Tutorial 2 : Complexity classes

Computational complexity theory, 5th semester.

2022

Exercice 1 — *Belonging to complexity classes*

For each of the problems in exercice 1 of the previous tutorial, tell to which of the following complexity class that problem belongs : P, NP, Co-NP, PSPACE, EXPTIME, NEXPTIME or EXPSPACE.

▶ **Correction**

Voici quelques exemples :

(BIP ?) : Pour savoir si un graphe $G = (V, E)$ est biparti, on peut utiliser l'algorithme suivant. On suppose que G est connexe, sinon on recommence l'algorithme pour chaque composante connexe.

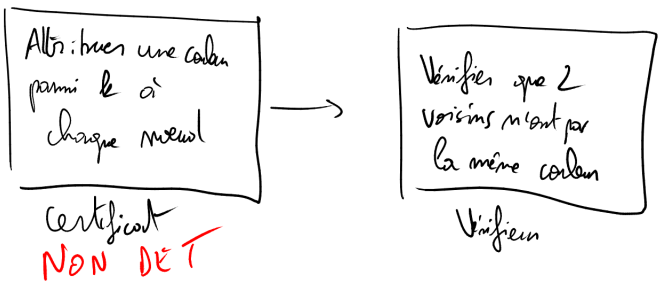
1: $A, B \leftarrow \emptyset$
2: Choisir un nœud v de G, le mettre dans A
3: **Tant que** Tant que tous les nœuds ne sont pas dans un ensemble A ou B **Faire**
4: Mettre dans B tous les voisins des nœuds de A
5: Mettre dans A tous les voisins des nœuds de B
6: Return OUI si A et B ne contiennent aucune paire de voisins et NON sinon

Ici A et B correspondent aux deux stables du graphe biparti. Cet algorithme résout bien le problème (BIP ?) (une preuve plus rigoureuse serait normalement attendue, mais elle est un peu longue et ce n'est pas le plus important ici).

Cet algorithme est polynomial car, à chaque itération de la boucle, on effectue au plus $O(|E|)$ itérations (pour chercher les voisins de A et de B) et à chaque itération au moins un nœud supplémentaire est ajouté à A ou B. Donc la complexité est $O(|V||E|)$. Cette complexité est polynomiale (quadratique) en la taille de G. Donc (BIP ?) appartient à DTIME(n^2) $\subset$ P.

(k-COL ?) : Savoir si un graphe $G = (V, E)$ est k coloriable (chaque nœud est colorié avec une couleur parmi k et on veut que deux voisins n'aient pas la même couleur) peut se résoudre avec l'algorithme non déterministe suivant. On note $c_1, c_2, \ldots, c_k$ les k couleurs.

1: CERTIFICAT
2: **Pour** $v \in V$ **Faire**
3: Choisir de manière **non déterministe** une couleur c_i parmi les k couleurs.
4: On note $c(v)$ cette couleur
5: VERIFIEUR
6: **Pour** $(u, v) \in E$ **Faire**
7: **Si** $c(u) = c(v)$ **Alors**
8: **Renvoyer** NON
9: **Renvoyer** OUI

