Classes de complexité en espace, alternances, et tutti quanti

Jean Goubault-Larrecq

21 février 2019

Résumé

Ceci est la version 10 de la première partie du poly du cours de complexité avancée, datant du 21 février 2019. Elle fait suite la version 9, datant du 02 octobre 2017, la version 8, datant du 20 décembre 2013, la version 7, datant du 28 octobre 2013, la version 6, datant du 01 octobre 2012, la version 5, datant du 26 octobre 2011, la version 4, datant du 05 octobre 2010, à la version 3, datant du 11 juillet 2009, à la version 2, datant du 27 septembre 2007, et à la version 1, datant du 24 septembre 2007. L'astuce du graphe G' du théorème 2.16 a été suggérée par un élève de l'ENS Cachan ; malheureusement, je ne me souviens plus qui, probablement François Bobot. De nombreuses améliorations ont été suggérées par Michaël Monerau et Marc Bagnol en 2009, par Guillaume Scerri en 2011, par Nathan Grosshans en 2013, par Paul-Nicolas Madelaine en 2019. Une erreur a été trouvée par Raphaël Bonaque en 2010, une autre par Laurent Feuilloley en 2012. Une erreur d'attribution a été découverte par Nicolas Blanchard en 2013, et un nom mal orthographié par Philippe Schnoebelen en 2017.

Table des matières

1 Classes en espace, NL, PSPACE **2**

2 Classes en espace **2**
2.1 La classe **NL**, et le problème d'accessibilité dans les graphes orientés 3
2.2 Le théorème de Savitch 6
2.3 Réductibilité logspace, **NL**-complétude 10
2.4 Le théorème d'Immerman-Szelepcsényi 11
2.5 **PSPACE** et QBF 15

3 Machines alternantes **19**
3.1 Le théorème de Chandra-Kozen-Stockmeyer I : temps alternant=espace . . . 22
3.2 Circuits et clauses de Horn 25
3.3 Le théorème de Chandra-Kozen-Stockmeyer II : espace alternant=exp(temps) 28

4 La hiérarchie polynomiale **32**

Ce document est librement inspiré, d'une part, du livre de Christos Papadimitriou [1], d'autre part du poly de Ralf Treinen [3].

1 Classes en espace, NL, PSPACE

Si les classes **P** (temps polynomial déterministe) et **NP** (temps polynomial non-déterministe) sont classiques, il est fructueux de définir aussi des classes dans lesquelles c'est la consommation mémoire (l'*espace*) qui est bornée.

Une de ces classes importantes est **PSPACE**, la classe des langages décidables en espace polynomial. Une autre est **L**, la classe des langages décidables en espace déterministe $O(\log n)$ (logarithmique), une autre encore est **NL**, la classe des langages décidables en espace non déterministe logarithmique. (Nous réserverons usuellement la notation x pour l'entrée, et n pour sa taille.)

2 Classes en espace

Attention ! Nous demanderons explicitement que les machines en espace borné terminent. C'est automatique pour les machines en temps borné... mais la machine qui boucle en ne faisant rien n'utilise qu'un espace constant.

Définition 2.1 (SPACE) *Soit f une fonction de $\mathbb{N}$ dans $\mathbb{N}$. La classe* $\mathbf{SPACE}(f(n))$ *(resp.,* $\mathbf{NSPACE}(f(n))$*) est par définition la classe des langages L tels qu'il existe une machine de Turing (resp. non déterministe) $\mathcal{M}$ qui termine sur toute entrée x, qui utilise un espace majoré par $f(n)$, où n est la taille de x, et qui accepte si et seulement si $x \in L$.*

Il est sans doute besoin de préciser ce que nous entendons par machine de Turing : une machine de Turing ayant un nombre fixé $k+2$ de bandes ($k \in \mathbb{N}$), dont une bande en lecture seule appelée bande d'*entrée* (et sur laquelle nous écrirons l'entrée x avant de lancer la machine), k bandes dites de *travail*, et une bande de *sortie* à écriture seule. Cette dernière ne sert à rien si l'on utilise la machine de Turing uniquement pour accepter ou pour rejeter une entrée x. En revanche, elle servira dans le cas de machines de Turing calculant des fonctions. Ce sera le cas notamment pour les fonctions de réduction entre problèmes, mais aussi pour la fonction dans la démonstration du théorème d'Immerman-Szelepcsényi que nous verrons plus loin. Une telle machine a trois alphabets : un alphabet pour écrire les mots fournis sur la bande d'entrée, un pour écrire les mots de la bande de sortie, et un alphabet pour les bandes de travail. Il n'y aucune raison que ces trois alphabets soient les mêmes ; mais ils sont tous finis.

Une telle machine est en *espace* $f(n)$ si et seulement la somme des tailles maximales des bandes de travail vaut au plus $f(n)$. Notons qu'on ne compte pas ni la taille de l'entrée ni la taille de la sortie ! C'est ce qui permettra de donner un sens, notamment, à la classe $\mathbf{NL} = \mathbf{NSPACE}(O(\log n))$, c'est-à-dire $\mathbf{NL} = \bigcup_{k\in\mathbb{N}} \mathbf{NSPACE}(k \log n)$.

Au passage, les pointilleux noteront que $\log n$ n'est pas défini si $n = 0$. Il se trouve qu'une définition équivalente de la complexité en espace $f(n)$ est de demander que l'espace utilisé soit majoré par $f(n)$ *pour n assez grand*, autrement dit pour tout $n \geq n_0$, où n_0 est une constante quelconque. Dans un sens, c'est évident : si on sait calculer en espace majoré par $f(n)$ pour tout n, alors prendre $n_0 = 0$. Réciproquement, si l'on sait calculer en espace majoré pour $f(n)$ mais uniquement pour tout $n \geq n_0$, disons à l'aide d'une machine $\mathcal{M}$, alors on peut construire une autre machine de Turing qui calcule comme $\mathcal{M}$ si $n \geq n_0$, et sinon retourne directement le résultat à partir d'une (immense) table retournant toutes les réponses pour toutes les entrées de taille inférieure à n_0 — il n'y a qu'un nombre fini, constant, de telles entrées ! On peut décider si l'entrée est de taille $n < n_0$ en n'utilisant que le contrôle fini de la machine, donc sans utiliser d'espace supplémentaire.

On utilisera souvent le théorème, dit de *speedup en espace* :

Théorème 2.2 *Pour toute fonction f :*
— $\mathbf{SPACE}(O(f(n))) = \mathbf{SPACE}(f(n))$ *;*
— $\mathbf{NSPACE}(O(f(n))) = \mathbf{NSPACE}(f(n))$.
Plus généralement, si $\mathcal{M}$ est une machine de Turing déterministe (resp., non déterministe) qui décide L (resp., qui calcule une fonction) en espace $O(f(n))$, alors il en existe une autre qui décide L (resp., qui calcule la même fonction) en espace $f(n)$.

Démonstration. Il suffit de démontrer que pour tout entier $\alpha > 1$, toute machine $\mathcal{M}$ qui accepte un langage L (resp., qui calcule la fonction donnée), et qui termine en espace $\alpha f(n)$, peut être transformée en une machine $\mathcal{M}'$ qui accepte L, mais termine en espace $f(n)$. (La ruse est que la notion d'espace ne compte que le nombre de lettres, pas la "taille" des lettres elle-même.) Soit k le nombre de bandes de travail de $\mathcal{M}$. On va supposer que l'alphabet Σ des bandes de travail de $\mathcal{M}$ contient un blanc $\sqcup$. $\mathcal{M}'$ fonctionne comme $\mathcal{M}$, sauf que son alphabet est formé de mots de longueur α sur Σ. (On ne peut représenter ainsi que des mots de $\mathcal{M}$ de longueur multiple de α ainsi, mais on peut toujours supposer que l'on rajoute suffisamment de blancs $\sqcup$ à la fin des bandes de $\mathcal{M}$ pour que ceci soit le cas.) Les états de contrôle de $\mathcal{M}'$ sont les uplets de la forme $(q, i_1, \ldots, i_k)$, où q est un état de contrôle de $\mathcal{M}$, et $i_1, \ldots, i_k \in \{0, 1, \ldots, \alpha - 1\}$. L'idée est que, si $\mathcal{M}'$ est dans un tel état, et que les lettres (dans Σ^α) lues sur chaque bande de travail sont w_1, …, w_k, ceci représente une configuration de $\mathcal{M}$ dans l'état q, où les lettres lues sont $w_1[i_1]$, …, $w_k[i_k]$ — en notant $w[i]$ la lettre numéro i du mot w, en commençant à 0. Les détails du codage, pénibles à souhait, sont omis. La démonstration est identique dans le cas des machines non déterministes. □

2.1 La classe NL, et le problème d'accessibilité dans les graphes orientés

NL est une classe intéressante car :

Lemme 2.3 (REACH) *Le problème REACH suivant, dit d'accessibilité dans les graphes orientés, est dans* **NL**.

ENTRÉE : un graphe orienté fini $G = (V, E)$, deux sommets $s, t \in V$;
QUESTION : existe-t-il un chemin de s à t dans G ?

Implicite ici est le fait que l'entrée est codée typiquement en binaire. Le format n'est pas très important, et l'on supposera que G est donné sous forme de listes d'adjacence ou d'une matrice d'adjacence. Ce n'est pas très important, car le passage d'un format à l'autre se fait en espace logarithmique... mais j'anticipe sur le notion de réduction logspace, que nous verrons plus loin. On supposera en revanche que les sommets de V sont numérotés de 0 à $N-1$. La description d'un sommet s'écrit donc sous forme d'un nombre à $O(\log N)$ bits. De plus, $\log N = O(\log n)$, où n est la taille de l'entrée.

Démonstration. Le principe de base est le suivant : poser $v := s$, tant que $v \neq t$ deviner un successeur v' de v, et poser $v := v'$; finalement accepter. Cette machine accepte si et seulement s'il existe un chemin de s à t dans G (et boucle sinon). Elle utilise un espace logarithmique : les bandes nécessaires pour stocker les représentations en binaire de v et v' sont de taille $O(\log n)$.

Mais cette machine ne termine pas toujours. Garder un historique de tous les sommets déjà visités n'est pas viable : ceci nécessiterait un espace bien au-delà du logarithmique. En revanche, on sait que s'il existe un chemin de s à t, alors il en existe un qui ne passe pas deux fois par le même sommet, donc passant par au plus N arcs (un "chemin court"). On écrit donc l'algorithme $REACHp(s, t, N)$ suivant :

$v := s$;
$k := 0$;
tant que $k \leq N$:
    si $v = t$ **alors stop** ; (* fin de la vérification. *)
    sinon deviner v' ;
        si v' est un successeur de v
          alors $v := v'$; $k := k + 1$;
        sinon rejeter ;
rejeter ;

On exécute ensuite $REACHp(s, t, N)$, et on accepte s'il termine normalement ("**alors stop**"), c'est-à-dire sans rejeter : si t est accessible depuis s, il y a un chemin court de s à t, qu'il suffit de suivre en devinant les bons v' les uns après les autres pour forcer cet algorithme à accepter. Sinon, quelles que soient les façons successives de choisir v', on sortira de la boucle lorsque $k > N$, et on rejettera. Ceci utilise un espace $3 \log n$ (3 bandes logarithmiques, pour stocker v, v', et k). □

Un problème de nature formelle est de savoir définir précisément quel est le *langage* L décrit par cette formulation d'un problème par "ENTRÉE" et "QUESTION". Précisément, on écrira souvent des langages sous la forme :
ENTRÉE : x, tel que $P(x)$ est vrai.
QUESTION : est-ce que $Q(x)$ est vrai ?
où P et Q sont deux prédicats. Ceci dénote de façon ambiguë le langage $\{x \mid P(x) \text{ et } Q(x)\}$

(une interprétation stricte : si x n'est pas la donnée d'un graphe, et de deux sommets s et t de ce graphe pour le problème REACH, alors x ne peut pas être dans le langage REACH... ce n'est même pas une donnée correctement formattée !), ou bien le langage $\{x \mid \text{si } P(x) \text{ alors } Q(x)\}$ (une interprétation laxiste : si x n'est pas correctement formattée, on accepte), ou bien encore d'autres langages L ayant seulement la propriété que pour tout x tel que $P(x)$ soit vrai, $x \in L$ si et seulement si $Q(x)$ est vrai.

On dira que de tels problèmes sont *bien formés* si $P(x)$ est un prédicat que l'on peut tester en espace logarithmique. Ce sera toujours le cas dans la suite (sauf lorsque nous traiterons du problème MAX3SAT (ϵ), bien plus tard). Ceci permettra de passer d'une interprétation à l'autre du problème en tant que langage sans rien changer à la complexité du langage... et donc d'ignorer le problème en toute sécurité.

▷ **Exercice 2.1**

Soit G un graphe orienté (V, E), avec $V = \{0, 1, \ldots, N-1\}$. Les sommets sont donc juste des nombres, que nous supposerons écrits en binaire. On peut représenter G par sa *matrice d'adjacence* M_G, qui est une matrice $N \times N$, dont l'entrée m_{ij} vaut le bit 1 si $(i, j) \in E$, 0 sinon. Plus précisément, G est décrit par le mot formé de N écrit en binaire, suivi d'un caractère spécial #, et de la liste des bits m_{00}, m_{01}, ..., $m_{O(N-1)}$, m_{10}, m_{11}, ..., $m_{1(N-1)}$, ..., $m_{(N-1)0}$, $m_{(N-1)1}$, ..., $m_{(N-1)(N-1)}$.

On peut aussi représenter G par *listes d'adjacence*. La liste d'adjacence de i est l'ensemble des j tels que $(i, j) \in E$, triée par ordre croissant, disons $j_1, \ldots, j_m$. Nous la représenterons sous la forme d'un mot $j_1; j_2; \ldots; j_m;$, où les j_i sont écrits en binaire, sur l'alphabet $\{0, 1\}$, et ";" est un troisième caractère. On représente alors G sous la forme $L_0 \bullet L_1 \bullet \ldots L_{N-1} \bullet$, où chaque L_i est la liste d'adjacence du sommet i, et $\bullet$ est un quatrième caractère.

Décrire une machine de Turing en espace logarithmique prenant en entrée un graphe G représenté sous forme de listes d'adjacence, et retournant la matrice d'adjacence de G.

▷ **Exercice 2.2**

Montrer que, réciproquement, on peut aussi fabriquer une machine de Turing en espace logarithmique qui prend en entrée une matrice d'adjacence pour un graphe G, et calcule sa représentation sous forme de listes d'adjacence.

Malgré la stupidité de l'algorithme $REACHp$, il mène à un algorithme qui est trivialement en temps polynomial. En effet, considérons le *graphe de configurations de taille* $k \log n$ d'une machine de Turing (même non déterministe) $\mathcal{M}$ en espace $k \log n$. C'est le graphe de toutes les configurations de $\mathcal{M}$, restreint à celles de taille au plus $k \log n$. Notons que l'on peut produire explicitement ce graphe en temps polynomial en n. Si a est le nombre de lettre de l'alphabet des bandes, le nombre de configurations distinctes de $\mathcal{M}$ est de l'ordre de $a^{k \log n} = n^{k \log a}$. Listons toutes ces configurations, explicitement. Pour chaque paire de configurations C et C', on produit un arc de C à C' si C' peut effectivement être obtenue à partir de C en un coup : ceci demande à consulter la table de transitions de $\mathcal{M}$ (en temps constant). Au total, on fabrique le graphe de configurations de $\mathcal{M}$ en temps $n^{2k \log a}$ multiplié par quelque chose de l'ordre de $O(\log n)$ (pour lire les configurations). Ceci est polynomial en n.

Fabriquons le graphe $\mathcal{G}(\mathcal{M}; k \log n)$ obtenu en rajoutant deux sommets frais OUI et NON au graphe de configurations, et en ajoutant une arête de C vers OUI (resp., NON) si C est une configuration acceptante (i.e., dont l'état interne est l'état d'acceptation ; resp., une

configuration rejetante). $\mathcal{G}(\mathcal{M}; k \log n)$ peut de nouveau être produit en temps polynomial en n.

Maintenant, on peut décider si $\mathcal{M}$ accepte x en se demandant si, dans le graphe $\mathcal{G}(\mathcal{M}; k \log n)$ (avec k le nombre de bandes de $\mathcal{M}$ et n la taille de x), il y a un chemin de la configuration $C_0(x)$ (la configuration initiale où x est écrit sur la bande d'entrée, toutes les autres bandes sont vides, l'état est l'état initial et toutes les têtes sont au début de leurs bandes respectives) vers le sommet OUI.

On peut ensuite décider en temps polynomial ce problème d'accessibilité, par un algorithme de *marquage*, bien plus classique que celui du lemme 2.3 : pour chaque sommet v du graphe, poser acc_v à faux ; initialiser une pile π de sommets à traiter à $[s]$; tant que π est non vide, dépiler un sommet v, et si acc_v est faux, mettre acc_v à vrai et empiler tous les sucesseurs de v sur π. Une fois π vide, on accepte si acc_t est vrai, on rejette sinon. Cet algorithme est en temps polynomial, grosso modo, car il ne progresse qu'en mettant un nouveau booléen acc_v à vrai, et il n'y a qu'un nombre polynomial de sommets v pour lesquels ceci se produit.

Résumons nous : pour résoudre un problème de **NL** (en espace $k \log n$ sur une machine $\mathcal{M}$) sur une entrée x de taille n, on peut construire un graphe $\mathcal{G}(\mathcal{M}; k \log n)$ en temps polynomial en n. Ce graphe est de taille polynomiale en n (nécessairement), et l'on peut tester si $\mathcal{M}$ accepte x en effectuant un test d'accessibilité, en temps polynomial de nouveau. Donc :

Lemme 2.4 $\mathbf{NL} \subseteq \mathbf{P}$.

Au passage, la technique consistant à transformer une machine en espace borné en son graphe de configurations nous sera utile plusieurs fois dans la suite, et permet de se ramener à un problème d'accessibilité. C'est pourquoi REACH, loin d'être un simple exemple, est en fait un modèle de calcul général. Nous allons le revoir au théorème de Savitch 2.9.

On a aussi trivialement :

Lemme 2.5 $\mathbf{L} \subseteq \mathbf{NL}$. *Plus généralement,* $\mathbf{SPACE}(f(n)) \subseteq \mathbf{NSPACE}(f(n))$.

