

Chapitre 9

Complexité en espace

9.1 Classes de complexité en espace

9.1.1 Mesure de l'espace d'un calcul

Selon la discussion du chapitre 4, à un algorithme A correspond un système de transitions, c'est-à-dire un ensemble d'états $S(A)$ et une fonction d'évolution $\tau_A : S(A) \rightarrow S(A)$ qui décrit les évolutions entre ces états. Soit w une entrée d'un algorithme A , sur lequel A termine : dans le chapitre précédent, nous avons dit que le *temps de calcul* de l'algorithme A sur l'entrée w correspondait au nombre d'étapes du système de transitions associé.

Nous voulons maintenant parler de la mémoire utilisée. En fait, en théorie de la complexité, on parle plutôt *d'espace*, ou d'espace mémoire, que de mémoire. Pour cela, on va préciser comment on mesure l'espace d'une structure : rappelons que les états d'un algorithme correspondent à des structures sur une même signature.

Définition 9.1 *Considérons une structure du premier ordre sur la signature $(f_1, \dots, f_u, r_1, \dots, r_v)$. On rappelle que l'on dit qu'un emplacement (f, \bar{m}) est utile, si f_i est un symbole de fonction dynamique, et son contenu $\llbracket (f, \bar{m}) \rrbracket$ n'est pas **undef**. La taille (mémoire) de la structure \mathfrak{M} , notée $\text{size}(X)$ est le nombre d'emplacements utiles de X .*

On appelle *espace (mémoire) de calcul sur l'entrée w* le maximum de la taille (mémoire) de chacun des états de l'exécution de l'algorithme associée à l'entrée w :

Définition 9.2 *Soit A un algorithme. A chaque entrée w est associée une exécution X_0, X_1, \dots, X_t du système de transitions associé à A où $X_0 = X[w]$ est un état initial qui code w , et chaque X_i est un état, et $X_{i+1} = \tau_A(X_i)$ pour tout i . Supposons que w soit acceptée, c'est-à-dire qu'il existe un entier t avec X_t terminal. L'espace (mémoire) de calcul sur l'entrée w , noté $\text{SPACE}(A, w)$, est*

défini comme

$$\text{SPACE}(A, w) = \max_{0 \leq i \leq t} \text{size}(X_i)$$

Comme pour le temps, on raisonne à taille de donnée fixée.

Définition 9.3 (Espace de calcul) Soit A un algorithme qui termine sur toute entrée. L'espace (mémoire) de l'algorithme A est la fonction $f : \mathbb{N} \rightarrow \mathbb{N}$ telle que pour tout entier n , $f(n)$ est le maximum de $\text{SPACE}(A, w)$ pour les entrées (mots) w de longueur n . Autrement dit,

$$f(n) = \max_{|w|=n} \text{SPACE}(A, w).$$

9.1.2 Notation $\text{SPACE}(f(n))$

Comme pour le temps, on introduit la notation suivante.

Définition 9.4 Soit $t : \mathbb{N} \rightarrow \mathbb{N}$ une fonction. On définit la classe $\text{SPACE}(t(n))$ comme la classe des problèmes ou des langages qui sont solvables par un algorithme qui fonctionne en espace $\mathcal{O}(t(n))$. Si l'on préfère,

$$\text{SPACE}(t(n)) = \{L \mid L \text{ est un langage décidé par un algorithme en espace } \mathcal{O}(t(n))\}.$$

Définition 9.5 Soit $\mathfrak{M} = (M, f_1, \dots, f_u, r_1, \dots, r_v)$ une structure. On définit la classe $\text{SPACE}_{\mathfrak{M}}(f(n))$ comme la classe des problèmes ou des langages qui sont solvables par un algorithme qui fonctionne en espace $\mathcal{O}(f(n))$ sur \mathfrak{M} . Si l'on préfère,

$$\text{SPACE}_{\mathfrak{M}}(f(n)) = \{L \mid L \text{ est un langage décidé par un algorithme sur } \mathfrak{M} \text{ en espace } \mathcal{O}(f(n))\}.$$

En fait, cette façon de mesurer l'espace (mémoire) est problématique si l'on souhaite parler de fonctions $f(n)$ qui croissent moins vite que n , comme $\log n$, car on compte (possiblement) dans la taille l'entrée et la sortie avec les définitions précédentes, et donc au moins n pour l'entrée. Pour éviter ce problème, on convient que l'entrée est accessible en *lecture uniquement*, et la sortie en *écriture uniquement* : l'entrée est codée par des symboles de fonctions statiques ; la sortie est codée par des symboles de fonctions dynamiques particuliers (comme par exemple **out**), et on suppose qu'un emplacement correspondant à ces symboles une fois écrit n'est jamais réécrit. On convient en outre dans la définition $\text{size}(\cdot)$ plus haut de ne pas compter les emplacements dynamiques qui correspondent à ces symboles de sortie (ceux d'entrées étant statiques, ils ne sont pas comptés de toute façon avec les conventions plus haut). C'est la convention qui sera utilisée dans la suite. D'autre part, on évitera d'utiliser des fonctions qui croissent moins vite que $\log n$, pour éviter certains soucis¹. Pour éviter des notations trop compliquées, nous laissons au lecteur le soin de corriger les définitions plus haut pour que ces conventions soient respectées.

¹Comme par exemple que l'espace ne soit pas suffisant pour parcourir l'entrée et mesurer sa longueur en binaire.

9.1.3 Classe PSPACE

Comme dans le chapitre 8, on pourrait observer que les machines de Turing, les automates à $k \geq 2$ -piles, où les algorithmes se simulent deux à deux en espace polynomial : la simulation de l'un par l'autre nécessite un espace qui reste polynomial en l'espace utilisé par le premier.

Cela invite aussi à considérer

Définition 9.6 PSPACE est la classe des langages et des problèmes décidés par un algorithme en espace polynomial. En d'autres termes,

$$\text{PSPACE} = \bigcup_{k \in \mathbb{N}} \text{SPACE}(n^k).$$

Plus généralement, on introduira

Définition 9.7 LOGSPACE est la classe des langages et des problèmes décidés par un algorithme en espace logarithmique. En d'autres termes,

$$\text{LOGSPACE} = \text{SPACE}(\log(n)).$$

9.1.4 Espace et structures non-finies

En fait, mesurer l'espace sur une structure dont l'ensemble de base est \mathbb{N} ou \mathbb{R} est souvent une mauvaise idée.

