Complexité des algorithmes et problèmes difficiles

Par JMV
Association LinuxMaine
Pôle Coluche, 72000 LE MANS

Table des matières

Préambule

À la genèse de ce document se trouve la présentation du Samedi 13 Septembre 2025 faite par Thierry Gayet dans les locaux de l'association mancelle LinuxMaine sur des sujets aussi divers que la cryptographie et les ordinateurs quantiques ; le besoin d'y apporter un éclairage sur la théorie de la complexité s'est vite fait sentir.

Puis en devenant indépendant de cette présentation ce document a évolué vers une sensibilisation à la notion de complexité d'un algorithme ainsi que celle de problèmes difficiles pour un ordinateur, tout cela rédigé à l'intention de lecteurs non scientifiques. Le fait de s'adresser à un public non scientifique impose un formalisme épuré où prévaut le langage courant. Durant cette vulgarisation, le souci que la rigueur n'en souffre pas est néanmoins resté constamment présent.

Dans un premier temps nous évoquerons la notion de complexité d'un algorithme (complexité en temps d'exécution et dans le pire des cas).
Un passage par les machines de Turing, ces modèles théoriques d'ordinateurs, devrait permettre de positionner ordinateurs classiques et ordinateurs quantiques dans ce cadre.
Nous présenterons également quelques problèmes réputés difficiles pour toucher du doigt la véritable problématique du temps d'exécution.
Bien que le sujet des problèmes NP-complets ne soit pas absolument nécessaire à la bonne compréhension de la présentation de Thierry, ne pas les évoquer dans un document indépendant qui traite de la complexité des algorithmes et de problèmes difficiles serait un manquement.

Bonne lecture.

1. Complexité des algorithmes et problèmes difficiles.

Il s'agit ici de comprendre ce qu'est la complexité théorique des algorithmes puis grâce à quatre exemples de problèmes réputés difficiles de voir exactement en quoi ils sont difficiles.

1.1. Complexité des algorithmes

Il y a 50 ans, les programmeurs de l'époque étaient surtout attentifs à la mémoire que leur programme allait utiliser. Cette mémoire était chère et pour ce qu'on leur faisait calculer, les ordinateurs étaient bien assez rapides.
De nos jours, les barrettes de RAM ne coûtent plus grand-chose. De plus les utilisateurs sont de plus en plus nombreux et les données de plus en plus volumineuses.
L'adage qui prévaut aujourd'hui est plutôt *le temps c'est de l'argent.*
On va donc chercher à évaluer le temps d'exécution d'un algorithme. Il ne s'agit pas de faire une mesure à la milliseconde près, ni même à la seconde près mais plutôt d'obtenir un **ordre de grandeur** d'une **borne supérieure** du temps d'exécution **dans le pire des cas**[1] en fonction de la **taille de la donnée**.
Dans ce contexte l'expression *« dans le pire des cas »* fait référence à une situation où l'algorithme devrait tester *toutes* les situations possibles avant de terminer. Imaginons par exemple le problème de trouver un mot de passe de 10 lettres par essais successifs, dans le pire des cas, il faudrait tester tous les mots de 10 lettres et ne trouver la réponse que lors de la dernière tentative. Dans un alphabet classique cela revient à tester 26^{10} possibilités.
La notation *O(fonction de n)* fournit une réponse à ce genre d'évaluation.

1.1.1. La notation *O(g)*

Étant données deux fonctions *f* et *g* d'un entier naturel à valeurs réelles positives, on dit que *f* est un grand *O* de *g*, noté $f = O(g)$, lorsqu'à partir d'un certain seuil, *f(n)* est majorée par $K \times g(n)$ où *K* est une constante positive.
Si on considère que *n* est la **taille** de la donnée, que *f(n)* représente le nombre d'opérations élémentaires requises pour l'exécution d'un algorithme, que *K* représente l'ordre de grandeur, on obtient une majoration de *f(n)* pour des données de grande taille.
Évidemment on ne retient que la plus petite des majorations comme borne supérieure.
On dispose ainsi d'une borne supérieure pour l'exécution de l'algorithme dans le pire des cas pour des données de très grande taille.

Le rôle du seuil : on s'intéresse à majorer le temps d'exécution pour de grandes tailles de données (grandes valeurs de *n).* Il importe peu que cette majoration soit valable pour des petites valeurs de *n*, il suffit qu'elle soit valable à partir d'un certain seuil.