2.2 Le théorème de Savitch

On peut penser que, de même que les problèmes de **NP** peuvent être jusqu'à exponentiellement plus complexes que ceux de **P**, les problèmes de **NL** devraient être aussi beaucoup plus complexes que ceux de **L**. Ce n'est pas le cas. On commence par observer qu'il existe un algorithme *déterministe* utilisant peu d'espace pour REACH. Ceci est dû à Savitch.

Proposition 2.6 $REACH \in \mathbf{SPACE}(\log^2 n)$.

Démonstration. S'il existe un chemin de s à t, alors il en existe un de longueur au plus N, comme plus haut. Soit p le nombre de bits nécessaires pour écrire N. On a $p = O(\log N) = O(\log n)$, et $N \leq 2^p$. Nous codons p en unaire, c'est-à-dire sous forme de la suite de p blancs. Le principe est d'écrire une fonction récursive $reach(v, v', q)$ décidant s'il existe ou non un chemin de v à v' de longueur au plus 2^q. Il suffira ensuite d'appeler $reach(s, t, p)$. La définition

de *reach* est simple : $reach(v, v', q)$ vaut vrai si $v = v'$ ou s'il y a un arc de v à v' ; sinon elle vaut faux si $q = 0$; sinon, elle vaut vrai si et seulement s'il existe un sommet v'' tel que $reach(v, v'', q-1)$ et $reach(v'', v', q-1)$ soient tous les deux vrais. Le quantificateur "il existe" dans la phrase précédente est réalisé sous forme d'une boucle qui parcourt tous les sommets possibles v'' (un compteur de taille $O(\log n)$). La récursion est implémentée à l'aide d'une pile : on empile les valeurs de v, v', q (en unaire), une adresse de retour (il n'y a qu'un nombre constant de possibilités ; cette adresse se code donc en taille constante), et une valeur pour le compteur v''. La profondeur maximale de la pile est $p = O(\log n)$, et chaque entrée est de taille $O(\log n)$ (la constante multiplicative du O ne dépendant pas du niveau de récursion). Ceci mène à une utilisation en espace de $O(\log^2 n)$. On se débarrasse ensuite du O à l'aide du théorème de speedup en espace. □

La notion de classe $\mathbf{SPACE}(f(n))$ ou $\mathbf{NSPACE}(f(n))$ ne sera intéressante que lorsque f est une fonction "raisonnable". On définit, comme pour les classes en temps :

Définition 2.7 (Propre) *Une fonction $f : \mathbb{N} \to \mathbb{N}$ est* propre *si et seulement si f est croissante, et il existe une machine de Turing déterministe $\mathcal{M}_f$ (à un nombre fixé de bandes) telle que, sur toute entrée x de taille n, $\mathcal{M}_f$ termine en écrivant exactement $f(n)$ blancs sur sa bande de sortie, en utilisant un temps $O(n + f(n))$ et un espace $O(f(n))$.*

Nous rappelons ou admettrons le résultat suivant :

Lemme 2.8 *La fonction $\log_2$ (logarithme de base 2) est propre. Tout polynôme à coefficients entiers positifs est une fonction propre. Si f est une fonction propre, alors $2^{f(n)}$ est une fonction propre de n. Si $f(n) \geq n$ pour tout $n \in \mathbb{N}$, et f et g sont propres, alors $f \circ g$ est propre.*

Le cas de $\log_2$ et de l'exponentielle demande essentiellement à savoir écrire un compteur en binaire sur une bande d'une machine de Turing. L'idée intuitive est qu'une fonction propre est une fonction que l'on peut calculer en utilisant le minimum possible de ressources en temps et en espace.

Ce que l'on entend par $\log_2$ est, si l'on souhaite être précis, la fonction qui à un entier n écrit en binaire retourne sa longueur, de nouveau écrite en binaire. Formellement, ce que l'on appelle $\log_2 n$ devrait donc être écrit $\lceil \log_2(n+1) \rceil$.

On a alors une forme de réciproque du lemme 2.5 :

Théorème 2.9 (Savitch) *Soit f une fonction propre avec $f(n) \geq \log_2 n$ pour tout $n \in \mathbb{N}$. Alors $\mathbf{NSPACE}(f(n)) \subseteq \mathbf{SPACE}(f^2(n))$.*

Démonstration. Soit $\mathcal{M}$ une machine non déterministe en espace $f(n)$. Comme f est propre, on peut réserver un certain nombre de bandes pour écrire $f(n)$ en unaire sur l'une d'entre elles. Posons p le résultat, et modifions légèrement l'algorithme de la proposition 2.6. Une configuration de $\mathcal{M}$ s'écrit (à un nombre de bits constant près) sous forme d'un mot sur p bits. Écrivons la fonction récursive $reach(v, v', q)$, où v et v' sont deux configurations de $\mathcal{M}$ et q un entier en unaire, tous de taille $O(f(n))$: $reach(v, v', q)$ vaut vrai si $v = v'$ ou si

l'on peut passer de la configuration v à la configuration v' de $\mathcal{M}$ en exactement une étape de calcul ; sinon elle vaut faux si $q = 0$; sinon, elle vaut vrai si et seulement s'il existe un sommet v'' tel que $reach(v, v'', q-1)$ et $reach(v'', v', q-1)$ soient tous les deux vrais. Tous les compteurs et la profondeur de pile sont de taille $O(p) = O(f(n))$, menant à un algorithme en espace $O(f^2(n))$. On enlève ensuite le grand O à l'aide du théorème de speedup en espace. □

Corollaire 2.10 **PSPACE** = **NPSPACE**.

Démonstration. **PSPACE** $\subseteq$ **NPSPACE** est évident. Réciproquement, un langage de **NPSPACE** est décidable en espace $p(n)$ sur une machine non déterministe, où p est un polynôme, donc en espace déterministe $p^2(n)$. □

C'est pourquoi nous ne parlerons plus jamais de **NPSPACE** ! Ceci est à opposer au fait que l'on est relativement persuadé que **NP** $\neq$ **P** (mais ce sont des classes en temps). De plus, le théorème de Savitch ne nous permet pas de conclure que **NL** égale **L**. En fait, on suppose généralement que **NL** $\neq$ **L**.

▷ **Exercice 2.3**

Montrer que l'on définit la même classe **SPACE**$(f(n))$ si l'on ne demande pas à la machine $\mathcal{M}$ de terminer sur toutes les entrées. Plus précisément, soit **SPACE**$'(f(n))$ la classe des langages L tels qu'il existe une machine de Turing $\mathcal{M}$ en espace $f(n)$ (n taille de l'entrée x), et qui accepte si et seulement si $x \in L$. (Noter qu'on ne demande pas que $\mathcal{M}$ termine lorsque $x \notin L$). Montrer que, si f est propre, alors **SPACE**$'(f(n)) =$ **SPACE**$(f(n))$.

▷ **Exercice 2.4**

Posons **SPACE**$''(f(n))$ la classe des langages L tels qu'il existe une machine de Turing $\mathcal{M}$ qui accepte en espace $f(n)$ sur toute entrée x de taille n. (Mais $\mathcal{M}$ a le droit de prendre davantage d'espace, et même de ne pas terminer sur les autres entrées.) Montrer que, si f est propre, alors **SPACE**$''(f(n)) =$ **SPACE**$(f(n))$.

Ce n'est pas une bonne pratique de faire référence aux entrailles d'une démonstration, comme nous l'avons fait au théorème 2.9 en nous référant aux entrailles de la proposition 2.6 : une proposition devrait être utilisée pour son énoncé, non pour les détails de sa démonstration. Voici donc une deuxième démonstration, montrant que le théorème 2.9 est un corollaire de la proposition 2.6. Ceci aura aussi l'avantage de présenter une nouvelle technique.

Soit $\mathcal{M}$ une machine non déterministe en espace $f(n)$. Il suffit de décider si OUI est accessible depuis $C_0(x)$ dans le graphe $G = \mathcal{G}(\mathcal{M}; f(n))$ défini de façon évidente. On applique alors la proposition 2.6 sur le graphe G en entrée... sauf que ceci nécessite un espace exponentiel en $f(n)$ pour stocker G ! C'est vexant : la construction de G se fait en espace $2f(n)$, ce n'est que la bande de sortie qui est de taille exponentielle en $f(n)$; d'autre part, la question de l'accessibilité se décide en espace un logarithme carré de cette exponentielle, à un facteur multiplicatif près, donc en espace $f^2(n)$, et ce n'est que la bande d'entrée qui est trop grosse.

Le problème est, plus généralement, de composer deux fonctions bornées en espace, sans utiliser l'espace (non comptabilisé) de la bande intermédiaire, servant de sortie à l'une des

machines et d'entrée à l'autre. Plutôt que de stocker la bande intermédiaire de taille exponentielle, on va simuler la première machine, à la demande, pour construire chaque caractère demandé par la seconde — et on ne stockera jamais qu'un seul caractère à la fois de la bande intermédiaire. La construction est décrite plus précisément au lemme suivant. De même que l'on note $g(n) = O(\log n)$ s'il existe $\alpha > 0$ tel que $g(n) \leq \alpha \log n$ pour tout n assez grand, on notera $g(n) = \Omega(\log n)$ s'il existe $\alpha > 0$ tel que $g(n) \geq \alpha \log n$ pour tout n assez grand.

Lemme 2.11 *Soit $\mathcal{M}'$ une machine de Turing (resp., non déterministe) en espace $g(n)$ calculant une fonction G (resp., décidant de l'appartenance à un langage L), avec $g(n) = \Omega(\log n)$, et F une fonction calculable en espace $h(n)$, cette fonction étant propre. Soit a le nombre de lettres de l'alphabet de bandes utilisé par la machine calculant F. On peut fabriquer une machine de Turing $\mathcal{M}''$ (resp., non déterministe) en espace $g(a^{h(n)})$ et pas plus, qui calcule $G \circ F$ (resp., décidant sur l'entrée x si $F(x) \in L$).*

Notons que la bande intermédiaire est de taille allant jusqu'à $a^{h(n)}$, puisque la machine calculant F, qui n'utilise qu'un espace $h(n)$, ne peut effectuer que $a^{h(n)}$ étapes de calcul sans boucler. Le résultat paraît donc intuitif. Il est un peu plus surprenant si l'on considère qu'il établit aussi que, si $g(n) = \log n$, alors $\mathcal{M}''$ calcule en espace $O(h(n))$, ce qui est beaucoup plus petit que la taille $a^{h(n)}$ de la bande intermédiaire.

Démonstration. On modifie $\mathcal{M}'$ comme suit. D'abord, on lui supprime sa bande d'entrée (oui ! Nous la considérerons maintenant comme une "bande virtuelle" de taille $a^{h(n)}$.) Ensuite, on lui ajoute une bande de travail de taille $h(n)$. Appelons cette bande le pointeur de lecture : nous nous en servirons pour écrire, sous forme d'un nombre de base a, le numéro de la case que nous souhaitons lire sur la bande virtuelle.

Chaque fois que $\mathcal{M}'$ souhaite lire la case de la bande virtuelle (l'entrée de $\mathcal{M}'$) sous la tête, on va simuler tout le calcul de $F(x)$, mais en jetant tous les caractères en sortie sauf à la position i formant le contenu du pointeur de lecture. Autrement dit, on a une bande d'entrée contenant x, toutes les bandes de travail de la machine calculant $F(x)$ en espace $h(n)\ldots$ mais on supprime la bande de sortie de cette machine, et on ne conserve à la place qu'une nouvelle bande de travail de taille $h(n)$ (le "pointeur d'écriture"). La machine calculant $F(x)$ est modifiée de sorte qu'au lieu d'écrire un caractère, disons c, à la position courante, elle compare le contenu j du pointeur d'écriture avec le contenu i du pointeur de lecture ; si $i = j$, alors on écrit c dans un endroit déterminé (ceci prenant un espace constant, on peut le stocker dans le contrôle de la machine), sinon, on le jette. (Ceci ne modifie par le comportement de cette machine, puisque sa bande de sortie est en écriture seule : son calcul est donc indépendant de ce qu'on a bien pu y écrire dans le passé.) Une fois tout le calcul de $F(x)$ simulé (et on n'en garde que le caractère numéro i, appelons-le c), $\mathcal{M}'$ reprend le calcul comme s'il avait lu c sur la bande virtuelle. Les machines simulant $F(x)$ et $\mathcal{M}'$ doivent aussi incrémenter et décrémenter leurs pointeurs de lecture et d'écriture, ce qui se fait sans avoir besoin d'espace supplémentaire. Au total, on utilise un espace $h(n)$ (où n est la taille de x) pour simuler le calcul de $F(x)$, plus deux compteurs de taille $h(n)$ chacun, plus un espace $g(n')$ pour $\mathcal{M}'$, où n' est la taille de la bande virtuelle, soit $n' = a^{h(n)}$. Le total fait $3h(n) + g(a^{h(n)})$. Or $h(n) = \log_a(a^{h(n)})$, et comme $g(n) = \Omega(\log n)$, il existe $\alpha > 0$

tel que $\log_a(n') \leq \alpha g(n)$ pour tout n assez grand, donc $h(n) \leq \alpha g(a^{h(n)})$. Le total est donc un $O(g(a^{h(n)}))$. On se débarrasse du O par le théorème de speedup en espace. □

On peut donc terminer la deuxième démonstration du théorème de Savitch, en utilisant le lemme 2.11 : en partant d'une machine $\mathcal{M}$ non déterministe en espace $f(n)$, on calcule son graphe $G = \mathcal{G}(\mathcal{M}; f(n))$. $F(x)$ est la fonction qui prend x en entrée, et retourne G, la configuration initiale $s = C_0(x)$, et la configuration à atteindre $t =$ OUI. La machine $\mathcal{M}'$ résoud REACH en espace $O(\log^2 n)$, comme assuré par la proposition 2.6. Le lemme 2.11 nous garantit qu'on peut alors calculer la composée de ces deux calculs en espace $O(\log^2(a^{f(n)})) = O(f^2(n))$, donc en espace $f^2(n)$ en utilisant le théorème de speedup en espace.

2.3 Réductibilité logspace, NL-complétude

Pour des classes de complexité aussi petites que **P** ou **NL**, la notion de réductibilité en temps polynomial n'a aucun sens. Nous utiliserons donc :

Définition 2.12 (Réductibilité logspace) *Un langage L est* logspace-réductible *à un langage L' si et seulement s'il existe une fonction $F(x)$ calculable en espace logarithmique en la taille n de x telle que pour tout x, $x \in L$ si et seulement si $F(x) \in L'$. On note alors $L \preceq_{\mathbf{L}} L'$.*

La relation $\preceq_{\mathbf{L}}$ est réflexive. Elle est transitive par le lemme 2.11. Notons aussi que tout langage logspace-réductible à un langage de **L**, resp. de **NL**, est encore dans cette classe. C'est de nouveau une conséquence du lemme 2.11. Tout langage logspace-réductible à un langage de **P**, **NP** ou **PSPACE** par exemple est encore dans cette classe : c'est une conséquence du fait que toute fonction $F(x)$ calculable en espace logarithmique est calculable en temps polynomial. La raison est bien plus simple qu'au lemme 2.4 : une machine déterministe en espace $k \log n$ ne passe jamais deux fois par la même configuration (sinon elle ne terminerait pas), et tourne donc en temps au plus $a^{k \log n} = n^{k \log a}$.

Proposition 2.13 *REACH est* **NL***-complet pour les réductions logspace.*

Démonstration. REACH est dans **NL** par le lemme 2.3. Réciproquement, si L est un langage de **NL**, soit $\mathcal{M}$ une machine non déterministe en espace $k \log n$ qui décide L. La réduction $F(x)$ s'effectue comme suit. D'abord, on calcule le graphe $G = \mathcal{G}(\mathcal{M}; k \log n)$, où n est la taille de x. Ceci n'utilise que deux bandes de taille $O(\log n)$ pour stocker deux configurations (la taille du graphe, qui est mis sur la bande de sortie, n'est pas comptée). On calcule $s = C_0(x)$, en espace constant, puis $t =$ OUI en espace constant. Il est clair que $F(x) \in$ REACH si et seulement si $\mathcal{M}$ accepte x, et de plus $F(x)$ est en espace logarithmique. □

Ceci est à mettre en parallèle du théorème classique suivant, que vous avez peut-être vu pour les réductions en temps polynomial.

Théorème 2.14 (Cook) *Le problème SAT suivant est* **NP***-complet pour les réductions logspace :*

ENTRÉE : un ensemble fini S de clauses propositionnelles.
QUESTION : S est-il satisfiable, autrement dit existe-t-il une valuation ρ, associant chaque variable de S à une valeur de vérité, qui rende toutes les clauses de S vraies ?

Démonstration. (Esquisse.) SAT est dans **NP** : il suffit de deviner ρ et de vérifier que ρ satisfait toutes les clauses de S en temps polynomial. Réciproquement, soit $\mathcal{M}$ une machine décidant un langage L de **NP**. Sur l'entrée x de taille n, $\mathcal{M}$ termine en temps $p(n)$, où p est un polynôme. Quitte à supposer que $\mathcal{M}$, en atteignant un état final, ne s'arrête pas mais boucle sans changer de configuration, on peut supposer que $\mathcal{M}$ atteint un état final en exactement $p(n)$ étapes. Visualisons un tableau de $p(n)$ lignes, chaque ligne étant une configuration possible de $\mathcal{M}$. Chaque ligne est représentée en binaire. On crée une variable x_{ij} pour chaque ligne i et chaque position j dans la ligne : i varie de 0 à $p(n)$, et comme l'on peut supposer que la machine n'utilise qu'au plus $p(n)$ cases, j varie de 0 à $q(n) = O(p(n))$. Pour chaque position (i, j) possible, on écrit les contraintes qui relient x_{ij} aux valeurs voisines (sur la ligne du dessus) représentant une étape d'exécution, de l'étape $i-1$ à l'étape i de $\mathcal{M}$; on écrit les contraintes portant sur la ligne 0 (le fait que la configuration d'entrée est $C_0(x)$) ; et sur la ligne $p(n)$ (le fait que la dernière configuration soit acceptante). Ceci fournit un ensemble de clauses de taille polynomiale, qui est satisfiable si et seulement si $\mathcal{M}$ accepte x. Mieux : on n'a besoin de maintenir que des compteurs pour i, j, $i-1$ et on itère sur un nombre constant de valeurs de colonne autour de j. Ceci est faisable en espace logarithmique. □

▷ **Exercice 2.5**
Soit 3-SAT la restriction de SAT aux clauses formées d'au plus trois littéraux (des 3-*clauses*). Autrement dit, l'entrée est un ensemble fini S de 3-clauses, et l'on se demande si S est satisfiable. Montrer que 3-SAT est **NP**-complet (pour les réductions logspace).

