Université de Lille
FST- Département Informatique

ACT- Cours

2018–2019
Master 1 d'Informatique
S.Tison

Complexité des problèmes : Notions élémentaires
Les classes P et NP

1 Le cadre général

1.1 *La problématique*

L'étude de la complexité des problèmes - computational complexity- s'attache à déterminer la complexité intrinsèque d'un problème et à classifier les problèmes selon celle-ci, donc à répondre à des questions comme :

- Quelle est la complexité minimale d'un algorithme résolvant tel problème ?
- Comment peut-on dire qu'un algorithme est optimal (en complexité) ?
- Existe-t-il un algorithme polynomial pour résoudre un problème donné ?
- Comment peut-on montrer qu'il n'existe pas d'algorithme polynomial pour un problème ?
- Qu'est-ce qu'un problème "dur" ?
- Comment prouver qu'un problème est au moins aussi "dur" qu'un autre ?

1.2 *La définition*

Définition. *La complexité d'un problème est la complexité minimale dans le pire des cas d'un algorithme qui le résout - c'est souvent la complexité en temps qu'on considère mais on peut s'intéresser à d'autres mesures comme par exemple la complexité en espace-* .

Attention : Cette définition -un peu floue- ne précise pas quel modèle d'algorithme on choisit : l'analyse de la complexité risque d'être différente si on programme en C ou "en Machine de Turing". On reviendra sur ce point et sur l'avantage de définir des classes de complexité indépendantes du modèle classique choisi.

1.3 *Comment calculer la complexité d'un problème ?*

Calculer la complexité d'un problème est ardu en général. On se contente souvent d'encadrer :

- pour trouver une borne supérieure, il suffit de trouver **UN** algorithme. Bien sûr, si l'algorithme n'est pas très performant, la borne trouvée ne sera pas de très bonne qualité.
- trouver une "bonne" borne inférieure s'avère souvent difficile. Par exemple, montrer qu'un problème est de complexité au moins exponentielle, revient à montrer que **TOUT** algorithme le résolvant est exponentiel.

Quand les deux bornes coïncident, c'est l'idéal mais c'est assez rare ! Par exemple, on conjecture que la multiplication de deux entiers de longueur n peut se faire en $O(n \log\ n)$ mais cela n'a pas été montré -à ma connaissance- ; le meilleur algorithme connu est en $O(n \log n \log\log n)$. La meilleure borne inférieure trouvée est $O(n)$.

Citons trois méthodes utilisées pour trouver une borne inférieure de la complexité d'un problème :

- Les méthodes dites d'**oracle** ou d'adversaire : on suppose qu'il existe un algorithme utilisant moins d'un certain nombre d'opérations d'un certain type ; on construit alors une donnée qui met en défaut l'algorithme.
  Exemple. *Montrer ainsi que rechercher le minimum dans une liste de n éléments nécessite $n-1$ comparaisons.*
- Les **arbres de décision** : ils sont utilisés pour les algorithmes de type recherche ou tri "par comparaisons". Dans ce cas, on suppose que seules des comparaisons entre les éléments sont utilisées pour obtenir de l'information sur l'ordre ou l'égalité des éléments. Un arbre de décision "représente" alors toutes les comparaisons exécutées par l'algorithme sur les données d'une certaine taille. Un noeud correspond à une comparaison, ses fils aux différentes réponses possibles de la comparaison (donc si le test est à valeur booléenne, les noeuds sont binaires) ; une exécution possible correspond donc à une branche, et deux données correspondant à la même branche correspondront à la même suite d'instructions.
  La hauteur minimale d'un arbre de décision donne la complexité minimale -en nombre de comparaisons- dans le pire des cas : tout algorithme résolvant le problème par comparaisons a une complexité dans le pire des cas au moins égale à cette quantité. On peut aussi en déduire des résultats en moyenne en utilisant des résultats sur la hauteur moyenne de l'arbre.
  Rappel : la hauteur d'un arbre binaire (resp $k-$aire, i.e. tel que chaque noeud a au plus k fils)

