Université de Montréal

**Le produit direct de fonctions et les programmes de branchement avec oracle**

par

Martin Lavoie

Département d'informatique et de recherche opérationnelle

Faculté des arts et des sciences

Mémoire présenté à la Faculté des études supérieures
en vue de l'obtention du grade de Maître ès sciences (M.Sc.)
en informatique

Avril, 2018

© Martin Lavoie, 2018.

# RÉSUMÉ

Ce mémoire explore divers résultats de la littérature à propos du produit direct de fonctions. Plus précisément, on retrouve la production de masse, le multi-calcul et l'espace catalytique. Le lien entre ces trois variantes du produit direct de fonctions est établi explicitement pour la première fois. La production de masse permet l'économie de ressources lors du calcul d'une fonction à plusieurs reprises sur des entrées différentes. On reprend en détails une borne de la littérature sur la taille des circuits qui calculent une fonction un nombre de fois qui excède tout polynôme. Le multi-calcul permet l'économie de ressources lors du calcul d'une multitude de fonctions sur la même entrée. L'espace catalytique est une combinaison des restrictions de la production de masse et du multi-calcul. Il s'agit du calcul d'une fonction une multitude de fois sur la même entrée. L'espace catalytique est étudié en utilisant des machines de Turing dont une partie de la mémoire doit être rétablie à son contenu initial. On reprend en détails un résultat de la littérature montrant l'inclusion de la classe $\mathsf{TC}^1$ dans l'espace catalytique logarithmique. Dans la deuxième partie de ce mémoire, notre contribution est une nouvelle forme de programme de branchement. À partir de celle-ci, on présente plusieurs langages paramétriques. Une analyse est faite pour montrer la $\mathsf{coNP}$-complétude de l'un de ces langages.



**Mots clés : Théorie de la complexité, production de masse, programme de branchement, espace catalytique.**

**ABSTRACT**

This dissertation explores various results from the literature about the direct product of functions. We discuss mass production, multi-computation and catalytic space. We explicitely show for the first time the link these three variants of the direct product of functions. Mass production allows economy of resources during the computation of a function on multiple inputs. We present a bound on the size of circuits which compute a function a super-polynomial number of times. With multi-computation, we can represent sharing of resources when computing multiple functions over the same input. The last one, catalytic space, is the intersection of mass production and multi-computation. A computation in catalytic space is computing a function multiple times over the same input. We study the catalytic space with Turing machines whose part of their memory needs to be restore to its initial content. We present in detail the result from the literature which shows the inclusion of $\mathsf{TC}^1$ in the logarithmic catalytic space. In the second section, we contribute with a new family of branching programs. We then derive parametric languages. We prove that one of them is $\mathsf{coNP}$-complete.

**Keywords: Complexity theory, mass production, branching program, catalytic space.**

# TABLE DES MATIÈRES

**RÉSUMÉ** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **i**

**ABSTRACT** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **ii**

**TABLE DES MATIÈRES** . . . . . . . . . . . . . . . . . . . . . . . . . . . . **iii**

**LISTE DES TABLEAUX** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **v**

**LISTE DES FIGURES** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **vi**

**REMERCIEMENTS** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **vii**

**INTRODUCTION** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **1**

**CHAPITRE 1 : NOTATIONS ET DÉFINITIONS** . . . . . . . . . . . . . . . . **3**

1.1 Quelques notations . . . . . . . . . . . . . . . . . . . . . . . . . . 3

1.2 Machine de Turing . . . . . . . . . . . . . . . . . . . . . . . . . . 3

1.3 Programme de branchement . . . . . . . . . . . . . . . . . . . . . 5

1.4 Circuit . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

1.5 Langage, réduction et complétude . . . . . . . . . . . . . . . . . . 8

1.6 Classes de complexité et problèmes complets . . . . . . . . . . . . 9

**CHAPITRE 2 : PRODUIT DIRECT DE FONCTIONS ET SPÉCIALISATIONS 12**

2.1 Production de masse . . . . . . . . . . . . . . . . . . . . . . . . . 14

2.1.1 Théorème d'Uhlig . . . . . . . . . . . . . . . . . . . . . . . 14

2.1.2 Théorème de Paul . . . . . . . . . . . . . . . . . . . . . . . 25

2.2 Multi-calcul . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25

2.2.1 Programme de branchement multi-calcul . . . . . . . . . . . 26

2.2.2 Multi-calcul avec des fonctions choisies aléatoirement . . . . . . . 26

**CHAPITRE 3 : ESPACE CATALYTIQUE** . . . . . . . . . . . . . . . . . . . . . . **28**

3.1 Calcul transparent . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28

3.2 La machine catalytique . . . . . . . . . . . . . . . . . . . . . . . . . . 30

3.3 Propriétés de CL . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31

3.3.1 Simulation d'une machine de Turing catalytique . . . . . . . . . . . 31

3.3.2 Composition de fonctions dans CSPACE$(f(n))$ . . . . . . . . . . . 32

3.4 $\mathsf{TC}^1$ dans CL . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33

3.4.1 Construction des programmes transparents . . . . . . . . . . . . . 34

3.4.2 Simulation de programmes transparents . . . . . . . . . . . . . . . 41

3.5 Potechin . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44

**CHAPITRE 4 : PROGRAMME DE BRANCHEMENT AVEC MODÈLE** . . . **48**

4.1 Définition des programmes de branchement avec modèles . . . . . . . . . . 48

4.2 Langages et outils basés sur les empreintes . . . . . . . . . . . . . . . . . 50

4.3 Définition des modèles MIN, MAX et MINMAX . . . . . . . . . . . . . . . 52

4.4 La complexité de MINMAX-GEN-négatif . . . . . . . . . . . . . . . . . . 54

4.5 Complexité des autres classes à partir de MINMAX et GEN . . . . . . . . . 61

**CONCLUSION** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **63**

**BIBLIOGRAPHIE** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **64**

## LISTE DES TABLEAUX


2.I Produit direct de fonctions et ces spécialisations. . . . . . . . . . . . 13

3.I Extrait du calcul de $T_{5,2}$. . . . . . . . . . . . . . . . . . . . . . . . . 39

3.II Valeurs des différents éléments du programme selon la valeur de $g$ et du noeud de départ. . . . . . . . . . . . . . . . . . . . . . . . . . . 47

4.I Les règles de la porte $\alpha_i = \wedge(\alpha_j, \alpha_k)$. . . . . . . . . . . . . . . . . . . 56

4.II Initialisation pour les variables d'entrées $i$ . . . . . . . . . . . . . . . 57

4.III Génération de l'ensemble $N$ . . . . . . . . . . . . . . . . . . . . . . . 57

## LISTE DES FIGURES


1.1 Configuration d'une machine de Turing . . . . . . . . . . . . . . . . 5

1.2 Programme de branchement qui calcule la fonction de parité sur 4 bits 7

1.3 Un circuit qui calcule la disjonction exclusive de deux bits. Il est de taille 7 et profondeur 4 . . . . . . . . . . . . . . . . . . . . . . . . . 9

2.1 Circuit qui utilise une décomposition d'Ashenhurst. . . . . . . . . . . 13

2.2 Représentation complète du circuit $A_{f,k,1}$ . . . . . . . . . . . . . . . 16

2.3 Extrait du circuit $T_{n,k}$ pour calculer le $p^{\text{e}}$ bit pour le circuit $g_l$ où $p \in \{1,2,\ldots,n-k\}$ et $g_l \in G$. . . . . . . . . . . . . . . . . . . . . . . . 18

2.4 Représentation complète du circuit $A_{f,k,2}$ . . . . . . . . . . . . . . . 20

3.1 Configuration initiale possible des rubans d'une machine de Turing catalytique. Le ruban de travail est souvent plus court que le ruban catalytique. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31

3.2 Algorithme pour le calcul de $f \circ g$ par une machine de Turing. . . . . 33

3.3 Programme complet du lemme 5. . . . . . . . . . . . . . . . . . . . . 38

3.4 Programme de branchement $P_2$ de la preuve de Potechin . . . . . . . 45

## REMERCIEMENTS

Je voudrais commencer par remercier mon directeur, Pierre, sans qui tout ceci n'aurait pas été possible. Merci pour ta patience, ta disponibilité quand j'en avais besoin et ta bonne humeur perpétuelle.

Merci à mes parents, Guylaine et Camil, qui m'ont poussé à toujours me dépasser.

Merci à ma conjointe Mélodie, qui a su écouter toutes mes idées farfelues au quotidien et rendre chaque jour un petit peu mieux que le précédent. Ton soutien a été inestimable pour me permettre de traverser ce long chemin qu'à été ma maîtrise.

Finalement, je remercie le Fonds de recherche du Québec – Nature et technologies (FR-QNT) qui m'a permis d'oublier tout souci financier pendant l'obtention de mon diplôme.

**INTRODUCTION**

Ce mémoire s'inscrit dans une longue tradition de recherche pour comparer $P$ versus $L$. Cette question définie par Cook en 1971 [Coo71] ne possède malheureusement toujours pas de réponse. Il s'agit de savoir si tous les problèmes solubles en temps polynomial le sont en utilisant seulement un espace logarithmique.

On considère une nouvelle classe de complexité définie en 2014 par Buhrman et coll. [BCK$^+$14] nommée espace catalytique logarithmique ($\mathsf{CL}$). Cette classe que l'on croit plus grande que $L$, pourrait servir de levier pour effectuer la différentiation de $P$ et $L$. Malgré les grandes aspirations de $\mathsf{CL}$ au niveau théorique, c'est plutôt pour les conséquences concrètes des résultats $\mathsf{CL}$ que je m'y suis intéressé. Habituellement, pour utiliser de la mémoire dans un ordinateur, il faut que celle-ci soit inoccupée et ensuite réservée au programme pour la durée de son exécution. Ce sont ces principes fondamentaux que l'espace catalytique remet en question. L'étude de l'espace catalytique représente l'étude de l'utilisation d'une mémoire déjà remplie pour augmenter les possibilités de calcul. Cela pourrait être utile entre autres dans les domaines suivants : l'informatique embarquée, l'informatique des objets, les ramasse-miettes. En effet, il s'agit de systèmes dont la mémoire est limitée et la possibilité d'en réutiliser une partie lors de temps mort pourrait être bienvenue. On étudie également l'espace catalytique sous la forme de programme de branchement avec le théorème de Potechin [Pot16]. Une analyse plus poussée a permis, dans ce mémoire, d'améliorer la borne originale de ce théorème de $32n$ à $6n$. On peut voir l'espace catalytique comme une variante restreinte du produit direct de fonctions.

On étudie deux autres variantes où les contraintes de l'espace catalytique sont relaxées. La première est la production de masse. Elle permet d'économiser du temps de calcul lorsque l'on calcule une fonction sur plusieurs entrées. Il s'agit d'un problème d'actualité dans cette ère des mégadonnées. On présente un résultat d'Uhlig [Uhl92], celui-ci indique que pour n'importe quelle fonction, il est possible de la calculer un nombre de fois super-polynomial tout en gardant les mêmes bornes que Lupanov [Lup62]. Ces bornes limitent la taille et la

profondeur d'un circuit qui calcule une fonction une seule fois. La seconde est le multi-calcul où l'on utilise la même entrée pour plusieurs fonctions. Le lien entre le produit direct de fonctions, le calcul catalytique, la production de masse et le multi-calcul n'avait jamais été explicitement mis en évidence.

Dans le dernier chapitre, on propose une nouvelle sorte de programme de branchement, les programmes de branchement avec modèles. Plusieurs outils sont définis pour permettre l'étude de ces programmes de branchement. Entre autres, on définit des nouveaux langages pour étudier la puissance des modèles. Une contribution de ce mémoire est de démontrer la NP-complétude de l'un de ces nouveaux langages.

# CHAPITRE 1

# NOTATIONS ET DÉFINITIONS

Dans ce mémoire, plusieurs outils seront utilisés pour l'analyse de la complexité dont :

- la machine de Turing ;
- le circuit ;
- le programme de branchement.

Ce chapitre est un rappel des définitions qui sont utilisées et de la notation associée à chacun de ces outils.

## 1.1 Quelques notations

Mais avant cela, on définit les ensembles suivants :

- $\mathbb{B}$ est l'ensemble binaire $\{0,1\}$ ;
- $\mathbb{N}$ est l'ensemble des naturels $\{0,1,2,\dots\}$ ;
- $\mathscr{B}_n$ est l'ensemble des fonctions booléennes $f : \mathbb{B}^n \to \mathbb{B}$ de $n$ bits.

On note par $\wedge$ la conjonction, par $\vee$ la disjonction, par $\oplus$ la disjonction exclusive et par $\neg$ la négation.

## 1.2 Machine de Turing

Une machine de Turing dans sa forme la plus simple est un automate avec un ruban de mémoire infini. L'automate lit une case pour décider l'état vers lequel il doit aller et son déplacement sur le ruban. Depuis son introduction par Turing [Tur36], de nombreuses variantes et restrictions ont été utilisées en théorie de la complexité. Ces variantes servent à faciliter l'étude de plusieurs problèmes. Voici une définition qui contient un seul ruban.

**Définition 1** ([HMRU00]). *Une machine de Turing déterministe est un octuplet* $(Q, \Gamma, \Sigma, \delta, q_0, B, q_a, q_r)$ *où :*

*$Q$ est l'ensemble fini des états de la machine ;*

*$\Gamma$ est l'ensemble fini des symboles qui peuvent être écrits sur le ruban ;*

*$\Sigma$ est l'ensemble fini non vide des symboles d'entrées ($\Sigma \subseteq \Gamma$) ;*

*$\delta$ est la fonction de transition de la machine ;*

$$\delta : Q \times \Gamma \mapsto Q \times \Gamma \times \{G, D\}$$

*$q_0 \in Q$ est l'état de départ de la machine ;*

*$B \in \Gamma$ est le symbole blanc de la machine ;*

*$q_a \in Q$ et $q_r \in Q$ sont l'état d'acceptation et de rejet respectivement.*

Pour évaluer le résultat d'une machine de Turing $M$ sur un mot $w \in \Sigma^*$, on écrit le mot $w$ sur le ruban de travail de la machine et positionne la tête de lecture à la première lettre de $w$ puis on commence l'exécution de la machine. Le reste du ruban contient le symbole blanc($B$). Il y a trois résultats possibles à l'exécution d'une machine de Turing :

- la machine accepte le mot $w$ en arrêtant sur l'état $q_a$ ;
- la machine refuse le mot $w$ en arrêtant sur l'état $q_r$ ;
- la machine ne s'arrête jamais.

À chaque étape de son calcul, la machine regarde le contenu de la case sous sa tête de lecture $x \in \Gamma$ et son état $q \in Q$ dans son automate. Elle applique sa fonction de transition $\delta(q,x) = (q' \in Q, x' \in \Gamma, m \in \{G, D\})$. L'état $q'$ est le nouvel état de la machine. Le contenu de la case sous la tête est remplacé par $x'$. La machine déplace d'une case la tête de lecture vers la gauche si $m = G$ ou vers la droite sinon. Elle répète ces étapes jusqu'à arriver à un de ces deux états terminaux ($q_a$, $q_r$) ou bien continue infiniment. Le langage d'une machine de Turing $M$ est l'ensemble des mots $L(M) \subseteq \Sigma^*$ tel que la machine de Turing $M$ s'arrête sur

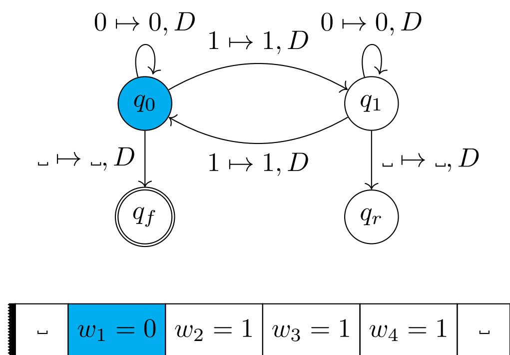


Figure 1.1 : Configuration d'une machine de Turing

l'état $q_a$ lorsqu'elle commence sur ces mots. Le temps d'une machine $M$ sur un mot $x$ est le nombre d'étapes requis à la machine $M$ sur entrée $x$. L'espace de $M$ sur $x$ est le nombre de cases visitées par la tête de lecture sur le ruban de travail.

On introduit les machines de Turing pour l'espace [Per14] pour étudier les langages qui nécessitent une petite quantité d'espace supplémentaire comparativement à la taille de l'entrée. Ces machines possèdent 2 rubans. Le premier ruban ne permet pas l'écriture et contient l'entrée de la machine. Le deuxième ruban permet l'écriture et la lecture et est vide au début du calcul. On regarde seulement le deuxième ruban pour déterminer la quantité de mémoire utilisée par la machine de Turing.

Dans une machine de Turing non déterministe, on remplace la fonction de transition par une relation

$$\delta : (Q \times \Gamma) \times (Q \times \Gamma \times \{G, D\}).$$

Il peut donc y avoir plusieurs possibilités lors d'une transition. Une machine de Turing non déterministe accepte s'il existe une suite de transitions qui se termine à l'état d'acceptation.

## 1.3 Programme de branchement

Les programmes de branchement sont la deuxième façon d'étudier la complexité des langages que l'on va utiliser.

**Définition 2** ([Weg87]). *Un programme de branchement est un graphe orienté acyclique fini. Tous les noeuds ont un degré sortant 0 ou 2. Les noeuds ne possédant aucun arc sortant sont étiquetés par $\top$ ou $\bot$. Les autres noeuds sont étiquetés par un nombre naturel. Pour chaque noeud de degré sortant 2, on étiquette un de ces arcs avec 0 et l'autre avec 1. Il existe exactement un noeud avec un degré entrant 0. On appelle ce noeud la source.*

L'abréviation BP signifie programme de branchement. Pour savoir si un programme de branchement reconnaît un mot $w = w_1 w_2 \dots w_n$ où $w_i \in \mathbb{B}$, on effectue la procédure suivante :

1. Se placer à la source du programme de branchement;
2. Tant que l'étiquette $x$ du noeud courant n'est pas $\top$ ou $\bot$, suivre l'arc étiqueté par la valeur de $w_x$.

Si le noeud final est $\top$ (*resp.* $\bot$), on dit que le programme reconnaît (*resp.* rejette) le mot $w$. De façon naturelle, on dit qu'un programme de branchement calcule une fonction $f : \mathbb{B}^k \to \mathbb{B}$ telle que $f(w) = 1$ si et seulement si $w$ est reconnu par ce programme de branchement. La taille d'un programme de branchement est le nombre de noeuds intérieur qu'il contient. On note la taille du programme $A$ $\mathsf{TAILLE}_{pb}(A)$. La profondeur d'un tel programme est la longueur du plus long chemin. On note $\mathsf{PROF}_{pb}(A)$.

### 1.4 Circuit

On défini les circuits booléens avec des noeuds $\wedge$, $\vee$ et $\neg$.

