N° d'ordre : XXX

École normale supérieure de Lyon
Laboratoire de l'informatique du parallélisme

Mémoire de thèse pour l'obtention du grade de
Docteur de l'École normale supérieure de Lyon en informatique
au titre de l'École doctorale de mathématiques et informatique fondamentale
Spécialité : Informatique

présentée et soutenue publiquement le 29 septembre 2000
par M. **Jean-Christophe Dubacq**

Notion de préfixe dans la complexité de Kolmogorov et les modèles de calcul

---

Après avis de : M. Adam Cichon et M. Jean-Yves Marion
M. Serge Grigorieff

Devant la commission d'examen formée de :
M. François Denis
M. Bruno Durand (directeur de thèse)
M. Serge Grigorieff (rapporteur)
M. Jean-Yves Marion (rapporteur)
M. Jacques Mazoyer (directeur de thèse)
M^lle Véronique Terrier

Sommaire

Page laissée intentionnellement blanche

Préliminaires

Introduction

> *« Je laisse intentionnellement de côté la plus ou moins grande longueur pratique des opérations ; l'essentiel est que chacune de ces opérations soit exécutable en un temps fini, par une méthode sûre et sans ambiguïté »*
>
> Émile Borel, *Le calcul des intégrales définies*

Les plus vieux textes mathématiques apparaissent souvent comme une succession d'algorithmes de calcul numérique ou de géométrie. Cependant, il a fallu attendre la fin du XIX^{e} siècle pour qu'on cherche à mieux identifier cette notion d'algorithme. La première ébauche de définition a — à notre connaissance — été proposée par Borel au début du siècle (dans [Bor12]). Ces problèmes fondamentaux ont de fait été traités simultanément avec la question de savoir ce qu'on peut calculer par algorithme. Cette question est devenue fondamentale lorsqu'on s'est aperçu qu'elle était liée à la complétude du calcul des prédicats, et donc à la fondation des mathématiques sur la logique du premier ordre, logique née à la fin du XIX^{e}siècle. La notion de fonction calculable de $\mathbb{N}$ dans $\mathbb{N}$ qui se trouve sous-jacente à la réponse négative de Gödel donnée au problème de complétude (on peut se référer à [Usp94]).

À partir des années 1930, des sous-ensembles des fonctions de $\mathbb{N}$ dans $\mathbb{N}$ ont été proposés comme candidats à être les fonctions calculables, la proposition qui nous est restée étant celle des « fonctions récursives selon Church ». Cet ensemble est obtenu algébriquement par clôture de trois opérations fondamentales (récurrence primitive, composition, minimisation) sur des objets simples (fonctions de $\mathbb{N}$ dans $\mathbb{N}$ intuitivement élémentaires). Deux faits sont à observer :

- Aucune description structurelle n'a été obtenue : si la notion de description a été cernée, il n'a pas été possible de comprendre celle de fonction calculable indépendamment de sa description. En d'autres termes, il n'est pas possible de comprendre la notion de « fonction calculable » sans celle de processus de calcul, liée à la notion d'algorithme. On ne peut pas se passer d'une notion « extérieure », celle d'algorithme.
- Les fonctions calculables ainsi définies ne sont pas des fonctions de $\mathbb{N}$ dans $\mathbb{N}$ mais des fonctions définies sur une partie de $\mathbb{N}^p$ (p quelconque) et à valeurs dans $\mathbb{N}$. Il a fallu attendre plusieurs décennies pour qu'une définition de ce type soit obtenue pour seulement les fonctions définies sur une partie de $\mathbb{N}$ à valeurs dans $\mathbb{N}$ (par R.

Péters dans [Pé67]). Ceci nécessitait de comprendre le rôle fondamental joué par le codage de $\mathbb{N}^2$ dans $\mathbb{N}$ dans la définition d'un modèle de calcul. Le rôle de ce codage est amplement développé dans cette thèse.

Quelques années plus tard, A. Turing a proposé une autre définition des fonctions calculables connue aujourd'hui sous le nom de *fonctions calculables par machines de Turing.* Cette définition est fondamentalement différente de celle de Church car elle consiste en deux points : d'abord la description d'une abstraction de « machine physique » et ensuite une définition des fonctions calculées par cette machine. Cette approche est actuellement la plus classique et nombre de modèles de calculs ont ensuite été définis, des systèmes de Post aux machines de Kolmogorov-Uspensky en passant par les machines de Markov ou celles de von Neumann pour ne citer que les modèles les plus célèbres, le dernier étant un des plus proches de nos ordinateurs.

Il y eut des réticences profondes dans la communauté des logiciens à admettre que les fonctions récursives algébriquement définies par Church étaient les mêmes que les fonction mécaniquement calculables de Turing. Ce qu'on appelle actuellement la thèse de Church est que quelle que soit la modélisation qu'on invente pour notre idée intuitive de ce qu'est un algorithme, on aboutit au même ensemble de fonctions calculables. Cette thèse fut historiquement combattue par Church avant qu'il ne s'y rallie. La preuve d'équivalence entre le modèle de Turing et celui de Church a été faite par Turing, la généralisation aux autre modèles de calcul proposée par Post, ainsi cette thèse devrait s'appeler thèse de Church-Turing-Post.

Pour pouvoir parler de calcul, il faut extraire les ingrédients fondamentaux de ces modèles ce qui a conduit à la notion de *système acceptable de programmation.* On observe qu'alors on peut étudier rigoureusement dans un cadre bien défini les systèmes de calcul. La notion de système acceptable de programmation structure mathématiquement le cadre formel du calcul. Cependant, elle ne dit rien sur les algorithmes qu'on s'autorise à considérer. Un modèle de calcul correspond à la description d'un ensemble dénombrable d'algorithmes. En se donnant de plus une bijection $\langle \cdot, \cdot \rangle$ de $\mathbb{N}^2$ sur $\mathbb{N}$, cet ensemble d'algorithmes peut alors apparaître comme étant structuré en système acceptable de programmation. Plus formellement, il importe de distinguer le numéro (ou le code) d'un algorithme (son numéro dans l'énumération des algorithmes admissibles dans le modèle de calcul considéré, l'algorithme lui-même et la fonction calculée par l'algorithme. L'habitude veut qu'on note n le code, ϕ_n l'algorithme et Φ_n (ou encore ϕ_n) la fonction calculée. Les notations utilisées dans ce texte sont précisées page 13.

Le fait qu'un système acceptable de programmation soit uniformément stable pour la composition peut être vu comme le fait que tout programme doit pouvoir, à un codage près ($\langle \cdot, \cdot \rangle$) accepter un résultat $\phi_n(m)$ comme une donnée possible. La bijection de $\mathbb{N}^2$ dans $\mathbb{N}$ permettant de mélanger deux variables et cet autre point permettent de considérer qu'il existe un programme sachant tout calculer. En outre si le système contient toutes les fonctions calculables par machines de Turing, on montre qu'on peut considérer un numéro de code comme une entrée et réciproquement (théorème *s-m-n*).

Donc, en langage moderne, un même entier peut être vu comme un numéro de code, une entrée ou une sortie. Il est alors possible de montrer, via le théorème de la récursion de

Kleene, que l'application $\phi : n \to \Phi_n$ n'est pas injective. Qui plus est toute fonction p.p.r. [1] est obtenue à partir d'un nombre infini de numéros de code.

Un système acceptable de programmation modélise correctement nos capacités de calcul si les fonctions p.p.r. qu'il fournit sont exactement les fonctions intuitivement calculables (celles obtenues par un des systèmes précédents) et si on sait (au sens intuitif) calculer $\langle \cdot, \cdot \rangle$ (en particulier, les deux composantes de la réciproque sont intuitivement calculables).

En outre si deux systèmes acceptables de programmation donnent comme ensemble de fonctions calculables les fonctions admises comme calculables, ils sont isomorphes : une fonction bijective, admise comme calculable, permet de passer de l'un à l'autre en laissant la fonction image invariante (théorème de l'isomorphisme de Rogers). Ceci implique qu'à un haut niveau d'abstraction toutes les façons de modéliser les fonctions calculables sont équivalentes. Cependant chaque modélisation correspond à un point de vue particulier et pose des questions spécifiques.

Dans notre thèse, nous considérerons que calculer c'est définir un système acceptable de programmation modélisant correctement le modèle de calcul que nous étudions.

Une grande différence entre la vision fonctions récursives à la Church et les machines de Turing est que la première modélisation concerne des fonctions sur les entiers et la seconde sur les mots. Il est possible de passer facilement d'une vision à l'autre (via l'écriture d'un entier dans une base ou un ordre bien fondé sur les mots) ; mais cela implique le choix d'une relation standard entre entiers et mots. Les ordinateurs modernes calculent sur des mots et pas des entiers, ce qui privilégie les modèles du genre de celui de Turing. Le modèle des machines de Turing est plus proche des ordinateurs actuels pour de nombreuses autres raisons :

— Il peut être vu comme un système dynamique avec une notion de temps naturelle (l'itération). Or la présence (pour des raisons technologiques) d'une horloge sur nos processeurs conduit à une notion forte de temps de calcul.
— Par la même, les sous-classes qui en émergent naturellement (classes de complexité) correspondent à des réalités de nos ordinateurs.
— Enfin, les mots ont une structure d'ordre, le ruban d'une machine de Turing est, naturellement, ordonné. L'ordre canonique sur les mots devient par codage de l'entrée l'ordre canonique du ruban. Les déplacements de la tête sont « locaux » vis-à-vis de l'ordre du ruban.

Une des caractéristiques d'une machine de Turing est l'aspect très « géométrique » de son ruban. Le fait que ce soit $\mathbb{Z}$ permet de connaître par déplacement de la tête la lettre suivante. En fait la structure $\mathbb{Z}$ n'est pas nécessaire, on peut prendre $\mathbb{Z}^2$ mais pas n'importe quoi (pas un graphe non récursif).

Ces ingrédients se retrouvent tous dans les machines de Turing :

— le contrôle central est l'ensemble des transitions d'états,
— le contenu du ruban est un mot (fini ou infini ?) et le reste étape après étape,
— l'interaction machine-configuration est la position de la tête de lecture-écriture. Elle est « locale » pour l'ordre du ruban.

1. fonctions partielles partiellement récursives

Un tel modèle est assez proche des ordinateurs (séquentiels) actuels où (sommairement) le ruban correspond aux divers disques, les états à la mémoire et la position de la tête à l'« origine » des coordonnées sur les disques.

Calculer n'a de sens que si les mots sur lesquels on calcule correspondent à une certaine réalité, « codent » la réalité. Cette notion de codage est assez fortement dépendante du modèle de calcul choisi : ainsi les entiers ne sont pas codés dans le cas des fonctions récursives à la Church. Dans le cas des mots, le codage d'un alphabet sur un autre a été particulièrement étudié. La notion de codage préfixe est fondamentale. Informellement, on peut la définir par le fait que partant du début d'un mot, on peut en reconnaître la fin. Un tel codage est naturel sur nos ordinateurs via la notion de pointeur où l'adresse indique le début du mot et la taille de l'objet permet de retrouver sa fin. Dans le cadre des machines de Turing telles qu'usuellement présentées, elle est présente via le fait que l'entrée est la partie non blanche connexe du ruban.

La notion d'algorithme que nous discutons dans cette thèse a pour objet de manipuler, de transformer l'information. Ce point est si clair dans l'esprit des chercheurs de notre communauté que nos collègues étrangers nous envient le terme « informatique » bien mieux adapté à notre science que les variations sur « computer science ». Ainsi l'articulation entre la notion d'algorithme et celle d'information peut être faite de façon puissante et c'est justement un des objectifs principaux de la théorie algorithmique de l'information, aussi appelée théorie de la complexité de Kolmogorov. Cette théorie constitue en fait le prolongement le plus puissant de la théorie générale des algorithmes : la calculabilité. Un des objectifs de notre thèse est d'expliquer les liens entre les théorèmes fondamentaux de la complexité de Kolmogorov, les différentes variantes de cette complexité, avec les modèles de calcul proprement dits. La complexité de Kolmogorov s'est développée fortement dans les années 1970 où certaines de ces variantes sont apparues comme bien plus puissantes que d'autres, permettant par exemple de définir des suites infinies incompressibles et de montrer que ce sont exactement les suites aléatoires selon Martin-Löf — théorème de Levin et Schnorr. Ce genre de théorème très puissant a eu de multiples retombées notamment dans le cadre de la combinatoire et de la complexité algorithmique. Ils ont été principalement popularisés par Chaitin et on peut lire dans nombre d'articles que ce sont ces variantes qui sont les plus proches de nos modèles d'ordinateurs réels. Néanmoins, une étude approfondie et sérieuse des liens entre ces variantes de la complexité de Kolmogorov et les modèles de calculs n'avait pas été entreprise à notre connaissance avant notre thèse. Ceci est certainement dû au fait qu'on n'a pas besoin de redescendre au modèle de calcul proprement dit pour définir la complexité de Kolmogorov et ses variantes : pour la définition et le principal théorème d'optimalité additive (Kolmogorov-Solomonoff) seules des hypothèses assez générales de codages et d'optimalité sont nécessaires, pour les variantes, on peut se contenter de propriétés de monotonie ou de préfixe. Nous avons choisi de descendre un peu plus bas dans les modèles de calculs pour comprendre en quoi ces dernières propriétés sont ou non naturelles pour la modélisation de nos ordinateurs réels.

Le chapitre 1 présente les définitions de base sur les codages et les différentes formes de complexité algorithmique ainsi que les résultats fondamentaux. Nous avons fait le choix de nous placer dans le cas où l'alphabet est quelconque et non plus $\{0, 1\}$. Pour aider la

lecture, nous avons parfois introduit quelques notations légèrement différentes (et, nous l'espérons, plus parlantes) de celles généralement utilisées. Le tableau situé page 13 en donne les principales, usuelles ou non.

Un des intérêts majeurs de la complexité algorithmique est qu'elle fournit une définition possible pour la notion d'aléatoire et permet donc de montrer le caractère aléatoire ou non aléatoire de divers phénomènes. Le chapitre 2 commence par rappeler les différentes définitions de la notion d'aléatoire et leurs propriétés.

Ce chapitre se termine par une utilisation de ces notions pour l'étude de la classification des automates cellulaires. Les automates cellulaires de dimension 1 apparaissent comme un système parallèle simple et bien formalisé dans lequel il est possible de montrer de nombreuses propriétés ensemblistes (réversibilité). Après avoir étudié comment simuler une machine de Turing par un automate cellulaire réversible [Dub95] ainsi que diverses extensions possibles de la notion d'automate cellulaire [Dub94], il était naturel d'examiner la « complexité » de l'évolution d'un tel système. Cette question rejoint la problématique de la classification des automates cellulaires, et, en particulier, celle de définir un paramètre (syntaxique, portant sur les tables de transitions d'états) rendant compte de leur « complexité » d'évolution. Nous montrons que la complexité algorithmique permet de donner sens et de définir un tel paramètre optimal. Elle permet aussi de montrer que les propriétés usuelles (injectivité, additivité) ne sont jamais satisfaites pour un automate de table aléatoire.

Dans le chapitre suivant 3, nous étudions les modèles induit par les machines de Turing lorsque nous contraignons leur dynamique. Les contraintes choisies sont de type « préfixiel ». Nous étudions comment contraindre l'arrêt et le résultat d'une machine de Turing sur une entrée $\langle u \cdot v, w \cdot t\rangle$ en fonction de son arrêt et son résultat sur $\langle u, w \cdot t\rangle$ et $\langle u \cdot v, w\rangle$.

Cela nous amène à définir ces objets en distinguant quatre notions principales : machines préfixes creuses, pleines, généralisées et doublement préfixes. Pour chacun de ces quatre types, nous obtenons un ensemble de machines (algorithmes) et les étudions en tant que systèmes acceptables de programmation. Tous ces systèmes admettent une fonction d'énumération (méta-)effective laissant invariante la fonction calculée, ce que nous appelons un *projecteur*. On en déduit alors qu'ils forment chacun un système acceptable de programmation (comme défini plus haut), que nous comparons alors entre eux (ils sont tous moins puissants que le modèle Turing). La classe des machines doublement préfixes diffère des trois autres car elle ne semble pas vérifier la propriété du *s-m-n* (interne). De même, et contrairement aux autres classes préfixes, elle n'admet pas une notion de complexité de Kolmogorov.

Lorsqu'on définit les machines de Turing, on est amené à montrer des équivalences entre les différentes variantes possibles du modèle (nombre de rubans, rubans bi- ou semi-infinis, type des mouvements autorisés, nombre d'états, nombre de lettres, auto-délimitation, etc.). Ces équivalences ne posent pas de problèmes sauf deux :

— Existence de marqueurs de début et de fin : dans ce cas, on doit supposer que l'entrée est déjà délimitée (partie connexe non blanche du ruban).
— Réduction du nombre de lettres : on arrive à un alphabet $\{0, 1, B\}$ mais pas $\{0, 1\}$ sans changer le codage initial.

Dans le chapitre 4, nous considérons les divers modèles de calcul obtenus lorsque l'alphabet de la machine de Turing est petit ($\{1, B\}$) et lorsqu'on ne suppose pas que la partie non significative du ruban ne contient que des blancs mais est constituée de bruit.

Si l'alphabet est $\{1, B\}$, alors le modèle ne calcule que les fonctions $f(\ell(x))$ où f est p.p.r et $\ell(x)$ est la longueur de x. Si en outre on suppose que à chaque étape du calcul la partie non blanche du ruban est connexe, on réduit la puissance de calcul à celle d'un transducteur rationnel.

Si on suppose que le ruban contient d'autres lettres que le blanc, on peut voir le contenu (infini) non utile du ruban comme un oracle. On est alors conduit à définir une hiérarchie de puissance. Autoriser de faire varier comment on peut coder au temps initial un objet, introduit la notion de plongement dont nous examinons les liens avec la complexité algorithmique.

Enfin, nous terminons par des analogies entre notre travail et les systèmes d'exploitation courants.

Table des notations

Notations relatives aux mots finis

u, v, etc.	Mots finis
ξ, ν, σ	Alphabets (finis)
0, 1, etc.	Lettres de ξ
Ξ	Ensemble des mots finis ($\Xi = \xi^\star$)
$\ell(x)$	Longueur du mot x
ε	Mot vide (de longueur nulle)
$u \cdot v$	Concaténation de u et v
$\mathsf{u}_{\langle n \rangle}$	n-ième lettre du mot u
a^n	Lettre a répétée n fois
$\underline{v}_{\{0,1\}}$	Mot écrit dans l'alphabet $\{0,1\}$

Notations relatives aux mots infinis

$\mathbf{u}$	Mot infini
$\Omega = \xi^\omega$	Ensemble des mots infinis sur ξ
$\lhd u$	Ensemble des mots infinis commençant par u
$u \odot \mathbf{w}$	Concaténation d'un mot fini et d'un mot infini
$u \wr \mathbf{w}$	« u par dessus $\mathbf{w}$ » (remplacement du début de $\mathbf{w}$ par u)
a^ω	Répétition à l'infini de la lettre a

Notations ensemblistes

$\{x, P(x)\}$	Ensemble des x vérifiant $P(x)$
$\lvert X \rvert$	Cardinal de l'ensemble X
$\sharp\{x, P(x)\}$	Cardinal de l'ensemble $\{x, P(x)\}$
$\mathbb{N}, \mathbb{Z}, \mathbb{Q}$	Ensemble des entiers naturels, relatifs, des rationnels
$\mathbb{K}$	Ensemble des numéros de machines tels que $\phi_n(n)$ converge
$\overline{X}$	Complémentaire dans Ξ ou $\mathbb{N}$ de X

Notations diverses sur les machines

$\hat{f} \mathrel{\hat{=}} \hat{\phi}_i$	Machines et égalité de machines
$\langle M \rangle$	Numéro (ou code) de la machine M
$M(x)\downarrow$	La machine M converge sur l'entrée x
$M(x) = \perp$	La machine M diverge sur l'entrée x

Notations diverses sur les fonctions

$f = \phi_i$	Fonctions et égalité de fonctions
$\mathrm{Dom}\, f$	Domaine de la fonction f
$\mathrm{Im}\, f$	Image de la fonction f
p.p.r.	Fonctions partielles partiellement récursives
$\overline{x}$	$1^{\ell(x)} \cdot 0 \cdot x$ (notation auto-délimitée de x)
$\mathcal{O}(f)$	Fonction négligeable ou du même ordre de grandeur que f
$\langle \cdot, \cdot \rangle$	Une bijection récursive de $\Xi^2 \to \Xi$ (dite standard)
$L(x)$	Distribution uniforme de probabilité des mots (voir page 43)
$\log_b(x)$	Logarithme en base b de x
$\lfloor x \rfloor$, $\lceil x \rceil$	Arrondi de x à l'entier inférieur, supérieur

Chapitre 1 Codes et Complexité algorithmique

Notre premier but est d'établir un discours cohérent allant de la complexité de Kolmogorov de base, jusqu'aux définitions de la notion de suite aléatoire. Plus précisément, nous partons de la notion générale de codage et à travers elle, nous expliquons les sources du concept de complexité de Kolmogorov. Intuitivement, la complexité de Kolmogorov d'un objet est la plus petite quantité d'information nécessaire pour reconstruire cet objet algorithmiquement. Bien sur, toutes les notions introduites dans cette phrase comme la *quantité d'information*, un *objet*, *plus petit* et *algorithmiquement* doivent être définies avec précaution car la théorie qui en découle est susceptible de perdre tout son sens si l'on fait une erreur au niveau de ces définitions.

1.1 Notion de codage

Supposons qu'il nous faille transmettre à un correspondant la réponse à une question. La durée de la correspondance ne dépend pas nécessairement du choix effectué. En effet, les réponses « oui » et « non » ont la même durée, indépendamment de la question posée. Lorsque le nombre de réponses possibles augmente, celles-ci s'expriment de manière plus complexe, nécessitant une augmentation de la durée de communication. Nous pouvons ainsi dégager un premier principe : la réponse à une question dépend du nombre de possibilités offertes et non pas de la nature même de la question.

Nous ne nous intéressons donc pas au sens des choix, mais à la « quantité d'information » nécessaire pour, *connaissant les choix possibles*, déterminer celui qui a été retenu.

Une donnée, du point de vue du traitement de l'information, est un choix parmi un ensemble de valeurs. Nous nous limitons en général aux données entières, quelques fois aux données réelles, vues comme une suite infinie de chiffres. Un deuxième grand principe, à la base de la théorie du codage, est que tout choix, et conséquemment toute donnée, peut être représenté de façon unique, par un certain nombre de réponses *binaires* à des questions. On se concentre uniquement sur les transmissions basées sur ce principe ; les autres ne relèvent pas du même domaine d'étude. [1]

1. En fait, on limite les réponses à un choix fini, ce qui est naturel.

Il existe plusieurs possibilités pour exprimer un seul et même choix. Par exemple, pour transmettre le nombre 1 372 073 885 318 497 127 491 074 758 162 987 278 899 500 548 096, nous pouvons envoyer une image noir et blanc, où apparaît l'écriture manuelle du nombre. Nous pouvons aussi envoyer les 49 caractères ASCII « 1 », ..., « 6 ». Et enfin, nous pourrions envoyer la phrase « L'exposant minimal de 14 d'au moins 49 chiffres ».

Ainsi, la façon d'indiquer son choix peut revêtir plusieurs formes. Il est également clair que le sens associé aux objets à choisir ne contraint pas la description : au contraire, toute signification supplémentaire d'un mot peut aider à le décrire de façon différente, ce qui, selon le contexte, peut être une façon de le décrire plus compacte.

Soit un ensemble de choses à coder ou ensemble de significations. Soit par ailleurs un ensemble de messages transmissibles. La première partie de cette étude traitera uniquement du cas où les messages transmissibles seront dénombrables. Il s'agira, dans leur représentation la plus simple, des mots de longueur finie sur un alphabet fini. Nous identifierons ainsi tous les messages à transmettre — et conséquemment toutes leurs identifications sémantiques — aux mots sur un alphabet σ (ou aux entiers, voir §1.2.1). Un *codage* sera donc une fonction de $\sigma^\star$ dans $\xi^\star$, injective, et le *code* sera l'image du codage (on l'appelle l'ensemble des mots de code). [2]

1.1.1 Les codes récursifs

Intéressons-nous aux codages qui peuvent être réalisés par algorithme : les autres codages sortent du cadre de cette étude. En se fondant sur la notion de fonction récursive, un codage est défini par une fonction totale récursive E, mais pas forcément par une fonction d'image récursive. En effet, supposons que les processus de codage et de décodage soient conçus en même temps : l'ensemble des messages ne serait donc pas *a priori* identifiable. Par exemple, codons le n-ième mot dans une certaine énumération par le numéro de la n-ième Machine de Turing s'arrêtant sur l'entrée vide (dans un ordre pas nécessairement croissant). Il n'est pas possible *a priori* de décider si un mot donné fait partie du code, mais il est possible de décoder un mot *si on est sûr qu'il fait partie du code*.

Soit D la fonction réciproque (inverse) de E, dite fonction de décodage. L'image de E (et donc le domaine de D) est appellé le *code*.

Les codages agissent sur des objets dénombrables. Ces objets peuvent donc être plongés indifféremment dans $\mathbb{N}$ ou dans les mots sur un alphabet ξ, ensemble que nous appellerons Ξ. Nous utilisons le plongement dans Ξ qui offre l'avantage de proposer directement une écriture des objets. Cet ensemble peut être muni de plusieurs ordres, dont un total (longueur-lexicographique), et un partiel (l'ordre préfixe). Rappelons que le premier ordre est $x < y$ si $\ell(x) < \ell(y)$ ou si x est avant y dans l'énumération canonique des mots de longueur fixe (étant donné un ordre sur ξ). L'ordre préfixe est un ordre partiel où $x < y$ si x est un préfixe de y (voir §1.2.1).

2. On utilise le cas général où l'alphabet de départ et de codage peuvent être différents. Pratiquement, ils sont souvent identiques, en particulier dans le cas de la modélisation du calcul, chapitre 4.

Définition 1 (Extension d'un codage) L'extension $\mathcal{E}$ d'un codage E aux listes ordonnées et finies de mots est l'application de $\Xi^\star$ dans Ξ muni de la concaténation « $\cdot$ » :

$$\begin{cases} \mathcal{E}(\Lambda) = \varepsilon, \Lambda \text{ étant la liste vide de } \Xi^\star \\ \forall X \in \Xi^\star, \forall y \in \Xi, \mathcal{E}(X, y) = \mathcal{E}(X) \cdot E(y) \end{cases} \tag{1.1}$$

L'application $\mathcal{E}$ n'est pas toujours injective. Par exemple, si $E = Id_\Xi$, alors $\mathcal{E}(00, 00) = \mathcal{E}(0, 000) = 0000$.

1.1.2 La notion de codage préfixe

Certains codes sont *décodables de façon unique*, c'est-à-dire que l'extension du codage aux listes ordonnées et finies de mots est injective. Cette notion se confond avec l'auto-délimitation : un codage d'extension injective est dit auto-délimité, car il contient en lui toutes les informations nécessaires pour séparer les mots de la suite d'origine. C'est pourquoi il est intéressant de savoir caractériser ces codes particuliers.

L'extension par la méthode décrite par l'équation (1.1) n'est pas la seule façon d'étendre un code à une suite finie de mots. On peut notamment décrire d'autres extensions qui perdent la propriété de conservation par concaténation. Par exemple, le codage d'une suite de mots pourrait être réalisé par l'utilisation d'un caractère spécial entre les mots, ou encore en utilisant un code de contrôle pour marquer la fin des mots : on marque la fin d'un mot par un caractère supplémentaire de sorte que la somme de toutes les lettres du mot (considérées comme des entiers) et du caractère de séparation soit constante modulo n. Nous ne nous interessons qu'aux extensions définies par concaténation simple.