▷ **Exercice 2.6**
Soit 2-SAT la restriction de SAT aux clauses formées d'au plus deux littéraux (des 2-*clauses*). Autrement dit, l'entrée est un ensemble fini S de 2-clauses, et l'on se demande si S est satisfiable. On sait que 2-SAT se résoud en temps polynomial : l'algorithme d'Even, Itai, et Shamir, que vous devrez écrire, consiste à voir chaque clause $L \vee L'$ comme deux implications $\neg L \Rightarrow L'$ et $\neg L' \Rightarrow L$, et à fabriquer un graphe G où les implications sont les arcs. (Indication : lorsqu'aucun chemin ne passe à la fois par une variable et sa négation, on construit une valuation satisfaisant S comme suit. On énumère les variables : $x_1, \ldots, x_n$. On construit la valeur de vérité de x_i, ainsi qu'un graphe G_i, par récurrence sur i, $1 \leq i \leq n$, comme suit. D'abord G_0 est le graphe G. Pour tout i, $1 \leq i \leq n$, s'il existe un chemin de $\neg x_i$ vers x_i dans G_{i-1}, alors on met x_i à vrai et G_i est G_{i-1}. Sinon on met x_i à faux et G_i est G_{i-1} plus un arc de x_i vers $\neg x_i$. On montrera que dans aucun des G_i il n'y a de circuit passant à la fois par $+x$ et $-x$ pour aucun x.) En déduire que la négation de 2-SAT (le problème de l'insatisfiabilité d'un ensemble de 2-clauses) est **NL**-complet.

2.4 Le théorème d'Immerman-Szelepcsényi

Continuons notre étude de **NL**. On pense communément que **NP** n'est pas clos par complémentaire. Par le théorème de Cook, le fait que **NP** soit clos par complémentaire serait

équivalent au fait que SAT soit **coNP**-complet (**coNP** étant la classe des complémentaires de langages dans **NP**), ou bien que **NP** = **coNP**, ou encore que VAL soit **NP**-complet, où VAL est le problème :
ENTRÉE : une formule propositionnelle F.
QUESTION : F est-elle valide, autrement dit est-il vrai que toute valuation ρ des variables de F rende F vraie ?
(Voir la proposition 4.4 plus loin.)

Si **P** = **NP**, bien sûr **NP** = **coNP** = **P**, puisque **P** est clos par complémentaire. On ne sait pas si **NP** = **coNP** implique **P** = **NP**. En revanche, et ceci n'est pas trivial, **NL** = **coNL** : c'est le théorème d'Immerman-Szelepcsényi plus bas. Par la proposition 2.13, ceci revient à dire que la *non*-accessibilité dans les graphes orientés se fait en espace logarithmique non déterministe.

On se heurte immédiatement à une difficulté : on a le droit de deviner des données (de taille logarithmique), c'est-à-dire de prouver l'existence de données menant à acceptation. Mais la non-accessibilité revient à demander que tout chemin partant de s a la propriété qu'il ne passe pas par t. En clair, ce théorème revient à coder un quantificateur universel par un quantificateur existentiel.

Pour résoudre ce dilemme, on va résoudre un problème voisin, celui de *compter* le nombre de sommets accessibles depuis un sommet s. On va le faire avec des programmes utilisant de plus en plus le non-déterminisme. Mais en premier, nous devons préciser ce que veut dire qu'une machine non-déterministe calcule un résultat. A priori, ceci peut dépendre des divers choix effectués. Nous l'exclurons donc d'emblée :

Définition 2.15 *On dit qu'une machine non-déterministe $\mathcal{M}$ calcule une fonction f si et seulement si :*

1. *Sur toute entrée x, il existe au moins une exécution de $\mathcal{M}$ qui accepte ;*
2. *À x fixé écrit sur la bande d'entrée, toute exécution de $\mathcal{M}$ qui accepte termine avec la valeur $f(x)$ écrite sur la bande de sortie.*

En particulier, toutes les exécutions qui acceptent écrivent la même valeur, $f(x)$, sur la bande de sortie. On demande qu'au moins une exécution accepte, sinon le résultat calculé par $\mathcal{M}$ serait indéfini.

Pour compter le nombre de sommets accessibles depuis s, on va compter le nombre de sommets accessibles depuis s en 0 étape (la réponse est 1), puis en 1 étape, puis en 2, ..., puis en $N-1$ étapes, où N est le nombre total de sommets. Nous commençons par un programme trop gourmand en mémoire, que nous corrigerons ensuite. Notons $y \to z$ en abrégé pour dire qu'il existe un arc de y à z dans le graphe $G = (V, E)$.

```
1 J := [s] ;
2 pour i = 1 à N − 1 :
3       (* Ici, J sera toujours la liste, sans duplicata,
4          des sommets accessibles depuis s en i − 1 étapes au plus. *)
5       K := [] ; (* On va mettre dans K ceux qui sont accessibles
```

```
6                   en i étapes au plus. *)
7       pour chaque sommet z de G :
8             (* z est accessible en ≤ i étapes ssi
9                il existe y ∈ J tel que y = z ou y → z. *)
10            found := faux ;
11            pour chaque sommet y de G :
12                  si y ∈ J et (y = z ou y → z)
13                     alors found := vrai ;
14            si found alors K := z :: K ; (* ajouter z à K *)
15      J := K ;
16retourner |J| ;
```

Ce qui ne va pas ici, c'est que J et K sont beaucoup trop gros, de taille polynomiale plutôt que logarithmique. Au lieu de les conserver, on va conserver uniquement leurs nombres d'éléments j, resp. k. Par inspection du code ci-dessus, le seul endroit où l'on aurait besoin de J ou K plutôt que j ou k, c'est à la ligne 12, où l'on teste si $y \in J$. (On peut remplacer sans problème $J := [s]$ à la ligne 1 par $j := 1$, $K := []$ à la ligne 5 par $k := 0$, et $K := z :: K$ par $k := k + 1$ à la ligne 14.)

Soyons audacieux, et *devinons* le booléen $y \in J$. (Nous n'avions pas encore utilisé les possibilités offertes par le non-déterminisme jusqu'ici !) Ceci pose évidemment un problème, puisque rien ne nous garantit que le booléen deviné est bien la valeur du test $y \in J$. Si on a deviné **vrai**, c'est facile : on peut immédiatement vérifier que y est accessible depuis s en au plus $i - 1$ étapes, par l'algorithme $REACHp(s, y, i - 1)$ du lemme 2.3 (qui est lui-même non déterministe). Si l'on a deviné **faux** en revanche, on ne peut pas vérifier, du moins tout de suite, que y n'est pas accessible en $i - 1$ étapes. Mais, grâce à la vérification faite par $REACHp(s, y, i - 1)$ au cas où on aurait deviné **vrai** pour $y \in J$ à la ligne 12bis, le nombre q de y pour lesquels on aura deviné **vrai** vaudra au plus le nombre j de sommets accessibles en au plus $i - 1$ étapes, si jamais on atteint la sortie de la boucle en ligne 13bis. De plus, si jamais q est strictement inférieur à j, c'est que l'on aura deviné **faux** au moins une fois de trop à la ligne 12. C'est ainsi que l'on détecte a posteriori que tous les booléens $y \in J$ devinés **faux** correspondent effectivement à des sommets y inaccessibles en $i - 1$ étapes ou moins. On aboutit ainsi au programme :

```
1     j := 1 ;
2     pour i = 1 à N − 1 :
3           (* Ici, j sera toujours le nombre
4              de sommets accessibles depuis s en i − 1 étapes au plus. *)
5           k := 0 ; (* On va mettre dans k le nombre de ceux qui sont
6                       accessibles en i étapes au plus. *)
7           pour chaque sommet z de G :
8                 (* z est accessible en ≤ i étapes ssi
9                    il existe y ∈ J tel que y = z ou y → z. *)
10                found := faux ;
10bis             q := 0
```

```
11              pour chaque sommet y de G :
11bis               deviner un booléen b ; (* b plutôt que "y ∈ J". *)
12                  si b
12bis                 alors REACHp(s, y, i − 1) ; (* rejette si b vrai mais y ∉ J. *)
12ter                       q := q + 1 ; (* incrémente le nombre de sommets qu'on pense dans J. *)
12quater              si y = z ou y → z
13                     alors found := vrai ;
13bis           si q < j alors rejeter ; (* un b faux a été mal deviné. *)
14              si found alors k := k + 1 ; (* ajouter z à K *)
15          j := k ;
16      retourner j ;
```

Ce programme utilise un espace logarithmique. On a en effet besoin de compteurs i, j, k, q valant au plus $\log N = O(\log n)$, plus les compteurs v, v' et t de $REACHp$. (On supposera que $REACHp$ est *inliné*, c'est-à-dire que l'appel à $REACHp$ est remplacé par son code, l'instruction **stop** permettant désormais de poursuivre l'exécution à la ligne 12ter.) Ceci représente un espace $7 \log n$.

Ceci ne résout pas tout à fait notre problème, mais nous n'en sommes plus loin :

Théorème 2.16 (Immerman-Szelepcsényi) *REACH est dans* **coNL**.

Démonstration. Partant du graphe G, des sommets s et t, produisons le graphe G' obtenu en ajoutant tous les arcs de t vers v, pour tout sommet v. Ceci se fait en espace logarithmique, ce qui est particulièrement clair si G est représenté sous forme de matrice d'adjacence : ceci revient à remplir de uns toute la ligne t. Si t est accessible depuis s dans G, alors tout sommet sera accessible depuis s dans G', en particulier le nombre de sommets accessibles depuis s dans G' vaudra N, le nombre de sommets dans G (ou G').

Si t n'est pas accessible depuis s dans G, alors il ne sera pas non plus dans G' : sinon, choisissons un chemin le plus court de s vers t dans G' ; il contiendrait au moins un des arcs supplémentaires $t \to v$ de G', puisque t n'est pas accessible depuis s dans G ; donc il serait de la forme $s \to^* t \to v \to^* t$, mais le préfixe $s \to^* t$ serait un chemin encore plus court, contradiction. Donc si t n'est pas accessible depuis s dans G, alors le nombre de sommets accessibles depuis s dans G' sera strictement inférieur à N.

On décide donc de la non-accessibilité de t depuis s dans G en produisant le couple (G', N), où G' est comme ci-dessus et N est le nombre de sommets de G, en espace logarithmique, puis en vérifiant que le nombre de sommets accessibles depuis s dans G' est strictement inférieur à N. La composée des deux opérations se fait dans **NL**, par le lemme 2.11. □

Corollaire 2.17 **NL** = **coNL**.

Démonstration. Par la proposition 2.13, tout problème de **NL** se réduit en espace logarithmique à REACH, qui est dans **coNL** par le théorème 2.16. Comme **coNL** est stable par réductions logspace, en utilisant le lemme 2.11, c'est que **NL** ⊆ **coNL**. Mais alors **coNL** ⊆ **NL** (si $L \in$ **coNL**, le complémentaire de L est dans **NL**, donc dans **coNL**, donc $L \in$ **NL**), d'où l'égalité. □

▷ **Exercice 2.7**
En utilisant l'exercice 2.6, montrer que 2-SAT est **NL**-complet.

Corollaire 2.18 *Pour toute fonction propre f telle que $f(n) = \Omega(\log n)$,* $\mathbf{NSPACE}(f(n)) = \mathbf{coNSPACE}(f(n))$.

Démonstration. Si $\mathcal{M}$ décide de l'appartenance à un langage L en espace non déterministe $f(n)$, on peut décider si $x \notin L$ en construisant $G = \mathcal{G}(\mathcal{M}; f(n))$ en espace $O(f(n))$, et en testant que OUI est inaccessible depuis $C_0(x)$ (x de taille n) en espace logarithmique (par le théorème 2.16) en la taille $a^{O(f(n))}$ de G, c'est-à-dire en espace $O(f(n))$. On applique le lemme 2.11, puis le théorème de speedup en espace, pour obtenir une machine non déterministe en espace $f(n)$ décidant si $x \notin L$. Ceci montre que L est dans $\mathbf{coNSPACE}(f(n))$. Comme L est arbitraire, $\mathbf{NSPACE}(f(n)) \subseteq \mathbf{coNSPACE}(f(n))$. On en déduit l'inclusion inverse par passage aux langages complémentaires. □

Notons que ce corollaire n'est pas une conséquence du théorème de Savitch (théorème 2.9) : tout ce que l'on pourrait en conclure c'est que $\mathbf{NSPACE}(f(n))$ est inclus dans $\mathbf{SPACE}(f^2(n))$, donc dans $\mathbf{coNSPACE}(f^2(n))$, par $\mathbf{coNSPACE}(f(n))$. On déduit en revanche aisément, soit du théorème de Savitch, soit du théorème d'Immerman-Szelepcsényi, que $\mathbf{NPSPACE} = \mathbf{coNPSPACE}$ ($= \mathbf{PSPACE}$).

2.5 PSPACE et QBF

En parlant de **PSPACE**, nous remarquons tout de suite que **PSPACE** a des problèmes complets, comme **NP** et comme **NL**. Le prototype en est le problème QBF ("Quantified Boolean Formulae") suivant :
ENTRÉE : une formule booléenne quantifiée $F = Q_1 x_1 \cdot \ldots \cdot Q_n x_n \cdot S$, où S est un ensemble fini de clauses portant uniquement sur les variables $x_1, \ldots, x_n$, et où $Q_1, \ldots, Q_n \in \{\forall, \exists\}$.
QUESTION : F est-elle vraie ?

Avant de passer à QBF, on observe le fait suivant. **EXPTIME** est la classe des langages décidables en temps déterministe exponentiel, c'est-à-dire borné par l'exponentielle d'un polynôme en n.

Lemme 2.19 $\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE} \subseteq \mathbf{EXPTIME}$.

Démonstration. $\mathbf{P} \subseteq \mathbf{NP}$ est trivial. Toute machine de Turing (même non déterministe) en temps polynomial ne peut utiliser qu'un espace polynomial, donc $\mathbf{NP} \subseteq \mathbf{NPSPACE} = \mathbf{PSPACE}$ (théorème de Savitch). Pour montrer $\mathbf{PSPACE} \subseteq \mathbf{EXPTIME}$, on peut décider un langage L de **PSPACE** en fabriquant le graphe $\mathcal{G}(\mathcal{M}; p(n))$ d'une machine $\mathcal{M}$ décidant L en espace $p(n)$ polynomial en n. Ceci prend un temps exponentiel en $p(n)$, puis l'on teste l'accessibilité par un algorithme de marquage déterministe. □

Le fait que QBF soit **PSPACE**-complet est dû à Stockmeyer et Meyer [2, théorème 4.3]. La question "F est-elle vraie ?" a un sens, car toutes les variables de S sont quantifiées.

Évaluer F se fait donc sans avoir besoin de fixer les valeurs des variables. On peut définir simplement la valeur d'une formule QBF comme suit : $[\![\forall x \cdot F]\!] = [\![F[x := \textbf{vrai}]]\!] \wedge [\![F[x := \textbf{faux}]]\!]$, $[\![\exists x \cdot F]\!] = [\![F[x := \textbf{vrai}]]\!] \vee [\![F[x := \textbf{faux}]]\!]$, $[\![S]\!]$ étant défini de la façon évidente pour les ensembles de clauses S sans aucune variable. Ce processus d'évaluation de F se fait cependant en temps exponentiel en le nombre n de variables. On peut voir d'autre part que QBF est **NP**-dur, car il contient le cas particulier où tous les quantificateurs Q_i valent $\exists$, qui n'est rien d'autre que SAT.

On peut décider QBF en un peu mieux que le temps exponentiel :

Lemme 2.20 *QBF est dans* **PSPACE**.