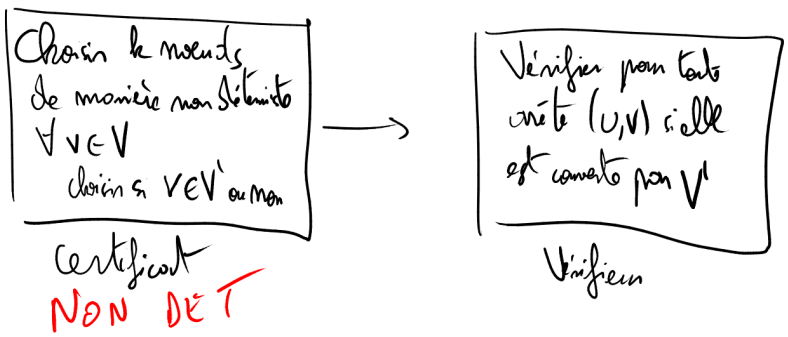


Cet algorithme résout (k-COL) : s'il existe une k Coloration de G, alors posons $d(v)$ la couleur du nœud v dans cette coloration. Si le certificat choisit la couleur $d(v)$ à la ligne 3, on sait que deux voisins n'ont pas la même couleur et le vérifieur répondra OUI. Donc les instances positives sont faiblement acceptées. S'il n'existe pas de k coloration de G alors quelques soient les couleurs $c(v)$ attribuées aux nœuds de G il existera toujours deux voisins a et b qui auront la même couleur et le vérifieur répondra NON à la ligne 8 au moment de l'itération où $(u, v) = (a, b)$. Donc les instances négatives sont fortement refusées.

Cet algorithme est polynomial, le certificat utilise k opérations pour effectuer son choix non déterministe pour chaque nœud, donc $O(k|V|) = O(|V|)$ opérations (on rappelle que, dans (k-COL ?), contrairement à (CHROMA), k ne fait pas partie de l'entrée, c'est une constante fixée, par exemple (3-COL) ou (4-COL)). Le vérifieur effectue $O(|E|)$ opérations. Donc la complexité de cet algorithme est $O(|V| + |E|)$ donc linéaire en la taille du graphe. Donc (k-COL ?) appartient à NTIME(n)$\subset$ NP.

(VERTEX COVER) : Savoir si dans un graphe $G = (V, E)$, on peut couvrir toutes les arêtes avec au plus k nœuds peut se résoudre avec l'algorithme non déterministe suivant.

1: CERTIFICAT
2: $V' \leftarrow V$
3: **Pour** $v \in V$ **Faire**
4: Choisir de manière **non déterministe** de mettre v dans V' ou non
5: VERIFIEUR
6: **Si** $|V'| > k$ **Alors**
7: **Renvoyer** NON
8: **Pour** $(u, v) \in E$ **Faire**
9: **Si** $u \notin V'$ et $v \notin V'$ **Alors**
10: **Renvoyer** NON
11: **Renvoyer** OUI



Cet algorithme résout (VERTEX COVER) : s'il existe une couverture des arêtes par les sommets, alors il existe un ensemble V' de nœuds qui couvre toutes les arêtes et $|V'| \leq k$. Si le certificat décide de mettre tous ces nœuds dans V' à la ligne 4, alors le vérifieur répondra OUI. Donc les instances positives sont faiblement acceptées. S'il n'existe pas de tel ensemble V', quel que soit le choix du certificat, soit $|V'| > k$ soit il existera deux voisins a et b qui ne seront pas dans V' et le vérifieur répondra NON respectivement aux lignes 7 ou 9 au moment de l'itération où $(u, v) = (a, b)$. Donc les instances négatives sont fortement refusées.

Cet algorithme est polynomial, le certificat utilise 1 opération pour effectuer son choix non déterministe pour chaque nœud, donc $O(|V|)$. Le vérifieur effectue 1 opération pour la comparaison et $O(|E|)$ opérations pour la vérification de la couverture. Donc la complexité de cet algorithme est $O(|V| + |E|)$ donc linéaire en la taille du graphe. Donc (VERTEX COVER) appartient à NTIME(n)$\subset$ NP.

(PLANARITY ?) : je vous montre ci-après une mauvaise preuve d'appartenance à NP. On pourrait penser que savoir si dans un graphe $G = (V, E)$ est planaire peut se résoudre avec l'algorithme non déterministe suivant pour dessiner le graphe sans que les arêtes ne se croisent.

