Calculabilité - Décidabilité

Ce cours dépasse le cadre du programme de NSI, mais il aborde des sujets pouvant servir de support au grand oral. En particulier, le classement des problèmes de décision, les équations diophantiennes, les machines de Turing et les preuves d'existence de fonctions incalculables ne sont pas au programme.

I. Introduction

Programme en tant que donnée

Un compilateur, un débugueur, un interpréteur sont des programmes qui prennent en paramètres d'autres programmes. C'est aussi le cas d'un logiciel de téléchargement, ou d'un système d'exploitation qui va faire tourner d'autres programmes. Ainsi, tout programme peut être perçu comme une donnée pour un autre programme.

Calculabilité et décidabilité, deux notions proches

Étudier la calculabilité d'une fonction ou la décidabilité d'un problème, c'est chercher si cette fonction peut être calculée, ou si ce problème peut être résolu, à l'aide d'un algorithme.

II. Problèmes de décision et décidabilité

Définitions

Un **problème de décision** est un problème qui prend une entrée (une donnée) et renvoie en sortie oui ou non.

Un problème de décision est dit **décidable** s'il existe un algorithme qui, pour chaque entrée, réponde par oui ou par non à la question posée par le problème. S'il n'existe pas un tel algorithme, le problème est dit **indécidable**.

Un classement des problèmes de décision

1. **Problèmes ayant une solution en temps polynomial**
   - Deux entiers *n* et *p* sont-ils premiers entre eux ?
   - Un programme en Python est-il syntaxiquement correct ?
   - Un graphe est-il connexe ?

On note **P** l'ensemble des problèmes de décision résolubles en temps polynomial.

2. **Problèmes ayant une solution, mais pas en temps polynomial**
   - Une grille de Sudoku a-t-elle une solution ?
   - Un graphe donné peut-il être coloré avec trois couleurs sans que deux sommets voisins aient la même couleur ?
   - Étant donné un entier *d*, des villes et les distances entre les villes, existe-t-il un trajet passant par toutes les villes de longueur inférieur à *d* ? (problème du voyageur de commerce, TSP).

Problèmes NP

Entre les catégories 1 et 2, on peut considérer la catégorie des problèmes NP (Non déterministe Polynomial). Il s'agit des problèmes pour lesquels la vérification d'une solution candidate donnée est en temps polynomial, mais la recherche d'une solution est a priori en temps exponentiel. On se demande s'il existe pour ces problèmes des solutions en temps polynomial. C'est une des plus grandes questions ouvertes en informatique. Est-ce que P = NP ? Cela peut être un sujet de grand oral.

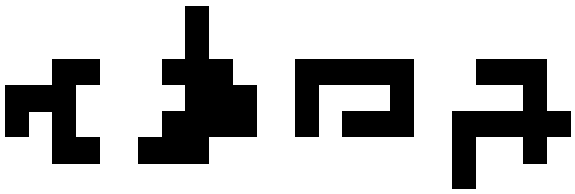
3. Problèmes pour lesquels on ne connait pas d'algorithmes

- Étant donné des entiers *n* et *k* et un chiffre *c*, existe-t-il une expression avec les opérateurs +, -, ×, /, √ et ! utilisant au plus *k* fois le chiffre *c* dont la valeur est *n* ? (Tchisla)

4. Problèmes indécidables (pour lesquels on sait qu'il n'existe pas d'algorithme)

- Peut-on paver le plan avec des polyominos de formes données ?

Un polyomino est une pièce type Tetris, c'est-à-dire constituée de carrés identiques juxtaposés.
Si certains polyominos permettent de former un rectangle, on pourra paver le plan, mais dans avec les pièces données ci-dessous, cela semble difficile.



En 1970, Golomb montre que ce problème est indécidable, qu'il n'existe pas d'algorithme acceptant un nombre fini de formes de polyominos en entrée et indiquant en sortie si on peut paver le plan avec ces formes.

- Une équation diophantienne donnée possède-t-elle une solution entière ?