On introduit la définition suivante :

Définition 2 Un code préfixe est un code vérifiant la propriété suivante : pour tout mot de code, c'est-à-dire l'image d'un mot par la fonction de codage, il n'existe aucun mot préfixe propre de ce mot de code qui fasse aussi partie du code.

On peut tout de suite tirer de la définition précédente la proposition suivante, qui traduit quelque chose de très naturel. Lorsque l'on rajoute des lettres à un mot, et qu'on obtient un mot valide, il ne peut plus se prolonger en un mot de code. Donc on sait reconnaître, dans un enchaînement de mots, la fin d'un mot :

Proposition 0.1 *Tout code préfixe injectif est uniquement décodable.*

◇ *Preuve.* Puisque aucun préfixe d'un mot du code ne fait partie du code, si l'on donne une concaténation de plusieurs mots, il en existe un unique préfixe qui sera un et un seul mot de code. Il existe, car $\mathcal{E}(x_0, \ldots, x_n) = E(x_0) \cdot \ldots \cdot E(x_n)$, et $E(x_0)$ est un mot du code ; il est unique de par la définition. Ainsi, il existe un unique $E(x_0)$ possible ; et par récurrence immédiate, la suite $E(x_0), \ldots, E(x_n)$ est aussi unique. Par injectivité, la suite $x_0, \ldots, x_n$ est également unique. □

Ces codes ont une propriété puissante : l'inégalité de Kraft. Cette inégalité caractérise la convergence pour les codes préfixes, ce qui aura des conséquences fondamentales dans

l'étude des machines préfixes. Ce ne sont pas les seuls codes vérifiant cette inégalité, mais tout code la vérifiant peut être transformé en code préfixe en gardant les longueurs des mots de code constantes, comme expliqué par le théorème suivant :

Théorème 1 (Inégalité de Kraft) *Ces deux propriétés sont vraies :*

(i) Tout code E dans $\xi^\star$ décodable de façon unique vérifie l'inégalité de Kraft :

$$\sum_{x\in\mathbb{N}} |\xi|^{-\ell(E(x))} \leqslant 1, \tag{1.2}$$

(ii) Pour toute suite de nombre vérifiant la condition de Kraft $\sum_{i\in\mathbb{N}} |\xi|^{-l_i} \leqslant 1$, il existe un code préfixe dans $\xi^\star$ tel que $\ell(E(i)) = l_i$.

◇ *Preuve.* Sous cette forme, ce théorème est attribué à B. Mc Millan dans [Gal68]. Il avait été prouvé auparavant pour la classe plus restreinte des codes préfixes.

Propriété *(i)* La preuve de l'équation (1.2) se fait sur les suites finies de mots. Supposons que l'on ait N mots de codes utilisant des mots de codes de longueur au plus m. Soit n_i, $1 \leqslant i \leqslant N$ les longueurs des mots de codes utilisés. Soit r_i le nombre de mots de i lettres qui peuvent être obtenus en accolant r mots parmi les N. On va regrouper, dans le produit $\left(\sum_{k=1}^{N} |\xi|^{-n_k}\right)^r$, tous les groupements de mots dont la longueur totale est identique, en développant le produit :

$$\begin{aligned}\left(\sum_{k=1}^{N} |\xi|^{-n_k}\right)^r &= \sum_{k_1=1}^{N} |\xi|^{-n_{k_1}} \sum_{k_2=1}^{N} |\xi|^{-n_{k_2}} \dots \sum_{k_r=1}^{N} |\xi|^{-n_{k_r}}, \\ &= \sum_{k_1=1}^{N}\sum_{k_2=1}^{N}\dots\sum_{k_r=1}^{N} |\xi|^{-\left(n_{k_1}+n_{k_2}+\dots+n_{k_r}\right)}.\end{aligned}$$

On simplifie en regroupant les mots selon leur longueur finale :

$$\left(\sum_{k=1}^{N} |\xi|^{-n_k}\right)^r = \sum r_i |\xi|^{-i}. \tag{1.3}$$

Toutes les séquences de i lettres doivent être décodables de façon unique, donc $r_i \leqslant |\xi|^i$. De plus le nombre de r_i non nuls est borné. Prenons r mots ; en admettant que toutes les sommes de longueurs soient distinctes, on a un nombre de sommes possibles qui est au plus égal au nombre de façon de répartir r objets dans N cases, soit $\binom{N}{N+r-1}$.

Ainsi, l'équation (1.3) peut se réécrire en une inégalité :

$$\left(\sum_{k=1}^{N} |\xi|^{-n_k}\right)^r \leqslant \binom{N}{N+r-1} \leqslant (r+N)^N,$$

puis par passage à la limite de r

$$\sum_{k=1}^{N} |\xi|^{-n_k} \leqslant \lim_{r\to\infty} (r+N)^{N/r} \leqslant 1. \tag{1.4}$$

On en déduit que l'inégalité (1.4) est vérifiée pour tout N et donc (par passage à la limite) l'inégalité de Kraft est vérifiée pour tout code décodable de façon unique.

Propriété *(ii)* On se donne une correspondance entre les intervalles fermés à gauche et ouverts à droite de $[0,1[$ et les mots de $\Xi = \xi^\star$. On ordonne ξ et on fait correspondre à chacune de ses lettres un numéro unique entre 0 et $|\xi|-1$. On identifie le mot x à l'intervalle $\left[\sum \mathsf{x}_{\langle i\rangle} |\xi|^{-i}, \sum \mathsf{x}_{\langle i\rangle} |\xi|^{-i} + |\xi|^{-\ell(x)}\right[$. Cette correspondance transforme l'ordre partiel d'inclusion sur les intervalles en l'ordre partiel de préfixe sur les mots de Ξ.

Si une suite l_i d'entiers vérifie la condition de Kraft, alors on peut choisir des intervalles disjoints chacun de largeur $|\xi|^{-l_i}$ dans $[0,1[$. Par la correspondance ci-dessus, on obtient un ensemble de mots qui vérifient encore la propriété de préfixe. On choisit un ordre quelconque de ces mots (ordre longueur-lexicographique par exemple). On construit alors un code préfixe en associant à l'entier i le mot de code représentant l'intervalle choisi pour l_i. On obtient alors un code préfixe vérifiant les conditions demandées.

□

1.1.3 Optimalité et entropie

Parmi les objectifs demandés lors du choix d'un codage, on s'attendra souvent à ce que le codage réalisé vérifie une propriété de *compacité* ou *d'optimalité*. On peut en effet s'attendre à ce qu'une information supplémentaire sur le texte à coder permette de choisir un codage dont les textes codés seront globalement plus courts. Un texte est une suite finie ou en quantité dénombrable de mots d'un même code E sur l'alphabet ξ. Les objets manipulés appartiennent donc à $\Xi^\star$ ou à Ξ^ω, et leur représentation est un mot de $\Xi = \xi^\star$ ou bien ξ^ω : leur image par l'extension $\mathcal{E}$ du code initial E.

Si on cherche à garder un codage décodable de façon unique, et si l'on connaît la répartition *a priori* des mots qui sont à coder, il existe un lien entre la *longueur moyenne des mots de codes* et l'entropie de Shannon, dont la valeur est exprimée par la formule $H(P) = -\sum_x P(x) \log_b P(x)$, avec $P(x)$ la probabilité *a priori* d'un mot x et $b = |\xi|$ (dans toute cette sous-partie, b désigne le cardinal de l'alphabet de codage ξ).

Définition 3 Soit P une distribution de probabilités sur l'ensemble des mots de Ξ. Pour tout code décodable de façon unique, ayant E pour fonction de codage et D pour fonction de décodage, la longueur moyenne d'un mot de code est $M_{E,P} = \sum_x P(x)\ell(E(x))$. La longueur moyenne minimale est définie comme étant :

$$\mathcal{M}_P = \min_E\{M_{E,P}, E \text{ décodable de façon unique}\}.$$

Tous les codes vérifiant $M_{E,P} = \mathcal{M}_P$ seront dit *optimaux*.

Cette définition d'optimalité fait apparaître une adéquation entre la fréquence des mots qui apparaissent et la longueur choisie pour le code de ce mot. Cette propriété est à la base de nombreux codages, comme le codage de Huffmann ou de Lempel-Ziv. Énonçons donc le lien entre entropie et codages.

Théorème 2 (Codage parfait) *Soit P une distribution de probabilités sur Ξ. $H(P) = -\sum_x P(x)\log_b P(x)$ est l'entropie (au sens de Shannon) de P. La formule suivante est vraie :*

$$H(P) \leqslant \mathcal{M}_P \leqslant H(P) + 1. \tag{1.5}$$

◇ *Preuve.*

$\underline{\mathcal{M}_P \leqslant H(P)+1}$ Soit $l_x = \lceil -\log_b P(x)\rceil$. Par définition d'une distribution de probabilité, $1 = \sum_x P(x) \geqslant \sum_x b^{-l_x}$. Ainsi, il existe un code préfixe ayant pour longueurs de mots de codes $\{l_i\}_{i\in\mathbb{N}}$ d'après l'inégalité de Kraft (théorème 1). Donc :

$$\mathcal{M}_P \leqslant \sum_x P(x) l_x \leqslant \sum_x P(x)(-\log_b P(x) + 1) \leqslant H(P) + 1.$$

$\underline{H(P) \leqslant \mathcal{M}_P}$ Soit $\{l_x\}$ l'ensemble des longueurs des mots de code d'un code E décodable de façon unique. On part de l'inégalité de convexité de l'exponentielle, qui s'écrit sous la forme :

$$\prod_i e^{\left(\frac{a_i}{\sum_j \alpha_j}\right)^{\alpha_i}} = e^{\frac{\sum_i \alpha_i a_i}{\sum_i \alpha_i}} \leqslant \frac{\sum_i \alpha_i e^{a_i}}{\sum_i \alpha_i}. \tag{1.6}$$

Les α_i doivent être des coefficients positifs, le facteur $\sum_i \alpha_i$ étant un facteur de normalisation. On pose dans cette formule $\alpha_i = P(i)$ et $e^{a_i} = \frac{b^{-l_i}}{\sum_j b^{-l_j} P(i)}$, et pour plus de commodité on écrira $S = \sum_j b^{-l_j}$. Ainsi, l'inégalité (1.6) se réécrit (en tenant compte du fait que $\sum_i P(i) = 1$) :

$$\prod_i \left(\frac{b^{-l_i}}{SP(i)}\right)^{P(i)} \leqslant \sum_i P(i)\left(\frac{b^{-l_i}}{SP(i)}\right) = \sum_i \left(\frac{b^{-l_i}}{S}\right) = \frac{S}{S} = 1. \tag{1.7}$$

On réécrit maintenant l'inégalité (1.7) en séparant le quotient :

$$\prod_i \left(\frac{b^{-l_i}}{S}\right)^{P(i)} \leqslant \prod_i P(i)^{P(i)},$$

puis en passant à l'opposé du logarithme :

$$-\sum_i P(i)\log_b P(i) \leqslant -\sum_i P(i)\log_b\left(\frac{b^{-l_i}}{S}\right).$$

On retrouve maintenant le terme de l'entropie et de la longueur moyenne d'un mot de code de E, et un terme dépendant de S, que l'on peut majorer par $\log_b S$:

$$H(P) \leqslant \sum_i P(i) l_i + \sum_i P(i)\log_b S \leqslant M_{E,P} + \log_b S.$$

Comme $S \leqslant 1$ par le théorème 1 (et donc $\log_b S \leqslant 0$), on a bien la formule $H(P) \leqslant M_{E,P}$ valable pour tout code décodable de façon unique E, et en particulier :

$$H(P) \leqslant \mathcal{M}_P.$$

□

Ce théorème établit une correspondance très forte entre la notion d'entropie et la notion de codage optimal, à condition que ce codage soit décodable de façon unique. En mettant en évidence ce lien, on voit déjà l'importance qu'aura cette notion par la suite, lors de l'étude de la complexité auto-délimitée (voir §1.3).

Cette notion d'optimalité à des limites. En effet, il est nécessaire de connaître des données sur le texte à coder afin d'atteindre cette optimalité (puisque l'on utilise $P(x)$ dans la construction du code vérifiant $M_{E,P} \leqslant H(P) + 1$). Existerait-il un codage qui aurait en plus la propriété de ne pas dépendre de la source ?

C'est pour répondre à cette question que A.N. Kolmogorov (et ultérieurement de nombreuses autres personnes) a cherché à définir la complexité intrinsèque d'une suite afin de savoir s'il existe une machine qui code toute suite au plus court possible. La réponse est partiellement négative, mais cette étude a beaucoup d'autres conséquences sur un grand nombre de domaines, en particulier l'étude des suites aléatoires (voir §2.1.1).

La complexité de Kolmogorov a fait l'objet de nombreux recueils. Un recueil très complet est le livre de M. Li et P. Vitányi [LV97a], mais on trouvera aussi d'excellents survols du domaine dans les livres de C. Calude [Cal93] et O. Watanabe [Wat92].

1.2 Information algorithmique

1.2.1 Hypothèses

Dans cette partie, nous détaillons les choix et les hypothèses nécessaires pour définir la complexité de Kolmogorov.

Entiers et mots On utilise toujours l'ensemble des mots finis sur l'alphabet ξ que nous noterons Ξ. On identifie les entiers aux mots de Ξ en utilisant l'écriture suivante pour un entier : 0 est écrit ε, 1 est écrit 0, 2 est écrit 1, 3 est écrit 00, etc. (ceci pour $\xi = \{0, 1\}$). Plus précisément, on va considérer qu'un entier n est écrit comme le n^{e} mot de Ξ lorsqu'on les classe dans l'ordre longueur-lexicographique : une chaîne plus courte est inférieure à une chaîne plus longue, puis on classe les chaînes de même longueur par ordre lexicographique. Les objets que l'on va manipuler seront donc des mots, mais que l'on peut identifier, par cet ordre qui est total, avec des entiers correspondant à leur numéro d'ordre.

Dans ce paragraphe, si le besoin s'en fait sentir, lorsqu'on écrit un entier de cette façon, on utilise la notation $\underline{x}_{\{\xi\}}$. Ainsi, si on note $\overline{x}^2$ l'écriture binaire classique de x (qui commence toujours par un 1, sauf pour $x = 0$), $\overline{x+1}^2$ est exactement $1 \cdot \underline{x}_{\{0,1\}}$. On retrouve entre autres choses que 0 s'écrit ε.

Notations On rappelle que la longueur d'une chaîne p, notée $\ell(p)$, est le nombre de caractères nécessaires à son écriture. En particulier, si p représente un entier n, alors $\ell(p) = \lfloor \log_{|\xi|}(n+1) \rfloor$. Rappellons de même que pour tout mot u, on note $\overline{u} = 1^{\ell(u)} \cdot 0 \cdot n$. On définit un codage κ noté $\langle \cdot, \cdot \rangle$ de $\Xi \times \Xi$ dans Ξ, fonction récursive totale bijective.

Ordre préfixe On peut ordonner Ξ selon deux ordres. D'une part, il y a l'ordre longueur-lexicographique, d'autre part, en utilisant la relation de préfixe. Un mot est plus petit qu'un autre si et seulement s'il en est un préfixe. On obtient alors un ordre partiel qui sera utile dans certaines démonstrations. Dans ce cas, on appellera « cylindre engendré par x » l'ensemble des mots dont x est un préfixe. On utilise le terme « continuation de x » pour un mot dont x est un préfixe.

Universalité Nous utilisons un système acceptable de programmation qui, sur une entrée $x \in \Xi$, sait calculer une sortie y. On peut par ce modèle de calcul calculer toutes les fonctions partielles partiellement récursives (fonctions p.p.r.). On peut énumérer les machines de ce système sous la forme $\phi_0, \phi_1, \ldots$ telle que l'on ait une machine universelle U vérifiant :

$$\forall \alpha, \beta, \quad U\langle \alpha, \beta \rangle = \phi_\alpha(\beta).$$

Enfin, on a une fonction s_1^1 récursive totale qui vérifie la propriété suivante :

$$\forall \alpha, \beta, \gamma, \quad \phi_\alpha \langle \beta, \gamma \rangle = \phi_{s_1^1 \langle \alpha, \beta \rangle}(\gamma).$$

Plutôt que de se baser sur un modèle de calcul en particulier, il est plus judicieux d'utiliser la définition plus générale d'un système acceptable de programmation :

Définition 4 On appelle *système acceptable de programmation* une liste de programmes indexée par $\mathbb{N}$ $\{\psi_i\}_{i\in\mathbb{N}}$ telle que :

(i) toute fonction Turing-calculable d'arité 1 soit calculable par au moins un des programmes ψ_i,

(ii) il existe un programme *universel* Turing-calculable :

$$\exists u \in \mathbb{N}, \forall i, j \in \mathbb{N}, \psi_u \langle x, y\rangle = \psi_x(y),$$

(iii) il existe une fonction de composition récursive totale c :

$$\forall i, j \ \psi_i \circ \psi_j = \psi_{c\langle i,j\rangle}.$$

1.2.2 Définition de la complexité de Kolmogorov

Définition 5 On définit la complexité (conditionnelle) de Kolmogorov de x sachant y selon la machine ψ, par la formule suivante :

$$\mathbf{K}_\psi(x|y) = \min\{\ell(p), \psi\langle p, y\rangle = x\}. \tag{1.8}$$

Très exactement, cette complexité représente la taille de la donnée minimale (ici p) qu'il faut fournir à la machine ψ pour qu'elle trouve le résultat x. Le mot y est une *connaissance extérieure* dont on ne tient pas compte dans la mesure de cette taille (on dit aussi un contexte). C'est de là que l'on choisit d'effectuer le calcul $\psi\langle p, y\rangle$ au lieu d'un simple $\psi(p)$. Si aucun calcul ayant une entrée de la forme $\langle p, y\rangle$ ne donne x comme résultat, alors on pose naturellement $\mathbf{K}_\psi(x|y) = +\infty$.

Nous utilisons plus loin la notation **KS** où **K** est pour « Kolmogorov » et **S** pour « simple ». Nous définirons ultérieurement des variantes de cette complexité. Dans la littérature, cette valeur est aussi notée, selon les auteurs, C ou K.[3]

Une autre notation qu'il convient de retenir : lorsque plusieurs arguments viennent prendre la place de y, comme dans l'expression $\mathbf{KS}(x^\star|x, \mathbf{KS}(X))$, on considère fixé au préalable une bijection récursive de Ξ^n dans Ξ, où n est le nombre d'arguments (2 dans l'exemple), et on pose y comme étant le résultat de cette bijection sur tous ces arguments. On utilise $\langle a, b\rangle$ pour $n = 2$ et n'importe quelle extension au-delà.

La complexité conditionnelle est donc une fonction à deux arguments qui dépend d'une machine. Nous allons essayer de se débarrasser de cette dépendance en trouvant une machine *universelle*, non pas au sens de la calculabilité, mais au sens où elle donne une complexité aussi petite que toutes les autres. Pour se débarrasser de cette ambiguïté de vocabulaire, on dira qu'elle est *additivement optimale*.

3. ...auquel cas la complexité préfixe (une des variantes) est notée K (respectivement H). C'est pour éviter cette ambiguïté que nous utilisons une notation à deux lettres.

1.2.3 Propriétés principales

Il n'est pas possible d'arriver à la propriété « il existe une machine qui soit plus efficace (en terme de longueur de codage) que toutes les autres (telle que les mots de code obtenus soient toujours plus courts) ». Cependant, on peut prouver une propriété assez proche mais moins contraignante, qui donne une valeur asymptotique à la notion. Plus précisément, l'obstacle est que l'on peut toujours construire une machine qui donne à n'importe quel mot une complexité fixée, voire nulle : l'exemple le plus frappant est encore si l'on prend la machine P_x qui écrit x sur la sortie quelle que soit l'entrée, d'où la complexité conditionnelle $\mathbf{K}_{P_x}(x|y) = 0$. En revanche, on peut obtenir un résultat satisfaisant si on travaille à une constante près, ce qui revient à regarder la complexité de façon asymptotique. En effet :

Définition 6 (Programme additivement optimal) Un programme ψ est dit additivement optimal pour une classe de programmes $\hat{\mathcal{C}}$ si et seulement si $\psi \in \hat{\mathcal{C}}$ et s'il est tel que :

$$\forall \psi' \in \hat{\mathcal{C}}, \exists c_{\psi'}, \forall x, y, \quad \mathbf{K}_{\psi}(x|y) \leqslant \mathbf{K}_{\psi'}(x|y) + c_{\psi'}. \tag{1.9}$$

Théorème 3 (Solomonoff-Kolmogorov) *Dans tout système acceptable de programmation, il existe une machine additivement optimale (pour toutes les machines du système).*

◇ *Preuve.* On construit une machine vérifiant l'inégalité (1.9), donc telle que :

$$M\,\langle \overline{\alpha} \cdot p, y\rangle = \phi_\alpha\,\langle p, y\rangle\,.$$

On utilise un codage pour α qui permet de reconnaître la fin de α dans la concaténation $\overline{\alpha}p$, donc il existe une machine qui transforme $\overline{\alpha}p$ en $\langle \alpha, \langle p, y\rangle\rangle$. Notre machine exécute ensuite la machine universelle U sur cette donnée.

Vérifions maintenant l'inégalité (1.9). On prend une machine $F = \phi_\alpha$. On a, d'après la construction précédente, l'égalité $M\,\langle \overline{\alpha}p, y\rangle = F\,\langle p, y\rangle$.

Donc on a, en particulier, pour p tel que $F\,\langle p, y\rangle = x$:

$$\begin{aligned} \forall x, y, \mathbf{K}_M(x|y) &\leqslant \ell(\overline{\alpha} \cdot p) && (1.10)\\ &\leqslant \ell(\overline{\alpha}) + \ell(p). && (1.11) \end{aligned}$$

L'inégalité (1.11) se déduit de la précédente parce que la longueur de deux chaînes concatenées est la somme des deux longueurs. La première indique juste que $\mathbf{K}_M(x|y)$ est un minimum. Cette inégalité étant vérifiée en particulier pour le p qui réalise le minimum dans le calcul de $\mathbf{K}_F(x|y)$, l'inégalité (1.9) est vérifiée en prenant la constante égale à $\ell(\overline{\alpha})$. □

Néanmoins, il convient de remarquer que cette machine M n'est pas universelle au sens du calcul. En effet, elle ne vérifie pas $M\,\langle \alpha, x\rangle = \phi_\alpha(x)$.

On définit la complexité de Kolmogorov d'un mot x en utilisant une machine fixée, dite *de référence*, additivement optimale. On note alors cette complexité $\mathbf{KS}(x|y)$, le ψ de

référence étant sous-entendu. Cependant, on peut toujours trouver, quelles que soient les valeurs y_1 et y_2 que l'on donne, une machine ψ_1 et une machine ψ_2 toutes les deux additivement optimales, telles que $\mathbf{K}_{\psi_1}(x|y_1) = \mathbf{K}_{\psi_2}(x|y_2)$. La définition d'une telle fonction n'a de sens que lorsque l'on fait varier y en fonction de x, comme par exemple dans l'expression de la complexité $\mathbf{KS}(x|l(x))$ qui est employée dans plusieurs applications.

On a par ailleurs un corollaire qui lie entre elles les machines additivement optimales :

Corollaire 0.2 *Soient deux machines ψ_1 et ψ_2 additivement optimales (vérifiant l'inégalité* (1.9)*). Alors il existe une valeur c ne dépendant que de ψ_1 et de ψ_2, telle que pour tout couple x, y :*

$$|\mathbf{K}_{\psi_1}(x|y) - \mathbf{K}_{\psi_2}(x|y)| \leqslant c$$

◇ *Preuve.* Ce corollaire est un des premiers pas dans l'application de la théorie de l'information algorithmique. On utilise l'inégalité (1.9) deux fois, une fois en prenant ψ_1 comme machine additivement optimale, et ψ_2 comme fonction majorante, et une autre fois en les échangeant, ce qui donne deux valeurs c_1 et c_2 . On peut alors prendre c comme le plus grand de c_1 et c_2. □

Il existe une vision différente, duale de celle qui consiste à fixer une machine de référence : on ne considère plus la complexité de Kolmogorov que comme une complexité de suites, définie à une constante additive près. Ainsi, on est sur de n'observer que des comportements asymptotiques. Énoncer que $\mathbf{KS}(10001011|001001110) = 42$ ne signifie rien en soi ; dire que la progression $\mathbf{KS}(x_n|l(x_n))$ est logarithmique a en revanche un sens qui donne réellement une information sur la suite x_n. Travailler à constante près convient à la plupart des usages : le fait de *fixer* ψ additivement optimale ou de ne considérer que les suites de valeurs (et ainsi des comportements asymptotiques) ne devrait donc pas gêner par la suite. On rappellera toutefois aux moments essentiels que l'on travaille à une constante près.

On peut donc maintenant poser en toute quiétude la définition suivante, sachant que le choix qui est fait ne change la fonction qu'en lui ajoutant (ou retranchant) une fonction de x bornée (notée $\mathcal{O}(1)$) :

Définition 7 La complexité (conditionnelle) de Kolmogorov de x sachant y est définie par $\mathbf{K}_\psi(x|y)$, avec ψ une machine additivement optimale fixée au préalable.

Comme elle sera souvent utilisée par la suite, on note de façon particulière $\mathbf{KS}(x|\varepsilon)$. Elle correspond à ne donner qu'une information constante, éventuellement nulle — mais on a vu précédemment que c'est équivalent —, on la dénomme donc complexité (inconditionnelle) de Kolmogorov de x et on la note $\mathbf{KS}(x)$. On pourra aussi considérer cette notation comme la définition d'une fonction de $\Xi \to \mathbb{N}$ en ordonnant totalement Ξ par l'ordre longueur-lexicographique ; on considérera alors que la fonction de référence a été fixée. De la même façon, on note $\mathbf{K}_\psi(x) = \mathbf{K}_\psi(x|\varepsilon)$.

Une proposition fondamentale établit que l'on connaît une fonction majorant $\mathbf{KS}(x)$. Nous donnons pour cette propriété deux énoncés, un énoncé qui conserve la constante,

et un énoncé plus simple et plus concis, afin de montrer les deux façons de percevoir la complexité de Kolmogorov.

Proposition 0.3 *Pour toute machine additivement optimale ψ, il existe une constante c, telle que pour tout x et tout y :*

$$\mathbf{K}_{\psi}(x|y) \leqslant \ell(x) + c. \tag{1.12}$$

Proposition 0.3 *Pour tout x et tout y :*

$$\mathbf{KS}(x|y) \leqslant \ell(x). \tag{1.12$'$}$$

◇ *Preuve.* On prouve l'énoncé (1.12), l'autre étant immédiat lorsqu'on veut bien se souvenir que l'on travaille à constante près si l'on ne précise pas explicitement la fonction de référence. Considérons par exemple la machine PRINT qui recopie l'entrée sur la sortie. Il est facile de voir que $\mathbf{K}_{\text{PRINT}}(x) = \ell(x)$ d'après l'équation (1.8). La machine ψ étant additivement optimale, on applique l'inégalité (1.9) à ces deux machines. Il en découle exactement l'inégalité recherchée. □

Avant d'aller plus loin dans l'étude de **KS** en tant que fonction, observons que les complexités conditionnelles sont liées aux complexités inconditionnelles par la proposition suivante :

Proposition 0.4 *Pour tout x et tout y :*

$$\mathbf{KS}(x|y) \leqslant \mathbf{KS}(x).$$

◇ *Preuve.* Notons bien que pour la démonstration, il y a implicitement un terme en $\mathcal{O}(1)$ dans l'égalité. Pour la preuve, on construit, à partir de la fonction de référence (additivement optimale) ψ, la machine F qui vérifie $F\langle x, y\rangle = \psi\langle x, \varepsilon\rangle$. Cette fonction induit une complexité conditionnelle $\mathbf{K}_F$ qui est (à une constante additive près) minorée par $\mathbf{K}_{\psi}$, c'est-à-dire que $\mathbf{KS}(x|y) \leqslant \mathbf{K}_F(x|y) + c$. Comme en tout point $F\langle x, y\rangle = \psi\langle x, \varepsilon\rangle$, $\mathbf{KS}(x|\varepsilon) = \mathbf{K}_F(x|y)$ (il n'y a pas de constante, car on compare là deux fonctions complètement définies ψ et F). Donc $\mathbf{KS}(x|y) \leqslant \mathbf{KS}(x)$. □

L'interprétation des deux résultats précédents est naturelle : la première exprime que quelque soit la complexité d'un objet, on pourra toujours le retrouver à partir de son écriture, et donc que sa complexité est asymptotiquement inférieure à sa longueur. La deuxième traduit qu'une information supplémentaire sur un mot — ou plutôt sur une suite de mots — ne peut pas augmenter la complexité des mots, mais seulement la diminuer.