1: CERTIFICAT
2: **Pour** $v \in V$ **Faire**
3: Choisir de manière **non déterministe** des coordonnées x_v et y_v de v
4: VERIFIEUR
5: **Pour** $(u, v); (w, z) \in E$ **Faire**
6: **Si** les segments $[(x_u, y_u); (x_v, y_v)]$ et $[(x_w, y_w); (x_z, y_z)]$ se croisent **Alors**
7: **Renvoyer** NON
8: **Renvoyer** OUI

Il n'y a pas de doute, cet algorithme résout bien (PLANARITY ?). Si le graphe est planaire, il existe des coordonnées pour chaque nœud de sorte que deux arêtes ne se croisent pas et si le certificat les choisit alors le vérifieur répondra OUI. Sinon quelque soit le choix du certificat, le vérifieur répondra NON.

Mais cet algorithme n'est pas polynomial. Car rien ne prouve que x_v et y_v sont bornés ; sous-entendu, combien de bits faut-il pour représenter x_v et y_v ? Si cela se trouve il faut des coordonnées très grandes ou des coordonnées très précises et très rapprochées. Dans les deux cas, la taille (en bits) de ces coordonnées est non bornées et rien ne dit qu'elle est polynomiale. Donc cet algorithme ne prouve pas que (PLANARITY ?) appartient à NP.

En fait, on peut prouver que

— on peut se restreindre à des coordonnées entières pas trop grandes, ce qui règle notre problème, mais la preuve est en dehors du cadre de ce cours

— en fait (PLANARITY ?) est dans P, il existe un algorithme déterministe qui résout ce problème en temps polynomial.

(TAU ?) : Ce problème est dans Co-NP. Pour le voir, il suffit de montrer que son complémentaire est dans NP. Il se trouve que son complémentaire est (SAT ?).

On rappelle la définition de (TAU ?) : Soit une formule φ avec n variables $x_1, x_2, \ldots, x_n$ est-ce que $\forall x_1, x_2, \ldots, x_n | \varphi(x_1, x_2, \ldots, x_n)$?

Le complémentaire (Co-TAU ?) de (TAU ?) s'écrit donc ainsi :

Soit une formule φ avec n variables $x_1, x_2, \ldots, x_n$ est-ce que $\neg(\forall x_1, x_2, \ldots, x_n | \varphi(x_1, x_2, \ldots, x_n))$? Autrement dit est-ce que $\exists x_1, x_2, \ldots, x_n | \neg\varphi(x_1, x_2, \ldots, x_n)$?

En appelant $\phi = \neg\varphi$ on obtient

Soit une formule ϕ avec n variables $x_1, x_2, \ldots, x_n$ est-ce que $\exists x_1, x_2, \ldots, x_n | \phi(x_1, x_2, \ldots, x_n)$.

On tombe bien sur (SAT?) qui est dans NP, donc (TAU?) est dans Co-NP.

(QBF ?). Ce problème est plus complexe, il est dans PSPACE. On va d'abord montrer qu'il est dans EXPTIME.

On rappelle sa définition : Soit une formule φ avec n variables $x_1, x_2, \ldots, x_n$ est-ce que $\exists x_1 \forall x_2 \exists \ldots \forall x_n | \varphi(x_1, x_2, \ldots, x_n)$.

Pour montrer qu'il est dans EXPTIME, on peut utiliser l'algorithme suivant :

1: $\phi_{n+1}(x_1, x_2, \ldots, x_n) \leftarrow \varphi(x_1, x_2, \ldots, x_n)$
2: **Pour** i de n à 1 **Faire**
3: **Si** x_i est associé à un quantificateur $\forall$ **Alors**
4: $\phi_i(x_1, x_2, \ldots, x_{i-1}) \leftarrow \phi_{i+1}(x_1, x_2, \ldots, x_{i-1}, \top) \wedge \phi_{i+1}(x_1, x_2, \ldots, x_{i-1}, \bot)$
5: **Sinon**
6: $\phi_i(x_1, x_2, \ldots, x_{i-1}) \leftarrow \phi_{i+1}(x_1, x_2, \ldots, x_{i-1}, \top) \vee \phi_{i+1}(x_1, x_2, \ldots, x_{i-1}, \bot)$
7: Ecrire le tableau des valeurs de $\varphi(x_1, x_2, \ldots, x_n)$ pour tout $x_1, x_2, \ldots, x_n$.
8: Utiliser ce tableau pour calculer ϕ_1

