La démonstration automatique

Nguyễn Lê Thành Dũng

Résumé

S'assurer automatiquement qu'une proposition mathématique est vraie est, en théorie, impossible en général. Pourtant, dans certains cas, ça marche très bien en pratique, et la vérification de programmes informatiques fournit le principal domaine d'application de ces méthodes.

(Note : les appendix A et B sont censés être des encadrés.)

1 Quelques distinctions préliminaires...

Qu'est-ce qu'une « preuve par ordinateur » ? On peut s'appuyer sur la machine pour

1. faire une partie du travail d'une démonstration rédigée par un humain, notamment des calculs mécaniques ;
2. vérifier que des fautes de raisonnement ne se sont pas glissées dans une preuve formelle fournie à l'ordinateur par l'humain ;
3. parvenir sans assistance à déterminer et à garantir qu'une conclusion est vraie.

C'est ce dernier cas dont nous allons parler ici. Le premier cas a fait couler beaucoup d'encre depuis quarante ans et l'encadré page (réf appendix A) revient très brièvement dessus. Le second est pris en charge par des logiciels appelés *assistants de preuve* (réf articles Curry-Howard et HoTT).

Le but d'un *prouveur automatique de théorèmes* est, à partir d'un énoncé mathématique, d'établir s'il est démontrable ou non à partir de certains axiomes. Ce problème a été posé par David Hilbert, qui lui a donné le nom de *problème de la décision*, en 1928 pour les énoncés de la logique dite « du premier ordre », qui englobe les mathématiques courantes. Alonzo Church, et indépendamment son élève Alan Turing, ont prouvé en 1936 qu'il était impossible à résoudre par un algorithme – dans ce qui constitue d'ailleurs l'acte fondateur de l'informatique théorique ! (voir aussi l'article Les limites de la preuve). Cependant, restreindre le problème permet de s'en sortir.

2 Le problème SAT

Regardons le cas de la *logique propositionnelle*, dont les énoncés sont formés à partir de *variables*, représentant des propositions qui peuvent être vraies ou fausses, combinées à l'aide des connecteurs logiques habituels (« et », « ou », « non », « implique », etc.). Il s'agit alors simplement de déterminer si une formule donnée est *toujours* vraie – on dira alors qu'elle est *valide*. C'est le cas par exemple de « non (A et (non A)) » car une proposition ne peut pas à la fois être vraie et fausse. Ce problème peut être résolu par un algorithme, qu'il est même facile d'appliquer à la main : on écrit la *table de vérité* (cf. article « Les bases de la logique ») de la formule et on regarde si toutes les lignes sont à « vrai » !

Ce faisant, on se heurte rapidement au problème suivant : pour s'assurer que « non (A et (non A)) » est valide, il suffit de vérifier pour A vraie et A fausse ; pour $((A \Rightarrow B) \Rightarrow A) \Rightarrow A$, il faut déjà regarder les 4 combinaisons A vraie et B vraie, A vraie et B fausse, A fausse et B vraie, et enfin A fausse et B fausse ; pour une formule faisant intervenir 100 variables, le nombre de lignes de la table de vérité qu'il faudrait traiter est de $2^{100} \simeq 10^{30}$, c'est de loin hors de portée des ordinateurs dont nous disposons ! Notre algorithme n'est pas capable de nous répondre *en un temps de calcul raisonnable*.

Une meilleure technique est de rechercher un contre-exemple, c'est-à-dire un choix des valeurs de vérité des variables rendant fausse la formule. Les algorithmes qui procèdent de cette façon vont essayer de deviner les valeurs prises par une partie des variables dans un contre-exemple. En effet, il arrive souvent que lorsque certaines des variables sont fixées, la formule soit vraie ou fausse indépendamment des valeurs prises par les autres. Dans le premier cas, on sait que ce n'est pas une bonne piste pour trouver un contre-exemple, et on peut donc se mettre à chercher ailleurs. Et dans le second cas, on a trouvé notre contre-exemple : il suffit de choisir arbitrairement pour le reste des variables. Cet angle d'attaque permet de « prendre des raccourcis » dans l'exploration du nombre *exponentiel* de possibilités, aboutissant à la conclusion que la formule est soit fausse pour certaines valeurs des variables, soit valide lorsqu'on a épuisé tous les contre-exemples possibles.