Sur \mathbb{N} , le problème est que tout algorithme peut se coder par une machine à 2 compteurs (voir le chapitre 5), et donc en espace $\mathcal{O}(1)$ (au prix d'une perte de temps qui peut être exponentielle).

Sur \mathbb{R} , on a le même type de phénomène :

Théorème 9.1 Sur la structure $(\mathbb{R}, 0, 1, +, -, *, =, <)$ tout algorithme peut être réalisé par un algorithme qui fonctionne en espace constant (au prix d'une perte de temps qui peut être exponentielle).

Nous n'en donnerons pas la preuve : on remarquera qu'il faut pour le prouver être capable d'effectuer toutes les opérations sur les réels en entrées en espace constant, ce qui nécessite plus que simplement dire qu'on utilise des machines à deux compteurs.

Pour ces raisons, on se limitera dans ce document aux structures finies dans tout ce qui concerne les discussions relatives à l'espace.

9.1.5 Espace non-déterministe

On peut aussi introduire une notion d'espace non-déterministe.

Définition 9.8 (Classe NPSPACE) Soit M un alphabet fini. Un problème $L \subset M^*$ est dans NPSPACE s'il existe un polynôme $p : \mathbb{N} \rightarrow \mathbb{N}$ et un problème A (appelé vérificateur pour L) tels que pour tout mot w ,

$$w \in L \text{ si et seulement si } \exists u \in M^* \text{ tel que } \langle w, u \rangle \in A,$$

et tels que déterminer si $\langle w, u \rangle \in A$ admette un algorithme en espace $p(|w|)$.

Remarque 9.1 *C'est une définition sur le principe de la définition 8.18 pour le temps non-déterministe. Contrairement à ce qui se passe pour le temps non-déterministe, où l'on pouvait reformuler la définition 8.18 en la définition 8.19 définissant NP à partir de P, ici on ne peut pas reformuler cette définition facilement en définissant NPSpace à partir de PSPACE : le problème est qu'un temps polynomial ne se relie pas directement à la longueur du certificat.*

On peut aussi se convaincre, en utilisant des arguments similaires au chapitre précédent, que cela correspond effectivement à l'espace non-déterministe polynomial pour les machines non-déterministes, en utilisant la notion de machine non-déterministe (avec des instructions **guess**) comme dans le chapitre précédent.

Théorème 9.2 *Soit M un alphabet fini. Un problème $L \subset M^*$ est dans NPSpace si et seulement s'il est décidé en espace non-déterministe polynomial.*

9.2 Relations entre espace et temps

Pour éviter de compliquer inutilement certaines preuves, nous nous limiterons dans cette section à des fonctions $f(n)$ de complexité propre : on suppose que la fonction $f(n)$ est non-décroissante, c'est-à-dire que $f(n+1) \geq f(n)$, et qu'il existe un algorithme qui prend en entrée w et qui produit en sortie $\mathbf{1}^{f(n)}$ en temps $\mathcal{O}(n + f(n))$ et en espace $\mathcal{O}(f(n))$, où $n = |w|$.

Remarque 9.2 *Cela n'est pas vraiment restrictif, car toutes les fonctions usuelles non-décroissantes, comme $\log(n)$, n , n^2 , \dots , $n \log n$, $n!$ vérifient ces propriétés. En outre, ces fonctions sont stables par somme, produit, et exponentielle.*

Remarque 9.3 *En fait, nous avons besoin de cette hypothèse, car la fonction $f(n)$ pourrait ne pas être calculable, et donc il pourrait par exemple ne pas être possible d'écrire un mot de longueur $f(n)$ dans un des algorithmes décrits.*

Remarque 9.4 *Dans la plupart des assertions qui suivent, on peut se passer de cette hypothèse, au prix de quelques complications dans les preuves.*

9.2.1 Relations triviales

Un problème déterministe étant un problème non-déterministe particulier, on a :

Théorème 9.3 $\text{SPACE}(f(n)) \subset \text{NSPACE}(f(n))$.

D'autre part :

Théorème 9.4 $\text{TIME}(f(n)) \subset \text{SPACE}(f(n))$.

Démonstration: Un algorithme écrit au plus un nombre fini d'emplacements (mémoire) à chaque étape. L'espace mémoire utilisé reste donc linéaire en le temps utilisé. Rappelons que l'on ne compte pas l'entrée dans l'espace mémoire. \square

9.2.2 Temps non-déterministe vs déterministe

De façon plus intéressante :

Théorème 9.5 *Pour tout langage de $\text{NTIME}(f(n))$, il existe un entier c tel que ce langage soit dans $\text{TIME}(c^{f(n)})$. Si l'on préfère :*

$$\text{NTIME}(f(n)) \subset \bigcup_{c \in \mathbb{N}} \text{TIME}(c^{f(n)})$$

Démonstration: Soit L un problème de $\text{NTIME}(f(n))$. En utilisant le théorème 8.8, on sait qu'il existe un problème A tel que pour déterminer si un mot w de longueur n est dans L , il suffit de savoir s'il existe $u \in M^*$ avec $\langle w, u \rangle \in A$, ce dernier test pouvant se faire en temps $f(n)$, où $n = |w|$. Puisqu'en temps $f(n)$ on ne peut pas lire plus que $f(n)$ lettres de u , on peut se limiter aux mots u de longueur $f(n)$. Tester si $\langle w, u \rangle \in A$ pour tous les mots $u \in M^*$ de longueur $f(n)$ se fait facilement en temps $\mathcal{O}(c^{f(n)}) + \mathcal{O}(f(n)) = \mathcal{O}(c^{f(n)})$, où $c > 1$ est le cardinal de M : tester tous les mots u peut par exemple se faire en comptant en base c . \square

Remarque 9.5 *Pour écrire le premier u à tester de longueur $f(n)$, nous utilisons le fait que cela doit être possible : c'est le cas, si l'on suppose $f(n)$ de complexité propre. On voit donc l'intérêt ici de cette hypothèse implicite. Nous éviterons de discuter ce type de problèmes dans la suite, qui ne se posent pas de toute façon pour les fonctions $f(n)$ usuelles.*

9.2.3 Temps non-déterministe vs espace

Théorème 9.6 $\text{NTIME}(f(n)) \subset \text{SPACE}(f(n))$.

