par Nicolas K. Blanchard

Non, les problèmes ne sont pas tous de même difficulté !

Des problèmes comme trier des nombres ou trouver le plus court chemin entre deux points posent une question naturelle : certains sont-ils plus compliqués que d'autres ? La théorie de la complexité permet de classifier les problèmes en ensembles de problèmes de difficulté similaire.

Vous savez tous additionner deux nombres A et B à n chiffres : on somme les chiffres deux par deux en conservant la retenue jusqu'à obtenir un nouveau nombre à n ou $n + 1$ chiffres, en faisant en gros n opérations. Par contre, multiplier ces deux nombres avec la méthode habituelle correspond à faire n^2 opérations. La *complexité en temps* d'un algorithme correspond au nombre d'opérations élémentaires faites en fonction de la taille de l'entrée. Ici, on compte la taille de l'entrée en nombre de chiffres (ou *bits* en binaires), et l'opération élémentaire est l'addition de deux chiffres, ainsi que la lecture–écriture d'un chiffre. Compter tout cela de manière explicite est généralement infaisable ; on utilise la notation de Landau, qui permet de ne pas se soucier des constantes. Un algorithme s'exécute en temps $O(n)$ s'il existe une constante k telle que l'algorithme ne fasse jamais plus de kn opérations (pour n suffisamment grand). De même, un algorithme s'exécute en $\Omega(n)$ s'il faut au moins $k'n$ opérations, et enfin en $\theta(n)$ s'il faut toujours entre kn et $k'n$ opérations. L'addition s'exécute donc en $\theta(n)$, et la multiplication usuelle en $\theta(n^2)$.

La théorie de la complexité mérite bien son nom.

Qui veut gagner un million ?

Existe-t-il une différence de nature entre ces deux problèmes ? Cela dépend en réalité du « niveau de zoom » considéré. D'un point de vue pratique, où les constantes comptent, il y a une vraie différence, mais pour un théoricien de la complexité, les facteurs polynomiaux ne sont souvent pas si importants. En effet, une des plus importantes classes de problèmes, la classe P (pour « temps po-

lynomial », voir plus loin), correspond aux problèmes pouvant être résolus en faisant au plus un nombre d'opérations polynomial en la taille de l'entrée, et un facteur 2 de différence a donc peu d'importance. À ce niveau de zoom, la différence se fait entre le temps polynomial et le temps exponentiel, ou bien le temps polynomial sur des machines spéciales, appelées *non-déterministes*. Si la classe P (pour *polynomial*) est la classe des problèmes « raisonnables », dont on peut trouver la solution en temps polynomial, la classe NP (pour *non déterministe polynomial*) est celle des problèmes dont on peut « deviner » cette solution en temps polynomial. De manière équivalente, c'est la classe des problèmes dont on peut vérifier en temps polynomial qu'une solution proposée est effectivement une solution du problème. Le problème fondamental de l'informatique théorique, P *versus* NP, revient à ce demander si ces deux classes coïncident ou non. Trouver la solution étant *a priori* plus difficile que la vérifier, l'avis est généralement que $P \subset NP$ (le symbole signifiant dans cet article une inclusion stricte).
Le fait que ce problème ait résisté plus de quarante ans malgré des grandes récompenses offertes pour une preuve (dont un million de dollars par le Clay Mathematics Institute) laisse planer le doute. Les conséquences en cas d'égalité ($P = NP$) seraient majeures : un bond en avant pour certaines sciences, mais potentiellement aussi une disparition plus ou moins complète de la cryptographie, et donc des services bancaires et de la sécurité sur Internet. Une belle pagaille !