Démonstration. Il suffit d'implémenter la fonction d'évaluation ci-dessus, en un peu raffiné. Définissons la valeur d'une formule quantifiée F dans un environnement ρ, par récurrence sur F, où ρ donne une valeur booléenne à au moins toutes les variables libres de F, par :

$Eval(F, \rho) =$
case F **of**
  '$\forall x \cdot F'$' $\Rightarrow$
    si $Eval(F', \rho[x \mapsto \textbf{vrai}]) = \textbf{vrai}$
    alors si $Eval(F', \rho[x \mapsto \textbf{faux}]) = \textbf{vrai}$
      alors retourner vrai ;
    sinon retourner faux ;
    sinon retourner faux ;
  '$\exists x \cdot F'$' $\Rightarrow$
    si $Eval(F', \rho[x \mapsto \textbf{vrai}]) = \textbf{faux}$
    alors si $Eval(F', \rho[x \mapsto \textbf{faux}]) = \textbf{faux}$
      alors retourner faux ;
    sinon retourner vrai ;
    sinon retourner vrai ;
  S $\Rightarrow$
    pour chaque $C \in S$:
      $OK := \textbf{faux}$;
      pour chaque $L \in C$:
        case L **of**
        'x' $\Rightarrow$ **si** $\rho(x) = \textbf{vrai}$ **alors** $OK := \textbf{vrai}$;
        '$\neg x$' $\Rightarrow$ **si** $\rho(x) = \textbf{faux}$ **alors** $OK := \textbf{vrai}$;
      si $OK = \textbf{faux}$ **alors retourner faux** ;
    retourner vrai ;

(On note $\rho[x \mapsto b]$ l'environnement qui à x associe b, et à tout $y \neq x$ associe $\rho(y)$.) Ceci est un algorithme récursif que l'on peut coder à l'aide d'une pile des appels et en stockant les valeurs des variables et résultats locaux. La taille de ces derniers est logarithmique, sauf pour ρ qui est polynomial, et la profondeur de la pile d'appels est d'au plus le nombre de variables quantifiées, donc polynomial aussi. On accepte enfin si $Eval(F, []) = \textbf{vrai}$, et on rejette sinon. □

Théorème 2.21 (Stockmeyer-Meyer) *QBF est* **PSPACE***-complet pour les réductions logspace.*

Démonstration. Soit $\mathcal{M}$ une machine en espace polynomial décidant de l'appartenance $x \in L$ (x de taille n). On peut coder chaque configuration de $\mathcal{M}$ sur $p(n)$ bits, où $p(n)$ est un polynôme de n. Comme pour le théorème de Cook, on va réserver des vecteurs $\vec{x} = (x_i)_{i=1}^{p(n)}$ de $p(n)$ variables booléennes pour coder chaque configuration. Mais $\mathcal{M}$ peut passer par un nombre exponentiel en $p(n)$ de configurations. Il est donc exclu de les représenter toutes, comme dans le théorème de Cook. En revanche, on peut fabriquer une formule $F_k(\vec{x}, \vec{y})$ représentant une exécution d'au plus 2^k étapes de $\mathcal{M}$, par récurrence sur k, comme suit. On note qu'on peut écrire une formule $Succ(\vec{x}, \vec{y})$ qui exprime que $\vec{y}$ est la configuration suivante de celle de $\vec{x}$ dans $\mathcal{M}$. On peut aussi écrire une formule $\vec{x} = \vec{y}$: c'est la conjonction des $\neg x_i \vee y_i$ et des $\neg y_i \vee x_i$. Alors $F_0(\vec{x}, \vec{y}) = (\vec{x} = \vec{y} \vee Succ(\vec{x}, \vec{y}))$. Il est tentant de définir $F_{k+1}(\vec{x}, \vec{y})$ par $\exists \vec{z} \cdot F_k(\vec{x}, \vec{z}) \wedge F_k(\vec{z}, \vec{y})$, mais le fait que F_k soit utilisée deux fois va faire grossir F_{k+1} exponentiellement en k. On pose plutôt :

$$\begin{aligned} F_{k+1}(\vec{x}, \vec{y}) \quad = \quad & \exists \vec{z} \cdot \forall b \cdot \exists \vec{u}, \vec{v} \cdot F_k(\vec{u}, \vec{v}) \wedge \\ & \Big[b \Rightarrow (\vec{u} = \vec{x} \wedge \vec{v} = \vec{z}) \Big] \\ & \wedge \Big[\neg b \Rightarrow (\vec{u} = \vec{z} \wedge \vec{v} = \vec{y}) \Big] \end{aligned}$$

Une façon de vérifier que ceci est équivalent à la définition que nous avons donnée plus haut est de remplacer $\forall b \cdot G(b)$ (où $G(b)$ est la formule qui suit la quantification sur b) par $G(\mathbf{vrai}) \wedge G(\mathbf{faux})$, et de simplifier. $G(\mathbf{vrai})$ est équivalent à $\exists \vec{u}, \vec{v} \cdot F_k(\vec{u}, \vec{v}) \wedge \vec{u} = \vec{x} \wedge \vec{v} = \vec{z}$, donc à $F_k(\vec{x}, \vec{z})$, et l'on opère de même pour $G(\mathbf{faux})$.

Ceci résout notre problème de taille, mais F_k est loin d'être une formule QBF : elle n'est pas en forme prénexe (tous les quantificateurs ne sont pas devant), et même en ignorant les quantificateurs, nous n'obtiendrons pas une forme clause (il reste des $\vee$ au-dessus de $\wedge$).

Réglons le problème de la forme clausale d'abord. Il est facile, quoique laborieux, de voir que l'on peut écrire $Succ(\vec{x}, \vec{y})$ sous forme clausale. Mais $F_0(\vec{x}, \vec{y})$ introduit un $\vee$ par-dessus. Mettre $\vec{x} = \vec{y} \vee Succ(\vec{x}, \vec{y})$ en forme clausale est assez impénétrable. On va donc préférer l'utilisation de l'astuce suivante, même si elle n'est pas indispensable : $A \vee B$ est logiquement équivalent à $\exists b \cdot (b \Rightarrow A) \wedge (\neg b \Rightarrow B)$. D'autre part, si S est une formule en forme clausale, et $\pm b$ désigne soit b soit $\neg b$, alors $\pm b \Rightarrow S$ est équivalent à la forme clausale $\pm b[S]$, obtenue en rajoutant le littéral $\mp b$ (avec le signe opposé à $\pm b$) en tête de chaque clause de S. On remplace donc $F_0(\vec{x}, \vec{y})$ par $F'_0(\vec{x}, \vec{y}) = \exists b \cdot b[\vec{x} = \vec{y}] \wedge \neg b[Succ(\vec{x}, \vec{y})]$. Au quantificateur près, ceci est maintenant bien une forme clausale.

On définit ensuite :

$$\begin{aligned} F'_{k+1}(\vec{x}, \vec{y}) \quad = \quad & \exists \vec{z} \cdot \forall b \cdot \exists \vec{u}, \vec{v} \cdot F'_k(\vec{u}, \vec{v}) \wedge \\ & b[\vec{u} = \vec{x} \wedge \vec{v} = \vec{z}] \\ & \wedge \neg b[\vec{u} = \vec{z} \wedge \vec{v} = \vec{y}] \end{aligned}$$

Il est ensuite facile de voir que l'on peut mettre $F'_k(\vec{x},\vec{y})$ en forme prénexe, pour tout k, en utilisant uniquement les deux règles :

$$\begin{array}{rcl}(\exists t\cdot F)\wedge G & \rightarrow & \exists t\cdot(F\wedge G)\\ (\forall t\cdot F)\wedge G & \rightarrow & \forall t\cdot(F\wedge G)\end{array}$$

où t n'apparaît pas dans G. Cette dernière condition est essentielle, et malheureusement on ne peut pas imprimer $F_k(\vec{x},\vec{y})$ sur la bande de sortie, puis la mettre en forme prénexe... puisqu'on ne peut lire sur la bande de sortie.

Il va donc falloir fabriquer la formule en forme prénexe directement. Pour ceci, on va normaliser les noms de toutes les variables utilisées : la variable b de F'_k sera désormais $b\#k$, les variables z_i (de $\vec{z}$) de F'_k seront $z_i\#k$ (et le vecteur de ces variables sera noté $\vec{z}\#k$), etc. Nous les coderons, concrètement sous forme de mots `b#`k`#` (k écrit en binaire), ou `z#`i`#`k`#`. En revanche, nous ne réserverons pas de nom de variable pour $\vec{u}$ et $\vec{v}$ dans F'_{k+1}, et nous réutiliserons $\vec{x}\#k$ et $\vec{y}\#k$. On écrit donc, en ignorant les quantificateurs (qui seront tous écrits en tête de la formule finale) :

$$\begin{array}{rcl}F''_0 & = & b\#0[\vec{x}\#0=\vec{y}\#0]\wedge\neg b\#0[Succ(\vec{x}\#0,\vec{y}\#0)]\\ F''_{k+1} & = & F''_k\\ & & \wedge b\#(k+1)[\vec{x}\#k=\vec{x}\#(k+1)\wedge\vec{y}\#k=\vec{z}\#(k+1)]\\ & & \wedge\neg b\#(k+1)[\vec{x}\#k=\vec{z}\#(k+1)\wedge\vec{y}\#k=\vec{y}\#(k+1)]\end{array}$$

On note que pour tout k, F''_k est une formule en forme clausale, de taille polynomiale en k et $p(n)$. Il s'agit maintenant de produire F''_{k_0}, pour un k_0 tel que 2^{k_0} soit supérieur ou égal au temps maximal pris par $\mathcal{M}$, disons $a^{p(n)}$. Si a s'écrit sur k_1 bits ($a\leq 2^{k_1}$), il suffit de prendre $k_0=k_1p(n)$. On réserve un compteur k (de taille logarithmique en $k_1p(n)$, donc en n !), initialisé à k_0, et qui décroîtra jusqu'à 0.

On imprime d'abord tous les quantificateurs : $\exists\vec{x}\#k_0,\vec{y}\#k_0,\vec{z}\#k_0\cdot\forall b\#k_0\cdot\exists\vec{x}\#(k_0-1),\vec{y}\#(k_0-1),\vec{z}\#(k_0-1)\cdot\forall b\#(k_0-1)\cdot\ldots\exists\vec{x}\#k,\vec{y}\#k,\vec{z}\#k\cdot\forall b\#k\cdot\ldots\cdot\exists\vec{z}\#1\cdot\forall b\#1\cdot\exists\vec{x}\#0,\vec{y}\#0\cdot$. Notons que ceci nécessite un deuxième compteur auxiliaire, comptant de 0 à $p(n)$, pour énumérer les variables des vecteurs $\vec{x}$, $\vec{y}$, etc. Ce compteur est aussi de taille logarithmique. On réinitialise k à k_0 de nouveau, et l'on énumère les clauses formant F''_{k_0}, en décrémentant

k jusqu'à 0 :

$$\begin{array}{rl}
b\#k_0[\vec{x}\#(k_0-1)=\vec{v}\#k_0] & b\#k_0[\vec{y}\#(k_0-1)=\vec{z}\#k_0] \\
\neg b\#k_0[\vec{x}\#(k_0-1)=\vec{z}\#k_0] & \neg b\#k_0[\vec{y}\#(k_0-1)=\vec{y}\#k_0] \\
\multicolumn{2}{c}{\dots} \\
b\#k[\vec{x}\#(k-1)=\vec{v}\#k] & b\#k[\vec{y}\#(k-1)=\vec{z}\#k] \\
\neg b\#k[\vec{x}\#(k-1)=\vec{z}\#k] & \neg b\#k[\vec{y}\#(k-1)=\vec{y}\#k] \\
\multicolumn{2}{c}{\dots} \\
b\#1[\vec{x}\#0=\vec{v}\#1] & b\#1[\vec{y}\#0=\vec{z}\#1] \\
\neg b\#1[\vec{x}\#0=\vec{z}\#1] & \neg b\#1[\vec{y}\#0=\vec{y}\#1] \\
b\#0[\vec{x}\#0=\vec{y}\#0] & \neg b\#0[Succ(\vec{x}\#0,\vec{y}\#0)]
\end{array}$$

De nouveau, ceci nécessite un compteur auxiliaire pour énumérer les vecteurs de clauses des égalités de la forme $\pm b[\vec{u}=\vec{v}]$, qui prend un espace logarithmique, ainsi que dans les clauses de la forme $\neg b\#0[Succ(\vec{x}\#0,\vec{y}\#0)]$.

On n'a pas terminé d'écrire notre formule. On ajoute toutes les clauses décrivant la configuration initiale : à savoir, pour chaque bit x_i de l'entrée x ($0 \leq i \leq n-1$), on écrit la clause $\mathtt{x\#}i\mathtt{\#}k_0\mathtt{\#}$ si x_i est à 1, $\neg\mathtt{x\#}i\mathtt{\#}k_0\mathtt{\#}$ sinon, et on définit de même tous les autres $\pm\mathtt{x\#}i\mathtt{\#}k_0\mathtt{\#}$ de sorte à exprimer que tous les autres caractères de la bande d'entrée sont des blancs, à coder la position des têtes et l'état de contrôle initial de $\mathcal{M}$. On décrit de même la contrainte selon laquelle la configuration finale de $\mathcal{M}$ doit être acceptante à l'aide des variables $\mathtt{y\#}i\mathtt{\#}k_0\mathtt{\#}$, $0 \leq i \leq n-1$, en testant juste l'état de contrôle. (Que l'on supposera codé au début des configurations, pour simplifier.) Il est clair que la formule ainsi écrite est une formule QBF, qu'elle est vraie si et seulement si $\mathcal{M}$ accepte sur l'entrée x. Et nous n'avons utilisé qu'un espace $O(\log p(n)) = O(\log n)$. On conclut par le théorème de speedup en espace. □

On notera que la démonstration ci-dessus ne dépend pas du fait que $\mathcal{M}$ soit déterministe ou non déterministe. Ceci ne devrait pas vous surprendre, étant donné que **PSPACE** = **NPSPACE**.

3 Machines alternantes

Une machine de Turing *alternante* est une machine de Turing dont l'espace des états non finaux est partitionné en deux : les états *existentiels* et les états *universels*. Intuitivement, sur un état existentiel, la machine de Turing fonctionne comme une machine non déterministe : elle devine l'état suivant à exécuter, de sorte à (tenter d') atteindre un état acceptant. Le cas des états universels est dual : sur un tel état, la machine de Turing va vérifier que quel que soit le choix de l'état suivant à exécuter (parmi ceux décrits dans la table de transition), on peut atteindre un état acceptant.

Une autre façon de le dire est qu'une machine de Turing alternante est un jeu à deux joueurs, ∃ et ∀. (Parfois appelés ∃loïse et ∀bélard, ou Eve et Adam, ou joueur et opposant.) Le but du joueur ∃ est d'atteindre un état acceptant, celui de ∀ d'empêcher d'atteindre un

$$\frac{}{\vdash C : \mathrm{OK}}\ (q_C \text{ acceptant})$$

$$\frac{\vdash C' : \mathrm{OK}}{\vdash C : \mathrm{OK}}\ (q_C \text{ existentiel, } C \to_{\mathcal{M}} C')$$

$$\frac{\vdash C_1 : \mathrm{OK} \ldots \vdash C_m : \mathrm{OK}}{\vdash C : \mathrm{OK}}\ (q_C \text{ universel, } \{C_1, \ldots, C_m\} = \{C' \mid C \to_{\mathcal{M}} C'\})$$

FIGURE 1 – Configurations menant à l'acceptation

état acceptant. Lorsque c'est le tour de $\exists$, $\exists$ devine quel coup jouer, dans le but de gagner in fine. Lorsque c'est le tour de $\forall$, $\forall$ devine aussi un coup à jouer, pour contrer $\exists$. Pour gagner, $\exists$ doit deviner un coup de sorte que, quoi que $\forall$ joue, $\exists$ puisse trouver un coup tel que, quoi que $\forall$ joue, ..., $\exists$ arrive dans un état acceptant.

Ceci est formalisé par la condition d'acceptation des machines de Turing alternantes. Notons $C \to_{\mathcal{M}} C'$ en abrégé pour dire que $\mathcal{M}$ peut passer de C à C' en une étape.

Définition 3.1 *L'ensemble des configurations C, d'état interne q, d'une machine de Turing alternante $\mathcal{M}$ qui* mènent à l'acceptation *est le plus petit ensemble tel que :*

- *si q est un état acceptant, alors C mène à l'acceptation ;*
- *si q est un état existentiel, et s'il existe une configuration C' menant à l'acceptation telle que $C \to_{\mathcal{M}} C'$, alors C mène à l'acceptation ;*
- *si q est un état universel, et si toute configuration C' telle que $C \to_{\mathcal{M}} C'$ mène à l'acceptation, alors C mène à l'acceptation.*

On dit que $\mathcal{M}$ accepte *l'entrée x si et seulement si la configuration initiale $C_0(x)$ mène à l'acceptation.*

Une façon équivalente est de définir le fait de mener à l'acceptation par un jugement $\vdash C : \mathrm{OK}$, et des règles de déduction de la figure 1 (où l'on note q_C l'état interne de C). Autrement dit, C mène à l'acceptation si et seulement s'il existe une dérivation π (un arbre de jugements, connectés par l'une des règles ci-dessus) dont la conclusion est $\vdash C : \mathrm{OK}$.

▷ **Exercice 3.1**

Démontrer cette affirmation, c'est-à-dire que C mène à l'acceptation si et seulement si on peut dériver $\vdash C : \mathrm{OK}$ à l'aide des règles de la figure 1.

Une autre façon équivalente est de définir une *exécution* d'une machine de Turing alternante comme étant un arbre, défini comme suit.

Définition 3.2 (Exécution) *L'ensemble des* exécutions *(finies) d'une machine de Turing alternante, partant de la configuration C, d'état interne q, est le plus petit ensemble d'arbres défini comme suit :*

- *si q est un état final (acceptant ou non), alors l'arbre réduit à un sommet contenant C est une exécution partant de C ;*

— *si q est un état existentiel et $C \to_{\mathcal{M}} C'$, alors pour toute exécution T' partant de C', l'arbre obtenu à partir de T' en ajoutant une nouvelle racine étiquetée C, de fils la racine de T', est une exécution partant de C ;*
— *si q est un état universel, et $C_1, \ldots, C_m$ sont toutes les configurations C_i telles que $C \to_{\mathcal{M}} C_i$, alors tout arbre T de racine étiquetée par C, ayant m fils qui sont les racines d'exécutions partant de C_1, …, C_m respectivement, est une exécution partant de C.*

Une machine non déterministe est le cas particulier d'une machine alternante sans état universel. On retrouve alors la notion d'exécution d'une machine non déterministe, qui est juste une suite de configurations qui se suivent conformément à la table de transitions de $\mathcal{M}$. La définition ci-dessus demande en réalité de ne considérer que des exécutions qui s'arrêtent, sur un état final. Ce n'est pas une restriction dans notre cadre, où de toute façon les machines de Turing devront s'arrêter.

Lemme 3.3 *Disons qu'une exécution est* acceptante *si et seulement si toutes ses feuilles sont des configurations acceptantes. Alors la machine alternante $\mathcal{M}$ accepte l'entrée x si et seulement s'il existe une exécution acceptante partant de $C_0(x)$.*

Démonstration. Montrons que pour toute exécution acceptante T partant de C, la configuration C mène à l'acceptation. C'est une récurrence immédiate sur la taille de l'arbre T. Le cas de base, où T est réduit à un sommet, est trivial, les autres cas sont faciles. On en déduit que s'il existe une exécution acceptante partant de $C_0(x)$, alors $C_0(x)$ mène à l'acceptation, c'est-à-dire que $\mathcal{M}$ accepte x.