1.1.2. Un exemple : la multiplication de matrices carrées

Soit *n* un nombre entier naturel. Considérons par exemple l'opération de multiplication d'un matrice carrée d'ordre *n* donnée *A* par *B*, une autre matrice carrée d'ordre *n.* Le résultat du calcul est une

1 On parle aussi de complexité dans le cas le plus défavorable.

matrice carrée d'ordre *n*.
La définition de l'opération induit l'algorithme :

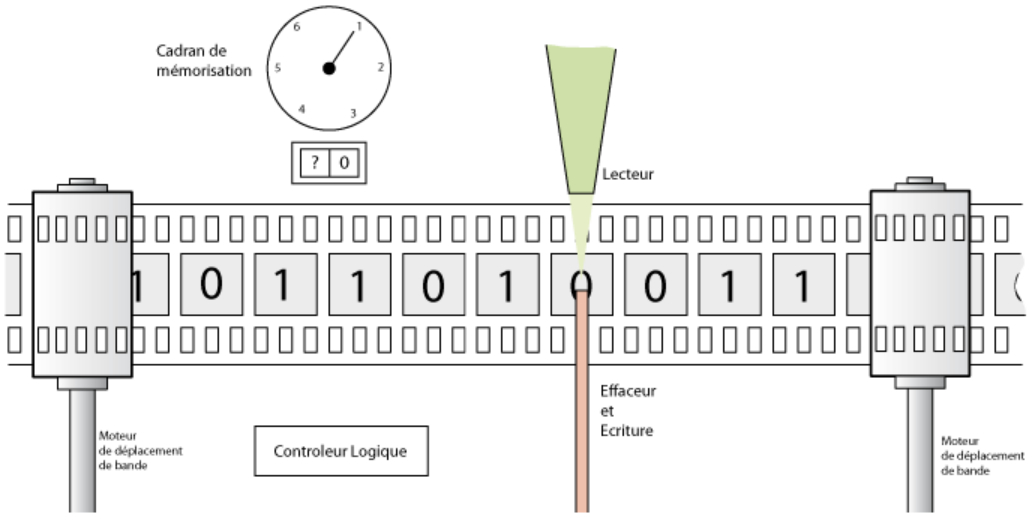
Pour chaque ligne i de la matrice A
Pour chaque colonne j de la matrice B
Calculer le produit scalaire du vecteur de la ligne i de A avec le vecteur de la colonne j de B, c'est le coefficient situé ligne i et colonne j de la matrice résultat.

Complexité : pour simplifier les opérations sur les coefficients sont considérées comme élémentaires. En tant que produit scalaire chaque coefficient de la matrice résultat est une somme cumulée de *n* produits, le calcul d'un coefficient est donc un $O(n)$ et comme il y a n^2 coefficients à calculer, la complexité en temps de cet algorithme dans sa totalité[2] est donc $O(n^3)$.

La complexité $O(n^3)$ est dite **polynomiale**, plus généralement c'est le cas pour les complexités de la forme $O(n^k)$ où *k* est un nombre entier naturel.

1.2. Machine de Turing[3]

Contrairement à ce que le mot *machine* pourrait laisser penser, une machine de Turing est une vue de l'esprit, c'est un modèle abstrait, théorique, conçu en 1936 par Alan Turing[4].



Définition :

Une machine de Turing comporte les éléments suivants :

1. Un **ruban infini** divisé en cases consécutives. Chaque case contient un symbole d'un alphabet fini donné. Ce ruban modélise la mémoire d'un ordinateur
2. Une **tête de lecture/écriture** qui peut lire et écrire les symboles sur le ruban, et se déplacer vers la gauche ou vers la droite du ruban ;
3. Un **registre d'état** qui mémorise l'état courant de la machine de Turing. Le nombre d'états possibles est toujours fini, et il existe un état spécial appelé « état de départ » qui est l'état initial de la machine avant son exécution ;

2 Des améliorations successives ont pu ramener la complexité de la multiplication des matrices à $O(n^{2.371339})$ *en 2025 (voir https://fr.wikipedia.org/wiki/Complexité_de_la_multiplication_de_matrices)*.
3 Voir https://fr.wikipedia.org/wiki/Machine_de_Turing
4 Voir https://fr.wikipedia.org/wiki/Alan_Turing

4. Une **table d'actions** qui indique à la machine quel symbole écrire sur le ruban, comment déplacer la tête de lecture (par exemple « $\leftarrow$ » pour une case vers la gauche, « $\rightarrow$ » pour une case vers la droite), et quel est le nouvel état, en fonction du symbole lu sur le ruban et de l'état courant de la machine. Si aucune action n'existe pour une combinaison donnée d'un symbole lu et d'un état courant, la machine s'arrête.