1.2.4 Non-calculabilité de KS

L'un des résultats les plus importants de la théorie de l'information algorithmique concerne l'effectivité du calcul de **KS**, c'est-à-dire la réponse à la question « connaissant x, pouvons-nous calculer ou approcher la valeur de $\mathbf{KS}(x)$? ». La complexité de Kolmogorov

est approchable par valeurs supérieurs. Il sera montré plus loin que cette approche n'est pas uniforme puisque la fonction **KS** *n'est pas* calculable.

Proposition 0.5 *Il existe une fonction totale récursive $S\langle t, \cdot\rangle$ convergeant simplement vers* **KS** *quand t tend vers l'infini, telle que :*

$$\forall t \in \mathbb{N}, \forall x \in \Xi, S\langle t, x\rangle \geqslant S\langle t+1, x\rangle \geqslant \mathbf{KS}(x). \tag{1.13}$$

◇ *Preuve.* On utilise le fait que la **KS**-complexité est bornée par $\ell(x)+c_x$ (proposition 0.3), et que c_x est calculable. On construit alors une machine qui simule la machine ψ sur toutes les entrées de longueur inférieure ou égale à $\ell(x)+c_x$ pendant un temps t, et on retient la meilleure solution qui existe, ou bien $\ell(x)+c_x$ si aucune entrée ne donne un résultat plus petit. Cette fonction vérifie bien l'inégalité (1.13). □

Remarque 1 Cette preuve peut être adaptée à toute machine M pour laquelle on connait une majoration de $\mathbf{K}_M$, car elle ne dépend que du fait que l'on sait borner la complexité au départ.

Ce résultat est important, car il limite, dans une certaine mesure, les propriétés de non-calculabilité de **KS**, et annonce les conditions que l'on utilisera plus tard pour faire des tests de caractère aléatoire : on utilise des fonctions de la même « catégorie » que **KS**, c'est-à-dire approchables par valeurs supérieures mais non calculables.

Il a été annoncé que **KS** n'est pas calculable. Cette propriété essentielle de **KS** a beaucoup de conséquences : si on s'intéresse par exemple à la compression de données, elle montre que l'on ne peut pas faire un programme de compression de données qui soit optimal. Pour cela, on définit pour la durée de cette sous-partie la fonction $m(x)$, qui est la plus grande fonction monotone qui minore **KS**. Mathématiquement, $m(x) = \min\{\mathbf{KS}(y), y \geqslant x\}$, c'est-à-dire que la fonction minore exactement toutes les valeurs de **KS** à venir. Un premier théorème indique que la croissance de m est plus lente que toute fonction p.p.r. croissante vers l'infini :

Théorème 4 (Kolmogorov) *Pour toute fonction partielle partiellement récursive f, monotone qui tend vers $+\infty$, pour tout $x \in \Xi$ sauf un nombre fini, on a $m(x) < f(x)$.*

◇ *Preuve.* Supposons qu'il existe une fonction p.p.r. f monotone tendant vers $+\infty$ et que le domaine R de f soit tel que pour une infinité de $x \in R$, on ait $f(x) \leqslant m(x)$. Il existe un ensemble R' récursif infini contenu dans R.

On peut prolonger $f|_{R'}$ par continuité en dehors de R', par

$$g(x) = \begin{cases} f(x) & \text{pour } x \in R' \\ f(y) & \text{avec } y = \max\{z : z \in R', z < x\} \\ 0 & \text{si } x < \min R' \end{cases}$$

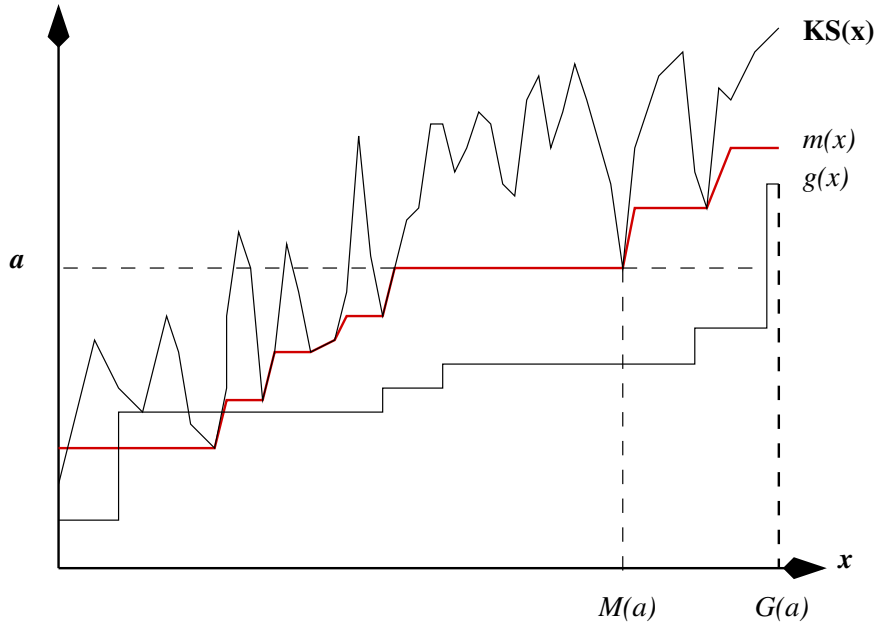


FIGURE 1.1 – Représentation des fonctions

Ainsi g est totale récursive puisque R' est récursif, et monotone croissante tendant vers $+\infty$. De plus si $m(x) \geqslant f(x)$, $m(x) \geqslant f(y)$ pour $y < x$ (puisque f est croissante), en particulier pour $y = \max\{z : z \in R', z < x\}$. Donc $m(x) \geqslant g(x)$.
Maintenant on définit $M(a)$ comme la plus grande valeur x telle que $m(x) \leqslant a$ (voir la figure 1.1). On vérifie instantanément que $M(a) + 1$ est la plus petite valeur telle que $m(x) > a$. De la même façon, on définit :

$$G(a) = \max\{x, g(x) \leqslant a\} + 1.$$

Dans cette expression, $G(a)$ est la plus petite valeur telle que $g(x) > a$. D'après l'hypothèse du départ, pour une infinité de a, $G(a) \geqslant M(a) + 1$. Ce qui signifie, en revenant à la définition de M puis de m, puis encore de l'optimalité, que :

$$a < m(G(a)) \leqslant \mathbf{KS}(G(a)) \leqslant \mathbf{K}_G(G(a)) + c \leqslant \ell(a) + c.$$

La dernière inégalité relève de la définition de $\mathbf{K}_G$ et de l'égalité (1.8). On en extrait donc que pour un nombre infini de a, $a < \ell(a) + c$ ce qui est absurde. Donc le théorème est démontré. □

On sait exprimer de façon plus précise le caractère calculable de **KS**. En fait, **KS** est non-calculable sur tout ensemble infini récursivement énumérable. C'est un peu la contrepartie de la proposition 0.5, qui précise que l'approximation que l'on peut en faire ne sera jamais exacte (et qui prouve aussi que la convergence ne peut pas être uniforme).

Théorème 5 *La fonction* **KS**(.) *n'est pas partielle partiellement récursive. De plus, aucune fonction partielle partiellement récursive ne coïncide avec elle sur un ensemble récursivement énumérable infini de points.*

◇ *Preuve.* Supposons qu'il existe un ensemble récursivement énumérable A sur lequel **KS** est calculable. On en extrait un ensemble récursif R infini sur lequel **KS** vérifie la même propriété. On définit pour cet ensemble R la fonction F suivante :

$$F(t) = \min\{x \in R, \mathbf{KS}(x) \geqslant t\}.$$

Cette fonction est récursive totale parce que **KS** est calculable sur R. Elle prend aussi des valeurs qui sont arbitrairement grandes. Par définition de F, on a $\mathbf{KS}(F(m)) \geqslant m$. Mais en même temps, $\mathbf{KS}(F(m)) \leqslant \ell(m) + c$, ce qui n'est pas possible pour m suffisamment grand. □

Faisons ici une remarque importante. Il est possible en effet de trouver des fonctions récursives qui rencontrent **KS** en un nombre infini de points ; mais dans ce cas, il n'est pas possible d'énumérer une famille infinie de points d'intersection (à cause du théorème précédent). C'est le cas par exemple de fonctions de la forme $\log_{|\xi|}(x) + c$, qui sont proches de la majoration de $\mathbf{KS}(x)$, qui rencontrent toutes les données peu compressibles. Pour cela, il suffit d'établir la proposition 0.6 qui va suivre.

1.2.5 La complexité sachant la longueur

Le concept de la complexité mesurée sachant la longueur correspond à une idée naturelle. Lorsque l'on décrit un objet, une des informations les plus utilisées est souvent sa longueur. Une des variantes de **KS** les plus utilisées est donc la fonction $\mathbf{KS}(x|\ell(x))$. Nous noterons cette fonction sous la forme **KL**. Elle a beaucoup de propriétés communes avec **KS**. On peut démontrer facilement les inégalités suivantes :

Proposition 0.6 $\forall x \in \Xi$,

$$\mathbf{KL}(x) \leqslant \mathbf{KS}(x) \leqslant \ell(x) \tag{1.14}$$

$$\mathbf{KS}(x) \leqslant \mathbf{KL}(x) + 2\mathbf{KS}(\ell(x) - \mathbf{KL}(x)) \tag{1.15}$$

◇ *Preuve.* La première inégalité (1.14) est un corollaire de la proposition 0.3 et de la proposition 0.4. La proposition 0.3 s'applique (avec une constante additive qui n'est pas mentionnée) avec $y = \varepsilon$ pour donner l'inégalité de droite. La proposition 0.4 s'applique aussi, au vu de la définition $\mathbf{KL}(x) = \mathbf{KS}(x|\ell(x))$ donnée ci-dessus.

Par ailleurs, on peut reconstruire x à partir de $\ell(x)$ et d'un plus petit programme u de taille $\mathbf{KL}(x)$ codant x sachant $\ell(x)$. La quantité $\ell(x)$ peut être déduite de $\mathbf{KL}(x)$ et de la différence entre $\ell(x)$ et $\mathbf{KL}(x)$. Si v est un programme de taille $\mathbf{KS}(\ell(x) - \mathbf{KL}(x))$ codant $\ell(x) - \mathbf{KL}(x)$, $\overline{v} \cdot u$ permet de retrouver x (la valeur de $\mathbf{KL}(x)$ peut être retrouvée par $\ell(u)$). Ce qui prouve l'inégalité (1.15). □

L'importance de la deuxième inégalité est que pour tous les x dont la complexité selon la longueur est proche de $\log_{|\xi|}(x)$ (à une constante c près par exemple), alors $\mathbf{KL}(x)$ et $\mathbf{KS}(x)$ ont des valeurs similaires. Or, comme va le montrer le prochain résultat, la majorité des éléments de Ξ sont dans ce cas-là.

1.2.6 Incompressibilité relativement à KS

Tous les mots ne peuvent pas avoir de codages petits. Plus précisément, si on se fixe une valeur maximale, on sait majorer le nombre de mots dont la complexité est inférieure à cette valeur :

Proposition 0.7 *Soit c un entier positif. Pour tout ϕ et tout y, tout ensemble fini A à m éléments a au moins $m - \frac{m|\xi|^{-c}-1}{|\xi|-1}$ éléments x tel que $\mathbf{K}_\phi(x|y) \geqslant (\log_{|\xi|} m) - c$.*

◇ *Preuve.* C'est une preuve combinatoire. Le nombre N de programmes de longueur strictement inférieure à $(\log_{|\xi|} m) - c$ est :

$$N = \sum_{i=0}^{(\log_{|\xi|} m)-c-1} |\xi|^i = \frac{|\xi|^{(\log_{|\xi|} m)-c} - 1}{|\xi| - 1} = \frac{m|\xi|^{-c} - 1}{|\xi| - 1} \tag{1.16}$$

Un programme produit au plus un mot. D'où le nombre de programmes de longueur supérieure ou égale à $(\log_{|\xi|} m) - c$ qui est $m - \frac{m|\xi|^{-c}-1}{|\xi|-1}$. □

Corollaire 0.8 *Il existe au moins un mot de longueur n tel que $\mathbf{KS}(x) \geqslant n$.*

◇ *Preuve.* On utilise la proposition 0.7, pour $c = 0$, et A l'ensemble des mots de longueur n. Le nombre de mots de complexité au moins égale à n est donc supérieur à $\frac{(|\xi|-2)|\xi|^n+1}{|\xi|-1}$, et donc supérieur à 1. □

Remarquons toutefois que, contrairement aux énoncés précédents, on ne considère pas ici les inégalités à constante additive près : il est donc exact de dire qu'il existe un élément x dont la complexité est au moins sa longueur, et de considérer donc la valeur de la complexité pour un objet individuel.

Cette remarque fonctionne aussi pour $\mathbf{KL}(x)$, puisque la longueur des objets considérés est une constante. Cela permet de définir la notion de compressibilité :

Définition 8 Un mot $x \in \Xi$ est dit c-incompressible si et seulement si $\mathbf{KL}(x) \geqslant \ell(x) - c$.

1.3 Information algorithmique auto-délimitée

Comme introduit précédemment (voir §1.1.2), le problème qui consiste à distinguer plusieurs mots donnés à la suite les uns des autres se présente de façon assez naturelle. L'emploi de caractères supplémentaires pour délimiter le début et la fin des mots augmente la longueur du codage, et donc de la transmission d'information. La notion de codage décodable de façon unique permet de contourner cet inconvénient. Or si l'on considère le codage naturellement dérivé de **KS** qui à x associe $x^\star$, l'un des plus petits programmes de longueur $\mathbf{KS}(x)$ permettant de retrouver x, on vérifie que ce codage viole l'inégalité de Kraft et n'est donc pas décodable de façon unique (voir le théorème 1). C'est pour

pallier à ce problème que indépendamment L.A. Levin [Lev74], P. Gács [Gác74] et aussi G.J. Chaitin [Cha75] ont proposé de restreindre les fonctions utilisables non pas aux fonctions partielles partiellement récursives, mais à un sous-ensemble de ces fonctions qui vérifient la propriété d'être préfixes (et donc d'induire des codages uniquement décodables).

Il est à noter que des cas plus généraux d'extension de ces propriétés ont été étudiées par V.A. Uspensky et A.K. Shen dans [US96]. Les preuves sont dans [US93b].

L'une des propriétés remarquables des machines que nous chercherons à caractériser est celle d'avoir un domaine de définition qui forme un code préfixe. Autrement dit, une machine vérifiant cette propriété et s'arrêtant sur un mot x ne doit pas s'arrêter sur un mot qui est une continuation de ce même x.

Définition 9 (Propriété préfixe $\mathfrak{P}$) On dit qu'une fonction F vérifie $\mathfrak{P}$ ou $F \in \mathfrak{P}$ si et seulement si pour tout couple de mots u, v :

$$\left.\begin{array}{c} F(u)\downarrow \\ F(uv)\downarrow \end{array}\right\} \Rightarrow v = \varepsilon$$

La notation $F(x)\downarrow$ signifie que la machine F converge sur l'entrée x.

On ne précisera plus désormais le fait que la machine converge, mais uniquement sa valeur : l'affectation $f(x) = y$ à une fonction calculée par une machine implique la convergence de la machine sur l'entrée x. On note $M(x) = \perp$ le fait que « la machine M diverge sur l'entrée x ».

1.3.1 Définition d'une machine préfixe

Les fonctions calculées par les machines vérifiant cette propriété sont les points fixes d'une fonction particulière.

Proposition 0.9 *Il existe une fonction α de $\mathbb{N}$ dans $\mathbb{N}$ telle qu'une machine $F = \phi_n \in \mathfrak{P}$ soit transformée en une machine $F' = \phi_{\alpha(n)}$ calculant la même fonction, et telle que toute machine $M = \phi_n$ soit transformée en une machine $M' = \phi_{\alpha(n)}$ telle que $M' \in \mathfrak{P}$. En termes plus concis, les points fixes de α calculent toutes les fonctions préfixes.*

◇ *Preuve.* Construisons l'algorithme de transformation. Soit $F = \phi_n$ une machine. On donne maintenant l'algorithme de calcul de $F' = \phi_{\alpha(n)}$ (la machine transformée) sur l'entrée $x = \mathsf{x}_{\langle 0\rangle}\mathsf{x}_{\langle 1\rangle}\ldots\mathsf{x}_{\langle n-1\rangle}$:

(i) On pose $z = \varepsilon$, une variable qui va servir d'entrée à la machine F ;

(ii) On pose $i = 0$ (nombre d'étapes de calcul) ;

(iii) On exécute l'étape *(iv) en donnant comme valeur à la variable z' tous les mots de longueur inférieure ou égale à i, par ordre croissant de longueur, puis par ordre lexicographique pour les mots de même longueur. Dès que l'un des calculs de l'étape (iv) donne un calcul convergent, on passe à l'étape (v), sinon on recommence en incrémentant i ;*

(iv) On répète cette étape avec plusieurs valeurs de z' comme expliqué ci-dessus. On simule $\ell(z) + i - \ell(z')$ étapes de calcul de F à partir de l'entrée $z \cdot z'$. On note ensuite si cela a donné un calcul convergent (c'est-à-dire si la machine s'est arrêtée sur l'entrée $z \cdot z'$ en moins de $\ell(z) + i$ étapes de calcul).

(v) Quatre cas se présentent, selon les valeurs de z et z'. Si $z = x$ et $z' = \varepsilon$, alors la machine s'arrête et rend $F(x)$ comme résultat. Si $\ell(z) < \ell(x)$ et que $z' = \varepsilon$, on boucle ($F'(x) = \perp$). Si $\ell(z) = \ell(x)$ et $z' \neq \varepsilon$, on boucle ($F'(x) = \perp$), et enfin si $z = x$ et $z' \neq \varepsilon$, on pose $z = x_1 \dots x_{\ell(z)+1}$, et on recommence à l'étape (v).

Si $F(u)$ converge en t étapes et $F(u \cdot v)$ converge en t' étapes, deux cas se présentent : soit $t \leqslant t' - \ell(v)$, et la machine résultante bouclera pour $u \cdot v$ (lorsque $z = u, z' = \varepsilon, i = t$), soit c'est le contraire, et dans ce cas la machine résultante boucle lors du calcul de u (lorsque $z = u, z' = \varepsilon, i = t' - \ell(v)$). Donc, si l'on prend pour un mot qui fait converger F celle de ses continuations dont le calcul se fait en le moins de (temps − longueur), on retrouve facilement la propriété « F' est préfixe » (puisque c'est soit l'un, soit l'autre qui convergera, mais pas les deux).

De plus, on peut remarquer que la fonction calculée par une machine vérifiant $\mathfrak{P}$ n'est pas modifiée par cet algorithme. Le numéro de la machine change mais la fonction calculée est inchangée.

Cette transformation est récursive, et donc toute machine de numéro i dans l'énumération se verra transformée en une autre machine $\alpha(i)$. Ainsi, les fonctions p.p.r. préfixes sont le « point fixe » de α (au sens donné dans l'énoncé). □

1.3.2 Définition de la complexité auto-délimitée

Le but de cette partie est, à l'instar de ce que l'on a fait pour l'ensemble des machines, d'essayer de trouver une mesure de complexité qui soit « universelle » pour la sous-classe des machines préfixes. L'universalité de la mesure de complexité doit se traduire par une certaine forme d'optimalité, elle doit refléter l'essentiel des avantages de toutes les autres machines de la classe. Pour cette classe, le théorème de Solomonoff-Kolmogorov est aussi vérifié, et c'est donc la notion d'optimalité additive que nous utilisons.

Toutefois, comme nos machines sont à deux arguments (l'entrée est toujours de la forme $\langle x, y \rangle$), on est amené à considérer en fait les machines qui vérifient la propriété $\mathfrak{P}'$ suivante :

Définition 10 (Propriété préfixe $\mathfrak{P}'$) Une fonction F vérifie $\mathfrak{P}'$ (ou $F \in \mathfrak{P}'$) si et seulement si pour tout triplet de mots u, v, w :

$$\left.\begin{array}{c} F\langle u, w\rangle \downarrow \\ F\langle u \cdot v, w\rangle \downarrow \end{array}\right\} \Rightarrow v = \varepsilon$$

Cette notion est très similaire à la propriété $\mathfrak{P}$(définition 9), et la même démonstration permet de montrer l'existence d'un α vérifiant le résultat suivant :

Proposition 0.10 *Il existe une fonction α de $\mathbb{N}$ dans $\mathbb{N}$ telle qu'une machine $F = \phi_n \in \mathfrak{P}'$ soit transformée en une machine $F' = \phi_{\alpha(n)}$ calculant la même fonction, et telle que toute machine $M = \phi_n$ soit transformée en une machine $M' = \phi_{\alpha(n)}$ telle que $M' \in \mathfrak{P}'$.*

Aidé de ce résultat, on montre qu'il existe une machine $\psi \in \mathfrak{P}'$ additivement optimale *pour la classe des machines* $\mathfrak{P}'$, dans l'énoncé suivant :

Théorème 6 *Il existe $\psi \in \mathfrak{P}'$, telle que :*

$$\forall \psi' \in \mathfrak{P}', \exists c_{\psi'}, \forall x, y, \quad \mathbf{K}_{\psi}(x|y) \leqslant \mathbf{K}_{\psi'}(x|y) + c_{\psi'}. \tag{1.17}$$

Cette machine vérifie l'équation (1.9) *pour la classes des machines $\mathfrak{P}'$, il existe donc une machine additivement optimale pour cette classe.*

◇ *Preuve.* On construit la machine, en posant :

$$M\left\langle \overline{\beta} \cdot p, y \right\rangle = \phi_{\alpha(\beta)} \langle p, y \rangle .$$

La fonction α est celle qui a été définie par la proposition 0.10. On s'assure aussi que si la donnée ne correspond pas à des valeurs correctement formatées (de la forme $\overline{\beta} \cdot p$), M diverge. On peut construire cette machine, car on peut séparer les trois données β, p et y puisque l'on a un codage préfixe pour β. On peut ensuite calculer $\langle p, y \rangle$, puis exécuter U sur $\alpha(\beta)$ et sur cette donnée.
On vérifie que :

$$\text{si } \begin{cases} M\left\langle \overline{\beta} \cdot u, w \right\rangle \downarrow \\ M\left\langle \overline{\beta} \cdot u \cdot v, w \right\rangle \downarrow \end{cases} \text{, alors } \begin{cases} \phi_{\alpha(\beta)} \langle u, w \rangle \downarrow \\ \phi_{\alpha(\beta)} \langle u \cdot v, w \rangle \downarrow \end{cases} ,$$

donc $v = \varepsilon$, puisque $\phi_{\alpha(\beta)}$ est préfixe. Donc M vérifie bien $\mathfrak{P}'$.
Pour toute machine $\psi' \in \mathfrak{P}'$ il existe une machine $\phi_{\alpha(\beta)}$ qui calcule la même fonction d'après la proposition 0.10, et on a alors $M\left\langle \overline{\beta} \cdot p, y \right\rangle = \psi' \langle p, y \rangle$. Donc en particulier

$$\forall x, y, \mathbf{K}_M(x|y) \leqslant \ell(\overline{\beta} \cdot p) \leqslant \ell(\overline{\beta}) + \ell(p).$$

Le théorème est donc prouvé avec $\psi = M$ et $c_{\psi'} = \ell(\overline{\beta})$. On remarquera que la constante est calculable en fonction du numéro de ψ' puisqu'il suffit de prendre la longueur auto-délimitée du code de ψ' car $\psi' = \phi_{\alpha(\beta)}$ (égalité des fonctions calculées). □

On a donc, de la même façon qu'avec la complexité de Kolmogorov, une notion de machine additivement optimale pour cette sous-classe de machines. Cette sous-classe vérifie donc le théorème de Solomonoff-Kolmogorov, c'est-à-dire la propriété de l'additivité optimale. Comme pour toutes les classes de machines vérifiant cette condition, on peut ainsi démontrer que pour tout couple de machines additivement optimales pour cette classe, la différence entre les deux reste bornée. On peut de même choisir une machine parmi toutes les machines additivement optimales pour se fixer un point de référence, et oublier

la constante qui intervient dans les comparaisons, comme pour la classe plus générale des fonctions partielles partiellement récursives. Posons les définitions suivantes :

Définition 11 La complexité auto-délimitée (dite aussi *complexité préfixe*) de x sachant y est égale à $\mathbf{K}_\psi(x|y)$ avec ψ une machine de référence préalablement fixée, additivement optimale pour la classe des machines vérifiant $\mathfrak{P}'$. On note cette complexité $\mathbf{KP}(x|y)$, le ψ étant alors sous-entendu.
La complexité auto-délimitée de x est égale à $\mathbf{KP}(x|\varepsilon)$, et est notée $\mathbf{KP}(x)$. Pour souligner le fait que ψ est une machine préfixe, il est possible de remplacer la notation $\mathbf{K}_\psi(x)$ par $\mathbf{KP}_\psi(x)$ lorsque ψ est préfixe.

1.3.3 Encadrements de la complexité auto-délimitée

Nous disposons maintenant de deux notions de complexité différentes, la complexité de Kolmogorov (dite simple) et la complexité auto-délimitée. Cette partie a pour but de donner des bornes pour la complexité auto-délimitée, en essayant en particulier de la comparer à la complexité de Kolmogorov.

Proposition 0.11

$$\mathbf{KS}(x|y) \leqslant \mathbf{KP}(x|y) \tag{1.18}$$

◇ *Preuve.* La preuve est très simple, car l'ensemble des machines vérifiant $\mathfrak{P}'$ est inclus dans l'ensemble des machines partielles partiellement récursives. Formellement, étant donné une machine ψ_p additivement optimale pour la classe des machines vérifiant $\mathfrak{P}'$, et une machine ψ additivement optimale pour la classe de toutes les machines, on peut écrire (par définition de ψ) que $\mathbf{K}_\psi(x|y) \leqslant \mathbf{K}_{\psi_p}(x|y)+c$. Étant donné l'équivalence des machines additivement optimales pour une même classe, on obtient que $\mathbf{KS}(x|y) \leqslant \mathbf{KP}(x|y) + c'$, ce qui est bien l'équation demandée. □

On peut aussi proposer un lien entre codage préfixe et complexité préfixe. Tout codage récursif induit en effet une majoration de la complexité comme suit :

Proposition 0.12 *Étant donné un code préfixe F, on a $\mathbf{KP}(x) \leqslant \ell(F(x))$.*

◇ *Preuve.* Soit la machine F^{-1} qui décode F, c'est-à-dire : $F^{-1}(F(x)) = x$ pour tout $x \in \Xi$ et $F^{-1}(y) = \perp$ si y ne s'écrit pas $F(x)$. Cette machine vérifie bien $\mathfrak{P}'$, puisque F est lui-même un code préfixe. Or $\mathbf{K}_{F^{-1}}(x) = \ell(F(x))$. Donc $\mathbf{KP}(x) \leqslant \ell(F(x))$. □

Ceci permet une majoration presque optimale de la **KP**-complexité :

Corollaire 0.13

$$\mathbf{KP}(x) \leqslant \ell(x) + \ell(\ell(x)) + \cdots + \ell(\ldots(\ell(x))\ldots) + 2\ell^\star(x),$$

où $\ell^\star(x)$ est le nombre de fois que l'on peut appliquer la fonction ℓ à un mot x sans atteindre la valeur 1.