Réciproquement, si $\mathcal{M}$ accepte x, c'est-à-dire si $C_0(x)$ mène à l'acceptation, il existe une dérivation π de $\vdash C_0(x)$: OK à l'aide des règles de la figure 1. On produit une exécution simplement en retournant l'arbre de dérivation π (les informaticiens mettent les racines des arbres en haut !), et en effaçant les symboles "$\vdash$" et ": OK". Elle est nécessairement acceptante, puisque ses feuilles sont obtenues par la règle du haut de la figure 1, et sont donc de la forme C, avec q_C un état acceptant. □

Définition 3.4 (Trace) *Une* trace *d'une machine de Turing $\mathcal{M}$ (déterministe, non déterministe, ou alternante) est une suite finie ou infinie de configurations $C_0 \to_{\mathcal{M}} C_1 \to_{\mathcal{M}} \ldots \to_{\mathcal{M}} C_k \to_{\mathcal{M}} \ldots$.*

On dit que $\mathcal{M}$ termine *en temps $f(n)$, où n est la taille de l'entrée x, si et seulement toutes les traces de $\mathcal{M}$ partant de la configuration initiale $C_0(x)$ sont finies, et de longueur au plus $f(n)$ pour tout x de taille n.*

$\mathcal{M}$ termine en espace *$f(n)$ si et seulement si $\mathcal{M}$ termine, et l'espace utilisé par toutes les configurations dans toute trace partant de $C_0(x)$ est majoré par $f(n)$, n étant la taille de x.*

Cette définition généralise les notions usuelles pour les machines déterministes ou non déterministes, au cas des machines alternantes.

3.1 Le théorème de Chandra-Kozen-Stockmeyer I : temps alternant=espace

On notera **ATIME**($f(n)$) la classe des languages décidables en temps $f(n)$ sur machine alternante (on dira même plus couramment "en temps alternant $f(n)$"), **ASPACE**($f(n)$) la classe des langages décidables en espace alternant $f(n)$, et de même pour **AL** (espace logarithmique alternant), **AP** (temps polynomial alternant), **APSPACE** (espace polynomial alternant), etc.

▷ **Exercice 3.2**
Montrer un théorème de speedup en espace pour les machines alternantes : **ASPACE**($O(f(n))$) = **ASPACE**($f(n)$).

La classe des machines alternantes peut sembler bizarre... mais nous venons de voir, via le théorème de Stockmeyer-Meyer, un exemple typique de problème qui se résout naturellement en temps polynomial alternant :

Lemme 3.5 *QBF est dans* **AP**.

Démonstration. Il suffit de laisser ∃loïse deviner les valeurs des variables existentielles, et de faire trouver les valeurs des variables universelles par ∀bélard. Le nombre de quantificateurs est, par définition, inférieur ou égal à la taille de l'entrée n. Cette suite de coups entre ∃loïse et ∀bélard, entremêlée des opérations déterministes (faites par n'importe quel joueur) de lecture des quantificateurs de la formule $F = Q_1 x_1 \cdot \ldots \cdot Q_n x_n \cdot S$ en entrée, prend donc un temps polynomial (et même linéaire) en n. On évalue ensuite S avec les valeurs choisies de $x_1, \ldots, x_n$ en temps polynomial déterministe (donc peu importe le joueur). □

Corollaire 3.6 **PSPACE** $\subseteq$ **AP**.

Démonstration. **AP** est clairement stable par réductions en temps polynomial, donc aussi par les réductions en espace logarithmique. Puisque QBF est complet pour **PSPACE** par le théorème de Stockmeyer-Meyer 2.21, et appartient à **AP** par le lemme 3.5, c'est que tout langage de **PSPACE** est dans **AP**. □

Ce corollaire est en fait la direction difficile du théorème suivant, dû à Chandra, Kozen, et Stockmeyer.

Théorème 3.7 (Chandra, Kozen, Stockmeyer I) **PSPACE** = **AP**.

Démonstration. Il suffit de montrer que tout ce que l'on peut faire en temps polynomial alternant peut aussi se faire en espace polynomial. Ceci se fait par simulation d'une machine alternante $\mathcal{M}$ en temps polynomial $p(n)$ sur une machine déterministe, par *backtracking*. Autrement dit, on écrit l'algorithme récursif suivant, décidant si C est une configuration menant à l'acceptation :

```
ACC (C) =
        si q_C est acceptant alors retourner vrai ;
        sinon, si q_C est rejetant alors retourner faux ;
        sinon, si q_C est existentiel
                alors { pour chaque C' tel que C →_M C' :
                                si ACC (C') = vrai alors retourner vrai ;
                        retourner faux ;
                      }
        sinon (* q_C est universel. *)
                alors { pour chaque C' tel que C →_M C' :
                                si ACC (C') = faux alors retourner faux ;
                        retourner vrai ;
                      }
```

La profondeur de la pile est majorée par la taille maximale des traces de $\mathcal{M}$, soit $p(n)$. À chaque appel, on doit créer une variable locale C', qui est une configuration de taille au plus $p(n)$. Cet algorithme utilise donc un espace $O(p^2(n))$. □

La morale de ce théorème est : "le temps alternant, c'est de l'espace". On a quand même besoin pour que ceci soit vrai de parler de temps alternant polynomial, et d'espace polynomial, et plus généralement de temps alternant ou d'espace borné par des fonctions $f(n)$ suffisamment grandes, comme nous le verrons plus bas. Mais ceci est faux lorsque le temps alternant ou l'espace disponible est trop faible : voir l'exercice 3.3 par exemple.

▷ **Exercice 3.3**

Montrer que $\mathbf{ATIME}(\log n) \neq \mathbf{L}$. (Indication : montrer que l'on peut tester si un mot en entrée est un palindrome en espace logarithmique, mais pas en temps, même alternant, logarithmique. On rappelle qu'un palindrome est un mot w tel que la suite des lettres de w lues à l'envers redonne w.)

On peut généraliser le théorème 3.7, grâce à une technique simple (et algorithmiquement délirante), celle du *bourrage* ("padding" en anglais). On note $\#^k$ le mot de longueur k obtenu par répétition du caractère #.

Proposition 3.8 (Bourrage) *Soit L un langage décidable en temps (resp., espace ; déterministe, resp. non déterministe, resp. alternant) $f(n)$, et supposons que $f(n) \geq n$ pour n assez grand, disons $n \geq n_0$, et que f soit propre.*

Le langage L' des mots de la forme $x\#^{f(n)-n}$, où $x \in L$, n est la longueur de x, $n \geq n_0$, et # est un caractère hors de l'alphabet de x, est décidable en temps (resp., espace ; déterministe, resp. non déterministe, resp. alternant) linéaire.

Démonstration. L'idée de base est simple. Sur l'entrée x', de taille n', on décide si $x' \in L'$ comme suit. D'abord, on vérifie que x' est de la forme $x\#^k$ pour un certain k, sinon on rejette. En posant $n = n' - k$, la longueur de x, on vérifie que $f(n) = n'$, puis on extrait x de x' et on teste si $x \in L$ en temps $f(n)\ldots$ qui est bien linéaire en n' !

Il y quelques subtilités dont il faut se prémunir, cependant.

D'abord, on ne peut pas vérifier que $f(n) = n'$ en calculant simplement $f(n)$ et et le comparant à n'. Pourtant, cela a l'air correct. Il suffit de recopier le préfixe x de x' précédant le premier # sur une nouvelle bande, et de calculer $f(n)$ à partir de cette bande, puis de vérifier que $f(n)$ (écrit en unaire) a exactement la mme longueur que x'. Le problème est que ceci prend un temps $O(n + f(n))$ et un espace $O(f(n))\ldots$ alors qu'on ne sait pas encore si $f(n) = n'$! Par exemple, imaginons que f soit la fonction exponentielle de base 2, mais que n soit gros, disons $n = n' - 1$ (un seul caractère #). Pour tester si $f(n) = n'$ (ce qui ne sera pas le cas ici), il nous faudra un temps de l'ordre de $n + 2^n \cong n' + \frac{1}{2}2^{n'}$ et un espace de l'ordre de $2^{n'}$, ce qui n'est pas linéaire du tout. Il est facile de corriger ce problème : on *simule* l'excution de la machine qui calcule $f(n)$ de sorte qu'elle écrive sur une bande de sortie que l'on aura préremplie de n' caractères # suivis d'un caractère de fin de bande \$; dans la simulation, la bande de sortie devient une bande de travail, et lors de chaque écriture sur la bande de sortie, on vérifie d'abord que l'on ne va pas réécrire par-dessus le caractère \$; si c'est le cas, on rejette (n trop grand). Si cette simulation réussit, alors $f(n) \leq n'$. On vérifie que $f(n) = n'$ en vérifiant qu'à la fin de la simulation, la bande ne contient plus que des blancs suivis d'un \$.

Maintenant que l'on sait que $f(n) = n'$, et *uniquement maintenant*, on peut tester si $x \in L$ en temps linéaire en n'. Il ne faut surtout pas tester si $x \in L$ avant d'avoir fait la vérification de format et de longueur.

Finalement, il ne faut pas oublier de vérifier que $n \geq n_0$. Comme n_0 est une constante, on peut le faire lors de l'extraction de x à partir de x', en maintenant un compteur de 0 à n_0 dans le contrôle interne de la machine de Turing, que l'on incrémente à chaque lecture d'un caractère de x tant qu'il n'a pas atteint n_0. À la fin de l'extraction, on rejette si le compteur n'a pas atteint n_0. □

On en déduit facilement, le théorème suivant, où $\mathbf{SPACE}(poly(f(n)))$ est une abréviation pour l'union des classes $\mathbf{SPACE}(p(f(n)))$, p parcourant l'ensemble des polynômes à coefficients entiers positifs en n, et de même pour les autres classes.

Théorème 3.9 *Soit f une fonction propre telle que $f(n) \geq n$ pour n assez grand. Alors* $\mathbf{SPACE}(poly(f(n))) = \mathbf{ATIME}(poly(f(n)))$.

Démonstration. Soit n_0 un entier tel que $f(n) \geq n$ pour tout $n \geq n_0$.

Supposons que L soit décidable en espace $p(f(n))$ pour un certain polynôme p. Sans perte de généralité, nous pouvons supposer $p(n') \geq n'$ pour tout n', de sorte que $p \circ f$ est une fonction propre avec $(p \circ f)(n) \geq n$ pour tout $n \geq n_0$.

Le langage L' obtenu par bourrage, des mots de la forme $x\#^{(p\circ f)(n)-n}$, n étant la taille de x, est décidable en espace linéaire par la proposition 3.8, donc $L' \in \mathbf{PSPACE}$. Par le théorème 3.7, $L' \in \mathbf{AP}$.

On décide si $x \in L$ comme suit. Si x est de longueur strictement plus petite que n_0, on répond la bonne réponse en tabulant. Sinon, on commence par écrire $(p\circ f)(n)-n$ caractères # à la suite de x, ce qui prend un temps $O(p(f(n)))$, $p\circ f$ étant propre, et il ne reste plus qu'à décider si $x\#^{(p\circ f)(n)-n}$ est dans L', ce qui se fait en temps alternant polynomial en $(p \circ f)(n)$. Donc $L \in \mathbf{ATIME}(poly(p(f(n)))) \subseteq \mathbf{ATIME}(poly(f(n)))$.

Réciproquement, si L est décidable en temps alternant $p(f(n))$, le langage L' obtenu par bourrage est décidable en temps alternant linéaire, donc en espace polynomial. De même que ci-dessus, on en déduit que L est décidable en espace $poly(f(n))$. □

▷ **Exercice 3.4**

Notons **EXPSPACE** la classe des langages décidables en espace exponentiel (c'est-à-dire, comme toujours, majorée par l'exponentielle d'un polynôme), et **AEXPTIME** celle des langages décidables en temps exponentiel alternant. Montrer que **EXPSPACE** = **AEXPTIME**.

3.2 Circuits et clauses de Horn

Si le théorème 3.7 et le théorème 3.9 montrent que "le temps alternant, c'est l'espace", qu'est-ce que l'espace alternant ? Nous allons montrer que c'est une *exponentielle* du temps. C'est l'autre résultat de Chandra, Kozen, et Stockmeyer.

Pour ceci, nous allons considérer les trois problèmes suivants. D'abord, HORN-SAT :
ENTRÉE : un ensemble fini S de clauses de Horn propositionnelles.
QUESTION : S est-il satisfiable ?

Une *clause de Horn* est une clause contenant au plus un littéral positif, c'est-à-dire sans signe $\neg$. On écrira souvent, parce que c'est plus pratique, une clause avec exactement un littéral positif $x \vee \neg x_1 \vee \ldots \vee \neg x_k$ (une clause *définie*) sous la forme $x \Leftarrow x_1, \ldots, x_k$; et une clause n'ayant pas de littéral positif $\neg x_1 \vee \ldots \vee \neg x_k$ (un *but*) sous la forme $\bot \Leftarrow x_1, \ldots, x_k$.

Ensuite, CIRCUIT VALUE :
ENTRÉE : un circuit clos $\mathcal{C}$.
QUESTION : $\mathcal{C}$ est-il vrai ?

Un *circuit clos* est un graphe orienté acyclique G, dont tous les sommets sont étiquetés par un symbole pris parmi $\wedge$, $\vee$, $\overline{\wedge}$, $\overline{\vee}$, plus un sommet distingué p (la *sortie*). La *valeur* d'un sommet u dans un circuit est définie comme étant la conjonction (resp., la disjonction, resp., la négation de la conjonction, resp., la négation de la disjonction) des valeurs des sommets v tels que $v \to u$ (des *prédécesseurs* de u), lorsque u est étiqueté $\wedge$ (resp., $\vee$, $\overline{\wedge}$, $\overline{\vee}$). Ceci est bien défini car le graphe est acyclique. La récurrence se termine sur les sommets u qui n'ont aucun prédécesseur v. Dans ce cas, u vaut toujours **vrai** si u est étiqueté $\wedge$ ou $\overline{\vee}$, et **faux** si u est étiqueté $\vee$ ou $\overline{\wedge}$. La valeur du circuit est par convention la valeur de la sortie p dans le circuit.

Un circuit clos est essentiellement la même chose qu'une formule propositionnelle construite à l'aide des opérateurs $\wedge$ (et), $\vee$ (ou), $\overline{\wedge}$ (non et), et $\overline{\vee}$ (non ou), à une différence essentielle près : plusieurs sommets peuvent être partagés. Autrement dit, là où la formule $(\mathbf{vrai} \wedge \mathbf{faux}) \vee (\mathbf{vrai} \wedge \mathbf{faux})$ est de taille 7, le circuit correspondant n'a que 4 sommets (un sommet **vrai** étiqueté $\wedge$, sans prédécesseur, un sommet **faux** étiquété $\vee$, sans prédécesseur, un sommet u étiqueté $\wedge$ et ayant **vrai** et **faux** pour prédécesseurs, et le puits étiqueté $\vee$ et ayant [deux fois ?] u comme prédécesseur).

Finalement, MONOTONE CIRCUIT VALUE :
ENTRÉE : un circuit clos monotone $\mathcal{C}$.
QUESTION : $\mathcal{C}$ est-il vrai ?

Un circuit clos est *monotone* si et seulement si aucun des ses sommets n'est étiqueté $\overline{\wedge}$ ou $\overline{\vee}$. Autrement dit, il est construit sans négation.

Lemme 3.10 *MONOTONE CIRCUIT VALUE* $\preceq_{\mathbf{L}}$ *CIRCUIT VALUE* $\preceq_{\mathbf{L}}$ *HORN-SAT.*

Démonstration. La réduction de gauche est triviale. Pour celle de droite, on prend en entrée un circuit clos $\mathcal{C}$. On crée, pour chaque sommet u, deux variables propositionnelles $+u$ ("u est vrai") et $-u$ ("u est faux"). Il n'est pas besoin de les créer explicitement, et il suffit de convenir qu'on les obtient en écrivant juste un signe, + ou −, devant les numéros de sommet u, les numéros étant écrits dans un alphabet ne contenant ni + ni −. À réduction logspace près, on peut supposer que le graphe G sous-jacent à $\mathcal{C}$ est donné sous forme de la liste d'adjacence du graphe où toutes les arêtes ont été renversées. (La réduction logspace à partir de la matrice d'adjacence suit un principe similaire à celui de l'exercice 2.2, par exemple.) Pour chaque sommet u de G, de prédécesseurs (données directement dans la liste d'adjacence) $v_1, \ldots, v_n$, produire les clauses :

— (si u est étiqueté $\wedge$) $+u \Leftarrow +v_1, \ldots, +v_n$ et d'autre part $-u \Leftarrow -v_1, \ldots, -u \Leftarrow -v_n$;
— (si u est étiqueté $\vee$) $+u \Leftarrow +v_1, \ldots, +u \Leftarrow +v_n$ et d'autre part $-u \Leftarrow -v_1, \ldots, -v_n$;
— (si u est étiqueté $\overline{\wedge}$) $-u \Leftarrow +v_1, \ldots, +v_n$ et d'autre part $+u \Leftarrow -v_1, \ldots, +u \Leftarrow -v_n$;
— (si u est étiqueté $\overline{\vee}$) $-u \Leftarrow +v_1, \ldots, -u \Leftarrow +v_n$ et d'autre part $+u \Leftarrow -v_1, \ldots, -v_n$.

Ceci se fait clairement en espace logarithmique. Par construction, ces clauses expriment exactement l'équivalence logique du fait que u soit vrai (resp., faux) dans le circuit avec la combinaison booléenne adéquate des valeurs de vérité des prédécesseurs $v_1, \ldots, v_n$. On produit ensuite la clause $\neg(-p)$, où p est le puits. Soit S l'ensemble de clauses résultant, et S^- l'ensemble des clauses sauf la dernière. Si la valeur de $\mathcal{C}$ est **faux**, et si toutes les clauses de S^- sont vraies alors $+p$ est nécessairement faux, ce qui contredit $\neg(-p)$, donc S est insatisfiable. Réciproquement, si la valeur de $\mathcal{C}$ est **vrai**, déclarons $+u$ vrai et $-u$ faux si la valeur de u dans $\mathcal{C}$ est vraie, et déclarons $+u$ faux et $-u$ vrai sinon. Toutes les clauses de S^- sont vraies par construction, et $+p$ aussi, donc $\neg(-p)$ aussi : nous avons donc un modèle de S. □

Pour étudier HORN-SAT, nous allons faire quelques remarques. Si S est un ensemble de clauses de Horn, et A un ensemble de variables propositionnelles (ou $\bot$), posons $T_S(A)$ l'ensemble des x (variable ou $\bot$) tels que S contienne une clause $x \Leftarrow x_1, \ldots, x_n$ avec $x_1, \ldots, x_n \in A$. L'opérateur T_S est croissant dans le treillis (complet, et en fait fini) des ensembles de variables propositionnelles de S. Il a donc un plus petit point fixe $lfp\ T_S$, qui est calculé comme $T_S^m(\emptyset)$ pour m assez grand. On note que, si l'on identifie un ensemble A de variables à une valuation des variables (**vrai** pour toute variable dans l'ensemble, **faux** pour les autres), A est un modèle de S (c'est-à-dire satisfait toutes les clauses de S) si et seulement si A est un point fixe de T_S. Si $lfp\ T_S$ est un ensemble de variables, c'est-à-dire s'il ne contient pas $\bot$, c'est donc un modèle de S, et S est satisfiable. Sinon, $\bot \in T_S^m(\emptyset)$ pour m assez grand. On peut reconstruire par récurrence sur m une preuve de $\bot$ à partir des clauses de S, ce qui montre que S est alors insatisfiable. Nous venons de montrer que S était insatisfiable si et seulement si $\bot \in lfp\ T_S$.

$$\frac{\vdash_S x_1 \ldots \vdash_S x_n}{\vdash_S x} (x \Leftarrow x_1, \ldots, x_n) \in S$$

FIGURE 2 – Calcul des conséquences d'un ensemble de clauses de Horn

Ceci peut aussi se formaliser à l'aide d'un système déductif, voir la figure 2 : les variables (ou $\bot$) x de $lfp\ T_S$ sont exactement celles telles que l'on peut dériver $\vdash_S x$ par la règle de la figure 2.