Pour classifier tous ces problèmes, il faut disposer d'une manière directe de comparer leur résolution : un modèle de calcul. Le modèle le plus utilisé, la machine de Turing déterministe, introduite en 1936, permet d'exécuter tout algorithme pouvant être conçu. Cette machine possède plusieurs caractéristiques : plusieurs rubans infinis sur lesquels elle peut lire et écrire des symboles ; une tête de lecture se déplaçant sur ces rubans ; un nombre fini d'états ; des instructions lui précisant dans quel sens déplacer la tête

Machine de Turing et avatars

Si la machine de Turing peut exécuter tout calcul imaginable, elle n'est toutefois pas le seul modèle utilisé. Tout d'abord, les ordinateurs ne suivent pas exactement ce modèle, et les machines RAM en sont plus proches (dans celles-ci, la tête de lecture n'a pas à se déplacer le long du ruban mais peut simplement sauter d'un endroit à un autre). La richesse cependant vient de l'étude des modèles théorique *a priori* non réalisables qui donnent de l'intuition sur le cas pratique. Ainsi, quand on parle de la classe de problème NP, l'une des représentations possible est sous la forme de machines de Turing dont les transitions (instructions) ne sont pas déterministes. Cela signifie par exemple que la machine a le choix entre plusieurs transitions possibles et « devine » en quelque sorte la bonne transition. On peut étudier de nombreux modèles : les machines probabilistes qui peuvent utiliser des *bits* aléatoires, les machines quantiques, dont les états sont remplacés par un espace de Hilbert (et sont donc en potentielle infinité), ou enfin les machines à oracle.

de lecture, son futur état et ce qu'elle doit écrire là où elle se trouve selon ce qu'elle vient de lire et son état actuel.

Munis de cette machine, et des instructions correspondant à un algorithme, il devient possible d'étudier le nombre d'étapes de calcul – complexité *en temps* – ou l'espace nécessaire sur les rubans de lecture – complexité *en espace*. Comme il existe un nombre infini de machines permettant de calculer le même algorithme, laquelle doit-on choisir ? En augmentant le nombre d'états de la machine et la quantité d'instructions, on pourrait en effet très bien augmenter la vitesse en mettant en quelque sorte toutes les réponses en mémoire. En fait, comme la machine n'a qu'un nombre fini d'états, le seul critère important est le comportement asymptotique quand n est grand, qui n'est pas influencé par ces quelques cas en mémoire.

Concentrons-nous sur les problèmes de décisions, comme l'existence d'un chemin de longueur k entre deux points, où la machine doit répondre « oui » ou « non » selon l'entrée. On peut ainsi définir un *langage*, qui est l'ensemble des entrées sur lesquelles la machine doit répondre oui. Une machine reconnaît un langage L si, et seulement si, elle s'arrête et répond oui sur les mots de L et uniquement sur ceux-ci (certaines machines pouvant boucler sans fin, cette condition d'arrêt est importante). Si une machine M reconnaît un langage L, reconnaître co-L, le complémentaire du langage L, revient à inverser les réponses. C'est très facile à faire si M est déterministe et termine toujours (il suffit de répondre oui à la place de non et réciproquement !), mais *a priori* impossible dans d'autres cas, comme quand M est non-déterministe ou probabiliste.

Éliminer les preuves erronées

Il existe plus de cent preuves annoncées résolvant le fameux problème P versus NP, nulle ne résistant cependant à un examen approfondi. Peut-on donc rapidement en éliminer certaines, pour ne pas avoir à tout vérifier ? Eh bien oui, grâce à un résultat de Theodore Baker, John Gill et Robert Solovay, qui énonce que toute preuve relativisante est impossible. Pour comprendre ce qu'est une preuve relativisante, il faut d'abord introduire la machine à oracle. C'est comme une machine de Turing habituelle, qui disposerait d'un ruban supplémentaire et de trois états de plus : q_{oui}, q_{non}, et q. Quand la machine va dans l'état q, elle demande à l'oracle si le mot inscrit sur le ruban spécial est dans le langage de l'oracle, et va en q_{oui} si c'est le cas et en q_{non} sinon. Une machine avec un oracle pour le langage L est notée M^L et par extension la classe des machines à oracle L répondant en temps polynomial est notée P^L. Le théorème montre que l'on peut trouver deux langages L et L' tels que $P^L = NP^L$, et $P^{L'} \neq NP^{L'}$, et cela sans dépendre de la réponse à P versus NP.

