow \mathcal{S}$$

Certains aspects de la théorie des langages permettent d'illustrer simplement des propriétés des problèmes.

Formalisation des problèmes

Les instances d'un problème sont représentées par une fonction accessible au programme qui résout le problème.

Soit par exemple un programme PASCAL qui resout un problème,

Les instances du problème résolu sont représentées les données en entrée du programme $\Rightarrow$ une collection d'entiers, de chaînes de caractères, ...

☞ il faut un alphabet qui permette de construire les mots qui représentent l'instance du problème.

L'ensemble des mots ainsi définis forme un langage.

On dit qu'un problème est caractérisé par le langage des encodages de ses *instances positives* (ou instances oui).

→ La résolution d'un problème est comparable à la *reconnaissance du langage* des instances positives du problème.

Les problèmes pour lesquels il existe une solution algorithmique, donc un traitement par ordinateur, se répartissent en deux classes :

- Les **problèmes de décision**. Ils sont modélisés par la notion de langage : déterminer si un mot appartient ou non à un langage.
- Les **problèmes de calcul** conduisant à un résultat qui est une valeur (numérique ou non).
  Ils sont modélisés par une fonction de transformation des mots d'un alphabet d'entrée en des mots d'un autre alphabet de sortie.

Calcul de coût

Problèmes de décision

Définitions
Un problème de décision est une classe d'énoncés auxquels on doit répondre par oui ou par non.

Chaque énoncé est appelé une **instance** du problème dont la réponse appartient à l'ensemble $\{oui,\ non\}$.

Donner une **réponse positive** à un problème de décision $\rightarrow$ **exhiber un algorithme pour le résoudre**.

Donner une **réponse négative** à un problème de décision $\rightarrow$ **prouver qu'il n'existe pas d'algorithme** pour le résoudre. (tâche difficile).

Calcul de coût

Problèmes de décision

Un énoncé d'un problème de décision se formule souvent sous la forme d'une question.

Exemples
- Etant donné un entier n, existe-t-il deux entiers p et q inférieurs à n tels que $n = p * q$?
  (si oui n est dit composé).
- Soient un graphe G et un entier k, G admet-il un cycle de longueur $\geq k$?
- Soient P et P' deux programmes, P et P' sont-ils équivalents ?
- Soit un graphe G, G peut-il être colorié avec 5 couleurs (sans nœud voisins de même couleur) ?

Généralement on décrit un problème de décision sous la forme d'une description générique d'objets (de valeurs fixées ou non) et d'une question.

Exemple

COMP	Instance : un entier n
	Question : n est-il composé ?

$COMP$ est le nom (généralement connu) du problème.

☞ tout problème de décision peut avoir une réponse *oui* ou *non* $\rightarrow$ on comprend aisément la modélisation qui consiste à penser un problème comme la question :

savoir si un mot donnée (codage en entrée) appartient ou non à un certain langage, celui représentant la totalité des mots pour lesquels la réponse est oui.

Problèmes : décision, optimisation

Tous les problèmes ne sont pas des problèmes de décision, mais beaucoup de problèmes peuvent se ramener à un problème de décision ou à un problème d'optimisation.

Quel est le nombre minimum de couleurs avec lesquelles on peut colorier un graphe G ?
(appelé nombre chromatique)

En général, lorsqu'on trouve une solution **rapide** pour un problème de décision, on arrive avec un peu d'effort à trouver une solution au problème d'optimisation correspondant.

Exemple
Soit G un graphe de 100 sommets.
Peut-on colorier G avec 10 couleurs

Problèmes : Décision, Vérification

Dans les problèmes de décision, on s'intéresse surtout à la **vérification**.

- Dire qu'une instance I d'un problème admet la réponse *oui* équivaut à affirmer que la proposition
  "I possède une certaine propriété L" est vraie.
  $\rightarrow$ il en existe une preuve convaincante.

- L'existence et la nature de la preuve ne relèvent pas forcément d'une procédure efficace.

– L'important c'est que la **vérification de la correction de cette preuve** vis à vis de la réponse *oui* puisse être entreprise **efficacement** sur le plan algorithmique.

Mathématiquement, cette perception s'apparente à la différenciation des deux activités suivantes :

– Prouver un théorème.

– Vérifier une preuve d'un théorème.

Problèmes de calcul

Nous ne développons pas cette partie dans ces notes de cours.