Lemme 3.11 *HORN-SAT* $\in \mathbf{P}$.

Démonstration. Nous calculons $lfp\ T_S$ directement, par une variante de l'algorithme de Davis-Putnam. (L'algorithme suivant est très rapide, et on laissera le soin au lecteur curieux de comprendre par lui-même comment il fonctionne, et pourquoi il termine en temps polynomial.) Soit *nvar* le nombre de variables dans S, *nc* le nombre de clauses.

pile := [] ; (* pile de variables connues pour être vraies
mais non encore traitées. *)
num_clause := 0 ;
clauses := un tableau de *nc* élements ;
index := un tableau de *nvar* listes, initialement vides ;
(* *index*[x] sera une liste de tous les numéros de clause
ayant x à droite du signe $\Leftarrow$. *)
pour chaque clause $x \Leftarrow x_1, \ldots, x_n$ de S : {
(* avec x variable ou $\bot$, et l'on supposera $x_1, \ldots, x_n$ distincts. *)
si $n = 0$ **alors** empiler x sur *pile* ;
sinon { incrémenter *num_clause* ;
clauses[*num_clause*] := (n, x) ; (* n est le nombre de variables
parmi $x_1, \ldots, x_n$ dont on ne sait pas encore si elles sont vraies. *)
pour $i = 1$ à n : ajouter *num_clause* à la liste *index*[x_i] ;
}
}
$A := \emptyset$;
tant que *pile* $\neq$ [] : {
dépiler un élément x de *pile* ;
si $x = \bot$ **alors rejeter** ;
sinon { ajouter x à A ;
pour tout $c \in$ *index*[x] : {
soit $(n, y) =$ *clauses*[c] ;
si $n = 1$ **alors** empiler y sur *pile* ; (* y est vrai ! *)
sinon *clauses*[c] := $(n - 1, y)$;
}
}
retourner A ;

□

Lemme 3.12 *MONOTONE CIRCUIT VALUE est* **AL***-difficile : tout langage L de* **AL** *est logspace-réductible à MONOTONE CIRCUIT VALUE.*

Démonstration. Soit $\mathcal{M}$ une machine de Turing alternante en espace $\log n$ décidant L. Pour des raisons techniques, nous en déduisons une autre machine $\mathcal{M}'$, décidant aussi L en espace $O(\log n)$, disons $k \log n$. La machine $\mathcal{M}'$ fonctionne comme $\mathcal{M}$, à ceci près qu'elle a une bande supplémentaire, qui compte le nombre d'instructions exécutées par $\mathcal{M}$; cette bande est initialisée à la bande vide (représentant 0). On ne demande pas spécialement que $\mathcal{M}'$ rejette lorsque le compteur dépasse une valeur de l'ordre de $a^{\log n} = n^{\log a}$. Le compteur ne sert pas à forcer $\mathcal{M}'$ à terminer, mais juste à garantir que : $(*)$ il n'y a aucune chaîne infinie de configurations de la forme $C_1 \leftarrow_{\mathcal{M}'} C_2 \leftarrow_{\mathcal{M}'} \ldots \leftarrow_{\mathcal{M}'} C_j \leftarrow_{\mathcal{M}'} \ldots$. (Ceci est vrai car le compteur décroîtrait le long de cette chaîne. Cette propriété $(*)$ peut sembler évidente déjà pour $\mathcal{M}$, mais nous voulons qu'elle soit vraie même pour des configurations C_j inaccessibles depuis aucune configuration $C_0(x)$ d'entrée.)

Considérons une entrée x de taille n. On construit un circuit ressemblant énormément au graphe de toutes les configurations de $\mathcal{M}'$, mais en inversant les arcs. Les sommets sont toutes les configurations de taille au plus $k \log n$. Il y a un arc $C \to C'$ si $C' \to_{\mathcal{M}'} C$ (et l'état de C' n'est pas un état final ; les états finaux n'auront pas de prédécesseur). Si l'état de C est final, alors C est étiqueté $\wedge$ si c'est un état acceptant, $\vee$ sinon. Si l'état de C' est existentiel, on l'étiquette $\vee$, sinon $\wedge$. Ceci forme bien un circuit, le graphe sous-jacent étant acyclique par la propriété $(*)$. On décide enfin que la sortie du circuit est la configuration initiale $C_0(x)$. Cette réduction prend clairement un espace logarithmique, et il est facile de voir (à l'aide des dérivations de jugements de la forme $\vdash C : \mathrm{OK}$), que $\mathcal{M}'$ accepte sur l'entrée x si et seulement si la valeur du circuit est **vrai**. □

3.3 Le théorème de Chandra-Kozen-Stockmeyer II : espace alternant=exp(temps)

On déduit des résultats de la section précédente, d'abord, que :

Corollaire 3.13 $\mathbf{AL} \subseteq \mathbf{P}$.

Démonstration. Tout langage L de **AL** se réduit en espace logarithmique à MONOTONE CIRCUIT VALUE par le lemme 3.12, donc à HORN-SAT par le lemme 3.10 et parce que $\preceq_{\mathbf{L}}$ est transitive (lemme 2.11). D'autre part, HORN-SAT est dans **P** par le lemme 3.11, et **P** est stable par réductions en temps polynomial, en particulier par les réductions en espace logarithmique. Donc $L \in \mathbf{P}$. □

Alors que l'algorithme du lemme 3.11 déduit de nouveaux faits en avant, on peut chercher directement une dérivation de $\vdash_S \bot$ en arrière. Ceci mène directement au résultat :

Lemme 3.14 *HORN-SAT* $\in \mathbf{AL}$.

Démonstration. La machine maintient sur une bande une variable (ou $\bot$) x, et cherche à démontrer $\vdash_S x$ par la règle de la figure 2. $\exists$loïse devine une clause $x' \Leftarrow x_1, \ldots, x_p$ de S et vérifie que $x' = x$. (Ceci revient à deviner une position sur la bande d'entrée : on ne recopie pas la clause, ce qui risquerait de prendre un espace non logarithmique, p n'étant pas majoré par un logarithme de la taille de l'entrée !) Si $p = 0$, la machine accepte. Sinon, $\forall$bélard choisit ensuite universellement un x_i, $1 \leq i \leq p$, autrement dit l'on continue en vérifiant tous les sous-buts x_1, ..., x_p. Telle qu'elle est, la machine ne termine pas forcément. On maintient donc un compteur sur une seconde bande, qui est initialisé au nombre de variables de S plus 1, et qui est décrémenté à chaque coup d'$\exists$loïse. Lorsqu'il passe à 0, on rejette. Ceci reste complet, car s'il existe une dérivation de $\vdash_S \bot$, il en existe une de taille minimale, laquelle ne cherchera pas à démontrer deux fois la même variable (ou $\bot$). □

On en déduit immédiatement que MONOTONE CIRCUIT VALUE, CIRCUIT VALUE, et HORN-SAT sont tous **AL**-complets. Mais nous allons dire quelque chose d'encore plus précis.

Lemme 3.15 *HORN-SAT est* **P***-difficile : tout langage L de* **P** *est logspace-réductible à HORN-SAT.*

Démonstration. Le principe est essentiellement le même que pour le théorème de Cook 2.14. Soit $\mathcal{M}$ une machine décidant le *complémentaire* de L en temps polynomial $p(n)$. Ceci est faisable, puisque **P** est clos par complémentaire. Pour simplifier, on peut supposer que $\mathcal{M}$ est une machine de Turing à une bande servant à la fois d'entrée, de bande de travail, et de sortie. (C'est un théorème classique, que nous admettrons.)

On crée un sommet (i, j) et un sommet $\overline{(i, j)}$ par case (i, j) du tableau de calcul de $\mathcal{M}$, c'est-à-dire représentant le fait que le bit numéro j de la configuration à l'étape numéro i de calcul de $\mathcal{M}$ sur l'entrée x est vrai, resp. faux. Chaque bit (i, j) $(i \geq 1)$ dépend d'un nombre constant de bits de la forme $(i-1, k)$. Nous supposerons que les configurations seront décrites sous forme d'un mot wqw', où w et w' sont formés sur l'alphabet de la bande (w décrit la partie de la bande à gauche de la tête, w' la partie droite), et q décrit l'état interne de $\mathcal{M}$ sur un alphabet disjoint (et en un nombre *constant* de lettres, codé sur un nombre constant, disons k_0 de bits). Nous supposerons aussi que $\mathcal{M}$ n'atteint l'état d'acceptation qu'après avoir repositionné la tête au début de la bande (donc les seules configurations finales sont de la forme $q_{accept}w'$, où q_{accept} est l'unique état acceptant). Ceci peut se faire en modifiant la machine pour que, une fois qu'elle atteint son état d'acceptation q, elle ramène la tête au début de la bande puis passe dans un état nouveau q_{accept}, qui devient le seul état acceptant. Nous construisons ensuite une machine $\mathcal{M}'$ qui fonctionne comme $\mathcal{M}$, mais qui, au lieu de s'arrêter lorsqu'elle atteint une configuration finale, boucle dessus. $\mathcal{M}'$ ne termine plus, mais atteint une configuration acceptante sur l'entrée x en temps *exactement* $p(n)$ si et seulement si $x \notin L$. (On rappelle que $\mathcal{M}$ décide le complémentaire de L.)

La valeur de chaque bit (i, j) du tableau dépend alors uniquement des valeurs des bits $(i-1, k)$, k étant dans un intervalle $[j-\ell, j+\ell]$ de largeur constante contenant j. Plus précisément, on sait en consultant la table de transitions de $\mathcal{M}'$ que le bit (i, j) sera vrai si une certaine formule F_{ij} sans négation dépendant au plus de $(i-1, j-\ell)$, $\overline{(i-1, j-\ell)}$,

$(i-1, j-\ell+1)$, $\overline{(i-1, j-\ell+1)}$, ..., $(i-1, j+\ell)$, $\overline{(i-1, j+\ell)}$ est vraie. De plus, F_{ij} est de taille constante. On écrit alors la forme clausale de l'implication $F_{ij} \Rightarrow (i, j)$ (un nombre constant de clauses, de taille constante, F_{ij} étant de taille constante). On écrit de même la forme clausale de l'implication $\overline{F}_{ij} \Rightarrow \overline{(i, j)}$, où $\overline{F}_{ij}$ est une formule de taille constante ne dépendant que de $(i-1, j-\ell)$, $\overline{(i-1, j-\ell)}$, $(i-1, j-\ell+1)$, $\overline{(i-1, j-\ell+1)}$, ..., $(i-1, j+\ell)$, $\overline{(i-1, j+\ell)}$ exprimant la condition à laquelle le bit (i, j) est faux. On code ensuite l'entrée en énumérant les valeurs de j, et en produisant la clause $(0, j)$ si le bit j de $C_0(x)$ est vrai, la clause $\overline{(0, j)}$ sinon.

Nous supposerons par commodité que q_{accept} s'écrit sous forme de la suite de k_0 bits égaux à 1. On écrit finalement la clause $\bot \Leftarrow (p(n), 0), (p(n), 1), \ldots, (p(n), k_0 - 1)$. Il est alors clair que l'ensemble de clauses S ainsi produit est insatisfiable si et seulement si l'on peut déduire $\bot$ par l'unique règle de la figure 2, si et seulement si $\mathcal{M}'$ atteint une configuration acceptante en partant de l'entrée x, si et seulement si $x \notin L$. Donc S est satisfiable si et seulement si $x \in L$. La réduction est de plus en espace logarithmique. □

On peut voir que l'ensemble de clauses de Horn ci-dessus peut se recoder en un circuit : il n'y a en effet aucune circularité entre variables induites par les clauses produites. De plus, le circuit est monotone. Avec un peu plus de travail, on pourrait donc modifier la démonstration ci-dessus pour montrer que MONOTONE CIRCUIT VALUE et CIRCUIT VALUE sont aussi **P**-difficiles. Mais on peut conclure la même chose par des moyens plus simples, quoiqu'un peu moins concrets. On commence par observer que le théorème de speedup en espace, et le lemme 2.11 de composition des fonctions bornées en espace, sont toujours vrais dans le cas de machines alternantes. Les démonstrations sont des adaptations triviales de celles du théorème 2.2 et du lemme 2.11.

Lemme 3.16 *Pour toute fonction f,* $\mathbf{ASPACE}(O(f(n))) = \mathbf{ASPACE}(f(n))$.

Lemme 3.17 *Soit $\mathcal{M}'$ une machine de Turing alternante en espace $g(n)$ calculant une fonction G (resp., décidant de l'appartenance à un langage L), avec $g(n) = \Omega(\log n)$, et F une fonction calculable en espace $h(n)$, cette fonction étant propre. Soit a le nombre de lettres de l'alphabet de bandes utilisé par la machine calculant F. On peut fabriquer une machine de Turing alternante $\mathcal{M}''$ en espace $g(a^{h(n)})$ et pas plus, qui calcule $G \circ F$ (resp., décidant sur l'entrée x si $F(x) \in L$).*

En particulier, pour $h(n) = \log n$, et $g(n) = \log n$, ceci montre que **AL** est stable par réductions logspace : si $L \preceq_{\mathbf{L}} L'$ et $L' \in \mathbf{AL}$, alors $L \in \mathbf{AL}$.

Théorème 3.18 (Chandra, Kozen, Stockmeyer II) $\mathbf{P} = \mathbf{AL}$. *HORN-SAT, CIRCUIT VALUE et MONOTONE CIRCUIT VALUE sont* **P***-complets.*

Démonstration. Pour tout langage L de **P**, on a $L \preceq_{\mathbf{L}}$ HORN-SAT par le lemme 3.15. Or HORN-SAT est dans **AL** par le lemme 3.14. Par le lemme 3.17, **AL** est stable par réductions en espace logarithmique. (C'est tout l'intérêt des réductions en espace logarithmique ici ! **AL** sera même stable par les réductions en temps polynomial, mais nous ne le savons pas encore,

puisque ceci revient à démontrer $\mathbf{P} \subseteq \mathbf{AL}$.) Donc $L \in \mathbf{AL}$. Comme L est arbitraire, $\mathbf{P} \subseteq \mathbf{AL}$. L'inclusion inverse est le corollaire 3.13. Donc $\mathbf{AL} = \mathbf{P}$.

HORN-SAT est dans $\mathbf{P}$ par le lemme 3.11, donc CIRCUIT VALUE et MONOTONE CIRCUIT VALUE aussi par le lemme 3.10. MONOTONE CIRCUIT VALUE est **AL**-difficile par le lemme 3.12, donc **P**-difficile ; donc CIRCUIT VALUE et HORN-SAT aussi par le lemme 3.10. Tous ces problèmes sont donc **P**-complets. □

Appliquons maintenant la technique du bourrage. On note $\mathbf{TIME}(2^{O(f(n))})$ la classe $\bigcup_{k\geq 1} \mathbf{TIME}(2^{kf(n)}) = \mathbf{TIME}(poly(2^{f(n)}))$, où $\mathbf{TIME}(g(n))$ dénote la classe des langages décidables en temps déterministe $g(n)$ (pour n assez grand).

Théorème 3.19 *Soit f une fonction propre telle que $f(n) \geq n$ pour n assez grand. Alors* $\mathbf{TIME}(2^{O(f(n))}) = \mathbf{ASPACE}(f(n))$.

Démonstration. Comme au théorème 3.9, avec juste quelques modifications mineures. Si L est décidable en temps $2^{kf(n)}$ ($k \geq 1$), alors le langage L', obtenu par bourrage, des mots de la forme $x\#^{2^{kf(n)}-n}$, est décidable en temps linéaire (proposition 3.8), donc en temps polynomial, donc en espace alternant logarithmique.

Pour décider L, sur l'entrée x, on écrit $kf(n)$ blancs sur une bande auxiliaire, puis on se sert de cette bande comme d'un compteur sur $kf(n)$ bits, initialisé à n. On recopie x sur la bande de sortie. À chaque incrémentation du compteur (jusqu'à débordement), on écrit un # sur la bande de sortie. Ceci fournit une réduction en espace $k(f(n))$ de L à L'. Par le lemme 3.17, la composition de ceci avec un algorithme en espace alternant logarithmique décidant L' fournit un algorithme en espace alternant $O(f(n))$ décidant L. Donc $L \in \mathbf{ASPACE}(f(n))$, par le lemme de speedup 3.16.

