MACHINES DE TURING ET COMPLEXITE ALGORITHMIQUE

Olivier Hudry*

Résumé
L'objectif de cet article consiste à esquisser le rôle des machines de Turing en théorie de la complexité (des algorithmes et des problèmes), dans le domaine de l'optimisation combinatoire. Après avoir rapidement rappelé le contexte historique dans lequel sont nées puis ont évolué les machines de Turing, nous détaillons le fonctionnement de celles-ci, en distinguant entre machines de Turing déterministes et machines de Turing non déterministes. Nous montrons ensuite comment les utiliser pour définir la complexité (en temps de calcul) des algorithmes. Enfin, nous décrivons grâce à elles plusieurs classes fondamentales en théorie de la complexité : la classe *P*, la classe *NP*, la classe des problèmes *NP*-complets et la classe co-*NP*.

Mots-clés
Machines de Turing, théorie de la complexité algorithmique, optimisation combinatoire, calculabilité, classe *P*, classe *NP*, problèmes *NP*-complets, classe co-*NP*.

Abstract
The aim of this paper is to outline the role of the Turing machines in the field of (algorithms and problems) complexity for combinatorial optimization. After a brief recall of the historic context in which the Turing machines are born and have then evolved, we detail their functioning, both for deterministic and nondeterministic Turing machines. Then we show how to use them in order to define the (time) complexity of algorithms. Last, we depict several fundamental classes of the theory of complexity: *P*, *NP*, the class of *NP*-complete problems and co-*NP*.

Keywords
Turing machines, theory of algorithmic complexity, combinatorial optimization, calculability, class *P*, class *NP*, *NP*-complete problems, class co-*NP*.

1. À l'origine des machines de Turing

*1.1. Alan Turing et l'*Entscheidungsproblem

La fin du 19e siècle et le début du 20e furent marqués, en mathématiques, par un vaste et profond mouvement d'interrogations sur les fondements des mathématiques. Au Congrès international des mathématiciens de 1928, à Bologne[1], David Hilbert actualisait sa conférence

* Professeur en mathématiques discrètes, École nationale supérieure des télécommunications - 46, rue Barrault, 75634 Paris Cedex 13.

[1] Voir David Hilbert, « Probleme der Grundlegung der Mathematik », *Mathematische Annalen* 102, 1929, 1-9.

du Congrès international des mathématiciens de 1900, à Paris, au cours duquel il avait formulé ses fameux vingt-trois problèmes qui allaient féconder la recherche mathématique du 20e siècle[2]. À ce congrès de Bologne participait Max Newman, spécialiste de topologie. Les années suivantes, celui-ci se fit l'écho des interrogations mathématiques de l'époque dans le cours de logique mathématique qu'il dispensait au King's College, à Cambridge. Il y exposa en particulier le *problème de la décision* (*Entscheidungsproblem*). Ce problème, considéré par l'école de Bourbaki comme « sans doute le plus ambitieux de tous ceux que se pose la métamathématique »[3], consiste à savoir s'il existe, pour un langage formalisé donné, une méthode permettant d'établir si une relation quelconque de ce formalisme est vraie ou non. Un des étudiants qui suivaient ce cours en 1935 s'intéressa particulièrement à ce problème : il s'agissait d'Alan Mathison Turing[4]. Cette année-là, A. Turing, licence de mathématiques en poche, était devenu en mars *Fellow* du *King's College* grâce à sa démonstration du théorème central limite[5]. L'année suivante, à l'âge de 24 ans, A. Turing résolut le problème de la décision, en montrant que la réponse est négative dès lors que le langage n'est pas réduit à quelques axiomes et quelques signes primitifs[6].

C'est aussi en 1936 qu'il partit pour son premier séjour aux États-Unis, à Princeton, pour y travailler avec Alonzo Church en logique mathématique et John von Neumann en théorie des groupes. En 1938, il déclina l'offre de J. von Neumann de devenir l'assistant de celui-ci et retourna à Cambridge, où il suivit un cours de cryptologie. Pendant la Seconde Guerre mondiale, il mit ces nouvelles connaissances au service de l'effort de guerre pour contribuer, au sein de la *Government Code and Cypher School* (service britannique du chiffre), au décodage des messages de la marine allemande[7]. Il retourna aux États-Unis de novembre 1942 à mars 1943, secrètement. Pendant ce séjour, il rencontra Claude Shannon, père de la théorie de l'information, et s'intéressa au problème du cryptage de la parole. De retour en Angleterre, il poursuivit ses investigations dans cette direction et aboutit à une machine opérationnelle en 1945. Cette recherche amena A. Turing à prêter attention à l'utilisation de l'électronique, ce qui le conduisit, en 1948, à rejoindre l'université de Manchester pour y étudier, avec son ancien professeur Max Newman, la construction d'un ordinateur. Cette étude constitua une de ses préoccupations essentielles, avec l'élaboration d'une théorie de la

---

[2] On pourra les trouver par exemple sur le site Internet http://aleph0.clarku.edu/~djoyce/hilbert/.

[3] Nicolas Bourbaki, *Éléments d'histoire des mathématiques*, Masson, Paris, 1984.

[4] Pour une biographie d'A. Turing, on pourra se reporter aux livres d'Andrew Hodges : *Alan Turing, the Enigma*, Random House, Londres, 1983 (dernière édition, augmentée, chez Walker and Company, New York, 2000 ; traduction française de N. Zimmermann : *Alan Turing ou l'énigme de l'intelligence*, Payot, Paris, 1988), et *Alan Turing; a Natural Philosopher*, Phœnix, Londres, 1997 ; A. Hodges tient aussi à jour un site Internet consacré à A. Turing, à l'adresse http://www.turing.org.uk/turing. On trouvera enfin une présentation de sa vie et de son œuvre dans le livre de Jean Lassègue : *Turing*, Les Belles Lettres, Paris, 1998.

[5] A. Turing ne connaissait pas la démonstration de J. W. Lindeberg, « Eine neue Herleitung des Exponential-gesetzes in der Wahrscheinlichkeitsrechnung », *Mathematische Zeitschrift* 15, 1922, 211-235.

[6] Précisons qu'A. Turing ne connaissait pas les travaux d'Alonzo Church aboutissant à la même conclusion (voir A. Church, « A note on the Entscheidungsproblem », *Journal of Symbolic Logic* I, 1936, 40-41). Les concepts utilisés par les deux mathématiciens sont nettement différents et permettent de ne pas douter de l'originalité des deux démarches. Par ailleurs, on trouvera dans S. Kleene, *Introduction to metamathematics*, Van Nostrand, Princeton, 1952, des systèmes formels pour lesquels de telles méthodes existent. Mais ces systèmes sont trop frustes pour être intéressants.

[7] Son rôle dans la guerre lui valut de devenir *Officer of the Order of the British Empire* en juin 1946. J. Lassègue (*opus* cité) rapporte les propos d'un collaborateur d'A. Turing pendant cette période, D. Michie, selon lequel « sans Turing, l'Angleterre aurait sans doute perdu la guerre » ; il est certain que le rôle d'A. Turing pendant la guerre fut de première importance.

morphogenèse[8]. En juillet 1951, il devint *Fellow* de la *Royal Society*. Mais sa situation fut bouleversée l'année suivante, du fait de son homosexualité. La guerre froide commencée en 1948 eut pour effet, entre autres, d'écarter les homosexuels de certains services officiels ; A. Turing perdit à son tour sa position de consultant à la *Government Code and Cypher School*. Il fut arrêté et traduit en justice pour homosexualité en mars 1952. Il ne se défendit guère pendant son procès, estimant que sa conduite ne présentait rien de répréhensible. Il fut néanmoins condamné : il dut choisir entre la prison ou des traitements psychanalytiques et chimiques ; ainsi, de février à mars 1953, il subit des injections d'hormones femelles destinées à supprimer son homosexualité en modifiant son équilibre chimique, tout en faisant l'objet d'une surveillance policière. Il se suicida le 7 juin 1954, à 42 ans, en mangeant une pomme qu'il avait fait macérer dans du cyanure.

Pendant les vingt ans séparant sa démonstration du théorème central limite et sa mort prématurée, Alan Turing aura contribué activement à la recherche en mathématiques : probabilités, théorie des nombres, théorie des groupes, logique mathématique, cryptologie, etc. On l'a vu, il s'est aussi intéressé à d'autres domaines, comme la biologie, en étudiant les liens entre cette discipline et la logique, et a contribué à la naissance de l'intelligence artificielle. Mais c'est son rôle dans la fondation de l'informatique théorique qui va ici retenir notre attention[9].

1.2. De la calculabilité à la complexité algorithmique

Pour résoudre le problème de la décision, A. Turing a d'abord dû préciser le concept de calculabilité. C'est dans cette optique qu'il a conçu, en 1936[10], ce qu'A. Church a presque aussitôt appelé la « machine de Turing ». Elle confortera la « thèse[11] de Church », laquelle stipule que les fonctions calculables par un procédé effectif coïncident avec les fonctions récursives, en précisant le fonctionnement d'un tel procédé effectif. La machine de Turing reste cependant un modèle théorique (A. Turing la qualifiait de « machine de papier »), destinée au départ à préciser la notion de fonction calculable, mais bientôt appelée à de fructueux prolongements, qui l'entraînent de la théorie de la calculabilité à la théorie des calculs.

Avec le développement de l'informatique, les machines de Turing vont en effet devenir un modèle universel des ordinateurs en même temps qu'elles vont permettre de formaliser le

---

[8] Dans un texte à caractère autobiographique (cité par A. Hodges, 1983, *opus* cité), A. Turing accordait autant d'intérêt à son article «The chemical basis of morphogenesis », *Philosophical Transaction of the Royal Society* B 237, 1952, 37-72, qu'à celui consacré au problème de la décision. Selon A. Hodges, A. Turing papillonnait autour de divers sujets au début de son séjour à Manchester, alternant entre d'anciens sujets d'étude et de nouveaux domaines de recherche.

[9] Ses travaux ont fait l'objet de rééditions sous forme de plusieurs volumes : *Collected Works of A.M. Turing*, Elsevier Science Publishers, Amsterdam, 1992.

[10] A. Turing, « On Computable Numbers with an Application to the Entscheidungsproblem », Proceedings of the London Mathematical Society 42, 1936, 230-265. Une traduction en français par Julien Basch est disponible dans A. Turing et J.-Y. Girard : *La machine de Turing*, Éditions du Seuil, 1995 ; ce livre contient un autre article d'A. Turing, traduit par Patrice Blanchard et consacré à l'intelligence artificielle : « Computing Machinery and Intelligence », *Mind* LIX, 1950, 433-460.

[11] « Thèse » est ici pris au sens de conjecture. Entrer dans les détails de la théorie de la calculabilité n'est pas l'objectif de cet article. Le lecteur intéressé pourra consulter le livre de Nigel J. Cutland : *Computability (An Introduction to Recursive Function Theory)*, Cambridge University Press, 1986 ou, pour un texte plus accessible, celui de Pierre Wolper : *Introduction à la calculabilité*, InterÉditions, Paris, 1991 (seconde édition chez Dunod, 2001).

concept d'« algorithme »[12], en définissant un algorithme comme la spécification d'une machine de Turing. Dans les années 1960, les algorithmiciens ont éprouvé le besoin de caractériser l'efficacité des algorithmes et la difficulté intrinsèque des problèmes, en termes de ressources (typiquement, temps de calcul ou place mémoire) consommées par un ordinateur pour traiter un problème donné. Grossièrement énoncé, l'enjeu consiste à établir une distinction entre les problèmes que l'on saura résoudre sans consommer des ressources en quantité « déraisonnable » et ceux pour lesquels il faudra renoncer à une résolution exacte, faute de temps ou de place suffisants. En anticipant sur la suite, précisons dès maintenant qu'une limite généralement acceptée entre « raisonnable » et « déraisonnable » correspond d'un côté à une croissance au plus polynomiale des ressources avec la taille (qu'il conviendra donc de définir soigneusement) des données du problème à traiter, contre une croissance exponentielle de l'autre côté (voir la figure 4 pour une illustration de telles croissances).