◇ *Preuve.* On utilise l'encodage préfixe suivant :

$$e(x) = 1^{\ell^\star(x)} \cdot 0 \cdot \ell(\dots(\ell(x)\dots) \cdot \dots \cdot \ell(x) \cdot x.$$

Cet encodage est bien préfixe. Il correspond à la clôture de la suite des codes préfixes dont le premier terme serait $1^{\ell(x)}0x$, le deuxième terme serait $1^{\ell(\ell(x))}0\ell(x)x$, etc. □

La proposition suivante permet d'établir des résultats importants sur la compressibilité des chaînes via des machines préfixes. La majoration donnée est assez précise, car la borne est atteinte pour toute longueur n, comme le prouve le théorème 7.

Proposition 0.14

$$\mathbf{KS}(x) \leqslant \mathbf{KP}(x) \leqslant \mathbf{KS}(x) + \mathbf{KP}(\mathbf{KS}(x)).$$

◇ *Preuve.* La partie droite de l'inégalité est simplement la répétition de l'équation (1.18). Pour la partie gauche, on construit une machine qui calcule x. On se donne $x^\star$ vérifiant $\ell(x^\star) = \mathbf{KS}(x)$ un programme permettant de calculer x avec une machine f additivement optimale pour l'ensemble des machines p.p.r. et q un programme permettant de calculer $\ell(x^\star)$ avec une machine g additivement optimale pour l'ensemble des machines vérifiant $\mathfrak{P}'$. On construit une machine h vérifiant la propriété $\mathfrak{P}'$ qui calcule x à partir de $q \cdot x^\star$: on transforme la machine g en faisant calculer la nouvelle machine sur tous ses préfixes, dont un seul (q) donnera un résultat, ce qui permet de séparer q et $x^\star$. On construit la machine h de façon à ce qu'elle lise q, en déduise $\ell(x^\star)$, puis ne lise que les mots de la forme $q \cdot p$, avec $\ell(p) = \ell(x^\star)$; de cette façon elle vérifie $\mathfrak{P}'$. Il suffit ensuite de simuler f sur l'entrée $x^\star$ pour retrouver x. On en déduit $\mathbf{KP}(x) \leqslant \ell(qx^\star) \leqslant \mathbf{KP}(\mathbf{KS}(x)) + \mathbf{KS}(x)$. □

Il existe un équivalent de la proposition 0.7 pour la **KP**-complexité :

Théorème 7 *Soit* $\Xi_n = \{x \in \Xi, \ell(x) = n\}$. *On a alors*

$$\max\{\mathbf{KP}(x), \ell(x) = n\} = n + \mathbf{KP}(n) + \mathcal{O}(1) \tag{1.19}$$

$$\sharp\{x \in A, \mathbf{KP}(x) \leqslant n + \mathbf{KP}(n) - r\} \leqslant |\xi|^{n-r+\mathcal{O}(1)} \tag{1.20}$$

◇ *Preuve.*

$\max\{\mathbf{KP}(x), \ell(x) = n\} = n + \mathbf{KP}(n) + \mathcal{O}(1)$ On peut utiliser la même technique que précédemment. Il existe une machine préfixe M qui sur une entrée du type $p \cdot q$, avec p un codage préfixe optimum pour $\ell(q)$, donne q comme résultat. Cette machine est préfixe selon la même preuve que précédemment, et on obtient donc $\mathbf{K}_M(q) \leqslant \ell(q) + \ell(p)$. En remplaçant, si $\ell(q) = n$, comme $\ell(p) = \mathbf{KP}(n)$, on obtient $\mathbf{KP}(q) \leqslant \mathcal{O}(1) + n + \mathbf{KP}(n)$. La preuve que la valeur est atteinte vient de l'équation 1.20.

$\sharp\{x \in A, \mathbf{KP}(x) \leqslant n + \mathbf{KP}(n) - r\} \leqslant |\xi|^{n-r+O(1)}$ Cette inégalité est difficile à prouver directement. On admettra donc l'égalité suivante, qui provient de la symétrie de l'information auto-délimitée prouvée, par exemple, dans [LV97b] :

$$\mathbf{KP}(x) + \mathbf{KP}(y|x, \mathbf{KP}(x)) = \mathbf{KP}(y) + \mathbf{KP}(x|y, \mathbf{KP}(y)) + \mathcal{O}(1). \tag{1.21}$$

Pour la signification de la notation $\mathbf{KP}(y|x, \mathbf{KP}(x))$, on pourra se reporter au paragraphe 1.2.2.

On pose $y = \ell(x) = n$. $\mathbf{KP}(n|x, \mathbf{KP}(x)) = c$, puisqu'il suffit de mesurer la longueur de x pour connaître n. Si on suppose que $\mathbf{KP}(x) \leqslant n + \mathbf{KP}(n) - r$, alors $\mathbf{KP}(x|n, \mathbf{KP}(n)) \leqslant n - r + c + \mathcal{O}(1)$.

On conclut la preuve par dénombrement, au plus $|\xi|^{n-r+\mathcal{O}(1)}$ programmes étant de longueur inférieure ou égale à $n - r + \mathcal{O}(1)$, au plus $|\xi|^{n-r+\mathcal{O}(1)}$ valeurs de x peuvent donc vérifier cette égalité.

□

1.3.4 Autres propriétés de la complexité préfixe

La **KP**-complexité possède un grand nombre de propriétés, très similaires à la **KS**-complexité, et dont la preuve est identique ou en découle. On donne ici ces propriétés, car elles sont importantes pour la compréhension générale de la théorie de l'information algorithmique.

Proposition 0.15

$$\forall x, y, \mathbf{KP}(x|y) \leqslant \mathbf{KP}(x). \tag{1.22}$$

Proposition 0.16 *Il existe une fonction totale récursive $SP\langle t, x\rangle$ convergeant vers $\mathbf{KP}(x)$ quand t tend vers l'infini, telle que :*

$$\forall t \in \Xi, SP\langle t, x\rangle \geqslant SP\langle t+1, x\rangle \geqslant \mathbf{KP}(x). \tag{1.23}$$

Proposition 0.17 *Soit $m(x) = \min\{\mathbf{KP}(y), y \geqslant x\}$. Pour toute fonction partielle partiellement récursive f, monotone qui tend vers $+\infty$, pour tout $x \in \Xi$ sauf un nombre fini, on a $m(x) < f(x)$.*

Théorème 8 *La fonction $\mathbf{KP}(.)$ n'est pas partielle partiellement récursive. De plus, aucune fonction partielle partiellement récursive ne coïncide avec elle sur un ensemble récursivement énumérable infini de points.*

Les preuves sont les mêmes que pour la **KS**-complexité.

1.3.5 Complexité et sous-additivité

Une question qui se pose naturellement est de savoir comment réussir à discerner deux objets « indépendants ». La méthode naturelle, inspirée d'une démarche similaire dans

l'étude de l'entropie de Shannon, consiste à comparer la complexité de Kolmogorov de l'un des objets d'une part seul, d'autre part en ayant connaissance de l'autre objet. Cette idée amène des résultats étranges. On veut ainsi donner une mesure de ce qui dans un objet y reflète une information sur un autre objet x.

On introduit ici la notion d'*information algorithmique* par la définition suivante :

Définition 12 L'information algorithmique contenue dans x à propos de y est la quantité

$$\mathbf{IA}(x : y) = \mathbf{KS}(y) - \mathbf{KS}(y|x). \tag{1.24}$$

Cette information mutuelle présente une dissymétrie, au sens où l'information de x sur y et celle de y sur x diffèrent avec une différence qui peut être de l'ordre de grandeur du logarithme des complexités. C'est partiellement contraire à l'intuition ; même si l'égalité est en effet vérifiée au logarithme près, on aurait pu s'attendre à une transposition plus directe de l'intuition qui peut être donnée par d'autres théories, comme celle de Shannon. Cette contradiction n'est d'ailleurs pas vérifiée pour la complexité préfixe, par exemple. Elle est due à la nécessité de marquer la séparation entre les deux données, contrainte qui disparaît avec l'auto-délimitation.

Remarque 2 On peut prouver que la quantité $|\mathbf{IA}(x : y) - \mathbf{IA}(y : x)|$ peut atteindre le logarithme de $\mathbf{KS}(x)$ ou de $\mathbf{KS}(y)$. En effet, de par le théorème 0.7, pour tout n dans $\mathbb{N}$, il existe x_n tel que $\ell(x) = n$ et $\mathbf{KS}(x_n|n) \geqslant n$. De par le même théorème, il existe un ensemble A infini de n tels que $\mathbf{KS}(n) \geqslant \ell(n)$. Pour tout $n \in A$, on a $\mathbf{KS}(n|x_n) = c_1$ et $\mathbf{KS}(n) \geqslant \ell(n)$. Donc $\mathbf{IA}(x_n : n) \geqslant \ell(n) - c_1$. D'autre part, $\mathbf{KS}(x_n) \leqslant \ell(x_n) + c_2$, et $\mathbf{KS}(x_n|n) \geqslant n$. Donc

$$|\mathbf{IA}(x_n : n) - \mathbf{IA}(n : x_n)| \geqslant |\ell(n) - (c_1 + c_2)|\,. \tag{1.25}$$

Ceci se traduit asymptotiquement par

$$|\mathbf{IA}(x_n : n) - \mathbf{IA}(n : x_n)| \geqslant \log(\mathbf{KS}(n)). \tag{1.26}$$

L'exemple précédent consiste finalement à exprimer le fait que la seule information à propos d'une chaîne très aléatoire est nulle, alors que la chaîne très aléatoire contient toutes les données nécessaires pour reconstruire sa longueur. La différence dans ce cas-là est alors de l'ordre du logarithme de n.

La théorie algorithmique de l'information permet de prouver que le théorème suivant ne peut pas être amélioré (il est impossible de supprimer le terme logarithmique) :

Théorème 9 *Pour toute bijection récursive C de Ξ^2 dans Ξ :*

$$\mathbf{KS}(C(x, y)) = \mathbf{KS}(x) + \mathbf{KS}(y|x) + \mathcal{O}\left(\log \mathbf{KS}\left\langle x, y\right\rangle\right). \tag{1.27}$$

Corollaire 0.18 *Pour toute fonction p.p.r. f de Ξ^2 dans Ξ, pour tout x, y tel que $f(x, y)$ converge,*

$$\mathbf{KS}(f(x, y)) \leqslant \mathbf{KS}(x) + \mathbf{KS}(y) + \mathcal{O}\left(\max\{\log \mathbf{KS}(x), \log \mathbf{KS}(y)\}\right). \tag{1.28}$$

◇ *Preuve.* On prouve d'abord l'égalité (1.27) pour la bijection $C = \kappa$.

$\mathbf{KS}\langle x, y\rangle \leqslant \mathbf{KS}(x) + \mathbf{KS}(y|x) + \mathcal{O}(\log \mathbf{KS}\langle x, y\rangle)$ Si l'on pose p et q comme étant des solutions permettant de calculer x et « y sachant x » de longueur respective $\mathbf{KS}(x)$ et $\mathbf{KS}(y|x)$, on peut écrire $c\langle x, y\rangle$ sur l'entrée $\overline{\ell(p)} \cdot p \cdot q$. Or la longueur de cette chaîne est $\mathbf{KS}(x) + \mathbf{KS}(y|x) + 2\log \mathbf{KS}(x) + 1$. Alternativement, on peut aussi utiliser l'entrée $\overline{\ell(q)}qp$, qui est de longueur $\mathbf{KS}(x) + \mathbf{KS}(y|x) + 2\log \mathbf{KS}(y|x) + 1$.

$\mathbf{KS}\langle x, y\rangle \geqslant \mathbf{KS}(x) + \mathbf{KS}(y|x) + \mathcal{O}(\log \mathbf{KS}\langle x, y\rangle)$ Supposons que pour tout c, on ait :

$$\exists x, y, \mathbf{KS}(y|x) > \mathbf{KS}\langle x, y\rangle - \mathbf{KS}(x) + c\ell(\mathbf{KS}\langle x, y\rangle). \tag{1.29}$$

On pose ensuite les ensembles suivants :

$$\begin{aligned} A &= \{\langle u, z\rangle, \mathbf{KS}\langle u, z\rangle \leqslant \mathbf{KS}\langle x, y\rangle\} \\ A_u &= \{z, \mathbf{KS}\langle u, z\rangle \leqslant \mathbf{KS}\langle x, y\rangle\} \\ &= \{z, \langle u, z\rangle \in A\} \end{aligned}$$

D'après la proposition 0.5, si l'on donne $\mathbf{KS}\langle x, y\rangle$, A est énumérable ; et si l'on donne $\mathbf{KS}\langle x, y\rangle$ et u, alors A_u est énumérable. Or $y \in A_x$:

$$\mathbf{KS}(y|x) \leqslant \ell(|A_x|) + 2\ell(\mathbf{KS}\langle x, y\rangle) + c'. \tag{1.30}$$

Soit $e = \mathbf{KS}\langle x, y\rangle - \mathbf{KS}(x) + (c-2)\ell(\mathbf{KS}\langle x, y\rangle)$. On peut déduire des deux inégalités (1.29) et (1.30) que pour toute constante c, il existe un couple x, y vérifiant aussi $|A_x| > |\xi|^e$. Essayons de majorer $\mathbf{KS}(x)$. Étant donné $\mathbf{KS}\langle x, y\rangle$ et e, on peut énumérer les chaînes u vérifiant $|\xi|^e < |A_u|$. Cet ensemble U contient évidemment x. De plus, l'ensemble $\{\langle u, z\rangle : u \in U, z \in A_u\}$ est inclus dans A. On a de plus une majoration de $|A|$: $|A| < |\xi|^{\mathbf{KS}\langle x,y\rangle+1}$. Ainsi, puisque pour chaque u dans U on a une minoration du cardinal de A_u :

$$|U| \leqslant \frac{|A|}{|\xi|^e} \leqslant \frac{|\xi|^{\mathbf{KS}\langle x,y\rangle+1}}{|\xi|^e} \leqslant |\xi|^{\mathbf{KS}\langle x,y\rangle+1-e}$$

Puisque $x \in U$, on peut donc reconstruire x à partir de son index dans U, de e et de $\mathbf{KS}\langle x, y\rangle$. L'index de x dans U est majoré par le cardinal de U, donc :

$$\begin{aligned} \mathbf{KS}(x) &\leqslant 2\ell(\mathbf{KS}\langle x, y\rangle) + 2\ell(e) + \mathbf{KS}\langle x, y\rangle - e + c'' \\ \mathbf{KS}(x) &\leqslant 2\ell(\mathbf{KS}\langle x, y\rangle) + 2\ell(e) + \mathbf{KS}\langle x, y\rangle + c'' \\ &\quad -(\mathbf{KS}\langle x, y\rangle - \mathbf{KS}(x) + (c-2)\ell(\mathbf{KS}\langle x, y\rangle)) \\ \mathbf{KS}(x) &\leqslant (4-c)\ell(\mathbf{KS}\langle x, y\rangle) + 2\ell(e) + \mathbf{KS}(x) + c'' \\ \mathbf{KS}(x) &\leqslant (6-c)\ell(\mathbf{KS}\langle x, y\rangle) + 2\ell((c-2)\ell(\mathbf{KS}\langle x, y\rangle)) + \mathbf{KS}(x) + c'' \\ 0 &\leqslant 2\log_{|\xi|}(t) - t + c'', t = (c-6)\ell(\mathbf{KS}\langle x, y\rangle) \end{aligned}$$

t peut prendre des valeurs arbitrairement grandes ; ce qui est absurde, et prouve l'égalité.

Extension à une bijection quelconque C Soit C une bijection récursive de Ξ^2 dans Ξ. Elle peut être considérée comme une bijection B de Ξ dans Ξ composée avec $\langle\cdot,\cdot\rangle$. Tout mot x pouvant être codé par un codage optimal de $B(x)$ et l'application de B, on a donc $\mathbf{KS}(x) \leqslant \mathbf{KS}(B(x))$. La même équation s'applique à B^{-1} et à $y = B(x)$, d'où $\mathbf{KS}(B(x)) \leqslant \mathbf{KS}(x)$. Donc $\mathbf{KS}(B(x)) = \mathbf{KS}(x)$, et la généralisation de l'égalité 1.27 en suit.

$\mathbf{KS}(f(x,y)) \leqslant \mathbf{KS}(x) + \mathbf{KS}(y) + \mathcal{O}(\mathbf{KS}\langle x,y\rangle)$ D'après la proposition 0.4 et l'équation fondamentale (1.9), $\mathbf{KS}(y) \leqslant \mathbf{KS}(y|x) + c$ et $\mathbf{KS}(f(a)) \leqslant \mathbf{KS}(a) + c_f$ pour tout a, l'inégalité est donc vraie. La transformation de la complexité du couple par le plus complexe des éléments du couple vient du fait que $\mathbf{KS}\langle x,y\rangle$ est majoré par $3\mathbf{KS}(x)$ si x est le plus complexe de x et de y. En effet, un codage pour le couple est donné par la concaténation d'un codage optimal autodélimité pour x et d'un codage optimal pour y, qui est donc plus petit.

□

Corollaire 0.19

$$|\mathbf{IA}(x:y) - \mathbf{IA}(y:x)| = \mathcal{O}\left(min\{\log \mathbf{KS}(x), \log \mathbf{KS}(y)\}\right). \tag{1.31}$$

◇ *Preuve.* De l'équation (1.27), on va tirer aisément l'égalité (1.31), en remarquant que

$$|\mathbf{KS}\langle x,y\rangle - \mathbf{KS}\langle y,x\rangle| \leqslant c$$

et en le réécrivant :

$$\begin{aligned}
|\mathbf{KS}\langle x,y\rangle - \mathbf{KS}\langle y,x\rangle| &\leqslant c \\
|\mathbf{KS}(x) + \mathbf{KS}(y|x) - \mathbf{KS}(y) - \mathbf{KS}(x|y)| &\leqslant \mathcal{O}\left(\log \min\{\mathbf{KS}(x), \mathbf{KS}(y)\}\right) \\
|\mathbf{KS}(x) - \mathbf{KS}(x|y) - \mathbf{KS}(y) + \mathbf{KS}(y|x)| &\leqslant \mathcal{O}\left(\log \min\{\mathbf{KS}(x), \mathbf{KS}(y)\}\right) \\
|\mathbf{IA}(x:y) - \mathbf{IA}(y:x)| &\leqslant \mathcal{O}\left(\log \min\{\mathbf{KS}(x), \mathbf{KS}(y)\}\right)
\end{aligned}$$

□

Corollaire 0.20 *La* **KS***-complexité n'est pas sous-additive, c'est à dire que l'on n'a pas pour tout couple x, y l'inégalité suivante :*

$$\mathbf{KS}\langle x,y\rangle \leqslant \mathbf{KS}(x) + \mathbf{KS}(y). \tag{1.32}$$

◇ *Preuve.* Si **KS** était sous-additive, il serait immédiat de constater que l'on pourrait majorer la quantité $|\mathbf{IA}(x:y) - \mathbf{IA}(y:x)|$ par une constante, puisque la réécriture de $\mathbf{KS}\langle x,y\rangle - \mathbf{KS}\langle y,x\rangle$ n'ajouterait pas de terme non borné. Or on a montré qu'il existe un ensemble de couples (n, x_n) qui diffèrent par un terme non borné. Donc **KS** ne peut pas être sous-additive. □

Théorème 10 *Pour toute fonction p.p.r. c de Ξ^2 dans Ξ, pour tout x, y tel que $c(x, y)$ converge,*

$$\mathbf{KP}(c(x, y)) \leqslant \mathbf{KP}(x) + \mathbf{KP}(y). \tag{1.33}$$

◇ *Preuve.* La preuve de la sous-additivité de la **KP**-complexité est similaire à celle de la **KS**-complexité. Supposons donc que l'on ait comme précédemment, deux versions auto-délimitées p et q de programmes qui permettent de calculer x et y, de longueurs respectives $\mathbf{KP}(x)$ et $\mathbf{KP}(y)$. On fait une machine V qui lit les deux arguments et qui applique c sur le résultat de U appliquée à chacun de ces deux arguments. V est une machine vérifiant $\mathfrak{P}'$. Donc on a bien $\mathbf{KP}(c(x, y)) \leqslant \mathbf{KP}(x) + \mathbf{KP}(y)$ à une constante près. □

Chapitre 2 Notion d'aléatoire

La notion de suite aléatoire est une notion très naturelle. Si l'on fait un jeu de pile ou face avec un adversaire, et que l'on gagne vingt fois de suite, il est naturel de penser qu'à moins d'un hasard *extraordinaire*, il y a eu tricherie. Cette notion d'*extraordinaire* va nous aider à définir un évènement *aléatoire*, permettant de caractériser efficacement le degré de rareté d'un ensemble de suites ; individuellement, chaque série de lancers de pièces donne un résultat qui est tout aussi improbable qu'un autre, avec une pièce parfaitement équilibrée. Il est possible de relier cette notion d'extraordinaire à la complexité de Kolmogorov, faisant ainsi un lien fort entre la notion de compressibilité et l'entropie d'une part, et la notion de hasard d'autre part.
Nous expliquerons ensuite comment on peut appliquer ces considérations à un modèle particulier, le cas des automates cellulaires. Les automates cellulaires sont un modèle de calcul à la fois élémentaire et très complexe, dont le problème de classification *a priori* peut être vu sous un nouveau jour à l'aide de la définition de l'aléatoire.

2.1 Définition de l'aléatoire

Lorsque l'on se pose la question de savoir ce que doit être une suite aléatoire, un certain nombre de critères apparaissent : égalité approximative des fréquences de caractères, impossibilité de deviner quel est le caractère suivant, etc. La formalisation de cette théorie connut plusieurs approches dont la première fut, historiquement, l'approche *stochastique* de R. von Mises. Les deux approches exposées dans cette partie sont celles du suédois P. Martin-Löf (que l'on peut trouver dans [ML66] et [ML71]) et celles de A.N. Kolmogorov.

Présentons ces deux approches. L'approche de la représentativité (anglais *typicalness*) est celle de Martin-Löf. Elle utilise la notion de test. Un « test », dans cette optique, est une méthode de séparation des mots finis, qui donne une « note » plus ou moins élevée selon la représentativité de la suite qui est testée, mais qui ne donne pas de notes trop élevées. Par exemple, compter le nombre de 0 en tête du mot est un test acceptable. Une suite aléatoire ne doit pas pouvoir être retenue par un test : ces tests élémentaires détectent

les régularités, et ces caractères sont à exclure des suites aléatoires.

L'approche de KOLMOGOROV consiste à utiliser la notion de régularité, qui est capturée par les chaînes compressibles. Si une chaîne est compressible, il est possible de repérer des régularités qui ne conviennent pas à une suite aléatoire. Le résultat principal présenté dans cette partie est le théorème de Levin-Schnorr [ZL70, Sch71], qui établit l'équivalence de ces deux notions pour les suites finies. On s'intéresse ensuite à la caractérisation des suites *infinies* aléatoires, pour constater la différence dans les deux points de vue. Il est important de noter qu'à ce jour, ces approches de la définition de l'aléatoire sont parmi les plus convaincantes, contrairement à l'approche stochastique (cf. [USS90]).

2.1.1 L'aléatoire par la représentativité

Le premier impératif de cette approche pour manipuler la « quantité d'aléatoire » dans une donnée, est de formaliser la notion d'être *typique de n'importe quelle majorité raisonnable*, et donc en premier, d'être *typique d'une majorité*. Ce que l'on va imposer à un critère de sélection (les tests évoqués plus haut) est de retenir au moins la moitié des mots d'une certaine longueur comme étant parfaitement typiques. Un tel test dégage deux catégories : les typiques, et les atypiques. Mais cette division n'est pas assez fine pour prétendre à un caractère général. En établissant une simple dichotomie, on s'enlève la possibilité de refléter la graduation qui existe naturellement. En effet, si un seul caractère est changé dans un mot, le caractère fondamentalement aléatoire de ce mot est altéré, mais pas au point que l'on puisse trouver une limite franche, ce qui ne manquerait pas de poser problème. En revanche, on peut imaginer de caractériser une quantité plus graduée, dont la valeur diminuerait progressivement à chaque altération. C'est même la variation de cette quantité d'aléatoire qui est importante, par rapport à une référence fixée.

Pour P. MARTIN-LÖF, il est possible de raffiner le procédé en le réitérant : parmi les suites qui sont atypiques, la moitié d'entre elles peuvent être considérées comme plus typiques que les autres, moins extraordinaires, et se voient attribuer une note de 1. Les suites les plus typiques ont une note de 0, et les autres auront au moins 2. On recommence ainsi pour étaler l'échelle de notation qui va de 0 à la longueur du mot. L'étape suivante consiste à donner comme définition du degré d'aléatoire d'une valeur le degré donné par un test *maximal*, ayant une valeur qui est globalement aussi élevée qu'avec tous les autres tests.

Définition 13 Soit P une distribution récursive de probabilité sur Ξ. Une fonction totale $\delta : \Xi \to \mathbb{N}$ est un P-test si et seulement si :

(i) δ est approximable par au-dessous, *i.e.* l'ensemble $V = \{(m, x) : \delta(x) \geqslant m\}$ est un ensemble récursivement énumérable.

(ii) elle respecte la condition dite des sections critiques :

$$\forall n \in \mathbb{N}, \forall m \in \mathbb{N} \sum_{\substack{\delta(x) \geqslant m \\ \ell(x)=n}} P(x|n) \leqslant |\xi|^{-m} \tag{2.1}$$

Un P-test est donc un test qui attribue un coefficient de rareté (ou plus exactement d'atypicité) à tous les mots, mais qui ne décrète pas que tout mot est atypique. En fait, l'ensemble des mots de même longueur ayant un coefficient supérieur à m ne doit pas avoir une mesure supérieure à $|\xi|^{-m}$. Par exemple, le test δ qui consiste à compter le nombre de 0 en tête d'un mot est bien un L-test, si on pose L la distribution de probabilité uniforme. [1]

Un test δ n'est pas forcément une fonction récursive. Un test δ pourra être vu, d'un point de vue calculatoire, comme une fonction qui produit les valeurs qui sont inférieures à $\delta(x)$ sans préciser s'il s'arrête. Ce sont donc des fonctions approximables par au-dessous.

On peut se demander pourquoi on a fait ce choix plutôt que de faire le choix d'une fonction totale récursive. Une importante propriété sous-jacente de ces tests est qu'on peut les énumérer, ce qui ne serait pas le cas si on avait imposé qu'ils soient récursifs :

Proposition 0.21 *Il existe une énumération effective des P-tests.*

◇ *Preuve.* Pour faire une énumération effective, il faut associer à chaque objet une description de l'objet. La description choisie est une fonction récursive qui va énumérer les paires mot/entier (x, m) telles que $\delta(x) \geqslant m$. Cette description peut facilement être transformée en une description où une fonction récursive accepte un argument x et donne des valeurs de m inférieures à $\delta(x)$.

On part d'une énumération des fonctions p.p.r. de $\mathbb{N} \to \Xi \times \mathbb{N}$, et on transforme cette énumération effective en une autre énumération effective de fonctions p.p.r. ayant la propriété d'être définies sur tous les segments initiaux (phase 1) et ensuite en une énumération effective des P-tests sous la forme donnée plus haut (phase 2). La preuve sera alors terminée parce que l'on aura éliminé (dans la phase 2) tout et seulement ce qui n'est pas un P-test, et que tous les ensembles récursivement énumérables sont représentés avant la phase 2, en particulier ceux qui représentent les P-tests.

