P versus NP

Notes de cours[1] de complexité des problèmes, M1

Rappel : Un problème est un *problème de décision* quand les seules réponses possibles sont `vrai` ou `faux`.

1 La classe P

On commence par définir la classe **P**, qui est un ensemble de problèmes de décision.[2]
Définition : Un problème de décision est dans la classe **P** quand il existe un algorithme de complexité $\mathcal{O}(n^d)$, pour un certain $d > 0$, qui le résoud. Un problème qui est dans **P** est dit *polynomial*.

On dit que les problèmes de la classe **P** sont *faciles*. Ce n'est pas tout à fait vrai : si le meilleur algorithme connu est de complexité $\mathcal{O}(n^{100})$, c'est difficile de dire que l'algorithme est efficace. Il faut plutôt voir cela dans l'autre sens : si un problème de décision n'est pas dans **P**, alors il n'y a pas d'algorithme efficace pour le résoudre.

Vous connaissez beaucoup de problèmes qui sont dans **P**: tester si un tableau est trié, tester si un nombre est premier (vraiment? voir plus bas), ...

Formalisons un peu, on en aura besoin dans certains cas un peu subtils.[3] On considère que les entrées sont écrites en binaire, et sont donc des mots de l'alphabet $B = \{0, 1\}$. La taille n d'une entrée est donc la longueur de son encodage binaire. Si Q est un problème de décision, alors il y a les entrées (mots binaires) pour lesquelles il retourne `vrai` et celles pour lesquelles il retourne `faux`. On peut donc définir le langage des entrées pour lesquelles il retourne `vrai` :

$$\mathcal{L}(Q) = \{u \in B^* \mid Q \text{ est vrai pour } u\}$$

Le langage $\mathcal{L}(Q)$ caractérise complètement le problème Q. On dit alors qu'un *langage* $\mathcal{L}$ est dans **P** s'il existe un algorithme de complexité polynomiale en la longueur du mot d'entrée qui décide si le mot est dans $\mathcal{L}$ ou non.

Ce qu'il faut retenir de cette façon de définir **P** c'est que *la façon dont on encode l'entrée est* **importante**. Car c'est elle qui va définir la complexité du problème.

Prenons un exemple issu de la cryptographie. On veut savoir si n est un nombre premier ou pas. Il suffit de tester s'il est divisible par un nombre entre 2 et $n-2$ avec une boucle `for`[4] La complexité de cet algorithme est $\Theta(n)$, c'est rapide ! Sauf que ... si on considère que n est écrit en binaire, on peut le représenter par $m \approx \log_2 n$ bits. C'est ça la "vraie" taille de l'entrée. Et si on exprime la complexité en fonction de m, c'est $\Theta(2^m)$: la complexité de cet algorithme est **exponentielle en** m ! La cryptographie utilise ce genre d'argument, qu'on dit qu'on a une clé de taille 512, cela signifie que c'est un nombre écrit sur $m = 512$ bits; on peut donc considérer des nombres jusqu'à $2^{512} - 1$ ce qui est beaucoup trop grand pour faire une boucle simple.

Pour nous le problème ne se posera que pour les entiers, la plupart des autres structures ne "changent pas significativement" d'échelle quand on considère l'encodage en binaire (on peut passer de n à $n \log n$ ou même à n^2, mais cela reste dans des proportions polynomiales).

[1] Ces notes sont (fortement) inspirées des notes de cours de Jeff Erickson : `http://web.engr.illinois.edu/~jeffe/teaching/algorithms/notes/30-nphard.pdf`

[2] En fait, on étend sans problème la définition de **P** aux problèmes qui ne sont pas des problèmes de décision. Je ne le fais volontairement pas dans ce cours pour rester dans un cadre restreint qui facilite la présentation des autres notions.

[3] La formalisation réelle passe par les machines de Turing, ce que l'on va éviter dans ce cours.

[4] On peut même s'arrêter à $\sqrt{n}$ par un argument élémentaire.

2 Stabilité de la classe P

L'avantage technique de la classe **P** réside dans ses propriétés de stabilité.

Tout d'abord, elle est robuste par changement de modèle d'ordinateur. Nous n'avons pas défini ces modèles formellement, mais vous connaissez au moins la machine de Turing. Il existe des modèles différents, par exemple le modèle RAM[5], qui est un modèle qui ressemble plus à un véritable ordinateur. On peut utiliser le postulat suivant :[6]

Tous les modèles raisonnables d'ordinateurs définissent la même classe **P**.

