Complexité - TD 1.2

Benjamin Bordais

18 Novembre 2021

On donne ici quelques définitions de classes de complexité usuelles :

— $\mathsf{L} = \mathsf{SPACE}(\log n)$: est la classe des langages décidés par des machines de Turing déterministes utilisant un espace logarithmique ;

— $\mathsf{NL} = \mathsf{NSPACE}(\log n)$: est la classe des langages décidés par des machines de Turing non-déterministes utilisant un espace logarithmique ;

— $\mathsf{P} = \cup_{k\in\mathbb{N}}\mathsf{TIME}(n^k) = \mathsf{TIME}(n^{O(1)})$: est la classe des langages décidés par des machines de Turing déterministes en temps polynomial ;

— $\mathsf{NP} = \cup_{k\in\mathbb{N}}\mathsf{NTIME}(n^k) = \mathsf{NTIME}(n^{O(1)})$: est la classe des langages décidés par des machines de Turing non-déterministes en temps polynomial.

On a les inclusions suivantes :

$$\mathsf{L} \subseteq \mathsf{NL} \subseteq \mathsf{P} \subseteq \mathsf{NP}$$

Non Déterminisme

Pour chacun des problèmes de décisions suivants, donner un algorithme qui optimise le temps et/ou l'espace utilisé (dans le pire cas, asymptotiquement) en explicitant s'il y a un intérêt à utiliser du non-déterminisme.

Lorsque l'on conçoit un algorithme qui utilise du non-déterminisme, cela peut être vu comme des choix que l'on fait pour éviter d'explorer toutes les possibilités, ce qui fonctionne à condition que la condition d'acceptation d'une entrée soit compatible avec la sémantique du non-déterminisme : on accepte s'il existe un (ou une suite de) choix qui mène à l'acceptation.

Il est également à noter que pour une machine donnée, les choix non-déterministe que l'on fait sont sur un ensemble fini dont la taille est fixe et ne dépend pas de l'entrée (typiquement, on choisit si un bit vaut 0 ou 1). Par exemple, un (seul) choix non-déterministe ne permet pas de choisir un entier 0 et $2^k - 1$ si k est fonction de l'entrée. Cependant, on peut implémenter ce choix en répétant l'utilisation du non-déterminisme : par exemple on peut deviner successivement k bits ce qui permet de deviner un nombre entre un entier 0 et $2^k - 1$ via sa représentation binaire.

Question 1

— ***Entrée*** *: une liste (non vide) l d'entiers (naturels), un entier n*

— ***Sortie*** *: oui ssi n est le minimum de la liste l*

Solution 1 *Il faut parcourir la liste pour vérifier à la fois qu'elle contient n et qu'il n'y a pas d'élément plus petit. Le non-déterminisme ne permet pas d'être beaucoup plus efficace puisqu'il faudra de toutes façons parcourir toute la liste (cela peut être implémenté en espace logarithmique).*

Question 2

— ***Entrée*** *: une liste (non vide) l d'entiers (naturels)*
— ***Sortie*** *: oui ssi l contient un nombre qui n'est pas premier*

Solution 2 *Ici, le non-déterminisme est doublement utile : on devine l'indice de la liste où il y aurait un nombre qui n'est pas premier (ce qui permet d'éviter un test de primalité sur l'ensemble des éléments de la liste dans le pire cas), puis on devine l'un de ses diviseur non-trivial (ce qui permet d'éviter de tous les tester).*

Question 3
— ***Entrée*** *: un graphe $G = (V, E)$ orienté, pondéré (poids entiers strictement positifs), deux sommets s, t, un entier k*
— ***Sortie*** *: oui ssi k est le poids d'un plus court chemin de s vers t*

Solution 3 *Étant donné que k doit être le poids d'un chemin le plus court entre s et t, il n'est pas utile de deviner un chemin entre s et t et de tester s'il est de poids k puisqu'il faut ensuite vérifier qu'il est de poids minimal. On peut directement utiliser un algorithme de plus court chemin (comme l'algorithme de Dijkstra) et comparer avec k.*

Question 4
— ***Entrée*** *: un graphe $G = (V, E)$ orienté, deux sommets s, t*
— ***Sortie*** *: oui ssi il existe un chemin de s vers t*

Solution 4 *Ici, le non-déterminisme est particulièrement utile puisque si l'on arrive à exhiber un chemin entre s et t, on peut directement accepter (typiquement, on n'a pas à vérifier que celui-ci vérifie un propriété de minimalité quelconque comme dans le cas de la question précédente). En fait, on peut le faire en espace logarithmique (ce problème est donc dans* NL*). On donne une intuition de ce résultat : On ne devine le chemin en entier, mais on garde plutôt un pointeur sur le sommet courant et on devine les successeurs au fur et à mesure. On utilise en plus un compteur (en binaire) pour borner la taille du chemin considéré, puisque l'on sait que s'il existe un chemin entre s et t, il en existe un dont la taille est inférieur à la taille du graphe.*