1. On a une énumération de toutes les fonctions p.p.r. et on veut la transformer en une énumération de fonctions définies sur les segments initiaux, c'est-à-dire que si $f(n)$ converge, alors $f(n-1)$ converge aussi. Cette transformation va être récursive, pour que ce soit bien une énumération effective. On suppose donc que l'on veut calculer $g(n)$, où g est la transformée de f. À l'étape $(n(n-1)+m+1)$, on effectue n étapes de calcul de f sur l'entrée m. Le premier calcul convergent de ces étapes de calcul est $g(0)$, le deuxième est $g(1)$, etc. Il est à noter que l'ensemble $\{g(x), x \in \operatorname{Dom} g\}$ est égal à $\{f(x), x \in \operatorname{Dom} f\}$. En particulier, tout les ensembles $\{(m, x) : \delta(x) \geqslant m\}$ pour δ un P-test sont bien énumérés.
2. On utilise l'algorithme suivant, qui va énumérer, pour δ un P-test, tous les ensembles $\{(m, x) : \delta(x) \geqslant m\}$. L'algorithme va aussi mémoriser la valeur maximale obtenue pour m dans les couples (m, x) pour chaque x. C'est faisable, car à tout moment on aura évalué un nombre fini de couples. On suppose donc que g est une fonction définie sur un segment initial, et on cherche à décrire une fonction δ.

1. La distribution uniforme de probabilité des mots sur un alphabet ξ est la fonction associant à x la probabilité $|\xi|^{-2\ell(x)}\,(|\xi|/(|\xi|-1))$.

Plus précisément, δ étant supposée être une fonction totale, on n'énumère pas les couples de la forme $(0, x)$ — il n'est pas besoin de les faire entrer dans la description.

(a) À tout moment dans l'algorithme si $\delta(x)$ n'a jamais été mis en mémoire, on suppose qu'il vaut 0. On pose $i = 0$. i va compter les couples (m, x) produits par g.

(b) On pose $i = i + 1$. On calcule ensuite $g(i-1)$ et on obtient un couple (m_i, x_i). Si $i - 1 \notin \text{Dom } g$, alors on a fini de décrire δ. Remarquons tout de suite que l'on ne sait pas de façon récursive quand la description est terminée.

(c) Si la nouvelle fonction δ où on définirait $\delta(x_i) = m_i$ n'est plus un P-test [2], alors on continue avec l'étape 2e ; sinon, on continue avec l'étape 2d.

(d) Énumérer (m_i, x_i). Poser ensuite $\delta(x_i) = m_i$ si $\delta(x_i) \leqslant m_i$. Recommencer à l'étape 2b.

(e) On ne continue pas le calcul de δ. On s'arrange pour que le graphe de δ soit cohérent, c'est à dire que pour tous les x où δ est défini, on énumère tous les couples (m, x) avec $m \leqslant \delta(x)$; puis on boucle.

Notons bien que si g est un test, alors le calcul ne s'arrête pas, et l'algorithme approximera bien δ par au-dessous. Si g diverge à un moment, alors δ reste inchangée ; et comme lorsque $i = 0$, δ est un test, une récurrence immédiate nous garantit que la fonction δ est un P-test ; si elle ne l'est pas, alors à un moment la condition de l'étape 2c nous garantit que l'on n'enregistre pas la modification finale, et que l'on ne touche plus à la définition interne de δ. □

On peut maintenant définir ce que serait un P-test universel :

Définition 14 Un test universel du caractère aléatoire au sens de Martin-Löf pour la distribution récursive de probabilité P, ou en raccourci un P-test universel, est un P-test δ_0 tel que pour tout P-test δ, il existe une constante c, telle que pour tout x, $\delta_0(x) \geqslant \delta(x) - c$.

Dans la littérature, on trouvera parfois en lieu et place du mot « universel », le mot « maximal » ou encore « optimal ».

Bien sûr, cette définition est non effective et ne prouve pas qu'il existe un tel P-test. On construit donc maintenant directement un ensemble récursivement énumérable qui est un P-test universel.

Théorème 11 *Soit $\delta_1, \delta_2, \ldots$ une énumération effective des P-tests. Alors*

$$\delta_0(x) = \max_{y \geqslant 1} \{\delta_y(x) - y\}$$

est un P-test universel.

◇ *Preuve.* On donne une description effective de δ_0, c'est-à-dire que l'on construit un algorithme énumérant tous les couples (m, x) tels que $\delta_0(x) \geqslant m$.

On applique pour cela l'algorithme suivant :

2. On peut le tester parce que la distribution de probabilité P est récursive.

1. À tout moment dans l'algorithme si $\delta_0(x)$ n'a jamais été mis en mémoire, on suppose qu'il vaut 0. On pose $i = 0$. i va servir de compteur pour les couples produits.
2. On incrémente i. On a $i = \langle n, s\rangle$. On utilise l'énumération effective des P-tests pour simuler l'algorithme énumérant δ_n pendant s pas. Si on arrive exactement à l'énumération d'un couple, on pose (m_i, x_i) égal à ce couple. Sinon on recommence à l'étape 2.
3. Si $m_i - n$ est plus petit que la valeur actuellement mémorisée pour $\delta_0(x_i)$, on recommence directement à l'étape 2.
4. On énumère tous les couples entre $(\delta_0(x_i) + 1, x_i)$ et $(m_i - n, x_i)$. On pose ensuite $\delta_0(x_i) = m_i - n$. On recommence ensuite à l'étape 2.

Cette méthode est bien constructive, on a donc une description effective de δ_0. Par construction, on vérifie aussi la condition d'énumérabilité pour les P-tests. Vérifions maintenant la condition dite des *sections critiques* qui prouvera que δ_0 est bien un P-test :

$$\forall n \in \mathbb{N}, \forall m \in \mathbb{N}, \sum_{\substack{\delta_0(x)\geqslant m \\ \ell(x)=n}} P(x|\ell(x)) \leqslant \sum_{y=1}^{\infty} \sum_{\substack{\delta_y(x)\geqslant m+y \\ \ell(x)=n}} P(x|\ell(x)) \tag{2.2}$$

$$\leqslant \sum_{y=1}^{\infty} |\xi|^{-m-y} \leqslant |\xi|^{-m} \tag{2.3}$$

Le passage de l'inégalité (2.2) à l'inégalité (2.3) se fait en invoquant la définition des sections critiques par l'équation (2.1).

Donc δ_0 est bien un P-test. Or par définition, on a $\delta_0(x) \geqslant \delta_y(x) - y$, donc δ_0 est bien un P-test universel. □

2.1.2 Le lien avec la complexité de Kolmogorov

Une des découvertes importantes, et un des sous-produits de l'étude de la théorie de l'information algorithmique, est la corrélation entre l'approche de Kolmogorov du hasard et l'approche de Martin-Löf. C'est cette égalité dans les approches qui fait que cette définition algorithmique du hasard est peut-être celle qui correspond le mieux à l'idée intuitive du hasard selon les quatre approches classiques : stochasticité, représentativité, compressibilité et imprédictabilité (c'est-à-dire pas de martingale gagnante). La stochasticité est la première approche : une suite aléatoire doit respecter des conditions de fréquence d'apparition des chiffres dans la suite et dans certaines sous-suites. La représentativité est l'approche de Martin-Löf, où une suite aléatoire doit faire partie de toute majorité raisonnable. L'approche par la compressibilité est celle de Kolmogorov, où une suite aléatoire est une suite incompressible. Enfin, l'approche par martingales est encore étudiée par Muchnik et d'autres [MSU96]. Si l'approche stochastique n'est pas satisfaisante — elle viole certaines lois statistiques (cf. [USS90]) —, la conjonction de la représentativité et

de la compressibilité est un indice encourageant, et la décision dépendra de l'égalité (problème encore ouvert) avec l'approche par martingales (une inclusion existe déjà, à savoir que les suites aléatoires au sens de l'incompressibilité sont incluses [USS90, Usp96]). On peut trouver de nombreuses références dans la littérature, en particulier [ZL70].

Pour faire le lien avec la complexité de Kolmogorov, on utilise une distribution récursive de probabilité particulière, la distribution uniforme de probabilité L. La probabilité selon L d'obtenir x est égale à $|\xi|^{-2\ell(x)-1}(|\xi|-1)$, et la probabilité conditionnelle d'obtenir x sachant sa longueur n est exactement $|\xi|^{-n}$, c'est à dire que l'on a une répartition équiprobable sur les éléments de même longueur.

Alors on vérifie une équation simple ($\mathbf{KL}(x)$ est la complexité selon la longueur) :

Théorème 12 (Martin-Löf) *La fonction $\Delta(x) = \ell(x) - \mathbf{KL}(x) - 1$ est un L-test universel.*

◇ *Preuve.* Il faut montrer que Δ remplit trois conditions : celle sur l'énumérabilité, celle sur les sections critiques (inégalité (2.1)), et celle sur l'universalité.

Énumérabilité On utilise la proposition 0.5 pour prouver que l'ensemble des (m, x) tels que $\Delta(x) \geqslant m$ est énumérable. En effet, on peut majorer aussi précisément que l'on veut $\mathbf{KL}(x)$, et donc minorer aussi précisément que l'on veut Δ.

Sections critiques Le nombre de x de longueur fixe n tels que $\mathbf{KL}(x) \leqslant k$ est majoré par $|\xi|^{k+1}$, le nombre de programmes de longueur inférieure ou égale à k. En posant $k = n - m - 1$, on a $\sharp\{x, \ell(x) = n, \Delta(x) \geqslant m\} \leqslant |\xi|^{n-m}$. En reportant ceci dans $\sum_{\substack{\Delta(x)\geqslant m \\ \ell(x)=n}} L(x|\ell(x))$, on a :

$$\sum_{\substack{\Delta(x)\geqslant m \\ \ell(x)=n}} L(x|\ell(x)) = \sum_{\substack{\Delta(x)\geqslant m \\ \ell(x)=n}} |\xi|^{-n} \leqslant |\xi|^{n-m}|\xi|^{-n} \leqslant |\xi|^{-m} \tag{2.4}$$

Donc Δ vérifie bien la condition sur les sections critiques, et Δ est donc bien un L-test.

Universalité On finit la preuve en montrant que Δ est L-test universel. Pour cela, on construit une définition de x qui fera que $\mathbf{KL}(x) \leqslant \ell(x) - \delta_y(x) - 1 + c(y)$ pour tout y et tout x. On aura ainsi $\Delta(x) \geqslant \delta_y(x) - c(y)$. Définissons d'abord l'ensemble Z des objets de même longueur que x (puisqu'on la connaît) et tels que $\delta_y(z) \geqslant \delta_y(x)$ pour tout élément de l'ensemble. Par définition d'un L-test, on peut énumérer cet ensemble si on connaît $\delta_y(x)$ et, bien sûr, y et $\ell(x)$. x appartient à cet ensemble, et on peut donc le décrire par son index dans cet ensemble (appelons-le j), y, $\delta_y(x)$ et $\ell(x)$. Revenons sur j : il peut être majoré par $|\xi|^{\ell(x)-\delta_y(x)}$, puisque l'on a :

$$\left.\begin{array}{l} \forall z \in \Xi, L(z|n) = |\xi|^{-n} \\ \displaystyle\sum_{\substack{\delta_y(z)\geqslant\delta_y(x) \\ \ell(z)=n}} L(z|n) \leqslant |\xi|^{-\delta_y(x)} \end{array}\right\} \Rightarrow |Z| \leqslant |\xi|^{-\delta_y(x)}|\xi|^{n} \tag{2.5}$$

Donc j est majoré, et on peut écrire une chaîne s de taille exactement $\ell(x)-\delta_y(x)+1$, qui commence par des 0 en nombre adéquat (éventuellement aucun), puis un 1, puis l'écriture de j. Ainsi, à partir de s et $\ell(x)$, on peut retrouver $\delta_y(x)$ (en comptant le nombre de 0 en tête de s) et j. Si on ajoute y, on peut donc retrouver x. On peut donc retrouver x sachant $\ell(x)$ et la chaîne $\overline{y}\cdot s$. On en déduit que $\mathbf{KS}(x|\ell(x)) \leqslant \ell(\overline{y}\cdot s)$; ce qui se développe en $\mathbf{KL}(x) \leqslant \ell(x) - \delta_y(x) + 2\ell(y) + 2$. En posant $c(y) = 2\ell(y) + 3$, on répond bien à ce qui était demandé.

□

Toutefois, cette définition n'est pas satisfaisante parce qu'elle dépend fortement de la distribution uniforme L, qui est très particulière. La proposition suivante essaye de poser un test universel qui ne dépendrait pas si fortement de la distribution de probabilité admise. La preuve de ce théorème est omise ; mais l'on montrera par la suite pourquoi le cas ci-dessus n'en était qu'un cas particulier.

Ce qui est intéressant de voir, c'est que l'on ne va plus utiliser **KS**, mais une variante de **KP**. On arrive aux limites des possibilités de la complexité simple de Kolmogorov, qui n'est pas elle capable d'exprimer cette nuance (à ce jour).

Définition 15 On pose $\overline{\mathbf{KP}}(x;k|y) = \min\{i, \mathbf{KP}(x|k-i,y) \leqslant i\}$.

Proposition 0.22 *$\delta_P(x) = -\log P(x|\ell(x)) - \overline{\mathbf{KP}}(x; -\log P(x|\ell(x))|\ell(x))$ est un P-test universel.*

◇ *Preuve.* La preuve se trouve dans [Gác78]. □

Bien qu'on n'utilise plus **KS**, le test défini dans la proposition précédente se confond avec celui du théorème 12 si on prend L comme distribution. On peut en effet montrer la relation suivante entre les complexités **KP** et **KS** :

Proposition 0.23 *À une constante additive près, les égalités suivantes sont vérifiées :*

$$\mathbf{KS}(x|y) = \min\{i, \mathbf{KP}(x|i,y) \leqslant i\} = \mathbf{KP}(x|\mathbf{KS}(x|y), y). \tag{2.6}$$

En particulier, on a $\mathbf{KS}(x) = \mathbf{KP}(x|\mathbf{KS}(x))$.

◇ *Preuve.* La preuve se fait en quatre temps :

$\underline{\mathbf{KS}(x|y) \geqslant \mathbf{KP}(x|\mathbf{KS}(x|y), y)}$ Soit $x^\star$ un mot dont la longueur est $\mathbf{KS}(x|y)$ qui code x, c'est-à-dire tel que $\Phi\langle x^\star, y\rangle = x$. Soit la machine préfixe P qui sur une entrée $\langle p, \langle q, y\rangle\rangle$ lit exactement q bits de p et fait le calcul de $\Phi\left\langle \mathsf{p}_{\langle 1\rangle} \ldots \mathsf{x}_{\langle q\rangle}, y\right\rangle$. Cette machine est préfixe — elle vérifie bien la propriété $\mathfrak{P}'$ —, qui calcule x sur l'entrée $\langle x^\star, \langle \mathbf{KS}(x|y), y\rangle\rangle$. D'après l'inégalité fondamentale (1.9), on a donc $\mathbf{KP}(x|\mathbf{KS}(x|y), y) \leqslant \mathbf{KS}(x|y) + \mathcal{O}(1)$;

$\underline{\mathbf{KS}(x|y) \geqslant \min\{i, \mathbf{KP}(x|i,y) \leqslant i\}}$ Avec le résultat précédent, on a réussi à montrer que $\mathbf{KP}(x|\mathbf{KS}(x), y) \leqslant \mathbf{KS}(x|y) + \mathcal{O}(1)$. Donc $\mathbf{KS}(x|y)$ est dans l'ensemble $\{i, \mathbf{KP}(x|i,y) \leqslant i\}$. D'où l'inégalité ;

$\underline{\mathbf{KS}(x|y) \leqslant \min\{i, \mathbf{KP}(x|i,y) \leqslant i\}}$ La première étape consiste à prouver que $\mathbf{KP}(x|i,y) \leqslant i$ implique $\mathbf{KS}(x|y) \leqslant i + \mathcal{O}(1)$. C'est suffisant pour démontrer le résultat demandé. Soit $x^{\star(i)}$ le plus petit programme pour la description préfixe de x sachant i et y. Par hypothèse, $\ell(x^{\star(i)}) \leqslant i$. Soit la machine (pas nécessairement préfixe) T qui fait le calcul de la machine de référence préfixe sur l'entrée $\langle t, \langle n-1, y\rangle\rangle$ lorsqu'on lui présente une entrée de longueur n de la forme $0^{i-\ell(t)}1t$. Si on pose $t = x^{\star(i)}$, on a bien une description de x, donc $\mathbf{KS}(x|y) \leqslant i + \mathcal{O}(1)$ (car la machine T ne dépend pas de i) ;

$\underline{\mathbf{KS}(x|y) \geqslant \mathbf{KP}(x|\mathbf{KS}(x|y), y)}$ On applique la même preuve que précédemment en posant $i = \mathbf{KS}(x,y)$, ce qui finit la preuve. On a bien la condition $\ell(x^\star) \leqslant \mathbf{KS}(x,y) + \mathcal{O}(1)$.

Cette preuve vient de [lev76]. □

Corollaire 0.24 *Le test universel δ_L correspondant à la distribution uniforme L défini ci-dessus correspond au test universel Δ.*

◇ *Preuve.* En appliquant l'équation (2.6), à y=k, sachant que $\mathbf{KS}(x|y,z) = \mathbf{KS}(x|z-y,z)$, l'égalité suivante est vérifiée à une constante additive près :

$$\overline{\mathbf{KP}}(x;k|k) = \mathbf{KS}(x|k).$$

Pour $P = L$, $-\log P(x|\ell(x)) = \ell(x)$, et donc à une constante additive près,

$$\begin{aligned}\delta_L &= \ell(x) - \overline{\mathbf{KP}}(x;\ell(x)|\ell(x)) \\ &= \ell(x) - \mathbf{KS}(x|\ell(x)) \\ &= \ell(x) - \mathbf{KL}(x) = \Delta(x)\end{aligned}$$

□

Remarque 3 Il est normal que cette égalité soit retrouvée : deux tests universels sont au plus distants d'une fonction bornée. Or les égalités précédentes sont vraies à constante près.

2.2 Caractère aléatoire des suites infinies

2.2.1 Problématique des mots infinis

L'ensemble de ce que l'on vient de voir s'applique en fait aux suites finies de mots. Mais, comme l'explique la remarque page 42 dans la section 2.1.1, le fait de pouvoir définir clairement la qualité d'être aléatoire pour une suite finie n'apparaît pas possible. Toutefois, cette séparation semble beaucoup plus logique si on considère une source de lettres. Une source est, dans le cadre de cette étude, une façon de produire une suite (potentiellement) infinie de caractères, étant eux-mêmes en nombre finis. Martin-Löf s'est aussi attaqué

à ce problème, et a défini une notion de P-test séquentiel qui permet de caractériser les sources aléatoires.

Il existe un lien entre la **KS**-complexité et l'aléatoire au sens de Martin-Löf, et une des premières choses à vérifier est de voir s'il est possible de transposer de façon simple la notion de P-test. Or, il s'avère que la notion qui paraît naturelle pour une source aléatoire, à savoir regarder toutes les sorties finies de la source et d'en déduire le caractère aléatoire de la source ne convient pas dans le cas général.

Pour ce, il va d'abord falloir une notation pour les sections initiales d'une source infinie. Une source étant une suite infinie de 0 et de 1, on l'identifiera à un élément de $\{0,1\}^\omega$:

Définition 16 Une source infinie $\mathbf{u}$ est un élément appartenant à ξ^ω, les suites infinies de caractères pris dans l'alphabet ξ. On définit les sections initiales $u_{1:n}$ comme étant les n premières lettres de la source ($\ell(u_{1:n}) = n$). L'ensemble $\{0,1\}^\omega$ est noté Ω.

Pour étudier ces suites infinies de façon récursive, il est souvent nécessaire de faire appel à leurs sections initiales. De façon équivalente, on peut avoir besoin de désigner l'ensemble des suites infinies correspondant à une même section initiale. On parle alors du cylindre, ou du cône, engendré par un mot x :

Définition 17 Le cylindre $\lhd x$ associé à un mot x est l'ensemble de toutes les suites infinies de Ω qui commencent par x.

On verra par la suite que l'on peut adopter plusieurs autres représentations pour les éléments de Ω. On peut en particulier les assimiler aux réels de $[0,1]$.

2.2.2 Caractérisation de Martin-Löf

La proposition suivante donne un minorant de l'amplitude des oscillations. On s'intéresse aux fonctions (presque) totales dont la série de terme général $|\xi|^{-f(n)}$ diverge (par exemple, les fonctions constantes et logarithmiques).

Proposition 0.25 *Soit f une fonction récursive totale de $\mathbb{N}\backslash\{0\}$ dans $\mathbb{N}$ dont la série associée $\sum_{n=1}^{\infty} |\xi|^{-f(n)}$ diverge. Alors :*

$$\forall \mathbf{u} \in \Omega, \exists A \subset \mathbb{N}, |A| = \infty, \forall n \in A, \mathbf{KL}(u_{1:n}) \leqslant n - f(n).$$

◇ *Preuve.* Dans cette preuve, on dénote par $\hat{f}$ la fonction associée à f qui vaut :

$$\hat{f}(n) = \sum_{i=1}^{n} |\xi|^{-f(i)}. \tag{2.7}$$

On construit d'abord une fonction g qui est plus grande que f, dont la différence avec f croît arbitrairement, mais telle que $\hat{g}$ soit toujours divergente. Cette augmentation permet de se débarrasser d'un terme constant par la suite. On ajoute donc à f la valeur $S(n) = \left\lfloor \log_{|\xi|} \hat{f}(n) \right\rfloor$. On pose alors :

$$g(n) = f(n) + S(n) = f(n) + \left\lfloor \log_{|\xi|} \hat{f}(n) \right\rfloor.$$

$\lim_{n\to\infty} \hat{g}(n) = \infty$ Montrons que $\hat{g}$ est toujours une fonction divergente. On va d'abord minorer $\sum_{S(n)=m} |\xi|^{-f(n)}$. Remarquons bien que $S(n) = m$ tant que l'on s'assure que $|\xi|^m \leqslant \hat{f}(n) < |\xi|^{m+1}$, c'est-à-dire que :

$$|\xi|^m \leqslant \sum_{i=1}^{n} |\xi|^{-f(n)} < |\xi|^{m+1}. \tag{2.8}$$

Soit n_0 le plus petit n vérifiant (2.8), et n_1 le plus grand.

$$\hat{f}(n_0 - 1) < |\xi|^m \tag{$*$}$$
$$\hat{f}(n_1 + 1) > |\xi|^{m+1} \tag{$**$}$$

En combinant $(*)$, $(**)$ et (2.7), on obtient :

$$\hat{f}(n_1 + 1) - \hat{f}(n_0 - 1) \geqslant |\xi|^{m+1} - |\xi|^m - |\xi|^{-f(n_1+1)}$$
$$\sum_{i=n_0}^{n_1} |\xi|^{-f(n)} \geqslant (|\xi| - 1)|\xi|^m - 1$$

On peut maintenant vérifier que $\hat{g}$ diverge toujours :

$$\sum_{n\geqslant 1} |\xi|^{-g(n)} = \sum_{n\geqslant 1} |\xi|^{-(f(n)+S(n))}$$
$$\lim_{n\to\infty} \hat{g}(n) = \sum_{m\geqslant 1} \sum_{S(n)=m} |\xi|^{-f(n)-m}$$
$$\lim_{n\to\infty} \hat{g}(n) \geqslant \sum_{m\geqslant 1} |\xi|^{-m}((|\xi| - 1)|\xi|^m - 1)$$
$$\lim_{n\to\infty} \hat{g}(n) \geqslant \sum_{m\geqslant 1} (|\xi| - 1) - \sum_{m\geqslant 1} |\xi|^{-m}$$
$$\lim_{n\to\infty} \hat{g}(n) = \infty$$

$\mathbf{KL}(u_{1:n}) \leqslant n - f(n)$ infiniment souvent On va utiliser une interprétation géometrique pour montrer la proposition. On projette pour cela les éléments de Ω sur le cercle complexe de rayon 1, ou bien de façon équivalente sur les réels modulo 1. On munit pour cela ξ d'une bijection sur $\{0, \ldots, |\xi| - 1\}$ et on associe un élément $\mathbf{u}$ à la valeur $\sum \mathsf{u}_{\langle i\rangle} |\xi|^{-i}$, que l'on peut noter $0.\mathbf{u}$. On peut faire de même pour tout mot fini et associer x à l'ensemble $\lhd x$ $\Gamma_x = \left[0.x, 0.x + |\xi|^{-\ell(x)}\right[$, qui est l'union de toutes les mots infinis commençant par x. On pose enfin en relation avec g les deux familles d'intervalles suivantes :

$$\Gamma_x = \left[\sum \mathsf{x}_{\langle i\rangle} |\xi|^{-i}, \sum \mathsf{x}_{\langle i\rangle} |\xi|^{-i} + |\xi|^{-\ell(x)}\right[$$
$$I_n \equiv [\hat{g}(n-1), \hat{g}(n)[\mod 1$$
$$A_n = \{x \in \Xi, \ell(x) = n, \Gamma_x \cap I_n \neq \emptyset\}$$

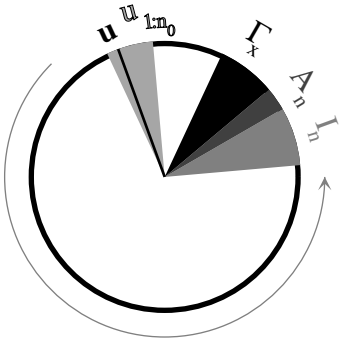


FIGURE 2.1 – Interprétation géométrique

Géométriquement (figure 2.1), on dispose sur un cercle un trajet (I_n) qui, de par la divergence de $\hat{g}(n)$, parcourt un nombre infini de fois le cercle complexe unité. De plus, on associe chaque mot fini à une tranche du cercle Γ_x. A_n contient l'ensemble de toutes les continuations possibles de longueur n associées à I_n. Supposons qu'il existe un nombre fini de n tels que $u_{1:n} \in A_n$. Cela voudrait dire que I_n n'intersecte plus l'intervalle $\Gamma_{u_{1:n}}$, à partir d'un certain n_0. Or $\hat{g}$ divergeant, I_n fait un nombre de fois infini le tour du cercle, donc $0.\mathbf{u}$ est dans une infinité de I_n, et intersecte donc tout intervalle $\Gamma_{u_{1:n_0}}$ une infinité de fois. C'est absurde, donc il existe $A \subset \mathbb{N}$ de cardinal infini, tel que pour tout $n \in A$, $u_{1:n} \in A_n$. En décrivant $u_{1:n}$ par son indice dans l'ensemble A_n, qui est borné par la largeur de I_n divisée par la largeur d'un mot de longueur n, on obtient :

$$\begin{aligned}
\mathbf{KL}(u_{1:n}) &\leqslant \log_{|\xi|} |A_n| + \mathcal{O}(1) \\
&\leqslant \log |\xi| \frac{|\xi|^{-g(n)}}{|\xi|^{-n}} + \mathcal{O}(1) \\
&\leqslant n - g(n) + \mathcal{O}(1)
\end{aligned}$$

et à cause de la divergence de S(n) qui dépasse tout $\mathcal{O}(1)$:

$$\mathbf{KL}(u_{1:n}) \leqslant n - f(n).$$

□

Ce résultat répond déjà à notre interrogation initiale : en posant $f(n) = c$, on constate qu'il n'existe pas de suite dont tous les préfixes sont c-incompressibles, quelque soit la constante c. Nous pouvons même montrer un résultat plus fort, qui porte alors sur la **KS**-complexité, et non pas sur la complexité sachant la longueur. On pose pour cela une condition supplémentaire sur f,[3] qui est que l'on peut retrouver n à partir de $n - f(n)$ par

3. MARTIN-LÖF a montré que l'on pouvait même se passer de cette condition ([ML], cité par [ML71] et [LV97a])

un programme de taille constante. C'est alors la courbe $\mathbf{KS}(u_{1:n})$ qui descend un nombre infini de fois sous la courbe de $n - f(n)$.