Si les méthodes dans une preuve de P *versus* NP sont toujours valables en passant du modèle habituel au modèle à oracle, on obtient une contradiction, donc une erreur dans la preuve annoncée. Ces méthodes, appelées *relativisantes*, ne peuvent donc pas résoudre P *versus* NP.

Position gagnante aux échecs

Revenons aux deux problèmes initiaux. Il existe des machines de Turing les résolvant en $O(n)$ et $O(n^2)$ réciproquement, et toute machine les résolvant prend au moins un temps $\Omega(n)$ car elle doit au moins lire l'entrée. On dit alors que ces problèmes appartiennent respectivement à $\text{DTIME}(n)$ et $\text{DTIME}(n^2)$, où $\text{DTIME}(f(n))$ est la classe des

Approximer un problème difficile

Il est parfois intéressant de disposer d'une solution approximative à un problème de la classe NP. Prenons le cas du voyageur de commerce : on cherche à minimiser la distance à parcourir pour visiter un ensemble de villes. On peut vouloir essayer de ne pas avoir à parcourir plus de deux fois la distance optimale. Si les villes sont dans un espace euclidien, on peut avoir cette 2-approximation (une solution dont le coût est au plus deux fois pire que la meilleure) en temps $O(n \log n)$, où n est le nombre de villes. Par contre si les distances ne suivent pas l'inégalité triangulaire, il est impossible (à moins que P = NP) d'avoir un algorithme qui donne toujours une solution ayant un coût au plus 123 / 122 fois plus grand que la solution optimale.
Pour montrer qu'un problème n'est pas approximable, on peut faire une réduction vers un problème connu comme étant difficile. Si l'on cherche à montrer qu'un problème de minimisation de coûts A est inapproximable, on peut trouver un problème NP-complet B, faire correspondre à chaque instance « oui » de B une instance de coût 1 de A, et à chaque instance « non » une instance de coût 3. Si l'on avait un algorithme donnant une 2-approximation sur ces nouvelles instances, il nous permettrait de différencier entre les instances « oui » et les instances « non » du problème NP-difficile en temps polynomial.

problèmes pouvant être résolus en temps au plus $f(n)$ par une machine de Turing déterministe. Pour permettre à la machine de lire l'entrée, on se restreint ici à des fonctions $f(n)$ « gentilles » et plus grandes que n, que l'on appelle *fonctions constructibles en temps*. Si $f(n) \leq g(n)$, alors $\mathrm{DTIME}(f(n)) \subseteq \mathrm{DTIME}(g(n))$, mais la question est de savoir si cette inclusion est stricte. On ne peut pas répondre *a priori*, car par exemple $\mathrm{DTIME}(2n) = \mathrm{DTIME}(n)$, mais un résultat théorique, le *théorème de hiérarchie temporelle*, montre que $\mathrm{DTIME}(f(n)) \subset \mathrm{DTIME}(f(n)\log^2(n))$. Cela montre qu'avec plus de temps on peut résoudre plus de problèmes ! Comme avoir une classe $\mathrm{DTIME}(f(n))$ pour chaque fonction f serait ingérable, on aimerait bien les rassembler en classes plus grandes ayant un sens. Considérons notamment les problèmes « raisonnables ». Quels sont les caractéristiques d'un problème « raisonnable » ? Tout d'abord, augmenter un peu la taille de l'entrée ne doit pas trop augmenter le temps de calcul. Ensuite, si l'on dispose d'un algorithme résolvant un problème raisonnable et que l'on peut utiliser cet algorithme plusieurs fois pour résoudre un autre problème, cet autre problème paraît lui aussi raisonnable. On est en train de construire la classe P des problèmes résolubles en temps polynomial, correspondant à $\bigcup_{k\in\mathbb{N}} \mathrm{DTIME}(n^k)$. La majorité des problèmes de la vie de tous les jours rentrent dans ce cadre. Certains n'y sont pourtant pas, comme de nombreux problèmes d'optimisation, ou le problème SAT. On peut aussi définir la classe EXPTIME, correspondant aux problèmes nécessitant un temps au plus exponentiel en un polynôme en n.