Démonstration: On utilise exactement le même principe que dans la preuve précédente, si ce n'est que l'on parle d'espace. Soit L un problème de $\text{NTIME}(f(n))$. En utilisant le théorème 8.8, on sait qu'il existe un problème A tel que pour déterminer si un mot w de longueur n est dans L , il suffit de savoir s'il existe $u \in M^*$ de longueur $f(n)$ avec $\langle w, u \rangle \in A$: on utilise un espace $\mathcal{O}(f(n))$ pour générer un à un les mots $u \in M^*$ de longueur $f(n)$ (par exemple en comptant en base c) puis on teste pour chacun si $\langle w, u \rangle \in A$, ce dernier test se faisant en temps $f(n)$, donc espace $f(n)$. Le même espace pouvant être utilisé pour chacun des mots u , au total cela se fait au total un espace $\mathcal{O}(f(n))$ pour générer les $u + \mathcal{O}(f(n))$ pour les tests, soit $\mathcal{O}(f(n))$. \square

9.2.4 Espace non-déterministe vs temps

Le problème de décision REACH, déjà mentionné dans le chapitre précédent, jouera un rôle important : on se donne un graphe orienté $G = (V, E)$, deux sommets u et v , et on cherche à décider dans ce problème s'il existe un chemin entre u et v . Nous avons vu dans le chapitre précédent que REACH est dans P.

Selon la discussion du chapitre 4 (le postulat 1) à tout algorithme A sur la structure $\mathfrak{M} = (M, f_1, \dots, f_u, r_1, \dots, r_v)$ est associé un système de transition, dont les sommets sont appelés des états, ou encore des configurations, et les arcs correspondent à la fonction d'évolution en un pas de A . Dans le cas d'un algorithme non-déterministe, la fonction d'évolution n'est plus une fonction, mais une relation qui précise quelles sont les évolutions possibles en un pas. Ce système de transition, dans les deux cas peut être vu comme un graphe, dont les sommets sont les configurations, et les arcs correspondent à la fonction d'évolution en un pas.

Selon le lemme 7.3, chaque configuration X peut se décrire par un mot $[X]$ sur l'alphabet M longueur $\mathcal{O}(\text{size}(X))$ qui en décrit les emplacements utiles : si on fixe l'entrée w de longueur n , pour un calcul en espace $f(n)$, il y a donc moins de $\mathcal{O}(c^{f(n)})$ sommets dans ce graphe G_w , où $c > 1$ est le cardinal de l'alphabet M .

D'autre part, une très légère modification du lemme 7.4 montre qu'on peut construire un circuit C_w de taille $\mathcal{O}(f(n))$ tel que $C_w([X], [X']) = 1$ si et seulement si X' est une configuration successeur de la configuration X : les arcs de ce graphe sont donc donnés par un circuit C_w que l'on peut facilement déterminer.

Un mot w est accepté par l'algorithme A si et seulement s'il y a un chemin dans ce graphe G_w entre l'état initial $X[w]$ codant l'entrée w , et un état acceptant. On peut supposer sans perte de généralité qu'il y a un unique état acceptant X^* . Décider l'appartenance d'un mot w au langage reconnu par A est donc résoudre le problème REACH sur $\langle G_w, X[w], X^* \rangle$.

On va alors traduire sous différentes formes tout ce que l'on sait sur le problème REACH. Tout d'abord, il est clair que le problème REACH se résout par exemple en temps et espace $\mathcal{O}(n^2)$, où n est le nombre de sommets, par un parcours en profondeur.

On en déduit :

Théorème 9.7 *Si $f(n) \geq \log n$, alors*

$$\text{NSPACE}(f(n)) \subset \bigcup_{c \in \mathbb{N}} \text{TIME}(c^{f(n)}).$$

Démonstration: Soit L un problème de $\text{NSPACE}(f(n))$ reconnu par l'algorithme non-déterministe A . Par la discussion plus haut, on peut déterminer si $w \in L$ en résolvant le problème REACH sur $\langle G_w, X[w], X^* \rangle$: on a dit que cela pouvait se faire en temps quadratique en le nombre de sommets, soit en temps $\mathcal{O}(c^{2\mathcal{O}(f(n))})$, où $c > 1$ est le cardinal de l'alphabet M . □

9.2.5 Espace non-déterministe vs espace déterministe

On va maintenant affirmer que REACH se résout en espace $\log^2(n)$.

Proposition 9.1 $\text{REACH} \in \text{SPACE}(\log^2 n)$.

Démonstration: Soit $G = (V, E)$ le graphe orienté en entrée. Étant donnés deux sommets x et y de ce graphe, et i un entier, on note $CHEMIN(x, y, i)$ si et seulement si il y a un chemin de longueur inférieure à 2^i entre x et y . On a $\langle G, u, v \rangle \in REACH$ si et seulement si $CHEMIN(u, v, \log(n))$, où n est le nombre de sommets. Il suffit donc de savoir décider la relation $CHEMIN$ pour décider $REACH$.

L'astuce est de calculer $CHEMIN(x, y, i)$ récursivement en observant que l'on a $CHEMIN(x, y, i)$ si et seulement si il existe un sommet intermédiaire z tel que $CHEMIN(x, z, i-1)$ et $CHEMIN(z, y, i-1)$. On teste alors à chaque niveau de la récursion chaque sommet possible z .

Pour représenter chaque sommet, il faut $\mathcal{O}(\log(n))$ bits. Pour représenter x, y, i , il faut donc $\mathcal{O}(\log(n))$ bits. Cela donne une récurrence de profondeur $\log(n)$, chaque étape de la récurrence nécessitant uniquement de stocker un triplet x, y, i et de tester chaque z de longueur $\mathcal{O}(\log(n))$. Au total, on utilise donc un espace $\mathcal{O}(\log(n)) * \mathcal{O}(\log(n)) = \mathcal{O}(\log^2(n))$. \square

Théorème 9.8 (Savitch) Si $f(n) \geq \log(n)$, alors

$$NSPACE(f(n)) \subset SPACE(f(n)^2).$$

Démonstration: On utilise cette fois l'algorithme précédent pour déterminer s'il y a un chemin dans le graphe G_w entre $X[w]$ et X^* .