Cet algorithme construit explicitement la formule logique égale à $\exists x_1 \forall x_2 \exists \ldots \forall x_n | \varphi(x_1, x_2, \ldots, x_n)$. Par exemple appliqué à $\exists x_1 \forall x_2 : x_1 \Rightarrow x_2$ on aurait successivement

— $\phi_3(x_1, x_2) = x_1 \Rightarrow x_2$
— $\phi_2(x_1) = (x_1 \Rightarrow \top) \wedge (x_1 \Rightarrow \bot)$
— $\phi_1 = ((\top \Rightarrow \top) \wedge (\top \Rightarrow \bot)) \vee ((\bot \Rightarrow \top) \wedge (\bot \Rightarrow \bot)$

Cette formule est de taille exponentielle ($O(2^n)$) et se calcule en temps exponentiel. Le problème est donc dans DTIME(2^n)$\subset$ EXPTIME. On voit que l'espace de cet algorithme est aussi exponentiel puisqu'il faut un espace exponentiel pour construire ϕ_1.

Mais on peut en faire résoudre ce problème en espace polynomial avec l'algorithme récursif suivant :

1: **function** SOLVE($\varphi(x_1, x_2, \ldots, x_n)$, $q \in \{\exists, \forall\}$)
2: $\phi_\top(x_1, x_2, \ldots, x_{n-1}) \leftarrow \varphi(x_1, x_2, \ldots x_{n-1}, \top)$
3: $\phi_\bot(x_1, x_2, \ldots, x_{n-1}) \leftarrow \varphi(x_1, x_2, \ldots x_{n-1}, \bot)$
4: **Si** $q = \exists$ **Alors**
5: **Renvoyer** SOLVE($\phi_\top(x_1, x_2, \ldots, x_{n-1}), \forall$) $\vee$ SOLVE($\phi_\bot(x_1, x_2, \ldots, x_{n-1}), \forall$)
6: **Sinon**
7: **Renvoyer** SOLVE($\phi_\top(x_1, x_2, \ldots, x_{n-1}), \exists$) $\wedge$ SOLVE($\phi_\bot(x_1, x_2, \ldots, x_{n-1}), \exists$)
8: **Renvoyer** SOLVE($\varphi(x_1, x_2, \ldots, x_n), \forall$)

Cet algorithme est à peu près le même que le précédent, à ceci près qu'on calcule la valeur de ϕ_1 récursivement. Il reste en temps exponentiel, car chaque appel récursif effectue lui même deux appels récursifs, donc on a $O(2^n)$ opérations. Mais contrairement à l'algorithme précédent, on ne stocke pas tous les résultats de tous les appels récursifs en mémoire en même temps, on les fait remonter petit à petit. La profondeur des appels récursifs est n. Donc on ne stocke en mémoire pas plus de $O(n)$ informations en même temps. Le problème est donc dans DSPACE(n)$\subset$ PSPACE.

Exercice 2 — *Démonstrations simples*

1. Show that DTIME($f(n)$) $\subset$ NTIME($f(n)$).

▶ **Correction**

Soit un problème Π de DTIME($f(n)$). Il existe une machine de Turing déterministe qui résout Π en temps $O(f(n))$.
On peut aisément transformer cette machine en machine non déterministe, par exemple en ajoutant un arc dans l'état initial qui boucle sur lui même avec n'importe quel symbole déjà utilisé. La machine a le choix de boucler ou de commencer le calcul.
La complexité de cette machine ne change pas car le choix le plus rapide consiste à ne pas boucler du tout. Elle est toujours en $O(f(n))$. Et elle résout bien le même problème (il se trouve qu'elle accepte fortement les réponses OUI donc elle accepte faiblement les réponses OUI ; et elle refuse fortement les réponses NON).
Donc Π est dans NTIME($f(n)$).