**Définition 3** ([Per14]). *Un circuit booléen est un graphe orienté acyclique avec les propriétés suivantes :*

- *les noeuds de degré entrant nul sont les entrées du circuit et chacun est étiqueté par une variable $x_i$ ou une constante ;*
- *les noeuds de degré entrant 1 sont étiquetés par $\neg$;*

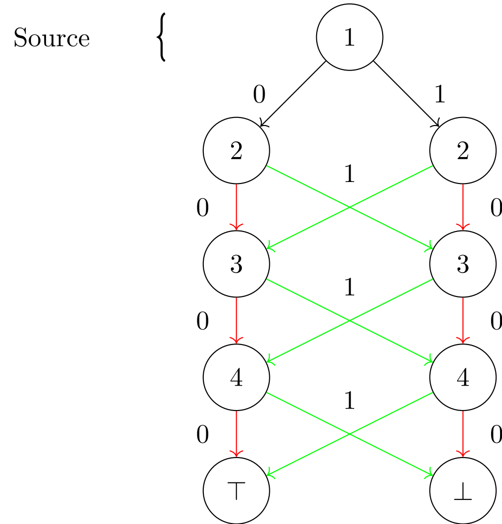


Figure 1.2 : Programme de branchement qui calcule la fonction de parité sur 4 bits

- *les autres noeuds (de degré entrant 2) sont étiquetés par* $\wedge$ *ou* $\vee$ *;*
- *les noeuds de degré sortant nul sont les sorties du circuit.*

Sauf indication contraire, les circuits utilisés ont une seule sortie et un degré entrant maximal de 2. La taille d'un circuit est le nombre de noeuds dans le graphe. On note $\mathsf{TAILLE}_{circ}(C)$. La profondeur d'un circuit est la longueur d'un plus long chemin. On note $\mathsf{PROF}_{circ}(C)$. De plus, on peut borner la taille et la profondeur du circuit minimal qui calcule une fonction de la façon suivante [Lup62] :

$$\max_{f \in \mathscr{B}_n} \mathsf{TAILLE}_{circ}(f) \sim \frac{2^n}{n}, \tag{1.1}$$

$$\max_{f \in \mathscr{B}_n} \mathsf{PROF}_{circ}(f) \sim n \tag{1.2}$$

où $a(n) \sim b(n)$ signifie qu'il existe une fonction $\alpha(n)$ qui possède les propriétés

$$\lim_{n \to \infty} \alpha(n) = 0 \tag{1.3}$$

$$a(n) = b(n)(1 + \alpha(n)).$$

On peut voir que $a(n) \sim b(n)$ implique $a(n) \in \Theta(b(n))$.

### 1.5 Langage, réduction et complétude

**Définition 4.** *Un langage est un ensemble de mots sur un certain alphabet fini.*

La plupart des langages utilisés dans ce mémoire seront sur l'alphabet binaire. Une réduction permet de mettre en relation 2 langages.

**Définition 5** ([Per14]). *Une réduction multivoque d'un langage $B$ (sur l'alphabet $\Sigma_B$) à un langage $A$ (sur l'alphabet $\Sigma_A$) est une fonction $f : \Sigma_B \to \Sigma_A$ calculable par une machine de Turing $M$ telle que :*

$$\forall x \in \Sigma_b^*, x \in B \iff f(x) \in A.$$

*On note $B \leq_m A$.*

Le plus souvent, on restreint la puissance de la machine de Turing. Si elle fonctionne en temps polynomial, on note $B \leq_m^{\mathsf{P}} A$ et si la machine fonctionne en espace logarithmique, on note $B \leq_m^{\mathsf{L}} A$.

**Définition 6.** *Une classe de complexité est un ensemble de langages.*

Habituellement, une classe de complexité est définie à l'aide d'un modèle de calcul et d'une borne sur les ressources de ce modèle. Elle contiendra tous les langages qui sont reconnus par ce modèle en respectant cette borne.

**Définition 7** ([Per14]). *Soit $\leq$ une réduction, $\mathscr{L}$ un langage et $\mathscr{C}$ une classe de complexité. Le langage $\mathscr{L}$ est $\mathscr{C}$-ardu pour $\leq$ si pour tout langage $A \in \mathscr{C}$, $A \leq \mathscr{L}$. Le langage $\mathscr{L}$ est $\mathscr{C}$-complet pour $\leq$ si $\mathscr{L}$ est $\mathscr{C}$-ardu pour $\leq$ et $\mathscr{L} \in \mathscr{C}$.*

Les réductions seront omises dans les prochains chapitres pour alléger le texte.

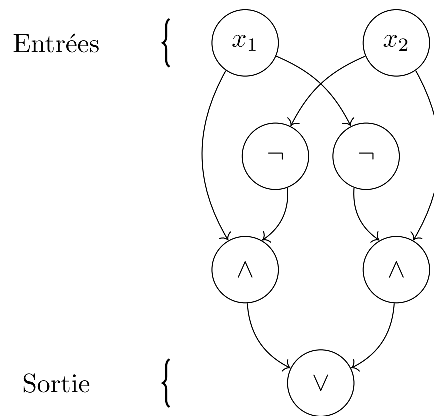


Figure 1.3 : Un circuit qui calcule la disjonction exclusive de deux bits. Il est de taille 7 et profondeur 4

### 1.6 Classes de complexité et problèmes complets

Nous les survolons ici en incluant des problèmes complets associés.

La classe $\mathsf{P}$ est l'ensemble des langages qui peuvent être décidés en temps polynomial par une machine de Turing. Le langage $\mathsf{GEN}$ (pour générateur) sera utilisé comme langage $\mathsf{P}$-complet selon $\leq_m^{\mathsf{L}}$.

**Langage:** $\mathsf{GEN}$

**Données:** Un tableau de $n \times n$ cases contenant chacune un nombre de 1 à $n$.

**Décider:** Considérons le tableau comme la table de multiplication de l'opération binaire $f(x,y)$ sur l'ensemble $\{1,2,\ldots,n\}$, est-ce que l'élément $n$ appartient à la clotûre de l'élément 1 sous l'opération binaire $f$.

La classe $\mathsf{NP}$ est l'ensemble des langages qui sont décidés par une machine de Turing non déterministe en temps polynomial. Le langage $\mathsf{SAT}$ (pour *satisfiability*) est $\mathsf{NP}$-complet selon $\leq_m^{\mathsf{P}}$. Il cherche à vérifier si une formule peut être vraie.

**Langage:** $\mathsf{SAT}$

**Données:** Une formule booléenne $\phi$ sur les variables $x_1$, $x_2$, …, $x_n$ avec les opérateurs $\neg$, $\wedge$

et $\vee$.

**Décider:** Existe-t-il une affectation $a_1$, $a_2$, ..., $a_n$ telle que la formule $\phi(a_1, a_2, \ldots, a_n)$ est satisfaite ?

La classe $\mathsf{coNP}$ est l'ensemble des langages dont le complément est dans $\mathsf{NP}$. Le langage $\mathsf{TAUT}$ (pour tautologie) est $\mathsf{coNP}$-complet selon $\leq_m^{\mathsf{P}}$. Une formule booléenne appartient à $\mathsf{TAUT}$ si et seulement si toutes les affectations rendent la formule vraie.

**Langage:** $\mathsf{TAUT}$

**Données:** Une formule booléenne $\phi$ sur les variables $x_1$, $x_2$, ..., $x_n$ avec les opérateurs $\neg$, $\wedge$ et $\vee$.

**Décider:** Est-ce que toutes les affectations $a_1$, $a_2$, ..., $a_n$ satisfont la formule $\phi(a_1, a_2, \ldots, a_n)$ ?

Les langages $\mathsf{SAT}$ et $\mathsf{TAUT}$ mettent bien en évidence la différence présumée entre $\mathsf{NP}$ et $\mathsf{coNP}$. Pour le premier, il est facile de vérifier qu'un mot appartient au langage, alors que pour le second, il est facile de vérifier qu'un mot n'appartient pas au langage.

La classe $\mathsf{L}$ est la classe des langages qui peuvent être décidés par une machine qui utilise un espace logarithmique. Le langage $\mathsf{GEN}_1$ est $\mathsf{L}$-complet selon $\leq_m^{\mathsf{AC}^0}$ ($\mathsf{AC}^0$ est une petite classe définie à partir de circuits [Per14]).

**Langage:** $\mathsf{GEN}_1$

**Données:** Un tableau de $1 \times n$ cases contenant chacune un nombre de 1 à $n$.

**Décider:** Considérons le tableau comme la première ligne de la table de multiplication d'une opération binaire $f(x, y)$ sur l'ensemble $\{1, 2, \ldots, n\}$ et dont le reste des valeurs est 1, est-ce que l'élément $n$ appartient à la clotûre de l'élément 1 sous l'opération $f$.

Ce langage démontre bien la limite de nos connaissances de $\mathsf{L}$ et de $\mathsf{P}$. Le langage $\mathsf{GEN}_1$

fait bien piètre figure face à $\mathsf{GEN}$ puisqu'il s'agit d'une version bien simplifiée de ce problème. Or, nous sommes toujours incapables de montrer qu'il n'existe pas de réduction $\leq_m^{\mathsf{L}}$ de $\mathsf{GEN}$ vers $\mathsf{GEN}_1$.

# CHAPITRE 2

## PRODUIT DIRECT DE FONCTIONS ET SPÉCIALISATIONS

Dans ce chapitre, notre contribution est la mise en évidence de la relation entre la production de masse, le multi-calcul et l'espace catalytique. On présente ensuite des résultats de la littérature sur la production de masse et le multi-calcul.

Il est souvent de mise de décomposer un problème à résoudre en plusieurs problèmes plus petits. En 1959, Ashenhurst [Ash59] émet la conjecture que sa décomposition produit des circuits de taille minimale.

**Définition 8** ([Ash59])**.** *Une décomposition d'Ashenhurst d'une fonction $f \in \mathscr{B}_n$ est donnée par $(g, h, \pi)$ tel que*

$$f(x_1, \ldots, x_n) = g(x_{\pi(1)}, \ldots, x_{\pi(m)}, h(x_{\pi(m+1)}, \ldots, x_{\pi(n)})) \tag{2.1}$$

*où $m \in \{1, 2, \ldots, n-2\}$, $g \in \mathscr{B}_{m+1}$, $h \in \mathscr{B}_{n-m}$ et $\pi$ est une permutation sur n.*

Selon cette conjecture, un circuit utilisant une telle décomposition (voir figure 2.1) serait optimal pour des circuits optimaux $g$ et $h$. Nous verrons dans ce chapitre plusieurs théorèmes qui réfutent cette conjecture. La décomposition aurait été un outil très puissant pour l'analyse de la complexité des fonctions. À l'inverse, la réfutation rend l'analyse du produit direct de fonctions un outil plus intéressant. Voici une définition du produit direct de fonction.

**Définition 9** ([Weg87])**.** *Soit $f : \mathbb{B}^n \to \mathbb{B}^k$ et $g : \mathbb{B}^m \to \mathbb{B}^l$ deux fonctions. Leur produit direct $f \times g$ est défini par*

$$\begin{aligned} f \times g : \mathbb{B}^{n+m} &\to \mathbb{B}^{k+l} \\ (x, y) &\mapsto (f(x), g(y)). \end{aligned}$$

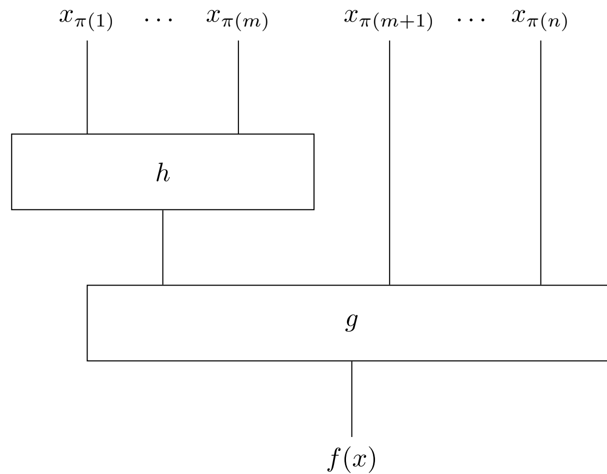


Figure 2.1 : Circuit qui utilise une décomposition d'Ashenhurst.

Une question naturelle que l'on peut se poser est la suivante : existe-t-il deux fonctions $f$ et $g$ tel que $\mathsf{TAILLE}_{circ}(f \times g) < \mathsf{TAILLE}_{circ}(f) + \mathsf{TAILLE}_{circ}(g)$. On s'intéressera plus particulièrement à trois variantes du produit direct de fonctions. Tout d'abord, la section 2.1 abordera une variante où l'on restreint le produit direct à la répétition d'une seule fonction. La question devient donc, est-ce qu'il existe une fonction $f$ telle que $\mathsf{TAILLE}_{circ}(f \times f) < 2 \cdot \mathsf{TAILLE}_{circ}(f)$. Ensuite, dans la section 2.2, on verra une variante du produit direct où chaque fonction utilise la même entrée. Finalement, dans le chapitre 3, on combinera ces deux restrictions. Le tableau 2.I illustre ces différentes possibilités.

| | Entrées différentes | Entrées identiques |
|---|---|---|
| Fonctions différentes | Cas général | Multi-calcul [GKM15, Pot16] |
| Fonctions identiques | Production de masse [Weg87, Pau76, Uhl74] | Espace catalytique [BCK$^+$14, GKM15, Pot16] |

Tableau 2.I : Produit direct de fonctions et ces spécialisations.

## 2.1 Production de masse

La production de masse est le calcul d'une fonction sur plusieurs entrées simultanément. On note le produit direct de la fonction $f$ avec elle-même $n-1$ fois $f^{\times n}$.

$$f^{\times n} : \mathbb{B}^{nk} \to \mathbb{B}^{n}$$
$$(x_1, \ldots, x_n) \mapsto (f(x_1), \ldots, f(x_n))$$

On veut donc savoir s'il existe un $f$ et un $n$ tel que $\mathsf{TAILLE}_{circ}(f^{\times n}) < n \cdot (f)$. Paul [Pau76] et Uhlig [Uhl74, Uhl92] nous proposent deux théorèmes qui répondent à cette question.

### 2.1.1 Théorème d'Uhlig

Le théorème d'Uhlig se veut un analogue aux résultats de Lupanov (voir équations 1.1 et 1.2) qui bornent la taille et la profondeur du circuit optimal de n'importe quelle fonction. Dans le cas d'Uhlig, on obtient une borne lorsque l'on calcule une fonction plusieurs fois, c'est-à-dire la production de masse. Ce théorème est intéressant puisqu'il utilise les mêmes bornes que Lupanov alors qu'il calcule n'importe quelle fonction plusieurs fois.

**Théorème 1** ([Uhl92])**.** *Soit $r(n) : \mathbb{N} \to \mathbb{N}$ une fonction telle que pour tout $n$ dans $\mathbb{N}$, $\log r(n) \leq \frac{n}{\log n}$. Pour toutes fonctions $f \in \mathscr{B}_n$, il existe un circuit $A^{f,r(n)}$ qui calcule $f^{\times r(n)}$ et qui satisfait les propriétés suivantes :*

$$\mathsf{TAILLE}_{circ}(A^{f,r(n)}) \in O\left(\frac{2^n}{n}\right);$$
$$\mathsf{PROF}_{circ}(A^{f,r(n)}) \in O(n).$$

*Démonstration.* Dans cette preuve, pour chaque $f \in \mathscr{B}_n$, $k,t \in \mathbb{N}$ où $k \cdot t < n$, on construit un circuit $A_{f,k,t}$ . Ce circuit calcule la fonction $f^{\times 2^t}$. Le paramètre $k$ sert de facteur de décomposition. Pour calculer la fonction $f$, le circuit calcule plusieurs fonctions de taille $n-k$.

En plus de construire ces circuits, on calcule leur taille et leur profondeur grâce à une

preuve par récurrence sur $t$. Considérons la fonction $T$ pour la taille et la fonction $P$ pour la profondeur :

$$T(n,k,t) = (2^k+1)^t \left[\frac{2^{n-tk}}{n-tk}(1+\alpha(n)) + 2^t \cdot c_1 \cdot n\right] \tag{2.2}$$

$$P(n,k,t) = (n-tk)(1+\beta(n)) + c_2 \cdot tk \tag{2.3}$$

où $\alpha$ et $\beta$ respectent la condition (1.3). Notre hypothèse de récurrence sur $t$ sera : pour tout $f$ et $k$ tel que $kt < n$, alors

$$\mathsf{TAILLE}_{circ}(A_{f,k,t}) \leq T(n,k,t) \text{ et} \tag{2.4}$$

$$\mathsf{PROF}_{circ}(A_{f,k,t}) \leq P(n,k,t). \tag{2.5}$$

Pour finir, on choisit les paramètres $t$ et $k$ pour que le circuit $A_{f,k,t}$ ait les propriétés du circuit $A^{f,r(n)}$ de l'énoncé du théorème.

Rappelons que l'on veut construire les circuits $A_{f,k,t}$ pour tout $f,k$ et $t$ avec $tk < n$. Commençons par les circuits lorsque $t = 1$. Posons $f \in \mathscr{B}_n$ et $k < n$ pour la construction de $A_{f,k,1}$. La figure 2.2 montre le circuit $A_{f,k,1}$ en entier. Chaque partie sera décrite en détail. Considérons toutes les fonctions obtenues en fixant les $k$ dernières variables de $f$. Si $i \in \mathbb{B}^k$, alors on dit que $f_i(x_1,x_2,\ldots,x_{n-k}) = f(x_1,x_2,\ldots,x_{n-k},i_1,i_2,\ldots,i_k)$. On cherche maintenant deux décompositions de $f$ qui utilisent les fonctions $f_i$. On définit les fonctions $g_i : \mathbb{B}^k \to \mathbb{B}$ pour $0 \leq i \leq 2^k$ de la façon suivante :

$$g_i = \begin{cases} f_0 & \text{si } i = 0 \\ f_{2^k-1} & \text{si } i = 2^k \\ f_{i-1} \oplus f_i & \text{sinon.} \end{cases}$$

On appelle $G$ l'ensemble des circuits $g_i'$ calculant $g_i$ pour $0 \leq i \leq 2^k$. Les équations (1.1) et

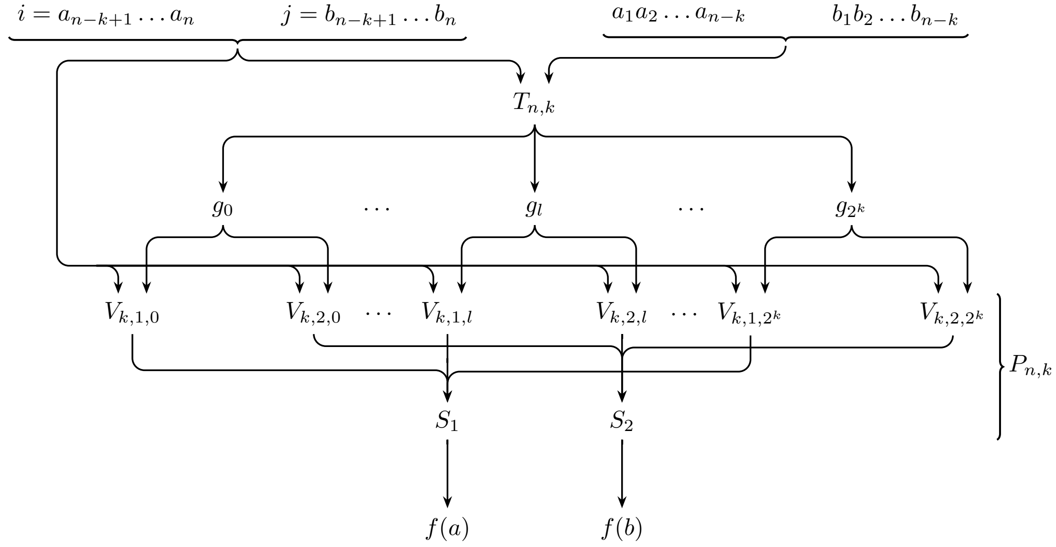


Figure 2.2 : Représentation complète du circuit $A_{f,k,1}$

(1.2) nous permettent de borner la taille et la profondeur des circuits $G$ :

$$\max_{g \in G} \mathsf{TAILLE}_{circ}(g) \sim \frac{2^{n-k}}{n-k}; \tag{2.6}$$

$$\max_{g \in G} \mathsf{PROF}_{circ}(g) \sim n-k. \tag{2.7}$$

On remarque que $f_i = g_0 \oplus g_1 \oplus \cdots \oplus g_i = g_{i+1} \oplus \cdots \oplus g_{2^k}$. Nous allons utiliser cette propriété pour calculer deux valeurs de $f$ à la fois. Prenons deux mots de $n$ bits $a$ et $b$. On nomme $i$ (*resp.* $j$) les $k$ dernières lettres de $a$ (*resp.* $b$). Si $i \leq j$ selon l'ordre lexicographique, alors on peut représenter $f(a)$ et $f(b)$ de la façon suivante :

$$\begin{array}{lclcl} f(a) & = & f_i(a_1, a_2, \dots, a_{n-k}) & = & g_0(a_1, \dots, a_{n-k}) \oplus \cdots \oplus g_i(a_1, \dots, a_{n-k}), \\ f(b) & = & f_j(a_1, a_2, \dots, a_{n-k}) & = & g_{j+1}(a_1, \dots, a_{n-k}) \oplus \cdots \oplus g_{2^k}(a_1, \dots, a_{n-k}). \end{array}$$