On remarquera que l'on a pas besoin de construire explicitement le graphe G_w mais que l'on peut utiliser l'algorithme précédent à la volée : plutôt que d'écrire complètement le graphe G_w , et ensuite de travailler en lisant dans cette écriture du graphe à chaque fois s'il y a un arc entre un sommet X et un sommet X' , on peut de façon paresseuse, déterminer à chaque fois que l'on fait un test si $C_w(X, X') = 1$. \square

Corollaire 9.1 $PSPACE = NSPACE$

Principe: On a $\bigcup_{c \in \mathbb{N}} SPACE(n^c) \subset \bigcup_{c \in \mathbb{N}} NSPACE(n^c)$, par le théorème 9.3, et $\bigcup_{c \in \mathbb{N}} NSPACE(n^c) \subset \bigcup_{c \in \mathbb{N}} SPACE(n^{2c}) \subset \bigcup_{c \in \mathbb{N}} SPACE(n^c)$ par le théorème précédent. \square

9.2.6 Espace logarithmique non-déterministe

En fait, on peut encore dire plus sur le problème $REACH$.

Définition 9.9 On note

$$NLOGSPACE = NSPACE(\log(n)).$$

Théorème 9.9 $REACH \in NLOGSPACE$.

Démonstration: Pour déterminer s'il y a un chemin entre u et v dans un graphe G , on devine le chemin arc par arc. Cela nécessite uniquement de garder le sommet que l'on est en train de visiter en plus de u et de v . Chaque sommet se codant par $\mathcal{O}(\log(n))$ bits, l'algorithme est en espace $\mathcal{O}(\log(n))$. \square

Définition 9.10 Soit \mathcal{C} une classe de complexité. On note $\text{co-}\mathcal{C}$ pour la classe des langages dont le complémentaire est dans la classe \mathcal{C} .

On parle ainsi de problème coNP , coNLOGSPACE , etc. . .

En fait, on peut montrer :

Théorème 9.10 Le problème REACH est aussi dans coNLOGSPACE .

Démonstration:

Supposons que l'on veuille savoir si t est accessible à partir d'un sommet s dans un graphe G à n sommets.

L'idée est d'arriver à calculer le nombre de sommets accessibles à partir d'un sommet s dans un graphe G . En effet, supposons que l'on connaisse ce nombre c : on peut alors prendre sommet par sommet, et deviner s'il est accessible ou non. Lorsqu'un sommet u est deviné comme accessible, l'algorithme vérifie ce fait en devinant un chemin de longueur au plus $n - 1$ entre s et ce sommet u . Si l'algorithme échoue dans cette étape de deviner un chemin, l'algorithme termine en rejetant. L'algorithme compte alors le nombre d de sommets qui ont été vérifiés comme atteignables. Si d n'est pas c , l'algorithme refuse.

En d'autres termes, si l'algorithme devine correctement c sommets u comme atteignables à partir de s différent de t et arrive à se convaincre que chacun d'entre eux est atteignable en devinant un chemin entre s et chaque sommet u , l'algorithme peut être sûr que tout autre sommet n'est pas atteignable, et donc il peut accepter.

Cela donne un programme du type :

```

d ← 0
for u sommet de G do
  test ← guess x ∈ {0, 1}
  if test then
    [ suivre de façon non déterministe un chemin
    de longueur n - 1 à partir de s et si aucun
    sommet rencontré sur ce chemin est u rejeter ]
    if u = t then rejeter
    d ← d + 1
  endif
endfor
if d ≠ c then rejeter
else accepter

```

Il reste à se convaincre que l'on peut calculer c : l'idée est de calculer par récurrence le nombre de sommets du graphe c_i qui sont atteignables à partir de s par un chemin de longueur inférieure ou égale à i . Bien entendu, $c = c_{n-1}$. Le calcul de c_{i+1} se fait à partir de c_i selon le même principe que celui évoqué plus haut.

Concrètement, cela donne au final un algorithme comme celui là :

```

 $c_0 \leftarrow 1$ 
for  $i \leftarrow 0$  to  $n - 2$ 
   $c_{i+1} \leftarrow 0$ 
   $d \leftarrow 0$ 
  for  $v$  sommet de  $G$  do
    for  $u$  sommet de  $G$  do
       $test \leftarrow \text{guess } x \in \{0,1\}$ 
      if  $test$  then
        [ suivre de façon non déterministe un chemin
          de longueur  $i$  à partir de  $s$  et si aucun
          sommet rencontré est  $u$  rejeter ]
         $d \leftarrow d + 1$ 
        if  $(u,v)$  est un arc de  $G$  then
           $c_{i+1} \leftarrow c_{i+1} + 1$ 
          [ sortir de la boucle for  $u$  sommet de  $G$ 
            pour passer au prochain  $v$  ]
        endif
      endif
    endfor
    if  $d \neq c_i$  then rejeter
  endfor
 $d \leftarrow 0$ 
for  $u$  sommet de  $G$  do
   $test \leftarrow \text{guess } x \in \{0,1\}$ 
  if  $test$  then
    [ suivre de façon non déterministe un chemin
      de longueur  $n - 1$  à partir de  $s$  et si aucun
      sommet rencontré est  $u$  refuser ]
    if  $u = t$  then rejeter
     $d \leftarrow d + 1$ 
  endif
endfor
if  $d \neq c_{n-1}$  then rejeter
else accepter

```

L'algorithme obtenu utilise bien un espace $\mathcal{O}(\log(n))$.

□

On en déduit :

Théorème 9.11 $\text{NLOGSPACE} = \text{coNLOGSPACE}$.

Démonstration: Il suffit de montrer que $\text{coNLOGSPACE} \subset \text{NLOGSPACE}$. L'inclusion inverse en découle car un langage L de NLOGSPACE aura son complémentaire dans coNLOGSPACE et donc aussi dans NLOGSPACE , d'où l'on déduit que L sera dans coNLOGSPACE .