Par exemple, si on a un algorithme en temps $\mathcal{O}(u_n)$ pour le modèle RAM, on peut montrer qu'il est toujours en $\mathcal{O}(u_n^2)$ pour les machines de Turing.

La deuxième stabilité est une conséquence de la stabilité des polynômes pour les opérations usuelles : si on utilise consécutivement deux algos polynomiaux, le résultat est polynomial; si on appelle un nombre polynomial de fois une fonction de complexité polynomiale, le résultat est de complexité polynomiale, etc.

3 Un exemple de problème (probablement) difficile : `SAT`

On considère le problème suivant : étant donné une formule logique avec des ET ($\wedge$), des OU ($\vee$) et des négations ($\overline{\cdot}$), est-ce qu'on peut affecter les variables de façon à ce que la formule soit vraie ?

Précisons le cadre. On considère des formules sous une forme particulière :

- On a des *variables* $\{x_1, \ldots, x_k\}$
- Les variables sont regroupées en *clauses*, une clause étant des OU portant sur des variables ou sur des négations de variables : $x_1 \vee x_3 \vee \overline{x_6}$ par exemple.
- Les clauses sont regroupées par des ET, ce qui forme la formule.

Par exemple, voilà une formule sous ce format :[7]

$$(x_1 \vee x_3 \vee \overline{x_6}) \wedge (x_2 \vee x_4 \vee x_5) \wedge (x_4 \vee \overline{x_3} \vee \overline{x_2} \vee x_1 \vee x_2)$$

Définition : Le problème `SAT` consiste à décider s'il existe au moins une façon d'affecter des valeurs `vrai`/`faux` aux variables de sorte que la formule s'évalue à `vrai`.

Il se trouve que ce problème est difficile, en un sens que nous allons bientôt expliciter. On pense (ce n'est pas prouvé) que ce problème n'est pas dans **P**, et qu'il n'admet donc pas d'algorithme polynomial.

On peut néanmoins faire une remarque importante : si on me donne une solution, c'est-à-dire quelles valeurs donner aux variables pour que la formule retourne `vrai`, on peut vérifier en temps polynomial que cette solution est correcte. Il suffit d'évaluer la formule pour cette valeur en calculant les $\vee$ et les $\wedge$.

[5] Ne pas confondre avec la RAM d'un ordinateur, cela n'a rien à voir.

[6] Pour être plus précis, il faudrait séparer le cas du modèle d'*ordinateur quantique* qui ne vérifie pas le postulat.

[7] Son nom est *forme normale conjonctive (FNC)*.

4 La classe NP et la classe co-NP

La classe **NP** est une classe de problème fondamentale en informatique. Sa définition formelle passe par les machines de Turing ... nous ne la donnerons pas.

L'idée intuitive d'une définition (équivalente) de la classe **NP** est la suivante : si on peut vérifier qu'une solution est valide en temps polynomial, alors le problème est dans **NP**. Précisons sur `SAT` : si une formule admet une affectation des variables qui lui fait retourner `vrai` alors on peut le confirmer en temps polynomial. Cette affectation de variables est difficile à trouver, mais *facile à vérifier*. C'est ainsi qu'on va définir la classe **NP**, l'affectation de variables s'appelle une *preuve* ou un *certificat*.

Remarque importante : on ne demande à pouvoir vérifier rapidement des certificats que pour les cas où la réponse au problème est `vrai`. On ne demande pas de certificats pour le cas où c'est `faux` (= aucune affectation ne marche dans le cas de `SAT`).

Notre définition, volontairement un peu imprécise, est la suivante :
Définition : Un problème de décision D est dans la classe **NP** quand pour toute entrée x pour laquelle $D(x)$ est `vrai`, il existe un certificat de x, de taille polynomiale en x, qui peut être vérifié en temps polynomial.

La classe **co-NP** suit la même définition que **NP**, mais on demande des certificats dans les cas où la réponse est `faux` au lieu de `vrai`.

On a bien sûr P $\subset$ NP, l'entrée est son propre certificat.

5 Question fondamentale de l'informatique

La question fondamentale est :

$$\boxed{\text{Est-ce que } \mathbf{P} \neq \mathbf{NP} \text{ ?}}$$

Autrement dit la question est : s'il est facile de vérifier qu'une solution est valide, est-il facile de *trouver* une solution ?

