



Les problèmes «problématiques» (1)

Rappel:

un problème (une tâche) sera dit *formalisable* s'il peut être exprimé sous la forme d'une association entre les entrées et les sorties d'une machine de Turing.

Exemple:

Calcul de la parité d'un nombre:

entrée: le nombre à traiter (sous la forme d'une séquence binaire).

sortie: 1 ou 0 selon que le nombre est ou n'est pas pair.



Les problèmes «problématiques» (2)

On va distinguer trois grandes familles de problèmes problématiques:

- Les problèmes *non formalisables*
- Les problèmes *non décidables*
- Les problèmes *NP-complets*



Les problèmes non formalisables

Ce type de problème est plus fréquent
qu'on pourrait le penser *a priori*.

Un problème peut être non formalisable:

☞ parce qu'il est intrinsèquement vague

exemples: *la vie a-t-elle un sens ?*
Dieu existe-il ?

☞ parce qu'il est difficile d'obtenir un consensus sur sa
spécification précise (c'est le cas le plus fréquent).

exemple: *déterminer la santé financière d'une entreprise*

Cette tâche ne paraît pas problématique *a priori*...
mais elle est en fait difficile à formaliser (spécifier) précisément,
car plusieurs approches sont possibles, entre lesquelles les experts sont partagés.



Les problèmes non décidables

Un des intérêts de l'introduction de la notion de machine de Turing est qu'elle permet de **démontrer** que **l'on ne peut pas résoudre tous les problèmes formalisables**.

Ceci est important, car cela montre les limites intrinsèques de l'informatique.

Les problèmes qui ne peuvent être résolus à l'aide d'une machine de Turing (et donc, si l'on admet l'hypothèse de Church, qui ne peuvent être résolus par aucune approche algorithmique) sont appelés des problèmes **non décidables**.

Exemple:

Ecrire un programme qui, pour tout programme P , accompagné de ses données d'entrée D , permet de déterminer si l'exécution de P sur D termine (après un nombre fini d'opérations).

Ce problème est appelé
le problème de la terminaison des programmes.

Problème non décidable: la terminaison (1)



Démontrons que le problème de la terminaison
d'une machine de Turing T sur des données d'entrée i
est non décidable;

en d'autres termes, qu'on ne peut construire une machine
de Turing permettant de déterminer, pour toute machine de Turing t
et toutes données d'entrée i , si $t(i)$ termine après un nombre fini d'opérations.

(*par l'absurde*) Supposons qu'une telle machine T existe:

T est donc telle que:

$$T(i, j) = \begin{cases} 1 & \text{si } T_i(j) \text{ termine} \\ 0 & \text{sinon} \end{cases}$$

A partir de T , construisons alors la machine T' telle que:

$$T'(i) \text{ termine } \underline{\text{ssi}} T(i, i) = 0$$

Problème non décidable: la terminaison (2)



Soit $i' = i(T')$ l'indice de T'

La question est maintenant de savoir si $T'(i')$ termine ?

Supposons tout d'abord que $T'(i')$ **termine**:

- ⇒ alors, par définition de T' , $T(i', i') = 0$,
- ⇒ et par définition de T , $T_{i'}(i')$ ne termine pas ...
mais $T_{i'}(i') = T'(i')$, et donc $T'(i')$ **ne termine pas**

Supposons maintenant que $T'(i')$ **ne termine pas**:

- ⇒ alors, par définition de T' , $T(i', i') = 1$,
- ⇒ et par définition de T , $T_{i'}(i')$ termine...
mais $T_{i'}(i') = T'(i')$, et donc $T'(i')$ **termine**.

Par conséquent, T' ne peut exister,
et il en va de même pour T

CQFD



Notion de complexité: introduction

On se rend intuitivement compte que tous les problèmes pouvant être résolus par une machine de Turing ne sont pas de même difficulté... mais comment mesurer cette difficulté ?