Réciproquement, si L est décidable en espace alternant $f(n)$, soit $\mathcal{M}$ une machine alternante décidant le *complémentaire* de L en espace $f(n)$. Elle a un nombre de configurations de taille $f(n)$ de l'ordre de $a^{f(n)} = 2^{O(f(n))}$. On crée une variable propositionnelle A_C pour chaque configuration C de taille $f(n)$. Pour chaque configuration acceptante C, on écrit la clause A_C. Pour chaque configuration C non finale, de successeurs $C_1, \ldots, C_m$, on écrit soit la clause $A_C \Leftarrow A_{C_1}, \ldots, A_{C_m}$ si l'état de C est universel, soit les m clauses $A_C \Leftarrow A_{C_i}$, $1 \leq i \leq m$. On écrit enfin la clause $\bot \Leftarrow A_{C_0(x)}$. L'ensemble résultat de clauses est insatisfiable si et seulement si $x \notin L$, et se calcule en temps polynomial en $2^{O(f(n))}$. En appliquant l'algorithme du lemme 3.11, on décide ainsi si $x \in L$ en temps $2^{O(f(n))}$. □

Soit **EXPTIME** la classe des langages décidables en temps exponentiel, autrement dit **EXPTIME** est l'union des $\mathbf{TIME}(2^{p(n)})$, p parcourant les polynômes à coefficients entiers positifs.

Corollaire 3.20 $\mathbf{EXPTIME} = \mathbf{APSPACE}$.

Démonstration. Si $L \in \mathbf{EXPTIME}$, il existe un polynôme p tel que $L \in \mathbf{TIME}(2^{p(n)})$, donc $L \in \mathbf{ASPACE}(p(n)) \subseteq \mathbf{APSPACE}$ par le théorème 3.19. Si $L \in \mathbf{APSPACE}$, il existe un polynôme p tel que $L \in \mathbf{ASPACE}(p(n))$, donc $L \in \mathbf{TIME}(2^{O(p(n))}) \subseteq \mathbf{EXPTIME}$, encore par le théorème 3.19. □

Ce dernier corollaire est très pratique. Il est très malcommode en général de montrer qu'un langage donné est **EXPTIME**-complet, à moins de lui réduire un problème déjà connu pour être **EXPTIME**-complet. Coder les exécutions d'une machine de Turing *déterministe* en temps exponentiel est difficile directement. Coder les exécutions d'une machine alternante en espace polynomial est en général beaucoup plus immédiat.

Terminons cette section sur l'intuition sous-jacente aux deux théorèmes de Chandra, Kozen et Stockmeyer. Un langage de **PSPACE** (= **AP**), typiquement QBF, c'est un jeu entre ∃loïse et ∀bélard qui termine après un nombre polynomial de coups. Imaginons par exemple une version sur un damier à $n \times n$ cases du jeu d'Othello : chaque joueur plaçant à chaque tour un nouveau pion sur le damier, le jeu termine en au plus $n \times n$ étapes. C'est un cas typique de jeu **PSPACE**. (Techniquement parlant, les joueurs peuvent être forcés de passer, mais le jeu s'arrête si les deux joueurs sont forcés de passer, et l'argument reste essentiellement valide.)

En revanche, un jeu sur un damier fini $n \times n$ dans lequel le nombre de coups n'est pas borné a priori (le jeu d'échecs, modulo quelques détails pénibles concernant l'interprétation de certaines règles, en serait une illustration), est un cas typique de langage **EXPTIME**. (De nouveau, il faut que le jeu s'arrête de façon garantie au bout d'un nombre fini de coups : on dit que le jeu est *déterminé*.) C'est ce que nous dit **EXPTIME** = **APSPACE** : le temps exponentiel, c'est un jeu (déterminé) que l'on joue sur un damier de taille polynomiale, sans limites de nombre de coups a priori.

4 La hiérarchie polynomiale

Une machine alternante *alterne* entre des suites de configurations existentielles et des suites de configurations universelles. Le nombre d'alternances entre ∃ et ∀, c'est-à-dire le nombre de coups à jouer entre ∃loïse et ∀bélard, est une ressource de calcul que l'on peut borner, au même titre que le temps ou l'espace.

On notera donc en général $\mathbf{ATIME}(f(n), g(n))$ la classe des langages décidables en temps $f(n)$ sur une machine alternante qui alterne au plus $g(n)$ fois entre ∃ et ∀, et de même $\mathbf{ASPACE}(f(n), g(n))$ la classe des langages décidables en espace $f(n)$ avec au plus $g(n)$ alternances.

Ces classes ont surtout leur intérêt lorsque $g(n)$ est une constante, auquel cas il sera nécessaire de préciser quel est le joueur qui commence. (Il y a des exceptions : notamment, le problème de la validité des formules de l'arithmétique dite de Presburger, c'est-à-dire avec juste l'addition comme opération mais pas la multiplication, est un problème complet pour la classe $\mathbf{ATIME}(2^{2^{O(n)}}, poly(n))$!)

Donnons-en une définition explicite d'abord, qui ne fait pas appel à la notion de machine alternante.

Définition 4.1 (Hiérarchie polynomiale) *On définit les classes Σ_n^p et Π_n^p, $n \in \mathbb{N}$, comme suit. D'abord, $\Sigma_0^p = \Pi_0^p = \mathbf{P}$. Ensuite, pour tout $n \in \mathbb{N}$:*

— Σ^p_{n+1} *est la classe des langages* L *de la forme* $\{x \mid \exists y \in A^* \text{ de taille } p(n) \cdot x\#y \in L'\}$, *où* n *dénote la taille de* x, *pour un certain polynôme* p, *un certain langage* L' *de* Π^p_n, *et où le séparateur* $\#$ *est un symbole hors de l'alphabet* A *du témoin* y ;
— Π^p_{n+1} *est la classe des langages* L *de la forme* $\{x \mid \forall y \in A^* \text{ de taille } p(n) \cdot x\#y \in L'\}$, *où* n *dénote la taille de* x, *pour un certain polynôme* p, *un certain langage* L' *de* Σ^p_n, *et où le séparateur* $\#$ *est un symbole hors de l'alphabet* A *du témoin* y.

La hiérarchie polynomiale **PH** *est l'union* $\bigcup_{n\in\mathbb{N}} \Sigma^p_n$.

On remarque par exemple que Σ^p_1 est la classe des langages L de la forme $\{x \mid \exists y \text{ de taille } p(n) \cdot x\#y \in L'\}$, où $L' \in \mathbf{P}$. Un tel langage est alors dans **NP** : deviner y en temps polynomial (ce qui est possible parce que y est de taille polynomiale), puis vérifier en temps polynomial que $x\#y$ est dans L'. Réciproquement, pour tout langage L de **NP**, on peut deviner la suite y de tous les choix non déterministes qui seront faits par la machine non déterministe $\mathcal{M}$ en temps polynomial $p(n)$ décidant L, puis simuler $\mathcal{M}$ en forçant les choix devinés par avance, ce qui ne prend qu'un temps polynomial. Une démonstration plus propre consiste à dire que $x \in L$ si et seulement s'il existe un mot y de taille $O(p^2(n))$ qui est exactement le codage d'une suite d'au plus $p(n)$ configurations de $\mathcal{M}$ commençant par $C_0(x)$ et se terminant sur une configuration acceptante. Le langage des telles suites est en temps polynomial (en fait même en espace logarithmique), donc L est dans Σ^p_1. Nous venons de montrer :

Proposition 4.2 $\mathbf{NP} = \Sigma^p_1$.

Il est facile de voir que :

Proposition 4.3 *Pour toute classe de langages* $\mathcal{C}$, *soit* $\mathbf{co}\mathcal{C}$ *la classe des complémentaires des langages de* $\mathcal{C}$. *Alors* $\Sigma^p_n = \mathbf{co}\Pi^p_n$, $\Pi^p_n = \mathbf{co}\Sigma^p_n$ *pour tout* $n \in \mathbb{N}$. *De plus,* $\Sigma^p_n, \Pi^p_n \subseteq \Sigma^p_{n+1} \cap \Pi^p_{n+1}$.

Démonstration. La première affirmation est une récurrence immédiate sur n, partant du fait que **P** est clos par complémentaire. La seconde est une conséquence directe de la définition. □

Proposition 4.4 *Le problème VAL suivant est* **coNP***-complet, c'est-à-dire* Π^p_1*-complet, pour les réductions logspace :*
ENTRÉE : une formule propositionnelle F.
QUESTION : F *est-elle valide, autrement dit est-il vrai que toute valuation* ρ, *associant chaque variable de* F *à une valeur de vérité, rende* F *vraie ?*

Démonstration. D'abord, $\mathbf{coNP} = \mathbf{co}\Sigma^p_1 = \Pi^p_1$. VAL est dans **coNP** : on énumère par choix $\forall$ les valeurs de toutes les variables, ce qui forme une valuation ρ, et on vérifie en temps polynomial que F est rendue vraie par ρ. Ensuite, pour tout langage L de **coNP**, par le théorème de Cook 2.14, on peut réduire la question $x \notin L$ à la vérité d'une formule de la forme $\exists x_1, \ldots, x_n \cdot F$, où F est une formule propositionnelle (en fait la conjonction des éléments d'un ensemble de clauses S), et $x_1, \ldots, x_n$ contient toutes ses variables. Ceci

revient à réduire la question $x \in L$ à la vérité de $\forall x_1, \ldots, x_n \cdot \neg F$, et $\neg F$ est alors l'instance de VAL cherchée. □

Il est clair, au vu de la preuve, qu'on peut même demander à ce que F soit d'une forme plus spéciale, typiquement une disjonction de conjonctions de littéraux.

Passons maintenant au rapport entre hiérarchie polynomiale et machines alternantes. Appelons, par commodité, *signature* d'une trace $C_0 \to_{\mathcal{M}} C_1 \to_{\mathcal{M}} C_2 \to_{\mathcal{M}} \ldots \to_{\mathcal{M}} C_k$ (où C_k est une configuration finale) la suite $Q_{C_0}Q_{C_1}Q_{C_2}\ldots Q_{C_{k-1}}$, où Q_{C_i} est $\exists$ si l'état de C_i est existentiel, $\forall$ sinon. Notons σ_n et π_n les langages de symboles définis comme suit :

— $\sigma_0 = \pi_0 = \epsilon$ (mot vide) ;
— $\sigma_{n+1} = \exists^* \pi_n$;
— $\pi_{n+1} = \forall^* \sigma_n$.

Autrement dit, σ_1 est le langage $\exists^*$, $\pi_1 = \forall^*$, $\sigma_2 = \exists^*\forall^*$, $\pi_2 = \forall^*\exists^*$, etc. On a alors :

Proposition 4.5 *Pour tout $n \geq 1$, Σ_n^p (resp., Π_n^p) est la classe des langages décidables en temps polynomial sur une machine alternante dont toutes les traces ont une signature dans σ_n^p (resp., π_n^p).*

Ce qui revient à dire que la machine n'alterne pas plus de $n-1$ fois entre $\exists$ et $\forall$, et commence par le quantificateur idoine.

Démonstration. Par récurrence sur n. D'abord, Σ_1^p est **NP**, et $\sigma_1^p = \exists^*$, autrement dit les machines dont toutes les traces ont une signature dans σ_1^p sont précisément les machines non déterministes. $\Pi_1^p = \mathbf{co}\Sigma_1^p$, et l'on peut clairement décider le complémentaire d'un langage L décidé par une machine alternante en faisant tourner une machine ayant les mêmes états et les mêmes transitions, mais où l'on a échangé états acceptants et rejetants, et échangé états universels et états existentiels. Les cas $n \geq 2$, de récurrence, sont immédiats. □

On en déduit immédiatement :

Proposition 4.6 $\mathbf{PH} \subseteq \mathbf{PSPACE}$.

On pense que la hiérarchie polynomiale ne s'écroule pas, autrement dit que toutes les classes Σ_n^p, Π_n^p, sont distinctes. Sous cette hypothèse, **PH** n'a pas de problème complet. En revanche, toutes les classes Σ_n^p, Π_n^p (ainsi que **PSPACE**) en ont. C'est le sujet des exercices qui suivent.

▷ **Exercice 4.1**

Supposons que L soit un langage **PH**-complet. Montrer que la hiérarchie polynomiale s'écroule.

▷ **Exercice 4.2**

Montrer que le problème suivant Σ_nQBF (resp., Π_nQBF), une restriction de QBF, est Σ_n^p-complet (resp., Π_n^p-complet) pour tout $n \geq 1$:
ENTRÉE : une formule booléenne quantifiée $F = Q_1 x_1 \cdot \ldots \cdot Q_n x_n \cdot S$, où S est un ensemble fini de clauses portant uniquement sur les variables $x_1, \ldots, x_n$, et où $Q_1 \ldots Q_n \in \sigma_n$ (resp., $\in \pi_n$).
QUESTION : F est-elle vraie ?

À quoi correspond ce problème lorsque $n = 1$?

On peut aussi définir Σ^p_{n+1} comme étant $\mathbf{NP}^{\Pi^p_n}$ ("**NP** oracle Π^p_n"), c'est-à-dire la classe des langages décidables en temps polynomial sur une machine non déterministe enrichie d'un oracle résolvant un problème Π^p_n-complet (Π_nQBF si $n \geq 1$, HORN-SAT si $n = 0$ par exemple), et de même Π^p_{n+1} comme étant $\mathbf{coNP}^{\Sigma^p_n}$ ("**coNP** oracle Σ^p_n"). On a en fait aussi $\Sigma^p_{n+1} = \mathbf{NP}^{\Sigma^p_n}$, $\Pi^p_{n+1} = \mathbf{coNP}^{\Pi^p_n}$. On remarque alors qu'il existe d'autres classes intéressantes, appelées Δ^p_n : $\Delta^p_0 = \mathbf{P}$, et $\Delta^p_{n+1} = \mathbf{P}^{\Sigma^p_n} = \mathbf{P}^{\Pi^p_n}$. Il est facile de voir que Σ^p_n et Π^p_n sont inclus dans Δ^p_{n+1}, qui est inclus à la fois dans Σ^p_{n+1} et dans Π^p_{n+1}. Mais nous n'examinerons pas ces classes (qui ont elles aussi des problèmes complets !)

Le panorama des classes que nous avons découvertes, pour résumer, se trouve représenté graphiquement en figure 3.

Références

[1] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[2] Larry Stockmeyer and Albert Meyer. Word problems requiring exponential time. In *Proceedings of the 5th ACM Symposium on the Theory of Computing*, pages 1–9. ACM Press, 1973.

[3] Ralf Treinen. Notes de cours. `http://www.pps.jussieu.fr/~treinen/teaching/stic/complexite/main.ps.gz`, April 2005.

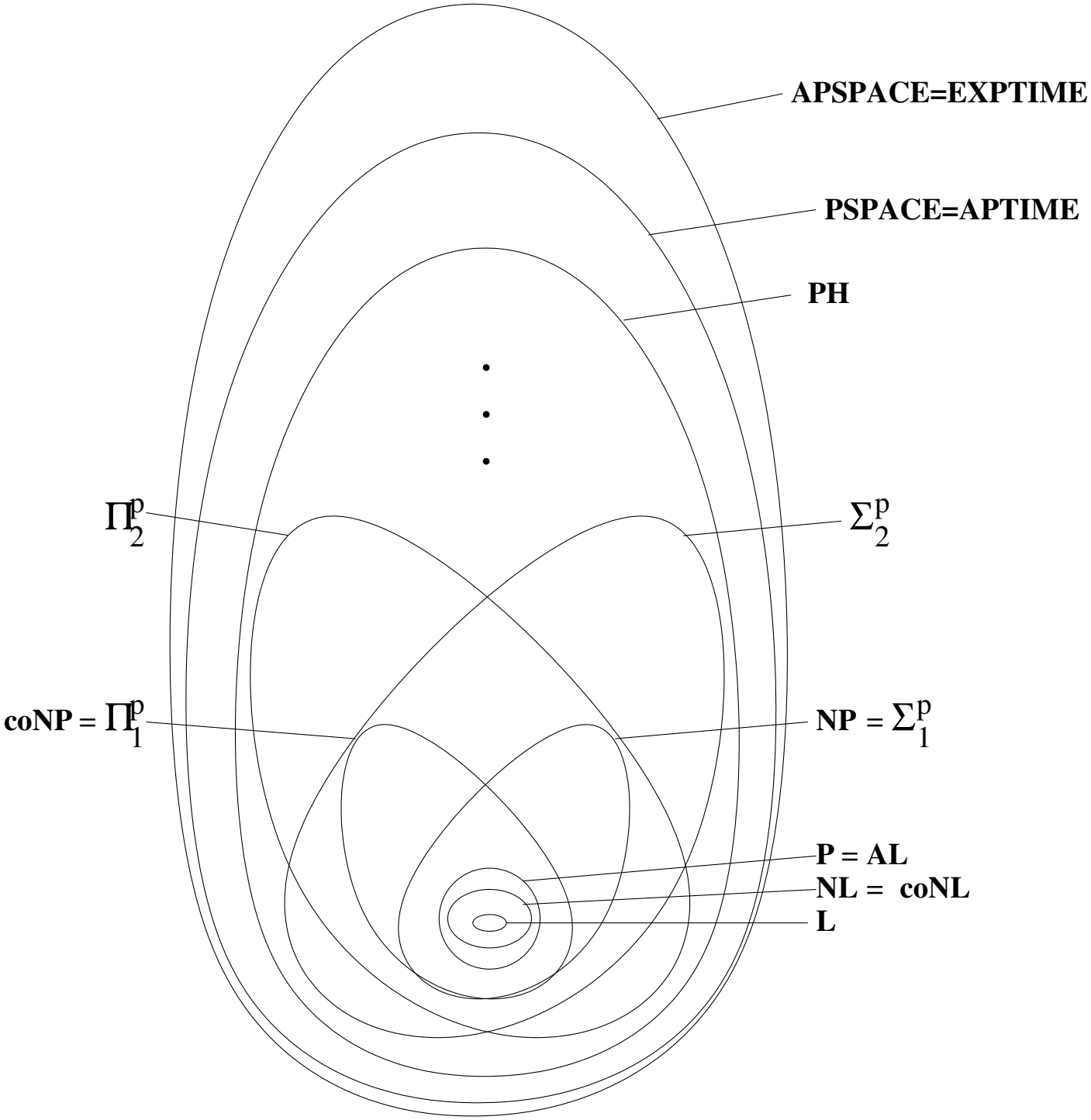

FIGURE 3 – Classes en espace, et en temps

▷ **1.1.** D'abord, la machine $\mathcal{M}$ doit calculer N en binaire. Or N est le nombre de • dans l'entrée. On fabrique donc un *compteur binaire*. On réserve une bande B, initialement vide. $\mathcal{M}$ parcourt son entrée de gauche à droite, et incrémente B chaque fois qu'elle lit un •. Pour incrémenter B, $\mathcal{M}$ parcourt B de gauche à droite. Tant que $\mathcal{M}$ lit un 1, elle le change en 0 et bouge à droite. Si $\mathcal{M}$ lit un 0, elle le change en 1 et revient au début de la bande B, l'incrémentation est terminée. Si $\mathcal{M}$ est à l'extrémité droite de B, elle étend la bande en y ajoutant un 1 et revient de nouveau au début de B, l'incrémentation est encore terminée.