Cette question est citée comme l'un des 7 problèmes du millénaire et il y a 1 million de \$ pour celui qui le prouve (ou qui prouve que $\mathbf{P} = \mathbf{NP}$).

Pour l'instant, personne n'a réussi à montrer que $\mathbf{P} \neq \mathbf{NP}$, mais la plupart des chercheurs pensent que c'est le cas. On pense également que **NP** est différent de **co-NP**. On sait que les deux contiennent **P**.

6 Machines non-déterministes

Le **N** de **NP** signifie *non-déterministe*. La vraie définition de cette classe fait intervenir des machines de Turing non-déterministes, qui ressemble aux automates non-déterministes : il y a plusieurs façons d'effectuer les calculs (des branchements dans le chemin de l'automate), et on accepte si l'un d'eux est correct. Comme on ne définit pas les machines de Turing dans ce cours, on ne définit pas non plus leur version non-déterministe.

En revanche, on peut comprendre la notion en enrichissant un langage de programmation comme Java ou Python avec des affectations à choix : on autorise d'écrire `x=1 ou 2` pour dire que `x` prend l'une des deux valeurs. Du coup le programme a *plusieurs exécutions possibles !* Il faut alors redéfinir la notion de calcul réussi et de complexité. On le fait de la façon suivante :

- Une fonction booléenne (qui retourne `vrai` ou `faux`) qui possède des affectations à choix, retourne `vrai` s'**il existe** une façon de faire les choix qui permet de retourner `vrai`. Elle retourne donc `faux` seulement s'il n'est pas possible de retourner `vrai`.

- La complexité d'une fonction avec des affectations à choix est calculée en prenant la pire parmi tous les choix possibles.

Prenons comme exemple le programme suivant :

```
def f(n):
  s = 0
  for i in range(n):
    x = 1 ou 2
    s += x
  return s == 5
```

On peut voir que `f(2)` retourne `faux`, car même si on prend `x = 2` à chaque fois, on ne peut pas atteindre la valeur 5. En revanche, `f(3)` est vrai, en prenant par exemple successivement `x = 2`, `x = 1` et `x = 2` : il suffit d'une combinaison qui fonctionne pour que cela fonctionne.[8] La complexité ici est bien $\mathcal{O}(n)$, d'ailleurs elle ne dépend pas des choix effectués sur cet exemple.

7 Problèmes NP-complets et NP-difficiles

Un problème Π est dit **NP**-difficile quand trouver un algorithme polynomial pour ce problème impliquerait que $\mathbf{P} = \mathbf{NP}$. Autrement dit, on pourrait se servir de cet algorithme comme d'une boîte noire pour résoudre tous les problèmes **NP**.

Bien entendu, comme on pense fortement que $\mathbf{P} \neq \mathbf{NP}$, on pense fortement qu'un problème **NP**-difficile n'admet pas d'algorithme polynomial.

Un problème est **NP**-complet quand il est à la fois **NP**-difficile et dans **NP**. Ce sont les problèmes les plus difficiles de classe **NP**.

Pour la suite nous admettons le résultat fondamental suivant :

Théorème 1 (Cook) *Le problème `SAT` est* **NP***-complet.*

On peut quand même montrer la partie facile, que `SAT` est bien dans **NP** :

- La preuve par certificat consiste montrer que si on a deviné la bonne solution (les bonnes affectations des x_i), alors on peut le vérifier en temps polynomial. Il suffit de vérifier que chaque clause contient bien un littéral qui vaut `vrai`, ce qui se fait facilement quand on a affecté les x_i.

- La preuve par machine non-déterministe revient au même, et peut être faite par le programme suivant :

  ```
  def SAT(Phi,n):
    X = [None] * n
    for i in range(n):
      X[i] = True ou False
    return Phi(X)
  ```

On voit sur cet exemple que "deviner la solution" revient à "faire les bons choix non-déterministes". C'est ce qui est utilisé pour montrer que la notion de certificat caractérise bien les machines non-déterministes.

[8] C'est comme pour les automates non-déterministes, il suffit de trouver un chemin acceptant pour que le mot soit accepté, peu importe que d'autres chemins échouent.

8 Many-one réductions

Pour prouver que des problèmes sont **NP**-difficile, c'est à peu près comme prouver des bornes inférieures, où ce qu'on a fait sur `3SUM`, on va utiliser des arguments de réduction.