Là encore, les machines de Turing ont fourni un modèle pour définir la *complexité des algorithmes*, qui mesure, pour la complexité en temps (respectivement en place), le nombre d'instructions élémentaires effectuées (respectivement la place utilisée en mémoire) par l'algorithme en question en fonction de la taille des données à traiter. À partir de la complexité des algorithmes, on pourra définir à son tour la *complexité des problèmes*, pour essayer de distinguer entre problèmes « faciles » et problèmes « difficiles »[13]. En anticipant une nouvelle fois et en schématisant, disons que ceux-là correspondront aux problèmes que l'on peut résoudre à l'aide d'un algorithme de complexité au plus polynomiale, tandis que l'on ne saura, dans l'état actuel de nos connaissances, résoudre ceux-ci qu'à l'aide d'algorithmes de complexité exponentielle. On voit que, pour la classification des problèmes[14], le degré des polynômes impliqués n'intervient pas, et la lecture de cet article sera facilitée si le lecteur se rappelle que, de manière intuitive, la classification des problèmes se fait « à des polynômes près »[15]...

En dépit de nombreux travaux, y compris d'éminents chercheurs, les résultats obtenus en théorie de la complexité depuis le théorème fondateur de S.A. Cook en 1971 (voir plus loin le théorème 7) n'ont pas encore permis de répondre à certaines questions fondamentales, qui seront mentionnées plus loin. Elles constituent actuellement des problèmes ouverts majeurs dans le domaine de la complexité.

L'objectif de cet article consiste à esquisser le rôle des machines de Turing en théorie de la complexité (des algorithmes et surtout des problèmes), dans le domaine de l'*optimisation*

---

[12] Du nom du mathématicien arabe Muhammad Al-Khwarizmi (fin du 8^{e} siècle – début du 9^{e} siècle), dont les travaux ont fortement influencé la science occidentale du Moyen Âge (voir par exemple Amy Dahan-Dalmedico et Jeanne Peiffer, *Une histoire des mathématiques. Routes et dédales*, Éditions du Seuil, Paris, 1986). Un algorithme est une méthode de résolution décrite sous la forme d'une suite finie d'instructions qui, exécutées pas à pas, permettent de traiter, en un temps fini, des données quelconques du problème pour lequel l'algorithme est conçu.

[13] Il convient de bien distinguer entre la complexité d'un algorithme, dont l'objectif est de mesurer l'efficacité de l'algorithme considéré, conçu pour un problème donné, et la complexité d'un problème, qui va essayer de caractériser la difficulté intrinsèque du problème considéré, indépendamment du choix de l'algorithme retenu pour le résoudre.

[14] Ceci n'étant pas valable pour la classification des algorithmes selon leurs efficacités...

[15] Au sens de la composition, l'addition ou la multiplication des fonctions. Par exemple, on ne distinguera pas les fonctions n, n^2 ou n^3 entre elles, ni les fonctions 2^n, 3^n ou $n2^{n^2}$ entre elles ; en revanche, on distinguera les trois premières fonctions des trois dernières, puisqu'on n'obtient pas les unes à partir des autres par composition, addition ou multiplication avec un ou plusieurs polynômes.

combinatoire. Les considérations qui suivent ne concernent que la complexité en temps de calcul[16]. Nous y distinguerons deux types de machines de Turing :
• les « machines de Turing déterministes » ;
• les « machines de Turing non déterministes ».

Nous commencerons par décrire le fonctionnement des machines de Turing. Nous montrerons ensuite comment utiliser celles-ci pour définir la complexité des algorithmes et construire quatre classes fondamentales contenant des problèmes d'un certain type (« problèmes de décision ») : la classe *P*, la classe *NP*, la classe des « problèmes *NP*-complets » et la classe co-*NP*. En guise de conclusion, nous évoquerons des prolongements possibles de ce qui aura été présenté ici.

2. Machines de Turing

2.1. Description d'une machine de Turing

Les machines de Turing (MT dans la suite) ne sont pas des machines matérielles destinées à être construites et à être utilisées en pratique. Elles fournissent un modèle théorique pour les ordinateurs et les algorithmes. Elles sont caractérisées par un dispositif que l'on serait tenté de considérer comme physique si elles devaient être réalisées, dispositif commun aux machines de Turing déterministes et aux machines de Turing non déterministes, et par une *fonction de transition*, qui dépend de l'algorithme à représenter. Les MT évoluent par étapes successives, appelées « pas ».

Le dispositif « physique » est constitué des éléments suivants (voir la figure 1 pour une illustration) :
• une sorte d' « unité centrale » qui contrôle l'ensemble du déroulement du programme et dont le contenu (la fonction de transition) dépend de l'algorithme à exécuter ;
• une sorte de place-mémoire, représentée par une bande infiniment longue, découpée en cases numérotées par les entiers relatifs ; ces cases contiennent initialement les données à traiter ; par la suite, y seront inscrits les calculs intermédiaires et les résultats finals ;
• une tête de lecture-écriture permettant de lire et éventuellement de modifier le contenu des cases de la bande conformément aux instructions fournies par l' « unité centrale ».

[16] Nous n'aborderons donc pas la *complexité en place*, de même que nous passerons sous silence d'autres aspects de la théorie de la complexité. Le lecteur désireux d'aller plus loin dans ce domaine pourra se reporter au livre de référence de Michael R. Garey et David S. Johnson, *Computers and Intractability. A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, New York, 1979. On pourra le compléter par le livre plus récent de S. Arora et B. Barak, *Complexity Theory: A Modern Approach*, Cambridge University Press, Cambridge, 2009, et, en français, par le livre de Jean-Pierre Barthélemy, Gérard Cohen et Antoine Lobstein : *Complexité algorithmique et problèmes de communications*, Masson, Paris, 1992, parmi d'autres ouvrages consacrés au même sujet.

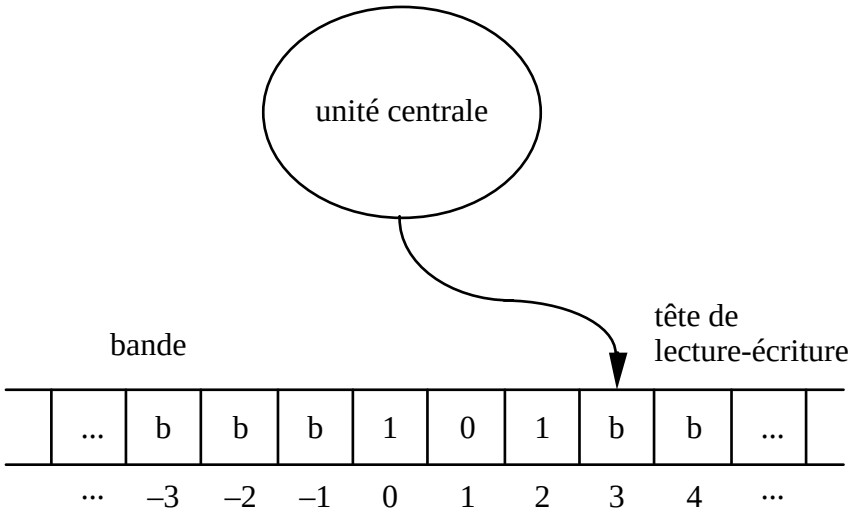

Figure 1 : représentation schématique d'une machine de Turing.

La partie de la MT associée à l'algorithme (et précisant le contenu de l' « unité centrale ») nécessite la définition de certains éléments supplémentaires :

• un ensemble fini *S* de symboles spécifiant les caractères susceptibles de figurer dans les cases de la bande, dont un symbole particulier, souvent noté b (pour « blanc »), jouant le rôle d'une espace (entre des données, des calculs intermédiaires ou des résultats) ;

• un ensemble fini *E* d'états ;

• une fonction de transition *t* permettant de définir ce que la MT doit faire à chaque étape, selon d'une part son état courant et d'autre part le symbole lu dans la case *c* sur laquelle pointe la tête de lecture-écriture ; plus précisément, elle doit permettre de choisir, en fonction de ces deux éléments seulement, le nouvel état dans lequel doit se mettre la MT, le nouveau symbole à écrire dans *c* et le déplacement de la tête le long de la bande (une conception un peu plus réaliste d'une MT laisserait plutôt cette tête fixe et ferait dérouler la bande devant la tête ; mais là n'est pas la question…).

On verra plus loin que le choix de *S*, s'il reste « raisonnable », n'est pas primordial. Pour fixer ses idées, le lecteur peut provisoirement penser à un choix classique : $S = \{0, 1, b\}$, les données, les calculs intermédiaires et les résultats étant donc codés en binaire[17] (c'est ce choix qui a été retenu pour l'illustration de la figure 1). La spécification d'une MT se fait donc principalement en précisant l'ensemble *E* des états et la fonction de transition *t*. Un « pas » d'une MT apparaît alors en fait comme la détermination de l'image par *t* du couple (*e*, *s*), où *e* désigne l'état courant de la MT et *s* le symbole contenu dans la case sur laquelle pointe la tête de lecture-écriture. C'est d'ailleurs la nature de *t* qui permet de distinguer entre machine de Turing déterministe (MTD dans la suite) et machine de Turing non déterministe (MTND dans la suite).

Il existe quelques variantes minimes sur les principes régissant le fonctionnement d'une MT. On peut sans perte de généralité adopter les conventions suivantes :

[17] Un codage un peu plus sophistiqué permettrait de coder l'espace b à l'aide de 0 et de 1, sans qu'il y ait de confusion possible avec les bits 0 et 1 utilisés pour les données, les calculs et les résultats (voir J.-P. Barthélemy *et alii*, *opus* cité). Bien qu'il soit donc théoriquement possible de se passer de ce symbole, il est plus commode de le conserver.

• les données sont initialement écrites sur les cases d'indices positifs, en commençant à la case de numéro 0 de la bande ;
• la tête de lecture-écriture pointe initialement sur la case de numéro 0 ;
• à chaque pas de la MT, la tête de lecture peut se déplacer au plus d'une case vers la droite ou vers la gauche : on notera −1 (respectivement 1) un déplacement d'une case vers la gauche (respectivement droite) tandis que 0 représentera l'absence de déplacement.

2.2. Machines de Turing déterministes

Comme on vient de le dire, ce qui distingue une MTD d'une MTND est la nature de la fonction de transition *t*. Pour une MTD donnée, *t* est une fonction définie de $E \times S$ dans $E \times S \times \{-1, 0, 1\}$. Autrement dit, à chaque couple (e, s), où *e* désigne l'état courant et *s* le symbole de la case courante, *t* associe au plus un triplet (e', s', d), où e' désigne l'état de la MTD après le pas, s' le symbole que la tête de lecture-écriture a substitué à *s* dans la case courante, et *d* le déplacement effectué par cette tête après avoir écrit s'. On notera que *t* n'est pas nécessairement une application : certains couples (e, s) peuvent ne pas avoir d'image. Dans ce cas, quand la MTD rencontre un tel couple, la MTD ne fait rien et s'arrête. On pourra donc considérer un tel couple comme une instruction d'arrêt de l'algorithme associé à la MTD.

La figure 2 donne un exemple de fonction de transition et illustre ainsi la modélisation d'un algorithme par une MTD. Dans cet exemple, *E* contient six états, dont un état initial e_1 (état dans lequel est supposée se trouver la MTD au début de l'exécution de l'algorithme) et deux états finals (au sens où un calcul normal est censé se terminer en l'un de ces états, et pas dans un autre) e_5 et e_6.

t	b	0	1
e_1	(e_2, b, –1)	(e_1, 0, 1)	(e_1, 1, 1)
e_2		(e_4, b, –1)	(e_3, b, –1)
e_3	(e_5, b, 1)	(e_3, b, –1)	(e_3, b, –1)
e_4	(e_6, b, 1)	(e_4, b, –1)	(e_4, b, –1)
e_5	(e_5, 1, 0)		
e_6	(e_6, 0, 0)		

Figure 2 : un exemple de fonction de transition pour une machine de Turing déterministe.