Maintenant, pour décider si un mot w doit être accepté par un langage de coNLOGSPACE , on peut utiliser l'algorithme non-déterministe précédent qui utilise un espace logarithmique pour déterminer s'il existe un chemin entre $X[w]$ et X^* dans le graphe G_w . \square

En fait, selon le même principe on peut montrer plus généralement.

Théorème 9.12 *Soit $f(n)$ une fonction telle que $f(n) \geq \log(n)$. Alors*

$$\text{NSPACE}(f(n)) = \text{coNSPACE}(f(n)).$$

9.3 Quelques résultats & résumé

9.3.1 Théorèmes de hiérarchie

On dit qu'une fonction $f(n) \geq \log(n)$ est *constructible en espace*, si la fonction qui envoie 1^n sur la représentation binaire de $1^{f(n)}$ est calculable en espace $\mathcal{O}(f(n))$.

La plupart des fonctions usuelles sont constructibles en espace. Par exemple, n^2 est constructible en espace puisqu'un algorithme peut obtenir n en binaire en comptant le nombre de 1 , et écrire n^2 en binaire par n'importe quelle méthode pour multiplier n par lui-même. L'espace utilisé est certainement en $\mathcal{O}(n^2)$.

Théorème 9.13 (Théorème de hiérarchie) *Pour toute fonction $f : \mathbb{N} \rightarrow \mathbb{N}$ constructible en espace, il existe un langage L qui est décidable en espace $\mathcal{O}(f(n))$ mais pas en espace $o(f(n))$.*

Démonstration: On considère le langage (très artificiel) L qui est décidé par l'algorithme B suivant :

- sur une entrée w de taille n , B calcule $f(n)$ en utilisant la constructibilité en espace de f , et réserve un espace $f(n)$ pour la simulation qui va venir.
- Si w n'est pas de la forme $\langle A \rangle \mathbf{10}^*$, pour un certain algorithme A , alors l'algorithme B rejette.
- Sinon, B simule l'algorithme A sur le mot w pendant au plus $2^{f(n)}$ étapes pour déterminer si A accepte en ce temps avec un espace inférieur à $f(n)$.
 - si A accepte en ce temps et cet espace, alors B rejette ;
 - sinon B accepte.

Par construction, L est dans $\text{SPACE}(f(n))$, car la simulation n'introduit qu'un facteur constant dans l'espace nécessaire : plus concrètement, si A utilise un espace $g(n) \leq f(n)$, alors B utilise un espace au plus $dg(n)$ pour une constante d .

Supposons que L soit décidé par un algorithme A en espace $g(n)$ avec $g(n) = o(f(n))$. Il doit exister un entier n_0 tel que pour $n \geq n_0$, on ait $dg(n) < f(n)$. Par conséquent, la simulation par B de A sera bien complète sur une entrée de longueur n_0 ou plus.

Considérons ce qui se passe lorsque B est lancé sur l'entrée $\langle A \rangle \mathbf{10}^{n_0}$. Puisque cette entrée est de taille plus grande que n_0 , B répond l'inverse de

l'algorithme A sur la même entrée. Donc B et A ne décident pas le même langage, et donc l'algorithme A ne décide pas L , ce qui mène à une contradiction.

Par conséquent L n'est pas décidable en espace $o(f(n))$. \square

Autrement dit :

Théorème 9.14 (Théorème de hiérarchie) *Soient $f, f' : \mathbb{N} \rightarrow \mathbb{N}$ des fonctions constructibles en espace telles que $f(n) = o(f'(n))$. Alors l'inclusion $\text{SPACE}(f) \subset \text{SPACE}(f')$ est stricte.*

Sur le même principe, on peut prouver :

Théorème 9.15 (Théorème de hiérarchie) *Soient $f, f' : \mathbb{N} \rightarrow \mathbb{N}$ des fonctions constructibles en espace telles que $f(n) = o(f'(n))$. Alors l'inclusion $\text{NSPACE}(f) \subset \text{NSPACE}(f')$ est stricte.*

On en déduit :

Théorème 9.16 $\text{NLOGSPACE} \subsetneq \text{PSPACE}$.

Démonstration: La classe NLOGSPACE est complètement incluse dans $\text{SPACE}(\log^2 n)$ par le théorème de Savitch. Hors cette dernière est un sous-ensemble strict de $\text{SPACE}(n)$, qui est inclus dans PSPACE . \square

Sur le même principe, on obtient :

Définition 9.11 *Soit*

$$\text{EXPSPACE} = \bigcup_{c \in \mathbb{N}} \text{SPACE}(2^{n^c}).$$

Théorème 9.17 $\text{PSPACE} \subsetneq \text{EXPSPACE}$.

Démonstration: La classe PSPACE est complètement incluse dans, par exemple, $\text{SPACE}(n^{\log(n)})$. Hors cette dernière est un sous-ensemble strict de $\text{SPACE}(2^n)$, qui est inclus dans EXPSPACE . \square

9.3.2 Classes de complexité usuelles

Les classes de complexité suivantes sont les classes les plus usuelles :

Définition 9.12

$$\text{LOGSPACE} = \text{SPACE}(\log n)$$

$$\text{NLOGSPACE} = \text{NSPACE}(\log n)$$

$$\text{P} = \bigcup_{c \in \mathbb{N}} \text{TIME}(n^c)$$

$$\text{NP} = \bigcup_{c \in \mathbb{N}} \text{NTIME}(n^c)$$

$$\text{PSPACE} = \bigcup_{c \in \mathbb{N}} \text{SPACE}(n^c)$$

9.3.3 Relations entre classes

On a :

Théorème 9.18 $\text{LOGSPACE} \subset \text{NLOGSPACE} \subset \text{P} \subset \text{NP} \subset \text{PSPACE}$.

Démonstration: Chaque inclusion est une instance des résultats précédents. \square

On a vu que $\text{NLOGSPACE} \subsetneq \text{PSPACE}$ mais on ne sait pas lesquelles des des inclusions strictes intermédiaires sont vraies.

Théorème 9.19 *Les classes déterministes sont closes par complément.*

Démonstration: Inverser l'état d'acceptation et de rejet dans un algorithme qui décide le langage. \square

9.4 Problèmes complets

Nous allons maintenant montrer que les classes précédentes admettent des problèmes complets.