   En outre, si dans la table d'actions, pour un état et un symbole donnés il n'y a qu'**une seule action** possible la machine est dite **déterministe**, dans le cas contraire elle est **non-déterministe**.

Les ordinateurs modernes sont construits sur ce modèle avec l'alphabet $\{0,1\}$.
Ils sont des réalisations concrètes de machines de Turing déterministes.

Machines de Turing non-déterministes[5]

Alors que, connaissant le caractère lu sur le ruban et l'état courant, une machine de Turing déterministe dispose d'au plus une transition possible, une machine de Turing non déterministe peut en avoir plusieurs. En conséquence, tandis que les calculs d'une machine de Turing déterministe forment une suite, ceux d'une machine de Turing non déterministe forment un arbre, dans lequel chaque chemin correspond à une suite de calculs possibles.
On peut se représenter l'évolution d'une machine de Turing non déterministe ainsi : dans un état où il y a plusieurs transitions possibles, elle se duplique (triplique, etc.) et une sous-machine est créée pour chaque transition différente. Une machine de Turing non déterministe accepte une entrée s'il existe une séquence de choix (une branche de l'arbre, une sous-machine) qui atteint un état acceptant.

Une autre façon de se représenter les choix d'une machine de Turing non déterministe est de l'imaginer aussi chanceuse que possible : autrement dit, s'il existe une suite de choix qui aboutit à un état final acceptant, la machine fait cette suite de choix parmi les suites de choix de transitions possibles.

2. Des problèmes réputés difficiles

Dans cette partie nous évoquerons certains problèmes réputés difficiles à traiter pour des ordinateurs classiques ; nous en viendrons à évoquer les classes **P** et **NP** de problèmes et terminerons par trois problèmes difficiles : le problème de satisfiabilité, autrement dit le problème SAT, ainsi que deux problèmes d'optimisation : le problème du sac à dos et celui du voyageur de commerce.

2.1. Énumérer les parties d'un ensemble

Il est fréquent dans ce domaine d'avoir à exhiber une partie d'un ensemble ayant certaines propriétés, il paraît donc opportun de vouloir énumérer **toutes les parties** d'un ensemble pour ne retenir ensuite que celles qui nous intéressent.

5 https://fr.wikipedia.org/wiki/Machine_de_Turing_non_d%C3%A9terministe

Par exemple, pour un ensemble à deux éléments : $\{a,b\}$ il y a quatre parties :

l'ensemble vide $\varnothing$ à 0 élément, deux singletons $\{a\}et\{b\}$ et l'ensemble $\{a,b\}$ lui même.

Par ailleurs on sait que pour un ensemble de *n* éléments, il y a 2^n parties.

Le professeur YAKA-FAUCON, mathématicien émérite, multi médaillé *Fields* s'est d'ailleurs exprimé sur ce sujet de l'énumération des parties d'un ensemble à *n* éléments :

> « *Je ne vois pas où est le problème* dit-il, *il suffit de placer les éléments dans un certain ordre, de numéroter les parties de 0 à* 2^n*-1, c'est une numérotation à* n *chiffres binaires. Le code binaire de chaque numéro de partie détermine la composition de cette partie* »

Exemple avec un ensemble à 3 éléments $\{a,b,c\}$, il y a 8 parties à lister numérotées de 0 à 7.

Numéro de partie	Code binaire du numéro	a est présent dans la partie	b est présent dans la partie	c est présent dans la partie	Partie
0	000	non	non	non	$\varnothing$
1	001	non	non	oui	$\{c]$
2	010	non	oui	non	$\{b\}$
3	011	non	oui	oui	$\{b,c\}$
4	100	oui	non	non	$\{a\}$
5	101	oui	non	oui	$\{a,c\}$
6	110	oui	oui	non	$\{a,b\}$
7	111	oui	oui	oui	$\{a,b,c\}$

Le professeur YAKA-FAUCON ne voit pas le problème parce qu'il ne prend pas en compte ce qu'il est convenu d'appeler l'**explosion combinatoire**, c'est à dire le nombre de cas à traiter.

Prenons l'exemple de l'énumération des parties d'un **ensemble à 100 éléments.** Il y a 2^{100} parties à lister soit environ $1{,}26765060022823\times10^{30}$ parties. Avec un ordinateur très rapide capable de lister un milliard (10^9) de parties par seconde et compte tenu qu'il y a environ[6] 31622400 secondes dans une année, il faudrait autour de $40087109145043{,}7$ années pour l'exécuter c'est à dire **un peu plus de 40 000 milliards d'années** !