Le lecteur intéressé peut consulter les ouvrages traitant des fonctions calculables. Par exemple

Wolper, *Introduction à la calculabilité*, InterEditions 1993,

Cormen, Leiserson, Rivest, *Introduction à l'algorithmique*, Dunod 1994

Réductibilité

Face à un problème ou à un cahier de charges, pouvoir dire *problème connu* (donc traité !)

Soient P et P' deux problèmes de décision.

On dit que P' est **rapidement réductible** en P si l'on peut convertir, en un **temps polynomial**, toute instance I' de P' en une instance I de P, de telle sorte que I **et** I' **aient la même réponse** *oui* ou *non*.

Exemple :
On veut résoudre un système linéaire de 100 équations à 100 inconnues de la forme $Ax = b$ (où A est une matrice).
On se procure d'un logiciel et on s'aperçoit désespérement que le logiciel ne fonctionne que sur des systèmes pour lesquels la matrice A est symétrique alors que notre matrice ne l'est pas.

Issue possible : se ramener à une matrice symétrique A^TA en cherchant la solution du système $A^TAx = A^Tb$.

Réductibilité

☞ **La démarche de réduction ne change pas la nature du problème.** Les problèmes indécidables ne peuvent pas se ramener à des problèmes décidables.

Théorème
Soient deux problèmes P et P' tels que P soit réductible en P' :
– si P est indécidable alors P' l'est aussi,
– si P' est décidable alors P l'est aussi.

Les classes P et NP

La notion d'efficacité des algorithmes permet de diviser les problèmes décidables en deux classes de complexité.

– La **classe** P des problèmes pour lesquels il existe une solution algorithmique de **coût polynomial** et

– la classe des autres.

Dans la pratique, lorsqu'on recherche pour un problème un algorithme effectivement utilisable, il faut **d'abord identifier la classe de complexité du problème**.

Les classes P et NP

Les problèmes décidables qui appartiennent à la classe P sont dits **faciles**.

Les problèmes **décidables qui n'appartiennent pas à la classe** P **sont dits** *difficiles* **ou** *intraitables*.

☞ Il n'est pas toujours simple de rattacher un problème donné à l'un des deux groupes, bien qu'ils soient disctincts.

Il faut
– exhiber un algorithme de coût polynomial ou
– démontrer l'inexistence d'un tel algorithme ;