qui a n feuilles est supérieure ou égale à $\log_2(n)$ (resp. $\log_k(n)$).
Exemple. *On cherche à deviner un nombre entier de 1 à 16. On a le droit de poser des questions* $x <= c$ *pour tout* c *de* 1 *à* 16. *Quelle stratégie adopter pour poser le moins de questions dans le pire des cas ? On peut représenter l'arbre des exécutions comme un arbre de décision binaire avec au moins* 16 *feuilles : l'arbre est donc au moins de hauteur 4. Quelle que soit la stratégie, on devra poser au moins 4 questions dans le pire des cas.*
Exemple. *Montrer que tout tri par comparaisons de* n *éléments effectue au moins de l'ordre de* $n \log n$ *comparaisons.*
— **Les réductions :** le principe est simple. Supposons par exemple qu'on sache que le problème $DURDUR$ soit de complexité exponentielle, donc tout algorithme le résolvant est au moins exponentiel. Si on arrive à réduire d'une façon "peu coûteuse" le problème $DURDUR$ dans le problème $MonPb$, le problème $MonPb$ sera lui aussi de complexité au moins exponentielle. Il faut bien sûr définir correctement ce qu'est une réduction "peu coûteuse" ; nous reviendrons sur cette notion importante lors de l'étude de la classe NP.

2 Les classes de Problèmes

L'idée va être de classifier les problèmes selon leur complexité ; tout d'abord pour simplifier, on va se limiter aux problèmes de décision :

2.1 Les Propriétés

2.1.1 *Oui ou Non ?*

On va se restreindre à des problèmes de décision (la sortie est de type OUI/NON), donc à vérifier des propriétés : une propriété est juste une fonction à valeurs booléennes. Un problème de décision peut donc être vu comme l'ensemble ($\mathcal{I}$) d'instances du problème avec pour chaque instance une réponse “oui” -on parle d'instances positives- ou “non” -on parle d'instances négatives-.

2.1.2 *Abstrait/Concret :*

A une propriété abstraite définie sur un ensemble quelconque, on peut associer une propriété concrète définie sur des mots -la représentation des instances- : par exemple à la propriété abstraite "être premier" définie sur les entiers, on associera la propriété "être la représentation binaire d'un entier premier" définie sur $\{0, 1\}^*$.

2.1.3 *Propriété/langage :*

On peut donc ensuite associer à la propriété le langage des (représentations des) données vérifiant cette propriété. Par exemple, à la propriété "être premier", on associera l'ensemble des représentations des entiers premiers dans une certaine base non unaire. Donc, vérifier si un entier est premier se ramène à tester si sa représentation (=un mot) est dans le langage. De la même façon, à la propriété "être 3-coloriable", on associera les représentations des graphes 3-coloriables, selon un codage donné.

Décider une propriété se ramène donc à tester l'appartenance d'un mot à un langage.

2.2 Un exemple de classe de problèmes : la classe P, celle des problèmes "praticables"

Définition. *La classe* P *(ou* $PTIME$*) est la classe des problèmes de décision pour lesquels il existe un algorithme de résolution polynomial en temps.*

Cette définition est indépendante du modèle (séquentiel) d'algorithme choisi : un algorithme polynomial en C sera polynomial si il est traduit en termes de machine de Turing et les modèles classiques de calcul (excepté les ordinateurs quantiques) sont polynomialement équivalents. Par convention, un problème de décision est dit **praticable** si il est dans P, impraticable sinon ; d'où on peut remplacer la question "existe-t-il un algorithme praticable pour ce problème" par "ce problème est-il dans P" ("être ou ne pas être dans P", voilà la question !). Cette convention vient du fait qu'un algorithme est dit praticable Ssi il est polynomial ; elle a quelques défauts, en particulier un algorithme polynomial de degré élevé n'est guère praticable....

Quelques exemples de problèmes qui ont été prouvés impraticables, donc ne pas être dans P *:* le problème des échecs généralisés (i.e. la taille de l'échiquier est non fixée, on a une configuration, on cherche à savoir si elle est gagnante) a été montré "réellement" exponentiel, i.e. tout algorithme le résolvant est exponentiel ; dans un autre domaine, il a été prouvé que tout algorithme décidant si une expression rationnelle étendue (avec le complémentaire en plus de l'union, la concaténation et l'étoile) représente tous les mots est non élémentaire, i.e. on ne peut pas borner la hauteur d'exponentielle des algorithmes résolvant ce problème !