Imaginons qu'on applique cette MTD à la donnée de la figure 1, donnée que l'on peut interpréter comme l'entier 5 écrit en binaire. Plus précisément, la chaîne binaire représentant l'entier 5 est placée sur la bande, dans les cases d'indices 0, 1 et 2. Les autres cases de la bande contiennent initialement des espaces b. La machine est initialement en l'état e_1, et la tête de lecture-écriture est positionnée sur la case de numéro 0. Le calcul se déroule alors étape par étape, comme indiqué par la figure 3. On y a précisé, à gauche, l'état de la MTD au début de chaque pas et, à droite, la partie utile de la bande (sans rappeler le numéro des cases), la case sur laquelle pointe la tête de lecture-écriture étant soulignée.

e_1 ... b <u>1</u> 0 1 b ...
e_1 ... b 1 <u>0</u> 1 b ...
e_1 ... b 1 0 <u>1</u> b ...
e_1 ... b 1 0 1 <u>b</u> ...
e_2 ... b 1 0 <u>1</u> b ...
e_3 ... b 1 <u>0</u> b b ...
e_3 ... b <u>1</u> b b b ...
e_3 ... <u>b</u> b b b b ...
e_5 ... b <u>b</u> b b b ...
e_5 ... b <u>1</u> b b b ...

Figure 3 : un calcul effectué par la machine de Turing de la figure 2.

Dans notre exemple, la MTD s'arrête donc dans l'état e_5 et la bande ne contient qu'un « 1 » entouré de blancs b. Il est facile d'interpréter cette MTD. En examinant attentivement les instructions associées à chaque état, le lecteur se convaincra aisément que le rôle de chaque état est le suivant. L'état e_1, qui sert d'état initial, consiste en fait à se déplacer vers la droite à la recherche d'un blanc, et donc de la fin de la chaîne binaire représentant l'entier *n* fourni comme donnée, sans rien changer. Quand on a trouvé la première espace, on change d'état et on passe à e_2 en revenant en arrière d'une case : on pointe désormais sur le dernier bit de *n*, que l'on efface ; si celui-ci est un « 1 » (respectivement « 0 »), on passe dans l'état e_3 (respectivement e_4) : l'état e_2 correspond donc à une structure d'aiguillage sensible au dernier bit de *n* (on constatera que l'état e_2 ne définit pas d'action au cas où un blanc est lu : ce cas ne peut en fait pas se produire si on a effectivement fourni un entier en entrée). Le rôle des états e_3 et e_4 est d'effacer *n* (ce que e_2 a commencé à faire) tout en mémorisant, par le choix antérieur de e_3 ou de e_4, la valeur du bit de droite de *n* qui a commandé l'aiguillage au début. Les derniers états e_5 et e_6 se contentent de récrire le dernier bit de *n* et arrêtent le calcul. Finalement, on constate que cette machine modélise un algorithme qui calcule le reste d'un entier modulo 2, ou encore qui permet de reconnaître la parité d'un entier.

On voit sur cet exemple que la modélisation d'un algorithme par une MTD constitue en général un exercice assez rébarbatif et fastidieux. On peut même d'abord douter, comme M. Newman quand A. Turing lui présenta ses travaux, qu'une machine d'apparence aussi fruste puisse receler un intérêt quelconque. Et pourtant, ainsi qu'on l'a souligné plus haut, les MTD fournissent un modèle universel pour les ordinateurs et les algorithmes. En fait, elles permettent de calculer toutes les fonctions récursives et la « thèse de Church » (de ce fait présentée parfois comme la « thèse de Church-Turing ») peut se reformuler, en termes non techniques, de la façon suivante : tout ce qui est calculable l'est par une machine de Turing.

Les MTD vont aussi nous permettre de définir la complexité d'un algorithme.

2.3. Complexité d'un algorithme

Un algorithme *A* étant assimilable à une MTD, on peut chercher à caractériser l'efficacité de *A* à l'aide du nombre de pas effectués par la MTD associée ; on obtiendra alors une mesure de la *complexité* (en temps) de *A* : un algorithme sera d'autant plus efficace, ou d'autant plus faible complexité, qu'il effectuera moins de pas pour traiter des données pour lesquelles *A* a été conçu. En supposant que tous les pas d'une MTD demandent le même temps pour être

effectués, la complexité de A permettra de mesurer le temps de calcul nécessaire pour traiter les données en question. Une difficulté évidente apparaît alors : le nombre de pas ne sera sûrement pas indépendant des données à traiter, alors qu'on aimerait obtenir une mesure qui soit le plus intrinsèque possible.

On adopte alors le compromis suivant. L'ensemble S des symboles étant fixé, on appelle « taille des données D » et on note $|D|$ le nombre de symboles nécessaires pour coder D à l'aide de S. Soit D un jeu de données, et soit $\phi_A(D)$ le nombre de pas effectués par A quand on résout D à l'aide de A. Pour définir la complexité de A, on regroupe toutes les données de même taille n et on considère le nombre maximum de pas effectués par A pour résoudre un tel jeu de données :

Définition 1. La complexité c_A de A est la fonction définie sur l'ensemble des entiers naturels n par $c_A(n) = \max\{\phi_A(D)$ pour D vérifiant $|D| = n\}$.

À cause du maximum dans l'expression de c_A, cette complexité s'appelle aussi *complexité dans le pire des cas*. On pourrait définir la *complexité dans le meilleur des cas* (respectivement *en moyenne*) en remplaçant le maximum par un minimum (respectivement une moyenne). Mais la complexité dans le meilleur des cas est souvent peu porteuse d'informations utiles et la complexité en moyenne, pourtant pertinente pour caractériser le comportement d'un algorithme, est souvent trop difficile à calculer (il peut même déjà être difficile d'avoir une idée appropriée de la loi de distribution des données…). Pour ces raisons, on se contente généralement de la complexité dans le pire des cas, appelée plus simplement *complexité* quand il n'y a pas de risque de confusion. En fonction de la nature de leurs complexités, on distingue deux types d'algorithmes :

Définition 2. Un algorithme est dit *polynomial* si sa complexité est majorable par une fonction polynomiale en la taille des données. Dans le cas contraire, il est dit *exponentiel*.

Une MTD étant assimilable à un algorithme, on parlera aussi de MTD polynomiale si elle représente un algorithme polynomial. Plus généralement, une MT (déterministe ou non) sera dite polynomiale si, pour traiter des données quelconques D du problème pour lequel elle est conçue, elle effectue un nombre de pas majoré par un polynôme en $|D|$. On dit alors qu'*elle résout ce problème en un temps polynomial* ou *polynomialement* et ce problème est lui-même dit *polynomial*.

On constate que cette terminologie est abusive d'un point de vue mathématique : un algorithme A de complexité $c_A(n) = \ln(n)$ sera dit polynomial bien que sa complexité ne soit pas un polynôme en n ; de même un algorithme A de complexité $c_A(n) = n!$ sera dit exponentiel bien que sa complexité ne soit pas une exponentielle de n.

Ce partitionnement est illustré par la figure 4[18]. Celle-ci indique le temps de calcul nécessaire pour traiter des données de taille n comprise entre 10 et 50 à l'aide d'un algorithme polynomial ou exponentiel selon les lignes, exécuté par une machine de Turing que l'on suppose capable d'effectuer mille pas en une seconde. Elle montre que la croissance exponentielle rend rapidement prohibitive l'utilisation d'un algorithme exponentiel si on doit traiter des données faisant partie des plus difficiles. Elle laisse aussi deviner que des progrès

[18] Adaptée, avec des compléments, du livre de M.R. Garey et D.S. Johnson, *opus* cité.

technologiques qui permettraient de rendre plus rapides nos ordinateurs actuels ne changeraient pas qualitativement la conclusion de la phrase précédente.

n	10	20	30	40	50
$\log_{10}(n)$	0,001 s	0,0013 s	0,0015 s	0,0016 s	0,0017 s
n	0,01 s	0,02 s	0,03 s	0,04 s	0,05 s
n^2	0,1 s	0,4 s	0,9 s	1,6 s	2,5 s
n^3	1 s	8 s	27 s	64 s	125 s
n^5	1,7 mn	53,3 mn	6,75 h	28,3 h	3,6 jours
2^n	1 s	17,5 mn	12,4 jours	34,9 ans	357 siècles
10^n	116 jours	3.10^7 siècles	3.10^{17} siècles	3.10^{27} siècles	3.10^{37} siècles
n !	1 h	$7{,}7.10^5$ siècles	$8{,}4.10^{19}$ siècles	$2{,}6.10^{35}$ siècles	$9{,}6.10^{51}$ siècles
n^n	116 jours	$3{,}3.10^{13}$ siècles	$6{,}5.10^{31}$ siècles	$3{,}8.10^{51}$ siècles	$2{,}8.10^{72}$ siècles

Figure 4. Croissance du temps de calcul pour diverses complexités.

En pratique, on ne passe pas par une MT pour calculer la complexité d'un algorithme (ce qui risquerait d'être assez fastidieux...). On utilise des *opérations élémentaires*, c'est-à-dire des opérations qui peuvent être modélisées par des MT de sorte que l'application d'une de ces opérations élémentaires ne nécessite qu'un nombre au plus polynomial de pas d'une MT. C'est le cas des opérations usuelles, notamment les opérations arithmétiques, les affectations de valeurs, les comparaisons, etc., quand elles sont appliquées par exemple à des entiers. Puisque, comme on l'a annoncé plus haut, la classification des problèmes que l'on cherche à établir est définie « à des polynômes près », adopter un point de vue plus « macroscopique » en considérant ces opérations élémentaires à la place des pas des MT ne perturbe pas la classification cherchée et s'avère bien plus pratique (on en verra un exemple plus bas).

2.4. Codage des données

La distinction résultant de la définition 2, de prime abord grossière, a cependant un gros avantage : elle permet un partitionnement des algorithmes indépendant du codage adopté, dès lors qu'on renonce au codage unaire (qui ne manipule qu'un seul symbole en plus du blanc). En effet, imaginons qu'on utilise un ensemble S possédant σ (avec $\sigma > 2$) symboles autres que le blanc au lieu de l'ensemble {0, 1, b} considéré plus haut et supposons, pour simplifier, que les données D se réduisent à un entier n. La taille binaire $|D|_2$ de D serait de l'ordre de $\log_2(n)$ et sa taille $|D|_\sigma$ quand n est codé à l'aide de S de l'ordre de $\log_\sigma(n)$, d'où la relation $|D|_\sigma \approx |D|_2 \times \log_\sigma 2$. À cause de ce lien linéaire entre les tailles, la complexité d'un algorithme ne change pas qualitativement : par exemple, une complexité est polynomiale pour S si et seulement si elle l'est pour le codage binaire. Cette relation n'est pas vraie seulement pour des données réduites à un ou plusieurs entiers, mais est générale[19].

[19] En fait, le gain apporté par un codage plus riche que le codage binaire n'est pas toujours exploité. C'est par exemple le cas quand on code un vecteur booléen v à q composantes : qu'on dispose de deux symboles utiles, assimilables à « vrai » et « faux », ou de plus de deux symboles utiles ne change rien à la taille d'un codage « naturel » de v (celui qui décrit v comme une suite de q booléens) ; dans les deux cas, la taille sera égale à q.

En revanche, cette conservation de la nature de l'algorithme n'est plus nécessairement vraie si on considère le codage unaire, celui qui représente un entier n sous la forme de n bâtons (S contient un seul symbole utile, le bâton, en plus du séparateur b). Il existe alors un saut exponentiel entre les tailles de codage, et un algorithme exponentiel pour le codage binaire (ou, de manière équivalente comme on vient de le voir, pour un codage possédant plus de symboles) peut devenir polynomial pour le codage unaire.

