

10. NP-complétude

*Enseignant: Arnaud Casteigts**Assistants: A. Berger, M. De Francesco**Moniteurs: E. Bussod, N. Beghdadi*

Dans ce cours, nous donnons un dernier exemple de réduction polynomiale entre problèmes de NP, puis nous définissons la classe des problèmes **NP-complets**, à savoir les problèmes de NP qui sont “au moins aussi difficiles” que n’importe quel autre problème de NP. L’existence même de tels problèmes est assez surprenante. Une conséquence forte est que si un seul de ces problèmes s’avère résoluble en temps polynomial, alors tous les problèmes de NP le sont également et on a $P = NP$. Cela étant jugé improbable, le fait qu’un problème soit NP-complet est vu comme un indicateur fiable que ce problème est difficile.

10.1 Réductions polynomiales (suite et fin)

Lors du dernier cours, nous n’avons pas eu le temps de couvrir cette dernière réduction.

10.1.1 $SAT \leq_p 3-SAT$

3-SAT : Même problème que SAT, mais où chaque clause est de taille 3, par exemple :

$$\phi_3 = (x_1 \vee \overline{x_2} \vee x_4) \wedge (\overline{x_1} \vee x_2 \vee \overline{x_4})$$

Réduction : Étant donné une formule SAT ϕ dont les clauses sont de longueur arbitraire, nous allons construire une formule 3-SAT ϕ' dont les clauses sont de longueur 3 telle que ϕ' est satisfaisable si et seulement si ϕ est satisfaisable. L’idée principale est que chaque clause longue peut être cassée en plusieurs clauses de longueur 3 tout en préservant l’équivalence entre les deux formules. Cela requiert l’ajout de variables supplémentaires.

Prenons l’exemple de la clause $(x_1 \vee x_2 \vee x_3 \vee x_4)$. Cette clause est satisfaisable si et seulement si au moins une variable parmi x_1, x_2, x_3, x_4 est à **vrai**. On peut la casser en deux en introduisant une nouvelle variable x_a , en écrivant $(x_1 \vee x_2 \vee x_a) \wedge (\overline{x_a} \vee x_3 \vee x_4)$. Cette nouvelle formule est bien équivalente. Pour s’en convaincre, supposons qu’au moins une variable soit vraie, disons x_1 , alors cela permet de satisfaire la première clause. Cela permet aussi de mettre x_a à faux et donc $\overline{x_a}$ à vrai, ce qui satisfait aussi la seconde clause (un raisonnement similaire s’applique si une autre variable que x_1 est vraie). À l’inverse, si les quatre variables sont fausses, alors la seule façon de satisfaire cette formule est d’avoir

à la fois x_a et $\overline{x_a}$ à vrai, ce qui est impossible. Les formule est donc bien satisfaisable si et seulement si au moins une des quatre variables est à vrai.

La même technique fonctionne avec des littéraux arbitraires, par exemple $(x_1 \vee \overline{x_2} \vee \overline{x_3} \vee x_4)$ devient $(x_1 \vee \overline{x_2} \vee x_a) \wedge (\overline{x_a} \vee \overline{x_3} \vee x_4)$. Par ailleurs, si la clause est plus grande, on peut chaîner les clauses avec d'autres variables intermédiaires, par exemple $(x_1 \vee x_2 \vee x_3 \vee x_4 \vee x_5 \vee x_6)$ devient $(x_1 \vee x_2 \vee x_a) \wedge (\overline{x_a} \vee x_3 \vee x_b) \wedge (\overline{x_b} \vee x_4 \vee x_c) \wedge (\overline{x_c} \vee x_5 \vee x_6)$. Par ailleurs, puisqu'on veut des clauses de longueur exactement 3, si une clause est trop petite, on peut simplement répéter l'un de ses littéraux sans affecter sa satisfaisabilité. Enfin, observons que la taille de la nouvelle formule est polynomiale en la taille de l'ancienne. $\square$

Pseudo-code :

```
def is_SAT( $\phi$ ):
     $\phi'$  = casser_les_grandes_clauses( $\phi$ )           // rapide à calculer
     $\phi'$  = compléter_les_petites_clauses( $\phi'$ )       // rapide à calculer
    return is_3_SAT( $\phi'$ )
```

10.2 NP-complétude

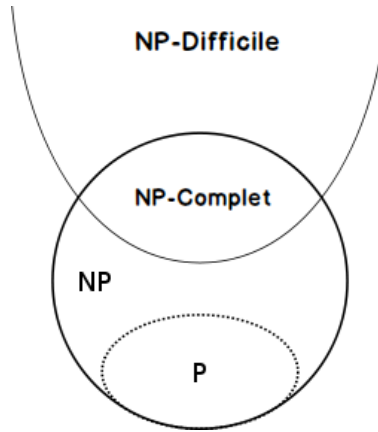
Un problème est dit NP-difficile si *n'importe quel problème* de NP peut s'y réduire en temps polynomial. Autrement dit, un algorithme qui résoudrait rapidement un tel problème peut être utilisé pour résoudre rapidement n'importe quel problème de NP. Les problèmes NP-difficile ne sont pas forcément eux-même dans NP, ils peuvent avoir une complexité plus élevée. Ils peuvent aussi ne pas être des problèmes de décisions.