Corollaire 0.26 *Soit f une fonction telle que la série $\sum_n |\xi|^{-f(n)}$ diverge et que $\mathbf{KS}(n|n - f(n)) = \mathcal{O}(1)$ (comme par exemple $f(n) = \log n$). Alors pour une infinité de n, $\mathbf{KS}(u_{1:n}) \leqslant n - f(n)$.*

◇ *Preuve.* On reprend les notations de la preuve de la proposition 0.25. Soit p une description de $u_{1:n}$ sachant n, de longueur minimale $n - f(n) + \mathcal{O}(1)$. Soit q un programme qui calcule n à partir de $n - f(n)$. D'après un argument déjà utilisé, pour n suffisamment grand, $\ell(\overline{q} \cdot p) < n - f(n)$. Donc on peut compléter $\overline{q} \cdot p$ en $\overline{q} \cdot 0^{n-f(n)-\ell(\overline{q}\cdot p)-1} \cdot 1 \cdot p$. Ce mot de longueur $n - f(n)$ est bien une description de $u_{1:n}$ car on peut d'abord retrouver q de $\overline{q}$, en déduire n puisque la longueur du mot est $n - f(n)$ et avec p en déduire $u_{1:n}$. Ce qui prouve le corollaire. □

En conclusion, il n'existe pas, à cause des oscillations de la complexité, de suite dont toutes les sections initiales soient de haute complexité, c'est-à-dire c-incompressibles. La définition qui semblait pourtant naturelle d'une suite infinie aléatoire — toutes ses sections initiales soient c-incompressibles — ne peut pas être utilisée.

Il faut donc chercher un autre moyen de quantifier cette quantité d'aléatoire, en essayant de mieux capturer la notion d'infini qui est liée à la notion d'espace continu. On retrouvera ainsi les associations faites dans les preuves précédentes, qui considèrent un mot fini comme le cône de toutes les continuations possibles.

On peut alors se donner une mesure de probabilité sur l'espace ainsi défini. On pose, à l'instar de la *distribution* uniforme de probabilité L, la *mesure* uniforme de probabilité $\lambda(\prec x) = |\xi|^{-\ell(x)}$. Dans ces mesures, on peut aussi caractériser les mesures récursives (qui sont calculables par un système acceptable de programmation).

Définition 18 Soit μ une mesure récursive de probabilité de Ω. Une fonction totale δ de Ω dans $\mathbb{N} \cup \{\infty\}$ est un μ-test séquentiel si :

(i) $\delta(\mathbf{u}) = \sup_{n\in\mathbb{N}}\{\gamma(u_{1:n})\}$, où γ est une fonction approximable par au-dessous ;

(ii) $\mu\{\mathbf{u}, \delta(\mathbf{u}) \geqslant m\} \leqslant |\xi|^{-m}$, pour tout m positif.

On notera $\mathcal{V}$ l'ensemble de tous les μ-tests séquentiels.

Un μ-test séquentiel est donc l'équivalent exact d'un P-test, sauf que l'on ne considère plus les mêmes objets. Le théorème qui suit utilise d'ailleurs exactement les mêmes arguments pour démontrer qu'il existe un μ-test séquentiel qui soit additivement optimal.

Proposition 0.27 *Il existe un μ-test séquentiel universel δ_μ, c'est-à-dire qui vérifie la condition :*

$$\forall \delta \in \mathcal{V}, \exists c \in \mathbb{N}, \forall \mathbf{u} \in \Omega, \delta_\mu(\mathbf{u}) \geqslant \delta - c.$$

◇ *Preuve.* La preuve est exactement identique à la preuve utilisée pour les P-tests, proposition 0.21 et théorème 11. En effet, l'existence d'une énumération effective des μ-tests

séquentiels s'obtient par la même méthode (à partir des énumérations des fonctions approximables par au-dessous) en changeant simplement le test de fin à la phase 2c de la preuve de la proposition 0.21 ; il est en effet possible en temps borné de vérifier si une fonction à support fini est un μ-test séquentiel. Ensuite, on définit $\delta_\mu(\mathbf{u}) = \sup_{i\in\mathbb{N}}\{\delta_i(\mathbf{u}) - i\}$. δ_μ est bien approximable par au-dessous, puisque chacun des δ_i l'est ; et pour la deuxième condition, on vérifie que :

$$\begin{aligned}\mu\{\mathbf{u}, \delta_\mu(\mathbf{u}) \geqslant m\} &\leqslant \sum_{i\geqslant 1} \mu\{\mathbf{u}, \delta_i(\mathbf{u}) \geqslant m+i\} \\ &\leqslant \sum_{i\geqslant 1} |\xi|^{-m-i} \leqslant |\xi|^{-m}\end{aligned}$$

Enfin la démonstration de l'universalité est évidente si on regarde la définition de δ_μ ; en effet,

$$\forall i \in \mathbb{N}, \forall \mathbf{u} \in \Omega, \delta_\mu(\mathbf{u}) \geqslant \delta_i - i,$$

ce qui conclut la preuve. □

On peut donner une autre définition de ce qu'est un μ-test séquentiel. Cette définition est assez naturelle, et apporte certains avantages. C'est la définition basée sur les ensembles de mesure constructivement nulle :

Définition 19 On appelle ensemble de mesure constructivement nulle relativement à la mesure μ un ensemble $\mathbf{A} \subset \Omega$ tel qu'il existe une fonction calculable $f\langle .,.\rangle$ de $\mathbb{N}\times\mathbb{N} \to \Xi$ vérifiant :

(i) $\mu(\bigcup_{j=0}^{\infty} \lhd f(i,j)) \leqslant |\xi|^{-i}$

(ii) $\mathbf{A} \subset \bigcap_i \bigcup_j \lhd f(i,j)$

On dit qu'une suite infinie $\mathbf{u} \in \Omega$ passe le test associé à A si et seulement si $\mathbf{u} \notin A$.

La suite des cylindres (on adopte la représentation géométrique des suites infinies par le segment $[0,1]$) est en fait réunie par niveaux (on laisse la valeur de i constante), et la mesure de cette union ne doit pas dépasser une certaine valeur, qui est arbitrairement petite. La notion de « constructivement nul » est différente de la notion d'être de mesure nulle par le fait que la progression vers la mesure nulle doit être faite par l'intermédiaire d'une fonction énumérable.

L'avantage de cette définition est qu'elle enlève la notion de note et de graduation, pour ne plus laisser que la différence entre les suites régulières à l'infini et les suites qui n'ont qu'une quantité finie de régularité.

Dans cette optique, on peut redéfinir ce qu'est un μ-test universel. C'est l'ensemble de toutes les suites qui passent tous les tests. C'est donc toute suite qui n'appartient pas à l'union de tous les ensembles constructivement nuls. Cet ensemble, union de toutes les suites qui ne passent pas les tests, est appelé l'ensemble maximal de mesure constructivement nulle. Martin-Löf a montré que ces deux définitions étaient équivalentes, c'est-à-dire qu'une suite passant un test séquentiel universel est un test qui n'appartient pas à l'ensemble maximal de mesure constructivement nulle.

On a donc désormais une notion solide de ce que peut être un μ-test séquentiel, et Martin-Löf a alors posé la définition d'un élément μ-aléatoire. Un μ-test séquentiel détecte jusqu'à quel point on peut trouver une régularité dans une suite infinie $\mathbf{u}$, simplement en regardant progressivement la chaîne $u_{1:n}$, de façon à s'arrêter dès qu'il y a une rupture de régularité. Si on ne s'arrête pas, on dit alors que $\mathbf{u}$ échoue au test δ. L'existence d'un μ-test séquentiel universel, qui en quelque sorte majore tous les tests prouve que l'on peut faire passer à une suite $\mathbf{u}$ tous les tests en même temps. Il est alors naturel d'appeler non-aléatoires les suites présentant une quelconque régularité, et de décréter que toutes les autres suites sont μ-aléatoires.

Définition 20 Une suite $\mathbf{u} \in \Omega$ est μ-aléatoire (au sens de Martin-Löf) si et seulement si $\delta_\mu(\mathbf{u}) < \infty$.

Il est à noter que cette définition ne dépend absolument pas du choix du μ-test séquentiel universel. En effet, il se distinguent tous au plus par un terme borné, dépendant des deux μ-tests universels choisis.

Une idée générale est que presque toutes les suites sont aléatoires. Déjà précédemment, on avait retrouvé cette idée (puisque presque toutes les suites sont incompressibles comme expliqué au théorème 0.7). On retrouve ce fait, avec une justification encore plus absolue, lorsque l'on qualifie les suites infinies de Ω :

Proposition 0.28 *L'ensemble de tous les éléments de Ω qui sont μ-aléatoires est de mesure 1 (relativement à μ).*

◇ *Preuve.* On utilise un argument simple. Pour tout μ-test séquentiel δ, on a par définition :

$$\mu\left(\bigcap_{m\geqslant 1}\{\mathbf{u}, \delta(\mathbf{u}) \geqslant m\}\right) = 0.$$

Ceci correspond à la définition de ce que l'on appelle un ensemble *constructivement nul*. Donc δ_μ l'est aussi, ce qui conclue la preuve puisque cet ensemble est exactement l'ensemble des $\mathbf{u}$ qui ne sont pas μ-aléatoires.

□

2.2.3 Caractérisation par la complexité de Kolmogorov

On veut maintenant caractériser la notion d'aléatoire par la complexité. Deux approches vont suivre — une pour la **KS**-complexité et une pour la **KP**-complexité —, mais elles ne s'intéresseront qu'au cas du λ-aléatoire. En effet, il semble que l'on ne pourra pas obtenir directement une relation simple entre aléatoire et **KS**-complexité, ou **KP**-complexité si on a des probabilités biaisées d'obtenir un caractère plutôt qu'un autre. En même temps, la mesure uniforme (ou mesure de Lebesgue) sur Ω apparaît moins sélective que pouvait apparaître la distribution uniforme sur Ξ, car il n'y a plus qu'équiprobabilité *pour les mots de même longueur*, et plus de quantification sur la longueur (on fixe $P(x|\ell(x)) = |\xi|^{-\ell(x)}$ sans donner de contraintes sur $P(x)$). La **KS**-complexité ne mène pas directement à une

caractérisation convenable ; en revanche, la **KP**-complexité donne une relation élégante et immédiate entre les suites **KP**-incompressibles et les suites λ-aléatoires.

Proposition 0.29 *Soit f une fonction telle que $\sum_n |\xi|^{-f(n)}$ soit une série récursivement convergente (c'est-à-dire que c'est une série convergente et qu'il existe une fonction récursive g telle que le reste de la série à partir du $g(m)$-ème terme est majoré par $|\xi|^{-m}$). Si $\mathbf{u}$ est λ-aléatoire, alors $\mathbf{KS}(u_{1:n}|n) \geqslant n - f(n)$ à partir d'un certain rang.*

◇ *Preuve.* On suppose que f donne naissance à une série récursivement convergente, et on veut montrer que les séquences λ-aléatoires vérifient la condition $\mathbf{KS}(u_{1:n}|n) \geqslant n - f(n)$ à partir d'un certain rang. Pour cela, on construit un λ-test séquentiel qui ne sera passé que par les éléments de Ω qui vérifient la condition du théorème. Comme les suites aléatoires passent tous les λ-tests séquentiels, elles passeront en particulier celui-ci.
On pose pour tout $\mathbf{u}$,

$$\delta(\mathbf{u}) = \sup\{m, \exists n \geqslant g(m+1), \mathbf{KS}(u_{1:n}|n) \leqslant n - f(n)\}.$$

Expliquons un peu la construction, avant d'en vérifier la validité. À tout moment, l'ensemble $\mathbf{V}_m$ des suites telles que $\delta(\mathbf{u}) \geqslant m$ est construit comme l'union des $\lhd x$ pour tous les $x = u_{1:n}$ tels que $\mathbf{u}$ et n qui satisfassent les conditions du théorème. Ces ensembles vont rejeter toutes les suites $\mathbf{u}$ qui présenteront comme particularité d'avoir une complexité qui n'est pas assez élevée. Prouvons donc que c'est un λ-test séquentiel. De par la proposition 0.5, et parce que f donne naissance à une série *récursivement* convergente, $\mathbf{V}_m$ est récursivement énumérable pour tout m. Pour chaque n, le nombre de mots de **KS**-complexité inférieure ou égale à $n - f(n)$ est majoré par $|\xi|^{n-f(n)+1}$, donc :

$$\begin{aligned}
\lambda(\mathbf{V}_m) &\leqslant \sum_{n \geqslant g(m+1)} \sum_{\substack{\ell(x)=n \\ \mathbf{KS}(x|n) \leqslant n-f(n)}} \lambda(\lhd x) \\
&\leqslant \sum_{n \geqslant g(m+1)} |\xi|^{n-f(n)+1} |\xi|^{-n} \\
&\leqslant |\xi| \sum_{n \geqslant g(m+1)} |\xi|^{-f(n)} \leqslant |\xi|^{-m}.
\end{aligned}$$

Notre test est donc bien un λ-test séquentiel, ce qui conclut cette preuve. □

Proposition 0.30 *Soit $\mathbf{u} \in \Omega$.*

1. ***(Lemme de la longueur)*** *Soit $\mathbf{u} \in \Omega$. Il existe une constante positive c telle que $\mathbf{KS}(u_{1:n}) \geqslant n - c$ pour une infinité de n si et seulement s'il existe une constante positive c telle que $\mathbf{KL}(u_{1:n}) \geqslant n - c$ pour une infinité de n ;*
2. ***(Lemme de la complexité)*** *S'il existe une constante c telle que pour une infinité de n $\mathbf{KS}(u_{1:n}) \geqslant n - c$, alors $\mathbf{u}$ est λ-aléatoire ;*
3. ***(Lemme de la densité)*** *L'ensemble des $\mathbf{u} \in \Omega$, tel qu'il existe c et une infinité de n tel que $\mathbf{KS}(u_{1:n}) \geqslant n - c$ est de λ-mesure 1.*

◇ *Preuve.*

Lemme de la longueur Ce petit lemme permet en fait de renforcer les énoncés suivants ; on utilisera alors dans la preuve la condition $\mathbf{KL}(u_{1:n}) \geqslant n - c$ au lieu de $\mathbf{KS}(u_{1:n}) \geqslant n - c$. Le sens de l'implication est le seul à prouver, l'autre découlant de la proposition 0.4.

On utilise une méthode de *padding*. On va construire une description de $u_{1:n}$ en se servant de la longueur du programme minimal comme d'une aide supplémentaire. Soit un mot $x^\star$ de longueur $\mathbf{KL}(u_{1:n})$ qui est un programme pour calculer $u_{1:n}$ sachant n. Soit $v = n - \mathbf{KL}(u_{1:n})$. Le mot $\overline{v} \cdot x^\star$ peut être décodé de la façon suivante : on extrait v et $x^\star$, on ajoute à v la longueur de $x^\star$, on en déduit donc n, et on peut ainsi en déduire $u_{1:n}$ par la machine de référence.

$$\mathbf{KS}(u_{1:n}) \leqslant 2\ell(n - \mathbf{KL}(u_{1:n})) + \mathbf{KL}(u_{1:n}) + c_1. \tag{2.9}$$

Supposons que $\mathbf{u}$ vérifie les conditions de l'énoncé. On peut donc majorer $A = n - \mathbf{KL}(u_{1:n})$ par $c + c_1 + 2\ell(n - \mathbf{KL}(u_{1:n}))$ pour tous les n tels que $\mathbf{KS}(u_{1:n}) \geqslant n - c$. On a donc $A \leqslant c + c_1 + 2\ell(A)$. Et cela n'est possible que si A est borné, donc si $\mathbf{KS}(u_{1:n}|n) \leqslant n - c_2$.

Lemme de la complexité Un μ-test séquentiel universel est défini par une fonction approximable par au-dessous γ_μ. Il est alors facile de voir que si on choisit μ égal à λ (la mesure de Lebesgue), γ_λ est aussi un L-test universel (avec L la distribution uniforme de probabilité). En effet, c'est bien une fonction énumérable par dessous de Ξ dans $\mathbb{N}$, et elle vérifie aussi les conditions sur les sections critiques. Si l'on considère maintenant f le L-test universel égal à $\ell(x) - \mathbf{KL}(x) - 1$, de par son universalité, $f(x) + c' \geqslant \gamma_\lambda(x)$. Si $\mathbf{u}$ vérifie la condition du théorème, alors pour une infinité de n et d'après la première partie du théorème :

$$\gamma_\lambda(u_{1:n}) \leqslant f(u_{1:n}) + c' + c \leqslant c. \tag{2.10}$$

Observons maintenant que l'on peut poser sans contraintes que γ_λ est monotone croissante. En effet, si l'on dispose d'une fonction approximable par au-dessous γ, la fonction $\gamma'(x) = \sup_{i \leqslant x}\{\gamma(i)\}$ est bien aussi une fonction approximable par au-dessous, définissant au final le même μ-test séquentiel.

Ainsi, si on suppose γ_λ monotone croissante, $\gamma_\lambda(u_{1:n})$ est bornée par c, et donc $\delta_\lambda(\mathbf{u}) \leqslant c$. Donc $\mathbf{u}$ est bien aléatoire, ce qui prouve le théorème.

Lemme de la densité On calcule la mesure de l'ensemble des $\mathbf{u}$ vérifiant la condition du théorème. Soit $X_{c,n} = \{x \in \Xi, \ell(x) = n, \mathbf{KL}(x) \geqslant n - c\}$ et les sections critiques $\mathbf{V}_{c,n} = \bigsqcup_{x \in X_{c,n}} \lhd x$ (l'union est disjointe). De par le théorème 0.7, $|X_{c,m}| \geqslant |\xi|^m - \frac{|\xi|^{m-c}-1}{|\xi|-1} \geqslant |\xi|^m(1 - |\xi|^{-c}/(|\xi| - 1))$. Donc :

$$\begin{aligned}\lambda\left(\bigcup_{n\geqslant m}\mathbf{V}_{c,n}\right) &\geqslant \lambda(\mathbf{V}_{c,m})\\ &\geqslant |\xi|^{-m}\,|X_{c,m}|\\ &\geqslant 1-|\xi|^{-c}/(|\xi|-1).\end{aligned}$$

Puisque $\bigcup_{n\geqslant m}\mathbf{V}_{c,n}$ est une suite monotone en fonction de m, on a donc aussi :

$$\lambda\left(\bigcap_{m\geqslant 1}\bigcup_{n\geqslant m}\mathbf{V}_{c,n}\right)\geqslant 1-\frac{|\xi|^{-c}}{(|\xi|-1)}.$$

Comme $\mathbf{V}_{c,n}\subset\mathbf{V}_{c+1,n}$, on peut écrire :

$$\begin{aligned}\lambda\left(\bigcup_{c\geqslant 1}\bigcap_{m\geqslant 1}\bigcup_{n\geqslant m}\mathbf{V}_{c,n}\right) &= \lim_{c\to\infty}\lambda\left(\bigcap_{m\geqslant 1}\bigcup_{n\geqslant m}\mathbf{V}_{c,n}\right)\\ &\geqslant \lim_{c\to\infty}1-|\xi|^{-c}/(|\xi|-1)=1.\end{aligned}$$

Or $\bigcup_{c\geqslant 1}\bigcap_{m\geqslant 1}\bigcup_{n\geqslant m}\mathbf{V}_{c,n}$ dénote exactement l'ensemble des $\mathbf{u}\in\Omega$ tels qu'il existe c, tel que pour une infinité de n on ait $\mathbf{KL}(u_{1:n})\geqslant n-c$. Ce qui prouve l'inégalité demandée.

□

On a donc réussi à cerner l'ensemble des suites infinies λ-aléatoires au sens de Martin-Löf. On n'en a pas une caractérisation exacte toutefois, mais on sait que c'est un ensemble de mesure 1, et on a trouvé une caractérisation exacte d'un sous-ensemble de mesure 1. Toutefois, il est possible de prouver que ces inclusions sont strictes. Alors, on regarde une autre caractérisation qui sera, elle, exacte et plus simple, la caractérisation par la complexité préfixe.

Proposition 0.31 *Un élément* $\mathbf{u}$ *de* Ω *est* λ*-aléatoire si et seulement s'il existe une constante* c *telle que pour tout* n*, on ait* $\mathbf{KP}(u_{1:n})\geqslant n-c$*.*

◇ *Preuve.*

u est aléatoire $\Rightarrow \exists c, \mathbf{KP}(u_{1:n})\geqslant n-c$ On suppose d'abord que $\mathbf{u}$ est aléatoire, et on construit un test particulier δ, tel que si $\delta(\mathbf{u})<\infty$, alors il existe une constante c telle que $\mathbf{KP}(u_{1:n})\geqslant n-c$, pour tout n.

On définit le test par la donnée de ses sections critiques, c'est-à-dire l'ensemble des $\mathbf{u}\in\Omega$ tels que $\delta(\mathbf{u})\geqslant k$. On pose donc c' une constante qui sera fixée ultérieurement et

$$\mathbf{V}_k=\bigcup_{\mathbf{KP}(y)\leqslant\ell(y)-k-c'}\lhd y.$$

On peut aussi définir δ par la donnée de la fonction γ qui va de Ξ dans $\mathbb{N}$ $\gamma(y) = \sup\{k, \mathbf{KP}(y) \leqslant \ell(y) - k - c'\}$. C'est la même fonction.
Prouvons que δ est un λ-test séquentiel. D'après la proposition 0.16, chacun des $\mathbf{V}_k$ est bien récursivement énumérable, et par conséquent, il ne reste plus qu'à vérifier que $\lambda(\mathbf{V}_k) \leqslant |\xi|^{-k}$. On sépare donc V_k en regroupant les y de même longueur. D'après le théorème 7, et en choisissant correctement c', le nombre de y de longueur n tels que $\mathbf{KP}(y) \leqslant \ell(y) - k - c'$ est inférieur ou égal à $|\xi|^{n-K(n)-k}$. On peut donc écrire les inégalités suivantes :

$$\begin{aligned}
\lambda(\mathbf{V}_k) &\leqslant \sum_{y \in \mathbf{V}_k} \lambda(\lhd y) \\
&\leqslant \sum_{n \in \mathbb{N}} |\xi|^{n - \mathbf{KP}(n) - k} |\xi|^{-n} \\
&\leqslant |\xi|^{-k} \sum_{n \in \mathbb{N}} |\xi|^{-\mathbf{KP}(n)} \qquad (*) \\
&\leqslant |\xi|^{-k}. \qquad (**)
\end{aligned}$$

Le passage de $(*)$ à $(**)$ provient de l'inégalité de Kraft appliquée au codage dont la machine de référence pour la **KP**-complexité est la fonction de décodage. δ est donc un λ-test séquentiel ; et si $\mathbf{u}$ est λ-aléatoire, $\delta(\mathbf{u}) < c$, donc $\mathbf{KP}(u_{1:n}) \leqslant n - c$ pour tout n.

u n'est pas aléatoire $\Rightarrow \forall c, \exists n, \mathbf{KP}(u_{1:n}) < n - c$ On montre que si $\mathbf{u}$ n'est pas une suite λ-aléatoire, il n'existe pas de constante c qui soit telle que $\mathbf{KP}(u_{1:n}) \geqslant n - c$ pour tout n. Il existe donc un λ-test séquentiel tel que $\delta(\mathbf{u}) = \infty$. Soit γ la fonction de Ξ dans $\mathbb{N}$ qui est associée à δ. On prend l'ensemble $Y_k = \{y, \gamma(y) \geqslant k\}$ et on en extrait le sous-ensemble préfixe maximal défini comme suit :

$$A_k = \{y, \gamma(y) \geqslant k, \forall x \text{ préfixe de } y, \gamma(x) \leqslant k\}.$$

Notons que, par définition de δ,

$$\sum_{y \in A_k} |\xi|^{-\ell(y)} = \sum_{y \in A_k} \lambda\left(\lhd y\right) = \lambda\left(\bigsqcup_{y \in A_k} \lhd y\right) = \lambda\left(\bigcup_{y \in Y_k} \lhd y\right) \leqslant |\xi|^{-k}.$$

On voit que l'ensemble d'entiers $L = \{\ell(y) - k, y \in A_{2k}, k > 1\}$ vérifie l'égalité de Kraft. En effet,

$$\sum_{k>1} \sum_{y \in A_{2k}} |\xi|^{-(\ell(y)-k)} = \sum_{k>1} |\xi|^{k} \sum_{y \in A_{2k}} |\xi|^{-\ell(y)} \leqslant \sum_{k>1} |\xi|^{k-2k} \leqslant 1.$$

Donc L correspond à l'ensemble des longueurs d'un code préfixe, et l'on peut construire à partir de γ une machine préfixe T qui décode ce codage. Donc, il existe c tel que pour tout y de A_{2k}, $\mathbf{KP}(y) \leqslant \ell(y) - k + c$.

Or $\delta(\mathbf{u}) = \infty$, donc pour tout k il existe n tel que $u_{1:n} \in A_{2k}$, ce qui veut dire que $\mathbf{KP}(u_{1:n}) \leqslant n - k + c$. Donc $n - \mathbf{KP}(u_{1:n})$ n'est pas borné, ce qui finit de prouver le théorème.

□

2.3 Aléatoire et Automates cellulaires

Le but de cette partie est de justifier le développement de ces définitions du caractère aléatoire de suites, en caractérisant en particulier le caractère aléatoire d'un modèle de calcul différent de la machine de Turing : les automates cellulaires.

2.3.1 Le modèles des automates cellulaires

Les automates cellulaires sont un modèle de calcul qui a été très tôt introduit par John von Neumann. D'un certain point de vue, il est facile de les imaginer comme étant une grille (de dimension 1, 2 ou plus) de cellules, chacune pouvant prendre un nombre fini d'états. On regarde ensuite comment cette grille peut évoluer, en faisant changer de façon synchrone (simultanée) et déterministe l'état de toutes ces cellules en fonction de l'état de leurs voisines, par un automate fini. Le voisinage de chaque cellule est défini à l'identique par des coordonnées relatives, et on applique partout la même fonction. On obtient ainsi, à partir d'une transformation déterministe, locale et uniforme, une évolution globale.

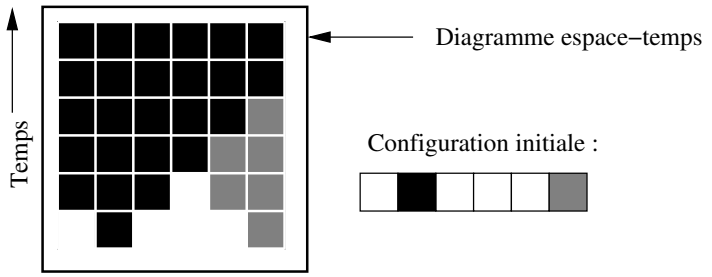
On pourra trouver les fondements de la théorie des automates cellulaires dans plusieurs en ouvrage, en particulier dans [Del99] (extrait de [DM99]) et dans [Wol86].

Les automates cellulaires sont des systèmes qui ont beaucoup été utilisés pour représenter des systèmes physiques à contraintes locales, ou encore en biologie pour l'étude de la dynamique des populations.