Formellement, la bonne notion de réduction pour les problèmes que l'on va considérer est la *réduction many-one*. C'est une réduction d'un problème Q à un problème R telle que

- le surcoût est au plus polynomial ;
- on n'utilise qu'une seule fois un algorithme pour R en boîte noire, et on répond exactement ce que répond cet algorithme (pas de traitement de la réponse de R pour trouver la réponse de Q, si R répond `vrai` on répond `vrai`, s'il répond `faux` on répond `faux`).

Le fait qu'on ne traite pas la réponse est rendu obligatoire car on a perdu la symétrie par complémentation dans la définition de **NP** : les réponses `vrai` ont des certificats, mais pas forcément les réponses fausses.

Voilà on a fini pour les concepts. Dans la suite, on va montrer que des problèmes Π sont **NP**-complets en utilisant :

1. La production d'un certificat pour prouver qu'ils sont bien dans **NP**.
2. Une réduction many-one d'un problème qu'on sait être **NP**-difficile à Π.

Comme pour le moment on ne connaît que `SAT` comme problème **NP**-difficile, on commencera par utiliser celui-ci.

9 `3SAT` est NP-complet

Tout d'abord, il est facile de voir que `3SAT` est dans **NP** : c'est un cas particulier de `SAT` qui est dans **NP**.

Soit une clause (il faudrait faire dépendre les y_i de j, mais ça devient vite illisible) $C_j = y_1 \vee y_2 \vee y_3 \vee ... \vee y_k$, on peut la transformer en une conjonction de 3-clauses équivalentes en introduisant $k-3$ nouvelles variables (il faudrait les faire dépendre de j aussi) :

$$\phi_j = (y_1 \vee y_2 \vee a_1) \wedge (\overline{a_1} \vee y_3 \vee a_2) \wedge \ldots \wedge (\overline{a_{k-4}} \vee y_{k-2} \vee a_{k-3}) \wedge (\overline{a_{k-3}} \vee y_{k-2} \vee y_{k-1})$$

Pour toutes valeurs prises par les y_i, C_j est vraie si et seulement si ϕ_j est satisfiable (par un choix adapté des a_i) :

- Si la C_j est fausse c'est que tous les y_i sont faux. On en déduit que si ϕ_j est vraie alors a_1 est vrai et donc a_2 est vrai ... a_{k-3} est vrai mais aussi que $\overline{a_{k-3}}$ est vrai, c'est impossible. Donc si C_j fausse, alors ϕ_j n'est pas satifiable.
- Réciproquement, si C_j est vraie alors au moins un des y_i est vrai. Si $3 \leq i \leq k-3$, on peut donc choisir $a_{i-2} =$ `vrai` et $a_{i-1} =$ `faux` et que la clause de y_i soit `vrai`. On peut alors propager les valeurs et remarquer que si $a_1 = \ldots = a_{i-1} =$ `vrai` et $a_{i-2} = \ldots = a_{k-3} =$ `faux` alors la formule ϕ_j s'évalue à `vrai` : elle est satisfiable.

On peut donc vérifier que la réduction est correcte, on part de $C = \bigwedge_j C_j$ qu'on transforme en $\phi = \bigwedge_j \phi_j$:

- Si la formule de départ est satisfiable, il existe une affectation des variables y telle que toutes les clauses C_j sont vraies. Et donc, il existe une affectation des y et des a telles que toutes les ϕ_j soient vraies. C'est exactement dire que la formule d'arrivée est satisfiable.

- Si ϕ est satisfiable, alors il existe des affectations pour les y et les a telles que toutes les ϕ_i soient vraies. Et donc toutes les C_j sont vraies. C'est exactement dire que C est satisfiable.

Pour chaque clause C_j, le nombre de nouvelles clauses crées est inférieur à trois fois le nombre de littéraux dans la clause. On peut donc construire la nouvelle formule en temps polynomial à partir de ϕ. On a donc prouvé l'existence d'une réduction many-one de `SAT` à `3SAT`, et donc `3SAT` est **NP**-difficile. On a donc démontré le résultat suivant :

Théorème 2 *`3SAT` est* **NP***-complet.*

10 Problèmes de graphes

Dans cette partie, on considère des algorithmes sur des graphes non-orientés avec n arêtes.

10.1 `IndependentSet`

Définition : Soit G un graphe d'ensemble de sommets V. Un *ensemble indépendant* de G est un sous-ensemble I de V tel que pour tout $x, y \in I$, il n'y a pas d'arête entre x et y.

