Calculabilité, décidabilité et complexité

La limite des ordinateurs

La puissance des ordinateurs ne cesse d'augmenter. L'intelligence artificielle peut faire des choses qui semblaient impossibles il y a quelques années. On peut donc se poser la question : *est-ce qu'il y a une limite à ce que peut faire un ordinateur ?* Cette question peut sembler quasi-philosophique, et pourtant elle a déjà été résolue il y a près d'un siècle.

Le théorème de l'arrêt

Nous écrivons régulièrement des programmes ou fonctions qui prennent des paramètres et renvoient les résultats correspondants. Ces données peuvent être de différents types et nous avons déjà vu que cela pouvait même être d'autres fonctions. De façon générale, les programmes sont des données comme les autres. Nous avons par exemple fait des programmes Python qui exécutaient des programmes de machines de Turing. On peut voir Thonny comme un programme Python qui prend d'autres programmes Python et les exécute.
Cela veut dire que l'on peut écrire des programmes qui prennent d'autres programmes et les analyse. Par exemple, Thonny vérifie que les programmes que vous écrivez ont une syntaxe correcte. C'est également ce que fait un compilateur avant de transformer le programme en code machine.
Imaginons que l'on ait un programme `halt(prog, entrees)` qui renvoie un booléen qui indique si l'exécution du programme `prog` va se terminer avec les entrées `entrees`.

Exercice : On considère le programme ci-contre.

1) Que va renvoyer `halt(decompte, 10)` ?
2) Que va renvoyer `halt(decompte, 4.5)` ?

```
def decompte(n):
    while n != 0:
        print(n)
        n = n - 1
    print("Go !")
```

Un tel programme peut sembler très pratique. Pourtant il ne peut pas exister. En effet, s'il existait, on pourrait écrire le programme ci-contre. Que se passerait si on appelle `sym(sym)` ? Il faudrait alors étudier `halt(sym, sym)`, qui va lui même déterminer si `sym(sym)` termine ou pas. Il y a alors 2 cas :

```
def sym(prog):
    if halt(prog, prog):
        while True:
            pass
    else:
        return 42
```

- Si `sym(sym)` termine, alors `halt(sym, sym)` va renvoyer `True` et donc `sym` va rentrer dans une boucle infinie et ne se terminera jamais.
- Si `sym(sym)` ne se termine jamais, alors `halt(sym, sym)` va renvoyer `False` et donc `sym` va renvoyer 42 et se terminer.

Dans tous les cas, `sym(sym)` se termine si et seulement si `sym(sym)` ne se termine pas. C'est donc une contradiction. Ce qui signifie que `halt` ne peut pas exister.
En 1936, Alan Turing et Alonzo Church ont tous les deux démontré, de façon indépendante, ce résultat, appelé le **problème de l'arrêt**. La conclusion est que ce problème est **indécidable**. Cela veut dire qu'il n'existe pas de méthode (mathématique, informatique, ou autre) capable de le résoudre. L'existence d'un tel problème est un problème en soit. Il y a donc des choses qu'un ordinateur ne peut pas faire, peut importe le langage ou la technologie utilisée.

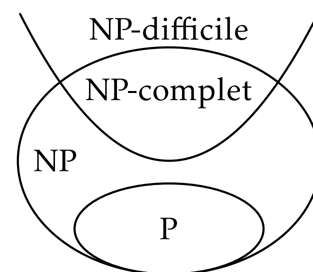
La calculabilité

Mais si un ordinateur ne peut pas tout faire, comment déterminer ce qu'il peut faire ? Là aussi, Turing et Church ont amené des réponses. Ils ont proposé deux modèles pour décrire ce qui était faisable de façon algorithmique. Turing a proposé son modèle de machine abstraite, appelée **machine de Turing**.
De son côté, Church a inventé le **lambda calcul**, une forme de langage fonctionnel qui permet d'écrire des algorithmes de façon formelle (rappelons qu'il n'y avait pas d'ordinateurs à l'époque). Il est basé sur la notation "$\lambda x.e$" qui correspond à "`lambda x: e`". Une fonction est dite **calculable** s'il est possible de l'écrire en lambda calcul. Turing a démontré que toute fonction calculable, au sens de Church, peut être représentée par un programme pour ses machines et réciproquement. On dit qu'un langage de programmation est **Turing complet** s'il est aussi puissant que les machines de Turing (ou le lambda calcul). Tous les langages "traditionnels" (C, Python, Java, ...) sont Turing complets. Même HTML+CSS est Turing complet et donc permet de réaliser n'importe quelle fonction calculable. La différence entre ces langages est uniquement la facilité avec laquelle ils vont permettre de résoudre tel ou tel problème.
De façon plus anecdotique, le jeu de cartes Magic est Turing complet, ainsi que le système de redstone de Minecraft ou les jeux Pokémon Jaune et Super Mario World, en exploitant des bugs permettant de programmer directement dans le jeu à l'aide de combinaisons de touches.