2.3 Quelques autres exemples de classes

— $PSPACE$: la classe des problèmes de décision pour lesquels il existe un algorithme de résolution polynomial en espace. Bien sûr $PTIME$

est inclus dans $PSPACE$: un algorithme polynomial en temps "consomme au plus un espace polynomial".
— $EXPTIME$: la classe des problèmes de décision pour lesquels il existe un algorithme de résolution exponentiel en temps.
— NP : voir section 3

Exemple. *Expliquer pourquoi $PSPACE$ est inclus dans $EXPTIME$.*

3 La classe NP

La classe NP contient P et est contenue dans *ExpTime* (et même *Pspace*). Elle contient beaucoup de propriétés associées à des problèmes courants : problèmes de tournées, d'emploi du temps, de placement de tâches, d'affectation de fréquences, de rotation d'équipages, de découpage...

Or pour beaucoup d'entre elles, on n'a pas trouvé d'algorithme polynomial, mais pour **aucune d'entre elles**, on n'a prouvé qu'elle ne pouvait pas être testée en temps polynomial. En fait, on suppose que $P \neq NP$ mais personne n'a su prouver jusque là cette conjecture émise en 1971 ![1]. Elle a été déclarée un des problèmes du millénaire par l'Institut Clay qui propose 1 million de dollars pour sa résolution !

Attention : NP veut dire Non-déterministe Polynomial (et non pas Non Polynomial !).

3.1 *La définition de NP via les certificats :*

Comme vu précédemment, une propriété peut être associée au langage L des représentations de ses instances positives. On définit donc la notion de NP sur les langages :

Définition. *L est dit NP si il existe un polynôme Q, et un algorithme polynomial à deux entrées et à valeurs booléennes tels que :*

$$L = \{u / \exists c, A(c, u) = Vrai, |c| \leq Q(|u|)\}$$

A est appelé algorithme de vérification, c certificat (ou preuve, ou témoin..). ($|c|$ représente la taille de c).

On dit que A vérifie L en temps polynomial.

On peut par exemple voir c comme une preuve, A comme un algorithme qui vérifie la preuve ; vous pouvez être capable de vérifier facilement la preuve courte qu'un gentil génie vous donne mais cela n'implique pas forcément qu'elle soit facile à trouver...

Une propriété NP sera donc une propriété pour laquelle les instances positives ont une preuve "courte" et "facile" à vérifier.

3.1.1 *Exemples*

— Soit la propriété "être satisfiable" pour une expression booléenne. Un certificat est juste une valuation (i.e. une valeur booléenne pour chaque variable). Il est valide si la valuation donne la valeur vraie à la formule. Il est bien de taille polynomiale (même linéaire), puisque c'est juste un vecteur de n booléens, si n variables sont présentes dans la formule ; vérifier qu'il est valide correspond à évaluer l'expression avec cette valuation : c'est aussi polynomial. La propriété est bien NP.
— Soit la propriété "être 3-coloriable " ; un certificat pour G est juste un coloriage des noeuds. Il est valide si aucun arc ne relie deux noeuds de même couleur.
La taille d'un certificat est bien linéaire par rapport à celle de la donnée et vérifier qu'il est valide est bien polynomial : "Etre 3-coloriable" est donc bien une propriété NP.
— Soit la propriété de graphe : "avoir un chemin sans cycle de longueur au moins k " ; un certificat pour G est juste une suite de noeuds distincts. Il est valide si il définit bien un chemin sans cycle de longueur au moins k. La taille d'un certificat est bien polynomiale par rapport à celle de la donnée et vérifier qu'il est valide est bien polynomial : la propriété est NP.
— Soit la propriété "être composé (i.e. non premier)" ; un certificat pour n peut être un couple $(p, q), p \geq 2, q \geq 2$ tel que $p \leq n, q \leq n$; il sera valide si $np = q$. (Remarque : il y a donc plusieurs certificats "valides" possibles !). La taille d'un certificat est linéaire par rapport à celle de la donnée et vérifier que $n = pq$ est polynomial. "Etre composé" est donc bien une propriété NP. Elle est même P, comme cela été montré en 2002 (algorithme AKS) !