À cause de ce saut exponentiel entre les tailles des codages unaire et binaire (ou autre), on n'adopte généralement pas le codage unaire, trop consommateur de symboles[20]. En revanche, le codage binaire conduisant aux mêmes résultats qualitatifs que les autres (sauf, donc, le codage unaire), il n'est pas gênant de supposer qu'il s'agit du codage adopté. On fera cette hypothèse pour les illustrations qui suivent.

2.4. Machines de Turing non déterministes

La seule différence entre une MTD et une MTND porte sur la fonction de transition. Alors que, pour une MTD, la fonction de transition t associe un seul triplet $(e', s', d) \in E \times S \times \{-1, 0, 1\}$ à un couple (e, s) où e désigne l'état courant de la MTD et s le symbole lu dans la case courante de la bande, t peut en associer plusieurs pour une MTND. Plus formellement, la fonction de transition d'une MTND est donc définie de $E \times S$ dans l'ensemble $P(E \times S \times \{-1, 0, 1\})$ des parties de $E \times S \times \{-1, 0, 1\}$. C'est là que réside le caractère aléatoire de la MTND : à chaque pas du fonctionnement de la MTND, on choisit au hasard le triplet définissant l'instruction que l'on va exécuter parmi les possibilités associées à (e, s).

En dépit de la contradiction apparente des termes, une machine de Turing déterministe est donc un cas particulier de machine de Turing non déterministe : c'est le cas pour lequel l'ensemble des triplets possibles est de cardinal 1 quel que soit le couple (e, s).

La figure 5 donne un exemple de MTND.

t	b	0	1
e_1	$(e_2, \text{b}, -1)$	$(e_1, 0, 1)$	$(e_1, 1, 1)$
e_2		$(e_2, \text{b}, -1)$ $(e_3, 0, -1)$ $(e_3, 1, -1)$	$(e_2, \text{b}, -1)$ $(e_3, 0, -1)$ $(e_3, 1, -1)$
e_3		$(e_3, 0, -1)$ $(e_3, 1, -1)$	$(e_3, 0, -1)$ $(e_3, 1, -1)$

Figure 5 : un exemple de fonction de transition pour une machine de Turing non déterministe.

Comme pour la MTD donnée plus haut, on peut se demander ce que fait la MTND de la figure 5. Il est facile de voir que l'état e_1 permet de se positionner sur le bit le plus à droite de la donnée (une chaîne binaire). On passe alors à l'état e_2, état dans lequel on persiste tant qu'on écrit des blancs à la place des caractères lus, en se déplaçant continuellement à gauche.

[20] Il existe cependant une exception notoire : on considère au contraire le codage unaire pour définir les *problèmes pseudo-polynomiaux* et les *problèmes fortement NP-complets* (il en sera rapidement question dans la conclusion, sans détails). Une définition précise de ces problèmes sort du cadre de cette présentation ; le lecteur intéressé pourra approfondir la question dans les ouvrages cités en référence au début.

Dès qu'on écrit un caractère autre que le blanc, on passe dans l'état suivant e_3. Dans cet état, on remplace les derniers caractères initiaux par des 0 ou des 1 aléatoirement, jusqu'à trouver le premier blanc bordant la donnée à gauche, moment où la machine s'arrête. On voit donc que, globalement, la MTND de la figure 5 commence par effacer la partie droite de la donnée et remplace la partie gauche par une chaîne binaire aléatoire (éventuellement vide). Le passage de l'état e_2 à l'état e_3 étant lui aussi aléatoire, la MTND de la figure 5 permet donc d'engendrer aléatoirement toute chaîne binaire de longueur inférieure ou égale à la longueur de la donnée.

3. Classes de problèmes

Nous allons maintenant nous intéresser à l'utilisation que l'on peut faire des MT pour définir des classes de problèmes selon leurs difficultés intrinsèques, ce que l'on appelle de nouveau *complexité* (des problèmes et non plus des algorithmes). Pour illustrer les concepts qui suivent, nous considérerons jusqu'à la fin de ce texte un problème d'optimisation combinatoire particulier, appelé le *problème du sac à dos (binaire)* et le *problème de décision*[21] qui lui est associé. Rappelons d'abord ce qu'est un problème d'optimisation combinatoire. La définition qui suit introduit une contrainte (positivité des valeurs prises par *f*) qui n'est pas indispensable mais qui simplifiera les considérations qui suivent[22].

Définition 3. Un problème d'optimisation combinatoire est un problème qui peut se mettre sous la forme : $\max_{x \in X} f(x)$, où X est un ensemble fini et où f est une fonction à valeurs entières positives ou nulles.

La finitude de X assure l'existence d'une solution optimale. Notons au passage que, le minimum d'une fonction f étant égal à l'opposé du maximum de $-f$, la définition 3 s'étend aussi à un problème de minimisation. Le problème du sac à dos décrit ci-dessous, que nous appellerons SD-O, est un problème d'optimisation combinatoire.

Définition 4. Le problème du sac à dos (binaire) SD-O est un problème de la forme : $\max \sum_{i=1}^{n} u_i x_i$, avec les contraintes $\sum_{i=1}^{n} p_i x_i \leq \pi$ et, pour $1 \leq i \leq n$, $x_i \in \{0, 1\}$, et où n, π, les u_i et les p_i ($1 \leq i \leq n$) sont des entiers strictement positifs donnés.

Le problème du sac à dos tire son nom d'une expérience que rencontre un randonneur. Celui-ci dispose de n objets qu'il envisage d'emporter dans son sac à dos. Chaque objet i possède une utilité u_i et un poids p_i. On suppose que les utilités sont additives (il n'y a donc pas d'objets redondants les uns par rapport aux autres) et que le sac à dos ne peut contenir des

[21] Il ne s'agit pas ici de l'*Entscheidungsproblem*... Les problèmes de décision sont définis un peu plus loin.

[22] Ici aussi, on rencontre des variantes sur la définition d'un problème d'optimisation combinatoire : certains ouvrages considèrent que X peut être infini dénombrable, ou que f doit être une forme linéaire, ou encore que les valeurs prises par f peuvent être réelles et non nécessairement entières (remarquons qu'il n'y a pas de différence, du moins d'un point de vue théorique, entre l'hypothèse de valeurs entières et celle de valeurs rationnelles si l'on suppose X fini).

objets de poids total strictement supérieur à π. On suppose enfin que l'objectif du randonneur est de sélectionner les objets qu'il emporte de manière à maximiser l'utilité des objets emportés sans dépasser le poids autorisé π. Il suffit d'associer une variable binaire x_i à chaque objet *i* en posant $x_i = 1$ si l'objet *i* est emporté et $x_i = 0$ sinon pour obtenir le problème formulé plus haut.

3.1. Problèmes d'optimisation et problèmes de décision

Bien qu'on s'intéresse souvent à des problèmes d'optimisation, la théorie de la complexité va d'abord considérer des *problèmes de décision* (parfois appelés aussi *problèmes de reconnaissance*).

Définition 5. Un problème de décision est un problème dans lequel on pose une question admettant la réponse « oui » ou « non ».

On notera que l'on exclut des réponses possibles des réponses comme « peut-être », « je ne sais pas », etc., et que l'on suppose que l'on n'a affaire qu'à des problèmes décidables. On supposera dans la suite qu'on ne s'intéresse qu'à des problèmes non triviaux, c'est-à-dire possédant des données admettant la réponse « oui » et d'autres données admettant la réponse « non ». Un exemple de problème de décision est celui de la primalité d'un entier : étant donné un entier *n*, *n* est-il premier ? Au contraire, un problème d'optimisation, dont la réponse est la valeur numérique du maximum cherché, n'est pas un problème de décision.

On peut cependant associer un problème de décision à un problème d'optimisation combinatoire, de façon canonique. Considérons pour cela un tel problème, que l'on présentera sous la forme standardisée suivante :

Nom : POC (problème d'optimisation combinatoire) ;
Données : un ensemble fini *X*, une fonction *f* à valeurs entières définie sur *X* ;
Objectif : déterminer la valeur maximum de *f* sur *X*.

On associe à POC le problème de décision suivant, là aussi présenté de manière standardisée.

Nom : PD (problème de décision associé à POC) ;
Données : un ensemble fini *X*, une fonction *f* à valeurs entières définie sur *X*, un entier *K* ;
Question : existe-t-il un élément *x* de *X* vérifiant $f(x) \geq K$?

Il est alors intéressant d'étudier les liens entre POC et PD du point de leurs complexités. Il est clair que tout algorithme résolvant POC pourra être utilisé pour résoudre PD sans changement significatif de sa complexité : pour résoudre un jeu de données de PD, il suffit en effet d' « oublier » provisoirement *K*, ce qui donne un jeu de données de POC, de résoudre ce jeu de données pour obtenir le maximum de *f* sur *X*, enfin de comparer ce maximum à *K* : si le maximum est supérieur ou égal à *K*, la réponse pour PD est « oui », elle est « non » sinon. Par conséquent, tout algorithme résolvant POC peut donner naissance à un algorithme permettant de résoudre PD avec la même complexité, à une comparaison près, ce qui peut être négligé.

En particulier, si POC peut être résolu à l'aide d'un algorithme polynomial, il en est de même de PD.

Qu'en est-il de la réciproque ? Sous réserve de quelques hypothèses suffisamment raisonnables pour être très souvent vérifiées, la réciproque est vraie. Plus précisément, on peut souvent, à partir d'un algorithme résolvant PD, construire un algorithme de même complexité « à des polynômes près » pour résoudre POC. On voit donc l'intérêt d'étudier les problèmes de décision plutôt que les problèmes d'optimisation combinatoire associés : les premiers permettent d'obtenir généralement des conclusions qualitatives concernant les seconds, tout en étant plus « faciles » à aborder, « faciles » n'étant pas à prendre ici au sens de la complexité, mais seulement dans la mesure où les problèmes de décision admettent moins de réponses possibles que les problèmes d'optimisation.

Plutôt que de décrire de manière générale comment résoudre POC en supposant connu un algorithme résolvant PD, nous allons nous contenter d'illustrer ce mécanisme sur l'exemple du problème du sac à dos. Le problème POC est alors le problème SD-O précisé par la définition 4, et le problème de décision associé est le suivant :

Nom : SD-D (problème de décision associé au problème du sac à dos) ;
Données : un entier n, n entiers u_i ($1 \leq i \leq n$), n entiers p_i ($1 \leq i \leq n$), un entier π, un entier K, tous ces entiers étant strictement positifs ;
Question : existe-t-il $x = (x_i)_{1 \leq i \leq n} \in \{0, 1\}^n$ vérifiant $\sum_{i=1}^{n} u_i x_i \geq K$ et $\sum_{i=1}^{n} p_i x_i \leq \pi$?

Soit A un algorithme de complexité c_A permettant de résoudre SD-D. La solution $x = 0$ respectant toutes les contraintes de SD-O, le maximum M cherché dans SD-O est positif ou nul. La binarité de x permet d'autre part de majorer trivialement M par $K_{\max} = \sum_{i=1}^{n} u_i$. On résout alors SD-D à l'aide de A avec $K = K_{\max}$. Si la réponse est « oui », on conclut : $M = K_{\max}$; sinon on applique A à SD-D avec $K = \left\lfloor \frac{K_{\max}}{2} \right\rfloor$. Si la réponse est « oui », on obtient $\left\lfloor \frac{K_{\max}}{2} \right\rfloor \leq M < K_{\max}$ et on recommence avec $K = \left\lfloor \frac{3K_{\max}}{4} \right\rfloor$; sinon, on obtient $0 \leq M < \left\lfloor \frac{K_{\max}}{2} \right\rfloor$ et on recommence avec $K = \left\lfloor \frac{K_{\max}}{4} \right\rfloor$. On recommence ainsi en réduisant à chaque fois de moitié l'intervalle contenant M. Le caractère entier de M permet alors de prévoir que ce procédé dichotomique ne sera pas appliqué plus qu'environ[23] $\log_2 K_{\max} = \log_2 \left(\sum_{i=1}^{n} u_i \right)$ fois. À partir de A, on a ainsi conçu un algorithme A' permettant de résoudre SD-O avec une complexité d'au plus $c_A(\tau) \times \log_2 \left(\sum_{i=1}^{n} u_i \right)$, où τ désigne la taille des données de SD-D.