La difficulté d'un problème peut être mesurée par **le temps d'exécution** d'un algorithme qui le résout, ou encore la **quantité de mémoire** requise lors de la mise en œuvre de cet algorithme.

Cependant, les notions de temps de traitement et d'espace mémoire sont dépendantes de la machine physique utilisée pour implémenter l'algorithme; on a donc besoin d'une mesure absolue, i.e. indépendante de toute implémentation.

La notion de machine de Turing va nous permettre de définir une telle mesure absolue de la difficulté d'un problème; elle sera appelée la **complexité (algorithmique) du problème**



Notion de complexité: définition

Considérons un problème P formalisable et décidable,
et une machine de Turing T résolvant P

- ➡ Nous appellerons **complexité (temporelle) pire cas** (i.e. dans le cas le plus défavorable) de T pour P , le nombre maximal de déplacements de la tête de lecture/écriture que devra effectuer la machine T pour résoudre une instance de P de taille maximale n (où la taille n de l'instance est la longueur de la séquence binaire codant cette instance comme entrée de la machine de Turing)
- ➡ Nous appellerons **complexité (spatiale) pire cas** de T pour P , la longueur de bande nécessaire à la machine T pour résoudre une instance de P de taille maximale n .



Complexité: efficacité des algorithmes

La notion de complexité permet de définir deux grandes classes de problèmes:

➡ les **problèmes faciles**,
pouvant être résolus par une machine de Turing de complexité pire cas bornée par un polynôme en la taille des entrées.

Ces problèmes seront aussi appelés **problèmes polynomiaux**, et les algorithmes permettant de les résoudre avec une complexité polynomiale seront appelés des **algorithmes efficaces**.

➡ les **problèmes difficiles**,
(i.e. les problèmes formalisables, décidables, qui ne sont pas *faciles*) pour lesquels la complexité pire cas n'est pas bornée par un polynôme...
Ces problèmes seront aussi appelés **problèmes non polynomiaux**, et les algorithmes permettant de les résoudre seront appelés des **algorithmes inefficaces**.

Complexité: influence du codage



Suivant la définition qui à été donnée de la complexité, cette dernière dépend de la manière dont les entrées sont codées...

... Or, il peut y avoir des codages plus ou moins efficaces !

Exemple: codage des entiers positifs

- Codage binaire efficace:
 i est codé par la séquence i_0, i_1, \dots, i_n t.q. $i = \sum_{k=0}^n i_k \cdot 2^k$ avec $i_k \in \{0,1\}$. La longueur $l_1(i)$ des codes de i est de l'ordre de $\log_2(i)$
- Codage binaire inefficace:
 i est codé par une séquence de 1 de longueur i .
 La longueur $l_2(i)$ des codes de i est $\approx 2^{l_1(i)}$

En fait, on dira qu'un codage est sympathique si, pour toute entrée, la longueur de la séquence obtenue est bornée par un polynôme en la longueur de la séquence obtenue par un codage minimal.



Complexité: notation $O(\dots)$

Tant qu'il s'agit de déterminer la classe de difficulté d'un problème (polynomial ou non), ce qui importe est de savoir si cette complexité est ou non majorée par un polynôme.

Pour cette raison, on introduit généralement la notation de Landau $O(\dots)$

Définition:

Pour deux fonctions f et g de \mathbb{N} dans \mathbb{R} , on écrit $f = O(g)$ **ssi** il existe deux entiers n_0 et k t.q. $\forall n \geq n_0, |f(n)| \leq k |g(n)|$

Dans ce cas, on dit que f est en $O(g)$.

L'intérêt de la notation de Landau est qu'on a le résultat suivant:

Un problème P est polynomial **ssi** il existe un entier k et une machine de Turing T résolvant P tq la complexité pire cas de T pour P , pour une entrée de taille n au sens d'un codage sympatique quelconque est en $O(n^k)$

Exemple de problème polynomial:

la résolution par machine de Turing du calcul de parité donné en cours est en $O(n)$ [dans ce cas, on parle d'algorithme *linéaire*]. C'est donc un problème polynomial.