2. Show that DTIME($f(n)$) $\subset$ DSPACE($f(n)$).

▶ **Correction**

Similaire à la question précédente, utiliser le fait qu'une machine ne peut pas écrire sur plus de cases que le nombre d'opérations qu'elle fait.

3. Show that, if NP = P then Co-NP = NP. Is the reciprocate true ?

▶ **Correction**

Il y a un piège ici, répondre d'abord à la question suivante et utiliser le résultat ici. La réciproque est peut être vraie, personne ne sait.

4. Show that P = Co-P.

▶ **Correction**

Soit un problème Π de P, montrons qu'il est dans Co-P. C'est à dire que le complémentaire de Π (le problème Π où les instances positives et négatives sont inversées) est dans P.

Il existe un algorithme déterministe polynomial $\mathcal{A}$ qui résout Π. Considérons l'algorithme qui applique $\mathcal{A}$ et renvoie la réponse inverse à $\mathcal{A}$. Cet algorithme résout le complémentaire de Π, et est déterministe et polynomial. Donc Π est bien dans Co-P.

5. Using the Savitch theorem, saying that NSPACE($f(n)$) $\subset$ DSPACE($f(n)^2$), prove that PSPACE = NPSPACE and EXPSPACE = NEXPSPACE. Deduce that NPSPACE = Co-NPSPACE and that NEXPSPACE = Co-NEXPSPACE.

▶ **Correction**

Soit un problème Π de NPSPACE, montrons qu'il est dans PSPACE. Il existe un algorithme non déterministe polynomial $\mathcal{A}$ qui résout Π. La complexité de cet algorithme est $O(n^c)$ où c est une constante indépendante de n. Donc Π $\in$ NSPACE(n^c) $\subset$ DSPACE(n^{2c}) $\subset PSPACE$. L'inclusion contraire consiste à prouver DSPACE($f(n)$) $\subset$ NSPACE($f(n)$), c'est à dire la même preuve que la première question de l'exercice. On a donc bien PSPACE = NPSPACE.

Un raisonnement similaire prouve que EXPSPACE = NEXPSPACE.

En utilisant un raisonnement similaire à la question précédente, on prouve que PSPACE = Co-PSPACE. Donc NPSPACE = Co-PSPACE. Ainsi Co-NPSPACE = PSPACE.

Donc NPSPACE = Co-PSPACE = PSPACE = Co-NPSPACE.

De même pour la partie exponentielle.

6. Using the deterministic time hierarchy theorem saying that DTIME($f(n)$) $\subsetneq$ DTIME($f(n)^2$) (among other results), prove that P $\subsetneq$ EXPTIME.

▶ **Correction**

On sait que, pour tout polynome n^c, on a $n^c = o(2^n) = O(2^n)$. Donc $DTIME(n^c) \subset DTIME(2^n)$. Donc P $\subset$ DTIME(2^n).

Or DTIME(2^n) $\subsetneq$ DTIME(2^{2n}) $\subset$ EXPTIME.

Donc P $\subsetneq$ EXPTIME.

Exercice 3 — *PSPACE $\subset$ EXPTIME*

1. Let Π be a PSPACE problem, how much cells of the tape can use a Turing machine that solve Π in polynomial space ?

▶ **Correction**

Il existe un algorithme déterministe qui résout Π en espace $O(n^c)$ où c est une constante indépendante de n.

La machine de Turing ne va donc écrire qu'au plus dans n^c cases.

2. In how many configurations that machine can be ? A configuration is defined by the position of the head, the state pointed by the state register and what is written on the tape.

▶ **Correction**

La tête peut alors aller sur au plus n^c cases et elle peut écrire sur au plus n^c cases. Il peut donc y avoir écrit sur la bande au plus 2^{n^c} mots différents. Le registre d'état peut être sur au plus un état par état de la machine de Turing et ce nombre est indépendant de l'entrée.