Le problème `IndependentSet` consiste à décider s'il y a des grands ensembles de sommets indépendants. Ce sera notre premier problème de graphes étudié dans ce chapitre.
Définition : Le problème `IndependentSet` consiste, étant donné un graphe G et un entier $k \geq 1$, à savoir s'il existe un ensemble indépendant de cardinal au moins k dans G.

Nous allons montrer le résultat suivant.

Théorème 3 *`IndependentSet` est* **NP***-complet.*

Pour prouver ce théorème nous allons d'abord montrer que `IndependentSet` est dans **NP**, puis que `3SAT` se réduit à `IndependentSet`.

Le fait que `IndependentSet` soit dans **NP** est facile. En effet, si on nous donne un ensemble S de taille au moins k, on peut vérifier en temps polynomial que c'est bien un ensemble indépendant : il suffit de vérifier que pour toute paire d'éléments de S, il n'y a pas d'arête qui les relie.

Pour la réduction, c'est un peu plus compliqué. On doit, pour toute formule ϕ sous forme 3FNC, construire un graphe G et un entier k tels que ϕ est satisfiable si et seulement si G admet un ensemble indépendant de taille au moins k. On procède de la façon suivante. Soit ϕ une formule avec k clauses. Pour chaque clause $C = y_1 \vee y_2 \vee y_3$ de ϕ on crée 3 sommets étiquetés par y_1, y_2 et y_3. Chaque y_i est un littéral, c'est donc soit une variable x_j, soit sa négation $\overline{x_j}$. On relie ces trois sommets les uns aux autres par des arêtes bleues. Dans un deuxième temps, on relie chaque sommet étiqueté par une variable x à toutes les variables $\overline{x}$, avec des arêtes rouges. On note G_ϕ le graphe obtenu ainsi à partir de ϕ.

Montrons que la réduction est valide :

- Si ϕ est satisfiable, il existe une affectation des variables qui la rend vraie. Pour chaque clause de ϕ, on choisit un des littéraux qui vaut vrai, et on sélectionne le sommet correspondant du graphe dans S. Il y a bien k sommets dans S, un par clause. Comme on n'a pris qu'un sommet dans chaque clause, il ne peut pas y avoir d'arête bleue entre deux sommets de S. Comme les arêtes rouges sont de la forme $x - \overline{x}$, et qu'on ne peut prendre qu'un des deux dans S (x si dans notre affectation $x =$ `vrai`, et $\overline{x}$ sinon), il n'y a pas non plus deux sommets de S reliés par une arête rouge.

- Supposons que G_ϕ ait un ensemble indépendant de taille au moins k. Alors il est de taille k (il ne peut y avoir qu'un seul sommet par triangle correspondant aux clauses à cause des arêtes bleues). Pour tout y_i dans S, on fixe la variable x_j correspondante pour que y_i soit `vrai`. Il ne peut pas y avoir d'incompatibilité à cause des arêtes rouges. S'il reste des variables libres on les fixe arbitrairement. A la fin, il y a au moins un littéral à vrai par clause, et donc ϕ est satisfiable.

Comme le graphe G_ϕ peut être construit en temps polynomial, on a bien une réduction many-one, ce qui complète la preuve.

10.2 `Clique`

Dans un graphe non-orienté, une *clique* est un ensemble de sommets qui sont tous reliés entre eux : le sous-graphe induit est le graphe complet.

Le problème `Clique` consiste étant donné un graphe G et un entier $k \geq 1$ de déterminer s'il existe une clique de taille au moins k dans G.

On se convainc facilement que V forme une clique dans G ssi V est un ensemble indépendant de $\overline{G}$, le graphe complémentaire de G. On a donc une réduction many-one de `IndependentSet` vers `Clique` et donc le théorème suivant :

Théorème 4 *`Clique` est* **NP***-complet.*

10.3 `VertexCover`

Une *couverture de sommets* C dans un graphe G est un sous-ensemble de sommets tel que toute arête de G a au moins une de ses deux extrémités dans C.

Le problème `VertexCover` consiste, étant donné un graphe G et un entier k, à déterminer si G admet une couverture de sommets de taille au plus k.

On a la propriété suivante : C est une couverture de sommets ssi $V \subset C$ est un ensemble indépendant. On en déduit le théorème suivant.

Théorème 5 *`VertexCover` est* **NP***-complet.*