Prb polynomiaux et non polynomiaux (1)

La distinction entre problèmes faciles et difficiles est en fait d'une grande importance pratique !

- ➡ Les problèmes **polynomiaux** correspondent à des problèmes pour lesquels on a de bonnes chances de pouvoir les résoudre (soit dès aujourd'hui, soit dans un proche avenir) avec une machine, pour toutes les tailles raisonnables des entrées.
- ➡ Les problèmes **non polynomiaux** correspondent à des problèmes que l'on a de bonnes chances de ne jamais pouvoir résoudre avec une machine, pour des tailles raisonnables des entrées.

En conséquence, si l'on sait qu'un problème est non polynomial, il ne sert à rien (dans la plupart des cas) d'essayer de construire un algorithme permettant de le résoudre de façon exacte... les contraintes de temps ou d'espace mémoire que va imposer cet algorithme seront toujours prohibitives, et rendront la résolution irréaliste (des siècles de traitements, ou des téra d'espace mémoire), pour des tailles d'entrées suffisamment importantes.

Prb polynomiaux et non polynomiaux (2)



Pour les problèmes non polynomiaux,
il faudra donc envisager des solutions approchées ou heuristiques...

Exemple numérique:

Considérons un problème de complexité en $O(n!)$,
comme par exemple un problème nécessitant de parcourir toutes
les permutations de n objets), alors pour résoudre une instance de taille
25 de ce problème, il faudrait près de 5 millions de siècles à une
machine effectuant 1 milliard d'instructions par seconde !!



Problème polynomial: 2-SAT (1)

Le problème k-SAT peut être défini de la façon suivante:

Soit une proposition logique E sous forme normale conjonctive (i.e. une conjonction de disjonctions) de degré k (i.e. chaque disjonction contient exactement k littéraux).

Question: E est-elle satisfiable, i.e. existe-t-il une affectation de valeurs de vérité aux littéraux qui rende E vraie ?

Exemple: une instance de 3-SAT

$E = (A \text{ ou } B \text{ ou } !C) \text{ et } (!A \text{ ou } B \text{ ou } !C)$ où !X représente la négation de X

A	B	C	A ou B ou !C	!A ou B ou !C	E
0	0	0	1	1	1
0	0	1	0	1	0
0	1	0	1	1	1
0	1	1	1	1	1
1	0	0	1	1	1
1	0	1	1	0	0
1	1	0	1	1	1
1	1	1	1	1	1

E est donc **satisfiable**.

Par exemple pour:

- A vrai
- B faux
- C vrai



Problème polynomial: 2-SAT (2)

Montrons que 2-SAT est un problème polynomial:

Une instance de 2-SAT est une proposition logique de la forme:

$$E = (x_1 \text{ ou } y_1) \text{ et } (x_2 \text{ ou } y_2) \text{ et } \dots \text{ et } (x_k \text{ ou } y_k)$$

où les x_i, y_i appartiennent à un ensemble de littéraux, $L(E) = \{z_1, \dots, z_e, !z_1, \dots, !z_e\}$

Mais comme $(x_i \text{ ou } y_i) \equiv (!!x_i \text{ ou } y_i) \equiv (!x_i \Rightarrow y_i) \equiv (!y_i \Rightarrow x_i)$

E peut être représenté sous la forme d'un graphe $G(E)$, appelé *graphe des implications*, de la façon suivante:

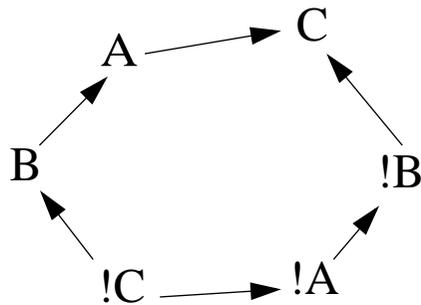
- les noeuds de $G(E)$ sont les éléments de $L(E)$;
- si E contient $(x_i \text{ ou } y_i)$ alors $G(E)$ contient les deux arcs $(!x_i, y_i)$ et $(!y_i, x_i)$