On peut donner ainsi une définition formelle des automates cellulaires :

Définition 21 (Automate cellulaire) Un **automate cellulaire** est un quadruplet $\mathcal{A} = (d, S, V, f)$ où :
- d désigne la *dimension* de l'automate ;
- S désigne l'ensemble fini des *états de l'automate* $\{s_1,\ s_2,\ \ldots, s_m\}$;
- V désigne le *vecteur de voisinage* $\{v_1,\ v_2,\ \ldots, v_n\}$;
- $\delta : S^n \to S$ désigne sa *fonction de transition locale.*

Un automate cellulaire est en fait pleinement défini lorsqu'on y ajoute la notion de configuration. Une configuration est une affectation des valeurs des cellules ; on la formalise comme étant une fonction de $\mathbb{Z}^d \to S$. Il est possible ainsi de définir l'application de l'automate cellulaire à la configuration, qui consiste, comme précisé plus haut, à transformer de façon synchrone la valeur de l'état de chaque cellule. La configuration détermine alors les « conditions initales ». Les notions d'automate cellulaire et de configuration interagissent ; c'est leur réunion qui transforme le modèle des automates cellulaires en un modèle de calcul.



L'automate cellulaire, de rayon 1*, choisit l'état le plus « foncé ».*

FIGURE 2.2 – Un diagramme espace-temps d'automate cellulaire à trois états

On définit ainsi la fonction globale d'un automate cellulaire :

Définition 22 (Fonction globale) La fonction globale $G_{\mathcal{A}}$ de l'automate cellulaire $\mathcal{A}$ est la fonction de $S^{\mathbb{Z}}$ définie par l'équation qui suit. Étant donné $c \in S^{\mathbb{Z}}$ une configuration (qui peut être vue comme une fonction de $\mathbb{Z}$ dans S), on définit l'image par $G_{\mathcal{A}}$ de c comme étant l'unique configuration telle que :

$$\forall x \in \mathbb{Z}, G_{\mathcal{A}}(c)(x) = \delta(c(x-r), c(x-r+1), \ldots, c(x+r)).$$

La notion de configuration amène ainsi tout droit à la notion de diagramme espace-temps. Le diagramme espace-temps d'un automate cellulaire [4] est une représentation de la suite des itérées successives. On peut la formuler comme étant une suite (finie ou infinie) de configurations, ou en donner des représentations graphiques (voir figure 2.2). On assimile alors le comptage des itérations à une notion de temps discret.

On examine ici surtout les automates de dimension 1, bien que l'analyse faite se transcrive aisément en dimension quelconque (il faut alors remplacer les mots comme triangle dans l'énoncé des propositions 0.33 et 0.34 par un équivalent dans la dimension adéquate). On supposera également que le voisinage est défini de manière uniforme comme étant toutes les cellules dont les coordonnées (relatives) ont une valeur absolue inférieure ou égale à r, qui est alors appelé le *rayon* de l'automate cellulaire. Le nombre de voisins est de $2r+1$. Un automate cellulaire est donc donné entièrement par son nombre d'états, et sa table (fonction de transition locale), qui est une fonction de $S^{2r+1} \to S$.

On va particulièrement s'intéresser à quantifier un comportement dynamique des automates cellulaires, en tant que système dynamique discret. Une approche plus complète peut être trouvée dans [For98].

2.3.2 Le problème de la classification

Dans [Wol84], S. WOLFRAM a observé et reconnu, de façon empirique, quatre classes principales d'automates cellulaire. Il a considéré d'abord le problème sur les automates de

4. L'appelation « diagramme espace-temps d'un automate cellulaire » est d'ailleurs impropre, puisque c'est le diagramme d'un automate cellulaire et d'une configuration spécifique, et non de l'automate cellulaire en général.

dimension 1, à 2 états, qui avaient un rayon de 1, et distingué les comportements suivants :

W_1 Certains automates évoluent vers une configuration homogène ;

W_2 D'autres atteignent des configurations stables ou périodiques ;

W_3 Certains trouvent des configurations apériodiques, qui n'ont pas de sens, ou « chaotiques » ;

W_4 Les autres évoluent vers des configurations localement complexes, parfois de longue durée de vie : des configurations « ayant du sens ».

Cette classification n'était pas prévue pour être bien formelle, ni complète. De nombreux travaux ont en fait essayé de donner des fondements ou une consistance à cette classification « primaire » (comme dans [ČPY89] ou [BCFQ95]). Des considérations purement topologiques ont parfois été utilisées, se concentrant sur la structure des attracteurs ou la propriété d'équicontinuité (dans [Hur90, Kur97]).

C'est donc la première catégorie de méthodes de classification qui ont été instaurées. Elles étudient la dynamique des automates cellulaires, c'est-à-dire l'ensemble configuration et automate cellulaire. Les méthodes se sont montrées de plus en plus larges, agissant sur une configuration, puis passant à l'analyse de propriétés vraies sur toutes les configurations, puis toutes les configurations sauf une quantité négligeable.

En 1994, J. Kari a démontré que si « classifier » un automate cellulaire était pouvoir associer algorithmiquement une classe à un automate cellulaire, alors, aucune des propriétés énumérées ci-dessus n'était décidable. Or aucune de ces évolutions n'essaye de prendre en compte une mesure de l'entropie des automates cellulaires considérés, ni la complexité de leur évolution. L'idée peut être alors de définir une fonction, qui est capable d'associer à chaque table d'automate (fonction locale) une valeur quantifiant le comportement prédictible de l'automate. L'une des premières applications de ce genre de paramètre est due à C. Langton, dans [Lan90]. On arrive dans une deuxième catégorie de classification, les critères qui essayent de qualifier un automate cellulaire dans sa généralité.

Notre idée est d'introduire un paramètre de classification qui puisse capturer la notion d'aléatoire et de complexité algorithmique. En utilisant un ordre fondé sur la quantité d'aléatoire, on peut imaginer ainsi ordonner tous les paramètres possibles, et en construire un, par un procédé de diagonalisation, qui puisse être maximal (dans un certain sens), et qui ait des propriétés universelles.

2.3.3 Paramètres de classification

Un *paramètre de classification* est une fonction de l'ensemble des tables des automates cellulaires et à valeurs dans l'intervalle $[0, 1]$. Chaque paramètre doit répondre aux requêtes minimales suivantes : bien quantifier les régularités des tables, être effectif, être adéquatement normalisé. [5]

5. À la lumière de ce chapitre, et en particulier de la section 2.1.1, il est clair que notre définition de paramètre de classification est fortement inspirée de celle de *tests de Martin-Löf*.

La première requête est vite satisfaite car on définira notre paramètre par la complexité de Kolmogorov, de façon à ce que la valeur du paramètre soit très basse pour les tables avec beaucoup de régularités et, réciproquement, que la valeur soit haute pour les tables sans régularités.

On définit les ensembles $\mathcal{E}_{p/q}$ par l'équation suivante, où x est une table et $p \in \mathbb{N}$, $q \in \mathbb{N} \setminus \{0\}$ et π est un paramètre de classification :

$$\mathcal{E}_{p/q} = \{(x, p, q), \pi(x) \leq p/q\} \tag{2.11}$$

L'effectivité est accomplie en demandant que chaque ensemble $\mathcal{E}_{p/q}$ soit énumérable. En pratique, demander que les ensembles du type (2.11) soient énumérables est équivalent à demander que le paramètre soit approximable par dessus par des fonctions calculables. Sans perte en généralité on restreindra les images des paramètres à $\mathbb{Q} \cap [0, 1]$.

La normalisation des paramètres est un point délicat. Nous voulons que tous les paramètres ainsi définis donnent à peu près la même valeur pour une même table. Évidemment, une normalisation uniforme n'est pas possible car, autrement, il serait facile de donner deux paramètres qui donnent des valeurs très écartées pour une même table. Nous choisissons de normaliser par rapport au nombre maximal de tables qui peuvent prendre une valeur donnée du paramètre. Nous utilisons la normalisation suivante, en se fondant sur la longueur des tables (représentée par le nombre de transitions à définir pour complètement spécifier l'automate considéré) :

$$\forall m, n \in \mathbb{N}, \sharp \left\{ x, \pi(x) \leqslant \frac{m}{n}, \ell(x) = n \right\} \leqslant S^m \tag{2.12}$$

qui est équivalente à la suivante

$$\forall m, n \in \mathbb{N}, \sum_{\substack{\pi(x) \leq m/n \\ \ell(x) = n}} S^{-n} \leqslant S^{m-n},$$

où S est la cardinalité de l'alphabet des automates cellulaires. Nous avons choisi cette normalisation très particulière pour des raisons techniques qui seront éclairées plus loin au théorème 13.

Formellement on peut donner la définition suivante :

Définition 23 Soit $\mathbb{T}$ l'ensemble des tables des automates cellulaires. Un *paramètre de classification* π est une fonction totale de $\mathbb{T}$ à valeurs dans $[0, 1] \cap \mathbb{Q}$ telle que les conditions (2.11) et (2.12) soient satisfaites.

Proposition 0.32 *L'ensemble des paramètres de classification est récursivement énumérable.*

◇ *Preuve.* À la notation près, la preuve est la même que celle utilisée pour montrer que la classe des P-tests de Martin-Löf est récursivement énumérable, comme dans la proposition 0.21. Une preuve détaillée se trouve dans [DDF00]. □

La proposition 0.32 est très importante car elle nous permettra de montrer l'existence d'un paramètre optimal dans le théorème 13.

2.3.4 Le paramètre κ

On a présenté dans la section précédente une classe de paramètres qui sont censés quantifier l'aléatoire qui est présent dans la table descriptive d'un automate cellulaire. Nous allons choisir notre paramètre κ dans cette classe. En particulier, nous choisissons κ de manière à ce qu'il soit optimal, c'est-à-dire qu'il soit au moins aussi bon que tous les autres. Formellement la définition d'optimalité est la suivante :

Définition 24 Un paramètre de classification π_o est *optimal* si et seulement si pour tout paramètre de classification π on a :

$$\exists c \in \mathbb{N}, \forall x \in \mathbb{T}, \pi_o(x) \leqslant \pi(x) + \frac{c}{\ell(x)},$$

où x est une table d'un automate cellulaire quelconque.

Dans cette définition, le fait que la constante c soit divisée par $\ell(x)$ est dû à la normalisation. En quelque sorte, on s'autorise moins de marge si on travaille sur des automates ayant une taille plus grande.

Théorème 13 *Soit x une table et $\kappa(x) = \frac{\mathbf{KS}(x|\ell(x))+1}{\ell(x)}$. La fonction κ est un paramètre de classification optimal.*

◇ *Preuve.* D'abord nous allons prouver que κ est un paramètre de classification et ensuite qu'il est optimal au sens de la définition 24.

κ est approximable par dessous car **KS** est approximable par dessus (cf. proposition 0.5). Considérons maintenant la cardinalité des ensembles

$$F_{m,n} = \{x, \ell(z) = n \text{ et } \mathbf{KS}(x|\ell(x)) \leqslant m\}.$$

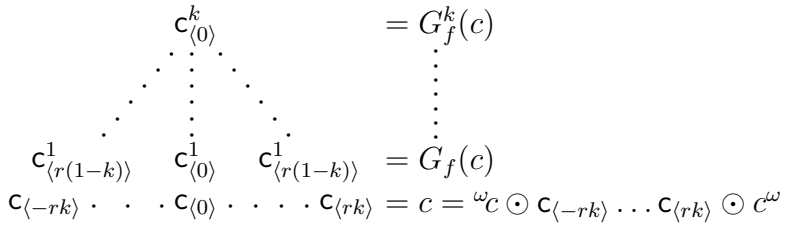
Pour m et n fixés, $|F_{m,n}|$ est inférieure à la cardinalité de l'ensemble de tous les programmes de longueur inférieure ou égale à $m-1$, c'est-à-dire $S^m - 1$. De la définition de κ on déduit $\sharp\left\{x, \pi(x) \leqslant \frac{m}{n}, \ell(x) = n\right\} \leqslant S^m - 1$. Donc κ est bien un paramètre de classification.

Arrivé à ce point, il nous reste à prouver l'optimalité de notre paramètre. De la proposition 0.32, nous pouvons tirer qu'il existe une énumération des paramètres de classification. Soit π_y le paramètre de position y dans cette numérotation. Pour toute table x fixée, nous allons construire une description de x telle que $\mathbf{KS}(x|\ell(x)) \leqslant \ell(x)\pi_y(x) - 1 + c_y$ où c_y est une constante que dépend seulement de l'indice y et non de la table x en question. Considérons l'ensemble

$$V = \{z, \ell(z) = n, \pi_y(z) \leqslant \pi_y(x)\}.$$

Remarquons que x appartient à V et que V peut être numéroté si nous connaissons $\pi_y(x)$ et $\ell(x)$. Donc on peut prendre l'indice j de la position de x dans la numérotation précédente comme description de x. Calculons la cardinalité de V :

$$|V| = \sharp\left\{z, \ell(z) = n, \pi_y(z) \leqslant \frac{\pi_y(x) \cdot \ell(x)}{\ell(x)}\right\} \leqslant S^{\pi_y(x)\ell(x)},$$



Les lettres $\mathsf{c}^i_{\langle j\rangle}$, pour $|j| \leqslant rk(k-i)$ et $i \geqslant 0$ sont indépendantes de la valeur de ${}^\omega c$ et c^ω. Le triangle de calcul est l'ensemble de ces lettres.

FIGURE 2.3 – Un triangle de calcul pour un automate cellulaire.

où la dernière inégalité est obtenue de l'équation (2.12).

Donc il suffit d'une chaîne s de longueur $\pi_y(x)\ell(x)+1$ pour décrire x. Remarquons qu'en sachant s et $\ell(x)$, on peut calculer $\pi_y(x)$ et que si l'on a $1^{\ell(y)}0ys$ en entrée, on peut toujours calculer x (en sachant $\ell(x)$). Donc, $\mathbf{KS}(x|\ell(x)) \leqslant \ell(1^{\ell(y)}0ys) \leqslant \ell(x)\pi_y(x) + 2\ell(y) + 2$. La constante cherchée est donc $c_y = 2\ell(y) + 3$. □

2.3.5 Une comparaison avec l'approche de Wolfram

Dans les applications pratiques, les évolutions des automates cellulaires sont étudiées à l'ordinateur par des simulations. Le but de ces simulations est de comprendre le comportement de l'automate et de prévoir ses évolutions futures, obtenues éventuellement à partir d'autres configurations initiales. Comme d'habitude, les comportements auxquels nous nous intéressons sont chaotiques ou aléatoires.

Dans cette section nous montrons que les évolutions des automates cellulaires ne sont pas aléatoires. Le deuxième résultat répond à une exigence que l'on pourrait assimiler à de « l'identification d'automate ». En d'autres mots, supposons que l'on ait une suite infinie de simulations. On se pose la question : combien d'automates cellulaires engendrent la même suite ? Existe-t-il un seul automate cellulaire qui les engendre toutes ? La réponse est oui si toutes les simulations sont suffisamment aléatoires (voir la proposition 0.34).

Tout d'abord, il faut remarquer qu'à cause du caractère borné des ordinateurs on ne peut gérer que des portions de simulation. Pour pallier ces inconvénients, on impose souvent des contraintes aux limites, comme, par exemple, la circularité [Wol86]. Pour s'échapper à ces limitations nous étudierons des *triangles de calcul*, c'est-à-dire la portion de calcul qui n'est pas influencée par les conditions aux limites. Plus formellement :

Définition 25 Soit c une configuration. Un *triangle de calcul* de base $2rk+1$ pour l'automate (S, r, f) et une configuration initiale c, est l'ensemble des lettres de coordonnées j telles que $|j| \leqslant rk(k-i)$, de la configuration obtenue par $G^i_f(c)$, pour $0 \leqslant i \leqslant k$ (voir la figure 2.3).

On constate tout de suite qu'avec cette définition, les configurations identiques sur les $2kr+1$ lettres centrées autour de l'origine donnent le même triangle de calcul. Quand on

parlera d'un triangle de calcul, on pourra parfois parler de la configuration correspondante (parlant alors de la partie commune à toutes les configurations), ou encore d'un représentant quelconque des configurations possibles.

Proposition 0.33 *Soit V un triangle de calcul d'un automate cellulaire de table x extrait d'une configuration initiale aléatoire c. Alors on a :*

$$\mathbf{KL}(V) \leqslant \mathbf{KL}(x) + 2rk + \mathcal{O}\left(\mathbf{KS}(k)\right). \tag{2.13}$$

◇ *Preuve.* Un triangle de calcul V est complètement déterminé par la table x de l'automate cellulaire et la configuration initiale c. Suppposons que k soit donné. On peut les coder dans une chaîne (codée de manière préfixe) de longueur au plus $2\mathbf{KS}(k)$. Connaissant $\ell(V) = k(k+1)(r+1)$, si on connaît k, on peut en déduire r. De plus, à partir de k et de r, on peut calculer $\ell(x)$ et $\ell(c)$. On peut ainsi déterminer V sachant $\ell(V)$, à partir d'un mot le plus court pour x sachant $\ell(x)$, d'un mot le plus court pour c sachant $\ell(c)$, d'un mot de longueur $2\mathbf{KS}(k)$, et d'un moyen de séparer les deux mots les plus courts pour x et c ($x^\star$ et $c^\star$). Or $c^\star$ est de longueur proche de $2kr+1$, et kr est connu. Donc le codage de $\ell(c^\star)$ prend une taille qui est majorée par le degré d'aléatoire de la configuration initale c, majorée donc par une taille constante. On peut de plus allonger la taille de $c^\star$ à $2rk+2$ et y coder la différence de longueur entre $\ell(c^\star)$ et $2rk+1$. Cette information est suffisante pour séparer $c^\star$ et $x^\star$, et donc on peut reconstituer V à partir de $\ell(V)$, de $x^\star$ et d'un mot de longueur $2rk+2$, et d'un mot de longueur $\mathcal{O}\left(\mathbf{KS}(k)\right)$. Ce qui prouve l'inégalité (2.13). □

Que nous apprend cette proposition ? Que finalement, la notion d'apparence aléatoire est une notion instable. La complexité d'un triangle de calcul est intrinsèquement limitée par la construction par un automate cellulaire du triangle de calcul. On peut exploiter cette faible quantité d'aléatoire de deux façons. D'une part, le caractère aléatoire très fort de la configuration de départ contraint d'une certaine façon l'évolution de la complexité d'une suite de triangles de calcul calculés avec la même table. On peut ainsi déduire la table exacte nécessaire à partir du moment où l'on est sûr que l'on aura des données suffisamment aléatoires. C'est ce qu'explique la proposition suivante :

Proposition 0.34 *Soit $(V_n)_{n\in\mathbb{N}}$ une suite de triangles de calcul de configuration initiale c_n et engendrés par un même automate cellulaire, tel que $\ell(c_n)$ tende vers l'infini quand n tend vers l'infini. Soit B_n le nombre de tables de taille minimale qui peuvent donner V_n quand appliquées à la configuration c_n. Si tout c_n est ϵ-aléatoire alors $\lim_{n\to\infty} B_n = 1$.*

◇ *Preuve.* Si une configuration initiale est ϵ-aléatoire alors elle doit contenir tous les blocs possibles de longueur $2r+1$. Autrement, la configuration peut être écrite avec un alphabet de taille $2^{2r+1}-1$, c'est-à-dire que sa complexité est bornée par $\ell(c_n/(2r+1))\log(2^{2r+1}-1)$. Il y a donc une déficience du caractère aléatoire qui vaut $\ell(c_n)\frac{\log(2^{2r+1}-1)}{2r+1}$ et elle ne peut pas être bornée par une constante. Au-delà d'une certaine taille, toutes les transitions sont obtenues. Il est donc possible de déterminer quelle est la taille minimale nécessaire à l'obtention du triangle de calcul complet. Ainsi, il existe un entier N au-delà duquel il

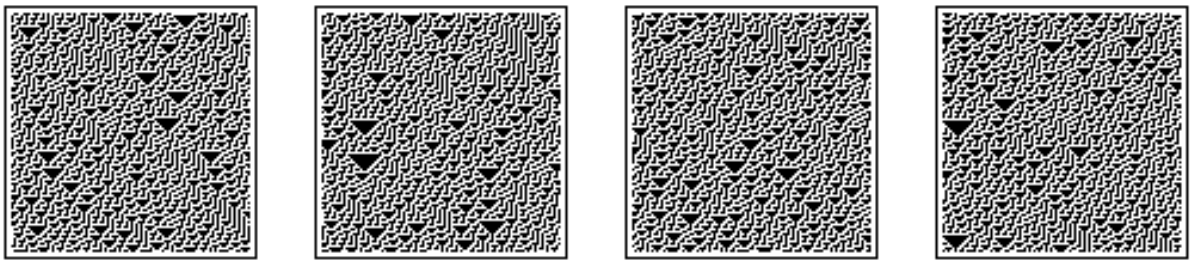


FIGURE 2.4 – Quelques diagrammes espace-temps de la règle élémentaire 30.

existe un unique automate cellulaire de taille minimale (on rencontre toutes les transitions possibles, déterminant ainsi complètement la table). Donc $B_n = 1$, pour $n \geqslant N$. □

Une autre façon de voir est de comparer un triangle de calcul à un triangle généré de façon réellement aléatoire. Si nous regardons les diagrammes espace-temps de certains automates cellulaires, par exemple ceux de la figure 2.4, ils ne nous semblent pas seulement complexes, mais on les qualifierait d'aléatoires. Cette dernière conclusion est fausse, comme nous montre la proposition suivante :

Proposition 0.35 *Aucun automate cellulaire ne produit des triangles de calcul aléatoires.*

◇ *Preuve.* Soit V un triangle (pas nécessairement un triangle de calcul) de largeur $2rt + 1$ et hauteur $t + 1$. Remarquons que le nombre de cellules dans V est $(t + 1)(r + 1)t$. Par définition, si V est c-aleatoire alors $\mathbf{KL}(V) \geqslant (t + 1)(r + 1)t - c$. Pour un automate cellulaire fixé,on déduit de la proposition 0.33 que la progression de la complexité de ses triangles de calcul par rapport à celle des triangles aléatoires est t par rapport à t^2. □

Remarquons que, grâce aux propriétés de la complexité de Kolmogorov, notre approche est robuste vis-à-vis des restrictions à des sous-ensembles récursifs de l'espace de phases. Ce cas se produit souvent dans les applications pratiques : pensons par exemple aux phénomènes de percolation laminaire. Les configurations considérées pour modéliser ce phénomène sont celles qui représentent un milieu poreux traversé par un fluide. Ce milieu peut être spécifié par un algorithme, donc l'ensemble des configurations auxquelles on se restreint est récursif. Notre approche reste donc également valide dans ce contexte et les expérimentations proposées plus loin dans le chapitre 2.3.7 gardent leur sens.

2.3.6 Complexité de Kolmogorov et chaos topologique

À partir de cette section, nous considérerons des automates cellulaires non plus à 2 états, mais avec un alphabet quelconque de taille S.

Nous avons vu que les calcul des automates cellulaires ne sont pas aléatoires. Il est donc plus approprié de parler de complexité ou bien de dégré de complexité à l'intérieur de la classe des automates cellulaires. Dans cette section, nous étudions la relation qui existe entre le comportement chaotique, c'est-à-dire le comportement le plus complexe

possible (du point de vue des systèmes dynamiques) et la quantité d'aléatoire de la table de l'automate.

Il est possible de définir les automates cellulaires dont la table a une propriété particulière : chaque valeur de la table peut être atteinte un nombre de fois exactement égal. C'est la propriété d'être 1-équilibré.

Définition 26 (Automate 1-équilibré) Un automate $\mathcal{A} = (d, \mathcal{S}, V, f)$ avec une fonction de transition locale f est 1-équilibré si et seulement si pour toute lettre $a \in \mathcal{S}$, $\sharp\{x, f(x) = a\} = S^{|V|}/S$.

Chaque état a donc exactement S^{2r} antécédents. Ces automates cellulaires ont des propriétés dynamiques assez particulières. En particulier, si un automate est sensible aux conditions initales, condition souvent assimilée au « chaos » dans le domaine des systèmes dynamiques, il vérifie la propriété d'être 1-équilibré.

Théorème 14 *La table des automates cellulaires* 1*-équilibrés n'est pas aléatoire.*

◇ *Preuve.* L'idée est de montrer que dans ce cas on peut donner une représentation de la table beaucoup plus courte que $S^{2r+1} \log S$. Considérons le codage suivant. D'abord, nous divisons l'espace des tables en un certain nombre de classes ; ensuite un automate cellulaire est déterminé par deux indices : celui de la classe et celui de sa position à l'intérieur de cette classe.

Une classe est déterminée par un vecteur ordonné d'entiers. L'entier de position i est la différence entre $S = \frac{S^{2r+1}}{S^{2r}}$ et le nombre d'entrées qui donnent comme sortie l'état s_i. Nous utilisons une représentation autodelimitée pour ces nombres.

Dans le cas des automates cellulaires 1-équilibrés, toutes les entrées du vecteur sont 0 ; donc pour déterminer cette classe on a besoin seulement de coder S et r. Maintenant nous allons calculer le nombre de bits nécessaires pour représenter l'indice d'un automate cellulaire à l'intérieur de cette classe. Ce nombre est au maximum le logarithme de la cardinalité de la classe. Nous calculons donc la cardinalité des automates cellulaires 1-équilibré pour S et r fixés.

On doit choisir S^{2r} places sur S^{2r+1}, après on doit choisir S^{2r} places sur $S^{2r+1} - S^{2r}$, et ainsi de suite. Nous obtenons :

$$\prod_{i=0}^{S-2} \binom{(S-i)A}{A} = \prod_{i=0}^{S-2} \frac{[(S-i)A]!}{A! \cdot [(S-i-1)A]!}, \tag{2.14}$$

où $A = S^{2r}$. En utilisant la formule de Stirling on approche chaque élément du produit

de l'équation (2.14) comme suit :

$$\begin{aligned}\binom{(S-i)A}{A} &\approx \frac{[(S-i)A]^{(S-i)A}\cdot e^{-A(S-i)}\sqrt{2\pi A(S-i)}}{[(S-i-1)A]^{(S-i-1)A}\cdot e^{-A(S-i-1)}\sqrt{2\pi A(S-i-1)}} \\ &\quad \cdot \frac{1}{A^A\cdot e^{-A}\sqrt{2\pi A}} \\ &= \left[\frac{S-i}{(S-i-1)}\right]^{(S-i)A}\cdot(S-i-1)^A\cdot\sqrt{\frac{S-i}{S-i-1}}\cdot\sqrt{\frac{1}{2\pi A}} \\ &= D_i\cdot E_i\cdot F_i\cdot G_i.\end{aligned}$$

On a :

$$\prod_{i=0}^{S-2} E_i\cdot F_i\cdot G_i = [(S-1)!]^A\cdot\sqrt{\frac{S!}{(S-1)!}}\cdot\sqrt{\frac{1}{(2\pi A)^{S-1}}},$$

et

$$\prod_{i=0}^{S-2} D_i = \left[\frac{S^S\cdot(S-1)^{S-1}\cdot\ldots\cdot 1}{(S-1)^S\cdot(S-2)^{S-1}\ldots\cdot 1}\right]^A = \left[\frac{S^S}{(S-1)!}\right]^A.$$

Enfin,

$$\prod_{i=0}^{S-2}\binom{(S-i)A}{A} \approx S^{SA}\sqrt{\frac{S}{(2\pi A)^{S-1}}}. \tag{2.15}$$

Donc le défaut de caractère aléatoire en termes de complexité de Kolmogorov est du même ordre de grandeur que Sr.[6] □

De la preuve précédente il faut noter que la déficience du caractère aléatoire tend vers l'infini quand S et r tendent vers l'infini, alors que si l'on calcule la même quantité par rapport à κ, notre paramètre, on obtient $\Omega\left(\frac{r}{S^{2r}}\right)$, qui tend vers 0. Ceci est un phénomène dû à la renormalisation sur $[0,1]$ de notre paramètre.