La difficulté de ces problèmes tiens précisément dans cette explosion combinatoire : le nombre de cas à traiter rend le problème insoluble en pratique.
Il n'est donc pas opportun de vouloir énumérer toutes les parties d'un ensemble de grande taille! ;=)

2.2. Deux classes de problèmes : la classe P et la classe NP

Du fait de cette explosion combinatoire, il y a lieu de savoir si les problèmes de grande taille dont on confie le calcul à nos ordinateurs modernes s'exécutent en un temps raisonnable ou pas. De cela émergent deux classes de problèmes, la classe **P** et la classe **NP.**

6 $3600\times24\times366=31622400$

Pour faire la différence, il convient de se souvenir que la résolution d'un problème passe par deux phases : une *première phase* de recherche des solutions potentielles et une d*euxième phase* de vérification que ces solutions potentielles conviennent.
La classe P : un problème est dit de classe **P** lorsque les deux phases de la résolution peuvent être réalisées par une machine de Turing déterministe avec une complexité en temps polynomiale (c'est à dire en $O(n^k)$ où k est un nombre entier naturel).
La classe NP (Non-deterministic polynomial time[7]) : pour un problème de la classe **NP**, on fait en quelque sorte l'impasse sur la phase de recherche : on considère que cette recherche a été faite en temps polynomial par une machine de Turing **non déterministe** et que la contrainte de complexité polynomiale porte aussi sur la deuxième phase.

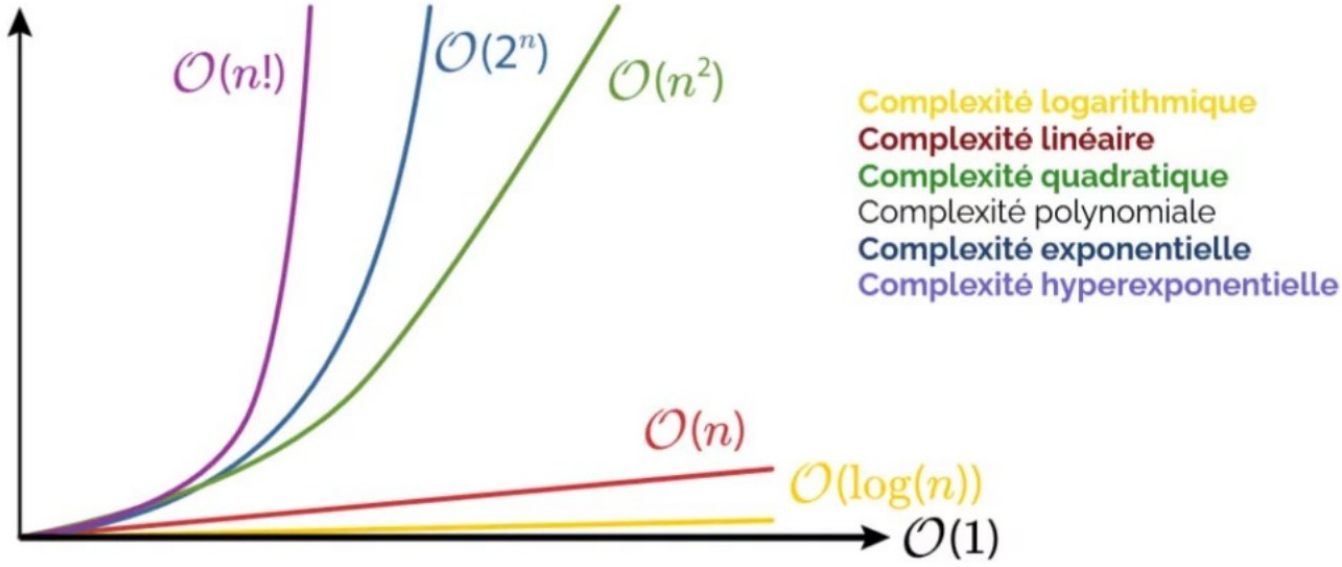
Vu que ce que fait une machine de Turing déterministe peut aussi être fait par une machine non-déterministe on a que $P \subseteq NP$. Les théoriciens de l'informatique rêvent de démontrer un jour que $P = NP$, D'ici là, cela demeure juste une conjecture.

De manière synthétique on peut dire que les problèmes de NP sont des problèmes difficiles à résoudre mais vérifiables rapidement.

Pourquoi une complexité polynomiale en $O(n^k)$ est-elle considérée comme raisonnable ?
Selon la loi de Moore, la puissance des semi-conducteurs double tous les deux ans, cette puissance évolue comme 2^p où p est le nombre de périodes de deux ans, il y a donc un espoir qu'au fil du temps 2^p dépasse $K \times n^k$. Tout cela est bien entendu théorique et traiter par exemple un problème en $O(n^{100})$ demeure un enfer.