3.2 *La définition via le non-déterminisme*

On peut définir les propriétés NP en utilisant la notion d'algorithme NON-déterministe polynomial :

Un algorithme non-déterministe peut être vu comme un algorithme avec des instructions de type "choix(i,1..n)" : on choisit aléatoirement un entier dans

1. Même si $P \neq NP$ reste de loin l'opinion majoritaire, l'unanimité n'est pas totale. Dans un sondage effectué en 2002 par Bill Gasarch, sur 79 chercheurs de grande renommée, 5 pensaient que le problème serait résolu avant 2009, plus de la moitié pensait qu'il ne le serait pas avant 2040. Mais 9 pensaient possible que la conjecture soit résolue en prouvant que $NP = P$.

2. On peut prendre comme modèle de calcul non-déterministe, les Machines de Turing non déterministes : on étudiera ce modèle dans les derniers cours.

l'intervalle $[1..n]$. (On peut se restreindre à juste choisir une valeur parmi deux.) [2]

Soit A un algorithme non déterministe à valeurs booléennes et dont tous les calculs s'arrêttent ; il décide la propriété Pr suivante : "u vérifie Pr Ssi il existe un calcul de A sur u qui retourne Vrai." (penser à un automate non déterministe : un mot est accepté si et seulement si il existe au moins un chemin acceptant.)

Complexité d'un algorithme non-déterministe : Un algorithme non-déterministe A est dit polynomial si il existe un polynôme Q tel que pour toute entrée u, **tous** les calculs de A sur u ont une longueur d'exécution bornée par $Q(|u|)$.

Définition Alternative. *Une propriété Pr est NP si il existe un algorithme non déterministe polynomial qui décide Pr.*

Les deux définitions sont équivalentes : un certificat correspond à une suite de choix dans l'exécution de l'algorithme non déterministe.

A partir d'un algorithme non-déterministe polynomial pour vérifier P, on construira donc la notion de certificat qui correspond à une suite de choix et d'algorithme de vérification qui consiste à vérifier que l'exécution de l'algorithme non déterministe correspondant à la suite de choix donnée par le certificat retourne Vrai.

A partir d'une notion de certificat et d'algorithme de vérification, on construit un algorithme non-déterministe qui consiste à d'abord générer aléatoirement un certificat -la partie non déterministe- et ensuite à le vérifier en utilisant l'algorithme -déterministe- de vérification.

3.3 *La problématique NP=P*

Bien sûr, P est inclus dans NP : toute propriété P est une propriété NP ; l'algorithme de vérification est l'algorithme de décision et n'a pas besoin de certificat : on peut prendre pour certificat le mot vide.

On peut aussi montrer que NP est inclus dans $PSPACE$ et donc dans $EXPTIME$: pensez à l'algorithme qui énumère et teste tous les certificats possibles.

On conjecture que $P \neq NP$, mais personne n'a su le prouver ! Aucune propriété NP n'a été prouvée à ce jour ne pas être P. D'un autre côté, pour beaucoup de propriétés NP, aucun algorithme polynômial n'a été trouvé (on verra dans le prochain cours qu'il existe des propriétés NP telles que si on trouvait un algorithme polynomial pour l'une d'entre elles, il y aurait un algorithme polynomial pour n'importe quelle propriété NP). La conjecture a été émise par S. Cook en 1971 et le "Clay Mathematics Institute" offre 1 million de dollars à celui qui trouve la réponse à la question ! De même on conjecture que NP est strictement inclus dans $Pspace$ sans avoir réussi à la prouver !

3.4 *NP et co-NP :*

Si une propriété Q est NP, cela n'implique à priori pas que *non* Q soit NP ; en fait, on ne sait pas si NP est close par complémentaire.

Une propriété Q telle que *non* Q soit NP est dite co-NP. Bien sûr, co-NP contient P.