[23] Il y a des abus dans la suite des calculs, les approximations étant traitées comme des égalités. En fait, un calcul plus rigoureux, mais aussi beaucoup plus lourd à décrire, montrerait que les abus ne portent que sur des termes négligeables et ne remettent pas en cause les conclusions qualitatives auxquelles on veut arriver. Il conviendra donc de considérer les calculs qui suivent comme des indications, du reste assez précises, sur les différentes étapes de la démarche plutôt qu'une démonstration parfaitement rigoureuse.

Soit τ' la taille des données associées de SD-O : les données de SD-O et de SD-D ne différent que de *K*, et on a donc $\tau = \tau' + |K|$, où $|K|$ désigne la taille de *K*. On a vu plus haut que coder un entier α en binaire nécessite environ $\log_2 \alpha$ bits. Il vient donc $\tau' \approx \log_2 \pi + \sum_{i=1}^{n} \log_2 p_i + \sum_{i=1}^{n} \log_2 u_i$ et $\tau \approx \tau' + \log_2 K$. On peut supposer qu'on ne s'intéresse qu'aux valeurs de *K* inférieures ou égales à $K_{\max}$ (sans quoi la réponse est trivialement « non »). On a alors $\log_2 K \leq \log_2 K_{\max} = \log_2\left(\sum_{i=1}^{n} u_i\right)$, ce que l'on peut majorer d'abord par $\log_2 n + \log_2 u_{\max}$, où $u_{\max}$ désigne le plus grand des u_i ($1 \leq i \leq n$), puis par $n + \log_2 u_{\max}$; on obtient donc : $\log_2\left(\sum_{i=1}^{n} u_i\right) \leq n + \log_2 u_{\max}$. Par ailleurs, chaque p_i ($1 \leq i \leq n$) nécessitant au moins un bit pour être codé, on a besoin d'au moins *n* bits pour coder tous les p_i ($1 \leq i \leq n$) ; d'où $\log_2 K \leq n + \log_2 u_{\max} \leq \tau'$ et donc $\tau \leq 2\tau'$.

On peut maintenant conclure en regroupant les différents éléments auxquels on a abouti. On a vu que l'application de A' à des données de taille τ' nécessite au plus $c_A(\tau) \times \log_2\left(\sum_{i=1}^{n} u_i\right)$ opérations élémentaires. En supposant que la fonction c_A est croissante (hypothèse vérifiée en pratique), on peut majorer ce nombre par $c_A(2\tau') \times \log_2\left(\sum_{i=1}^{n} u_i\right)$, puis par $\tau' \times c_A(2\tau')$. Si on désigne par $c_{A'}$ la complexité de A', on obtient finalement $c_{A'}(\tau') \leq \tau' \times c_A(2\tau')$. Le majorant de cette inégalité étant, à des polynômes près (un polynôme multiplicatif qui est l'identité, un autre polynôme égal à deux fois l'identité qui intervient par composition), de même nature que c_A, on en déduit que A' n'est pas de complexité qualitativement plus élevée que *A*. En particulier, si *A* est polynomial, alors A' l'est aussi.

De ceci et de la remarque qui suit la formulation du problème PD plus haut, il appert que SD-O et SD-D sont, « à des polynômes près », de même complexité. On constate qu'on a peu fait intervenir les spécificités du problème du sac à dos dans cette illustration. Cela laisse penser que le résultat auquel on vient d'aboutir a des chances de se retrouver pour de nombreux problèmes d'optimisation combinatoire ; c'est en effet le cas.

On peut ne pas se contenter de calculer l'optimum de la fonction à optimiser, mais aussi vouloir une solution permettant d'atteindre cet optimum. Une méthode du même genre que celle que l'on vient de voir, mais qui suppose vérifiées des hypothèses plus contraignantes, montrerait que savoir résoudre un problème de décision permet souvent d'exhiber une solution optimale avec une complexité qualitativement identique. Pour le problème du sac à dos, elle consisterait à calculer le maximum *M* comme on vient de faire, puis à s'intéresser à SD-D en fixant $K = M$ et $x_1 = 1$; si la réponse est « non », on en déduit que toute solution optimale est telle que x_1 est nul : on supprime des données tout ce qui concerne x_1 et on recommence avec les données restantes (en particulier avec $K = M$) ; sinon, c'est qu'il existe une solution optimale avec $x_1 = 1$: on supprime tout ce qui concerne x_1 des données et on recommence avec $K = M - u_1$ et $\pi - p_1$ à la place de π. En itérant le processus *n* fois, on obtient une solution optimale.

3.2. La classe P

Par définition, la classe *P* (pour *polynomial*) ne contient que des *problèmes de décision polynomiaux*, c'est-à-dire que l'on peut résoudre à l'aide d'un algorithme polynomial. Rappelons que « résoudre » signifie pour un problème de décision pouvoir exhiber la réponse « oui » ou « non » selon les données à traiter. Du fait de l'équivalence existant entre algorithme et MTD, on peut définir la classe *P* de la manière suivante.

Définition 6. La classe *P* est l'ensemble des problèmes de décision que l'on peut résoudre à l'aide d'une MTD en un temps polynomial.

S'il est vrai que la classe *P* contient de nombreux problèmes de décision, certains autres ne sont pas connus pour être polynomiaux (ce qui ne signifie pas qu'ils ne le sont pas). C'est le cas par exemple de SD-D. Nous allons maintenant décrire une autre classe, plus vaste, qui contiendra ce problème : la classe *NP*.

3.3. La classe NP

Contrairement à la classe *P*, la classe *NP* (pour *nondeterministic polynomial*[24]) ne préserve pas la symétrie entre la réponse « oui » et la réponse « non ». Elle fait d'autre part appel à des machines non déterministes.

Définition 7. La classe *NP* est l'ensemble des problèmes de décision dont on peut résoudre les données admettant la réponse « oui » en un temps polynomial à l'aide d'une MTND.

La manipulation des MT (déterministes ou non) étant en général délicate et fastidieuse, on raisonne rarement directement sur des MTND quelconques pour montrer qu'un problème est dans *NP*. On préfère plutôt considérer des MTND dans lesquelles on pourra distinguer deux parties : une partie aléatoire suivie d'une partie déterministe. En reprenant les notations du problème générique appelé plus haut PD, la partie aléatoire, qui pourra être du même genre que la MTND de la figure 5, aura pour mission d'exhiber une solution *x* devant donner à *f* une valeur supérieure ou égale à *K*. La partie déterministe, simple MTD, se contentera alors de vérifier que ce *x*, trouvé sans qu'on sache bien comment du fait du caractère aléatoire de la démarche (rappelons-nous que le fonctionnement d'une MTND nécessite de faire des choix quand la fonction de transition associe plusieurs triplets à un couple (état courant, symbole lu) donné et qu'on ne sait pas *a priori* comment effectuer ces choix), convient en effet pour justifier une réponse positive.

[24] Le sigle *NP* est trop souvent perçu, à tort, comme signifiant « non polynomial », ce qui engendre une confusion et rend incompréhensible la relation d'inclusion de *P* dans *NP* (voir théorème 2). Il y aurait sans doute avantage à appeler cette classe *PN* en français ; l'usage consacre néanmoins le sigle anglo-saxon *NP* ; nous suivrons ici cet usage.

On parle alors de *devin*[25]. Comme son nom l'indique, le devin a pour mission de deviner les choix qu'il convient de faire pour construire ce que le devin présente comme une solution permettant de justifier la réponse « oui », c'est-à-dire *x* dans le cas de PD, solution que nous appellerons plus généralement un *certificat succinct* ou simplement *certificat*. Si, pour des données quelconques *D* admettant la réponse « oui », le devin peut exhiber son certificat en un temps polynomial en $|D|$ et si vérifier que ce certificat conduit bien à la réponse « oui » peut se faire aussi en un temps polynomial en $|D|$, on aura montré que le problème traité est dans la classe *NP*. On aurait donc pu proposer pour la classe *NP* la définition suivante au lieu de la définition 7.

Définition 7 bis. La classe *NP* est l'ensemble des problèmes de décision pour lesquels on peut vérifier en un temps polynomial à l'aide d'une MTD qu'un certificat proposé par un devin permet bien de conclure que la réponse est « oui ».

Pour illustrer cette autre façon de voir une MTND, imaginons que plusieurs personnes travaillent sur le problème du sac à dos SD-D (dont on reprend les notations), et traitent des données admettant, à leur insu bien sûr, la réponse « oui ». Arrive (peut-être) un moment où l'une d'entre elles, qui va jouer le rôle du devin, trouve, ou croit trouver, un certificat devant conduire à la réponse « oui ». Ce certificat pourra être, et sera très vraisemblablement, une suite binaire donnant les valeurs des *n* variables x_i ($1 \leq i \leq n$). Les autres personnes ne savent pas comment le devin a effectué ses choix et, si on devait recommencer, peut-être le devin proposerait-il d'autres valeurs ; c'est la partie non déterministe, qui est ici terminée. Commence maintenant la vérification effectuée par les autres. Ceux-ci devront d'abord lire le certificat, ce qui implique au passage que la taille du certificat soit majorée par un polynôme en la taille des données. Ils devront ensuite calculer l'utilité et le poids de la solution proposée par le devin et enfin comparer cette utilité à *K* et ce poids à π pour conclure. Il n'est pas trop difficile de constater que la complexité de l'algorithme de vérification est polynomiale, surtout si l'on adopte le point de vue « macroscopique » des opérations élémentaires. Dans ce cas, on effectue en effet environ *n* affectations pour définir le certificat, puis 2*n* additions pour calculer l'utilité et le poids, et enfin deux comparaisons pour savoir si les inégalités sont satisfaites. Or, on a établi plus haut que la taille des données était au moins de l'ordre de *n*. On peut donc facilement majorer le nombre total d'opérations élémentaires par un polynôme en la taille des données, et nous venons d'établir que SD-D est un élément de *NP* :

Proposition 1. SD-D $\in$ *NP*.

Cet exemple permet aussi de justifier la différence de traitement entre la réponse « oui » et la réponse « non ». Autant le devin peut facilement nous convaincre d'une réponse positive en exhibant une solution qu'il prétend être satisfaisante, autant il semble difficile de nous convaincre, en temps polynomial, d'une réponse négative : pour SD-D, on ne peut pas passer

[25] C'est d'ailleurs ainsi que M.R. Garey et D.S. Johnson (*opus* cité) introduisent directement la notion de MTND : ils présentent une MTND comme une MTD pourvue d'un tel devin, sans utiliser des fonctions de transition pouvant associer plusieurs instructions à un couple (état courant, symbole lu) donné. Si on y gagne en simplicité au moment d'appliquer une MTND à un problème de décision, certains résultats deviennent en revanche plus difficiles à établir, comme le théorème 3 énoncé plus loin. De ce point de vue, le livre de J.-P. Barthélemy *et alii* (*opus* cité) est plus satisfaisant car plus rigoureux.

en revue les 2^n possibilités pour *x*, puisqu'on sort alors des vérifications polynomiales. Bien souvent, pour les problèmes de *NP*, on ne trouve pas (ce qui ne signifie d'ailleurs pas qu'il n'en existe pas) d'argument simple, c'est-à-dire polynomial, pour prouver une réponse négative...