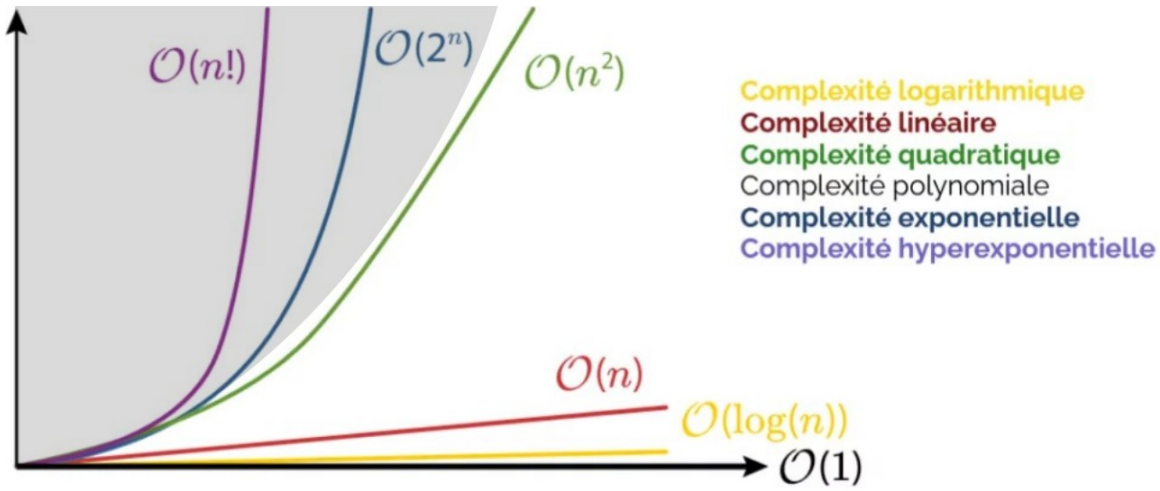
Une comparaison des complexités algorithmiques

La figure ci-dessous est très pertinente pour comparer des complexités algorithmiques. Elle est aussi trompeuse : elle laisse croire que la frontière entre les problèmes traitables et ceux qui ne le sont pas se situe entre une complexité linéaire ($O(n)$) et une complexité quadratique ($O(n^2)$).



7 **NP** est mis pour **Non-deterministic polynomial time ,** ce serait une erreur de croire que cela signifie *Non-Polynomial*.

En réalité, le gap se situerait plutôt entre complexité polynomiale ($O(n^k), k \in \mathbb{N}$) et complexité exponentielle ($O(2^n)$), voir ci-dessous en grisé.



2.3. Le problème SAT

Les formules de la logique booléenne sont construites à partir de variables booléennes (pouvant être soit vraies, soit fausses) et des connecteurs booléens "et" ($\wedge$), "ou" ($\vee$), "non" ($\neg$).
Une formule est **satisfiable** s'il existe une assignation des variables booléennes qui rend la formule logiquement vraie.

Le problème SAT part d'une formule de la logique booléenne, la question est de déterminer si cette formule est satisfiable. C'est un problème de décision.

Par exemple :

- La formule $(p \wedge q) \vee \neg p$ est satisfiable car si p prend la valeur *faux*, la formule est évaluée à *vrai*.
- La formule $p \wedge \neg p$ n'est pas satisfiable car aucune valeur de p ne peut rendre la formule vraie.

Le problème SAT peut se ramener à une formule *conjonctive*, faite de ~~termes~~ facteurs connectés avec $\wedge$, chaque ~~terme~~ facteur constitue alors une *clause* à satisfaire. Il est assez emblématique parmi les problèmes réputés difficiles en informatique et fait l'objet de nombreuses recherches.

Par exemple, la formule $(\neg p \vee q) \wedge (p \vee r) \wedge (\neg q \vee \neg r)$ revient à satisfaire 3 clauses :

- Clause 1 : $\neg p \vee q$
- Clause 2 : $p \vee r$
- Clause 3 : $\neg q \vee \neg r$

Le Professeur YAKA-FAUCON disait à propos de ce problème de satisfiabilité à n variables :

« Je ne vois pas le problème, il y a 2^n instanciations possibles, il suffit de vérifier si l'une d'entre-elles satisfait la formule. »

On sait déjà à quoi s'en tenir !