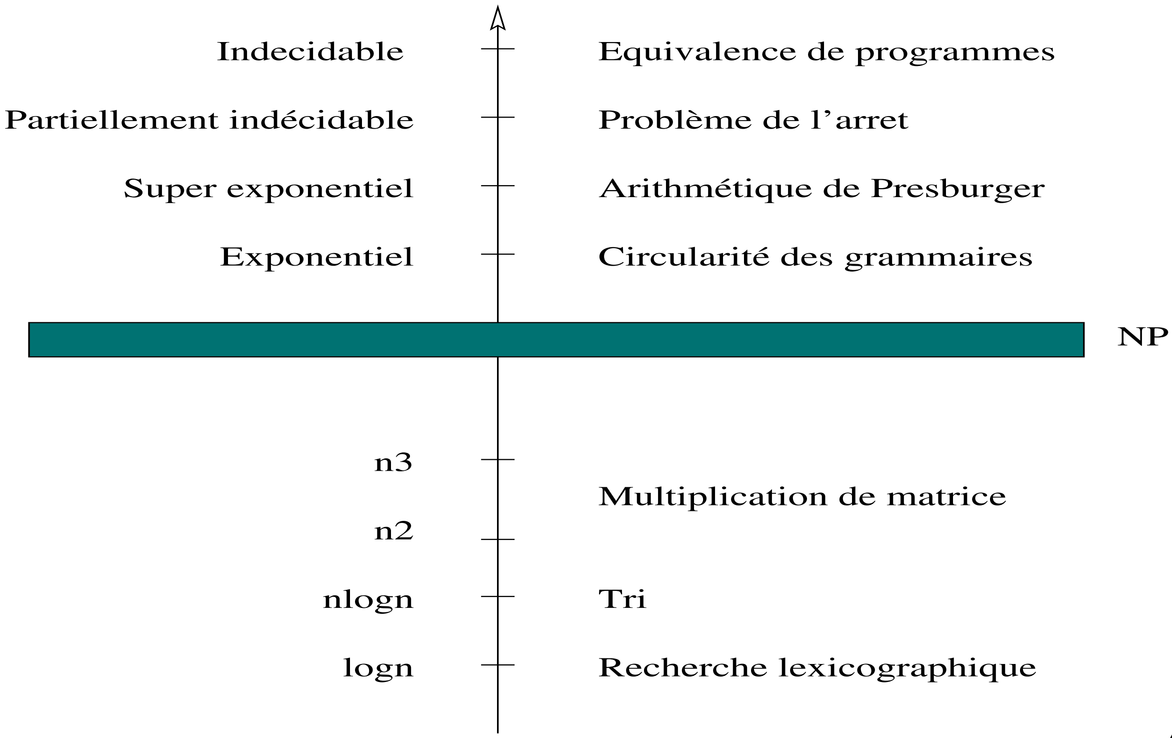
Les classes P et NP

Entre les problèmes faciles et les problèmes difficiles,

il y a une série de problèmes intéressants de par leurs propriétés et pour lesquels on ne connaît pas d'algorithmes efficaces.

– Les meilleurs algorithmes connus actuellement pour ces problèmes ont un coût (en temps d'exécution) exponentiel et
– personne n'a encore réussi à démontrer que ces problèmes n'appartiennent pas à la classe P.

En attendant, on les met dans une classe dénommée NP (*Nondeterministic Polynomial*).



La classe P

Un problème de décision est dit de la classe P
s'il existe un algorithme polynomial qui reconnait le langage des *instances oui*. Un tel problème est dit **facile**.

En d'autres termes, un problème de décision appartient à la classe P

- s'il existe un algorithme A et un nombre c tel que
- **pour toute instance I** du problème,
  - A donne une solution en un temps polynomial $O(n^c)$
  - n est la taille de l'instance (par exemple le nombre de bits de la chaîne d'entrée qui représente l'instance I).

Problème P : Exemples

- **Recherche d'une informatin parmi N**
  Problème de décision soluble en $O(n)$ par recherche séquentielle
  On peut atteindre $O(log\ n)$ avec une recherche dichotomique sur tableau trié ou un arbre binaire.

- **Tri de nombres**
  Problème de décision (d'optimisation),
  soluble en $O(nlogn)$ par de bons algorithmes
  (par exemple tri par tas).

- **Exploration de graphes**
  Soit $G = (X, U)$ un graphe orienté. Soient $s, t \in X$ deux sommets.
  Existe-til un chemin de s à t ?
  problème soluble en $O(M)$, M est le nombre d'arcs, par un algorithme appelé procédure de marquage.

- **Chemin de coût minimal**
  Soit $G = (X, U, C)$ un graphe orienté valué, soient $s, t \in X$ deux sommets.
  Trouver un chemin de coût minimal
  Ce problème apparait comme sous-problème dans de nombreux problèmes de graphes :
  BELLMAN en $O(N.M)$ pour G, C quelconques
  DIJKSTRA en $O(N^2)$ pour les coûts > 0

La classe NP

Un problème de décision Q appartient à la classe NP s'il existe un algorithme A tel que :

1. à chaque instance I (i.e. mot $\in Q$) pour laquelle la réponse est *oui*, un certificat $C(I)$ est associé, tel que
   - lorsque la paire $(I, C(I))$ est une entrée de l'algorithme A,
   - celui ci reconnaît que I appartient au langage Q,

2. si I est un mot $\notin Q$,
   il n'existe aucun certificat tel que A reconnaisse I,

3. l'algorithme A s'arrête en un temps polynomial.

La classe NP . . .

NP est la classe des problèmes de décision pour lesquels *il est aisé de* **vérifier** que les réponses sont correctes.

On ne cherche pas un moyen de trouver une solution mais de vérifier qu'une solution donnée est correcte.

☞ Les problèmes de la classe P ont des solutions **faciles à trouver** alors que
ceux de la classe NP ont des solutions **faciles à vérifier** même si elles sont difficiles à trouver.

La classe NP : exemple

Le problème du coloriage de graphes n'est pas dans P mais dans NP.

Soit un graphe G coloriable par k couleurs,
le certificat de G peut être une liste de couleurs affectées à chaque sommet de G.

Comment a-t-on obtenu cette liste de couleurs ? (problème ardu !)
En fait, nous ne nous intéressons qu'à la vérification.

Pour vérifier qu'un certain graphe G est k coloriable, il suffit d'avoir la couleur de chaque sommet telle que le graphe soit correct.

La classe NP . . .

Si on a un certificat, la vérification de l'algorithme est simple.