La notion de réduction, basée sur le temps polynomial, ne s'avère pas pertinente pour parler de problèmes complets pour des classes incluses dans P. Aussi, nous allons introduire une notion de réduction basée sur l'espace logarithmique.

9.4.1 Réduction en espace logarithmique

Nous commençons par introduire la notion de fonction calculable en espace logarithmique :

Définition 9.13 (Fonction calculable en espace logarithmique) *Soient M et N deux alphabets. Une fonction $f : M^* \rightarrow N^*$ est calculable en espace logarithmique s'il existe un algorithme A , tel que pour tout mot w , A avec l'entrée w termine en utilisant un espace $\mathcal{O}(\log(n))$, où $n = |w|$, avec le résultat $f(w)$. On utilise toujours la convention que l'on ne compte pas l'entrée, w qui est en lecture seulement, et la sortie $f(w)$ qui est en écriture seulement dans l'espace utilisé.*

En fait, il s'agit d'une contrainte plus forte que d'être calculable en temps polynomial.

Proposition 9.2 *Une fonction calculable en espace logarithmique est calculable en temps polynomial.*

Démonstration: Utiliser le principe de la preuve du théorème 9.7. \square

Le résultat suivant est vrai (mais la preuve n'est pas directe contrairement au résultat similaire pour les fonctions calculables en temps polynomial. La difficulté est que le résultat intermédiaire de la première fonction ne peut pas être stocké en mémoire car sa taille n'est pas nécessairement logarithmique).

Proposition 9.3 (Stabilité par composition) *La composée de deux fonctions calculables en espace logarithmique est calculable en espace logarithmique.*

Démonstration: Pour calculer la composition de f et de g calculables en espace logarithmique, on ne peut pas, sur une entrée w , calculer $w' = f(w)$, puis $g(w')$ car $w' = f(w)$ peut être de taille trop importante. Cependant, chaque pas de calcul de $g(w')$ nécessite au plus de lire une lettre de w' , disons la lettre numéro i de w' .

Or, étant donné le numéro i de cette lettre en binaire, on peut déterminer la valeur de cette lettre en espace logarithmique : simuler le calcul de f tout en maintenant un compteur qui compte le nombre de symboles j écrits jusque là, jusqu'à ce que $j = i$.

En maintenant la valeur de i , on simule le calcul de g sur $w' = f(w)$ sans jamais écrire complètement w' : à chaque fois que l'on a besoin de lire une nouvelle lettre de w' , on utilise la procédure ci-dessus pour déterminer la valeur de cette lettre. \square

On peut alors introduire :

Définition 9.14 (Réduction) *Soient A et B deux problèmes d'alphabet respectifs M_A et M_B , et de langages respectifs M_A et M_B . Une réduction log-space de A vers B , ou encore réduction en espace logarithmique, est une fonction $f : M_A^* \rightarrow M_B^*$ calculable en espace logarithmique telle que $w \in L_A$ ssi $f(w) \in L_B$. On note $A \leq_L B$ lorsque A se réduit à B de cette façon.*

Il découle de la proposition 9.2 que cette notion de réduction est plus contraignante que la précédente.

Corollaire 9.2 *Soient A et B deux problèmes. Supposons $A \leq_L B$. Alors $A \leq B$.*

Exactement pour les mêmes raisons qu'auparavant, on a :

Théorème 9.20 \leq_L est un préordre :

1. $L \leq_L L$
2. $L_1 \leq_L L_2, L_2 \leq_L L_3$ implique $L_1 \leq_L L_3$.

Définition 9.15 *Deux problèmes L_1 et L_2 sont équivalents pour \leq_L , noté $L_1 \equiv_L L_2$, si $L_1 \leq_L L_2$ et si $L_2 \leq_L L_1$.*

et

Proposition 9.4 (Réduction) *Si $A \leq_L B$, et si $B \in P$ alors $A \in P$.*

Proposition 9.5 (Réduction) *Si $A \leq_L B$, et si $A \notin P$ alors $B \notin P$.*

Théorème 9.21 *Les mêmes résultats sont vrais si on remplace P par les classes de complexité NLOGSPACE, NP, PSPACE dans les propositions précédentes.*

Définition 9.16 (C-complétude) Soit \mathcal{C} une classe de langages. Un problème A est dit \mathcal{C} -complet pour \leq_L si

1. il est dans la classe \mathcal{C}
2. tout autre problème B de la classe \mathcal{C} est tel que $B \leq_L A$.

Corollaire 9.3 Soit \mathcal{C} une classe de langages. Tous les problèmes \mathcal{C} -complets sont équivalents au sens de \equiv_L .

9.4.2 Retour sur la NP-complétude

Dans le chapitre précédent nous avons montré la NP-complétude d'un certain nombre de problèmes, pour la relation \leq basée sur le temps polynomial. On observera que dans chacune des preuves, la fonction de réduction utilisée est en fait calculable non seulement en temps polynomial mais en espace logarithmique.

Autrement dit :

Théorème 9.22 Chacun des problèmes NP-complets évoqués dans le chapitre précédent sont en fait NP-complets pour \leq_L et pas seulement pour \leq .

Lorsqu'on parlera de complétude dans la suite, on parlera toujours de complétude pour \leq_L .

9.4.3 Un problème PSPACE-complet

Comme pour la NP-complétude, en jouant avec les définitions, on peut prouver assez facilement l'existence d'un problème PSPACE-complet.

Théorème 9.23 Le problème

$$K = \{ \langle A, w, \mathbf{1}^n \rangle \mid \text{l'algorithme } A \text{ accepte l'entrée } w \text{ en espace } n \}$$

est PSPACE-complet.

Démonstration: Le problème est dans PSPACE car sur une entrée $\langle A, w, \mathbf{1}^n \rangle$ il suffit de simuler l'algorithme A et de vérifier qu'il accepte w en espace n , ce qui se fait en espace linéaire.

Soit L un problème de PSPACE. L est accepté par un algorithme en espace $p(n)$ pour un certain polynôme p . L se réduit à K par la fonction qui à un mot w associe le triplet $\langle A, w, \mathbf{1}^{p(|w|)} \rangle$. Cette fonction est bien calculable en espace logarithmique en la longueur de w . \square

Cependant, le problème précédent n'est pas très naturel. Comme dans le chapitre précédent, nous allons parler de problèmes plus naturels en utilisant l'équivalence entre algorithmes et circuits.