2-SAT

Le problème 2-SAT est une restriction du problème SAT où chaque clause implique au plus 2 variables. C'est le cas par exemple de la formule $(\neg p \vee q) \wedge (p \vee r) \wedge (\neg q \vee \neg r)$ déjà évoquée avec ses 3 clauses :

- Clause 1 : $\neg p \vee q$
- Clause 2 : $p \vee r$
- Clause 3 : $\neg q \vee \neg r$

Résolution : Si *p* est *vrai*, la clause 2 est satisfaite et la clause 1 force *q* à *vrai*, et donc la clause 3 force *r* à *faux*. Ce qui satisfait la formule.

Il a été démontré que 2-SAT est dans **P**.

3-SAT

Le problème 3-SAT est en apparence une restriction du problème général SAT, chaque clause du problème 3-SAT implique au plus 3 variables. Il a été démontré que 3-SAT est dans **NP**. Il a été aussi démontré que 3-SAT est aussi général que SAT, en ce sens que résoudre 3-SAT revient à résoudre SAT.

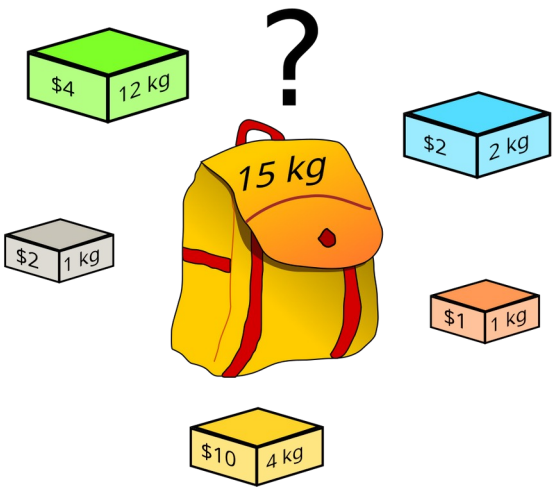
2.4. Le problème du sac à dos[8]

Le problème du sac à dos[9] est le suivant :
On dispose d'un sac à dos de capacité limitée en poids et d'un certain nombre de boites.
Chaque boite a un *poids* et une *valeur.*
Le problème est de charger le sac à dos avec les boites, sans dépasser la capacité du sac afin d'avoir un chargement de la plus grande *valeur* possible.

Évidemment on se place dans la situation où :

- Aucune boite n'est trop lourde pour entrer dans le sac
- Le poids cumulé des boites dépasse la capacité du sac
- Toutes les boites sont à *valeur* positive.

C'est un problème d'optimisation. Il est dans **NP**.



Le problème du sac à dos : quelles boîtes choisir afin de maximiser la somme emportée tout en ne dépassant pas les 15 kg autorisés ?

8 Image : CC BY-SA 2.5, https://commons.wikimedia.org/w/index.php?curid=985491
9 *Knapsack Problem* en anglais. Voir : https://fr.wikipedia.org/wiki/Problème_du_sac_à_dos

Le professeur YAKA-FAUCON s'est intéressé au problème du sac à dos à *n* boites :

> *« Je ne vois pas le problème, il suffit d'examiner un à un tous les chargements possibles et de ne retenir que le meilleur ! »*

Complexité de cette solution : Pour un chargement, on choisit d'abord la première boite (*n* choix possibles) puis la deuxième (*n-1* choix possibles), puis la troisième (*n-2* choix possibles) et ainsi de suite …
On arrive à un nombre de chargements de l'ordre de *n!* Même si on réussit à en éliminer quelques-uns au passage, cela reste marginal, on est en $O(n!)$.

Les algorithmes de chiffrement asymétrique fondés sur le problème du sac à dos ont tous été cassés à ce jour.
En pratique des heuristiques sont utilisées pour des situations en lien avec ce problème comme : optimiser la gestion des stocks, le chargement d'un porte-containers, la découpe de matériaux, …

2.5. Le problème du voyageur de commerce[10]

Le **problème du voyageur de commerce**, ou problème du commis voyageur est un problème de NP. C'est un problème d'optimisation qui consiste à déterminer, étant donné un ensemble de villes, le circuit le plus court passant par **chaque ville une seule fois**. Évidemment on connaît les distances entre les villes.

C'est un problème algorithmique célèbre, qui a donné lieu à de nombreuses recherches et qui est souvent utilisé comme introduction à l'algorithmique ou à la théorie de la complexité. Il présente de nombreuses applications, que ce soit en planification, en logistique ou dans des domaines éloignés, comme la génétique, les gènes étant les villes et la similarité la distance.