En pratique, les méthodes esquissées ci-dessus, et bien d'autres techniques et optimisations résultant de plusieurs décennies de recherche et d'ingénierie, sont mises en œuvre dans des logiciels nommés *solveurs SAT*. Ceux-ci sont entièrement dédiés à la résolution du *problème SAT*, qui consiste à trouver un choix de valeurs de variables qui *satisfasse* une formule, autrement dit la rende vraie, ou d'indiquer si c'est impossible. (Pour vérifier qu'une formule est valide, on demandera donc à un solveur SAT de s'assurer que sa négation est insatisfaisable.) Les plus performants arrivent à résoudre en moins d'une seconde de nombreux cas comportant plusieurs centaines de variables ! Mais pas tous : SAT fait partie de la famille des problèmes dits NP-complets, réputés difficiles ; un des « problèmes du millénaire », avec

une récompense d'un million de dollars à la clé, est de prouver qu'aucun algorithme ne peut résoudre un problème NP-complet efficacement dans tous les cas.

Ce cadre très restreint pour la démonstration automatique mène déjà à de nombreuses applications : combinatoire (cf. encadré A), certification de circuits électroniques, etc. Et malgré la NP-complétude, les solveurs SAT jouent très bien leur rôle en pratique.

3 La satisfaisabilité au-delà du cadre propositionnel

Pour faire quelque chose qui ressemble à des vraies maths, il faut tout de même pouvoir traiter des énoncés qui parlent de nombres, d'opérations arithmétiques, d'équations, etc. Prenons donc nos variables dans les formules et remplaçons-les par des propositions telles que $3x + 5y = 42$! Satisfaire une telle formule signifie alors trouver des valeurs des variables x, y, etc. rendant la formule vraie – sauf que les variables désignent ici non pas des propositions, mais des objets mathématiques de la théorie considérée : dans notre exemple, des nombres.

Pour résoudre ça, on ne va évidemment pas énumérer l'infinité de nombres possibles. On va s'y prendre de la même façon que pour résoudre SAT, en déterminant les valeurs de vérité prises par les propositions. Mais là où on aurait choisi « A vrai et B faux » (A, B étant des variables) pour satisfaire une formule propositionnelle, on aura ici « $3x + 5y = 42$ vrai et $x \geq 0$ faux » (par exemple) et il faut déterminer si ce choix sur les propositions peut être réalisé par un choix de nombres x, y. Pour cela, nous aurons besoin d'un programme auxiliaire qui permet de trouver une solution à une conjonction des propositions : une *procédure de décision*. En l'occurrence, il suffit de résoudre un système d'équations et d'inéquations linéaires, ce qu'un ordinateur sait très bien faire !

C'est la procédure de décision qui donne un sens à la formule au-delà des simples symboles logiques « et », « ou », etc. C'est celle-ci qui est chargée de s'occuper de toutes les choses spécifiques à la théorie dans laquelle on travaille, alors que le reste de l'algorithme ne sait s'occuper que de logique pure. Si l'on se dit qu'une démonstration d'un énoncé est un enchaînement d'étapes de raisonnements liées par les règles du raisonnement et partant des axiomes de la théorie, il faut voir solveur SAT et procédure de décision comme les analogues respectifs des règles logiques et des axiomes. Les deux interagissent subtilement dans des programmes nommés *solveurs SMT* (pour « SAT modulo théories »), qui défient des difficultés théoriques encore plus effrayantes que la NP-complétude pour satisfaire le genre de formules qu'on s'est posé plus haut.

Les possibilités des procédures de décision sont évidemment limitées :

ainsi, Iouri Matiassevitch a montré en 1970 qu'il ne pouvait pas en exister une qui marche pour toutes les équations diophantiennes (équations algébriques dont on cherche des solutions entières). La démonstration automatique ne peut donc traiter l'arithmétique générale (les théorèmes de Gödel et de Turing en sont une conséquence !). On se limite donc à des théories plus faibles qui pourtant ont déjà une grande utilité.

4 La vérification déductive de programmes

Ainsi, historiquement, l'un des premiers solveurs SMT ne gérait que la théorie de l'égalité et l'arithmétique linéaire. C'était bien suffisant dans le but de prouver qu'un programme ne tente jamais d'effectuer certaines opérations, typiquement un accès à un tableau, avec des paramètres illégaux. La vérification de programmes constitue d'ailleurs le domaine d'application privilégié du SMT, à travers le paradigme de la *vérification déductive* : pour garantir des propriétés d'un programme, convertir celles-ci en des énoncés mathématiques, et les soumettre à un prouveur automatique.