Question 5
— ***Entrée*** *: un graphe $G = (V, E)$ orienté, deux sommets s, t*
— ***Sortie*** *: oui ssi il n'existe pas de chemin de s vers t*

Solution 5 *Ici, le non-déterminisme existentiel, comme il est défini pour les classes* NP *et* NL *puisque l'on ne peut pas deviner l'inexistence de chemin. On peut donc utiliser un algorithme déterministe polynomial de parcours en largeur/profondeur pour calculer l'ensemble des états accessibles depuis s et vérifier si t est dedans. Ainsi ce problème est dans* P*. Il est à noter qu'en fait ce problème est aussi dans* NL *(le résultat n'est pas évident, c'est le théorème d'Immerman-Szelepcsényi).*

Question 6
— ***Entrée*** *: un ensemble d'entiers $S = \{n_1, \ldots, n_k\}$, en entier n*
— ***Sortie*** *: oui ssi il existe un sous-ensemble non-vide $T \subseteq S$ tel que $\Sigma_{x \in T} x = n$*

Solution 6 *Ce problème est en fait connu sous le nom de* SubbetSum *et est un problème* NP*-complet c'est-à-dire qu'il est dans* NP *mais qu'il est également plus dur que n'importe quel autre problème dans* NP *(le problème précédent de la question 10 est, lui,* NL*-complet). Ainsi, on peut le résoudre en temps polynomial avec du non-déterminisme : on devine*

l'ensemble T et on vérifie que celui-ci vérifie la propriété souhaitée. Il n'est pas (à ce jour) connu d'algorithme déterministe qui permette de résoudre ce problème qui, dans le pire cas, ne prendrait pas un temps exponentiel en la taille de l'entrée (il faut énumérer toutes les possibilités, il y en a $2^{|S|}$).

Question 7

— ***Entrée*** *: un ensemble d'entiers $S = \{2^{n_1}, \ldots, 2^{n_k}\}$, en entier 2^n*
— ***Sortie*** *: oui ssi il existe un sous-ensemble non-vide $T \subseteq S$ tel que $\Sigma_{x \in T} \log x = n$*

Solution 7 *Ce problème, comme le précédent, est en fait mal posé puisque la classe de complexité que l'on a dépend beaucoup de la manière dont sont représentés les entiers considérés, spécifiquement en binaire (le problème précédent) ou en unaire (ce problème-ci). Le problème de* SubbetSum *est bien* NP*-complet lorsque les entiers sont écrits en binaire, mais il est dans* NL *lorsqu'ils sont écrits en unaire. Pour cela, on peut par exemple garder une somme courante (écrite en binaire), qui est incrémentée par n_i ou non selon un choix non-déterministe (il est important ici de ne pas deviner a priori n bits, un pour chaque entier n_i, pour savoir si on le rajoute car alors l'espace pris est linéaire et non logarithmique).*

Question 8

— ***Entrée*** *: un formule de la logique propositionelle φ ;*
— ***Sortie*** *: oui ssi cette formule φ est satisfiable*

Solution 8 *Il s'agit de l'exemple canonique de problème* NP*-complet, c'est-à-dire un problème dans* NP *qui est 'au moins aussi dur' que n'importe quel autre problème dans* NP*. Ici, le non-déterminisme est très utile puisqu'il permet de deviner une valuation satisfaisant la formule φ et de vérifier qu'elle satisfait bien cette formule.*

Question 9

— ***Entrée*** *: un formule de la logique propositionelle φ ;*
— ***Sortie*** *: oui ssi cette formule φ est une tautologie*

Solution 9 *De la même manière que pour la question 5, ici le non-déterminisme ne permet pas de deviner l'inexistence de valuation ne satisfaisant pas la formule φ. Cependant, il est intéressant de remarquer que le non-déterminisme peut être utilisé pour résoudre le complémentaire de ce problème. En effet, une formule n'est pas une tautologie si et seulement sa négation est satisfaisable, ce qui nous ramène à la question précédente. En fait, ce problème est co*NP*-complet.*

Question 10 (*)

— ***Entrée*** *: un formule de la logique propositionelle φ ;*
— ***Sortie*** *: oui ssi cette formule φ admet une unique valuation la satisfaisant*

Solution 10 *Ici, a priori, le problème n'est ni dans* NP *ni dans co*NP*. Pour le résoudre en temps polynomial, on a besoin à la fois de deviner une valuation satisfaisant la formule et de vérifier que c'est la seule. (d'une certaine manière, on a besoin à la fois de non-déterminisme existentiel – comme* NP *– et universel – comme co*NP*).*

Question 11 *On fixe un $k \in \mathbb{N}$.*

— ***Entrée*** *: un formule de la logique propositionelle φ en CNF à au plus k variables ;*
— ***Sortie*** *: oui ssi cette formule φ est satisfiable*

Solution 11 *Il est important de remarquer ici que le paramètre k ne dépend pas de l'entrée. Ainsi, on peut énumérer les 2^k valuations possibles et vérifier que l'une d'entre elles satisfait la formule φ. La valeur 2^k est une constante pour le problème, ainsi la complexité est en fait en espace logarithmique (ce qui est nécessaire pour vérifier qu'une valuation satisfait une formule en CNF). Ainsi ce problème est dans* L.