Le problème SD-D est loin d'être le seul à appartenir à *NP*. Comme on va le voir, tous les problèmes de *P* sont aussi dans *NP*. En outre, de nombreux problèmes qui ne sont pas connus pour être dans *P* sont dans *NP*. Mais ne pas réussir à montrer qu'un problème est dans *P* ne signifie pas qu'il n'y est pas réellement. Ce qui nous amène à nous intéresser aux relations entre *P* et *NP*.

3.4. Relations entre P et NP

On a déjà remarqué que toute MTD peut être considérée comme une MTND particulière. On obtient donc immédiatement le théorème suivant.

Théorème 2. $P \subseteq NP$.

La question de savoir si cette inclusion est stricte ou s'il s'agit d'une égalité constitue l'une des questions ouvertes fondamentales de la théorie de la complexité[26]. Sa résolution aurait bien sûr des conséquences en théorie de la calculabilité, mais aussi des répercussions pratiques dans tous les domaines où interviennent l'informatique et l'ordinateur, c'est-à-dire finalement à peu près tous les domaines...

Problème ouvert 1[27]. A-t-on $P \subset NP$ ou $P = NP$?

À défaut de résoudre le problème précédent, on peut montrer que les problèmes de la classe *NP* ne sont pas pour autant d'une complexité aussi grande qu'on pourrait l'imaginer. Le théorème suivant précise cette limitation.

Théorème 3. Soit Π un problème de *NP*. Il existe un polynôme *Q* et une MTD associée à un algorithme (déterministe) de complexité $2^{Q(n)}$ pour résoudre n'importe quelle donnée *D* de Π admettant la réponse « oui » et de taille $|D| = n$.

Idée de la preuve. Dire que Π appartient à *NP* revient à dire qu'il existe une MTND résolvant Π polynomialement. Autrement dit, il existe un polynôme Q' tel que, quelle que soit la

[26] Cette question est aussi l'un des sept problèmes, parfois qualifiés de « problèmes du millénaire », considérés par un comité international de mathématiciens réuni par le *Clay Mathematics Institute* (CMI) comme étant les plus difficiles et les plus importants des mathématiques contemporaines (voir le site du CMI : http://www.claymath.org/millennium/). Le CMI récompense d'un prix de un million de dollars la résolution de chacun de ces problèmes. C'est ainsi que le CMI a annoncé, le 18 mars 2010, l'attribution de ce prix à Grigori Perelman – lequel a refusé le prix, après avoir refusé la médaille Fields qui lui était attribuée pour les mêmes raisons en 2006 – pour ses travaux relatifs à la résolution de la conjecture de Poincaré. Restent donc six problèmes non encore résolus, dont le nôtre (pour une présentation vulgarisée de ces problèmes, voir par exemple le livre de K. Devlin, *The Millenium Problems: The Seven Greatest Unsolved Mathematical Puzzles of our Time*, Basic Books, New York, 2002 ; traduction française de C. Laroche : *Les Énigmes mathématiques du troisième millénaire*, Le Pommier, Paris, 2002).

[27] Nous utiliserons le symbole $\subset$ pour représenter l'inclusion stricte exclusivement.

donnée D de Π admettant la réponse « oui » à traiter, le nombre de pas effectués par la MTND est majoré par $Q'(|D|)$. La MTND étant fixée, il existe une constante k majorant le nombre de triplets associés par la fonction de transition de la MTND à un couple (état, symbole). On peut supposer k supérieur ou égal à 2, sans quoi la MTND est une MTD et la croissance relative d'un polynôme et d'une exponentielle permet de conclure. Il est alors facile de concevoir un algorithme déterministe (qui sera la MTD cherchée) qui envisage successivement tous les triplets possibles quand il y a un choix à effectuer. Par conséquent, le premier pas de la MTND engendrera au plus k pas de la MTD, les deux premiers pas de la MTND engendreront au plus k^2 pas de la MTD, et ainsi de suite, jusqu'au dernier pas de la MTND : les pas, en nombre au plus égal à $Q'(|D|)$, effectués par la MTND engendreront au plus $k^{Q'(|D|)}$ pas de la MTD. De plus, si la MTND conduit à la réponse « oui » en au plus $Q'(|D|)$ pas, notre MTD trouvera aussi la réponse « oui » en au plus $k^{Q'(|D|)}$ pas, puisque les choix effectués par la MTND sont forcément examinés à un moment ou à un autre par notre MTD. L'algorithme déterministe d'exploration exhaustive que nous venons d'envisager permet donc de résoudre Π avec une complexité majorée par $k^{Q'(|D|)}$. Il suffit de poser $Q = Q' \times \log_2 k$ pour pouvoir conclure. ♦

Ce théorème montre qu'il peut y avoir un saut exponentiel entre la difficulté des problèmes de *P* et celle des problèmes de *NP*, mais montre aussi qu'il ne peut pas y avoir, en termes imagés, deux niveaux d'exponentielles les séparant. Une autre façon d'interpréter le résultat consiste à dire que l'introduction d'un fonctionnement non déterministe permet de réduire la complexité des problèmes d'un facteur exponentiel, mais pas de deux ou plus — c'est l'intérêt du devin des MTND : celui-ci réduit qualitativement la complexité en indiquant à la MTD envisagée dans la preuve quels choix effectuer pour adopter un déroulement identique à celui de la MTND. On peut dès lors se demander s'il n'existe pas des problèmes de décision situés à l'extérieur de *NP*. Mais le fait de ne pas savoir répondre à la question posée dans le problème ouvert 1 formulé plus haut indique que la non-vacuité de l'extérieur de *NP* n'est pas un problème simple à aborder...

Dans l'immédiat, nous allons nous intéresser à l'intérieur de *NP*.

3.5. Transformations polynomiales et problèmes NP-complets

Pour structurer l'intérieur de *NP*, nous allons utiliser la notion de *transformation* (ou *réduction*) *polynomiale*, que nous noterons $\prec$.

Définition 8. Soient Π et Π' deux problèmes de décision. On dit que Π *se transforme* (ou *se réduit*) polynomialement en Π' s'il existe un algorithme polynomial transformant toute donnée D de Π en une donnée D' de Π' admettant la même réponse que D. On écrit alors $\Pi \prec \Pi'$.

Il est intéressant de donner une interprétation de la relation $\Pi \prec \Pi'$. Elle consiste finalement à dire que toute donnée D de Π est une donnée de Π' sous une sorte de « déguisement », « déguisement » que traduit l'algorithme polynomial de la définition 8 mais qui ne travestit pas la donnée de Π au point d'en changer la réponse. De ce fait, tout

algorithme A' conçu pour résoudre Π' pourra en particulier traiter ces données « déguisées » et donc indirectement résoudre Π. Or, l'algorithme de déguisement ne « coûte » pas plus qu'un polynôme en $|D|$. Par conséquent, la méthode qui consiste, pour traiter une donnée D de Π, à transformer d'abord D en une donnée D' de Π', puis à traiter D' à l'aide de A' constitue un algorithme de complexité identique, « à des polynômes près », à celle de A'. On en déduit que, toujours « à des polynômes près », Π ne peut pas être plus difficile que Π'. On pourra donc interpréter la relation $\Pi \prec \Pi'$ en « Π n'est pas plus difficile que Π' », ou encore « Π' est au moins aussi difficile que Π ».

Illustrons ce concept de transformation polynomiale sous forme d'un lemme qu'on utilisera plus loin. Elle fait intervenir un nouveau problème, qu'on transformera polynomialement ensuite en SD-D.

Définition 9. On appelle *Partition*[28] le problème de décision suivant :
Nom : Partition ;
Données : un entier m et une suite de m entiers strictement positifs a_i ($1 \leq i \leq m$) de somme paire 2σ ;
Question : existe-t-il $I \subset \{1, 2, \ldots, m\}$ tel que $\sum_{i \in I} a_i = \sigma$?

Lemme 4. Partition $\prec$ SD-D.

Preuve. Considérons une donnée (m ; a_1, a_2, ..., a_m) quelconque de Partition avec $\sum_{i=1}^{m} a_i = 2\sigma$. Associons-lui la donnée (n ; u_1, u_2, ..., u_n ; p_1, p_2, ..., p_n ; π ; K) de SD-D définie par :
• $n = m$;
• pour $1 \leq i \leq n$, $u_i = p_i = a_i$;
• $\pi = K = \sigma$.

Il est facile de vérifier que la définition de la donnée de SD-D ainsi créée ne nécessite qu'un nombre d'opérations polynomial en la taille de la donnée de Partition ; autrement dit, cette transformation est polynomiale.

Montrons qu'elle conserve la réponse.

Supposons que la réponse admise par (m ; a_1, a_2, ..., a_m) soit « oui ». Il existe donc un ensemble d'indices $I \subset \{1, 2, \ldots, m\}$ vérifiant $\sum_{i \in I} a_i = \sigma$. Posons $x_i = 1$ si $i \in I$ et $x_i = 0$ sinon. Il vient alors $\sum_{i=1}^{n} u_i x_i = \sum_{i \in I} a_i = \sigma \geq K$ et $\sum_{i=1}^{n} p_i x_i = \sum_{i \in I} a_i = \sigma \leq \pi$: ce choix des x_i ($1 \leq i \leq n$) montre que la réponse à la donnée (n ; u_1, u_2, ..., u_n ; p_1, p_2, ..., p_n ; π ; K) de SD-D ainsi construite est « oui ».

Réciproquement, supposons que (n ; u_1, u_2, ..., u_n ; p_1, p_2, ..., p_n ; π ; K) admette la réponse « oui ». Soit $x = (x_i)_{1 \leq i \leq n}$ une solution donnant à (n ; u_1, u_2, ..., u_n ; p_1, p_2, ..., p_n ; π ; K) la

[28] Aussi appelé « Partage » selon les auteurs cités plus haut en références.

réponse « oui ». Posons $I = \{i$ tel que $1 \leq i \leq n$ et $x_i = 1\}$. On a alors $\sum_{i \in I} a_i = \sum_{i=1}^{n} u_i x_i \geq K = \sigma$ et $\sum_{i \in I} a_i = \sum_{i=1}^{n} p_i x_i \leq \pi = \sigma$: la réponse admise par (m ; a_1, a_2, ..., a_m) est donc « oui ».

La transformation envisagée étant polynomiale et conservant la réponse, on a bien montré la relation Partition $\prec$ SD-D. ♦

Il est facile d'établir la proposition suivante, qui résulte du fait que l'ensemble des polynômes est stable par composition.

Proposition 5. La relation $\prec$ définit un préordre[29] sur l'ensemble des problèmes de décision.

La classification des problèmes qu'élabore la théorie de la complexité ne distinguant pas des polynômes de degrés différents, on ne sera pas surpris par le résultat suivant, dont nous ne donnerons pas une preuve formelle, mais seulement des indications de preuve.

Proposition 6. $\forall\ \Pi \in P,\ \forall\ \Pi' \in NP,\ \Pi \prec \Pi'$.

Idée de la preuve. Pour toute donnée D de Π, il suffit de résoudre D polynomialement (ce qui est possible puisque Π appartient à P) puis, si la réponse est « oui » (respectivement « non ») lui associer une donnée fixée de Π' admettant la réponse « oui » (respectivement « non »)[30]. Cette transformation est bien polynomiale (le calcul de la réponse l'est par hypothèse sur Π, et la construction de l'image de D est de complexité constante) et, par construction, préserve la réponse. D'où la relation. ♦

Ce résultat montre que les problèmes de P sont les « plus faciles » (au sens de $\prec$) de NP. On peut en déduire certains autres résultats, par exemple :
• $(\Pi \prec \Pi'$ et $\Pi' \in P) \Rightarrow \Pi \in P$;
• $(\Pi \prec \Pi'$ et $\Pi \notin P) \Rightarrow \Pi' \notin P$;
• $P = NP \Leftrightarrow (\exists\ \Pi' \in P$ tel que $\forall\ \Pi \in NP,\ \Pi \prec \Pi')$...