Un problème central : SAT

Une conséquence intéressante de ces différentes notions est l'apparition de langages dits complets. Un langage L est *complet* pour une classe si l'on peut

Les preuves sans connaissances

Quels protocoles utiliser pour prouver son identité sans révéler d'informations ? Cette question peut se formaliser sous la forme de protocoles nommés *Arthur–Merlin*, où Merlin donne une information à Arthur, qui peut poser des questions. Merlin (le prouveur) est d'une puissance illimitée mais n'est pas nécessairement honnête. Arthur (le vérificateur) doit vérifier si un mot x est dans un langage L, avec l'aide des informations fournies par Merlin, d'un temps de calcul limité, et de *bits* aléatoires. On peut définir un protocole avec les contraintes suivantes : si $x \in$ L, Merlin peut répondre de manière à ce qu'Arthur accepte dans deux tiers des cas. Si $x \notin$ L, quoi que dise Merlin, Arthur refusera dans deux tiers des cas.

On regarde les langages qui peuvent être reconnus par un protocole ayant k échanges (ce qui définit la classe AM[k]). Un premier résultat surprenant est que le fait de montrer ou de cacher les bits aléatoires (et donc les calculs que fait Arthur dans son coin) ne change presque rien : s'il suffit de k étapes pour reconnaître L avec des bits aléatoires privés, il y a un protocole avec bits publics nécessitant $k + 2$ étapes.

Un résultat important dû à Adi Shamir énonce que IP = PSPACE, où IP correspond à la classe des protocoles dont le nombre d'étapes est polynomial en la taille de l'entrée x (IP pour temps *polynomial interactif*). Pour rendre ces protocoles « sans connaissance » (voir dans le dossier suivant), il faut de plus qu'Arthur n'obtienne pas plus d'informations que nécessaire (par exemple, pas de preuve détaillée que $x \in$ L). En faisant une hypothèse raisonnable (l'existence d'un encodage fort), il se trouve alors que tout problème NP admet un protocole IP.

réduire tout problème de cette classe à une instance d'appartenance à L après un certain prétraitement. Une réduction de L' à L est un algorithme transformant une question d'appartenance à L' en une question d'appartenance à L équivalente. Vu que l'on autorise généralement le prétraitement à prendre un temps polynomial, les problèmes P-complets ne sont pas si intéressants, et on s'occupe plutôt de problèmes NP-complets. Cependant, avec cette même contrainte, il existe des langages EXPTIME-complets, comme celui des positions gagnantes aux échecs ou au go (la version généralisée de ces jeux sur des plateaux de taille n). Cela veut dire que tout problème pouvant être résolu en temps exponentiel est équivalent à la question : *« Les noirs gagnent-ils sur cette position ? »* sur un plateau précis. Si l'on prend une formule de logique propositionnelle, comme (a et b) et (c ou d), il est naturel de savoir s'il existe une assignation des variables satisfaisant la formule. On peut en fait se restreindre au problème de la satisfiabilité de formules sous forme normale conjonctive, où les formules sont du type (A et B et C), où A B, C sont des « clauses », chacune de la forme (x ou y ou z). Ce problème, appelé SAT (pour *satisfiabilité*), est fondamental : il est trivialement dans NP (il suffit de deviner la bonne assignation et de vérifier qu'elle convient), il est NP-complet (ce fut même le premier problème prouvé NP-complet), il sert le plus souvent à montrer que d'autres problèmes sont NP-complets, et il est *auto-réductible* (si l'on sait répondre s'il existe ou non une solution, on sait aussi trouver une solution). Cette dernière propriété, essentielle, fonctionne ainsi en pratique : supposons qu'une formule F soit satisfiable et que l'on ait un algorithme décidant SAT. On lance l'algorithme sur (F et x) ; si cette formule est toujours satisfiable, on fixe x à vrai, sinon à faux, et on procède ainsi pour toutes les variables pour trouver une assignation correcte au final.

Cela implique que quel que soit le problème dans NP, montrer qu'une solution existe n'est pas vraiment plus difficile que de trouver la solution : il suffit d'exprimer le problème sous une forme adaptée à SAT (ce qui est toujours possible).
On ne sait pas aujourd'hui résoudre SAT en temps inférieur à 2^{cn} pour $c \simeq 0{,}3$. Montrer qu'un temps exponentiel est effectivement nécessaire aurait pour conséquence que $P \neq NP$.