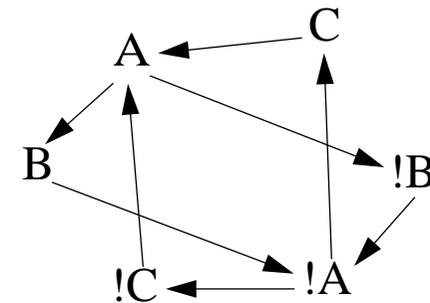
Exemple:

Si $E_1 = (A \text{ ou } !B) \text{ et } (B \text{ ou } C) \text{ et } (!A \text{ ou } C)$,
 $E_2 = (!A \text{ ou } B) \text{ et } (!A \text{ ou } B) \text{ et } (A \text{ ou } !C) \text{ et } (A \text{ ou } C)$

alors $G(E_1)$:



et $G(E_2)$:



On peut montrer que:

(1) E est satisfiable ssi (2) $\forall i$, $G(E)$ ne contient pas de circuit passant par z_i et $!z_i$

Si pour tout graphe orienté (2) peut être vérifié en un temps polynomial en le nombre de sommets, cela implique que (1) est bien polynomial.



Les problèmes de décision

Nous allons désormais nous restreindre à un type particulier de problèmes, les *problèmes de décision*.

Définition:

Un problème de décision est un problème qui peut s'exprimer sous la forme d'une question attendant une réponse binaire (*oui* ou *non*).

La forme générale d'un problème de décision sera donc:

- 1) une description de l'instance du problème;
- 2) l'expression d'une question oui/non portant sur cette instance.

Exemples:

3-SAT: déjà vu

clique:

entrée = un graphe G et un entier positif k

question = « G contient-il une clique d'ordre k ?» (i.e. un sous-graphe complet à k sommets)

recouvrement:

entrée = un ensemble X et une famille f de sous-ensembles de X

question = «existe-il une sous-famille f' de f tq tout élément de X appartient exactement à un élément de f' ?»



Les problèmes de décision NP (1)

Les problèmes de décision sont très fréquents dans beaucoup de situations pratiques où l'on peut être amené à chercher à écrire des résolutions algorithmiques.

Il est donc important de pouvoir déterminer, pour un problème de décision donné, s'il est ou non polynomial. Ceci est souvent difficile, en particulier pour une classe spécifique de problèmes de décision, la classe des *problèmes NP*.

Définition:

Un **problème de décision** sera dit **NP** si chacune de ses instances positives peut être associée à une preuve (de sa positivité) de taille polynomiale en la taille de l'instance et vérifiable en un temps polynomial [en la taille de l'instance].

Une preuve vérifiant les conditions ci-dessus sera également appelée un *certificat succinct* de l'instance positive.



Les problèmes de décision NP (2)

En plus des exemples de problèmes de décision précédemment mentionnés (3-SAT, clique, recouvrement), un problème typique de la classe NP est celui du voyageur de commerce:

voyageur de commerce:

entrée = un ensemble fini de villes, de distances 2 à 2 connues,
un entier positif k

question = «existe-il un circuit, passant exactement une fois par chacune des villes, de longueur inférieur à k ?»

<i>problèmes</i>	<i>certificats succints</i>
3-SAT	indication d'une affectation de valeurs de vérité aux littéraux qui rende la proposition d'entrée vraie.
clique	indication de k noeuds du graphe d'entrée constituant une clique
recouvrement	indication d'une sous-famille f' vérifiant la propriété demandée
voyageur de commerce	indication d'un circuit de longueur inférieure à k



Les problèmes de décision NP (3)

Une caractéristique importante de la classe des problèmes (de décision) NP est que, pour un grand nombre d'entre eux, **on ne sait pas dire s'il sont ou non polynomiaux.**

On est en effet souvent dans la situation où, pour le problème NP considéré, on ne sait pas produire une solution polynomiale...