Puisque le sous-ensemble de $G$ utilisé pour calculer $f(a)$ est disjoint de celui de $f(b)$, alors il suffit d'avoir une instance de chacun des circuits de $G$ dans $A_{f,k,1}$. Dans le cas inverse où $i > j$, alors il suffit de prendre les fonctions $g_0$ à $g_j$ pour calculer $f(b)$ et $g_{i+1}$ à $g_{2^k}$ pour

$f(a)$. Notre circuit $A_{f,k,1}$ contient tous les circuits de $G$. Il pourra les utiliser pour calculer $f$ à deux reprises. Pour ce faire, il aura besoin de savoir laquelle de ces deux entrées est la plus grande. On construit un circuit trieur $T_{n,k}$ qui reçoit deux mots de $n$ bits puis retourne les entrées qu'il faudra envoyer à chacun des circuits de $G$. On utilisera comme entrées de $T_{n,k}$ les mots $a$ et $b$ définis précédemment avec leurs $k$ derniers bits nommés $i$ et $j$ respectivement. Il y a $2^k+1$ circuits dans $G$ qui prennent chacun $n-k$ entrées, pour un total de $(2^k+1)(n-k)$ sorties. Pour chacun des circuits $g'_l$ dans $G$, le circuit $T_{n,k}$ envoie $a_1, a_2, \ldots, a_{n-k}$ si $l \leq i \leq j$ ou $l > i > j$, sinon, le circuit envoie $b_1, b_2, \ldots, b_{n-k}$. Il est possible de construire ce circuit de façon à obtenir les mesures ci-dessous :

$$\mathsf{TAILLE}_{circ}(T_{n,k}) \leq (2^k+1)(c_3 \cdot n); \tag{2.8}$$

$$\mathsf{PROF}_{circ}(T_{n,k}) \leq c_4 \cdot k. \tag{2.9}$$

La figure 2.3 propose une partie d'une telle construction. Les plus grands facteurs pour la taille et la profondeur de ce circuit sont les noeuds $l \leq i$ et $i \leq j$. Pour la profondeur, les noeuds $\leq$ se réalisent en profondeur $k$ et le reste du circuit est constant. Pour l'analyse de la taille de $T_{n,k}$, regardons le nombre de fois que chacun des sommets de la figure 2.3 doit être répété dans le circuit complet. Un noeud qui dépend de $l$ devra être répété $2^k+1$ fois, c'est-à-dire une fois pour chacun des circuits dans $G$. Ceux qui dépendent à la fois de $l$ et de $p$ devront être répété $(2^k+1)(n-k)$ fois, une pour chacun des bits de sorties de $T_{n,k}$ où plutôt pour chacun des bits d'entrées des circuits de $G$. Le sommet $i \leq j$ est calculé une seule fois puisqu'il ne dépend ni de $l$, ni de $p$. Le sommet $l \leq i$ est le seul qui dépend seulement de $l$. Le reste dépend de $l$ et $p$ et utilise un nombre constant de sommets à chaque fois. Avec la réalisation des sommets $\leq$ en taille linéaire, on obtient

$$\mathsf{TAILLE}_{circ}(T_{n,k}) = c_5 \cdot k + (2^k+1)(c_5 \cdot k) + c_6 \cdot (2^k+1)(n-k) \leq (2^k+1)(c_3 \cdot n) \tag{2.10}$$

tel que désiré.

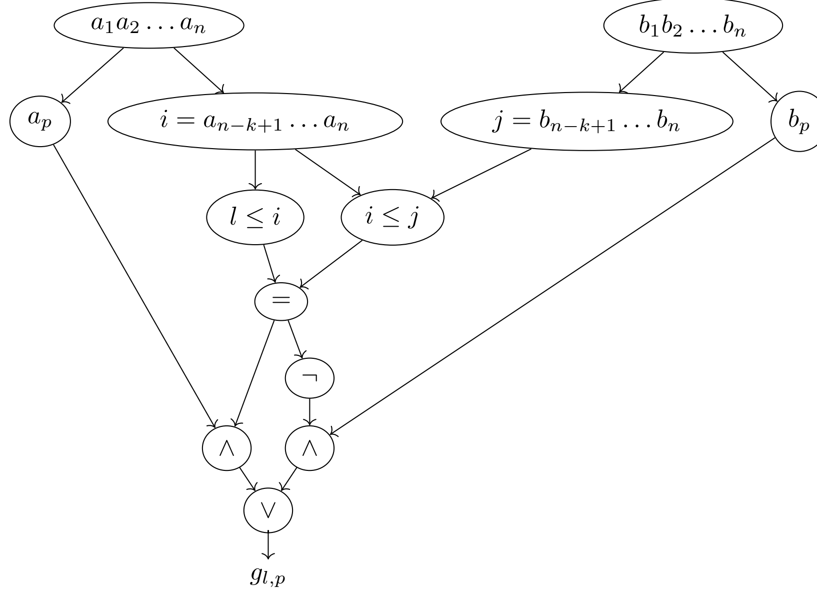


Figure 2.3 : Extrait du circuit $T_{n,k}$ pour calculer le $p^{\text{e}}$ bit pour le circuit $g_l$ où $p \in \{1,2,\dots,n-k\}$ et $g_l \in G$.

Avec l'aide du circuit trieur $T_{n,k}$, nous sommes en mesure d'initialiser correctement les circuits de $G$. Après l'évaluation de ces circuits, il faut identifier lesquels doivent être utilisés pour calculer $f(a)$ et $f(b)$. Les circuits vérificateurs $V_{k,j,l}$ pour tout $j \in \{1,2\}$ et $l \in \{1,2,\dots,2^k\}$ servent à cela. Les circuits $V_{k,1,l}$ (*resp.* $V_{k,2,l}$) pour $l \in \{1,2,\dots,2^k\}$ identifieront quels circuits de $G$ il faut utiliser pour calculer $f(a)$ (*resp.* $f(b)$). Pour ce faire, le circuit $V_{k,j,l}$ calcule la fonction $f_{k,j,l} : \mathbb{B}^k \times \mathbb{B}^k \times \mathbb{B} \to \mathbb{B}$ appropriée ci-dessous :

$$f_{k,1,l}(i,j,z) \mapsto \begin{cases} z & \text{si } l \leq i \leq j \text{ ou } l > i > j \\ 0 & \text{autrement} \end{cases}$$

$$f_{k,2,l}(i,j,z) \mapsto \begin{cases} z & \text{si } i \leq j < l \text{ ou } i > j \leq l \\ 0 & \text{autrement} \end{cases}$$

Tout comme le $T_{n,k}$, on voit que les présents circuits sont composés de l'opération $\leq$ entre mots de $k$ bits. La complexité de chacun de ces circuits est donc au plus linéaire par rapport

à $k$ tant pour la taille que la profondeur.

$$\mathsf{TAILLE}_{circ}(V_{k,j,l}) \leq c_7 \cdot k \tag{2.11}$$

$$\mathsf{PROF}_{circ}(V_{k,j,l}) \leq c_8 \cdot k \tag{2.12}$$

Il manque une dernière partie pour compléter le circuit $A_{f,k,1}$. Il s'agit du circuit $S_1$ calculant la parité de toutes les sorties des circuits $V_{k,1,l}$ où $l \in \{0,1,\ldots,2^k\}$ et du circuit $S_2$ qui fait la même chose pour les circuits $V_{k,2,l}$. On peut construire ces deux circuits tels que

$$\mathsf{TAILLE}_{circ}(S_1) = \mathsf{TAILLE}_{circ}(S_2) \leq c_9 \cdot 2^k \text{ et} \tag{2.13}$$

$$\mathsf{PROF}_{circ}(S_1) = \mathsf{PROF}_{circ}(S_2) \leq c_{10} \cdot k. \tag{2.14}$$

On nomme $P_{n,k}$ le sous-circuit du circuit $A_{f,k,1}$ qui contient $S_1$, $S_2$ et $V_{k,j,l}$ pour $j \in \{1,2\}$, $l \in \{0,\ldots,2^k\}$.

Voyons la construction des circuits $A_{f,k,t}$ lorsque $t > 1$. Pour construire $A_{f,k,t}$, on utilise $2^{t-1}$ copies des circuits $T_{n,k}$ et $P_{n,k}$ nommés $T^c_{n,k}$ et $P^c_{n,k}$ pour $c \in \{1,2,\ldots,2^{t-1}\}$. Il reste maintenant à ajouter les circuits $g_0$ à $g_{2^k}$ correspondants. Or, au lieu d'en ajouter $2^{t-1}$ copies pour chacun des circuits de $G$, on utilise les circuits $A_{g_i,n-k,t-1}$ pour $0 \leq i \leq 2^k$. Ces circuits seront eux-mêmes construits de façon récursive. La figure 2.4 illustre le circuit $A_{f,k,2}$ et permet de voir la construction récursive. Pour chaque $c \in \{1,2,\ldots,2^{t-1}\}$ et $i \in \{0,\ldots,2^k\}$, on relie la $i^{\text{e}}$ sortie du circuit $T^c_{n,k}$ à la $c^{\text{e}}$ entrée du circuit $A_{g_i,n-k,t-1}$ et on relie la $c^{\text{e}}$ sortie du circuit $A_{g_i,n-k,t-1}$ à la $i^{\text{e}}$ entrée du circuit $P^c_{n,k}$.

On souhaite montrer que les circuits $A_{f,k,t}$ pour tous $f,k,t$ valides respectent les bornes (2.4) et (2.5). Pour ce faire, on utilise une preuve par récurrence. Tel que mentionné précédemment, l'hypothèse de récurrence sur $t$ est : pour tout $f : \mathscr{B}_n \to \mathbb{B}$ et $k \in \mathbb{N}$ tel que $kt < n$,

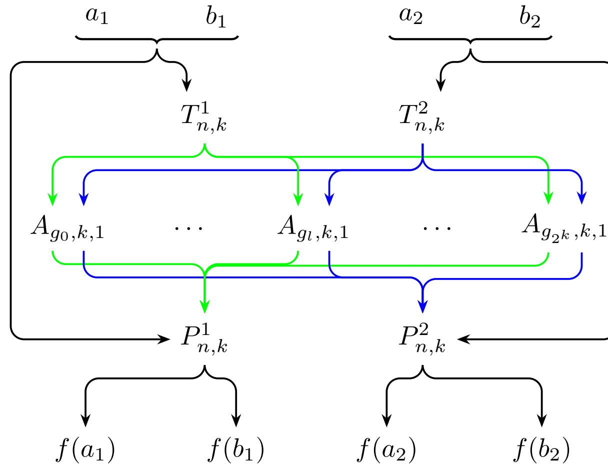


Figure 2.4 : Représentation complète du circuit $A_{f,k,2}$

alors

$$\mathsf{TAILLE}_{circ}(A_{f,k,t}) \leq T(n,k,t) \text{ et}$$
$$\mathsf{PROF}_{circ}(A_{f,k,t}) \leq P(n,k,t).$$

Le cas de base est $t = 1$. Pour tout $f$, le circuit $A_{f,k,1}$ est divisé en trois parties :

- le circuit trieur $T_{n,k}$ ;
- l'ensemble $G$ de circuits qui calculent des sous-fonctions de $f$ ;
- le circuit $P_{n,k}$.

L'équation (2.8) nous donne la taille de $T_{n,k}$ et l'équation (2.6) des circuits de $G$. Avec les équations (2.11) et (2.13), on calcule la taille de $P_{n,k}$

$$\mathsf{TAILLE}_{circ}(P_{n,k}) \leq 2 \cdot (2^k + 1)(c_7 \cdot k) + 2 \cdot (c_9 \cdot 2^k)$$
$$\mathsf{TAILLE}_{circ}(P_{n,k}) \leq c_{11} \cdot k 2^k,$$

ce qui nous donne pour $A_{f,k,1}$

$$\begin{aligned}
\mathsf{TAILLE}_{circ}(A_{f,k,1}) &\leq \overbrace{(2^k+1)\cdot\left(\frac{2^{n-k}}{n-k}\right)(1+\alpha(n-k))}^{G}+\overbrace{(2^k+1)(c_3\cdot n)}^{T_{n,k}}+\overbrace{c_{11}\cdot k2^k}^{P_{n,k}}\\
\mathsf{TAILLE}_{circ}(A_{f,k,1}) &\leq (2^k+1)\left[\frac{2^{n-k}}{n-k}(1+\alpha(n-k))+(c_3\cdot n)+k\right]\\
\mathsf{TAILLE}_{circ}(A_{f,k,1}) &\leq (2^k+1)\left[\frac{2^{n-k}}{n-k}(1+\alpha(n-k))+(2\cdot c_1\cdot n)\right]\\
\mathsf{TAILLE}_{circ}(A_{f,k,1}) &\leq T(n,k,1).
\end{aligned}$$

Pour le calcul de la profondeur, on utilise les équations (2.9), (2.7), (2.12) et (2.14). On obtient

$$\begin{aligned}
\mathsf{PROF}_{circ}(P_{n,k}) &\leq c_8\cdot k+c_{10}\cdot k\\
\mathsf{PROF}_{circ}(P_{n,k}) &\leq c_{12}\cdot k
\end{aligned}$$

pour la profondeur de $P_{n,k}$ et

$$\begin{aligned}
\mathsf{PROF}_{circ}(A_{f,k,1}) &\leq \overbrace{(n-k)(1+\beta(n))}^{G}+\overbrace{c_4\cdot k}^{T_{n,k}}+\overbrace{c_{12}\cdot k}^{P_{n,k}}\\
\mathsf{PROF}_{circ}(A_{f,k,1}) &\leq (n-k)(1+\beta(n))+c_2\cdot k\\
\mathsf{PROF}_{circ}(A_{f,k,1}) &\leq P(n,k,t)
\end{aligned}$$

pour la profondeur de $A_{f,k,1}$. Il nous faut maintenant faire l'étape inductive. Dans la construction récursive des circuits lorsque $t>1$, on retrouve aussi trois parties. On sait que

$$\mathsf{TAILLE}_{circ}(A_{f,k,t})=2^{t-1}(\mathsf{TAILLE}_{circ}(T_{n,k})+\mathsf{TAILLE}_{circ}(P_{n,k}))+\sum_{i=0}^{2^k}\mathsf{TAILLE}_{circ}(A_{g_i,k,t-1})$$

et

$$\mathsf{PROF}_{circ}(A_{f,k,t}) = \mathsf{PROF}_{circ}(T_{n,k}) + \mathsf{PROF}_{circ}(P_{n,k}) + \max_{i=0}^{2^k} \mathsf{PROF}_{circ}(A_{g_i,k,t-1}).$$

Grâce à notre hypothèse d'induction, on sait que $\mathsf{TAILLE}_{circ}(A_{g_i,k,t-1}) \leq T(n,k,t-1)$ pour toutes les fonctions $g_i$. Cela nous permet de calculer la taille de $\mathsf{TAILLE}_{circ}(A_{f,k,t})$

$$\begin{aligned}
\mathsf{TAILLE}_{circ}(A_{f,k,t}) &\leq \overbrace{(2^k+1)T(n-k,k,t-1)}^{G} + \overbrace{2^{t-1}(2^k+1)(c_3 \cdot n)}^{T_{n,k}} + \overbrace{2^{t-1}c_{11}k2^k}^{P_{n,k}} \\
\mathsf{TAILLE}_{circ}(A_{f,k,t}) &\leq (2^k+1)\left[T(n-k,k,t-1) + 2^{t-1}(c_1 \cdot n)\right] \\
\mathsf{TAILLE}_{circ}(A_{f,k,t}) &\leq (2^k+1)\left[(2^k+1)^{t-1}\left[\frac{2^{(n-k)-(t-1)k}}{(n-k)-(t-1)k}(1+\alpha(n-tk))\right.\right. \\
&\qquad \left.\left. +2^{t-1} \cdot c_1 \cdot (n-k)\right] + 2^{t-1}(c_1 \cdot n)\right] \\
\mathsf{TAILLE}_{circ}(A_{f,k,t}) &\leq (2^k+1)\left[(2^k+1)^{t-1}\left[\frac{2^{n-tk}}{n-tk}(1+\alpha(n-tk)) + 2^t \cdot c_1 \cdot n\right]\right] \\
\mathsf{TAILLE}_{circ}(A_{f,k,t}) &\leq (2^k+1)^t\left[\frac{2^{n-tk}}{n-tk}(1+\alpha(n-tk)) + 2^t \cdot c_1 \cdot n\right] \\
\mathsf{TAILLE}_{circ}(A_{f,k,t}) &\leq T(n,k,t).
\end{aligned}$$

De la même façon, pour la profondeur :

$$\begin{aligned}
\mathsf{PROF}_{circ}(A_{f,k,t}) &\leq \overbrace{P(n-k,k,t-1)}^{G} + \overbrace{(c_4 \cdot k)}^{T_{n,k}} + \overbrace{c_{12}k}^{P_{n,k}} \\
\mathsf{PROF}_{circ}(A_{f,k,t}) &\leq (n-tk)(1+\beta(n-tk)) + c_2 \cdot (t-1)k + (c_4 \cdot k) + c_{12}k \\
\mathsf{PROF}_{circ}(A_{f,k,t}) &\leq (n-tk)(1+\beta(n-tk)) + c_2 \cdot tk \\
\mathsf{PROF}_{circ}(A_{f,k,t}) &\leq P(n,k,t).
\end{aligned}$$

Nous avons démontré que pour tout $f \in \mathscr{B}_n \to \mathbb{B}$ et $k,t \in \mathbb{N}$ tel que $tk < n$,le circuit $A_{f,k,t}$ respecte les bornes $T(n,k,t)$ et $P(n,k,t)$.

Il reste maintenant à conclure l'énoncé du théorème grâce à notre famille de circuits.