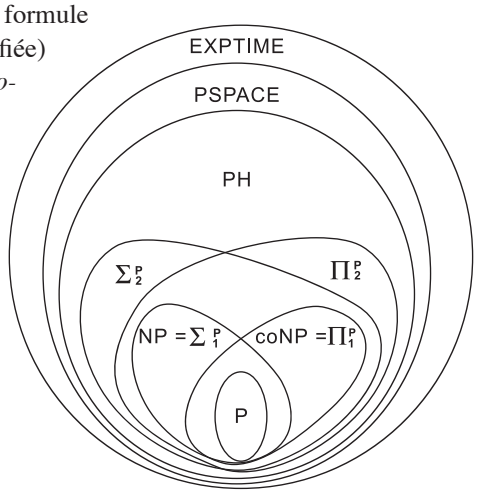
Des inclusions en cascade

Résoudre SAT revient à décider si une formule du premier ordre de la forme $\exists$ V, F est vraie, où F est une formule en forme normale conjonctive, et V est l'ensemble de variables de F. On peut donc se demander à quoi correspondent les formules $\forall$ V, F, et il se trouve que résoudre ces formules-là correspond à un problème coNP-complet. Ici, peut importe le nombre de quantificateurs, tant qu'ils sont du même type. Une formule du type $\forall\ V_0\ \exists\ V_1$ F alterne les quantificateurs, et la complexité augmente en conséquence, ce problème étant complet pour une nouvelle classe appelée $\sum_0^P$. On peut ainsi construire deux séries de formules selon le nombre d'alternances et le type du premier quantificateur. Les classes de complexité respectives, $\sum_i^P$ et $\prod_i^P$ (où $i > 0$), s'imbriquent les unes dans les autres et leur union forme la hiérarchie polynomiale.

On peut aussi décider de regarder l'espace que prend la machine (plutôt que le temps), ce qui produit un autre type de classes de complexités, toutefois lié au premier type. Déjà, une machine prenant un temps $f(n)$ ne peut occuper que $f(n)$ cases mémoires, d'où $\mathrm{DTIME}(f(n)) \subseteq \mathrm{DSPACE}(f(n))$. On a aussi une inclusion dans l'autre sens car si une machine ne peut utiliser qu'une mémoire bornée, elle n'a alors qu'un nombre borné de configurations possibles (une configuration est la donnée d'un état interne et de tout ce qui est écrit sur les rubans). Cela veut dire que ou bien la machine termine en temps inférieur au nombre de configurations possibles, ou bien elle passe deux fois par le même état (ce qui signifie qu'elle boucle et ne termine pas). Comme le nombre de configurations est au plus exponentiel en la taille disponible sur les rubans, cela donne l'inégalité suivante :
$\mathrm{DSPACE}(f(n)) \subseteq \mathrm{DTIME}(\exp(f(n)))$.
On peut alors écrire la série d'inclusions suivante :