Ceci fait, $\mathcal{M}$ dispose sur la bande B de l'écriture de N en binaire, et à l'envers. Elle recopie donc B de droite à gauche sur la bande de sortie, puis écrit #. Elle écrit ensuite N^2 zéros sur la bande de sortie, à l'aide de deux compteurs i et j en binaires comptant de 0 à $N-1$, puis remet la tête de la bande de sortie juste après le #. (On ne peut pas le faire en revenant en arrière jusqu'à rencontre #, puisque la bande de sortie est en écriture seule. Mais on peut recompter N^2 cases de droite à gauche, à l'aide des deux compteurs i et j.) $\mathcal{M}$ n'a maintenant plus qu'à parcourir les listes d'adjacence, en revenant au début de la bande d'entrée, qu'elle parcourt de gauche à droite, et en écrivant un bit 1 à la position $Ni+j$ après le # pour chaque j trouvé dans la liste L_i. Pour ceci, $\mathcal{M}$ maintient un compteur i sur une bande, initialement 0, et qui sera de nouveau incrémenté chaque fois que l'on lira un •. De la sorte, au moment où $\mathcal{M}$ lira L_i, la bande B contiendra i en binaire. $\mathcal{M}$ collecte tous les bits qu'elle voit sur une deuxième bande j, jusqu'à rencontrer un ";", et ce répétitivement. Pour chaque couple (i,j) ainsi rencontré, $\mathcal{M}$ déplace la tête de sortie de $Ni+j$ cases à droite, écrit un 1, et revient de $Ni+j$ cases à gauche. Pour compter jusqu'à $Ni+j$, on maintient deux autres compteurs i' et j' ; i' compte de 0 à $i-1$, et j' compte de 0 à $N-1$ (ceci compte jusqu'à Ni), puis on refait compter j' de 0 à $j-1$.

Si l'on réutilise la bande B pour l'un des compteurs, ceci utilise au total quatre compteurs de taille maximale $\log_2 N$. Or la taille de l'entrée n vaut au moins N, puisque l'entrée contient N caractères •. Ceci fonctionne donc en espace $O(\log n)$, et l'on conclut par le théorème de speedup en espace. □

▷ **1.2.** La machine $\mathcal{M}$ réserve deux compteurs i et j, et entre dans deux boules imbriquées : **pour** $i = 0$ **à** $N-1$: **pour** $j = 0$ **à** $N-1$. Ceci est possible car l'on sait incrémenter un compteur, et l'on peut comparer la valeur d'un compteur à N, qui est présent au début de la bande d'entrée, délimité par #. Pour chaque valeur de (i,j), $\mathcal{M}$ consulte le caractère à la position $Ni+j$ après le # sur la bande d'entrée, ce qui se fait avec deux autres compteurs i' et j' comme à l'exercice 2.1. Si ce caractère vaut 1, $\mathcal{M}$ écrit j en binaire sur la bande de sortie, suivi de ";". Chaque fois que $\mathcal{M}$ incrémente i, elle écrit de plus • sur la bande de sortie. Ceci utilise un espace $4\log_2 N$. Or la taille de l'entrée est $n = \log_2 N + 1 + N^2$, donc $N^2 = O(n)$, donc $N = O(\sqrt{n})$, donc $4\log_2 N = O(\log n)$. On conclut par le théorème de speedup en espace. □

▷ **1.3.** Clairement $\mathbf{SPACE}(f(n)) \subseteq \mathbf{SPACE}'(f(n))$. Réciproquement, soit $\mathcal{M}$ une machine comme dans l'énoncé. Si a est le nombre de lettres dans l'alphabet de bande de $\mathcal{M}$, $\mathcal{M}$ termine en temps $O(a^{f(n)})$ ou ne termine pas. En effet, il n'y a qu'un nombre $g(n) = O(a^{f(n)})$ de

configurations distinctes de $\mathcal{M}$, et si $\mathcal{M}$ passait deux fois par la même, elle bouclerait. Il suffit donc de modifier $\mathcal{M}$ en une machine $\mathcal{M}'$ avec sur une bande supplémentaire B un compteur, initialisé à 0, incrémenté de 1 à chaque instruction de $\mathcal{M}$. Pour simplifier, supposons que $g(n) = a^{f(n)}$ exactement. Le compteur est écrit en base a, sous forme $b_0 b_1 \dots b_{f(n)-1}$, les b_i étant des lettres parmi l'alphabet de bande (appelons-les 0, 1, ..., $a-1$). On peut initialiser la bande B en espace $f(n)$, parce que f est propre. Incrémenter un compteur consiste à parcourt B de gauche à droite, et à changer tous les b_i en 0, tant que b_i valait $a-1$. Si l'on arrive ainsi à l'extrémité droite de B, on rejette. Sinon, c'est qu'on est tombé sur le premier b_i ne valant pas $a-1$, et on le remplace par $b_i + 1$. Ceci utilise un espace $O(f(n))$, et l'on conclut par le théorème de speedup en espace. □

▷ **1.4.** Clairement, $\mathbf{SPACE}(f(n)) \subseteq \mathbf{SPACE}''(f(n))$. Réciproquement, soit $\mathcal{M}$ comme dans l'énoncé. On peut forcer $\mathcal{M}$ à n'utiliser qu'un espace $f(n)$ quelle que soit l'entrée x, autrement dit on peut fabriquer $\mathcal{M}'$ à partir de $\mathcal{M}$ qui n'utilise qu'un espace $f(n)$. Pour ceci, $\mathcal{M}'$ commence par calculer $f(n)$ blancs sur une bande B, en espace $O(f(n))$. Ceci est l'endroit où nous utilisons le fait que f est propre. En plus de ces bandes, $\mathcal{M}'$ dispose des bandes de $\mathcal{M}$. $\mathcal{M}'$ recopie ensuite B sur toutes les bandes de travail de $\mathcal{M}$, et ajoute à la fin de chacune un symbole spécial #, différent de tous ceux de l'alphabet de bande de $\mathcal{M}$. Ce symbole est utilisé comme marqueur pour vérifier que $\mathcal{M}'$ n'utilise pas plus d'un espace $f(n)$ sur chacune de ses bandes de travail. $\mathcal{M}'$ calcule ensuite comme $\mathcal{M}$, à ceci près que lorsqu'une tête d'une bande de travail lit le caractère #, $\mathcal{M}'$ rejette. Si $x \in L$, $\mathcal{M}$ accepte x en espace $f(n)$. Donc $\mathcal{M}'$ ne lira jamais #, et acceptera. Si $x \notin L$, $\mathcal{M}'$ ne peut pas accepter x (sinon $\mathcal{M}$ l'accepterait). De plus, l'espace utilisé par $\mathcal{M}'$ est un $O(f(n))$ (pour calculer la bande B) plus $kf(n)$, où k est le nombre de bandes de travail de $\mathcal{M}$. Au total, $\mathcal{M}'$ utilise un espace $O(f(n))$. Donc L est dans $\mathbf{SPACE}'(O(f(n)))$. Par l'exercice 2.3, L est donc dans $\mathbf{SPACE}(O(f(n))) = \mathbf{SPACE}(f(n))$. □

▷ **1.5.** Il est clairement dans **NP**, comme restriction de SAT. Réciproquement, on peut transformer toute instance de SAT en instance de 3-SAT en introduisant des variables supplémentaires. L'idée est que l'on ne change pas le statut de satisfiabilité d'un ensemble de clauses si l'on remplace une clause de la forme $L_1 \vee L_2 \vee C$ (C non vide) par les deux clauses $L_1 \vee L_2 \vee x$ et $\neg x \vee C$, où x est une variable fraîche. En effet, si l'on peut rendre vraie $L_1 \vee L_2 \vee C$, alors soit $L_1 \vee L_2$ est vraie et il suffit de choisir x fausse, soit C est vraie et l'on prend x vraie ; et réciproquement, si $L_1 \vee L_2 \vee x$ et $\neg x \vee C$ sont toutes les deux vraies, soit x est vrai et C l'est nécessairement, soit x est faux et $L_1 \vee L_2$ l'est nécessairement, donc $L_1 \vee L_2 \vee C$ est vraie dans les deux cas.

Reste à montrer que cette transformation peut s'opérer en espace logarithmique. Une fois ceci fait, on saura que SAT $\preceq_{\mathbf{L}}$ 3-SAT $\subseteq$ SAT. Pour ceci, on commence par calculer sur une bande B le numéro N de la plus grande variable présente dans l'entrée (par balayage de l'entrée, stockage de chaque variable dans une bande auxiliaire B', et comparaison de B avec B' ; la comparaison de nombres en binaire n'est rien d'autre que la comparaison dans l'ordre lexicographique.) On traite ensuite les clauses les unes après les autres. Pour chacune,

on stocke le premier littéral (s'il existe) sur une bande B_1, puis le second littéral (s'il existe) sur une bande B_2, puis le troisième sur une bande B_3. S'il n'y a plus de littéraux dans C, on imprime $B_1 \vee B_2 \vee B_3$. Sinon, on imprime $B_1 \vee B_2 \vee z$, où z est un nom de variable fraîche, on écrit $\neg z$ dans B_1, puis on récupère le littéral suivant et celui d'encore après dans B_2 et B_3, et l'on continue. Ceci nécessite trois bandes de taille logarithmique, plus une pour stocker z. La bande de z est initialisée à $N + 1$ (N étant sur la bande B), et incrémentée à chaque fois. (On peut pour ceci réutiliser la bande B.) Il est facile de voir qu'on crée une nouvelle variable z pour deux occurrences de littéraux de l'entrée au plus, donc que z ne dépasse pas $N + n/2 = O(n)$; donc z est aussi de taille logarithmique. □

▷ **1.6.** Considérons le graphe G dont les sommets sont toutes les variables de l'ensemble de 2-clauses S, et leurs négations. Notons $\neg L$ le littéral $\neg x$ si $L = x$, x si $L = \neg x$. Pour chaque clause à deux littéraux $L \vee L'$, on crée un arc de $\neg L$ vers L', et un de $\neg L'$ vers L. Chaque clause à un littéral L est vu comme une clause à deux littéraux $L \vee L$, donc comme l'arc de $\neg L$ vers L. S'il existe une clause vide, on peut directement conclure que S est insatisfiable ; dans ce cas, néanmoins, on créera deux sommets supplémentaires $*$ et $\neg*$, et l'on créera un arc de $*$ vers $\neg*$ et un de $\neg*$ vers $*$.

S'il existe un chemin d'une variable x vers $\neg x$ et aussi un de $\neg x$ vers x, alors S est insatisfiable. En effet, toutes les arêtes représentent des implications qui sont vraies dans tout modèle de S, donc tous les chemins aussi, par transitivité de l'implication. En conséquence, dans tout modèle de S, x aurait la même valeur de vérité que $\neg x$, ce qui est impossible. (Le cas particulier de la clause vide est trivial.)

Montrons la réciproque, en suivant l'indication. On démontre que G_i ne contient aucun chemin passant par une variable et sa négation, par récurrence sur i. C'est vrai pour $i = 0$ par hypothèse. Lorsque $G_i = G_{i-1}$, c'est évident. Sinon, G_i est G_{i-1} plus un arc de x_i vers $\neg x_i$, sachant qu'il n'y a pas de chemin de $\neg x_i$ vers x_i dans G_{i-1}. S'il y avait un circuit de x_j à x_j en passant par $\neg x_j$ dans G_i, il devrait passer par le nouvel arc. En prenant un circuit qui passe une fois par chaque arc (par exemple un de taille minimale), le chemin obtenu en enlevant le nouvel arc, on obtiendrait un chemin de $\neg x_i$ vers x_i dans G_{i-1}, ce qui est impossible.

On définit alors les valeurs de vérité des variables $x_1, \ldots, x_n$ comme dans l'indication. Au vu de la construction, il y a dans G_n un chemin de $\neg x_i$ vers x_i ou de x_i vers $\neg x_i$ pour chaque i, $1 \leq i \leq n$. Nous avons vu qu'on ne pouvait pas avoir les deux cas en même temps. De plus, il est clair que le second cas survient dès que x_i est mis à faux, donc que le premier cas survient si x_i est mis à vrai. On en déduit que, pour tout littéral L, L est mis à vrai si et seulement s'il existe un chemin de $\neg L$ vers L, et à faux si et seulement s'il existe un chemin de L vers $\neg L$. Pour chaque arc $L \to L'$ du graphe G_n, en particulier de G, si L valait vrai et L' valait faux, il y aurait donc un chemin π dans G_n de $\neg L$ vers L, un autre de L' vers $\neg L'$, donc par concaténation un chemin de $\neg L$ vers $\neg L'$ passant par L (ainsi que par L'). Mais, par construction, lorsque $L \to L'$ est un arc de G (pas de G_n), $\neg L' \to \neg L$ est aussi un arc de G, donc de G_n. En le concaténant avec π, on obtiendrait ainsi un circuit de $\neg L$ vers $\neg L$ passant par L, ce qui est impossible. Donc pour tout arc $L \to L'$ de G, l'implication $L \Rightarrow L'$

est satisfaite. Ceci revient à dire que toutes les clauses de S sont satisfaites.

Comme G est produit en espace logarithmique, l'insatisfiabilité de S se réduit à résoudre le problème suivant : dans un graphe formé sur des sommets de la forme x ou $\neg x$, y a-t-il un sommet x, un chemin de x vers $\neg x$ et un chemin de $\neg x$ vers x ? Ceci ne se réduit pas de façon immédiate à REACH, mais au moins on peut le résoudre en espace non déterministe logarithmique en devinant x (en espace logarithmique), puis en appelant $REACHp(x, \neg x, N)$ et $REACHp(\neg x, x, N)$, où N est le nombre de variables.

Réciproquement, pour montrer que 2-SAT est **NL**-complet, il suffit de réduire la REACH à l'insatisfiabilité d'un ensemble de 2-clauses par une réduction logspace. C'est facile : pour chaque sommet on crée une variable propositionnelle, pour chaque arête (u, v) on écrit la clause $\neg u \vee v$ $(u \Rightarrow v)$, puis on écrit les clauses s et $\neg t$. Soit S l'ensemble des clauses ainsi obtenues. Si t est accessible depuis s, alors le chemin $s \to^* t$, vu comme une implication $s \Rightarrow t$, est une conséquence des clauses que l'on a écrites ; avec s, on en déduit t, ce qui contredit $\neg t$. Donc S est insatisfiable. Réciproquement, si t n'est pas accessible depuis s, mettons toutes les variables accessibles depuis s à vrai, et les autres à faux : les clauses s et $\neg t$ sont ainsi vérifiées, et les autres clauses $u \Rightarrow v$ sont vraies aussi, car si u est accessible depuis s, alors v aussi. Ceci fournit donc un modèle de S. □

▷ **1.7.** 2-SAT est dans **coNL**, donc dans **NL** par le corollaire 2.17. Pour tout langage L de **NL**, le complémentaire $\overline{L}$ de L est dans **NL**, toujours par le corollaire 2.17. Donc $\overline{L}$ se réduit en espace logarithmique à la négation de 2-SAT, puisque ce dernier est **NL**-complet. Mais cette réduction est alors automatiquement une réduction en espace logarithmique de L à 2-SAT. □

▷ **2.1.** Si π est une dérivation de $\vdash C$: OK, une récurrence immédiate sur la taille de π montre que C mène à l'acceptation. Réciproquement, l'ensemble des configurations C telles que $\vdash C$: OK soit dérivable est de façon évidente un ensemble S vérifiant les conditions de la définition 3.1 : toute configuration acceptante est dans S, si $C \to_{\mathcal{M}} C'$ et C' est dans S alors C est dans S, et ainsi de suite. Mais l'ensemble S_0 des configurations menant à l'acceptation est le plus petit vérifiant ces conditions. Donc $S_0 \subseteq S$. Ceci implique que toute configuration qui mène à l'acceptation est dans S, donc par définition il existe une dérivation de $\vdash C$: OK. □

▷ **2.2.** C'est réellement la même démonstration qu'au théorème 2.2 ! □

▷ **2.3.** Testons si w est un palindrome en espace logarithmique. On initialise un compteur i écrit en binaire, qui ira de 0 à la longueur de w (divisée par 2). On initialize un autre compteur j à la longueur n de w : ceci se fait en parcourant w de gauche à droite, et en incrémentant j au fur et à mesure en partant de 0. En maintenant l'invariant que $i + j = n$ et $i < j$, pour i croissant à partir de 0, on vérifie que le caractère numéro i de w égale le caractère numéro j. C'est le cas pour tout i si et seulement w est un palindrome. De plus, les deux compteurs i et j n'utilisent qu'un espace logarithmique.

En revanche, une machine, alternante ou non, en temps logarithmique en n n'aura le temps de lire que les $\log n$ premiers caractères de w, et en particulier sera obligé de donner les mêmes réponses sur wa et wb pour w assez long. Donc le test de palindromes n'est pas dans $\mathbf{ATIME}(\log n)$. □

▷ **2.4.** Soit $f(n) = 2^n$. C'est une fonction propre, et $f(n) \geq n$ pour tout $n \in \mathbb{N}$. Pour tout polynôme p, en utilisant le théorème 3.9, $\mathbf{SPACE}(2^{p(n)}) \subseteq \mathbf{SPACE}(poly(2^{p(n)})) = \mathbf{ATIME}(poly(2^{p(n)})) \subseteq \mathbf{AEXPTIME}$. Comme p est arbitraire, $\mathbf{EXPSPACE} \subseteq \mathbf{AEXPTIME}$. Réciproquement, pour tout polynôme p, $\mathbf{ATIME}(2^{p(n)}) \subseteq \mathbf{ATIME}(poly(2^{p(n)})) = \mathbf{SPACE}(poly(2^{p(n)})) \subseteq \mathbf{EXPSPACE}$. Donc $\mathbf{AEXPTIME} \subseteq \mathbf{EXPSPACE}$. □