Complexité - TD 01

Benjamin Bordais

19 Novembre 2020

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 :

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

Question 1 *Rappeler quel est l'espace utilisé par une machine de Turing (en particulier, pour les cas où l'on considère des machines qui utilisent un espace $f(n) < n$ où n est la taille de l'entrée).*

Solution :

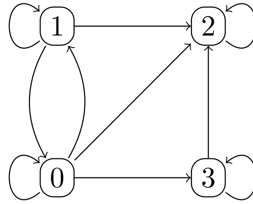
Une machine de Turing possède $k + 2$ rubans avec deux rubans distingués : le ruban d'entrée – en lecture seule – et le ruban de sortie – en écriture seule. Il faut noter que le ruban de sortie n'est pas pertinent lorsque l'on considère des machines de Turing traitant de problèmes de décision (étant donné que la sortie est alors oui ou non et est encodée dans les états de la machine). Les k autres rubans (avec k fixe, ne dépendant pas de l'entrée) sont les rubans de travail sur lesquels l'espace utilisé par la machine est compté : il s'agit alors du maximum (sur toute la durée de l'exécution) du nombre de cases mémoire utilisées cumulé sur toutes les bandes de travail.

Représentation des graphes

Un graphe orienté est un couple (V, E), où $V = \{0, \ldots, n-1\}$ est un ensemble fini de n sommets (*vertex*) et $E \subseteq V \times V$ est un ensemble fini d'arêtes (*edge*). Il existe deux façons standard de représenter un graphe avec un alphabet fini Σ contenant au moins $\{0, 1\}$, que nous décrivons ci-dessous. Étant donné un entier m, on note $\overline{m} \in \{0, 1\}^*$ son codage en base deux.

Liste d'adjacence : Une représentation en liste d'adjacence de G code le graphe comme une liste l de longueur n, où le u-ième élément de la liste est la liste de tous les sommets v tels que $(u, v) \in E$.

Matrice d'adjacence : Une représentation en matrice d'adjacence de G code le graphe comme une matrice M carrée à n lignes et n colonnes, et à valeurs dans $\{0, 1\}$, telle que $M[u][v] = 1$ si et seulement si $(u, v) \in E$.



Question 2 *Proposer un alphabet Σ pour représenter un graphe (que l'on pourrait utiliser pour les deux représentations).*

Solution :

On peut par exemple considérer l'alphabet $\Sigma = \{\mathtt{0}, \mathtt{1}, /, \bullet, \#\}$. Dans ce cas, le codage d'un graphe G par liste d'adjacence serait de la forme :

$$l_G \stackrel{\text{def}}{=} k_0^0/\dots/k_{m_1}^0 \bullet \dots \bullet k_0^{n-1}/\dots/k_{m_{n-1}}^{n-1}\#$$

Ainsi, on code les nœuds (représenter en tant qu'entiers) en binaire [1] avec $\mathtt{0}, \mathtt{1}$ et $/, \bullet$ sont des séparateurs respectivement des entiers et des listes, tandis que $\#$ est le symbole de fin de mots.

De la même manière, le codage par matrice d'adjacence est de la forme :

$$m_G \stackrel{\text{def}}{=} m_{0,0}m_{0,1} \ \dots m_{0,n-1} \bullet \dots \bullet m_{n-1,0} \dots m_{n-1,n-1}\#$$

Question 3 *Donnez une représentation du graphe ci-dessus en liste d'adjacence et en matrice d'adjacence à l'aide de cet alphabet.*

Solution :

Dans le cas de graphe G, on a :

$$l_G = \mathtt{0/01/10/11} \bullet \mathtt{0/01/10} \bullet \mathtt{10} \bullet \mathtt{10/11}\#$$

et

$$m_G \stackrel{\text{def}}{=} \mathtt{1111} \bullet \mathtt{1110} \bullet \mathtt{0010} \bullet \mathtt{0011}\#$$

Question 4 *Montrez qu'il existe une fonction logspace (i.e. qui peut être implémenté par une MT qui utilise un espace en $\log(n)$) qui à toute représentation d'un graphe G en liste d'adjacence associe une représentation de G en matrice d'adjacence.*

Réciproquement montrez qu'il existe une fonction logspace qui à toute représentation d'un graphe G en matrice d'adjacence associe une représentation de G en liste d'adjacence.