– vérifier d'abord que chaque sommet a bien une et une seule couleur,

– puis qu'on n'a pas utilisé plus de k couleurs et

– enfin que pour toute arête e de G les deux extrémités sont bien de deux couleurs différentes.

Le problème du coloriage appartient donc bien à la clase NP.

Quelques problèmes classiques

– **Le voyageur de commerce**.
Soient n points du plan d'une ville et une distance D.
Existe-t-il un chemin qui passe par toutes les villes, retourne au point de départ et dont la longueur totale n'excède pas D ?

– **Le coloriage de graphe**.
Soit un graphe G et un entier k.
Peut-on colorier correctement les sommets de G avec k couleurs ?

Quelques problèmes classiques . . .

– **Le bin-packing**.
Soient un ensemble fini S d'entiers positifs, un entier k (la "bin-capacity") et un entier N (le nombre de "bin").
Existe-t-il une partition S en N sous-ensembles au plus, telle que la somme des entiers de chaque sous-ensemble soit $\leq k$?
En d'autres termes, *peut-on "empaqueter" les entiers de S dans au plus N cases sachant que la capacité de chaque case est k ?*

– etc

Exemples NP

Ces problèmes sont difficiles à résoudre.
Considérons celui du coloriage de graphe.

On peut essayer tous les moyens possibles de colorier les sommets de G en k couleurs pour voir si ça marche.

Si G a n sommets, il y a k^n possibiltés.

Si $n = 50$ et $k = 10$, ... les capacités des machines d'aujourdhui restent encore limitées.

Les problèmes NP-complets

Les problèmes NP-complets

Un problème de décision est *NP-complet* si :
– il est *NP* et
– si chaque problème *NP* est rapidement réductible en lui.

Ce sont les problèmes **les plus difficiles de** NP.
Les problèmes *NP-complets* sont **nombreux**.

Si l'on devrait trouver un algorithme de temps polynomial pour un certain problème *NP-complet* Q,

on aurait trouvé par la même occasion un algorithme rapide pour tous les problèmes NP.

NP-complets . . .

Soit une certaine instance I' d'un problème NP Q'.

Puisque Q' est rapidement réductible en Q, on peut transformer I' en une instance I de Q.

On applique alors l'algorithme trouvé à I.

On aura ainsi utilisé du début à la fin un temps polynomial.

Supposons que l'on arrive un jour à démontrer que le problème de coloriage de graphes est *NP-complet* et que l'on trouve le jour suivant un algorithme rapide pour le résoudre.

NP-complet . . .

Considérons alors une instance du "bin-packing".

Elle peut être convertie en une instance de coloriage ayant la même réponse *oui* ou *non*.

Utilisons alors l'algorithme rapide trouvé pour le coloriage, la réponse obtenue est aussi correcte pour le problème initial.

Réciproquement, supposons que l'on montre qu'il est impossible de trouver un algorithme rapide pour un certain problème Q *NP-complet*.

On ne peut alors trouver d'algorithme rapide pour aucun des problèmes *NP-complets* Q',

car sinon, on pourrait résoudre les instances de Q en les réduisant en instances de Q'.

NP-complet . . .

Si on peut démontrer qu'il n'existe pas de méthode rapide pour tester si un entier est premier,

on aurait alors montré qu'il n'existe pas de méthode rapide pour décider si des graphes sont $k-$coloriables,

car le premier problème est NP est le deuxième est *NP-complet*.

En résumé,

la rapidité pour un problème *NP-complet* entraîne la rapidité pour tous les NP ;

la lenteur prouvée pour un problème NP entraîne la lenteur de tous les problèmes *NP-complets*.

Quelques propriétés des *NP-complets*

– Les problèmes sont tous très difficiles à résoudre et aucun algorithme se terminant en un temps polynomial n'a été trouvé pour aucun d'entre eux.

– **L'impossibilité de trouver des algorithmes polynomiaux n'a pas été prouvée.**

– Ce n'est pas simplement une liste de problèmes difficiles. Si un algorithme rapide pouvait être trouvé pour un d'entre eux, il y aurait des algorithmes rapide pour chacun d'entre eux.

Propriétés des *NP-complets* . . .

– Réciproquement, s'il pouvait être prouvé qu'aucun algorithme n'existe pour un des problèmes *NP-complets*, il ne pourrait y en avoir pour aucun autre.

– **L'existence ou non d'algorithmes polynomiaux pour les problèmes** *NP-complets* **est probablement considéré comme un des principaux problèmes encore irésolus en Informatique**.