Si un problème NP-difficile est aussi dans NP, alors c'est un problème NP-complet.

En y réfléchissant, le fait même qu'il existe des problèmes NP-complets n'est pas évident. Il aurait pu exister plusieurs problèmes maximalelement difficiles dans NP qui ne se réduisent pas les uns aux autres. Et pourtant, les problèmes NP-complet existent et sont même très nombreux. En fait, la plupart des problèmes que nous avons discuté ces dernières semaines sont NP-complets.

D'une certaine manière, ces problèmes sont tous les mêmes. Trouvez un algorithme en temps polynomial pour n'importe lequel d'entre eux et vous aurez résolu n'importe quel autre. Bien sûr, cela est jugé improbable.

Voici le diagramme d'inclusion des classes, en supposant que $P \neq NP$:



10.2.1 Concrètement, comment prouver qu'un problème est NP-complet

Pour montrer qu'un problème/langage L est **NP-complet**, on doit montrer deux choses :

1. Montrer que L est dans **NP**
2. Montrer que L est **NP-difficile**.

En général, la première étape est assez facile. Il suffit d'établir qu'il existe des certificats positifs pour n'importe quel mot de L . Autrement dit, s'assurer que pour tout $w \in L$, il existe une "preuve", rapide à vérifier, que w est bien dans L . Nous avons déjà couvert cela au cours n°8 (par exemple, pour 3-COLORING, un mot w correspond à un graphe, et pour chaque graphe 3-colorable, il existe un certificat positif qui peut être vérifié en temps polynomial : la coloration elle-même).

Pour la deuxième étape, il suffit d'identifier un autre problème L' dont on sait déjà qu'il est **NP-difficile**, et de montrer que L' se réduit en temps polynomial à L , exactement comme nous l'avons fait dans le cours n°9 (et ci-dessus pour 3-SAT). On montre ainsi que L est au moins aussi difficile que L' et qu'il est donc lui-aussi **NP-difficile**. Attention à ne pas vous tromper de direction ! En l'occurrence, réduire L vers un problème difficile ne démontrerait en rien sa difficulté.

Exemple complet : Comment montrer que CLIQUE est NP-complet ?

Pour rappel, CLIQUE = $\{(G, k) \mid \text{le graphe } G \text{ admet une clique de taille } k\}$. On doit montrer deux choses pour arriver au résultat :

1. CLIQUE est dans **NP** (déjà vu au cours n°8) : Si un graphe G admet une clique de taille k , alors une telle clique constitue elle-même une preuve facile à vérifier. Si on nous donne cette clique K , il suffit d'énumérer toutes les paires de sommets de K et vérifier qu'une arête existe bien entre ces deux sommets. Cela prend essentiellement k^2

étapes, et comme $k \leq n$, disons au plus $O(n^2)$ pour un graphe à n sommets, ce qui est polynomial. CLIQUE est donc dans NP.

2. CLIQUE est NP-difficile : En supposant que l'on sache déjà que SAT est NP-difficile, la réduction polynomiale de SAT à CLIQUE que nous avons vu dans le cours n°9 implique que CLIQUE est NP-difficile.

Les réductions peuvent varier significativement selon les problèmes, mais la mécanique générale de ces deux étapes est toujours exactement la même. Veillez à bien la comprendre.

10.3 Questions connexes

10.3.1 Premier problème NP-complet ?

Mais alors, quel a été le premier problème NP-complet de l'histoire ? Et comment a-t-il été démontré, sachant qu'il n'existait pas d'autre problème NP-difficile pour effectuer une réduction ? Et bien c'est le fameux théorème de Cook-Levin (1971), qui démontre que le problème SAT est NP-complet. Ce théorème utilise la première définition de NP (basée sur le non-déterminisme). Il montre que pour toute machine de Turing non-déterministe M qui termine en temps polynomial et pour toute entrée w pour cette machine, on peut construire en temps polynomial une formule SAT telle que cette formule est satisfaisable si et seulement si M accepte w . Nous verrons cette preuve lors du prochain cours.

Ce résultat est le seul qui a vraiment besoin de non-déterminisme. Une fois que l'on dispose de ce premier problème NP-complet, on peut l'utiliser pour établir la NP-complétude de nombreux autres problèmes, et en cascade, d'utiliser ces autres problèmes pour d'autres problèmes...