On conjecture aussi que $NP \neq co-NP$ mais, là encore ce n'est qu'une conjecture ! Bien sûr, si $NP = P$, on aurait NP=co-NP. Mais on pourrait avoir $NP = co - NP$ sans que NP soit égal à P !

Bref, ce qu'on sait c'est qu'on ne sait rien, ou presque !

4 Les propriétés NP-dures

Quelle est la problématique ? Supposons qu'on soit dans la situation suivante : on cherche un algorithme pour vérifier une propriété ; on a montré qu'elle est NP mais on ne trouve pas d'algorithme P. Si on arrive à prouver qu'il n' y en a pas, on prouve $P \neq NP$, ce qui résoudrait une conjecture sur laquelle se sont "acharnés" depuis 30 ans de nombreux chercheurs. Cela nous ferait gagner les 1 million de dollars offerts par l'Institut Clay pour résoudre un des sept problèmes du millénaire (http://www.claymath.org/prizeproblems/index.htm) ... mais risque donc d'être difficile. Mais on peut peut-être montrer qu'elle est NP-dure : intuitivement cela signifie qu'elle contient toute la difficulté de la classe NP, et que donc si on trouvait un algorithme polynomial pour cette propriété, on en aurait un pour toute propriété NP : cela justifie en quelque sorte de ne pas avoir trouvé d'algorithme polynomial.

4.1 Les réductions polynomiales

Intuitivement, la notion de réduction permet de traduire qu'un problème n'est pas "plus dur" qu'un autre. Si un premier problème se réduit en un second, le premier problème est (au moins) aussi facile que le second - si la réduction est "facile"-. On peut a priori utiliser la réduction de deux façons :

> pour montrer qu'un problème est dur : si le premier est réputé dur, le second l'est aussi ;
>
> pour montrer qu'un problème est facile : si le deuxième est facile, le premier l'est aussi.

C'est en fait surtout le premier raisonnement que l'on va utiliser. Formellement :

Définition. *Soient L et L' deux langages de Σ^*, correspondant à deux propriétés. Une* **réduction polynomiale** *de L dans L' est une application red calculable polynomiale de Σ^* dans Σ^* telle que $u \in L$ SSi $red(u) \in L'$. On note alors $L \leq_P L'$.*

En pratique, une réduction polynomiale sera donc une transformation d'un problème dans un autre. Par exemple, une réduction d'un problème d'emploi du temps en un problème de graphes sera une transformation qui associera à toute instance i du problème d'emploi du temps, une instance $red(i)$ du problème de graphes qui aura une solution Ssi l'instance i avait une solution. De plus, cette transformation doit être polynomiale, i.e. calculable par un algorithme polynomial (Remarque : on en déduit donc que la taille de $red(i)$ est bornée polynomialement par rapport à celle de i).

4.1.1 Comment montrer qu'une propriété se réduit polynomialement en une autre.

Soient par exemple les deux propriétés suivantes :

```
Clique
Entrée:
G=(S,A) --un graphe non orienté
k       -- un entier
Sortie:
Oui, Ssi G contient une clique de cardinal k.

Indépendant
Entrée:
G=(S,A) --un graphe non orienté
k       -- un entier
Sortie:
Oui, Ssi G contient un ensemble indépendant
de cardinal k.
```

On peut choisir comme réduction de Clique dans Indépendant l'application *red* qui à l'instance $(G = (S, A), k)$ associe $(G' = (S, A'), k)$ avec $A' = \{(x, y)/(x, y) \notin A \text{ et } (x, y) \notin A\}$.

Montrer que c'est bien une réduction polynomiale de Clique dans Independent consiste en :

— Montrer que c'est correct i.e., pour tout (G, k), $Clique(G, k)$ Ssi $Independent(G', k)$.

— Montrer que la transformation est polynomiale.

En résumé : pour prouver qu'une propriété $P1$ se réduit polynomialement en une autre $P2$, il faut :

. proposer une réduction, i.e. un algorithme *red* qui à une instance I de $P1$, associe une instance $red(I)$ de $P2$.