Rappelons que $r : \mathbb{N} \to \mathbb{N}$ est la fonction qui indique qu'un circuit $A^{f,r(n)}$ où $f : \mathbb{B}^n \to \mathbb{B}$ doit calculer la fonction $f$ sur exactement $r(n)$ entrées. Sans perte de généralité, posons $r(n) = 2^{e(n)}$ où $e : \mathbb{N} \to \mathbb{N}$. Dans la construction, le paramètre $t$ est celui que l'on peut utiliser pour contrôler le nombre d'entrées à calculer. On utilise la valeur de $e(n)$ pour le paramètre $t$. Par la condition $\log r(n) \leq \frac{n}{\log n}$, on prend $e(n) \in o(\frac{n}{\log n})$. Pour le paramètre $k$, on lui donne aussi une valeur selon $n$. On choisit $k(n) = \lceil \log n - \frac{1}{2} \log\log n \rceil$. De cette façon, on respecte la condition suivante :

$$e(n)k(n) \in o\left(n - \left(\frac{n \log\log n}{2 \log n}\right)\right) = o(n) \tag{2.15}$$

On remanie le coefficient de $T(n,k,t)$.

$$\begin{aligned}
(2^{k(n)}+1)^{e(n)} &= 2^{k(n)e(n)}\left(1+\frac{1}{2^{k(n)}}\right)^{e(n)} \\
&\leq 2^{k(n)e(n)}\left(1+\frac{1}{2^{k(n)}}\right)^{2^{k(n)}} \\
&\leq \mathrm{e}2^{k(n)e(n)} && \text{car } \lim_{x\to\infty}(1+\frac{1}{x})^x = \mathrm{e}
\end{aligned}$$

On simplifie une partie de la formule de $T(n,k,t)$.

$$n - e(n)k(n) \in n + o(n)$$

Il reste à montrer que la taille de $T_{n,k}$ et de $P_{n,k}$ est moins importante que celles des circuits qui réalisent les fonctions de $G$, c'est-à-dire que $2^{e(n)} \cdot c_1 \cdot n \in o\left(\frac{2^{n-e(n)k(n)}}{n-e(n)k(n)}\right)$. Rappelons que

$f(n) \in o(g(n))$ si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

$$\begin{aligned}
\lim_{n\to\infty} \frac{2^{e(n)} n c_1}{\frac{2^{n-e(n)k(n)}}{n-e(n)k(n)}} &= \lim_{n\to\infty} \frac{2^{e(n)+\log(n)+\log(c_1)+\log(n-e(n)k(n))+e(n)k(n)}}{2^n} \\
&= \lim_{n\to\infty} \frac{2^{o(n)}}{2^n} \\
&= 0
\end{aligned}$$

Voyons que la taille des circuits $A_{f,k(n),e(n)}$ est $O\left(\frac{2^n}{n}\right)$ :

$$\begin{aligned}
\mathsf{TAILLE}_{circ}(A_{f,k(n),e(n)}) &\leq T(n,k(n),e(n)) \\
&\leq (2^{k(n)}+1)^{e(n)} \left[\frac{2^{n-e(n)k(n)}}{n-e(n)k(n)}(1+\alpha(n-e(n)k(n))) + 2^{e(n)} c_1 n\right] \\
&\leq \mathrm{e} 2^{k(n)e(n)} \left[\frac{2^{n-e(n)k(n)}}{n-e(n)k(n)}(1+\alpha(n-e(n)k(n))) + 2^{e(n)} c_1 n\right] \\
&\in 2^{e(n)k(n)} \left[\theta\left(\frac{2^{n-e(n)k(n)}}{n-e(n)k(n)}\right) + o\left(\frac{2^{n-e(n)k(n)}}{n-e(n)k(n)}\right)\right] \\
&\subseteq \theta\left(\frac{2^n}{n+o(n)}\right) + o\left(\frac{2^n}{n+o(n)}\right) \\
&\subseteq O\left(\frac{2^n}{n}\right).
\end{aligned}$$

De la même façon pour la profondeur de $A_{f,k(n),e(n)}$, on obtient

$$\begin{aligned}
\mathsf{PROF}_{circ}(A_{f,k(n),e(n)}) &\leq P(n,k(n),e(n)) \\
&\leq (n-e(n)k(n))(1+\beta(n-e(n)k(n))) + c_2 \cdot e(n)k(n) \\
&\in O(n+o(n)) + o(n) \\
&\subseteq O(n).
\end{aligned}$$

Pour toute fonction $f$, le circuit $A_{f,k(n),e(n)}$ respecte la taille et la profondeur souhaitée en plus de calculer le bon nombre de fois la fonction $f$. □

### 2.1.2 Théorème de Paul

Voici le théorème Paul [Pau76] qui permet lui aussi de réfuter la conjecture de Ashenhurst.

**Théorème 2** ([Pau76])**.** *Pour tout $\varepsilon > 1$, il existe $d, n \in \mathbb{N}$ et $f : \mathbb{B}^n \to \mathbb{B}$ tel que*

- $\mathsf{TAILLE}_{circ}(f) \geq 2^{n^{\frac{1}{d}}}$
- $\mathsf{TAILLE}_{circ}(f \vee f) \leq (1+\varepsilon)\mathsf{TAILLE}_{circ}(f)$

Celui-ci est plus faible que le résultat de la section précédente, mais utilise une tout autre approche. Il ne calcule pas la production de masse d'une fonction, mais la disjonction de celle-ci. Cela est suffisant pour réfuter la conjecture. Pour toute fonction $f : \mathbb{B}^n \to \mathbb{B}$, la fonction $g_f = f(x_1, \ldots, x_n) \vee f(y_1, \ldots, y_n)$ peut être décomposée en deux parties qui calculent $f$. La conjecture affirme qu'un circuit optimal pour $g_f$ utilise cette décomposition. Un tel circuit $C$ aurait une taille plus grande que $2 \cdot \mathsf{TAILLE}_{circ}(f)$. Cela contredit le théorème de Paul.

## 2.2 Multi-calcul

Le multi-calcul est le calcul de plusieurs fonctions sur la même entrée.

**Définition 10.** *Soit $f : \mathbb{B}^n \to \mathbb{B}$ et $g : \mathbb{B}^n \to \mathbb{B}$ deux fonctions. Leur multi-calcul $f \| g$ est défini par*

$$\begin{aligned} f \| g : \mathbb{B}^n &\to \mathbb{B}^2 \\ x &\mapsto (f(x), g(x)). \end{aligned}$$

On note le multi-calcul de $f$ avec lui-même $k-1$ fois $f^{\|k}$. On verra au chapitre 3 que le multi-calcul d'une fonction avec elle-même est équivalent à l'espace catalytique.

### 2.2.1 Programme de branchement multi-calcul

Un programme de branchement $k$-multi-calcul est un programme de branchement possédant $k$ noeuds sources $\nabla_1, \nabla_2, \ldots, \nabla_k$ et $2k$ noeuds finaux $\top_1, \bot_1, \top_2, \bot_2, \ldots, \top_k, \bot_k$. Un programme $P$ $k$-catalytique calcule une fonction $f_1 \| f_2 \| \ldots \| f_k$ si pour chaque $i \in \{1, 2, \ldots, k\}$ et chaque $x \in \mathbb{B}^n$, le calcul de $P$ en commençant au noeud initial $\nabla_i$ sur l'entrée $x$ atteint $\top_i$ (*resp.* $\bot_i$) si et seulement si $f_i(x) = 1$ (*resp.* $f_i(x) = 0$). La proposition suivante nous informe que l'on ne peut pas faire pire dans un programme de branchement $k$-catalytique que calculer chaque fonction indépendamment en utilisant un programme de branchement optimal pour chacune des fonctions.

**Proposition 1** ([GKM15])**.** *Pour n'importe quelles fonctions $f_1, f_2, \ldots, f_k : \mathbb{B}^n \to \mathbb{B}$ :*

$$\mathsf{TAILLE}_{pb}(f_1 \| f_2 \| \ldots \| f_k) \leq \sum_{j=1}^{k} \mathsf{TAILLE}_{pb}(f_i).$$

*En particulier,* $\mathsf{TAILLE}_{pb}(f^{\|k}) \leq k \cdot \mathsf{TAILLE}_{pb}(f)$.

### 2.2.2 Multi-calcul avec des fonctions choisies aléatoirement

On présente deux résultats de Girard et coll. [GKM15] à propos de la taille espérée du multi-calcul de $k$ fonctions choisies aléatoirement.

**Proposition 2** ([GKM15])**.** *Soit $n$ et $k$ des nombres naturels et $f_1$, $f_2$, $\ldots$, $f_k : \mathbb{B}^n \to \mathbb{B}$ des fonctions choisies uniformément et aléatoirement. Alors l'expression*

$$\mathsf{TAILLE}_{pb}(f_1 \| \ldots \| f_k) \geq \frac{1}{3} \cdot k \cdot \frac{2^n}{n + \log k}$$

*est vraie avec forte probabilité.*

*Démonstration.* Il y a au plus $S^{2S} n^S \leq S^{3S}$ programmes de branchement $k$-multi-calcul non

isomorphes de taille $S \geq n$. Pour $S = \frac{1}{3} \cdot k \cdot \frac{2^n}{n+\log k}$, la borne supérieure devient

$$\left(\frac{1}{3} \cdot k \cdot \frac{2^n}{n+\log k}\right)^{k \cdot \frac{2^n}{n+\log k}} \ll 2^{k \cdot 2^n}.$$

Il y a $2^{k \cdot 2^n}$ façons possibles d'obtenir un tuple de fonctions $(f_1, \ldots, f_k)$. Puisque la limite lorsque $n$ tend vers l'infinie du rapport du nombre de programmes de branchement sur le nombre de fonctions tend vers 0, elle nous permet de conclure la preuve de la propostion.

□

Puisque l'on sait que la taille de programme de branchement pour n'importe quelle fonction $f : \mathbb{B}^n \to \mathbb{B}$ est $\mathsf{TAILLE}_{pb}(f) \in O(2^n/n)$, alors obtient le corollaire suivant.

**Corollaire 1** ([GKM15])**.** *Soit n et k des nombres naturels et $f_1$, $f_2$, ..., $f_k : \mathbb{B}^n \to \mathbb{B}$ des fonctions choisies uniformément et aléatoirement. Alors l'expression*

$$\mathsf{TAILLE}_{pb}(f_1 \| \ldots \| f_k) \in \Theta\left(\sum_{j=1}^{k} \mathsf{TAILLE}_{pb}(f_j)\right).$$

*est vraie avec probabilité qui tend vers 1 lors n tend vers l'infinie.*

Contrairement à la production de masse où l'on sait répéter une fonction plusieurs fois grâce au théorème d'Uhlig, il semble qu'il soit difficile d'exploiter le multi-calcul dans le cas général.

# CHAPITRE 3

# ESPACE CATALYTIQUE

L'espace catalytique est aussi une variante du produit direct de fonctions. Il s'agit de calculer une fonction une multitude de fois sur la même entrée. Pourquoi voudrait-on répéter exactement le même calcul ? On peut voir dans [GKM15] qu'il y a une équivalence entre un programme de branchement $k$-multi-calcul pour la fonction $f^{\|k}$ et une machine de Turing catalytique. Chacune des $k$ entrées du programme représente l'une des configurations de départ du ruban auxiliaire. Ce chapitre reprend des résultats connus [BCK$^+$14, Pot16] en offrant parfois une précision supplémentaire. On retrouve cette précision entre autres au niveau de la présentation du lemme 7 et au niveau technique lors de l'amélioration de la borne du théorème 5.

## 3.1 Calcul transparent

Pour faire un calcul dit transparent sur un anneau $R$, on utilise une machine avec les registres $\vec{r} = r_1, r_2, \ldots, r_m$ et $\vec{x} = x_1, x_2, \ldots, x_n$. Les registres $\vec{x}$ sont les registres d'entrées et l'on ne peut y écrire. Les valeurs initiales des registres $\vec{r}$ seront notées $\vec{\tau} = \tau_1, \tau_2, \ldots, \tau_m$. On considère toujours le vecteur $\vec{\tau}$ comme quelconque, c'est-à-dire qu'il pourrait prendre n'importe quelle valeur. Chaque registre contient un élément de l'anneau $R$. Cette machine comprend des instructions de la forme suivante : $r_i \leftarrow r_i \pm (u \times v)$ où $u$ et $v$ peuvent être un élément de $R$ ou un autre registre que $r_i$. Les trois opérateurs $+, -$ et $\times$ sont ceux de l'anneau $R$. On peut voir qu'il est facile de renverser n'importe laquelle de ces instructions. Il suffit d'inverser les $+$ et les $-$. Pour $I$ une instruction, on notera cette nouvelle instruction (l'inverse de $I$) $I^{-1}$.

Un programme pour cette machine est une suite d'instructions. Chaque programme est automatiquement réversible. Pour un programme $P = I_1, I_2, \ldots, I_k$, son inverse est le pro-

gramme $P^{-1} = I_k^{-1}, \ldots, I_2^{-1}, I_1^{-1}$. Ainsi, les programmes $PP^{-1}$ et $P^{-1}P$ ne modifient pas les registres et sont équivalents au programme vide.

Voyons un exemple pour s'en convaincre. Prenons une machine simple sur l'anneau $\mathbb{B}$ avec $\vec{r} = r_1, r_2$ et $\vec{x} = x_1$ et $P$ définie de la façon suivante.

$$P : \begin{cases} r_1 \leftarrow r_1 + (r_2 \times x_1) \\ r_2 \leftarrow r_2 - (r_1 \times r_1) \\ r_1 \leftarrow r_1 - (r_2 \times 1) \end{cases}$$

Montrons que $P^{-1}P$ est équivalent au programme vide.

$$P^{-1}P : \begin{cases} I_1 \quad : r_1 \leftarrow r_1 + (r_2 \times 1) \\ I_2 \quad : r_2 \leftarrow r_2 + (r_1 \times r_1) \\ I_3 \quad : r_1 \leftarrow r_1 - (r_2 \times x_1) \\ I_4 \quad : r_1 \leftarrow r_1 + (r_2 \times x_1) \\ I_5 \quad : r_2 \leftarrow r_2 - (r_1 \times r_1) \\ I_6 \quad : r_1 \leftarrow r_1 - (r_2 \times 1) \end{cases}$$

On peut voir que $I_3 = I_4^{-1}$. Si l'on exécute $I_3$ suivie de $I_4$, on obtient que $r_1' \leftarrow r_1 - (r_2 \times x_1) + (r_2 \times x_1) = r_1$. Les valeurs de tous les registres après l'exécution de ces deux instructions sont les mêmes que celles avant l'exécution. On peut donc effacer ces deux instructions de notre programme. Il suffit alors de répéter ce processus deux fois en remarquant que $I_2 = I_5^{-1}$ et $I_1 = I_6^{-1}$. Ce faisant, on trouve que $P^{-1}P$ est équivalent au programme vide.

On dit que la fonction $f(\vec{x})$ *peut être calculée de façon transparente* s'il existe un programme réversible $P$ tel que, pour toute valeur initiale $\vec{\tau}$ affectée initialement aux registres $\vec{r}$, un registre $r_i$ contient la valeur $\tau_i + f(\vec{x})$ à la fin de son exécution.

On remarque qu'il est possible de changer quel registre recevra la valeur de $f(\vec{x})$. Par

exemple, on pourrait vouloir utiliser $r_1$ au lieu de $r_i$ pour obtenir $r_1 = \tau_1 + f(\vec{x})$ après l'exécution du programme. Pour ce faire, on construit un programme $P'$ où l'on échange toutes les occurrences de $r_1$ par $r_i$ et $r_i$ par $r_1$.

**Définition 11** ([BCK⁺14]). $\mathsf{TP}(R,s,m)$ *est la classe de fonctions qui peuvent être calculées par un programme réversible de façon transparente sur l'anneau R, en utilisant au plus s instructions et m registres.* $\mathsf{TP}(R)$ *est la classe de fonctions dans* $\mathsf{TP}(R, poly(n), poly(n))$.

### 3.2 La machine catalytique

Une machine catalytique est une machine de Turing avec un ruban d'entrée, un ruban de travail et un ruban auxiliaire. La particularité de cette machine est que le contenu du ruban auxiliaire est quelconque au début du calcul et doit être restauré à la fin de celui-ci. La machine possède donc un ruban potentiellement plus grand que son espace de travail, souvent de façon exponentielle, qu'elle peut utiliser de façon limitée. La question est donc de savoir si l'ajout de ce ruban auxiliaire augmente la puissance d'une machine de Turing qui possède une quantité limitée d'espace de travail régulière.

**Définition 12** ([BCK⁺14]). *Une machine de Turing catalytique est une machine de Turing équipée d'un ruban de travail auxiliaire en plus du ruban d'entrée et du ruban de travail des machines de Turing pour l'espace. Pour chaque configuration initiale du ruban auxiliaire, la machine doit retourner le contenu de ce ruban à sa valeur initiale à la fin du calcul.*

Nous utiliserons le calcul transparent pour montrer un exemple où la machine catalytique reconnaît plus de langages que ne semble le permettre la meilleure borne non catalytique.

**Définition 13** ([BCK⁺14]). *Soit* $S, S_c : \mathbb{N} \to \mathbb{N}$. *La classe* $\mathsf{CSPACE}(S(n), S_c(n))$ *est définie comme l'ensemble de tous les langages qui peuvent être décidés par une machine de Turing catalytique qui utilise* $S(n)$ *espace sur son ruban de travail et* $S_c(n)$ *espace sur son ruban auxiliaire sur une entrée de longueur n.*

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Ruban d'entrée | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $\cdots$ | $x_n$ | | |
| Ruban de travail | 0 | 0 | 0 | $\cdots$ | $\log(n)$ | | | |
| Ruban catalytique | 0 | 1 | 1 | 0 | 1 | 0 | $\cdots$ | $poly(n)$ |

Figure 3.1 : Configuration initiale possible des rubans d'une machine de Turing catalytique. Le ruban de travail est souvent plus court que le ruban catalytique.

On utilisera $\mathsf{CSPACE}(S(n))$ comme abréviation pour $\mathsf{CSPACE}(S(n), 2^{S(n)})$. On s'intéresse à ces classes puisqu'avec $S(n)$ espace sur le ruban de travail, il est impossible de noter notre position dans le ruban auxiliaire pour plus de $2^{S(n)}$ positions. Il ne faut toutefois pas éliminer la possibilité qu'une classe $\mathsf{CSPACE}(S(n), \omega(2^{S(n)}))$ soit intéressante. Il est possible d'imaginer une machine qui fonctionne sans avoir besoin de noter sa position dans le ruban. On s'intéresse tout particulièrement à la classe où la machine de Turing catalytique possède une quantité logarithmique de mémoire sur son ruban de travail. On appelle cette classe $\mathsf{CL}$ pour *catalytic logspace*.

### 3.3 Propriétés de $\mathsf{CL}$

Je vais refaire quelques résultats classiques de la théorie de la complexité pour l'espace catalytique. L'objectif est de démontrer que certaines de nos intuitions et habitudes provenant de l'étude de l'espace déterministe restent valides.

#### 3.3.1 Simulation d'une machine de Turing catalytique

Une façon facile d'établir une borne supérieure sur la capacité de notre nouveau modèle est de le simuler par un autre modèle. Les deux lemmes suivants présenteront des simulations d'une machine de Turing catalytique.

**Lemme 1.** *Soit $\mathscr{L} \in \mathsf{CSPACE}(S(n), S_c(n))$ un langage et $M_c$ une machine de Turing catalytique reconnaissant ce langage. Il existe une machine régulière $M$ utilisant $O(\max(S(n), S_c(n)))$ espace qui reconnaît $\mathscr{L}$.*

*Démonstration.* Rappelons que les machines de Turing catalytiques doivent être correctes sur toutes les configurations de départ de leur ruban catalytique. Or, on peut utiliser cette propriété à notre avantage. Il suffit de considérer le ruban auxiliaire comme un ruban rempli exclusivement de zéros lors de notre simulation. La simulation de $M_c$ n'est donc pas bien différente d'une simulation d'une machine avec 2 rubans de travail de taille $\max(S(n), S_c(n)$. Il suffit alors à la machine $M$ de simuler la machine $M_c$ en espace $O(\max(S(n), S_c(n))$ en utilisant ses 2 rubans de travail régulier. □

Cela nous donne notre première borne supérieure pour $\mathsf{CL}$.