Solution :

Pour passer de la représentation par liste d'adjacence à la représentation par matrice d'adjacence, on considère deux compteurs i et j qui varient entre 0 et $k-1$ (où k est le nombre de symbole $\bullet$ plus 1) initialisés à 0 ainsi qu'un pointeur qui va parcourir (plusieurs fois) la liste d'indice i. Le compteur i est incrémenté à chaque fois qu'un symbole $\bullet$ est

1. La manière dont on représente les entiers, en binaire ou en unaire, peut changer radicalement la complexité d'un problème étant donné que celle-ci dépend de la taille de l'entrée qui est exponentiellement plus grande avec une représentation unaire plutôt qu'une représentation binaire. Pour le cas qui nous intéresse, on peut s'en sortir avec les deux représentations.

vu. Ensuite pour chaque valeur de j entre 0 et $k-1$, le pointeur va chercher dans la liste d'indice i un nœud égal à j. Si on en trouve un, un `1` est écrit sur la bande de sortie, sinon un `0` est écrit. Lorsque l'on a finit la boucle pour $j = k-1$, le symbole $\bullet$ est écrit et j est réinitialisé à 0. L'espace utilisé par cette algorithme est bien logarithmique car on considère deux compteurs et un pointeur qui, s'ils sont implémentés en binaire, prennent un espace logarithmique en la taille de l'entrée.

Réciproquement, on considère un algorithme qui fait varier un compteur i entre 0 et $k-1$ (où k est le nombre de symbole $\bullet$ plus 1). Ce compteur i est incrémenté dès qu'un bit `0` ou `1` est vu et est réinitialisé à chaque symbole $\bullet$. A chaque symbole $\bullet$, ce même symbole est écrit sur la bande de sortie est dès qu'un `1` est vu, le compteur i est copié sur la bande de sortie, précédé d'un symbole / si ce n'est pas la première fois que le compteur i est copié depuis qu'il a été réinitialisé. L'espace utilisé par cet algorithme correspond à la taille prise par ce compteur qui, s'il est implémenté en binaire, prend un nombre de bits logarithmique en la taille de l'entrée.

Fonctions LOGSPACE

Question 5 *Montrer que toute machine de Turing déterministe qui calcule en espace logarithmique, s'arrête après un nombre d'étapes polynomial.*

Solution :

Considérons une machine de Turing M sur un alphabet Q et un alphabet Σ qui fonctionne en espace logarithmique (sans perte de généralité, on suposera une seule bande de travail), c'est-à-dire tel qu'à tout instant d'un calcul sur une entrée w de taille $n = |w|$, le nombre de cellules sur les bande de travail utilisé est inférieur à $k \cdot \log(n)$ pour un k donné. Alors, sur une entrée w de taille n, le nombre de configurations différentes est borné par $f(n) = |Q| \cdot |\Sigma|^{k \cdot \log(n)} \cdot (k \cdot \log n) \cdot n$ (les deux derniers termes faisant référence à l'emplacement de la tête d'écriture sur la bande de travail et de lecture sur la bande d'entrée). On a alors qu'il existe d tel que $f(n) = O(n^d)$. Étant donné que la machine est déterministe, celle-ci boucle nécessairement si elle passe deux fois par la même configuration. Ainsi, le nombre d'étapes prise par cette machine pour s'arrêter est au plus polynomial.

Question 6 *Soient $h_1 : L_1 \mapsto L_2$ et $h_2 : L_2 \mapsto L_3$ deux fonctions calculables en espace logarithmique par des machines déterministes.*

Montrer que la fonction $h_2 \circ h_1 : L_1 \mapsto L_3$ peut aussi être calculée en espace logarithmique par une machine déterministe.

Solution :

La question précédente nous donne qu'il existe un certain d tel que, pour toute entrée w telle que $n = |w|$, on a $|h_1(w)| \leq n^d$. Ainsi, l'espace pris par un calcul de h_2 sur $h_1(w)$ pend un espace en $O(\log(n))$ et est donc logarithmique. Cependant, on doit ici remarquer qu'il n'est pas possible d'écrire la résultat du calcul de h_1 sur une bande de travail pour ensuite lancer directement h_2 dessus vu qu'on utiliserait alors un espace plus grand que logarithmique. Il faut donc que, lors du calcul de $h_2(h_1(w))$, lorsque h_2 doit lire le i-ème bit de $h_1(w)$ l'on effectue une nouvelle fois le calcul de $h_1(w)$ en ne renvoyant que le bit d'intérêt. Comme on considère la complexité en espace, et non en temps, on peut effectuer plusieurs fois le même calcul sans que cela affecte la complexité (à condition que l'on réutilise l'espace utilisé par les calculs précédents).

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 7

- ***Entrée*** *: une liste (non vide) l d'entiers (naturels), un entier n*
- ***Sortie*** *: oui ssi n est le minimum de la liste l*

Solution :

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 8

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

Solution :

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 9

- ***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 :

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 10

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

Solution :

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 11

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

Solution :

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 12 *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 :

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 13 (Bonus) *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* $S \to (S) \mid [S] \mid SS \mid \epsilon$*).*

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

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 nécessite d'énumérer toutes les solutions possibles).