. prouver qu'elle est correcte, i.e. qu'une instance I de P1 est positive **Si et Seulement Si** $red(I)$ est une instance positive de $P2$.

. prouver qu'elle est polynomiale !

4.1.2 Pourquoi prouver qu'un problème se réduit en un autre ?

Proposition : Si L' est dans P et si $L \leq_P L'$, alors L est dans P.

En effet, pour décider de l'appartenance à L, il suffit d'appliquer la transformation, et de décider de l'appartenance du transformé à L'.

Donc si L' est P, L est P ; si L n'est pas P, L' n'est pas P non plus.

Propriété : La relation $\leq_P$ est transitive : si $L \leq_P L'$ et $L' \leq_P L''$, alors $L \leq_P L$".

4.2 Les propriétés NP-dures, les propriétés NP-complètes

Définition. *Une propriété R est dite* **NP-dure** *Si toute propriété NP se réduit polynomialement en R ;*

Donc, d'après ce qui précède, si une propriété $NP - dure$ R était P, on aurait $NP = P$: R contient donc d'une certaine façon toute la difficulté de NP !

Définition. *Une propriété est dite $NP-$***complète** *Si elle est NP et $NP-dure$.*

Cook fut le premier à découvrir ... l'existence de propriétés NP-dures :

Théorème de Cook. $3 - CNF - SAT$ *est $NP-$complète.*

Rappel : $3 - CNF - SAT$ est le problème de la satisfiabilité d'une formule sous forme conjonctive où chaque clause contient 3 littéraux.

4.2.1 Comment montrer qu'une propriété est $NP-$dure ?

Pour montrer que R est NP-dure, plutôt que de montrer "directement" que toute propriété NP se réduit polynomialement en R, il suffit de montrer qu'une propriété connue $NP-$dure se réduit polynomialement en R. En effet, comme $\leq_P$ est transitif, on a :
Proposition : si $L \leq_P L'$ et L est $NP-$dure, alors L' est $NP-$dure.

4.2.2 Pourquoi montrer qu'une propriété est $NP-$dure ?

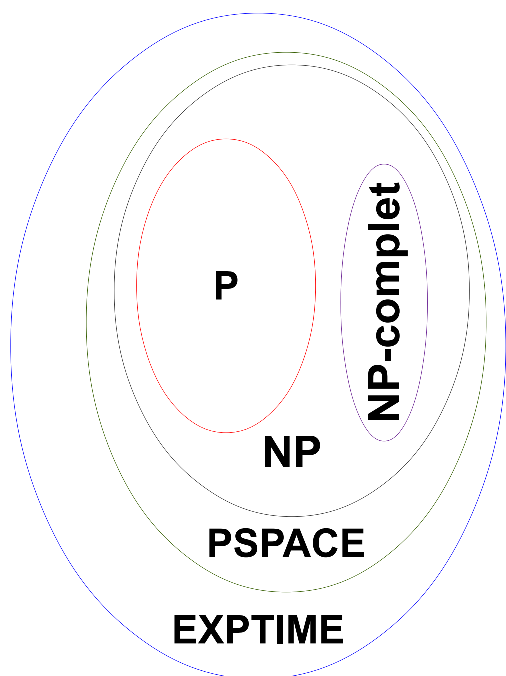
D'après la proposition précédente, si une propriété $NP-$dure se révélait être P, on aurait $P = NP$, ce qui est peu probable et en tout cas a résisté aux efforts de nombreux chercheurs ! Donc, montrer qu'une propriété est $NP-$dure justifie raisonnablement qu'on n'ait pas trouvé d'algorithme polynomial pour décider de cette propriété !

Remarque : par contre, montrer qu'une propriété est NP, c'est montrer qu'elle n'est pas "si dure", même si elle n'est peut-être pas P.

4.2.3 Hiérarchie des classes

On a $P \subseteq NP \subseteq PSpace \subseteq EXPTime$, mais, pour le moment, on ne sait ni si $P \neq NP$, ni si $NP \neq PSpace$, ni si $PSpace \neq EXPTime$. On sait par contre que P est **strictement inclus** dans EXP-Time.