10.3.2 Existe-t-il des problèmes intermédiaires ?

Existe-t-il des problèmes dans NP qui ne sont ni dans P ni NP-complets ? Le théorème de Ladner nous dit que si $P \neq NP$, alors il doit exister des problèmes intermédiaires dans NP, c'est à dire des problèmes qui ne sont ni dans P ni NP-complets. Cependant, la grande majorité des problèmes de NP s'avère être soit dans P, soit NP-complets. Bizarrement, la difficulté semble sauter directement de l'un à l'autre. Par exemple, 2-SAT est dans P, mais 3-SAT est NP-complet. De même, 2-COLORING est dans P, mais 3-COLORING est NP-complet. Savoir s'il existe des problèmes "naturels" qui sont intermédiaires est une question ouverte (le problème défini par Ladner est assez artificiel).

Deux candidats sont GRAPH ISOMORPHISM et FACTORING, tous deux dans NP. À ce jour, personne n'a réussi ni à résoudre ces problèmes en temps polynomial, ni à montrer qu'ils sont NP-complets. Il est peu probable qu'ils soient NP-complets, car cela réfuterait plusieurs autres

conjectures. En revanche, ils pourraient s'avérer être dans P sans conséquence théorique majeure (mais bien sûr, avec des conséquences pratiques dramatiques pour FACTORING).

10.3.3 Qu'en est-il des certificats négatifs ?

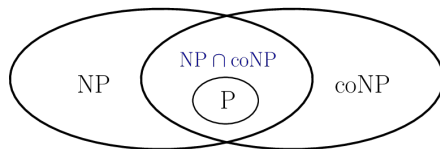
De la même manière que NP correspond aux problèmes qui admettent des certificats positifs vérifiables en temps polynomial, on peut définir la classe $co-NP$ comme suit :

- $co-NP$: Langages qui admettent des *certificats négatifs* vérifiables en temps polynomial.

Autrement dit, les langages tels que si la réponse est NON, il existe une preuve que $w \notin L$ rapide à vérifier. Réfléchissons quelques instants aux problèmes que l'on connaît. Existe-t-il une preuve (rapide à vérifier) qu'un graphe G n'admet pas de clique de taille k ? Ou qu'un graphe donné n'est pas 3-colorable ? Comment vous y prendriez-vous pour convaincre quelqu'un ? Je vous invite à faire une pause pour y réfléchir quelques instants... Une preuve évidente serait de lister toutes les colorations possibles et montrer que cela échoue. Mais cette preuve est de taille exponentielle dans la taille du graphe ! Ce n'est pas vérifiable en temps polynomial.

En fait, il est probable qu'aucun langage NP -complet ne soit dans $co-NP$. En revanche, leurs compléments y sont tous (ils sont même $co-NP$ -complets). Par exemple, le langage $NON-CLIQUE = \{(G, k) \mid G \text{ n'admet pas de clique de taille } k\}$ est $co-NP$ -complet. (Dire cela est vraiment la même chose que dire que $CLIQUE$ est NP -complet.)

Bien sûr, il existe des problèmes qui sont dans NP et dans $co-NP$ (mais qui ne sont complets dans aucune des deux). C'est notamment le cas de tous les problèmes dans P . En effet, on a clairement $P \subseteq NP$ et $P \subseteq co-NP$, car un algorithme de décision peut être lui-même vu comme un vérifieur d'appartenance ou de non-appartenance au langage utilisant un certificat vide. La situation est illustrée ci-dessous :



On connaît des problèmes qui sont dans $NP \cap co-NP$, mais potentiellement pas dans P . C'est notamment le cas de FACTORING. Pour GRAPH ISOMORPHISM, c'est moins clair, on sait qu'il est dans NP mais on ne sait pas le mettre dans $co-NP$: comment convaincre quelqu'un que deux graphes sont différents ? C'est un autre problème ouvert. (On sait le faire en utilisant des preuves interactives et probabilistes, et ces preuves ont d'ailleurs donné naissance à la notion de preuve à divulgation nulle de connaissance (*zero-knowledge proofs*), discutées brièvement pendant le dernier cours.)

Enfin, si $P = NP$, alors toutes ces classes s'effondrent et il ne reste plus que P .

10.4 PSPACE-complétude

Pour rappel, un langage est dans **PSPACE** s'il peut être décidé en utilisant une quantité polynomiale d'espace (quel que soit le temps). Nous avons déjà évoqué que $\text{NP} \subseteq \text{PSPACE}$. On peut montrer cela de deux manières, l'une ayant été vue en exercices (via le théorème de Savitch), l'autre consistant à observer qu'on peut énumérer tous les certificats possibles (l'énumération pouvant réutiliser l'espace entre deux certificats consécutifs).

De la même manière que certains problèmes sont **NP-complet**, on peut définir une notion de problème **PSPACE-difficile** et **PSPACE-complet**. Un problème est **PSPACE-difficile** si n'importe quel problème de **PSPACE** peut s'y réduire *en temps polynomial*. Il est **PSPACE-complet** s'il est **PSPACE-difficile** et dans **PSPACE**. De nombreux jeux combinatoires sont **PSPACE-complets** (p.ex. décider si un joueur a une stratégie gagnante au jeu *Reversi/Othello* généralisé en dimension $n \times n$).