Il y a donc au plus $O(n^c 2^{n^c})$ configurations.

(Détail, cette réponse est partiellement fausse. Ce n'est pas parce que la machine écrit sur au plus n^c cases que la tête va se balader uniquement sur ces cases là. Rien ne l'empêche d'aller voir sur d'autres cases plus loin sur la bande. Rien n'empêche la tête d'aller sur la case d'indice $n^c + 1$ par exemple. Elle peut potentiellement aller à l'infini sur la bande sans écrire. J'en reparle dans la correction de la question suivante)

3. Can the machine be twice in the same configuration during the computation ?

► **Correction**

Non sinon elle boucle. Or un problème de PSPACE doit être résolu par une machine qui s'arrête pour chaque entrée.

C'est aussi ce qui permet de dire que la tête ne va pas aller à l'infini sur la bande. On sait que, au début du calcul, la bande possède l'entrée de la machine et que cette entrée est entourée par des cases vides. On sait donc, puisque la tête n'écrit pas sur plus de n^c cases qu'il y a au mieux, en plus de l'entrée, n^c cases non vides. On peut montrer que si la tête s'écarte trop de ces cases, et qu'elle rentre dans ce désert de cases vide trop profondément, alors la machine va boucler. Par exemple une machine qui dirait "Si la tête voit une case vide, elle va à droite, sinon elle écrit un 1 et va à gauche" risquerait de boucler s'il n'y avait que des cases vides sur la droite de la tête.

4. Deduce that the complexity of the machine is at most exponential.

► **Correction**

Donc la machine ne peut pas utiliser plus d'itérations que le nombre max de configurations qui est exponentiel.

Exercice 4 — *P = NP ⇒ EXPTIME = NEXPTIME*

We assume that P = NP.

1. Let Π be a NEXPTIME problem. What is the complexity of a non-deterministic Turing machine solving Π in exponential time ?

► **Correction**

Π a une complexité exponentielle, donc il existe une constante d telle que Π se résout en $O(2^{n^d})$

2. Let c be a constant, let Π_2 be a decision problem for which the instances are $\{x \cdot \pi^{2^{|x|^c}} \backslash x \subset \{0,1\}^*\}$, where π^d is the symbol π repeated d times. The positive instances are the ones where x encodes a positive instance of Π. Build a non deterministic algorithm that solves Π_2 in polynomial time. To obtain such a complexity, choose well the constant c.

► **Correction**

Posons $c = d$ où d est la constante de la question 1. Considérons l'algorithme $\mathcal{A}$ qui applique la machine de Turing de la question 1, cette machine résout le problème en temps $O(2^{|x|^d})$. Cet algorithme est exponentiel en $|x|$ mais il est polynomial en la taille d'une instance de Π_2 parce qu'une instance de Π_2 est de taille $|x| + 2^{|x|^d}$.

3. Deduce that Π ∈ EXPTIME.

► **Correction**

On sait donc que Π_2 appartient à NP. Donc, puisque par hypothèse P = NP, on a $\Pi_2 \in$P. Il existe un algorithme $\mathcal{B}$ déterministe qui résout Π_2 en temps polynomial

On peut donc résoudre Π en temps exponentiel déterministe avec l'algorithme suivant : construire l'instance $x \cdot \pi^{2^{|x|^d}}$ de Π_2 (ce qui prend un temps $O(2^{|x|^d})$) et la résoudre avec $\mathcal{B}$.

Puisque cet algo est polynomial en la taille de cette nouvelle instance, alors le résultat se calcule en temps exponentiel vis-à-vis de $|x|$.

On en déduit qu'il existe un algorithme déterministe exponentiel qui résout Π et donc $\Pi \in$EXPTIME.

4. Deduce that EXPTIME = NEXPTIME.

▶ **Correction**

Car les questions précédentes prouvent que NEXPTIME $\subset$ EXPTIME (et l'inverse vient du cours).