Écoutons le professeur YAKA-FAUCON sur le problème du voyageur de commerce à n villes :

> *« Je ne vois pas où est le problème, il suffit d'examiner tous les circuits possibles et d'exhiber le meilleur ! »*

Complexité : Pour construire un circuit on choisit d'abord la ville de départ (*n* choix possibles) puis la première étape (*n-1* valeurs possibles) puis la deuxième (*n-2* choix) et ainsi de suite … Ce qui conduit à $O(n!)$. Bien sûr, chaque circuit apparaît plusieurs (*2n*) fois : chaque ville peut en être un point de départ et il y a deux sens de parcours. On peut donc diviser ce nombre par *2n*.
On arrive à $\frac{(n-1)!}{2}$ circuits, ce qui ne change pas fondamentalement la complexité.
Des travaux en programmation dynamique ont fait *tomber* la complexité en $O(n^2 2^n)$.

3. Problèmes NP-complets

La notion vient du besoin d'exhiber, parmi les problèmes de NP, ceux qui seraient « les plus difficiles ».

10 https://fr.wikipedia.org/wiki/Problème_du_voyageur_de_commerce

3.1. Que faut-il entendre par « problème plus difficile qu'un autre. » ?

Considérons 2 problèmes simples. Le premier, disons *M*, est le problème de la *multiplication de deux nombres entiers :* étant donnés deux nombres entiers *a* et *b*, il s'agit d'en calculer le produit $a \times b$. Le deuxième problème, que nous appellerons *C* ou encore *élévation au carré* prend un nombre entier *x* et calcule son carré c'est à dire le produit de *x* par lui-même.
Ces deux problèmes font référence à des opérations que l'on considère comme élémentaires, ils sont dans P. La question n'est donc pas celle de leur complexité mais de savoir si l'un est *plus difficile* que l'autre.
Une réponse que l'on peut y apporter est la suivante :
toute instance *x* du problème *C* peut être résolue par le problème *M*, il suffit de calculer $x \times x$, en d'autres termes le problème *M* permet de résoudre le problème *C*. De ce point de vue, c'est donc que le problème *M* est au moins aussi difficile que le problème *C*[11].

3.2. Réduction polynomiale et problèmes NP-complets.

3.2.1. Réduction polynomiale

La réduction polynomiale reprend cette idée : *"Étant donnés deux problèmes A et B. Si je peux transformer B en A sans trop d'effort et si je peux résoudre rapidement le Problème A, alors je peux aussi résoudre rapidement le Problème B".*
Dans ce cas on dit que le problème *B* se réduit au problème *A.*

- **"Sans trop d'effort"** signifie que la transformation se fait en un temps raisonnable (complexité polynomiale).
- **Exemple concret** : Si on dispose d'une machine qui classe des objets par couleur, et que l'on veux classer des objets par leur valeur, on peut d'abord les colorier en fonction de leur valeur, puis utiliser la machine. La coloration est la "réduction".

3.2.2. Problèmes NP-complets

3.2.2.1. Problèmes de décision et problèmes d'optimisation.

Précisons en premier lieu ce qu'est un **problème de décision** : un problème de décision est un problème à réponse booléenne, c'est à dire que la réponse à un tel problème est *Vrai* ou *Faux*, *Oui* ou *Non.*
Le problème *SAT* est un exemple de problème de décision puisqu'il s'agit de dire si une formule est satisfiable ou pas.
Contrairement aux problèmes de décision, pour lesquels il n'existe qu'une seule bonne réponse pour chaque entrée, les problèmes d'optimisation visent à trouver la *meilleure* réponse à une entrée

11 **Remarque** : étant donnés deux nombres entiers *a* et *b*, si on calcule $(a+b)^2-(a-b)^2$ on trouve *4ab.* En d'autres termes $a \times b = \frac{(a+b)^2-(a-b)^2}{4}$. C'est à dire que l'on peut calculer le produit à partir d'opérations arithmétiques élémentaires (la division par 4 revient à un décalage de deux chiffres) et d'élévations au carré. Autrement dit on peut résoudre le problème *M* à l'aide du problème *C*. C'est donc que le problème *C* est au moins aussi difficile que le problème *M*. En fait les deux problèmes sont aussi difficiles l'un que l'autre.
Tout ceci n'est que théorique, personne ne s'aventurerait à calculer $\frac{(a+b)^2-(a-b)^2}{4}$ pour multiplier *a* par *b* sauf à disposer d'une machine très performante pour élever des nombres au carré...

donnée. Par exemple, le problème du sac à dos ou le problème du voyageur de commerce, tels qu'ils sont formulés ne sont pas des problèmes de décision mais des problèmes d'optimisation : il s'agit de trouver une solution qui optimise une certaine valeur (valeur du sac à dos pour le premier ou longueur du circuit pour le deuxième).