**Corollaire 2.** $\mathsf{CL} \subseteq \mathsf{PSPACE}$.

### 3.3.2 Composition de fonctions dans $\mathsf{CSPACE}(f(n))$

Il est connu que l'espace requis pour représenter $f(x)$ peut être beaucoup plus grand que l'espace requis pour la calculer. Cela pose problème puisqu'une machine calculant $f \circ g$ ne peut donc pas nécessairement calculer la valeur de $g$ au complet et stocker le résultat, car elle ne possède peut-être pas assez de place sur son ruban de travail. Or, il existe une solution à ce problème. Il s'agit de calculer la valeur de retour de $g$ une case à la fois. Lorsque l'on veut connaître une case différente, on recalcule $g$ au complet. On montre une proposition analogue à celle décrite dans Périfel [Per14] à la section 4.1.3 sur la composition de fonctions par une machine bornée en espace, mais en considérant ici l'espace catalytique.

**Proposition 3.** *Soit $f, g : \mathbb{B}^* \to \mathbb{B}^*$ deux fonctions. Si la fonction $f$ est calculable en espace catalytique $\mathsf{CSPACE}(s_f(n), cs_f(n))$ où $s_f(n) \in \Omega(\log(n))$ et $s_g(n)$ en $\mathsf{CSPACE}(s_g(n), cs_g(n))$. Leur composition $f \circ g$, $x \mapsto f(g(x))$, est calculable en espace catalytique $\mathsf{CSPACE}(s_g(|x|) + s_f(|g(x)|), \max(cs_g(|x|), cs_f(|g(x)|)))$.*

*Démonstration.* Il est possible en espace $\mathsf{CSPACE}(s_g(|x|) + \log(|g(x)|), cs_g(|x|))$ de calculer la valeur du i[e] symbole de $g(x)$. Soit $M_g$ une machine de Turing catalytique qui calcule $g(x)$ en espace $cs_g(n)$. Définissons une machine de Turing catalytique qui, sur entrée $(i, x)$, calcule le i[e] bit de $g(x)$.

- Simuler $M_g(x)$ une première fois en notant la case la plus à gauche sur laquelle la machine inscrit un caractère non blanc. Cela se fait en espace $cs_g(|x|)$ pour la simulation en plus d'un $\log(|g(x)|)$ pour noter la position de la case.

- Simuler $M_g(x)$ une deuxième fois en notant la valeur de la i$^{e}$ case sur le ruban de sortie. On peut déterminer la position exacte de cette case grâce à la simulation de l'étape précédente.

À la fin du calcul, nous possédons bien la dernière valeur inscrite dans la i$^{e}$ case de la sortie de $g(x)$. Il reste à faire une machine qui calcule $f \circ g$. Elle pourrait suivre l'algorithme 3.2. Cet algorithme exécute $g$ à chaque fois qu'il a besoin d'un de ces bits. On peut réutiliser l'espace catalytique de $f$ lors des calculs de $g$ puisqu'un calcul catalytique garanti que son ruban catalytique reviendra à son état initial.

□

## 3.4 $\mathsf{TC}^1$ dans $\mathsf{CL}$

Dans cette section, je vais présenter les étapes et techniques utilisées par Buhrman et coll. pour montrer la simulation de $\mathsf{TC}^1$ par $\mathsf{CL}$ [BCK$^+$14]. La classe $\mathsf{TC}^1$ (pour *logarithmic depth treshold circuit*) contient les langages qui peuvent se résoudre par une famille de circuits de profondeur logarithmique avec les changements suivants :

- les noeuds peuvent avoir un degré entrant arbitraire ;

- les noeuds sont étiquetés par un nombre naturel appelé le seuil du sommet.

$i \leftarrow 1$
**tant que** $M_f$ n'a pas terminé son calcul **faire**
  $a \leftarrow$ le $i^{e}$ symbole de $g(x)$
  Simuler une étape de $M_f$ en remplaçant la valeur de la tête de lecture d'entrée par $a$
  Ajuster $i$ selon le mouvement de la tête de lecture d'entrée de $M_f$
**fin tant que**

Figure 3.2 : Algorithme pour le calcul de $f \circ g$ par une machine de Turing.

Un noeud avec $k$ entrées et un seuil $s$ calcule la fonction $T_{k,s}$ suivante

$$T_{k,s} : \mathbb{B}^k \to \mathbb{B}$$
$$b_1, b_2, \ldots, b_k \mapsto \sum_{i=1}^{k} b_i \geq s.$$

On borne les seuils par le degré entrant maximal du circuit. En effet, un seuil plus élevé que cela ne pourrait jamais être vrai. La fonction de seuil est une généralisation des fonctions $\wedge$ et $\vee$ que l'on a utilisées précédemment. On remarque que $\wedge = T_{2,2}$ et $\vee = T_{2,1}$. Pour commencer, on construit un programme transparent à partir d'un circuit de type $\mathsf{TC}^1$. Ensuite, on simule un programme transparent avec une machine de Turing catalytique.

### 3.4.1 Construction des programmes transparents

Nous allons montrer quelques lemmes techniques pour pouvoir simuler les noeuds de seuil.

**Lemme 2** ([BCK⁺14])**.** *Soit $r_0, r_1, \ldots r_k$ des registres sur un anneau $R$. Soit $P$ un programme transparent qui n'utilise pas $r_0$ et calcule $r_i \leftarrow \tau_i + f_i(\vec{x})$ pour chaque $i = 1, \ldots k$. Il existe deux programmes $I_1$ et $I_2$ tels que le programme $I_1, P, I_2$ calcule $r_0 \leftarrow \tau_0 + \sum_{i=1}^{k} f_i(\vec{x})$. La longueur totale de $I_1$ et $I_2$ est $2k$.*

*Démonstration.* Considérons le programme qui ajoute simplement les registres $r_1, \ldots, r_k$ à $r_0$ après l'exécution de $P$.

$P$

$\forall i \in \{1, 2, \ldots, k\} (r_0 \leftarrow r_0 + r_i)$

Le registre $r_0$ vaut $\tau_0 + \sum_{i=1}^{k} (\tau_i + f_i(\vec{x}))$ à la fin de l'exécution. Il faut enlever les valeurs initiales des registres pour obtenir le résultat souhaité. Avant l'exécution de $P$, on souhaite que la valeur de $r_0$ soit $\tau_0 - \sum_{i=1}^{k} \tau_i$. Le programme suivant avec les sections $I_1$ et $I_2$ répond aux critères du lemme.

$I_1: \forall i \in \{1,2,\ldots,k\}(r_0 \leftarrow r_0 - r_i)$

$P$

$I_2: \forall i \in \{1,2,\ldots,k\}(r_0 \leftarrow r_0 + r_i)$

Le nombre d'instructions est bien $2k$ pour $I_1$ et $I_2$. □

**Lemme 3** ([BCK+14])**.** *Il y a un programme transparent $P$ avec $2k-1$ instructions sur un anneau $R$ qui calcule pour $i \in \{1,2,\ldots,k\}$,*

$$r_i \leftarrow \tau_i + m_1 \times \cdots \times m_i, \tag{3.1}$$

*où $m_1,\ldots,m_k$ sont des registres d'entrées, des registres de travail (autres que les registres $r_i$ pour $i \in \{1,2,\ldots,k\}$) ou des constantes.*

*Démonstration.* Le programme suivant calcule les valeurs souhaitées.

1. Pour $i$ de $k$ à 2, exécuter l'instruction $r_i \leftarrow r_i - r_{i-1} \times m_i$.
2. $r_1 \leftarrow r_1 + m_1$.
3. Pour $i$ de 2 à $k$, exécuter l'instruction $r_i \leftarrow r_i + r_{i-1} \times m_i$.

□

Contrairement au lemme 2, le lemme précédent ne montre pas comment multiplier le résultat de programmes réversibles arbitraires, seulement de constantes ou de registres. Un programme qui multiplie le résultat de fonctions arbitraires aurait comme signature

$$r \leftarrow \tau + \prod_{i=1}^{k} f_i(\vec{x}).$$

Ce résultat est malgré tout suffisant pour démontrer les lemmes qui suivent.

**Lemme 4** ([BCK⁺14])**.** *Soit $k$ un nombre naturel, $r$ et $r_f$ des registres sur un anneau commutatif $R$ et $\tau$ la valeur initiale du registre $r$. Il existe des programmes $I_1$, $I_2$ et $I_3$ sur $R$ tels que pour n'importe quel programme transparent $P$ qui n'utilise pas les registres de $I_1$, $I_2$, $I_3$ autres que $r$ et calcule*

$$r \leftarrow \tau + f(\vec{x}),$$

*le programme $I_1$, $P$, $I_2$, $P^{-1}$, $I_3$ calcule*

$$r_f \leftarrow \tau_f + [f(\vec{x})^k].$$

*Les programmes $I_1$, $I_2$ et $I_3$ ont une longueur dans $O(k)$ et utilisent $O(k)$ registres.*

*Démonstration.* Posons $y = f(\vec{x})$ et $a = \tau$. On veut calculer $y^k = (a+y-a)^k$. En utilisant l'expansion binomiale, on obtient

$$y^k = \sum_{i=0}^{k} \underbrace{(-1)^i \binom{k}{i}}_{c_i} a^i (a+y)^{k-i}.$$

On nomme la partie constante de la sommation $c_i$. Considérons le programme suivant :

$I_1$ : Pour $i \in \{0,\dots,k\}$, faire $r_f \leftarrow r_f - c_i \times r_i \times r^i$ ;

$P$ ;

$I_2$ : Pour $i \in \{0,\dots,k\}$, faire $r_i \leftarrow r_i + r^{k-i}$ ;

$P^{-1}$ ;

$I_3$ : Pour $i \in \{0,\dots,k\}$, faire $r_f \leftarrow r_f + c_i \times r_i \times r^i$.

Voyons le contenu des registres $r_f$ et $r_i$ pour $i \in \{0,\dots,k\}$ après les instructions de $I_1$, $I_2$ et $I_3$.

| | $r_f$ | $r_i$ |
|---|---|---|
| $I_1$ | $\tau_f - \sum_{i=0}^{k} c_i \tau_i a^i$ | $\tau_i$ |
| $I_2$ | $\tau_f - \sum_{i=0}^{k} c_i \tau_i a^i$ | $\tau_i + (a+y)^{k-i}$ |
| $I_3$ | $\tau_f - \sum_{i=0}^{k} c_i \tau_i a^i + \sum_{i=0}^{k} c_i (\tau_i + (a+y)^{k-i}) a^i$ | $\tau_i + (a+y)^{k-i}$ |

On peut réduire la valeur de $r_f$

$$\begin{aligned} r_f &= \tau_f - \sum_{i=0}^{k} c_i \tau_i a^i + \sum_{i=0}^{k} c_i (\tau_i + (a+y)^{k-i}) a^i \\ r_f &= \tau_f - \sum_{i=0}^{k} c_i \tau_i a^i + \sum_{i=0}^{k} c_i \tau_i a^i + \sum_{i=0}^{k} c_i (a+y)^{k-i} a^i \\ r_f &= \tau_f + y^k. \end{aligned}$$

Le registre $r_f$ contient bien la valeur souhaitée. Pour la construction de $I_1$, $I_2$ et $I_3$, on utilise le lemme 3 qui nous permet de le faire avec une longueur dans l'ordre de $O(k)$. Il y a $O(k)$ registres utilisés. □

Pour le prochain lemme, on définit la valeur de $[\![a \neq b]\!]$ à 1 si $a \neq b$ et 0 sinon.

**Lemme 5** ([BCK⁺14])**.** *Soit $p$ un nombre premier, $R$ l'anneau $\mathbb{Z}_p$, $s \in R$ et $r_0,\dots,r_k$ des registres sur $R$. Il y a des programmes $I_1$, $I_2$ et $I_3$ sur $R$ tels que pour n'importe quel programme transparent $P$ qui calcule pour tout $i \in \{1,\dots,k\}$*

$$r_i \leftarrow \tau_i + f_i(\vec{x}),$$

*le programme $I_1$, $P$, $I_2$, $P^{-1}$, $I_3$ calcule*

$$r_0 \leftarrow \tau_0 + \left[\!\!\left[ \sum_{i=1}^{k} f_i(\vec{x}) \neq s \right]\!\!\right]. \tag{3.2}$$

*La longueur totale de $I_1$, $I_2$ et $I_3$ est $O(p+k)$ et $O(p)$ registres sont utilisés.*

*Démonstration.* Par le lemme 2, il existe des programmes $S_1$ et $S_2$ pour construire le programme $P' = S_1, P, S_2$ qui calcule de façon transparente $d = \sum_{i=1}^{k} f_i(\vec{x}) - s$ dans le registre $r$. La longueur totale de $S_1$ et $S_2$ est $2k$. On remarque que $d$ est différent de zéro si et seulement si $\sum_{i=1}^{k} \neq s$. Puisque $R$ est un anneau de taille $p$, par le petit théorème de Fermat, la valeur de $d^{p-1}$ est 1 si et seulement si $d$ n'est pas égal à zéro. Par le lemme 4, on a des programmes $E_1$, $E_2$ et $E_3$ tels que $E_1$, $P'$, $E_2$, $P'^{-1}$, $E_3$ calcule de façon transparente $d^{p-1}$, c'est-à-dire $[\![\sum_{i=1}^{k} f_i(\vec{x}) \neq s]\!]$. On trouve le programme complet à la figure 3.3. On y trouve la construction de $I_1$, $I_2$ et $I_3$ qui nous donne le programme voulu. La longueur totale de ces parties est $O(k+p)$ et elles utilisent $O(p)$ registres. □

Le lemme suivant est une version plus générale et complète du lemme de [BCK+14]. À l'origine, les auteurs ne considéraient que le seuil $\lfloor k/2 \rfloor + 1$ pour la simulation d'un noeud majorité. De plus, le cas où le nombre de bits $k$ est impair était omis.

**Lemme 6.** *Soit $s > 0$ un nombre naturel et $b_1, b_2, \ldots b_k$ des bits, alors on peut calculer la fonction $T_{k,s}$ comme suit :*

$$T_{k,s}(b_1, b_2, \ldots, b_k) = \left[\!\!\left[ \sum_{j=0}^{s-1} \left[\!\!\left[ \sum_{i=1}^{k} b_i \neq j \right]\!\!\right] \neq s-1 \right]\!\!\right].$$

*Démonstration.* La partie $[\![\sum_{i=1}^{k} b_i \neq j]\!]$ vérifie que la somme des bits $b_1, \ldots, b_k$ n'est pas $j$. On fait cette vérification $s$ fois (pour $j$ de 0 à $s-1$). Au plus une de ces vérifications peut échouer. Dans ce cas, cela signifie que la somme des bits $b_1, \ldots, b_k$ est plus petite que $s$ et

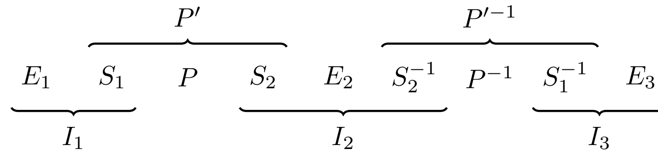


Figure 3.3 : Programme complet du lemme 5.

donc ne dépasse pas le seuil. La sommation sur $j$ sera égale à $s-1$ et donc la formule sera fausse, tel que souhaité. Dans l'autre cas où la somme des bits est plus grande ou égale à $s$, la valeur de la sommation sur $j$ sera $s$. Puisque $s \neq s-1$, on retournera 1 tel que souhaité. Le tableau 3.I montre une trace du calcul de $T_{5,2}$. □

**Lemme 7** ([BCK$^+$14]). *Soit une fonction $f$ calculée par un circuit de profondeur $d$ et taille $s$ composée exclusivement de noeuds seuil, chacun ayant au plus $k$ entrées. Soit $p > k$ un nombre premier. Alors $f \in \mathsf{TP}(\mathbb{Z}_p, O(dpks4^d), O(dpks))$.*

*Démonstration.* On transforme par le lemme 5 le circuit $C$ de profondeur $d$ qui calcule $f$ en un circuit $C'$ de profondeur $2d$ qui utilise seulement des noeuds $[\![\cdots \neq c]\!]$. Le nombre de noeuds dans $C'$ est au plus $O(ks)$. On construit le circuit $C'$ de façon à ce qu'il soit étagé, ce qui multiplie le nombre de noeuds par un facteur de $2d$. Voyons la construction du programme transparent qui calcule $f$. Pour chaque noeud $n_i$ du circuit $C'$, on ajoute un registre auxiliaire $r_i$ qui va calculer la valeur de $n_i$. On calcule les valeurs des noeuds par niveau par récurrence.

Si $n_i$ est une entrée du circuit, alors il s'agit d'une constante ou d'une variable d'entrée. Dans le premier cas, on ajoute l'instruction $r_i \leftarrow r_i + c$ où $c \in \{0,1\} \subseteq \mathbb{Z}_p$. Dans le second cas, on ajoute l'instruction $r_i \leftarrow r_i + x_j$ où $x_j$ est le registre d'entrée approprié. Une suite d'instructions de cette sorte pour tous les noeuds d'entrée donne un programme $P_1$ qui calcule de façon transparente les valeurs des noeuds d'entrée du circuit $C'$.

Notre hypothèse de récurrence est que l'on peut construire un programme $P_l$ pour calculer

| $\sum_{i=1}^{k} b_i$ | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| $[\![\sum_{i=1}^{k} b_i \neq 0]\!]$ | 0 | 1 | 1 | 1 | 1 | 1 |
| $[\![\sum_{i=1}^{k} b_i \neq 1]\!]$ | 1 | 0 | 1 | 1 | 1 | 1 |
| $\sum_{j=0}^{1} [\![\sum_{i=1}^{k} b_i \neq j]\!]$ | 1 | 1 | 2 | 2 | 2 | 2 |
| $\left[\!\!\left[\sum_{j=0}^{1} [\![\sum_{i=1}^{k} b_i \neq j]\!] \neq 1\right]\!\!\right]$ | 0 | 0 | 1 | 1 | 1 | 1 |

Tableau 3.I : Extrait du calcul de $T_{5,2}$.

de façon transparente l'étage $l$ du circuit $C'$. On construit un programme $P_{l+1}$ qui calcule l'étage $l+1$. Si $n_i$ est un noeud au niveau $l+1$, alors on sait qu'il s'agit d'un noeud $[\![\cdots \neq c]\!]$ dont toutes les entrées sont au niveau $l$. Par le lemme 5, il existe des programmes $I_1^i$,$I_2^i$ et $I_3^i$ tels que $I_1^i$,$P_l$,$I_2^i$,$P_l^{-1}$, $I_3^i$ calcule la valeur de $n_i$ dans le registre $r_i$. On choisit ces programmes pour qu'ils utilisent des registres différents. Si $n_{i_1}$, ..., $n_{i_q}$ sont les noeuds à l'étage $l+1$, alors le programme

$$P_{l+1} = I_1^{i_1},\ldots,I_1^{i_q},P_l,I_2^{i_1},\ldots,I_2^{i_q},P_l^{-1},I_3^{i_1},\ldots,I_3^{i_q}$$

calcule de façon transparente les valeurs de noeuds à l'étage $l+1$. On obtient ainsi le programme $P_d$ qui calcule la valeur de $C'$.

On nomme $S_l$ la longueur de $P_l$, alors $S_l \leq 2S_{l-1} + 2cdks(k+p)$ où $ks$ est le nombre maximal de noeuds dans le circuit $C'$, $c(k+p)$ nous vient de la longueur nécessaire pour calculer chaque noeud $[\![\cdots \neq c]\!]$ par le lemme 5 et $c$ est une constante. On pose $a = 2cdks(k+p)$. On sait que $S_1 \leq ks \leq a$. On résout la récurrence pour obtenir $S_l \leq 2^l a - a \leq 2^l a$. La taille du programme $P_{2d}$, $S_{2d}$, appartient à $O(2^{2d} \times 2cdks(k+p)) = O(4^d dksp)$ puisque $k < p$. Pour chacun des $O(dks)$ noeuds, on utilise $O(p)$ registres, ce qui nous donne un nombre total de registres dans l'ordre de $O(dpks)$. □

On peut adapter le théorème 3.8 de Allender et Koucký [AK10] pour démontrer le lemme suivant. Ce lemme nous permet de remplacer chaque noeud seuil de degré entrant $n$ par un petit circuit de profondeur constante avec un degré entrant maximal borné par $n^\varepsilon$ pour $0 < \varepsilon < 1$. Cela permet de réduire autant que l'on souhaite le degré entrant maximal de notre circuit.