Mieux encore, on peut vouloir montrer qu'un programme est correct, qu'il fait bien ce qu'on lui demande de faire. Pour cela, il faut *spécifier* formellement le comportement attendu. La spécification pourra alors être décomposée (par un algorithme, bien sûr), en suivant la structure du programme, en petites *conditions* dont on espère que le solveur SMT saura montrer leur validité, et dont découle le fait que le programme respecte la spécification. La procédure de décision employée dans le solveur doit alors gérer la théorie dans laquelle est exprimée la spécification.

Parfois, les solveurs SMT échouent à prouver une condition de vérification. Un exemple célèbre est celui d'un algorithme pour générer des nombres premiers dont la preuve de correction repose sur le postulat de Bertrand : il faut d'abord démontrer ce dernier, ce qui est hors de portée des prouveurs automatiques d'aujourd'hui qui ne savent même pas raisonner par récurrence ! Il est alors nécessaire de se rabattre sur des assistants de preuve pour venir à bout de la condition récalcitrante, ou alors simplement de l'admettre comme vraie.

Mais dans la plupart des cas, les conditions de vérification sont simples et ennuyeuses, mais nombreuses. L'automatisation prend alors tout son intérêt.

5 Conclusion

Nous n'avons vu là qu'une méthode spécifique de démonstration automatique (les solveurs SMT) et de vérification de programmes. Il en existe bien d'autres, et plus généralement, la preuve automatique de théorèmes et de programmes cohabite dans le monde des *méthodes formelles* avec bien

d'autres approches. Citons par exemple les *analyses statiques* destinées à déterminer, sans aucune aide humaine, qu'un programme n'a pas de risque de présenter certaines classes de bugs susceptibles par exemple d'interrompre son exécution ; c'est plus faible que de savoir qu'il respecte une spécification donnée, mais c'est plus simple…

Dans un monde où l'informatique prend de plus en plus d'importance, la certification des logiciels est devenue un enjeu crucial, et les problèmes algorithmiques liés à la démonstration sont au cœur des recherches sur le sujet. Il semblerait donc que la preuve automatique ait de beaux jours devant elle, malgré l'indécidabilité et la NP-complétude !

A Les solveurs SAT appliqués à la combinatoire

[Ça serait bien de citer l'article Gabriel Scherer à ce propos, dont je me suis largement inspiré : `http://gallium.inria.fr/blog/large-proofs/`]

Soit (x_n) une suite dont chaque terme vaut 1 ou -1, et C un nombre réel. Il existe alors des entiers d et k tels que

$$|x_d + x_{2d} + \ldots + x_{kd}| > C$$

Il s'agit de la *conjecture de discrépance d'Erdős*, posée par le célèbre mathématicien hongrois Paul Erdős dans les années 1930, dont la démonstration a échappé aux mathématiciens pendant des décennies. En 2014, Boris Konev et Alexei Lisitsa de l'université de Liverpool ont réussi à résoudre le cas particulier de ce problème lorsque $C = 2$, en utilisant… un solveur SAT !

Pour comprendre leur méthode de démonstration, il faut savoir qu'on peut généralement traduire la recherche d'une solution à un problème combinatoire en une formule SAT à satisfaire. Prenons par exemple la résolution d'une grille de Sudoku. On peut définir des variables p_{ijk} signifiant « la case à la i-ème ligne et j-ème colonne est occupée par le nombre k ». On écrit alors des formules pour dire que :

- chaque case (i, j) contient exactement un nombre, c'est-à-dire
  - elle contient au moins un nombre : $p_{ij1} \vee \ldots \vee p_{ij9}$
  - et elle contient au plus un nombre : $\neg p_{ij1} \vee \neg p_{ij2}$, …, $\neg p_{ij1} \vee \neg p_{ij3}$, …, $\neg p_{ij8} \vee \neg p_{ij9}$
- chaque ligne i contient un 1 : $p_{i11} \vee \ldots \vee p_{i91}$
  - de même pour les nombres de 2 à 9
  - de même pour chaque colonne et chaque carré 3×3
- chaque case dont on nous donne la valeur contient cette valeur.

Pour résoudre le Sudoku, il suffit alors de trouver une affectation convenable des p_{ijk} satisfaisant la conjonction de ces formules, et de remplir la case (i, j) avec le nombre k lorsque p_{ijk} est vrai. Si c'est insatisfaisable, on sait par contre que le Sudoku *n'a pas* de solution.