Peut-être que le problème PSPACE-complet le plus fondamental est celui de la satisfiabilité quantifiée d'un circuit booléen, noté QCIRCUITSAT : étant

donné un circuit booléen $C(x_1, \dots, x_n)$ à n variables, peut-on déterminer s'il existe une valeur pour x_1 telle que pour toute valeur de x_2 , il existe une valeur pour la variable x_3 telle que pour toute valeur de x_4 il existe une valeur pour la variable x_5 telle que toute valeur de $x_6 \dots$ etc, telles que $C(x_1, x_2, \dots, x_n) = 1$?

En d'autres termes, $\exists x_1 \forall x_2 \exists x_3 \dots C(x_1, \dots, x_n) = 1$?

Les quantificateurs de rang impairs sont existentiels, et ceux de rang pair sont universels.

Remarque 9.6 *En fait, pour s'assurer d'une alternance stricte de quantificateurs de ce type, devant une suite de quantificateurs qui ne serait pas exactement de ce type, on peut toujours insérer des variables "stupides" qui n'interviennent pas dans le circuit C pour s'y ramener. Par exemple $\forall x_1 \forall x_2 \exists x_3 C(x_1, x_2, x_3) = 1$ peut se réécrire $\exists x \forall x_1 \exists y \forall x_2 \exists x_3 C(x_1, x_2, x_3) = 1$.*

Théorème 9.24 *Le problème QCIRCUITSAT est PSPACE-complet.*

Démonstration: Le problème QCIRCUITSAT est dans PSPACE car il est facile de construire un algorithme récursif qui le résout : pour une formule de la forme $\exists x_1 \phi$ (respectivement : $\forall x_1 \phi$) on fixe la valeur de x_1 à 0, on propage cette valeur pour x_1 dans ϕ pour obtenir $\phi_{x_1=0}$, et on s'appelle récursivement pour savoir si la formule $\phi_{x_1=0}$ est satisfiable. On fixe alors la valeur de x_1 à 1, on propage cette valeur pour x_1 dans ϕ pour obtenir $\phi_{x_1=1}$, et on s'appelle récursivement pour savoir si la formule $\phi_{x_1=1}$ est satisfiable. On accepte si et seulement si $\phi_{x_1=0}$ ou (resp. et) $\phi_{x_1=1}$ sont satisfiables. L'algorithme fonctionne avec $\mathcal{O}(n)$ appels récursifs, chacun utilisant un espace constant, et donc en espace total $\mathcal{O}(n)$, où n est la taille de la formule.

Pour montrer qu'il est complet, on va utiliser le principe de la preuve du théorème de Savitch. Supposons que le problème L soit accepté en espace polynomial. On va écrire par un circuit quantifié $C_i([X], [X'])$ le fait qu'il existe un chemin de longueur inférieure à 2^i dans le graphe G_w entre les configurations X et X' . Le graphe G_w est de taille $2^{p(n)}$ pour un certain polynôme p , où $n = |w|$.

Ce circuit s'obtient par récurrence sur i . Pour $i = 0$, c'est le circuit qui donne les arcs du graphe G_w . Pour $i > 0$, on a envie d'écrire $C_i([X], [X'])$ comme

$$\exists [X''] C_{i-1}([X], [X'']) \wedge C_{i-1}([X''], [X']).$$

Cependant si l'on fait ainsi, le circuit obtenu pour C_i est au moins de taille double de celui de C_{i-1} , et donc C_i sera de taille exponentielle en i .

L'idée est de quantifier en réutilisant l'espace comme dans la preuve du théorème de Savitch : on écrit $C_i([X], [X'])$ comme

$$\exists [X''] \forall [D_1] \forall [D_2] C'_i([X], [X'], [D_1], [D_2], [X''])$$

où $C'_i([X], [X'], [D_1], [D_2], [X''])$ teste la relation

$$(([D_1] = [X] \wedge [D_2] = [X'']) \vee ([D_1] = [X''] \wedge [D_2] = [X'])) \Rightarrow C_{i-1}([D_1], [D_2]).$$

Cette fois la taille de C_i sera de l'ordre de celle de C_{i-1} plus $\mathcal{O}(p(n))$. Un mot w est dans le langage L si et seulement si $C_{p(n)}([X[w]], [X^*])$.

La fonction qui à w associe le circuit quantifié $C_{p(n)}([X[w]], [X^*])$ est bien calculable en espace logarithmique. \square

En utilisant le fait que la satisfiabilité d'un circuit peut se ramener à celui d'une formule du calcul propositionnel en forme normale conjonctive (c'est-à-dire en introduisant une variable booléenne par porte du circuit comme dans la preuve du théorème 8.21) on obtient :

Le problème QSAT (auss appelé QBF) consiste étant donnée une formule du calcul propositionnel en forme normale conjonctive ϕ avec les variables x_1, x_2, \dots, x_n (c'est-à-dire un instance de SAT) à déterminer si $\exists x_1 \forall x_2 \exists x_3 \dots \phi(x_1, \dots, x_n)$?

Théorème 9.25 *Le problème QSAT est PSPACE-complet.*

Les jeux stratégiques sur les graphes donnent naturellement naissance à des problèmes PSPACE-complet. Par exemple.

Le jeu GEOGRAPHY consiste à se donner un graphe orienté $G = (V, E)$. Le joueur 1 choisit un sommet u_1 du graphe. Le joueur 2 doit alors choisir un sommet v_1 tel qu'il y ait un arc de u_1 vers v_1 . C'est alors au joueur 1 de choisir un autre sommet u_2 tel qu'il y ait un arc de v_1 vers u_2 , et ainsi de suite. On a pas le droit de repasser deux fois par le même sommet. Le premier joueur qui ne peut pas continuer le chemin $u_1 v_1 u_2 v_2 \dots$ perd. Le problème GEOGRAPHY consiste à déterminer étant donné un graphe G et un sommet de départ pour le joueur 1, s'il existe une stratégie gagnante pour le joueur 1.

Théorème 9.26 *Le problème GEOGRAPHY est PSPACE-complet.*

9.4.4 Un problème P complet

Rappelons que le problème CIRCUITVALUE consiste, étant donné un circuit $C(x_1, \dots, x_n)$, et un ensemble de valeurs booléennes pour ses variables $\bar{x} \in \{0, 1\}^n$, à déterminer si $C(\bar{x}) = 1$.