**Lemme 8.** *Pour tout $0 < \varepsilon < 1$, un noeud $T_{k,s}$ se réduit à un circuit uniforme de profondeur $O(1/\varepsilon)$, de taille $O(n)$ et ayant un nombre d'arcs dans $O(n^{1+\varepsilon})$ et ayant un degré entrant maximal borné par $k^\varepsilon$.*

L'idée de la preuve est de construire un petit circuit qui fait la somme des $n$ entrées de

1 bit et place le résultat dans un nombre de taille $\log n$ bit. Ensuite, il suffit de comparer ce nombre avec le seuil $s$ souhaité. Il s'agit de la même construction que pour le théorème 3.8 de [AK10], mais on compare le nombre final avec $s$ au lieu de $n/2$ où $n/2$ vient du calcul de la fonction majorité dans la version originale.

**Théorème 3** ([BCK+14]). *Pour n'importe quelle suite de nombres premiers $(p_n)_{n\in\mathbb{N}}$ de taille polynomiale en n,*

$$\mathsf{TC}^1 \subseteq \bigcup_{p_n} \mathsf{TP}(\mathbb{Z}_{p_n}).$$

*Démonstration.* Soit $(p_n)_{n\in\mathbb{N}}$ une suite de nombres premiers de taille polynomiale en $n$. Soit $(C_n)_{n\in\mathbb{N}}$ une famille de circuits appartenant à $\mathsf{TC}^1$. On construit une famille $(C'_n)_{n\in\mathbb{N}}$ de circuits tels que le degré entrant maximal de $C'_n$ est plus petit que $p_n$ avec le lemme 8. On transforme la famille $(C'_n)_{n\in\mathbb{N}}$ en famille de programmes transparents grâce au lemme 7. $\square$

### 3.4.2 Simulation de programmes transparents

Pour faire la simulation d'un programme transparent, la machine de Turing doit être en mesure d'accomplir les opérations de base sur l'anneau du programme. On définit la notion de suite d'anneau uniforme.

**Définition 14.** *Une fonction $h : R \to \mathbb{B}^*$ est un encodage compact de l'anneau R si h est une bijection entre R et les $|R|$ premiers mots de longueur $l = \lceil \log |R| \rceil$ selon l'ordre lexicographique.*

**Définition 15.** *Une suite d'anneaux $(R_n)_{n=1}^{\infty}$ est uniforme selon l'espace logarithmique s'il existe une suite $(h_n)_{n=1}^{\infty}$ d'encodage compact de $(R_n)_{n=1}^{\infty}$ et des machine de Turing qui utilise un espace logarithmique $M_{op}$, $M_c$ et $M_t$ tel que*

1. *la machine $M_{op}$ sur entrée $(1^n, h_n(u) \circ h_n(v))$ retourne $h_n(u \circ v)$ sur son ruban de travail où $u, v \in R$ et $\circ \in \{+, \times\}$ ;*
2. *la machine $M_c$ sur entrée $1^n$ retourne les trois constantes $h_n(-1)$, $h_n(0)$ et $h_n(1)$ ;*

*3. la machine $M_t$ sur entrée $1^n$ retourne $|R_n|$, la taille de l'anneau.*

Le lemme suivant présente la simulation d'un programme $P \in \mathsf{TP}(R_n, O(n^c), O(n^c))$ par une machine de Turing qui utilise un espace de travail logarithmique et une espace auxiliaire polynomiale.

**Lemme 9** ([BCK$^+$14])**.** *Pour n'importe quelle suite d'anneaux uniforme en espace logarithmique $(R_n)_n$, il y a une machine en espace catalytique logarithmique $M$ qui sur entrée $(P,x)$ retourne $f(x)$, où $P$ is un programme transparent qui utilise les registres $r_1, r_2, \ldots, r_m$ sur l'anneau $R_{|x|}$ pour calculer $f(x)$ sur $r_1$. De plus, la machine $M$ utilise $m^2 \cdot \lceil log|R_{|x|}| \rceil$ bits sur l'espace auxiliaire et $O(\log(P,x))$ bits sur l'espace de travail.*

*Démonstration.* Posons $n = |x|$. La machine $M$ se sert de son ruban auxiliaire pour stocker les registres de $P$ et simuler $P$. On divise ce ruban en bloc de taille $b = \lceil |R_n| \rceil$. Ces blocs pourront servir à stocker des registres de $R_n$. Puisque la suite d'anneaux est uniforme en espace logarithmique, on pourra effectuer les différentes opérations de base sur notre anneau avec seulement un espace de travail logarithmique.

Par la définition de $P$, son exécution ajoute la valeur de $f(x)$ dans le registre $r_1$. Après son exécution, on a $r_1 = \tau_1 + f(x)$. Pour extraire $f(x)$ du registre, la machine va copier sur le ruban de travail puis effacer le contenu du bloc qui sert à stocker $r_1$. De cette façon, on pourra facilement avoir la valeur de $f(x)$ après l'exécution de $P$ puisque $\tau_1$ sera 0. Après avoir noté la valeur de $f(x)$, la machine simule $P^{-1}$ pour rétablir le contenu de tous les registres. Il reste à rétablir le contenu du bloc pour le registre $r_1$ pour terminer l'exécution de la machine. Voyons maintenant comment la machine stocke les différents registres dans le ruban auxiliaire.

Considérons le cas où la taille de l'anneau $|R_n|$ est une puissance de 2. Cela simplifie la simulation puisque la taille des blocs $b$ est exactement $\log |R_n|$. Par conséquent, toutes les valeurs possibles des blocs $b$ sont des encodages d'un élément de $|R_n|$. Il suffit à la machine d'utiliser les $m$ premiers blocs, un bloc par registre, pour simuler $P$.

Le deuxième cas est légèrement plus difficile. En effet, les blocs ne sont plus garantis d'être un encodage d'une valeur de $R_n$. De plus, puisque notre encodage est compact, on sait qu'un bloc invalide commence par un 1. Pour représenter ces registres, la machine $M$ divise son ruban auxiliaire en $m$ groupes de $m$ blocs de $b$ bits. On divise ce cas en deux possibilités :

A) il y a un groupe qui contient exclusivement des blocs qui ne sont pas des encodages de $R_n$ ;

B) chaque groupe possède au moins un bloc qui représente un élément de $R_n$.

Dans le cas A, on sait qu'un des groupes contient $m$ blocs invalides. La machine utilise ce groupe pour faire la simulation de $P$. Elle remplace le premier bit de chaque bloc de 1 vers 0, ce qui les rend des encodages d'élément de $R_n$. La machine fait l'opération inverse après la simulation de $P^{-1}$ pour retrouver l'état initial du ruban auxiliaire. Dans le cas B, la machine utilise le premier registre qui représente un élément de $R_n$ dans le $i^e$ groupe pour le $i^e$ registre de $P$. Puisque les registres de $P$ contiennent toujours des valeurs de $R_n$ et la machine ne change pas les valeurs des autres blocs, elle peut toujours retrouver ceux utilisés pour les registres lors du calcul.

La machine utilise bien $m^2 \cdot \lceil \log |R_{|x|}| \rceil$ bits sur son ruban auxiliaire une quantité logarithmique d'espace sur son ruban de travail. □

**Corollaire 3** ([BCK⁺14])**.** *Soit $(P_n)_n$ une suite de programme uniforme en espace logarithmique sur une suite d'anneau $(R_n)_n$ constructible en espace logarithmique. Le programme $P_n$ calcule la fonction $f_n$ dans le registre $r_1$. Alors la famille de fonctions $(f_n)_n$ est calculable en espace catalytique.*

On remarque que les constructions de la sous-section 3.4.1 sont toutes uniformes en espace logarithmique.

**Théorème 4.** *La classe $\mathsf{TC}^1$ uniforme en espace logarithmique est incluse dans $\mathsf{CL}$.*

Puisque que l'on ne sait pas si $\mathsf{TC}^1 \subseteq \mathsf{L}$, alors ce théorème semble être une indication que l'utilisation d'espace auxiliaire peut augmenter la puissance de calcul d'une machine de Turing.

### 3.5 Potechin

Calculer de façon catalytique, c'est calculer plusieurs fois la même fonction sur la même entrée. On le voit particulièrement bien lorsque l'on s'intéresse au programme de branchement catalytique. Dans ce mémoire, seul le cas où l'on calcule une fonction un nombre exponentiel de fois a été étudié. Par [GKM15], on sait qu'une machine de Turing catalytique qui calcule $f \in \mathsf{CL}$ est équivalente à une famille de programmes de branchement $2^{poly(n)}$-multicalcul qui calculent $f^{\|2^{poly(n)}}$. Potechin [Pot16] lui s'intéresse à la taille de $\mathsf{TAILLE}_{pb}(f^{\|k})$ lorsque $k \to \infty$.

**Définition 16** ([Pot16])**.** $\mathsf{TAILLE}_{avg}(f) = \lim_{k\to\infty} \frac{\mathsf{TAILLE}_{pb}(f^{\|k})}{k}$

**Proposition 4** ([Pot16])**.** *Pour toutes fonctions $f$, $\mathsf{TAILLE}_{avg}(f)$ est bien défini et est égal à* $\inf\left\{\frac{\mathsf{TAILLE}_{pb}(f^{\|k})}{k} \mid k \geq 1\right\}$.

*Démonstration.* On note que pour tout $m_1, m_2 \geq 1$, $\mathsf{TAILLE}_{pb}(f^{\|m_1+m_2}) \leq \mathsf{TAILLE}_{pb}(f^{\|m_1}) + \mathsf{TAILLE}_{pb}(f^{\|m_2})$. Étant donné un programme de branchement calculant $f$ $m_1$ fois et un autre calculant $f$ $m_2$ fois, il suffit de considérer l'union de ces deux programmes pour en obtenir un qui calcule la fonction $f$ $m_1+m_2$ fois. Par conséquent, pour tout $m_0 \geq 1$, $k \geq 1$ et $0 \leq r < m_0$, $\mathsf{TAILLE}_{pb}(f^{\|km_0+r}) \leq k\mathsf{TAILLE}_{pb}(f^{\|m_0}) + \mathsf{TAILLE}_{pb}(f^{\|r})$. Cela implique que $\lim_{m\to\infty} \frac{\mathsf{TAILLE}_{pb}(f^{\|m})}{m} \leq \frac{\mathsf{TAILLE}_{pb}(f^{\|m_0})}{m_0}$ et prouve la proposition. □

**Lemme 10.** *S'il existe un nombre naturel $k$ et un programme $P$ qui calcule $f^{\|k}$ avec $\mathsf{TAILLE}_{pb}(P) \leq ck$, alors $\mathsf{TAILLE}_{avg}(f) \leq c$.*

*Démonstration.* On sait de la preuve précédente que pour tout $m_0 > 1$, $\lim_{m\to\infty} \frac{\mathsf{TAILLE}_{pb}(f^{\|m})}{m} \leq \frac{\mathsf{TAILLE}_{pb}(f^{\|m_0})}{m_0}$. Soit $k$ et $P$ tel que définit dans l'énoncé du lemme. On choisit $m_0 = k$ et on

obtient directement que

$$\mathsf{TAILLE}_{avg}(f) \leq \frac{\mathsf{TAILLE}_{pb}(f^{\|k})}{k} \leq \frac{\mathsf{TAILLE}_{pb}(P)}{k} \leq c.$$

□

Le théorème 5 raffine l'analyse du théorème 3.1 de Potechin et améliore le coefficient de linéarité de 32 à 6.

**Théorème 5.** *Pour toute fonction* $f : \mathbb{B}^n \to \mathbb{B}$*,* $\mathsf{TAILLE}_{avg}(f) \leq 6n$*.*

*Démonstration.* L'objectif est de construire un programme de branchement $k$-catalytique pour $k = 2^{2^n}$ de taille $6nk$. Le programme de branchement de base qui sera utilisé est celui qui calcule toutes les fonctions $f : \mathbb{B}^n \to \mathbb{B}$ à la fois. Il sera répété 6 fois. Il possède un noeud $\nabla_f$ pour chaque fonction $f$, des noeuds $\top_1, \ldots, \top_{k/2}$ et $\bot_1, \ldots, \bot_{k/2}$. Si $f(x) = 1$ (*resp.* $f(x) = 0$), alors l'exécution du programme commençant à $\nabla_f$ se terminera à $\top_i$ (*resp.* $\bot_i$) pour un $i$ quelconque. Voyons la construction de ce programme. Il est constitué de $n+1$ couches qui contiennent $k$ noeuds chacune. La couche $j$ représente les fonctions $f : \mathbb{B}^{n-j} \to \mathbb{B}$. Chacune de ces fonctions sera représentée $2^{2^n - 2^{n-j}}$ fois. Lorsque $j = 0$, on a bien chacune des fonctions de $n$ bits qui sont présentes une fois, il s'agit de nos noeuds de départ $\nabla_f$. Lorsque $j = n$, on a deux fonctions qui ne prennent pas de paramètre, l'une retourne toujours vrai et l'autre toujours faux. On retrouve nos noeuds $\top_i$ et $\bot_i$ avec $i \in \{1, \ldots, k/2\}$. Sur la couche $j$, tous les noeuds auront comme étiquette $x_j$. D'une fonction

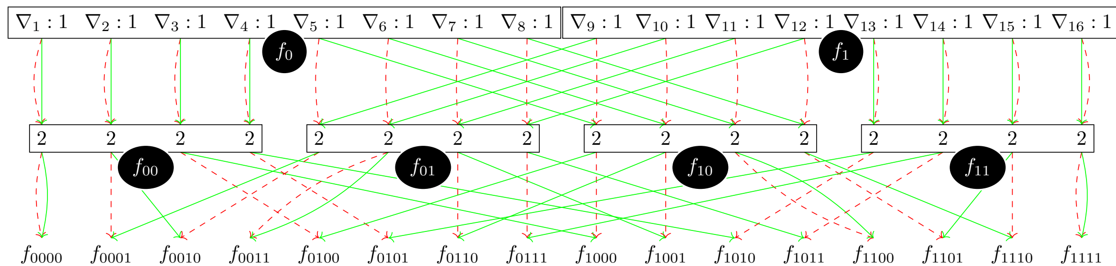


Figure 3.4 : Le programme de branchement $P_2$ de la preuve de Potechin pour $n = 2$. Les arcs pointillés (*resp.* pleins) sont étiquetés par 0 (*resp.* 1).

$f$ sur l'étage $j$, on dessine un arc étiqueté $l \in \mathbb{B}$ vers une fonction $f'$ de la couche $j+1$ telle que $f(b_1,\ldots,b_{n-j-1},l) = f'(b_1,\ldots,b_{n-j-1})$. Le choix des noeuds $f'$ est arbitraire avec la seule condition supplémentaire que chaque noeud ait seulement un arc entrant étiqueté 0 et un arc entrant étiqueté 1. On note un tel programme $P_1$. Ce programme est problématique puisque l'on ne possède pas de certitude quant aux noeuds finaux lorsque l'on commence une exécution. On verra plus tard comment régler ce problème.

Le deuxième programme aura deux comportements. Sur la première moitié de ses noeuds de départ $\nabla_1,\ldots,\nabla_{k/2}$, il identifiera toutes les fonctions de $n$ bits qui retournent 0 sur entrée $x$. En effet, chacun des $k$ noeuds terminaux de ce programme représente une fonction de $n$ bits. On note $t_f$ chacun de ces noeuds. On aura la condition supplémentaire que si $f(x) = 0$, alors il existe un $i$ tel que le programme va de $\nabla_i$ vers $t_f$ sur entrée $x$. Simultanément, la deuxième moitié de ses noeuds de départ $\nabla_{\frac{k}{2}+1},\ldots,\nabla_k$ identifiera les fonctions $f$ qui retournent 1 sur entrée $x$ avec une condition similaire. Pour construire un tel programme, il suffit d'inverser le programme $P_1$. Conceptuellement, celui-ci cherchait à réduire les fonctions jusqu'à arriver aux fonctions constantes sur 0 bit, celui-là cherche à partir de ces 2 fonctions à arriver à classer chaque fonction selon son résultat sur $x$. On note un tel programme $P_2$. La figure 3.4 propose un schéma du programme $P_2$ lorsque $n = 2$.

Soit $g : \mathbb{B}^n \to \mathbb{B}$ la fonction que l'on veut calculer. On note $h_f$ la fonction qui vérifie si $g$ et $f$ ont la même image sur $x$ :

$$\begin{aligned} h_f : \mathbb{B}^n &\to \mathbb{B} \\ x &\mapsto g(x) = f(x). \end{aligned}$$

Il reste à faire un programme pour $g$, ce qui est notre but. L'idée est d'exécuter $P_2$ puis ensuite $P_1$. Pour chaque noeud $t_f$, on le fusionne avec le noeud $\nabla_{h_f}$. Les noeuds finaux sont ceux de $P_1$. Voyons le comportement de cette construction lorsque $g(x) = 1$ et l'on commence par un noeud $\nabla_i$ avec $i \in \{1,2,\ldots,\frac{k}{2}\}$. Dans ce cas, le programme $P_2$ nous donne des chemins de $\{\nabla_i |\ i \in \{1\ldots\frac{k}{2}\}\}$ vers $\{t_f |\ f(x) = 1\}$. Puisque $g(x) = 1$ et $f(x) = 1$, alors $h_f(x) = 1$. La

suite du programme nous donne un chemin de $\{t_f |\ f(x) = 1\} = \{\nabla_{h_f} |\ f(x) = 1\}$ vers $\{\top_i |\ i \in \{1, 2, \ldots, \frac{k}{2}\}$. On retrouve les trois autres cas dans le tableau 3.*II*. Il y a deux problèmes avec notre construction jusqu'à maintenant :

- lorsque l'on utilise la deuxième moitié des noeuds de départ, la réponse est inversée ;
- un chemin commençant à $\nabla_i$ ne finit pas nécessairement à $\top_i$ ou $\bot_i$.

Pour régler ces deux problèmes, il suffit d'ajouter deux copies inversées de notre programme actuel à la fin. La première (*resp.* deuxième) de ces copies sera reliée aux noeuds $\top_i$ (*resp.* $\bot_i$). Les copies vont effectivement retrouver le noeud initial tout en gardant un bit d'information supplémentaire, soit si l'hypothèse du noeud de départ était exacte. Les noeuds finaux dans la première copie seront $\top_1, \ldots, \top_{\frac{k}{2}}, \bot_{\frac{k}{2}+1}, \ldots, \bot_k$ et inversement pour la deuxième copie. Ceci complète notre programme pour reconnaître $g$ $k$ fois. On peut voir que $\mathsf{TAILLE}_{pb}(P_1) = \mathsf{TAILLE}_{pb}(P_2) = nk$. Avec les deux copies, on compte trois fois $P_1$ et $P_2$ pour un total de $6nk$. Il reste à appliquer le lemme 10 pour obtenir que $\forall f\ \mathsf{TAILLE}_{avg}(f) \leq 6n$. $\square$

Ce résultat met en évidence que l'ajout de mémoire catalytique permet de réduire la quantité de mémoire régulière pour calculer une fonction. On remarque que chaque étage correspond à une permutation selon la valeur de la variable interrogée à cet étage. Rappelons que les programmes de branchement utilisés par Barrington dans [Bar86] sont aussi formés par des permutations. Il serait très facile de transformer ses 5-PBP en programmes de branchement 2-catalytiques. Il suffirait d'utiliser le lemme 1 de [Bar86] pour changer la permutation d'acceptation vers $(13245)$.

| $g(x)$ | Noeud de départ | $h_{f(x)}$ | Ensemble d'arrivée |
|---|---|---|---|
| 0 | $\{\nabla_i \mid i \in \{1, 2, \ldots, \frac{k}{2}\}\}$ | 0 | $\{\bot_i \mid i \in \{1, 2, \ldots, \frac{k}{2}\}\}$ |
| 1 | $\{\nabla_i \mid i \in \{1, 2, \ldots, \frac{k}{2}\}\}$ | 1 | $\{\top_i \mid i \in \{1, 2, \ldots, \frac{k}{2}\}\}$ |
| 0 | $\{\nabla_i \mid i \in \{\frac{k}{2}+1, \ldots, k\}\}$ | 1 | $\{\top_i \mid i \in \{1, 2, \ldots, \frac{k}{2}\}\}$ |
| 1 | $\{\nabla_i \mid i \in \{\frac{k}{2}+1, \ldots, k\}\}$ | 0 | $\{\bot_i \mid i \in \{1, 2, \ldots, \frac{k}{2}\}\}$ |

Tableau 3.II : Valeurs des différents éléments du programme selon la valeur de $g$ et du noeud de départ.