La dernière relation reviendrait à montrer l'existence d'un problème de décision polynomial au moins aussi difficile que tout problème de NP. À défaut de pouvoir exhiber un tel problème ou de démontrer qu'il n'en existe pas (ce qui reviendrait à résoudre le problème ouvert 1), on peut s'interroger sur l'existence, non triviale, d'un problème de NP au moins aussi difficile que tout problème de NP. Le théorème fondateur de S.A. Cook[31] (1971) montre qu'un tel problème existe. Avant de l'énoncer et d'en donner une vague idée de preuve, nous avons besoin de certaines définitions issues du domaine de la logique.

---

[29] Rappelons qu'un préordre est une relation réflexive et transitive.

[30] Une erreur classique consiste à faire dépendre la transformation de la donnée qu'on veut modifier. Cela est bien sûr illicite, puisque cette réponse ne fait pas partie de la donnée (sans quoi on ne la chercherait pas !), sauf si on peut la trouver en temps polynomial, autrement dit sauf si le problème appartient à P, ce qui est le cas ici.

[31] S.A. Cook, « The complexity of theorem-proving procedures », *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, Association for Computing Machinery, New York, 1971, 151-158.

Définitions 10.
• Soit *u* une variable booléenne ; on note $\overline{u}$ le *conjugué* associé à *u*, c'est-à-dire la variable booléenne qui vaut « vrai » (respectivement « faux ») quand *u* vaut « faux » (respectivement « vrai »). Soit *U* une famille de variables booléennes ; on note $\overline{U}$ la famille des conjugués des éléments de *U*. Un élément de $U \cup \overline{U}$ s'appelle un *littéral*.
• Soit *U* une famille de variables booléennes. On appelle *fonction d'assignation* de *U* toute fonction définie de *U* dans {faux, vrai}.
• Soit *U* une famille de variables booléennes. On appelle *clause* définie sur *U* une formule *C* logique de la forme[32] $w_1 \vee w_2 \vee \ldots \vee w_{k(C)}$ (où $k(C)$ désigne le nombre de littéraux de *C*) telle que, pour $1 \leq i \leq k(C)$, $w_i \in U \cup \overline{U}$. On dit qu'une clause *C* définie sur *U* est *satisfiable* s'il existe une fonction d'assignation ϕ de *U* telle que *C* contienne au moins un littéral dont l'image par ϕ soit égale à « vrai ».
• On appelle *Satisfiabilité* (ou simplement *SAT*) le problème suivant :
Nom : SAT ;
Données : une famille *U* de variables booléennes, une famille de clauses définies sur *U* ;
Question : existe-t-il une fonction d'assignation définie sur *U* permettant de satisfaire simultanément toutes les clauses ?

Théorème 7 (théorème de Cook, 1971). $\forall\ \Pi \in NP,\ \Pi \prec \mathrm{SAT}$.

Idée de la preuve. Il est exclu de détailler ici la preuve du théorème de Cook (le lecteur intéressé la trouvera dans les ouvrages cités plus haut). Disons seulement que l'idée globale consiste à modéliser le comportement d'une MTND sous forme de clauses logiques.

Pour cela, on considère des variables booléennes de trois types :
• les premières traduisent le fait qu'à un instant donné (le temps étant mesuré par le nombre de pas effectués par la MTND), la MTND est ou non dans un état donné ;
• les deuxièmes traduisent le fait qu'à un instant donné, la tête de lecture-écriture pointe ou non sur une case donnée ;
• les troisièmes traduisent le fait qu'à un instant donné et dans une case donnée se trouve ou non un symbole donné.

Les clauses se répartissent en six groupes. Ils correspondent respectivement aux six propriétés suivantes :
• à tout instant, la MTND est dans un état et un seul ;
• à tout instant, la tête de lecture-écriture pointe sur une case et une seule ;
• à tout instant, chaque case de la bande contient un symbole et un seul ;
• à tout instant autre que celui de départ, la configuration de la MTND résulte de l'application de la fonction de transition à la configuration associée à l'instant précédent ;
• à l'instant de départ, les données sont enregistrées sur la bande et la MTND est dans sa configuration initiale ;
• au dernier instant, la MTND est dans la configuration qu'elle doit avoir quand les calculs sont terminés.

Grâce à tout ceci, on peut modéliser le comportement d'une MTND quelconque à l'aide de formules logiques. Quand on veut ensuite transformer un problème Π de *NP* en SAT, il suffit de considérer une MTND permettant de résoudre Π et de traduire celle-ci sous forme de

[32] Le symbole « $\vee$ » désigne le « ou » logique.

clauses pour obtenir une donnée de SAT admettant la même réponse que la donnée de Π transformée. La traduction sous forme de clauses pouvant être faite en temps polynomial, on peut conclure que tout problème Π de *NP* n'est pas plus difficile que SAT, ou encore que SAT est au moins aussi difficile que tout problème de *NP*. ♦

Les problèmes de *NP* au moins aussi difficiles que tout autre problème de *NP* portent un nom que précise la définition suivante.

Définition 11. Un *problème NP-complet* est un problème Π de *NP* vérifiant la propriété suivante : $\forall\ \Pi' \in NP\ ,\ \Pi' \prec \Pi$. On note *NPC* l'ensemble des problèmes *NP*-complets.

Autrement dit, les problèmes *NP*-complets sont les problèmes les plus difficiles de la classe *NP*. On peut ainsi reformuler le théorème de Cook en disant que SAT est *NP*-complet. Il existe de nombreux problèmes *NP*-complets : M.R. Garey et D.S. Johnson[33] en recensent plusieurs centaines, chacun d'entre eux pouvant à son tour donner de nombreuses variantes *NP*-complètes[34], dans des domaines très divers des mathématiques et de l'informatique : théorie des graphes, conception des réseaux, théorie des ensembles, ordonnancement de tâches, programmation mathématique, algèbre, théorie des nombres, théorie des jeux, logique, théorie des automates, etc.

D'un point de vue pratique, il est à la fois fastidieux et inutile, pour montrer qu'un problème de décision est *NP*-complet, de procéder comme on vient de faire pour établir la *NP*-complétude de SAT. En effet, la transitivité de la relation $\prec$ (voir la proposition 5) permet de démontrer le théorème suivant.

Théorème 8. Soit $\Pi \in NP$. S'il existe $\Pi' \in NPC$ vérifiant $\Pi' \prec \Pi$, alors $\Pi \in NPC$.

Pour montrer qu'un problème de décision Π est *NP*-complet, il suffit donc d'adopter la démarche suivante (impossible cependant à appliquer pour démontrer le théorème de Cook, puisque c'est celui-ci qui montre que les problèmes *NP*-complets existent) :

• montrer que Π appartient à *NP* ;
• choisir un problème Π' connu pour être *NP*-complet ;
• établir la relation $\Pi' \prec \Pi$.

Ainsi, à partir de SAT, on montre progressivement la *NP*-complétude d'autres problèmes qui, à leur tour, peuvent être utilisés pour démontrer la *NP*-complétude de nouveaux problèmes, et ainsi de suite. Par exemple, dans cette cascade de résultats, on arrive assez rapidement à établir la *NP*-complétude du problème qu'on a appelé plus haut Partition. À l'aide de ce résultat, on obtient le théorème 9 comme corollaire de la proposition 1 et du lemme 4.

Théorème 9. SD-D $\in NPC$.

---

[33] *opus* cité.

[34] Pour donner une idée du nombre de variantes que l'on peut atteindre, citons un dénombrement dû à B.J. Lageweg, E.L. Lawler, J.K. Lenstra et A.H.G. Rinnooy Kan, « Computer aided classification of deterministic scheduling problems », Mathematisch Centrum, 1978. À partir d'un problème d'ordonnancement appelé *Preemptive Scheduling* en anglais, ils ont énuméré 4536 problèmes d'ordonnancement dont 3730 (soit 82 %) sont associés à des problèmes de décision *NP*-complets, 416 (soit 9 %) sont polynomiaux et 390 (soit 9 %) étaient de complexité inconnue à l'époque.

Le problème du sac à dos SD-D peut alors être lui-même utilisé pour montrer la *NP*-complétude d'autres problèmes, par exemple le problème de décision associé à un problème important en optimisation combinatoire : celui de la programmation linéaire en nombres entiers[35].

3.6. Structure de NP

On a déjà avoué notre ignorance concernant la possible identité entre *P* et *NP*. On est donc confronté à deux hypothèses. Si *P* est égal à *NP*, on a tout dit sur la structure de *NP* : c'est simplement celle de *P*[36] ; cette situation est illustrée par le dessin de gauche de la figure 6. Si *P* diffère de *NP*, alors on montre facilement que *P* et *NPC* sont disjoints :

Proposition 10.
• $P \neq NP \Leftrightarrow P \cap NPC = \varnothing$;
• $P = NP \Leftrightarrow \exists\ \Pi \in P, \exists\ \Pi' \in NPC$ vérifiant $\Pi' \prec \Pi$.

Vient inévitablement une nouvelle question : dans le cas où *P* et *NP* sont distincts, qu'y a-t-il entre *P* et *NPC* dans *NP* ? Le vide ? Un nombre fini de classes de difficultés intermédiaires ? Un nombre infini de telles classes ? Le théorème suivant, conséquence d'un résultat dû à R.E. Ladner[37], montre que cette dernière hypothèse est la bonne.

Théorème 11 (théorème de Ladner, 1975). Si $P \neq NP$, alors $\forall\ \Pi \in NP - P, \exists\ \Pi' \in NP - P$ tel que $\Pi' \prec \Pi$ et $\Pi \not\prec \Pi'$.

Autrement dit, pour tout problème Π de *NP* non polynomial, on peut construire un problème non polynomial et de difficulté qualitativement (c'est-à-dire, pas seulement « à des polynômes près ») différente de celle de Π. En recommençant une telle construction, on obtient un nombre infini de classes de problèmes de difficultés différentes (et d'ailleurs non nécessairement comparables, au sens de $\prec$).

On peut illustrer la situation $P \neq NP$ par le dessin de droite de la figure 6 (où les classes de complexités intermédiaires, situées entre *P* et *NP*, sont symbolisées par des espèces de briques).

---

[35] Celui-ci consiste à optimiser, à l'aide de variables à valeurs entières, une forme linéaire soumise à des contraintes linéaires. Pour démontrer le résultat annoncé, il suffit *grosso modo* de constater que SD-O est un tel problème de programmation linéaire en nombres entiers et d'utiliser l'identité comme transformation polynomiale...

[36] Tout n'est pas tout à fait dit pour autant : on peut aussi s'intéresser à une structure plus fine de *P*, en s'intéressant par exemple aux problèmes dont on peut résoudre les données *D* à l'aide d'algorithmes dont la complexité est majorée par un polynôme en $\log|D|$ après lecture des données elles-mêmes (laquelle est forcément linéaire en $|D|$). Ces considérations sortent de notre cadre.

[37] R.E. Ladner, « On the structure of polynomial time reducibility », *Journal of the Association for Computing Machinery* 22, 1975, 155-171.

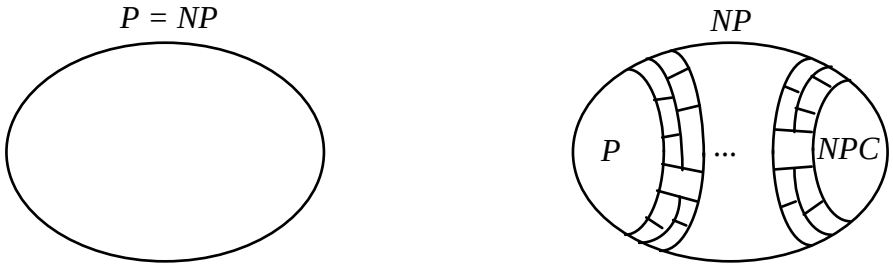

Figure 6. Illustration des deux cas $P = NP$ (à gauche) ou $P \neq NP$ (à droite).