Une équation diophantienne est une équation polynomiale à plusieurs variables à coefficient entiers, comme $2x - 4y + 6z = 0$, $x^2 - 4xy + y^3 = 8$ ou $x^3 + y^3 + z^3 = 42$ …

En 1900, David Hilbert demande s'il existe un algorithme capable de décider pour n'importe quelle équation diophantienne si elle possède ou non une solution. C'est le dixième problème d'une série de 23 problèmes difficiles qu'il pose au congrès international des mathématiciens à Paris. Ce n'est qu'en 1970 qu'un mathématicien russe, Iouri Matiassevitch, prouve qu'il n'existe pas un tel algorithme, donc qu'il s'agit d'un problème indécidable.
Cela ne signifie pas qu'une équation donnée ne peut pas être résolue, mais qu'il n'existe pas une méthode unique applicable à toutes les équations diophantiennes.

À propos des équations diophantiennes : problème de la somme de trois cubes

- $x^3 + y^3 + z^3 = 43$ admet pour solution $x = 2$, $y = 2$ et $z = 3$.
- $x^3 + y^3 + z^3 = 42$ est restée sans solution jusqu'en septembre 2019… !

Andrew Booker et Andrew Sutherland l'ont trouvé après une recherche exhaustive sur Charity Engine :
https://en.wikipedia.org/wiki/Charity_Engine
Des travaux mathématiques préalables leur ont permis de réduire les cas à explorer plutôt que de faire une simple triple boucle.
Plus d'un million d'heures de calcul parallélisé (114 ans) ont permis de trouver comme solution :

$x = -80538738812075974$
$y = 80435758145817515$
$z = 12602123297335631$.

En mars 2019, Andrew Booker avait trouvé une solution à $x^3 + y^3 + z^3 = 33$.

On connait maintenant des solutions à $x^3 + y^3 + z^3 = k$ pour tous les entiers k entre 0 et 1000 à l'exception des valeurs 114, 390, 627, 633, 732, 921 et 975 (au moment où j'écris ces lignes).

III. Calculabilité

Fonctions calculables

Intuitivement, une fonction *f* est une fonction calculable s'il existe un algorithme qui, étant donné un argument *x*, permet d'obtenir l'image *f*(*x*).

Une définition plus précise repose sur des formalisations mathématiques, dans lesquelles on définit les méthodes de calcul.
Dans les années 30 ont émergé plusieurs systèmes concurrents, dont principalement :

- la machine de Turing, développée par Alan Turing
- le λ-calcul, formalisme complètement différent proposé par Alonzo Church
- les fonctions récursives, système proposé par Stephen Cole Kleene

En 1938, Kleene montre l'équivalence des trois notions : les fonctions calculables sont les mêmes dans les trois systèmes. Ses travaux servent de base à la thèse de Church, ou thèse de Church-Turing, qui définit les fonctions calculables comme des fonctions qui se calculent avec une machine de Turing.

Machines de Turing

Une machine de Turing est un modèle théorique d'ordinateur simplifié muni d'une tête de lecture et d'écriture se déplaçant vers la droite ou la gauche sur les cases d'un ruban infini. Elle dispose d'un nombre fini d'états et d'une liste d'instructions qui constituent le programme.
Un exemple d'instruction : $\delta(i, 0) = (i, 1, \rightarrow)$
Cette instruction va s'appliquer si la machine est dans l'état *i* (état initial) et si la tête de lecture lit un 0 sur la bande. Alors, elle va rester dans l'état *i*, remplacer le 0 par un 1 puis la tête de lecture va se déplacer d'une case vers la droite.
Toute entrée d'un problème peut être codée en binaire sur la bande d'une machine de Turing.
Lorsque la machine s'arrête (s'il n'y a pas de boucle infinie), on lit le résultat sur la bande.

Le langage des machines de Turing est proche de l'assembleur. On peut d'ailleurs écrire un compilateur traduisant l'assembleur en langage des machines de Turing. Comme tout langage actuel peut être compilé en assembleur, il peut aussi être compilé en langage des machines de Turing.
Les fonctions calculables et les problèmes indécidables sont donc les mêmes dans tous les langages.