Décidabilité et complexité

Un **problème de décision** est un problème dont la réponse est oui ou non. Par exemple "*est-ce que ce nombre est premier ?*", "*est-ce qu'il existe un chemin de moins de* x km *permettant de relier deux villes données ?*" ou "*est-ce qu'il va pleuvoir demain ?*" sont tous des problèmes de décision. Un problème de décision est **décidable** s'il existe un algorithme capable de résoudre ce problème dans tous les cas possibles. L'existence d'un tel algorithme ne sous-entend rien sur le temps de calcul nécessaire pour obtenir la réponse. Par exemple, étant donné un graphe G et une distance d, il est possible de déterminer s'il existe un chemin passant par tous les sommets de G dont la distance est inférieure à d. Par contre, cela peut prendre beaucoup de temps, puisque ce problème a une complexité exponentielle, jusqu'à preuve du contraire. On appelle ce problème, le problème décisionnel du voyageur de commerce.
Afin de distinguer les problèmes "faciles" des problèmes "difficiles", on utilise les **classes de complexité**. Il en existe une multitude, mais nous allons nous concentrer sur les principales, concernant les problèmes de décision.
Les problèmes les plus simples sont classés dans **P**, la classe des problèmes décidables en temps polynomial par rapport à la taille des entrées. Par exemple, tester si un élément se trouve dans un tableau ou déterminer si deux tableaux contiennent les mêmes éléments sont des problèmes dans P.
La classe suivante s'appelle **NP**, pour *non-deterministic polynomial time*, c'est-à-dire les problèmes qui peuvent être résolus à l'aide d'une machine de Turing non déterministe. Une machine non déterministe peut "choisir" entre plusieurs instructions à exécuter et cette définition signifie qu'il y a une exécution possible qui résout le problème en temps polynomial. Par exemple, le problème décisionnel du voyageur de commerce est dans NP. S'il y a une exécution possible qui trouve une solution, la machine non déterministe la trouvera.

Cette définition n'est pas très à simple appréhender alors on utilise généralement la définition suivante : un problème est dans NP s'il est possible de vérifier une solution en temps polynomial. Étant donné un chemin, il est facile de tester qu'il passe par tous les sommets et que sa longueur totale est inférieure à la borne fixée. Par contre, rien n'est demandé sur le cas où il n'y a pas de solution.



On a trivialement P ⊆ NP. Par contre le fait de savoir si P = NP ou P ≠ NP est un problème à un million de dollars. Le consensus est quand même en faveur de l'inclusion stricte. Cela repose sur le fait que certains problèmes sont dans NP sans qu'on sache les résoudre en temps polynomial. On appelle **NP-complet** l'ensemble des problèmes de NP qui sont plus durs que tous les autres problèmes de NP. Pour vérifier cela, on utilise le principe de la **réduction polynomiale**.

Considérons une instance du problème du sac à dos décisionnel : étant donné une liste d'objets, est-il possible de prendre plus qu'une valeur donnée en ne dépassant pas une limite de poids ? Imaginons que l'on puisse transformer cette instance en une instance du problème du voyageur de commerce en temps polynomial de telle sorte que la réponse soit `True`, si et seulement si la réponse pour le problème du sac à dos soit `True` aussi. Alors si on peut résoudre le problème du voyageur, on peut aussi résoudre le problème du sac à dos. Mais cela ne veut pas dire qu'on ne peut pas résoudre le problème du sac à dos sans savoir résoudre l'autre. Le problème du voyageur est donc au moins aussi dur que celui du sac à dos.

Mais ce n'est pas fini. Les problèmes plus durs que les problèmes NP-complets sont dits **NP-difficiles**. C'est le cas du problème de l'arrêt. C'est aussi le cas de tous les problèmes indécidables. Les problèmes de recherche de solutions optimales pour le voyageur de commerce ou le sac à dos sont également NP-difficiles.

Cryptographie et ordinateurs quantiques

Pour les problèmes sur lesquels reposent la cryptographie, on aimerait bien qu'ils soient NP-complet ou NP-difficile. Malheureusement, ce n'est pas le cas. Prenons le problème décisionnel associé à la factorisation : *"est-ce qu'il existe un facteur de n inférieur à k ?"*. Ce problème est dans NP, puiqu'on peut facilement vérifier qu'un facteur satisfait le critère demandé. Mais, à moins d'avoir P = NP, il ne peut pas être NP-complet. En effet, il est également dans Co-NP, puisqu'une factorisation en nombre premiers permet de vérifier qu'il n'y a pas de facteurs inférieurs à k. Et les problèmes qui sont à la fois dans NP et Co-NP ne peuvent pas être NP-complets. C'est le cas de la plupart des problèmes sur lesquels reposent les algorithmes de chiffrement.

Mais être NP-complet n'est peut-être pas souhaitable pour la cryptographie. En effet, les problèmes NP-complets le sont quand on les considère dans leur généralité, ce qui ne veut pas dire que la plupart des cas sont facilement décidables. Pour la cryptographie, on veut que les problèmes soient difficiles en général. Il faut aussi que la création des clefs, le chiffrement et le déchiffrement restent suffisamment rapide pour pouvoir être utilisés. Il faut donc bien choisir son problème.

Les ordinateurs quantiques, par contre, peuvent être un vrai danger pour les algorithmes de chiffrement. En effet, les problèmes utilisés sont dans BQP, les problèmes qui peuvent être résolus en temps polynomial par un ordinateur quantique. NP-Complet, par contre, n'est pas dans BQP. Il existe donc des problèmes qui résistent aux ordinateurs quantiques. Il y a d'ailleurs déjà des algorithmes de chiffrement quantiques qui existent, et qui permettront de prendre le relai lorsque les ordinateurs quantiques deviendront suffisamment puissants.