3.7. La classe co-NP

On a remarqué, au moment de définir la classe *NP*, la dissymétrie entre les réponses « oui » et « non » introduite par cette définition. On peut alors tout reprendre en remplaçant « oui » par « non » dans les définitions des classes *P*, *NP* et *NPC* : on obtient de cette manière respectivement les classes co-*P*, co-*NP*, co-*NPC*. De manière assez immédiate, on voit que si un problème Π appartient à une de ces classes *C*, alors le problème co-Π obtenu par négation de Π appartient à co-*C*. Ainsi, la négation co-SD-D de SD-D est un problème co-*NP*-complet :

Proposition 12. Le problème co-SD-D appartient à co-*NPC*, où co-SD-D est défini par :
Nom : co-SD-D ;
Données : un entier *n*, *n* entiers u_i ($1 \leq i \leq n$), *n* entiers p_i ($1 \leq i \leq n$), un entier π, un entier *K*, tous ces entiers étant strictement positifs ;
Question[38] : quel que soit $x = (x_i)_{1 \leq i \leq n} \in \{0, 1\}^n$ vérifiant $\sum_{i=1}^{n} p_i x_i \leq \pi$, a-t-on $\sum_{i=1}^{n} u_i x_i \leq K$?

Du fait de la symétrie jouée par les réponses « oui » et « non » dans la définition de *P*, on obtient facilement les deux résultats de la proposition 13.

Proposition 13. $P =$ co-$P \subseteq$ co-*NP*.

Fort de ce résultat, on peut légitimement s'interroger sur les liens entre *NP* et co-*NP*. De même qu'on ne sait pas conclure à propos des liens exacts entre *P* et *NP*, l'égalité ou la distinction de *NP* et co-*NP* reste inconnue, ce qui constitue le deuxième problème ouvert.

Problème ouvert 2. A-t-on $NP =$ co-*NP* ou $NP \neq$ co-*NP* ?

Compte tenu de ce qui a déjà été dit, on constate que l'on peut avoir $NP =$ co-*NP* sans avoir $P = NP$ (en revanche, l'égalité $P = NP$ entraîne bien sûr l'égalité $NP =$ co-*NP* puisque alors toutes ces classes sont confondues). En revanche, si *NP* et co-*NP* sont distincts (ce qui implique donc que *P* est distinct à la fois de *NP* et de co-*NP*), on peut montrer le résultat de la proposition 14, selon lequel aucun problème ne peut être à la fois *NP*-complet et co-*NP*-complet :

[38] Le lecteur notera le choix adapté des quantificateurs par rapport à la négation du problème SD-D.

Proposition 14. Si $NP \neq$ co-NP, alors $NPC \cap$ co-$NPC = \varnothing$.

En fait, d'une manière plus générale, si *NP* et co-*NP* sont distincts, aucun problème co-*NP*-complet n'est dans *NP* et aucun problème *NP*-complet n'est dans co-*NP*. Ainsi co-SD-D n'est pas connu pour être dans *NP* et SD-D n'est pas connu pour être dans co-*NP* (une telle connaissance résoudrait le problème ouvert 2, dont la solution serait alors *NP* = co-*NP*). Par ailleurs, *P* étant égal à co-*P*, on a déjà remarqué que *P* est à la fois dans *NP* (théorème 2) et dans co-*NP* (proposition 13), donc dans leur intersection. Ces résultats sont résumés par la proposition 15, qui généralise la proposition précédente.

Proposition 15.
• Si $NP \neq$ co-NP, alors $NPC \cap$ co-$NP = \varnothing$ et co-$NPC \cap NP = \varnothing$.
• $P \subseteq NP \cap$ co-NP.

La dernière inclusion suscite une nouvelle fois une interrogation : l'intersection de *NP* et de co-*NP* est-elle réduite à *P*, ou y a-t-il dans $NP \cap$ co-*NP* d'autres problèmes que ceux de *P* ? De nouveau, on ne connaît pas la réponse :

Problème ouvert 3. A-t-on $P = NP \cap$ co-NP ou $P \subset NP \cap$ co-NP ?

3.8. Résumé

Résumons les différentes possibilités rencontrées précédemment, en fonction des réponses aux trois problèmes ouverts énoncés plus haut. Elles sont illustrées par les figures 6 et 7.

Le premier problème ouvert rencontré porte sur l'identité de *P* et *NP*. Si *P* et *NP* sont confondus, le dessin de gauche de la figure 6 résume l'ensemble des classes définies plus haut, puisqu'on a alors $P = NP = NPC =$ co-$NP =$ co-NPC. Sinon, il existe une infinité de classes dans *NP*, partiellement ordonnées (il s'agit de l'ordre induit par la relation $\prec$). Cet ordre admet un élément minimal : la classe *P*, constituée des problèmes les plus faciles (au sens donné jusqu'à présent) de *NP*, et un élément maximal dans *NP* : la classe *NPC*, constituée des problèmes les plus difficiles (toujours au même sens) de *NP*.

Apparaît ensuite le deuxième problème ouvert, relatif à l'égalité de *NP* et co-*NP*. Si *NP* et co-*NP* sont égaux, la fin du paragraphe précédent suffit à décrire la situation, illustrée par le dessin de droite de la figure 6. Sinon, on sait que l'intersection de *NP* et de co-*NP* contient au moins *P*, tandis que l'intersection de *NP* et de co-*NPC* est vide, ainsi que l'intersection de *NPC* et de co-*NP*. On sait de plus qu'il existe une infinité de classes, certaines incomparables les unes aux autres, de complexités intermédiaires entre *P* et *NPC* d'une part, *P* et co-*NPC* d'autre part. Cette situation est illustrée par la figure 7, sans qu'on s'y prononce sur la vacuité ou la non-vacuité de $(NP \cap$ co-$NP) - P$, ce qui constitue l'objet du troisième problème ouvert.

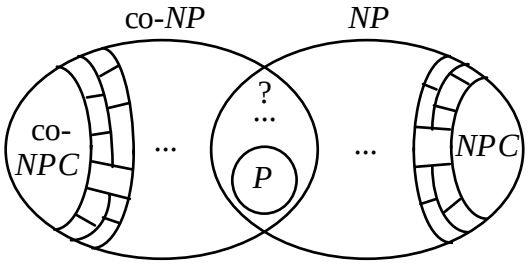

Figure 7. Cas $NP \neq$ co-NP

4. Conclusion

Cette présentation ne prétend pas épuiser le sujet, loin de là. Son ambition est plutôt de montrer comment les machines de Turing, initialement conçues pour résoudre l'*Entscheidungsproblem*, sont devenues un outil fondamental pour étudier la complexité algorithmique et pour classer les problèmes selon leur difficulté.

Elles nous ont d'abord fourni un modèle universel d'ordinateur et d'algorithme. À partir de cela, on a pu définir les classes de base de la complexité des problèmes : *P*, *NP*, *NPC*, puis co-*NP* et co-*NPC*. On pourrait aller plus loin, soit à l'intérieur de *NP* (voire de *P*), soit à l'extérieur.

À l'intérieur de *NP*, on pourrait par exemple s'intéresser aux problèmes qui sont polynomiaux quand on accepte de coder les entiers en unaire, bien que nous ayons considéré un tel codage comme une hérésie jusqu'à maintenant, à cause des tailles de codage déraisonnables qu'il engendre. De tels problèmes sont qualifiés de *pseudo-polynomiaux*. On constaterait que certains problèmes *NP*-complets sont pseudo-polynomiaux ; c'est par exemple le cas de SD-D ou de Partition. Mais d'autres, par exemple SAT, persisteraient malgré cette permissivité à ne pas paraître polynomiaux ; ceux-ci seront dits *fortement NP-complets*.

À l'extérieur de *NP*, outre les problèmes de co-*NP* si co-*NP* et *NP* sont distincts, on pourrait s'intéresser aux problèmes, de décision ou non, au moins aussi difficiles que des problèmes *NP*-complets, au sens que l'on a considéré plus haut, c'est-à-dire des problèmes admettant des algorithmes de résolution qu'on pourrait utiliser pour résoudre les problèmes *NP*-complets sans accroître la complexité de la résolution, « à des polynômes près » bien sûr... De tels problèmes sont qualifiés de *NP-difficiles*. De cette « définition »[39], il résulte que les problèmes *NP*-complets sont *NP*-difficiles. Il serait assez facile de montrer que les problèmes co-*NP*-complets sont aussi *NP*-difficiles. On déduirait aussi des remarques faites vers le début de la partie 3.1. que des problèmes d'optimisation dont le problème de décision associé est *NP*-complet sont eux-mêmes *NP*-difficiles. C'est le cas par exemple de SD-O. Résoudre un jour un problème *NP*-difficile en un temps polynomial permettrait alors de résoudre le problème ouvert 1 (et du même coup les deux autres) en montrant que *P* et *NP* coïncident. La conséquence pratique de la complexité des problèmes *NP*-difficiles est qu'on ne connaît pas

[39] Une véritable définition verrait le retour en force de certaines machines de Turing, dites *machines de Turing à oracle*, plus sophistiquées que celles présentées ici et qui permettraient de définir une transformation plus puissante que la transformation polynomiale décrite dans cette introduction : les *transformations de Turing*. De telles considérations sortent de notre cadre, mais le lecteur pourra, une fois de plus, se reporter avec profit aux livres cités au début.

d'algorithme polynomial pour les résoudre. On doit donc, si on veut une solution exacte, mettre en œuvre des méthodes exponentielles, avec le risque de consommer un temps de calcul rapidement prohibitif (voir le tableau de la figure 4). Ou alors il faut se contenter de solutions approchées. La résistance plus ou moins grande des problèmes à se laisser approcher par de bonnes solutions en un temps polynomial est encore un autre sujet d'étude possible du point de vue de la complexité[40].

Une investigation plus profonde nous conduirait, incités par le théorème 3, à nous interroger sur la possibilité de poursuivre la classification des problèmes de décision au-delà de *NP*. On construirait alors la *hiérarchie polynomiale*, dont la base est constituée de *P*, de *NP* et de co-*NP*. Elle permettrait d'aborder des problèmes de décision enchevêtrant les quantificateurs existentiels (on remarquera que la plupart des problèmes de *NP* posent justement une question faisant intervenir un tel quantificateur ; c'est par exemple le cas de SD-D) et les quantificateurs universels (on remarquera de même que la plupart des problèmes de co-*NP* posent une question avec un tel quantificateur ; c'est ainsi le cas de co-SD-D). On y placerait par exemple des problèmes relatifs à l'unicité d'une solution optimale (en reprenant les notations du problème appelé POC plus haut, la question serait de la forme : existe-t-il $x^* \in X$ tel que, quel que soit $x \in X - \{x^*\}$, $f(x^*) > f(x)$?). On aboutirait à de nouveaux problèmes ouverts, par exemple celui de savoir si cette construction hiérarchique s'arrête au bout d'un nombre fini de classes ou non.

On pourrait aussi s'intéresser à la *complexité en place*, c'est-à-dire celle qui serait fondée sur le nombre de cases d'une MT utilisées pour effectuer un calcul, ou encore étudier l'apport de nouvelles ressources techniques comme le parallélisme...

Tous ces thèmes, situés en dehors de notre présent objectif, montrent la richesse de la théorie de la complexité. Quant aux problèmes ouverts sur lesquels on bute souvent, ils en soulignent aussi la difficulté. Finalement, ces divers aspects, alliés aux retombées théoriques et pratiques des résultats qu'on peut espérer obtenir un jour, ne constituent-ils pas aussi de puissants stimulants, propres à susciter l'intérêt du chercheur ?

Index

[40] Voir à ce propos M. Yannakakis, « Computational complexity », *in* Local Search in Combinatorial Optimization, sous la direction d'E. Aarts et J.K. Lenstra, Wiley, Chichester, 1997, 19-54 et V. Paschos, *Complexité et approximation polynomiale*, Hermès, Paris, 2004.