3.2.2.2. Problèmes NP-complets

Un **problème NP-complet** est un problème de décision vérifiant les propriétés suivantes :

- Le problème est dans NP .
- **Tous** les problèmes de la classe NP s'y ramènent via une réduction polynomiale.

Cela signifie qu'un tel problème est d'une part, au moins aussi difficile que tous les autres problèmes de la classe NP, on retrouve ainsi l'idée d'exhiber les problèmes les plus difficiles de la classe NP. D'autre part, résoudre efficacement un tel problème résoudrait, en même temps, tous les problèmes de la classe NP.

On comprend bien que trouver une méthode rapide pour résoudre un problème NP-complet serait une révolution et que cela intéresse nombre de chercheurs en informatique. Cela résoudrait également la question, encore ouverte de nos jours, de savoir si P=NP (un prix d'un million de dollars y est d'ailleurs consacré, via le prix Clay Millennium !).

SAT est le premier problème identifié comme NP-complet[12], depuis de nombreux autres problèmes ont aussi été identifiés comme NP-complets[13].

3.2.2.3. Et les problèmes d'optimisation ?

Au prix d'une reformulation, nombre de problèmes d'optimisation peuvent se transformer en une suite de problèmes de décision.

Prenons l'exemple du problème du voyageur de commerce qui, rappelons-le, consiste à :

Trouver le circuit le plus court qui passe une fois et une seule par n *villes.*

Il peut se reformuler en :

Pour chaque longueur de circuit possible, disons l,

existe-t-il un circuit de cette longueur l *qui passe une fois et une seule par chacune des* n *villes ?*

Sous cette forme il se ramène à une suite de problèmes de décision.

Plus précisément : les circuits sont de longueur *n* et donc les longueurs *l* de circuit possibles sont majorées en proportion de n^{14}, c'est donc un *O(n)*.

Cela revient à dire que le problème d'optimisation se transforme en *O(n)* problèmes de décision.

Au prix d'un abus de langage certains problèmes d'optimisation seront aussi considérés comme NP-complets alors qu'en principe, cela n'est réservé qu'à des problèmes de décision.

C'est en particulier le cas des problèmes évoqués plus haut que sont le problème du sac à dos et celui du voyageur de commerce.

---

12 C'est le Théorème de Cook (1971)

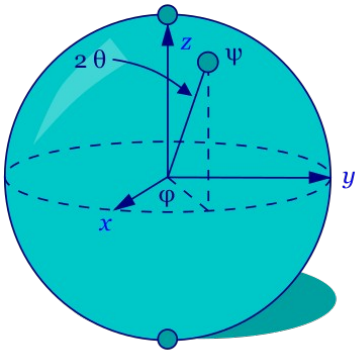
13 Pour démontrer qu'un problème est NP-complet, il suffit de trouver un problème connu pour être NP-complet qui s'y réduit de manière polynomiale.

14 La longueur d'un circuit est au plus $L \times n$, où *L* est la plus grande des distances possibles entre deux villes.

4. Quelle place pour les ordinateurs quantiques ?

En informatique quantique, un **qubit** est un système quantique à deux niveaux, qui représente la plus petite unité de stockage d'information quantique. Ces deux niveaux, notés $|0\rangle$ et $|1\rangle$ représentent chacun un état de base du qubit et en font donc l'analogue quantique du bit.

Grâce à la propriété de superposition quantique, un qubit stocke une information qualitativement différente de celle d'un bit. D'un point de vue quantitatif, un qubit peut être dans une multitude d'états combinaison linéaire de la forme $\alpha|0\rangle+\beta|1\rangle$ avec α et β, des nombres complexes vérifiant : $|\alpha|^2+|\beta|^2=1$. Tout se passe comme si les états quantiques d'un qubit se trouvaient à la surface d'une sphère de rayon 1



Il se réduira à un seul bit d'information au moment de sa mesure, $|\alpha|^2 et |\beta|^2$ sont les probabilités d'obtenir respectivement $|0\rangle$ ou $|1\rangle$.

La question se pose de savoir quels types de machines de Turing sont les ordinateurs quantiques ?
Ne sont-ce peut-être que des machines de Turing ***non-déterministes*** *capables de résoudre de manière polynomiale les phases de recherches de nos problèmes de* ***NP*** *?*

Autre question : pour un fonctionnement optimal un qubit a besoin d'un vide quasi absolu et d'une température proche du zéro absolu.
À quand donc l'idée d'envoyer un ordinateur quantique dans l'espace, où ces conditions sont plus facilement réunies ?
Qui sait, l'idée a peut-être déjà germé dans l'esprit de nos dirigeants ?