# CHAPITRE 4

# PROGRAMME DE BRANCHEMENT AVEC MODÈLE

Dans ce chapitre, on développe un nouveau formalisme équivalent aux programmes de branchement avec oracle de [BM91], les programmes de branchement avec modèle. Lors de l'utilisation de programmes de branchement avec oracle, l'intuition est de s'imaginer que chaque sommet pose une certaine question à un oracle. L'oracle du programme de branchement est construit de façon à pouvoir répondre à chacune des questions possibles.

Avec les programmes de branchement avec modèle, les questions sont plutôt exprimées sous forme de fonction. Ainsi, le modèle d'un programme de branchement avec modèle contient toutes les fonctions que le programme de branchement peut utiliser. Plusieurs outils sont développés pour étudier ces modèles. Une question naturelle est un certain modèle permet-il à un programme de branchement de reconnaître un certain langage ? Pour répondre à cette question, on construit un langage qui indique quels mots du langage peuvent être reconnus par ce modèle. On considère également d'autres langages qui répondent à des questions similaires. Finalement, on étudiera la complexité de quelques-uns de ces nouveaux langages. De nouveaux résultats de $\mathsf{coNP}$-complétude sont un apport original de ce mémoire.

## 4.1 Définition des programmes de branchement avec modèles

**Définition 17.** *Soit $n$ un entier naturel. Une question sur $n$ bits $\mathsf{q}^n$ est un triplet $(b, \Sigma, f)$ où*

- $b \subseteq \{1, 2, \ldots, n\}$ *;*
- $\Sigma$ *est un ensemble fini ;*
- $f : \mathbb{B}^{|b|} \to \Sigma$ *est une fonction surjective.*

L'exécution d'une question $\mathsf{q}^n = (b, \Sigma, f)$ sur un mot $w \in \mathbb{B}^n$ (ou réponse de $\mathsf{q}^n$ sur $w$) est $\mathsf{q}^n(w) = f(w_{b_1}, w_{b_2}, \ldots, w_{b_{|b|}})$. De plus, le mot formé des $|b|$ paramètres de $f$ lors du calcul

de la question $\mathsf{q}^n$ sur le mot $w$ est appelé le sous-mot de $w$ induit par $\mathsf{q}^n$.

**Définition 18.** *Soit $n$ un entier naturel. Un modèle $\mathsf{M}_n$ de rang $n$ est un ensemble fini de questions sur $n$ bits.*

Une famille de modèles $\mathscr{M} = \{\mathsf{M}_i\}_{i>0}$ est une suite infinie de modèles où $\mathsf{M}_i$ est de rang $i$. Elle est de taille $q$ si $|\mathsf{M}_i| \in O(q(i))$. Le calcul de l'empreinte de $w$, c'est-à-dire l'ensemble des réponses aux questions sur $w$, est défini sur un modèle $\mathsf{M}_n$ et une famille de modèle $\mathscr{M}$ des façons suivantes :

$$\mathsf{M}_{|w|}(w) = \{(\mathsf{q}^{|w|}, \mathsf{q}^{|w|}(w)) |\ \mathsf{q}^{|w|} \in \mathsf{M}_{|w|}\}; \tag{4.1}$$

$$\mathscr{M}(w) = \mathsf{M}_{|w|}(w). \tag{4.2}$$

**Définition 19.** *Soit $\mathsf{M}_n$ un modèle. Un $\mathsf{M}_n$-$\mathsf{BP}$ est un graphe acyclique. Les noeuds sans arcs sortants sont définis de la même façon que dans les programmes de branchement. Les noeuds internes sont étiquetés par une question du modèle $\mathsf{M}_n$. Soit un noeud $v$ étiqueté de la question $\mathsf{q}^n = (b, \Sigma, f)$, alors chaque élément de $\Sigma$ étiquette un et un seul arc sortant de $v$.*

On peut voir un exemple très simple de l'utilisation de ces programmes dans le prochain lemme. Avant cela, il faut définir un premier modèle :

$$\mathsf{REG}_n = \{\underbrace{(\{i\}, \mathbb{B}, f)}_{\mathsf{q}_i^n} | 1 \leq i \leq n\}$$

où la fonction $f$ est la fonction identité sur un bit. Les questions de ce modèle sont très simples. Elles retournent exactement le contenu d'un des bits du mot reçu.

**Lemme 11.** *Soit $P$ un programme de branchement sur $n$ bits. Alors, il existe $P'$ un $\mathsf{REG}_n$-$\mathsf{BP}$ reconnaissant le même langage et possédant la même taille.*

*Démonstration.* Soit $P$ un programme de branchement sur $n$ bits. On construit un $\mathsf{REG}_n$-$\mathsf{BP}$ $P'$ en remplaçant les étiquettes de $P$ par des questions dans $\mathsf{REG}_n$. Les étiquettes $i$ sont remplacées par la question $\mathsf{q}_i^n$. Le programme $P'$ possède la même taille que $P$. Il reste à montrer

que les deux programmes reconnaissent le même langage. Soit $w$ un mot et considérons son chemin $(x_1, x_2, \ldots, x_m)$ dans $P$ où $x_i = (p_i \in \{1, 2, \ldots, n\}, b_i \in \mathbb{B})$, $p_i$ représente l'étiquette du noeud courant et $b_i$ l'étiquette de l'arc suivi à la suite de ce noeud. Alors il existe un chemin $(x'_1, x'_2, \ldots, x'_m)$ dans $P'$ où $x'_i = (\mathsf{q}^n_{p_i} \in \mathsf{REG}_n, b_i \in \mathbb{B})$. Ce chemin pose des questions équivalentes et obtient les mêmes réponses que le chemin dans $P$. Le programme $P'$ va donc accepter $w$ si et seulement si $P$ l'accepte. □

## 4.2 Langages et outils basés sur les empreintes

Définissons des classes d'équivalence à partir d'une famille de modèles $\mathscr{M}$ :

$$[x]_{\mathscr{M}} = \{w \in \mathbb{B}^{|x|} \mid \mathscr{M}(x) = \mathscr{M}(w)\}. \tag{4.3}$$

Chaque classe d'équivalence contient tous les mots qu'aucune question du modèle ne peut distinguer de $x$.

**Définition 20.** *Soit $\mathscr{M}$ une famille de modèles et $\mathscr{L}$ un langage, alors on peut définir les langages suivants :*

$$\begin{aligned}
\mathscr{M}\text{-}\mathscr{L}\text{-positif} &= \{w \in \mathscr{L} \mid \forall w' \in [w]_{\mathscr{M}}\ (w' \in \mathscr{L})\}; \\
\mathscr{M}\text{-}\mathscr{L}\text{-négatif} &= \{w \notin \mathscr{L} \mid \forall w' \in [w]_{\mathscr{M}}\ (w' \notin \mathscr{L})\}; \\
\mathscr{M}\text{-}\mathscr{L}\text{-confus} &= \{w \in \mathbb{B}^* \mid \exists a, b \in [w]_{\mathscr{M}}\ (a \in \mathscr{L} \wedge b \notin \mathscr{L})\}; \\
\mathscr{M}\text{-}\mathscr{L}\text{-fiable} &= \mathscr{M}\text{-}\mathscr{L}\text{-positif} \cup \mathscr{M}\text{-}\mathscr{L}\text{-négatif}.
\end{aligned}$$

Considérons une famille de modèles $\mathscr{M}$ et un langage $\mathscr{L}$. Pour chaque mot $w \in \Sigma^*$, la classe d'équivalence $[x]_{\mathscr{M}}$ peut être :

1. un sous-ensemble de $\mathscr{L}$ ;
2. un sous-ensemble de $\overline{\mathscr{L}}$ ;
3. un ensemble qui contient au moins un mot de $\mathscr{L}$ et un mot de $\overline{\mathscr{L}}$.

Le langage $\mathscr{M}$-$\mathscr{L}$-positif regroupe tous les mots qui sont dans cette première catégorie. Du point de vue d'un $\mathscr{M}$-BP qui veut reconnaître $\mathscr{L}$, il doit accepter seulement les mots qui sont dans $\mathscr{M}$-$\mathscr{L}$-positif. Ce sont les seuls mots dont l'empreinte garantie qu'ils sont dans $\mathscr{L}$. Le langage $\mathscr{M}$-$\mathscr{L}$-négatif regroupe les mots qu'un tel programme peut exclure avec certitude, car l'ensemble des mots de leur classe d'équivalence n'appartiennent pas à $\mathscr{L}$. Dans le cas du langage $\mathscr{M}$-$\mathscr{L}$-confus, il s'agit des mots dont la classe d'équivalence contient à la fois un mot qui appartient à $\mathscr{L}$ et un qui n'appartient pas à $\mathscr{L}$. Puisque toutes les questions du modèle donnent les mêmes réponses pour ces deux mots, cela signifie qu'il est impossible pour un programme de branchement utilisant cette famille de modèles de déterminer si ces mots appartiennent ou non à $\mathscr{L}$.

Les deux propositions suivantes présentent quelques propriétés des langages que l'on a définis précédemment.

**Proposition 5.** *Soit $\mathscr{M}$ une famille de modèles et $\mathscr{L}$ un langage, alors les 5 énoncés suivants sont équivalents :*

- *il existe une famille de $\mathscr{M}$-BP qui reconnaît $\mathscr{L}$ ;*
- *$\mathscr{L} = \mathscr{M}$-$\mathscr{L}$-positif;*
- *$\overline{\mathscr{L}} = \mathscr{M}$-$\mathscr{L}$-négatif;*
- *$\mathscr{M}$-$\mathscr{L}$-confus $= \emptyset$;*
- *$\mathscr{M}$-$\mathscr{L}$-fiable $= \Sigma^*$.*

**Proposition 6.** *Soit $\mathscr{M}_1$ et $\mathscr{M}_2$ deux familles de modèles et $\mathscr{L}$ un langage, alors*

$$\mathscr{M}_1\text{-}\mathscr{L}\text{-}positif \cup \mathscr{M}_2\text{-}\mathscr{L}\text{-}positif \subseteq (\mathscr{M}_1 \cup \mathscr{M}_2)\text{-}\mathscr{L}\text{-}positif;$$
$$\mathscr{M}_1\text{-}\mathscr{L}\text{-}n\acute{e}gatif \cup \mathscr{M}_2\text{-}\mathscr{L}\text{-}n\acute{e}gatif \subseteq (\mathscr{M}_1 \cup \mathscr{M}_2)\text{-}\mathscr{L}\text{-}n\acute{e}gatif;$$
$$\mathscr{M}_1\text{-}\mathscr{L}\text{-}confus \cup \mathscr{M}_2\text{-}\mathscr{L}\text{-}confus \supseteq (\mathscr{M}_1 \cup \mathscr{M}_2)\text{-}\mathscr{L}\text{-}confus$$

*où l'union de deux familles de modèles* $\mathscr{M}_1 = (\mathsf{M}_i^1)_{i>0}$ *et* $\mathscr{M}_2 = (\mathsf{M}_i^2)_{i>0}$ *est la famille* $(\mathsf{M}_i^1 \cup \mathsf{M}_i^2)_{i>0}$.

Les langages positif, négatif et confus sont mutuellement exclusifs. Pour trouver le langage auquel appartient un mot, il faut étudier sa classe d'équivalence. Rappelons que le sous-mot induit de $w$ par $\mathsf{q}^n = (b, \Sigma, f)$ est le mot formé des $|b|$ paramètres de $f$ lors du calcul de la question $\mathsf{q}^n$ sur le mot $w$.

**Définition 21.** *Soit* $\mathscr{M}$ *une famille de modèles et x un mot de longueur n. Une question* $\mathsf{q}^n = (b, \Sigma, f)$ *appartenant à* $\mathscr{M}$ *est confuse par* $[x]_{\mathscr{M}}$ *si*

$$|\{s \mid \exists w \in [x]_{\mathscr{M}} \text{ tel que le sous-mot induit de } w \text{ par } \mathsf{q}^n \text{ est } s\}| > 1.$$

Autrement dit, une question est confuse par $[x]_{\mathscr{M}}$ si tous les mots de la classe n'ont pas le même sous-mot induit par cette question. On note que toutes les questions de la famille de modèles $\mathsf{REG}$ ne peuvent pas être confuses puisque leur fonction est bijective. Lorsqu'une question est bijective, le changement du sous-mot induit entraînerait forcément un changement dans l'empreinte du mot. Or, les mots dans les classes d'équivalence ont tous la même empreinte.

**Définition 22.** *Soit* $\mathscr{M}$ *une famille de modèles et x un mot de longueur n. Alors on dit que la i*$^{e}$ *lettre de la classe d'équivalence* $[x]_{\mathscr{M}}$ *où* $0 < i \leq n$ *est fixée si la condition suivante est vraie :*

$$|\{w_i | \ w_1 w_2 \dots w_n \in [x]_{\mathscr{M}}\}| = 1.$$

## 4.3 Définition des modèles MIN, MAX et MINMAX

Lors de la définition d'un modèle, il est d'usage de le faire en pensant à problème particulier. Le prochain modèle que l'on va définir est spécialisé pour le problème GEN et utilise la représentation des exemplaires de GEN sous forme de tableau. Soit $n$ un entier naturel qui représente le nombre de ligne et de colonne dans notre instance de GEN, $p = \lceil \log(n+1) \rceil$

le nombre de bits nécessaires pour une case et $m = pn^2$ le nombre total de bits. Nous allons définir le modèle $\mathsf{MIN}_m$ en trois parties. Les questions de notre modèle impliquant chacune deux paramètres identifiant une case dans le tableau. On commence par les questions où les deux paramètres sont différents. Ces questions considèrent les quatre cases $i \times i, i \times j, j \times i$ et $j \times j$ puis retournent la plus petite valeur de ces cases autre que $i$ et $j$ ou 1 si une telle valeur n'existe pas. On utilise la fonction $\mathrm{pos}(i, j, n)$ qui retourne l'ensemble des positions des bits des quatre cases mentionnées précédemment. On a les questions

$$\mathsf{MIN}^1_m = \{\underbrace{(\mathrm{pos}(i,j,n), \{1,2,\ldots,n\} \setminus \{i,j\}, f^1_{i,j})}_{\mathsf{q}^m_{i,j}} \mid i \neq j\}_{1 \leq i \neq j \leq n}$$

$$f^1_{i,j} : \mathbb{B}^{p^4} \to \{1,2,\ldots,n\} \setminus \{i,j\}$$

$$a,b,c,d \mapsto \begin{cases} 1 & si\ \{a,b,c,d\} \setminus \{i,j\} = \emptyset \\ \min(\{a,b,c,d\} \setminus \{i,j\}) & sinon \end{cases}$$

où $a, b, c$ et $d$ sont les éléments de l'exemplaire de $\mathsf{GEN}$ représentés par les 4 suites de $p$ bits formant l'argument de $f^i_{i,j}$. On considère maintenant le cas où les deux paramètres sont égaux en excluant le cas $i = j = 1$ pour l'instant. Contrairement au cas précédent, il suffit de regarder une seule case du tableau pour répondre à ces questions.

$$\mathsf{MIN}^2_m = \{\underbrace{(\mathrm{pos}(i,i,n), \{1,2,\ldots,n\} \setminus \{i\}, f^2_i)}_{\mathsf{q}^m_{i,i}}\}_{1 < i \leq n}$$

$$f^2_i : \mathbb{B}^p \to \{1,2,\ldots,n\} \setminus \{i\}$$

$$a \mapsto \begin{cases} 1 & si\ a = i \\ a & sinon \end{cases}$$

La question $1 \times 1$ demande une attention particulière, car elle peut répondre n'importe

quelle valeur, au contraire des questions $\mathsf{q}^n_{i,j}$ qui ne peuvent pas retourner les valeurs $i$ et $j$.

$$\mathsf{MIN}^3_m = \{\underbrace{(\mathrm{pos}(1,1,n),\{1,2,\ldots,n\},f^3)}_{\mathsf{q}^m_{1,1}}\}$$

$$f^3 : \mathbb{B}^p \to \{1,2,\ldots,n\}$$
$$a \mapsto a$$

Pour finir, on rassemble les trois ensembles pour avoir le modèle souhaité :

$$\mathsf{MIN}_m = \mathsf{MIN}^1_m \cup \mathsf{MIN}^2_m \cup \mathsf{MIN}^3_m.$$

On peut définir $\mathsf{MAX}_m$ de la même façon en remplaçant la fonction min par la fonction max. Notons que le nombre d'éléments de $\mathsf{MINMAX}_m$ en fonction de $m$ et de $n$ est polynomial. Le modèle $\mathsf{MINMAX}_m$ est l'union de ces deux modèles. On utilisera $Q_{i,j}$ comme nom pour les questions provenant du modèle $\mathsf{MAX}_m$ et $q_{i,j}$ pour celle de $\mathsf{MIN}_m$.

### 4.4 La complexité de MINMAX-GEN-négatif

Cette section contient l'analyse de la complexité de MINMAX-GEN-négatif. Je pense qu'une telle preuve aurait été plus ardue à exprimer avec les programmes de branchement avec oracle.

**Théorème 6.** MINMAX-GEN-*négatif est* coNP-*complet.*

*Démonstration.* Montrons que MINMAX-GEN-négatif appartient à coNP. À l'aide de l'approche avec certificat, il suffit de montrer qu'une machine peut reconnaître un contre-exemple en temps polynomial. Pour MINMAX-GEN-négatif, un contre-exemple du mot $v \in \overline{\mathsf{GEN}}$ est un mot $w \in \mathsf{GEN}$ tel que $w \in [v]_{\mathsf{MINMAX}}$. Une machine de Turing peut donc suivre les étapes suivantes pour valider que $v \in$ MINMAX-GEN-négatif :

1. vérifier que $v \in \overline{\mathsf{GEN}}$ ;

2. vérifier que $w \in \mathsf{GEN}$ ;

3. vérifier que $\mathsf{MINMAX}(v) = \mathsf{MINMAX}(w)$.

Ces trois étapes se calculent en temps polynomial. Une telle machine peut donc faire la vérification d'un certificat qui est un contre-exemple en temps polynomial. Il reste à montrer que MINMAX-GEN-négatif est coNP-ardu pour compléter la preuve de complétude. Pour ce faire, montrons que $\overline{\mathsf{SAT}} \leq_m^{\mathsf{P}}$ MINMAX-GEN-négatif.