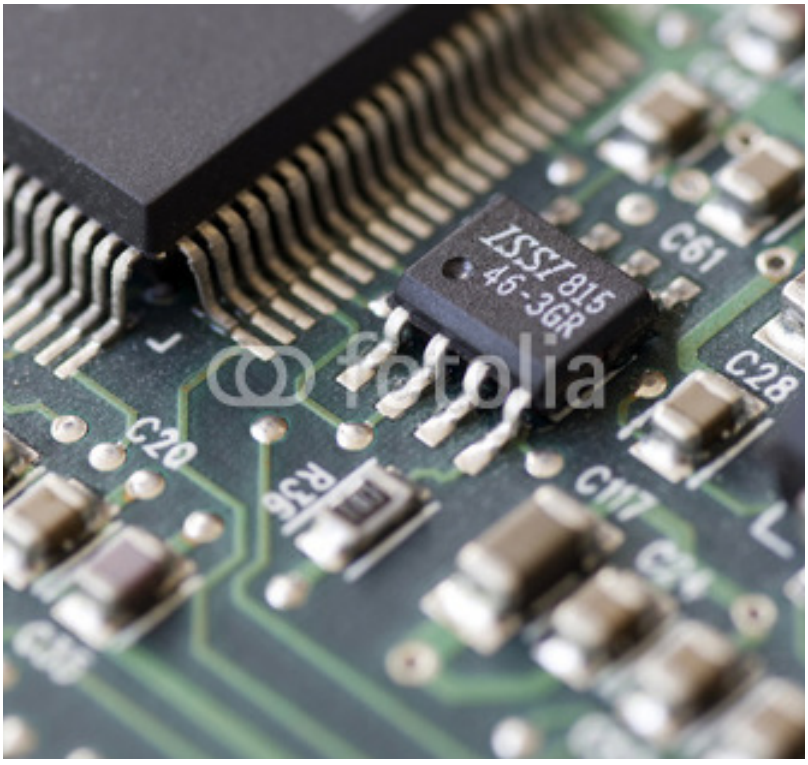
$$P \subseteq NP \subseteq \sum_i^P \subseteq PH \subseteq PSPACE \subseteq EXPTIME,$$

où PSPACE est l'ensemble des langages pouvant être décidés avec un espace polynomial (donc l'analogue de P). Tout ensemble de formule à alternance bornée est dans PH, mais le problème généralisé sans bornes sur le nombre d'alternances (problème dit QBF, pour formule booléenne quantifiée) n'appartient *a priori* pas à PH, étant PSPACE-complet. On sait aujourd'hui qu'au moins une de ces inégalités est stricte, et de nombreux experts conjecturent qu'elles le sont toutes.



Électronique et modèles de circuits

Beaucoup plus proche de la réalité physique, le *modèle des circuits* a suscité de l'intérêt ces dernières années. On se préoccupe comme en électronique de circuits composés de portes ET, OU et NON, acceptant n bits en entrée et sortant un bit, correspondant à oui (1) ou non (0). Un premier problème est qu'il faut un circuit différent pour chaque taille d'entrée, et donc on a plutôt tendance à étudier des familles de circuits. Pour mesurer la complexité de ces circuits, regardons le nombre total de portes – qui peuvent ne pas être binaires (par exemple, un OU sur n *bits* répond 1 si au moins un des *bits* vaut 1) – ou bien la profondeur (nombre maximal de portes entre l'entrée et la sortie). Par exemple, la classe ACC_0 correspond aux circuits qui possèdent un nombre polynomial de portes ET, OU, NON et MOD (qui calcule la somme de k bits modulo m, à m fixé), et qui sont de profondeur bornée. Même si ces circuits ont une puissance faible (*a priori*, le problème de savoir si la majorité des *bits* vaut 1 n'est pas dans ACC_0), prouver que cette puissance est faible est une autre paire de manches ! Ce n'est qu'en 2011 que Ryan Williams a montré que ACC_0 était strictement incluse dans NEXPTIME (temps exponentiel non déterministe), et de nombreuses inégalités du même type sont encore à l'état de conjecture.



Il existe même un résultat montrant les limites de ces méthodes, le *théorème de Razborov–Rudich sur les preuves naturelles*. L'idée derrière est que pour prouver que P est différent de NP à base de circuits on procède généralement en trois étapes. On fixe d'abord une certaine notion de « variabilité » d'une fonction booléenne (une fonction linéaire est peu variable, une fonction ressemblant à une fonction aléatoire est au contraire d'une grande variabilité). On montre ensuite que les circuits de taille polynomiale ne peuvent calculer que des fonctions de variabilité faible (comme les fonctions linéaires). On termine en montrant que SAT a une forte variabilité. Le problème est que, sous une hypothèse raisonnable (l'existence de fonctions pseudo-aléatoires), une telle preuve, appelée *preuve naturelle*, ne peut pas fonctionner. Tous ces problèmes ouverts et ces conjectures, beaucoup plus fréquents que les rares théorèmes (souvent accompagnés de preuves longues et techniques d'ailleurs…) font que la théorie de la complexité mérite bien son nom.

N.B.