Pour le problème de discrépance d'Erdős, les chercheurs ont d'abord traduit la construction d'une suite *finie* d'une longueur donnée dont toutes les sommes de la forme ci-dessus sont au plus 2. À l'aide d'un solveur SAT, ils ont pu déterminer que pour des suites de taille 1161, le problème était insatisfaisable, autrement dit, que toute suite de longueur 1161 (donc à fortiori, toute suite infinie) contient des sommes dépassant 2. C'est bien ce que l'énoncé d'Erdős affirme !

Cette preuve a bien entendu suscité le même genre de commentaires que celle du *théorème des 4 couleurs* par Appel et Haken en 1976, qui s'appuyait sur un programme pour vérifier des milliers de cas particuliers. Comment faire confiance à une preuve nécessitant d'immenses calculs qu'aucun mathématicien humain ne pourrait vérifier à la main ? Non seulement cela constituait un choc culturel à l'époque, mais il était parfaitement sain de se méfier des bugs qui auraient pu se glisser dans le programme. Aujourd'hui, on peut répondre qu'il est possible de *certifier* le programme qui travaille à notre place avec tout l'arsenal des techniques de vérification logicielle qui existent (vérification déductive, programme prouvé en Coq…). Et les solveurs SAT certifiés, ça existe !

En fin de compte, une preuve « traditionnelle » de la conjecture complète (pour C quelconque) a été annoncée par Terence Tao très récemment, en septembre 2015.

B Un ordinateur peut-il raisonner comme un mathématicien humain ?

Les techniques abordées dans cet article permettent à des ordinateurs de prouver des théorèmes, mais en suivant des procédures de recherche qui n'ont rien à voir avec la façon dont un humain s'y prendrait pour attaquer le problème. Surtout, elles ne produisent pas de preuve lisible, qu'un humain pourrait facilement comprendre par lui-même. Pas étonnant si on garde à l'esprit que leur développement est motivé par des applications où l'essentiel est simplement d'être sûr que des propositions sont vraies, sans avoir forcément plus de justification.

Les méthodes « humaines » pour la preuve automatique n'ont pourtant pas été complètement délaissées par les chercheurs. Woody Bledsoe, pionnier de l'intelligence artificielle, s'y est ainsi intéressé dès les années 70. Après avoir fait de nombreuses contributions techniques en démonstration automatique, il a notamment tenté d'incorporer le raisonnement par analogie en démonstration automatique, en concevant un programme capable de prouver

des théorèmes en adaptant des preuves d'autres résultats. Plus récemment, l'idée de s'inspirer du raisonnement tel que pratiqué par les mathématiciens est également à la base de recherches de Timothy Gowers et Mohan Ganesalingam. Leur objectif : reproduire algorithmiquement les preuves qu'un humain aurait naturellement trouvé pour des problèmes simples, en partant d'une analyse linguistique du langage des mathématiciens, et obtenir ainsi des textes de preuves en langue naturelle indistinguable du contenu d'un livre de maths quelconque. Certains prouveurs plus ordinaires, sans aller jusque là, peuvent déjà fournir des sorties à peu près lisibles : c'est le cas par exemple d'ACL2 (qui compte, parmi ses concepteurs, un élève de Bledsoe).

Le cas de la géométrie euclidienne est instructif quant à la difficulté de produire des démonstrations digestes. On savait déjà, depuis des travaux du logicien Alfred Tarski dans les années 50, que le problème de la décision pouvait être résolu pour la géométrie euclidienne élémentaire (contrairement à l'arithmétique, qui se révèle être bien plus complexe d'un point de vue logique !). La réalisation pratique d'algorithmes pour résoudre des problèmes de géométrie est passée dans un premier temps par l'application d'outils sophistiqués de géométrie algébrique (tels les *bases de Gröbner*, qui permettent notamment de résoudre des systèmes d'équations polynomiales à plusieurs inconnues). Ce n'est que dans les années 90 que la *méthode des aires* et la *méthode des angles pleins* ont été inventées, permettant la génération par ordinateur de preuves courtes et lisibles en géométrie qui parlent de points, droites, longueurs, etc. sans formalisation algébrique. Toutefois, on n'a pas encore atteint le stade où un ordinateur peut trouver un argument astucieux à base de symétrie !