... mais on ne sait pas non plus démontrer que le problème n'en admet pas (c-à-d que le problème est non polynomial) !

Que faire dans ce cas ?

On cherche alors à comparer la difficulté du problème NP considéré à celle d'autres problèmes NP, grâce à la notion de **réduction d'un problème à un autre**



Réduction d'un problème à un autre

Considérons deux problèmes de décision P_1 et P_2 .

S'il existe une transformation permettant de convertir toute instance positive (respect. négative) de P_1 en une instance positive (respect. négative) de P_2 , de complexité polynomiale (en la taille des instances de P_1), alors P_1 est dit

[polynomialement] réductible à P_2 ,

et l'on écrira: $P_1 \leq P_2$

La conséquence importante de la réductibilité de P_1 à P_2 est que si on peut montrer que P_2 est polynomial, alors P_1 l'est aussi.

En d'autres termes, **P_1 est au plus aussi difficile que P_2**



Les problèmes NP-complets

L'intérêt de la notion de réduction de problème pour la classe NP est qu'elle permet de démontrer qu'il existe, au sein de cette classe, des problèmes particulièrement difficiles, appelés *problèmes NP-complets*.

Plus précisément, on dira qu'un problème de décision P est NP-complet si, pour tout problème de décision NP P', on a: $P' \leq P$

En d'autres termes, un problème NP-complet est un problème NP au moins aussi difficile que tout autre problème NP.

L'intérêt de la notion de NP-complétude est que l'on peut démontrer qu'il existe effectivement des problèmes NP-complets...

3-SAT, clique, recouvrement, voyageur de commerce en sont des exemples.



P = NP ?

Malgré des recherches intensives,
aucune solution polynomiale n'a jusqu'à présent
pu être trouvée pour un problème NP-complet...

Notons qu'il suffit de trouver une telle solution pour n'importe lequel
des problèmes NP-complets connus pour pouvoir dériver des solutions
polynomiales **pour tous** ces problèmes.

Dans l'état actuel des connaissances, on ne sait donc pas si
les problèmes de la classe NP sont polynomiaux dans leur ensemble,
ou s'il en est certains qui sont intrinsèquement difficiles.

Cette question se résume par la formulation lapidaire:

$$P = NP ?$$

L'opinion généralement partagée est que:

- soit $P \subset NP$, i.e. les problèmes NP-complets ne sont pas **tous** polynomiaux;
- soit $P = NP ?$ n'est pas décidable ...



Que faire face à un problème NP

Lorsque l'on est amené à chercher à résoudre un problème NP, on a les deux options suivantes:

- ➡ soit on considère que le problème P est **facile**, et on cherche alors à en donner une résolution polynomiale.
- ➡ soit on considère que le problème P est **difficile**, et on cherche alors à monter qu'il est NP-complet, en essayant de montrer qu'un des problèmes NP-complets connus peut être polynomialement réduit à P



Que faire face à un problème NP-complet

Confronté à un problème NP-complet, on peut:

- ➡ chercher à en donner une **résolution polynomiale** (chances de succès très limitées, mais gloire assurée en cas de réussite !)
- ➡ vérifier que l'on n'est pas, en fait, intéressé par un **sous-problème**, qui pourrait alors être polynomial (par exemple, 3-SAT est NP-complet, mais 2-SAT est polynomial).
- ➡ **appliquer des schémas de résolution algorithmique efficaces** (diviser pour résoudre, programmation dynamique, ...) qui peuvent se montrer suffisants dans la pratique, si la probabilité des instances difficiles est faible (l'algorithme du simplexe à par exemple une complexité pire cas exponentielle, mais une complexité moyenne polynomiale).
- ➡ **mettre en œuvre des algorithmes approchés**, produisant efficacement des solutions avec une faible erreur.
- ➡ utiliser des approches à base de **machines parallèles**
- ➡ ...



**problèmes non
formalisables**

non décidables

difficiles

NP-difficiles

polynomiaux