Le schéma ci-dessous montre la hiérarchie des classes **en supposant que NP est différent de P**.



Hiéarchie en supposant NP différent de P

4.3 Problèmes NP-durs

Ce qui précède ne s'applique qu'aux propriétés. Parler de problèmes NP, si ils ne sont pas simplement des problèmes de décision, n'a guère de sens ; par contre on peut parler de problèmes $NP-$durs : si on peut montrer que l'existence d'un algorithme polynomial pour le problème R impliquerait $P = NP$, on peut dire que R est $NP-$dur.

En particulier, à tout problème d'optimisation de type "trouver une solution de coût minimal (resp. de gain maximal), on peut associer une propriété : " existe-t-il une solution de coût inférieur (resp. de gain supérieur) à une valeur donnée en entrée" ;

Si le problème de décision est $NP-$dur, le problème d'optimisation le sera aussi. Si on avait un algorithme P pour le problème d'optimisation, on en aurait un pour celui de décision donc pour toute propriété NP, puisqu'il est $NP-$dur !

Par exemple, soit le problème du coloriage de graphes : pour un graphe, trouver un coloriage correct utilisant le moins de couleurs possibles. On peut lui associer la propriété du $k-coloriage$: étant donné un graphe et un entier k, existe-t-il un coloriage correct de G avec k couleurs ? Cette propriété étant $NP-$dure, on peut dire que le problème de coloriage de graphe est $NP-$dur.

4.4 Un court extrait du catalogue "Les propriétés $NP-$complètes"

Sont $NP-$complètes les propriétés suivantes :

- 3-CNF-SAT : décider si une formule sous forme conjonctive avec au plus 3 littéraux par clause est satisfiable.
- 3-coloriage de graphes : existe-t-il un coloriage en 3 couleurs d'un graphe tel que deux sommets adjacents soient coloriés différemment.
- le probl`eme de l'existence d'une clique de taille k dans un graphe.
- le problème du BinPacking.
- Le problème du sac à dos (non fractionnable).
- le problème de l'existence d'un circuit hamiltonien.
- Le problème du voyageur de commerce.
- la programmation linéaire en entiers.
- Le problème du recouvrement d'ensembles.
- Le problème du démineur

Remarque 1 : ATTENTION à la spécification des problèmes : il suffit de changer à peine un problème pour qu'il ne soit plus $NP-$dur : par exemple, le 2-coloriage de graphes, le problème de l'existence d'un circuit eulérien, la programmation linéaire en réels sont dans P.

Remarque 2 : pour prouver qu'une propriété R est $NP-$dure, on peut donc choisir une propriété au catalogue des NP-dures et essayer de la réduire dans R : le problème est de bien choisir (bien sûr si elle est déjà dans le catalogue, c'est encore plus simple !...) i.e. de trouver une propriété connue NP-dure qui se réduit "facilement" et polynomialement en R.

Il existe un catalogue référence des propriétés $NP-$dures, le livre de Garey et Johnson :" Computers and Intractability : A guide to the theory of NP-completeness " qui contient beaucoup de problèmes mais date de 1979 ; il existe des sites Web qui essaient de tenir à jour le catalogue comme le compendium de problèmes d'optimisations NP.

$P = NP$ dans la "presse" ☺

Le problème $P = NP$ a été "popularisé". Il est aussi sous-jacent à un film (*The traveling Salesman*) ou un épisode de la série *Elementary.*

Voici la présentation "populaire" du problème $P = NP$ faite par l'Institut Clay (on trouve aussi sur le site une présentation officielle par S. Cook) au début du millénaire :