Soit $\phi$ une formule booléenne non satisfaisable sur $n$ variables et donc une instance de $\overline{\mathsf{SAT}}$. Cette formule est aussi, par la définition de formule, un circuit booléen. Soit $\alpha$ le circuit de $\phi$ où l'on ajoute deux portes $\neg$ sur tous les arcs du circuit. Puisqu'on a ajouté des portes de négation sur tous les arcs, alors on a que chaque entrée d'une porte $\wedge$ et $\vee$ est unique, c'est-à-dire que si $\alpha_i = \wedge(\alpha_j, \alpha_k)$, alors $\alpha_j$ et $\alpha_k$ n'apparaissent pas comme entrées à d'autres portes du circuit que $\alpha_i$. On peut maintenant construire une instance $\psi$ de GEN tel que $\psi \in$ MINMAX-GEN-négatif si et seulement si $\phi \in \overline{\mathsf{SAT}}$. Pour assurer un bon comportement des questions de MINMAX, on s'assurera que le nombre d'éléments de $\psi$ soit une puissance de 2. Pour chaque sommet $\alpha_i$, on crée un élément pour ses deux valeurs possibles :

$$A = \bigcup_{\alpha_i} \{s_i, \overline{s_i}\}.$$

Pour chaque entrée dans un sommet de type négation, on ajoutera également un élément.

$$E = \{e_{ji} | \alpha_i = \neg(\alpha_j)\}$$

On associera à ces éléments les règles suivantes (voir figure 4.I) :

- si $\alpha_i = \wedge(\alpha_j, \alpha_k)$, alors $s_j \times s_k = s_k \times s_j = s_i$, $\overline{s_j} \times \overline{s_j} = \overline{s_i}$ et $\overline{s_k} \times \overline{s_k} = \overline{s_i}$ ;
- si $\alpha_i = \vee(\alpha_j, \alpha_k)$, alors $\overline{s_j} \times \overline{s_k} = \overline{s_k} \times \overline{s_j} = \overline{s_i}$, $s_j \times s_j = s_i$ et $s_k \times s_k = s_i$ ;
- si $\alpha_i = \neg(\alpha_j)$, alors $e_{ji} \times \overline{s_j} = \overline{s_j} \times e_{ji} = s_i$ et $e_{ji} \times s_j = s_j \times e_{ji} = \overline{s_i}$.

| | $\overline{s_j}$ | $s_j$ | $\overline{s_k}$ | $s_k$ |
|---|---|---|---|---|
| $\overline{s_j}$ | $\overline{s_i}$ | | | |
| $s_j$ | | | | $s_i$ |
| $\overline{s_k}$ | | | $\overline{s_i}$ | |
| $s_k$ | | $s_i$ | | |

Tableau 4.I : Les règles de la porte $\alpha_i = \wedge(\alpha_j, \alpha_k)$.

Un élément $s_i$ (resp. $\overline{s_i}$) est généré si et seulement si le sommet $\alpha_i$ du circuit vaut VRAI (resp. FAUX) lorsque les éléments initiaux sont initialisés correctement. Jusqu'à maintenant, la construction permet la simulation de $\phi$ et est tirée du théorème 6 de [Geh01]. On ajoutera deux autres parties à notre instance de GEN, une pour permettre à un MINMAX-BP d'essayer toutes les entrées possibles pour le circuit et une autre pour faire l'initialisation des éléments initiaux. Soit $n$ le nombre d'entrées pour notre circuit. On ajoutera des éléments pour chacune de ces $n$ entrées :

$$W_1 = \bigcup_{i \in \{1,2,\ldots,n\}} \{a_i, w_i, \overline{w_i}, b_i\};$$

$$W_2 = \bigcup_{i \in \{1,2,\ldots,n\}} \{d_i, d_i'\}.$$

Les éléments $w_i$ et $\overline{w_i}$ représentent la valeur de la $i^{\text{e}}$ variable de $\phi$ dans $\psi$. Les autres éléments de $W_1$ et $W_2$ servent à l'énumération des entrées de la formule $\phi$ dans $[\psi]$. Les détails du fonctionnement de cette énumération seront fournis plus loin. On associe à $W_1$ et $W_2$ les règles suivantes (voir figure 4.II) :

- $d_i \times d_i = d_i' \times d_i' = d_i \times d_i' = a_i$ ;
- $d_i' \times d_i = b_i$ ;
- $w_i \times w_i = s_i$ ;
- $\overline{w_i} \times \overline{w_i} = \overline{s_i}$.

Il reste maintenant à générer l'ensemble des éléments initiaux que l'on veut pour amorcer

| | $d_i$ | $d'_i$ | $a_i$ | $w_i$ | $\overline{w_i}$ | $b_i$ |
|---|---|---|---|---|---|---|
| $d_i$ | $a_i$ | $b_i$ | | | | |
| $d'_i$ | $a_i$ | $a_i$ | | | | |
| $a_i$ | | | | | | |
| $w_i$ | | | | $s_i$ | | |
| $\overline{w_i}$ | | | | | $\overline{s_i}$ | |
| $b_i$ | | | | | | |

Tableau 4.II : Initialisation pour les variables d'entrées $i$

le calcul, soit $N = E \cup W_2 = \{u_1, \ldots, u_k\}$. On ajoute $k+1$ nouveaux éléments générateurs

$$G = \{n_0, n_1, \ldots, n_k\}.$$

On associe aux éléments de $G$ pour $0 < i < k$ les règles

- $n_i \times n_i = n_{i+1}$,
- $n_0 \times n_i = u_i$.

On ajoute aussi les règles $n_0 \times n_0 = n_1$ et $n_0 \times n_k = u_k$. Lorsqu'on est capable de générer $n_0$, alors on peut générer $N$. Établissons un ordre total entre tous les éléments :

$$\underbrace{n_0 < \cdots < n_k}_{G} < \underbrace{d_1 < d'_1 < a_1 < w_1 < \overline{w_1} < b_1 < \ldots}_{W_1 \cup W_2} < E < \underbrace{\overline{s_1} < s_1 < \cdots < s_m}_{A},$$

où un ordre total arbitraire est choisi pour les éléments de $E$. Ce sera l'ordre utilisé par la fonction MINMAX pour comparer les éléments. On montre maintenant que seules les cases $d_i \times d'_i$ pour tout $i$ ne sont pas fixées par $[\psi]_{\mathsf{MINMAX}}$ et peuvent par conséquent changer la

| | $n_0$ | $n_1$ | $\cdots$ |
|---|---|---|---|
| $n_0$ | $n_1$ | $u_1$ | |
| $n_1$ | | $n_2$ | |
| $\vdots$ | | | |

Tableau 4.III : Génération de l'ensemble $N$

valeur de vérité de l'instance de GEN. Selon la définition de MINMAX, ce modèle possède 6 catégories de questions, le minimum et le maximum pour :

- la case $1 \times 1$ ($n_0$ selon l'ordre établi précédemment) ;
- les autres cases dans la diagonale ;
- le croisement des deux lignes et deux colonnes $i$ et $j$ lorsque $i \neq j$.

Considérons une case $d_i \times d'_i$ quelconque. Cette case peut être interrogée par 4 questions : $q_{d_i,d'_i}, Q_{d_i,d'_i}, q_{d'_i,d_i}, Q_{d'_i,d_i}$. Toutes ces questions considèrent seulement les cases $d_i \times d'_i$, $d'_i \times d_i$, $d_i \times d_i$, $d'_i \times d'_i$. La figure 4.II montre les valeurs dans ces cases. Les questions min répondront $a_i$ tandis que les max répondront $b_i$.

$$q_{d_i,d'_i} = q_{d'_i,d_i} = a_i \tag{4.4}$$

$$Q_{d_i,d'_i} = Q_{d'_i,d_i} = b_i \tag{4.5}$$

On cherche s'il existe des valeurs de $d_i \times d'_i$ tels que les équations (4.4) et (4.5) restent vrais. Ces valeurs doivent être entre $a_i$ et $b_i$ inclusivement selon notre ordre. Or, on sait que $w_i$ et $\neg w_i$ sont compris entre ces deux valeurs. La classe d'équivalence $[\psi]_{\mathsf{MINMAX}}$ contient des éléments dont la case $d_i \times d'_i$ prend la valeur $w_i$, $\neg w_i$ ou $b_i$ sans que les réponses aux questions (4.4) et (4.5) changent. La case n'est donc bien pas fixée. Or, cela permet l'initialisation la $i^e$ variable de l'expression $\phi$ dans notre construction lors de la génération de $w_i$ et $\neg w_i$.

Considérons maintenant chacune des autres cases de la construction et montrons qu'elles sont fixes dans $[\psi]_{\mathsf{MINMAX}}$ (voir définition 22). On remarque que dans la construction jusqu'à maintenant de $\psi$, pour une case $i \times j = a$, alors la valeur de $a$ est différente de $i$ et $j$. En effet, une case $i \times j = i$ ne serait pas très utile pour faire progresser le calcul puisqu'elle ne permet pas la génération d'une nouvelle valeur. Or, pour compléter le reste du tableau, on utilise des cases $i \times j = i$. De cette façon, on s'assure de ne pas changer la valeur de vérité de $\psi$. Une case qui n'a pas encore été définie dans la construction jusqu'à maintenant sera représentée

par le symbole $\tau$. Avant de voir les cas où l'on considère quatre cases à la fois, on peut déjà tirer un peu d'information en regardant seulement les cases de la diagonale de façon isolée.

1.

| | $i$ |
|---|---|
| $i$ | $a$ |

Lorsqu'une case de la diagonale est définie, alors on sait que tous les mots de $[\psi]_{\mathsf{MINMAX}}$ contiennent cette même valeur à cette position. Prenons un mot $w$ différent de $\psi$, mais appartenant à sa classe d'équivalence. Montrons qu'une case $i \times i$ définie est égale dans les deux instances de GEN. Par contradiction, si la valeur est plus petite (*resp.* grande) dans $w$, alors $w$ ne ferait pas partie de la classe d'équivalence puisque $q_{i,i}$ (*resp.* $Q_{i,i}$) changera de réponse.

2.

| | $i$ |
|---|---|
| $i$ | $\tau$ |

On affecte la valeur $i$ à une case $i \times i$ qui n'a pas encore été définie. On a alors qu'une telle case dans la classe d'équivalence peut prendre les valeurs $i$ et 1. Rappelons que les questions $q_{i,i}$ et $Q_{i,i}$ retournent 1 lorsque la case $i \times i$ vaut $i$ ou 1. La suite de l'analyse enlèvera la possibilité du 1 dans ces cases.

Pour les prochains cas, on considère les cases $i \times i$, $i \times j$, $j \times i$ et $j \times j$ pour $i \neq j$.

3.

| | $i$ | $j$ |
|---|---|---|
| $i$ | $\tau$ | $a$ |
| $j$ | $a$ | $\tau$ |

On trouve ce cas entre autres dans le tableau 4.I. Les deux cases sur la diagonale ne sont pas définies et les deux autres ont la même valeur. Dans un tel cas, les questions $q_{i,j}$ et $Q_{i,j}$ auront $a$ comme réponse. Dans un argument similaire au premier cas, on peut voir que n'importe quel changement des cases provoque un changement aux réponses

des questions concernées. De plus, les cases de la diagonale ne peuvent pas prendre la valeur 1 dans la classe d'équivalence puisque cela changerait la réponse $q_{i,j}$.

4.

| | $i$ | $j$ |
|---|---|---|
| $i$ | $a$ | $\tau$ |
| $j$ | $\tau$ | $\tau$ |

Pour fixer ce cas dans la classe d'équivalence, on met $i \times j = j \times i = a$ et $j \times j = j$ comme prévu.

5.

| | $i$ | $j$ |
|---|---|---|
| $i$ | $a$ | $\tau$ |
| $j$ | $\tau$ | $b$ |

Pour restreindre adéquatement la classe d'équivalence dans ce cas, on devra ajouter un nouvel élément à $\psi$ qui est plus grand que $a$ et $b$ sauf si ceux-ci génèrent le dernier élément. La meilleure place est donc entre l'avant-dernier élément et le dernier. On le note $\Upsilon$. On remarque que l'on n'a affecté la valeur 1 à aucune case. On affectera 1, le premier élément de $\psi$, à l'une et $\Upsilon$ à l'autre. Le choix importe peu puisque les deux possibilités feront partie de la classe d'équivalence. Voyons que ces modifications ne permettent pas à la classe d'équivalence $[\psi]_{\mathsf{MINMAX}}$ de générer le dernier élément de $\mathsf{GEN}$. La case avec 1 comme valeur force la question $q_{i,j}$ à répondre 1. Puisque l'on sait que chacune des deux cases sur la diagonale ne vaut pas 1 dans $\psi$, alors ces deux cases ne peut prendre la valeur 1 dans $[\psi]_{\mathsf{MINMAX}}$. Donc, une des deux autres cases doit prendre la valeur 1. Pour la question $Q_{i,j}$, on trouve deux réponses possibles selon $a$ et $b$, soit $\Upsilon$ et $s_m$ où $s_m$ est le dernier élément de $\psi$ qui symbolise la valeur vraie pour la réponse de la formule $\phi$. Dans la première possibilité, $\Upsilon$ force les réponses de la même façon que le 1. La seconde possibilité signifie que $a = s_m$ ou $b = s_m$. Puisqu'au moins $i \times i$ ou $j \times j$ génère $s_m$, on peut ignorer la case $i \times j$.

6.

|   | $i$ | $j$ |
|---|---|---|
| $i$ | $a$ | $c$ |
| $j$ | $\tau$ | $b$ |

Voici le cas que l'on trouve lors de la génération de l'ensemble $N$ (voir tableau 4.III). On remarque que $1 < a, b < c$. On peut alors utiliser un argument similaire au cas précédent pour conclure que mettre 1 dans la case restante va fixer ces quatre cases.

7. Il reste le cas trivial où aucune des cases n'est définie. En combinant les cas 2, 4 et 5, on obtient les valeurs qui stabilisent ces cases sans influencer la classe d'équivalence.

La construction de $\psi$ restreint la classe d'équivalence $[\psi]$ sauf pour les cases $d_i \times d'_i$. Il reste à montrer que $\phi \in \overline{\mathsf{SAT}}$ si et seulement si $\psi \in$ MINMAX-GEN-négatif. Considérons l'hypothèse que $\phi \notin \overline{\mathsf{SAT}}$. Il existe donc une assignation $v = v_1 v_2 \ldots v_n$ des $n$ variables telle que $\phi(v)$ est vrai. On choisit $\psi' \in [\psi]$ tel que $d_i \times d'_i = w_i$ si $v_i = 1$ et $\overline{w_i}$ autrement. On voit que $\psi' \in$ GEN et $\psi \notin$ MINMAX-GEN-négatif. En effet, la première partie de $\psi'$ génère l'assignation $v$ et la deuxième partie simule $v$ sur $\phi$. Voyons le cas où $\phi \in \overline{\mathsf{SAT}}$. Soit $\psi' \in [\psi]$. En regardant les cases $d_i \times d'_i$ de $\psi'$, on trouve l'assignation $v$ qui est utilisée par $\psi'$. Il est possible que $v$ soit incomplet. Ensuite, $\psi'$ simule $\phi$ sur $v$. Puisque $\phi$ n'accepte aucune solution, alors $\psi'$ ne générera pas son dernier élément et donc $\psi' \notin$ GEN. Ceci termine la preuve que MINMAX-GEN-négatif est NP-ardu et NP-complet. □

## 4.5 Complexité des autres classes à partir de MINMAX et GEN

Suite à la preuve de la section précédente, il est naturel de s'interroger à propos de la complexité des problèmes positif, confus et fiable. On peut montrer que le langage MINMAX-GEN-confus est NP-complet, MINMAX-GEN-fiable est coNP-complet et MINMAX-GEN-positif est coNP-complet. Pour ce dernier langage, on suit la même preuve que le langage négatif en effectuant les changements suivants :

- on fait une réduction de TAUT au lieu de $\overline{\mathsf{SAT}}$ ;

- on ajoute les règles $a_i \times a_i = b_i \times b_i = s_m$.

# CONCLUSION

Dans ce mémoire, nous avons mis en évidence des liens entre plusieurs variantes du problème de calculer un produit direct de fonctions, soit la production de masse, le multi-calcul et l'espace catalytique. Pour chacune de ces variantes, nous avons présenté un résultat significatif provenant de la littérature. En plus, nous avons introduit les programmes de branchement avec modèle.

Dans le cadre de la production de masse, le théorème d'Uhlig nous indique que l'on peut faire une économie de ressource lors du calcul d'une fonction un nombre de fois plus grand que polynomial. Nous avons vu la proposition de Girard et coll. qui montrent que peu de fonctions se composent de façon efficace dans le contexte du multi-calcul. Pour l'espace catalytique, nous avons présenté l'inclusion de $\mathsf{TC}^1$ dans l'espace catalytique logarithmique, un résultat de Burhman et coll. Dans le cadre du calcul catalytique, nous avons observé que la construction originale de Potechin donne lieu à une borne dont la constante multiplicative est meilleure que celle décrite dans l'article publié.

Les programmes de branchement avec modèle sont une nouvelle famille de programmes de branchement définie dans ce mémoire. Nous avons utilisé le langage paramétré MINMAX-GEN-négatif pour étudier la puissance du modèle MINMAX. Il s'agit d'un nouveau problème coNP-complet. D'autres modèles et leurs fonctions sous-jacentes pourront être étudiés lors de travaux futurs.

## BIBLIOGRAPHIE


[AK10] Eric Allender and Michal Koucký. Amplifying lower bounds by means of self-reducibility. *J. ACM*, 57(3) :14 :1–14 :36, 2010.

[Ash59] Robert L. Ashenhurst. The decomposition of switching functions. *Ann. Computation Lab.*, 29 :74–116, 1959.

[Bar86] David Barrington. Bounded-width polynomial-size branching programs recognize exactly those languages in $\mathsf{NC}^1$. pages 1–5, 01 1986.

[BCK+14] Harry Buhrman, Richard Cleve, Michal Koucký, Bruno Loff, and Florian Speelman. Computing with a full memory : catalytic space. pages 857–866, 2014.

[BM91] David Barrington and Pierre McKenzie. Oracle branching programs and logspace versus p. *Information and Computation*, 95(1) :96 – 115, 1991.

[Coo71] Stephen A. Cook. Characterizations of pushdown machines in terms of time-bounded computers. *J. ACM*, 18(1) :4–18, 1971.

[Geh01] Dominik Gehl. Programmes de branchement restreints pour un problème p-complet. Master's thesis, Université de Montréal, 2001.

[GKM15] Vincent Girard, Michal Koucký, and Pierre McKenzie. Nonuniform catalytic space and the direct sum for space. 2015.

[HMRU00] John E. Hopcroft, Rajeev Motwani, Rotwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computability*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2000.

[Lup62] Oleg B. Lupanov. On a class of circuits with functional elements (partial-memory formulas). *Problems of cybernetics*, 7 :68–136, 1962.

[Pau76] Wolfgang J. Paul. Realizing boolean functions on disjoint sets of variables. *Theoretical Computer Science*, 2(3) :383 – 396, 1976.

[Per14] Sylvain Perifel. *Complexité algorithmique*. Éditions Ellipses, 2014.

[Pot16] Aaron Potechin. A note on amortized space complexity. 11 2016.

[Tur36] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1) :230–265, 1936.

[Uhl74] Dietmar Uhlig. On the synthesis of self-correcting schemes from functional elements with a small number of reliable elements. *Mathematical notes of the Academy of Sciences of the USSR*, 15(6) :558–562, 1974.

[Uhl92] Dietmar Uhlig. Networks computing boolean functions for multiple input values. In *Poceedings of the London Mathematical Society Symposium on Boolean Function Complexity*, pages 165–173, New York, NY, USA, 1992. Cambridge University Press.

[Weg87] Ingo Wegener. *The Complexity of Boolean Functions*. 1987.