Fonctions incalculables

On démontre qu'il existe des fonctions incalculables, c'est-à-dire pour lesquelles il n'existe pas d'algorithme permettant de calculer la valeur *f*(*x*) pour toute entrée *x*.

Une première preuve

Infini dénombrable et infini non dénombrable

On s'appuie sur le fait qu'il existe des infinis de différentes tailles : un ensemble infini peut être dénombrable ou non dénombrable. Dénombrable signifie qu'on peut numéroter ses éléments.
L'ensemble des nombres entiers naturels {0, 1, 2, 3, …} est infini dénombrable.
L'ensemble des réels est infini non dénombrable. En effet, si $\mathbb{R}$ était dénombrable, on pourrait numéroter tous ses éléments, et donc en particulier numéroter tous les nombres de [0, 1[: le premier x_1, le deuxième x_2, le troisième x_3, …. Chacun de ces nombres x_n admet une écriture décimale du type :

$x_n = 0,a_{n,1}\, a_{n,2}\, a_{n,3}\ldots$ où $a_{n,k}$ est la *k*-ième décimale de x_n.

$x_1 =$	0,	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{1,5}$	…
$x_2 =$	0,	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$	$a_{2,5}$	…
$x_3 =$	0,	$a_{2,1}$	$a_{3,2}$	$a_{3,3}$	$a_{3,4}$	$a_{3,5}$	…
$x_4 =$	0,	$a_{3,1}$	$a_{4,2}$	$a_{4,3}$	$a_{4,4}$	$a_{4,5}$	…
$x_5 =$	0,	$a_{4,1}$	$a_{5,2}$	$a_{5,3}$	$a_{5,4}$	$a_{5,5}$	…
…	…	…	…	…	…	…	…

On définit alors un nombre $x = 0,a_1a_2a_3…$ en choisissant chaque décimale a_n différente de $a_{n,n}$.
Par exemple, on peut poser $a_n = 2$ si $a_{n,n} = 1$ et $a_n = 1$ sinon.
Alors ce nombre n'est dans aucune ligne du tableau, donc différent de tous les x_n, et pourtant il appartient bien à [0, 1[. On arrive à une contradiction.
On en déduit que l'intervalle [0, 1[, et donc $\mathbb{R}$, ne sont pas dénombrables.

Cette démonstration est due au mathématicien allemand Georg Cantor (1891).

L'ensemble des algorithmes est dénombrable.
En effet, un algorithme peut être écrit avec un nombre fini de caractères pris dans un alphabet fini, et les algorithmes peuvent être triés et numérotés : les algorithmes à 1 caractère, puis à 2, à 3… et on utilise l'ordre alphabétique pour trier entre eux les algorithmes ayant le même nombre de caractères.

L'ensemble des fonctions sur un domaine infini n'est pas dénombrable.
En effet, considérons par exemple les fonctions de $\mathbb{N}$ dans $\mathbb{N}$. On numérote toutes les fonctions $f_1, f_2, f_3, \ldots$.

f_1	$f_1(0)$	$f_1(1)$	$f_1(2)$	$f_1(3)$	$f_1(4)$	$f_1(5)$	…
f_2	$f_2(0)$	$f_2(1)$	$f_2(2)$	$f_2(3)$	$f_2(4)$	$f_2(5)$	…
f_3	$f_3(0)$	$f_3(1)$	$f_3(2)$	$f_3(3)$	$f_3(4)$	$f_3(5)$	…
f_4	$f_4(0)$	$f_4(1)$	$f_4(2)$	$f_4(3)$	$f_4(4)$	$f_4(5)$	…
f_5	$f_5(0)$	$f_5(1)$	$f_5(2)$	$f_5(3)$	$f_5(4)$	$f_5(5)$	…
…	…	…	…	…	…	…	…

On construit une fonction f qui à tout entier n associe un entier différent de $f_n(n)$.
On peut définir $f(n)$ simplement, par exemple :

- si $f_n(n) = 1$, on pose $f(n) = 2$
- sinon, on pose $f(n) = 1$.