Théorème 9.27 *Le problème CIRCUITVALUE est P-complet.*

Démonstration: On a déjà vu dans le chapitre précédent que le problème CIRCUITVALUE était dans P.

Soit L un langage de P . On sait qu'il existe une famille de circuits de taille polynomiale $(C_n)_{n \in \mathbb{N}}$ qui reconnaît L (Théorème 8.2). L se réduit au problème CIRCUITVALUE par la fonction qui à un mot w associe $\langle \langle C_{|w|} \rangle, w \rangle$, en observant que la preuve du théorème 8.2 ne montre pas seulement que la fonction qui à l'entier n associe la description $\langle C_n \rangle$ du circuit C_n est calculable en temps polynomial, mais aussi que cette fonction est calculable en espace logarithmique. \square

On peut en déduire que d'autres problèmes sont P -complets. Tout d'abord, comme d'habitude on peut exprimer l'équivalence entre circuits et formules du calcul propositionnel en forme normale conjonctive.

Le problème SATVALUE consiste, étant donné une formule du calcul propositionnel en forme normale conjonctive ϕ avec les variables x_1, x_2, \dots, x_n (c'est-à-dire un instance de SAT), et un vecteur de valeurs $\bar{x} \in \{0, 1\}^n$, à déterminer la valeur de $\phi(\bar{x})$.

Théorème 9.28 *Le problème SATVALUE est P-complet.*

Démonstration: Le problème est clairement dans P. D'autre part, on peut transformer tout circuit en une formule équivalente du calcul propositionnel en forme normale conjonctive en introduisant une variable par porte du circuit comme dans la preuve du théorème 8.21, le tout en espace logarithmique. \square

En fait, on peut aussi prouver.

Le problème MONOTONECIRCUITVALUE consiste, étant donné un circuit booléen $C(x_1, \dots, x_n)$ sans porte \neg et un ensemble de valeurs booléennes pour ses variables $\bar{x} \in \{0, 1\}^n$, à déterminer si $C(\bar{x}) = 1$.

Théorème 9.29 *Le problème MONOTONECIRCUITVALUE est P-complet.*

9.4.5 Un problème NLOGSPACE-complet

Théorème 9.30 *Le problème REACH est NLOGSPACE-complet.*

Démonstration: Nous avons déjà vu que le problème REACH était dans NLOGSPACE.

Montrons qu'il est complet. Soit L un langage de NLOGSPACE. L se réduit au problème REACH par la fonction qui à un mot w associe l'instance $\langle G_w, X[w], X^* \rangle$: cette fonction est bien calculable en espace logarithmique. \square

Rappelons que le problème 3-SAT est NP-complet. Le problème 2-SAT, c'est-à-dire celui de la satisfiabilité d'une formule du calcul propositionnel en forme normale conjonctive avec 2-littéraux par clause est lui soluble en temps polynomial.

En fait, il est NLOGSPACE-complet :

Théorème 9.31 *Le problème 2-SAT est NLOGSPACE-complet.*

9.4.6 Des problèmes EXPTIME et NEXPTIME-complets

Soit C un circuit tel que chaque porte soit le prédécesseur d'au plus deux portes. Chaque sommet du graphe associé à donc au plus 4-voisins que l'on peut numéroté de 0 à 3. Sa *représentation succincte* consiste à se donner un autre circuit avec plusieurs sorties : sur l'entrée $\langle\langle i \rangle, \langle k \rangle\rangle$, où $\langle i \rangle$ et $\langle k \rangle$ sont les codages en binaire du numéro d'un sommet du graphe du circuit C , et de $k \in \{0, 1, 2, 3\}$, cet autre circuit produit $\langle\langle j \rangle, s \rangle$ où s est l'étiquette de la porte, c'est-à-dire soit $\neg, \vee, \wedge, \mathbf{0}$ ou $\mathbf{1}$ où le numéro d'une variable en binaire, et $\langle j \rangle$ est le codage en binaire du voisin numéro k .

Nous laissons en exercice les résultats suivants.

Le problème SUCCINTSAT consiste étant donné la représentation succincte d'un circuit à déterminer si le circuit correspondant est satisfiable.

Théorème 9.32 *Le problème SUCCINTSAT est NEXPTIME-complet.*

La représentation succincte d'une valuation de variables booléennes consiste à se donner un circuit qui prend en entrée le codage en binaire d'un entier représentant le numéro d'une variable et qui répond sa valeur.

Le problème SUCCINTSATVALUE consiste étant donné la représentation succincte d'un circuit, et la représentation succincte d'une valuation de ses variables, à déterminer si le circuit correspondant vaut 1 sur cette entrée.

Théorème 9.33 *Le problème SUCCINTSATVALUE est EXPTIME-complet.*

9.5 Notes bibliographiques

Ce chapitre contient des résultats classiques en complexité. Nous avons utilisé ici principalement leur présentation dans les ouvrages [Arora and Barak, 2009], [Papadimitriou, 1994], [Sipser, 1997] et [Kozen, 2006]. Les rares considérations sur les structures arbitraires sont tirées de [Poizat, 1995]. En particulier, le théorème 9.1 est dû à [Michaux, 1989].

Bibliographie

- [Arora and Barak, 2009] Arora, S. and Barak, B. (2009). *Computational Complexity : A Modern Approach*. Cambridge University Press.
- [Kozen, 2006] Kozen, D. (2006). *Theory of computation*. Springer-Verlag New York Inc.
- [Michaux, 1989] Michaux, C. (1989). Une remarque à propos des machines sur \mathbb{R} introduites par blum, shub et smale. *Comptes Rendus de l'Académie des Sciences de Paris, Série I*, 309 :435–437.
- [Papadimitriou, 1994] Papadimitriou, C. H. (1994). *Computational Complexity*. Addison-Wesley.
- [Poizat, 1995] Poizat, B. (1995). *Les petits cailloux : Une approche modèle-théorique de l'Algorithmie*. Aléas Editeur.
- [Sipser, 1997] Sipser, M. (1997). *Introduction to the Theory of Computation*. PWS Publishing Company.