$P = NP$ par l'Institut Clay

Suppose that you are organizing housing accommodations for a group of four hundred university students. Space is limited and only one hundred of the students will receive places in the dormitory. To complicate matters, the Dean has provided you with a list of pairs of incompatible students, and requested that no pair from this list appear in your final choice. This is an example of what computer scientists call an NP-problem, since it is easy to check if a given choice of one hundred students proposed by a coworker is satisfactory (i.e., no pair from taken from your coworker's list also appears on the list from the Dean's office), however the task of generating such a list from scratch seems to be so hard as to be completely impractical. Indeed, the total number of ways of choosing one hundred students from the four hundred applicants is greater than the number of atoms in the known universe! Thus no future civilization could ever hope to build a supercomputer capable of solving the problem by brute force; that is, by checking every possible combination of 100 students. However, this apparent difficulty may only reflect the lack of ingenuity of your programmer. In fact, one of the outstanding problems in computer science is determining whether questions exist whose answer can be quickly checked, but which require an impossibly long time to solve by any direct procedure. Problems like the one listed above certainly seem to be of this kind, but so far no one has managed to prove that any of them really are so hard as they appear, i.e., that there really is no feasible way to generate an answer with the help of a computer. Stephen Cook and Leonid Levin formulated the P (i.e., easy to find) versus NP (i.e., easy to check) problem independently in 1971.

Pour aller plus loin, en septembre 2009, un article de Lance Fortnow sur le problème est paru dans "Communications of the ACM"(http://cacm.acm.org/magazines/2009/9/38904-the-status-of-the-p-versus-np-problem). Son succès a conduit à cet article dans le New York Times :

$P = NP$ dans le New York Times

The September cover article in the Communications of the Association of Computing Machinery touched off a distinct buzz last month when more than 10 times the usual number of readers downloaded the article in the first two days it went online.
The subject ? A survey of progress being made (not much, apparently) in solving the grand challenge for the fields of theoretical computer science and complexity theory. The problem is described rather opaquely as P vs. NP, and it has to do with real world tasks like optimizing the layout of transistors on a computer chip or cracking computer codes. Like earlier grand math challenges like Fermat's last theorem, there is a lot at stake, not the least of which is a $1 million cash prize that was offered for the solution as one of seven Mount Everest-style "Millennium Problems" the Clay Mathematics Institute offered almost a decade ago.
So far no one appears to be close to picking up a check. The challenge, in its simplest, but not most understandable phrasing, is to determine whether or not P equals NP. P stands for the class of problems that can be solved in polynomial time, or reasonably quickly. NP stands for the class of problems that can be verified in polynomial time - quickly. If it can be shown that P=NP, then it is possible that the world will be a very different place.
The issues center around a group of classic computer science problems, like the traveling salesman problem - how to figure out the most efficient route through a number of cities given certain constraints. Factoring a large number is also a good example of what is referred to as an NP problem. At present, there is no understood method for doing it in polynomial time, but given an answer it is easy to check to see if it is correct. The most efficient layout of transistors on a computer chip is another example of problems that are said to be NP.
The allure of the challenge, in addition to the fame and the money, is that if it is possible to prove that P does in fact equal NP some of the hardest computing challenges may collapse, leading to a burst of new economic and technological productivity. All of the tasks above, like the traveling salesman problem, are similar in that that as the problems grow in size - one more city for the salesman, one more transistor for the chip - the computing time required to solve them increases exponentially. In other words, one more city makes the problem 10 times harder.
Checking to see if any particular solution is correct is simple, but finding the best solution in the case of very large problems is infinitely difficult. "There are a lot of smart people who have tried to solve this problem and failed," said Patrick Lincoln, the director of the computer science laboratory at SRI International, a research group based in Menlo Park, Calif. "But they also failed to prove the problem can't be solved."
The editors of the journal said they were tickled to have a hit article on their hands. "I don't think we've ever had an article that started off with this kind of a bang," said Moshe Y. Vardi, a Rice University computer scientist who is editor in chief of the journal. "Our e-mail blast went out and thousands of people felt they had to click." The author of the article, Lance Fortnow, a computer scientist at Northwestern University McCormick School of Engineering, initially told Dr. Vardi that the article would be a short one. "Still open," he writes, was in first reaction to writing about the state of the work on solving the problem. There remains a glimmer of hope, he noted. An esoteric branch of mathematics known as algebraic geometry may offer an avenue to prove or disprove the problem, which was first outlined by Stephen A. Cook, a University of Toronto computer scientist, in 1971.That prospect feels a bit intimidating. As Dr. Vardi said, "It's a bit scary because we have to start learning a very difficult mathematical field."