Cette fonction existe bien, et elle est différente de toutes les fonctions f_n puisqu'elle n'est dans aucune ligne du tableau. Cette contradiction montre qu'on ne peut pas dénombrer les fonctions calculables.

L'ensemble des fonctions est donc « plus grand » que l'ensemble des algorithmes, ce qui prouve qu'il existe des fonctions incalculables.

Une deuxième preuve

Pour montrer qu'il existe des fonctions incalculables, il suffit d'en trouver une. La complexité de Kolmogorov fournit un exemple.

Complexité de Kolmogorov
La complexité de Kolmogorov est la fonction, qui à tout objet, associe la longueur du plus petit algorithme permettant de produire cet objet. Elle dépend du formalisme ou langage choisie.

Ainsi, en considérant des algorithmes écrits à l'aide de fonctions Python :

- 'abababababababababababababababab' s'obtient avec la fonction : f=lambda:'ab'*16
- 'kf2fWkieHMp56cwadZoG59floCb4t3H8' avec f=lambda:'kf2fWkieHMp56cwadZoG59floCb4t3H8'

On peut penser que la première chaine a une complexité de Kolmogorov de 16 et la seconde de 43.

La complexité de Kolmogorov est incalculable.
En effet, supposons qu'il existe une fonction kolmo qui, pour toute entrée s, renvoie sa complexité de Kolmogorov kolmo(s) et notons k le nombre de caractères du code source de cette fonction.
On considère l'algorithme suivant :

```
n ← 1
Tant que kolmo(n) < k + 100, faire:
   n ← n + 1
Fin Tant que
Renvoyer n
```

Cet algorithme renvoie le plus petit nombre qui a une complexité de Kolmogorov supérieure à k + 100.
Ce nombre existe puisqu'il n'y a qu'un nombre fini de programmes de taille plus petite que k + 100 et il y a une infinité de nombres entiers naturels.

Or, cet algorithme s'écrit avec moins de k + 100 caractères. Il est donc de complexité inférieure à k + 100 et il écrit un nombre de complexité supérieure à k + 100, ce qui est absurde.
Il n'existe donc pas de fonction qui calcule la complexité de Kolmogorov.

Le problème de l'arrêt fournit un autre exemple de fonction incalculable.

IV. Indécidabilité du problème de l'arrêt

On considère une fonction, qui pour tout algorithme et valeur d'entrée pour cet algorithme, indique si l'algorithme s'arrête ou pas. Cette fonction est incalculable, ce qui revient à dire que le problème de l'arrêt est indécidable. Ce résultat a été prouvé par Alan Turing en 1936.
En d'autres termes, il n'existe pas de programme qui prenne en argument n'importe quel programme avec une valeur d'entrée et qui, en temps fini, renvoie « oui » si l'exécution du programme reçu en argument finit par s'arrêter avec l'entrée spécifiée et « non » s'il ne finit pas.

Preuve classique
On suppose qu'il existe un programme halt, qui prend en argument le code d'un programme P (sous forme de chaîne de caractères) et un argument *x* de P et renvoie True si P s'arrête sur x, et False sinon.

On s'intéresse plus précisément aux programmes qui prennent en paramètre une chaine de caractères : *x* est donc une chaine de caractères. On considère la fonction suivante :

```
def contradiction(P: str):
    if halt(P, P):
        while True:
            print("Le code boucle")
    else:
        print("Le code s'arrête")
```

Que se passe-t-il quand on lance contradiction(code_contradiction) ?

Ou bien cet appel s'arrête : dans ce cas, halt(code_contradiction, code_contradiction) renvoie True et on entre dans une boucle infinie. C'est absurde !
Ou bien cet appel ne s'arrête pas : halt(code_contradiction, code_contradiction) renvoie False, puis le programme s'arrête !
Dans les deux cas, on aboutit à une contradiction. On en déduit que le programme halt ne peut pas exister.

Une version en vidéo en anglais : https://www.youtube.com/watch?v=92WHN-pAFCs