Question 12

- ***Entrée*** *: un formule de la logique propositionelle φ en CNF et un $k \in \mathbb{N}$ tel que φ a au plus k variables ;*
- ***Sortie*** *: oui ssi cette formule φ est satisfiable*

Solution 12 *Ici, on ne gagne rien par rapport à la question 8 car le nombre de variable nous est déjà indiqué dans la formule.*

Question 13 *On considère l'alphabet $\Sigma = \{(,)\}$.*

- ***Entrée*** *: un mot $w \in \Sigma^*$*
- ***Sortie*** *: oui ssi w est bien parenthésé (cela peut être reformulé en : w est généré par la grammaire $S \to (S) \mid SS \mid \epsilon$).*

Solution 13 *Il s'agit du langage de Dyck. Le non-déterminisme n'est pas très utilise ici, on peut résoudre ce problème de manière déterministe en espace logarithmique. Il suffit de considérer un compteur (en binaire) initialisé à 0 et de parcourir le mot en entrée. Le compteur est augmenté de 1 lorsque l'on voit une parenthèse ouvrante et diminué de 1 lorsque l'on voit une parenthèse fermante. Si ce compteur devient négatif, ou bien s'il ne vaut pas 0 à la fin du mot on rejette, sinon on accepte.*

Question 14 (*) *On considère l'alphabet $\Sigma = \{(,),[,]\}$.*

- ***Entrée*** *: un mot $w \in \Sigma^*$*
- ***Sortie*** *: oui ssi w est bien parenthésé (cela peut être reformulé en : w est généré par la grammaire $S \to (S) \mid [S] \mid SS \mid \epsilon$).*

Solution 14 *Il s'agit d'une version un peu plus complexe du langage de Dyck, qui peut également être décidé en espace logarithmique.*

Chaque symbole a un type, soit (ou [, et que chaque symbole est droit ou gauche indépendamment de son type. Chaque symbole gauche a un partenaire droit, et le but est de vérifier que chaque partenaire d'un symbole gauche est de même type. Pour le trouver, on utilise un compteur comme dans la question précédente. D'abord, on vérifie que le mot appartient au langage bien parenthésé de la question précédente indépendamment de leur type. Ainsi, on sait que symbole gauche a un partenaire droit. Il faut maintenant vérifier chaque symbole gauche est associé à une symbole fermant de même type. On observe alors que si $x[i]$ est un symbole gauche associé au symbole droit $x[j]$, alors le facteur $x[i+1{:}j-1]$ est un mot bien parenthésé (indépendamment du type). Ainsi, il suffit de boucler sur tous les $i = 1, \ldots, n$ tel que $x[i]$ est un symbole gauche et d'exécuter l'algorithme de la question précédente à partir de l'indice $i+1$. Quand le compteur devient négatif, cela indique que l'on est arrivé à la position j associé à la position i. On vérifie alors que le symbole à l'indice j a le même type que son partenaire d'inde i. On utilise donc un compteur et un pointeur. Par exemple, on peut avoir le pseudocode :

```
verifier que l est bien parenthese (question precedente)
i = 1
tant que i n'excede pas la longueur de l {
```

```
        aller a l'indice i-1
        lire le symbole a     // a = w(i)
        si a est est un symbole gauche {
                c = 1
                tant que c > 0 {     // trouve le partenaire de w(i)
                lire symbole a droite b
                si b symbole gauche, incremente c
                si b symbole droit, decremente c
        }
        si a et b sont de type different, rejetter
}
accepter
```

Pour conclure

Question 15 — *D'un manière générale, dans quels cas l'usage du non-déterminisme permet d'être (beaucoup) plus efficace ?*

— *Comment les classes de complexité $\mathcal{C}$ présentées plus haut se comparent-elles à leur dual co$\mathcal{C}$? (On rappelle que co$\mathcal{C} = \{\bar{L} \mid L \in \mathcal{C}\}$)*

Solution 15 — *Le non-déterminisme est particulièrement utile lorsqu'il est facile de vérifier qu'une proposition de solution (autrement appelé certificat) est effectivement une solution, mais qu'il n'est pas facile d'exhiber une solution (il est souvent nécessaire d'énumérer toutes les solutions possibles).*

— *Pour les classes $\mathcal{C}$ déterministes (c'est-à-dire qui n'utilise pas de machine non-déterministe), on a toujours $\mathcal{C} =$ co$\mathcal{C}$ (car il suffit d'inverser les état d'acceptation et de rejet). C'est donc le cas pour* L *et* P*. Cependant, ce n'est pas le cas a priori pour les classes non déterministes comme* NL *ou* NP*. Ainsi, la question* NP = co NP? *est à ce jour encore un problème. Toutefois, on a en fait* NL = co NL*.*