Corollaire 0.36 *Les automates cellulaires avec les propriétés suivantes n'ont pas une table aléatoire : transitifs, réguliers, surjectifs, injectifs.*

◇ *Preuve.* Toutes ces propriété impliquent la surjectivité qui à son tour implique la propriété d'être 1-équilibré. En particulier, les automates cellulaires surjectifs sont 1-équilibrés ([MK76, MK79, Hed69]) ; comme l'injectivité est synonyme de bijectivité (théorème de Moore-Myhill [Moo62, Myh63]), c'est aussi vrai pour les injectifs. On peut en trouver la preuve dans divers ouvrages sur le sujet (entre autres [For98]). À ce point, on applique le théorème précèdent. □

Voyons maintenant une autre classe de règles : les automates cellulaires additifs. Il s'agit d'un ensemble des règles simples, où l'on calcule le nouvel état par combinaison linéaire des voisins, et en même temps, ces automates présentent de nombreux comportements chaotiques intéressants (voir par exemple certains ouvrages traitant de la dynamique

6. Plus précisément, $\log((2\pi)^{-1/2}S^{r(S-1)-\frac{1}{2}})$.

des automates cellulaires [CFMM99, MM97, For98]). Néanmoins, la proposition suivante montre que la complexité algorithmique de ces règles est très proche de zéro.

Proposition 0.37 *Soit $\mathcal{A}_\mathcal{A}$ un automate cellulaire additif. Alors $\kappa(\mathcal{A}_\mathcal{A}) = \Omega\left(\frac{r}{S^{2r+1}}\right)$.*

◇ *Preuve.* Pour S et r fixés, la cardinalité de l'ensemble des automates cellulaires additifs est S^{2r+1}. Donc $\mathbf{K}(x) \leqslant (2r+1)\log S + c$, où x est la table d'un automate cellulaire additif et c est une constante qui tient compte du codage des tables. □

2.3.7 Proposition d'un protocole

La complexité de Kolmogorov **KS**, de même que toutes ses variantes, n'est pas calculable mais seulement approximable par dessus. Donc notre paramètre κ a le même désavantage. Pour pallier cet inconvénient, nous proposons le protocole suivant :

Protocole expérimental Pour toute table x, nous proposons d'approcher $\kappa(x)$ par le taux de compression de la table x ; pour ceci on peut utiliser un algorithme de compression suffisamment efficace. [7]

Une autre possibilité parfois choisie consiste à se fixer une machine de référence, et à utiliser cette machine pour calculer la complexité de certaines classes de machine. Les deux approches sont équivalentes, et sont valables par leurs résultats aymptotiques.

L'application pratique de ce protocole nécessite des précautions. Supposons que l'on ait un alphabet de cardinalité différente d'une puissance de 2. Dans la représentation des éléments de l'alphabet comme suites de symboles on a des interstices vides qui peuvent par exemple altérer les taux de compression. obtenus

Un autre problème est la manière dont nous allons transformer les triangles de calcul en un mot pour les comprimer. Il est clair que si les triangles de calcul sont tous aléatoires ceci n'est pas un problème, car la complexité ne changera que d'une constante. Mais si, au contraire, les triangles de calcul ne sont pas aléatoires, les performances du programme de compression peuvent être loin de la réalité car ils ne pourront pas repérer certaines régularités. Nous pensons que les cas pathologiques seront très peu nombreux — ou à étudier en fonction de la classe particulière d'automates. L'application de cette méthode est une méthodologie générale, pas une solution universelle. Ainsi, il serait intéressant d'appliquer sur des cas pratiques l'étude des relations entre le taux de compression des tables et l'évolution dynamique des automates cellulaires.

7. Plus précisément, on utilisera l'algorithme de compression le plus efficace possible à notre connaissance, en essayant ainsi de prendre en compte des symétries évidentes.

Chapitre 3

Machines préfixes

Il n'existe pas une seule définition, mais plusieurs de ce qu'est la notion de préfixe. Si la plus utilisée est une notion de codage préfixe absolue, il est possible de définir plusieurs formes de machines préfixes. Nous allons en discuter dans ce chapitre et analyser les conséquences de leurs différences en théorie de la calculabilité et de la complexité de Kolmogorov. Pour cela, nous donnons des définitions générales sur le calcul qui permettent de définir des théorèmes valides sur une sous-classe de machines. Pour ce chapitre, nous ferons très attention à la distinction entre fonctions et machines. On ne notera pas de la même façon une machine, et la fonction calculée par cette machine. Il s'avère préférable de distinguer les deux, en particulier dans les définitions. On utilisera la notation f pour la fonction calculée par la machine $\hat{f}$. On emploiera, à l'inverse, la notation $\langle f \rangle$ pour le numéro d'une machine, bien que cette notation puisse être omise, ou la même notation pour le numéro d'une machine calculant une fonction f. On notera $\hat{\phi}_n$ la n-ième machine du système acceptable de programmation de référence. Ainsi, $\hat{\phi}_{\langle f \rangle} \mathrel{\hat{=}} \hat{f}$ (en tant qu'égalité de machines, et non pas d'égalité de fonctions) ; et $\phi_n = g$ n'implique pas $\langle g \rangle = n$ (le $=$ est utilisé pour marquer l'égalité des fonctions, et le $\hat{=}$ est utilisé pour marquer l'égalité des machines).

Sommaire

3.1 Quatre définitions de machines préfixes

Il existe plusieurs façons de considérer des machines préfixes. Intuitivement, une machine préfixe est une machine dont on limite le format des entrées à un code préfixe. C'est-à-dire que ses entrées sont toujours des mots finis, mais que l'on peut calculer où est la fin du mot. D'une autre façon, on peut aussi imaginer que les mots de l'entrée sont plongés dans un ruban qui est recouvert de bruit, et qu'on écrit ainsi le mot sur le bruit initial. On cherche alors à obtenir toujours le même résultat, quelque soit le bruit entourant l'entrée. Enfin, on peut imaginer un autre système où ce qui est important, c'est que si on prolonge un mot donnant un résultat, le résultat plus long ne vienne pas contredire le premier résultat. Par contre, si on le prolonge, on ne veut pas forcément se contraindre à avoir une convergence systématiquement. En fait, on désire simplement que les résultats soient cohérents, si et seulement s'il y a convergence dans le cas du mot de base et du mot rallongé. Ce modèle plus général est en fait plus simple (mathématiquement parlant), mais moins intuitif. Étant donné que la théorie de l'information algorithmique, et en particulier la complexité de Kolmogorov, utilisent une notion de contexte, nous allons définir des modèles de calcul munis de cette notion. Lorsque nous parlerons de propriété préfixe, nous distinguerons une des entrées (*l'argument* de la fonction calculée) de tout le reste, qui peut être regroupé sous la forme d'une seule variable, le *contexte*. Ainsi, l'expression de $\mathbf{KL}(x)$ recouvre naturellement le calcul de programmes ayant pour contexte $\ell(x)$, indépendamment du format de codage des entrées du programme. Pour mieux se représenter les divers types de fonctionnement de ces machines, il peut être intéressant de se les représenter par rapport à la frontière, définie comme suit : la frontière d'une machine préfixe est l'ensemble des mots appartenant au domaine de la machine, et qui sont minimaux dans cet ensemble pour l'ordre induit par la relation préfixe.

3.1.1 Machines préfixes creuses

Les machines préfixes creuses sont les machines dont le domaine est un code préfixe. Pour une telle machine préfixe, le domaine est égal à la frontière. C'est en cela qu'elle est « creuse ». Cette définition capture bien l'essentiel de la fonction calculée si l'on considère l'entrée comme fournie entière, et que l'objectif est de vérifier la conformité à une contrainte préfixe.

Définition 27 (Classe MPC) Une machine $\hat{f}$ appartient à la classe des machines $\mathcal{MPC}$ si et seulement si la machine $\hat{f}$ vérifie la condition suivante :

$$\forall u, v, y, \hat{f}\langle u, y\rangle \downarrow \text{ et } \hat{f}\langle u \cdot v, y\rangle \downarrow \Longrightarrow v = \varepsilon.$$

On rappelle que l'expression $u \cdot v$ représente la concaténation. Cette définition correspond à la notion : on doit pouvoir calculer la longueur d'un mot en le lisant de gauche à droite.

3.1.2 Machines préfixes pleines

Les machines préfixes pleines sont les machines dont le domaine est égal à l'union des cônes engendrés par la frontière. Si un calcul est convergent, alors il est convergent sur toutes ses continuations. On capture ainsi la notion inhérente à un modèle où l'entrée est faite à la demande, et où une machine ne peut pas savoir si elle a utilisé toute l'entrée — la demande d'une entrée supplémentaire faisant partir la machine dans un état « divergent ».

Définition 28 (Classe MPP) Une machine $\hat{f}$ appartient à la classe des machines $\mathcal{MPP}$ si et seulement si la machine $\hat{f}$ vérifie la condition suivante :

$$\forall u, v, y, \hat{f}\langle u, y\rangle \downarrow \Longrightarrow \hat{f}\langle u \cdot v, y\rangle \downarrow \text{ et } f\langle u, y\rangle = f\langle u \cdot v, y\rangle.$$

3.1.3 Machines préfixes généralisées

Les machines préfixes généralisées sont la plus grande classe de machines préfixes. Une seule condition de cohérence est maintenue : si deux mots dont l'un est préfixe de l'autre appartiennent au domaine, ils doivent donner le même résultat.

Définition 29 (Classe MPG) Une machine $\hat{f}$ appartient à la classe des machines $\mathcal{MPG}$ si et seulement si la machine $\hat{f}$ vérifie la condition suivante :

$$\forall u, v, y, \left.\begin{array}{c}\hat{f}\langle u, y\rangle \downarrow \\ \hat{f}\langle u \cdot v, y\rangle \downarrow\end{array}\right\} \Rightarrow f\langle u, y\rangle = f\langle u \cdot v, y\rangle.$$

3.1.4 Machines doublement préfixes

Il existe des codes plus contraints encore que les codes préfixes qui permettent de contraindre l'entrée. Une extension naturelle du problème « est-il possible de trouver la fin d'un mot en étant placé au début » qui est le problème des machines préfixes creuses, est la question suivante : « est-il aussi possible de retrouver les deux extrémités d'un mot ? » En d'autres termes, peut-on coder les entrées et extraire du bruit ambiant un mot sans savoir *a priori* où il commence et où il finit ?

Le système peut être imaginé de la façon suivante : notre modèle de calcul recopie les données sur un ruban initialement recouvert de bruit, et il commence à calculer. Mais entre

la phase de recopie et la phase de calcul proprement dite, le système ne retient qu'une seule cellule du mot, choisie au hasard, comme point de départ. Pour pouvoir s'y retrouver, et faire des calculs cohérents, il faut être une machine doublement préfixe. Il faut pouvoir retrouver le début et la fin du mot.

Pour répondre à cette question par une classe de machines, l'idée est de ne pas avoir de sous-mot valide à l'intérieur d'un mot reconnu par la même machine. Ainsi, indépendamment de l'endroit où serait la tête de lecture au début du calcul, il serait possible de localiser les bords du mot par une recherche « en spirale », en augmentant petit à petit la taille de l'entrée considérée.

Il est à noter que ces codes sont appelés des codes *comma free* dans des ouvrages de théorie des codes (par exemple [BP85]).

Définition 30 (Classe MDPC) Une machine $\hat{f}$ appartient à la classe des machines $\mathcal{MDPC}$ si et seulement si la machine $\hat{f}$ vérifie les conditions suivantes :

$$\forall u, v, y, \quad \left.\begin{array}{c} \hat{f}\,\langle v, y\rangle \downarrow \\ \hat{f}\,\langle u \cdot v \cdot w, y\rangle \downarrow \end{array}\right\} \Rightarrow u = w = \varepsilon.$$

Cette définition ne s'étend pas facilement à des catégories comme les machines doublement préfixes pleines. La concaténation de deux mots du code ne pouvant pas appartenir au domaine de la fonction, il n'est pas direct de concevoir une expression qui puisse réunir les deux concepts sans diminuer grandement l'intérêt de la classe considérée.

3.2 Restriction de la calculabilité

Les définitions que nous avons données représentent des ensembles cohérents de machines, réunies par une propriété commune. Il paraît naturel de s'interroger pour savoir s'il est possible de restreindre l'étude de ces machines à elles-mêmes, sans considérer des machines ayant d'autres propriétés. Les théorèmes classiques de calculabilité peuvent ainsi être modifiés pour essayer de refléter une indépendance entre l'ensemble de toutes les machines, et la classe considérée. De plus, dans les quatre cas précités, la propriété porte uniquement sur les fonctions calculées. On désignera par le vocable « classe de fonction » l'ensemble des fonctions vérifiant une certaine propriété, et « classe de machines » l'ensemble des machines dont la fonction calculée vérifie la même propriété. Dans [MY78], une notation différente est introduite pour les deux classes ($\mathcal{C}$ désignant la classe de fonctions et $P_{\mathcal{C}}$ désignant la classe de machines), mais cette distinction ne semble pas nécessaire ici. Éventuellement, on notera $\mathcal{C}$ la classe des fonctions pour la distinguer de la classe des machines $\hat{\mathcal{C}}$.

3.2.1 Représentabilité et universalité

On a déjà évoqué les systèmes acceptables de programmation (cf. définition 4). Ceux-ci capturent l'ensemble des conditions nécessaires pour décrire un système de calcul complet. Notre désir est de voir comment on peut définir un sous-sytème qui conserve de nombreuses

propriétés des systèmes acceptables de programmation, et pour cela, il faut affaiblir une partie des conditions sur les objets considérés. La définition est composée de trois parties : représentabilité, universalité, composition.

La représentabilité reste une condition que l'on peut difficilement modifier. Nous allons considérer un sous-ensemble de la classe des machines qui est défini par une propriété sur la fonction calculée par ces machines, et non par le mécanisme spécifique de ces machines. Cela a pour conséquence immédiate, par l'application du théorème de Rice, qu'il est indécidable de déterminer si une machine appartient à la classe. L'important est surtout de disposer d'une représentation d'un sous-ensemble de la classe qui suffit à décrire toutes les fonctions calculées par une machine. On propose donc la formulation suivante :

Définition 31 (représentabilité) Soit $\hat{\mathcal{C}}$ une classe de machines d'un système acceptable de programmation. $\hat{\mathcal{C}}$ vérifie la propriété de **représentation effective** si et seulement s'il existe une fonction récursive totale f de $\mathbb{N} \to \mathbb{N}$, telle que pour tout $n \in \mathbb{N}$, la machine $\hat{\phi}_{f(n)}$ soit dans $\hat{\mathcal{C}}$, et que pour toute fonction M calculée par une machine $\hat{M}$ de $\hat{\mathcal{C}}$, il existe n tel que $\phi_{f(n)} = M$ (ce qui peut s'écrire $\left\{\phi_{f(i)}\right\}_{i\in\mathbb{N}}$ est égal à l'ensemble des fonctions calculées par $\hat{\mathcal{C}}$). La fonction f est alors une *représentation effective* de la classe $\hat{\mathcal{C}}$, et $\left\{\phi_{f(i)}\right\}_{i\in\mathbb{N}}$ est une *énumération effective* ou une *représentation* de $\hat{\mathcal{C}}$.

La condition d'universalité est plus facile à appréhender. Il faut en fait une fonction qui permette d'associer à toute machine de la classe $\hat{\mathcal{C}}$ une donnée correspondant à son principe de fonctionnement, qu'une machine doit être capable d'exploiter. Toutefois, on ajoute une condition d'internalité à cette simulation (dans le cas contraire, la machine universelle de Turing répond trivialement à cette condition) : on exige que la machine faisant la simulation fasse partie de la sous-classe.

Définition 32 (universalité) Soit $\hat{\mathcal{C}}$ une classe de machines d'un système acceptable de programmation. $\hat{U}$ est une machine universelle pour $\hat{\mathcal{C}}$ si et seulement si $U \in \hat{\mathcal{C}}$, et pour toute machine $\hat{\phi}_n \in \hat{\mathcal{C}}$, $U\langle n, x\rangle = \phi_n(x)$.

Enfin, la troisième condition, celle de composition, peut être gardée intacte. Il semble normal que toutes les classes raisonnables aient besoin d'interagir, et l'existence d'une opération interne de composition est naturelle. On peut donc donner l'expression suivante :

Définition 33 (composabilité) Soit $\hat{\mathcal{C}}$ une classe de machines. Elle vérifie la propriété de **composabilité** si et seulement s'il existe une fonction c de composition récursive totale de $\mathbb{N}^2 \to \mathbb{N}$, telle que pour tout couple i, j les deux propriétés suivantes soient vérifiées :

$$\begin{cases} \phi_i \circ \phi_j = \phi_{c\langle i,j\rangle} \\ \hat{\phi}_i \in \hat{\mathcal{C}} \text{ et } \hat{\phi}_j \in \hat{\mathcal{C}} \Longrightarrow \hat{\phi}_{c\langle i,j\rangle} \in \hat{\mathcal{C}} \end{cases} .$$

3.2.2 Théorèmes classiques de la calculabilité

Considérons d'abord le cas de deux théorèmes classiques de la calculabilité : le théorème de Rice et le premier théorème du point fixe (aussi appelé théorème de la récursion).

Trouver une extension du théorème *s-m-n* sera traité plus loin, après avoir discuté de l'arité[1] des fonctions. On peut aussi citer dès maintenant la possibilité d'étendre à une sous classe le théorème fondateur de la théorie de la complexité de Kolmogorov : l'équation de Solomonoff-Kolmogorov, c'est-à-dire la propriété de l'additive optimalité.

Le premier théorème du point fixe statue ceci : pour toute fonction f p.p.r., il existe n, récursif en $\langle f \rangle$ telle que $\phi_n = \phi_{f(n)}$. Pour étendre cette propriété, il serait idéal de pouvoir confiner le choix de n à une machine qui appartient à $\hat{\mathcal{C}}$. Ce n'est pas possible. Prenons par exemple la fonction constante $P_{=n}$, telle que n soit le numéro d'une machine qui n'appartient pas à la classe $\hat{\mathcal{C}}$ ($\hat{\phi}_n \notin \hat{\mathcal{C}}$). Le théorème du point fixe n'est donc pas vrai tel quel — il faut l'adapter pour l'étendre.

La fonction dont on cherche le point fixe dans ce théorème est en fait une transformation de machines. L'idée que l'on ajoute est donc qu'il doit être possible de restreindre cette même transformation. Le problème est identique pour le cas du théorème de Rice. Le résultat que l'on retient est basé sur la notion de représentation effective :

Définition 34 (point fixe) Une classe de machines $\hat{\mathcal{C}}$ vérifie la propriété dite du **point fixe** si et seulement pour toute représentation effective α et pour toute fonction p.p.r. f, il existe n, récursif en $\langle f \rangle$ et en $\langle \alpha \rangle$, tel que $\phi_n = \phi_{\alpha(f(n))}$, et $\hat{\phi}_n \in \hat{\mathcal{C}}$.

Comme α est une représentation effective, il est à noter que $\hat{\phi}_{\alpha(f(n))} \in \hat{\mathcal{C}}$.

Définition 35 (Rice) Soit une classe de machines $\hat{\mathcal{C}}$. La propriété de **Rice** est vérifiée pour $\hat{\mathcal{C}}$ si et seulement si pour tout ensemble F inclus dans l'ensemble des fonctions et pour toute représentation effective α de $\hat{\mathcal{C}}$, l'ensemble $\left\{i, \phi_{\alpha(i)} \in F\right\}$ est soit non-récursif, soit trivial ($\emptyset$ ou $\mathbb{N}$).

Une branche de la théorie de la calculabilité est celle qui s'intéresse à la description des objets, en particulier via la complexité de Kolmogorov. Il se trouve que l'une des équations les plus caractéristiques de la complexité de Kolmogorov contient en elle-même la notion de sous-classe. On rappelle donc la définition de machine additivement optimale :

Définition 36 (Machine additivement optimale) Une machine $\hat{\psi}$ est additivement optimale pour $\hat{\mathcal{C}}$ si et seulement si elle vérifie $\hat{\psi} \in \hat{\mathcal{C}}$ et :

$$\forall \hat{\psi} \in \hat{\mathcal{C}}, \exists c_\phi, \forall x, y \in \Xi, \quad \mathbf{K}_\psi(x|y) \leqslant \mathbf{K}_\phi(x|y) + c_{\hat{\phi}}.$$

Définition 37 (Additive optimalité) La propriété de **l'additive optimalité**, ou propriété de **Solomonoff-Kolmogorov**, est vérifiée pour une classe de machines $\hat{\mathcal{C}}$ si et seulement si il existe une machine additivement optimale pour $\hat{\mathcal{C}}$.

Le théorème *s-m-n*, par contre, n'a pas besoin d'être adapté. Ce théorème véhicule la notion d'application partielle, permettant de transformer un programme utilisant un certain nombre de variables en un programme ou les valeurs d'une partie de ces variables

1. Arité : nombre d'arguments d'une fonction

est fixée. Cette transformation est récursive. Mais cette notion est fortement liée à la façon dont on va transformer et générer les fonctions d'arité supérieure à 1. Cela se fait par des *couplages*, qui sont définis par [MY78] comme étant des bijections récursives primitives de $\Xi^2 \to \Xi$, strictement monotones en leurs deux arguments.

Définition 38 (Couplage) On appelle couplage toute bijection κ primitive récursive de $\Xi^2 \to \Xi$, qui vérifie que $\kappa(x+1,y) > \kappa(x,y)$ et $\kappa(x,y+1) > \kappa(x,y)$ (ordre longueur-lexicographique).
On définit à partir d'un couplage les couplages d'ordre supérieurs, définis comme suite :

$$\begin{array}{rll} \kappa_n : & \Xi^n & \to \Xi \\ & x_1, \ldots, x_n & \mapsto \kappa(x_1, \kappa_{n-1}(x_2, \ldots, x_n)) \end{array}$$

On prend bien évidemment $\kappa_2 = \kappa$, $\kappa_1 = \mathrm{Id}$. On se fixe un couplage κ de référence. Par abus de notation, une fonction f de Ξ dans Ξ pourra être écrite $f(x_1, \ldots, x_n)$ où il sera sous-entendu en fait $f(\kappa_n(x_1, \ldots, x_n))$. Avec cet abus de notation, on peut n'utiliser que des fonctions récursives d'arité 1.

On peut alors facilement donner le théorème *s-m-n*. Celui-ci se démontre d'ailleurs à partir de l'existence d'une part du couplage et de certaines fonctions récursives primitives qui y sont liées, et de la récursivité de la composition d'autre part.

Proposition 0.38 (*s-m-n*) *Il existe une fonction s récursive totale de $\Xi \to \Xi$ qui vérifie la propriété suivante :*

$$\phi_a(x_1, \ldots, x_m, y_1, \ldots, y_n) = \phi_{s(a,m,x_1,\ldots,x_m)}(y_1, \ldots, y_n)$$

◇ *Preuve.* On commence par simplifier le problème : il existe une fonction récursive primitive Con qui vérifie :

$$\mathrm{Con}(m, \kappa_m(x_1, \ldots, x_m), \kappa_n(y_1, \ldots, y_n)) = \kappa_{n+m}(x_1, \ldots, x_m, y_1, \ldots, y_n).$$

Il suffit donc de prouver l'existence de s totale récursive telle que :

$$\phi_{s(i,m,x)} = \phi_i(\mathrm{Con}(m, x, y)).$$

L'étape suivante consiste à définir R vérifiant $\phi_{R(x)}(y) = \kappa(x,y)$. On prend P telle que $P(y) = \kappa(0,y)$, et Q telle que $Q(x,y) = \kappa(x+1,y)$. Alors la fonction R définie par $R(0) = \langle P \rangle$ et $R(x+1) = c(\langle Q \rangle, R(x))$ vérifie la propriété demandée. De plus, R est récursive puisque c est récursive totale. Enfin, on définit s par $s(i,m,x) = c(i, c(\langle \mathrm{Con} \rangle, c(R(m), R(x))))$ En développant la fonction de composition c, on obtient bien que

$$\begin{aligned} \phi_{s(i,m,x)}(y) &= \phi_i \circ \phi_{\langle \mathrm{Con} \rangle} \circ \phi_{R(m)} \circ \phi_{R(x)} \\ &= \phi_i \circ \mathrm{Con} \circ \kappa(m, \kappa(x,y)). \\ &= \phi_i(\mathrm{Con}(m,x,y)). \end{aligned}$$

□

Tel quel, le théorème *s-m-n* ne mentionne pas de sous-classe, et reste donc vrai. Si l'on garde un couplage quelconque, il est difficilement envisageable que toutes les propriétés d'une fonction soient conservées, notamment pour la question d'être préfixe. Ainsi, il n'y a pas de raison qu'un continuation de $\kappa(a,b)$ soit le couplage d'une continuation de a et de b. La bonne question est donc de savoir s'il est possible de définir un couplage différent de façon adéquate pour que la notion de préfixe soit conservée. On peut regarder quelles seraient les conditions sur le couplage μ qui pourraient aider à vérifier la propriété suivante : étant donnée une machine ϕ_a qui est préfixe, il faudrait que l'application partielle de variable (l'utilisation de la fonction *s-m-n*) soit conservative pour la propriété préfixe, c'est-à-dire que l'on ait $\phi_s(a,x)$ qui soit aussi une fonction préfixe. Comme on vient de le voir, la fonction *s-m-n* dépend uniquement de la composition, qui est imposée par le système acceptable de programmation choisi, et du codage de référence. Comme cette propriété n'est pas vraie pour tous les couplages, il faut pouvoir faire intervenir un couplage différent. La question d'internalité peut donc se reposer sous la forme suivante : ayant pris toute valeur possible pour un élément x, le *s-m-n* interne est vérifié s'il existe μ un couplage tel que $\phi_{s(a,x)}(y) = \phi_a(\mu(x,y))$, et que l'on ait pour toute valeur possible de x, une fonction appartenant à la même classe pour $\phi_{s(a,x)}$. Comme le s dépend du couplage de référence, cette égalité n'est pas vérifiable par la même fonction *s-m-n*. On peut donc définir la nouvelle fonction de deux façons, à partir du couplage de référence κ — on écrit alors que $\mu = \mu' \circ \kappa$, et on définit [2] $s_\mu(a,x) = s(c(a,\langle\mu'\rangle),x)$, qui vérifie bien $\phi_{s(a,x)}(y) = \phi_a(\mu(x,y))$ —, ou alors en remplaçant κ par μ dans la démonstration de l'existence de $fonctions$ de la proposition 0.38, ce qui revient au même. Il semble peu probable que cette propriété puisse être vérifiée si l'on n'a pas une certaine quantité de régularité dans le couplage : supposons que l'ajout d'un suffixe v à une valeur convergente y pour un x donné et un a adéquat essaye de vérifier la propriété préfixe ; on se retrouve à calculer donc $\phi_a(\mu(x, y\cdot v))$. Si μ n'a aucune propriété de régularité, il n'y a pas de raison que $\mu(x, y\cdot v)$ ait un lien quelconque avec $\mu(x,y)$, et en particulier d'en être un préfixe. Si bien que toute information sur ϕ_a est *a priori* inutile... et donc on ne peut en espérer une preuve. On peut donc écrire une propriété interne possible :

Définition 39 (*s-m-n* interne) Soit une classe de machines $\hat{\mathcal{C}}$. La propriété ***s-m-n* interne** est vérifiée si et seulement s'il existe un couplage μ tel que :

$$\hat{\phi}_a \in \hat{\mathcal{C}} \implies \forall x \in \Xi,\ \hat{\phi}_{s_\mu(a,x)} \in \hat{\mathcal{C}},$$
$$\forall a,x,y \in \Xi,\ \phi_{s_\mu(a,x)}(y) = \phi_a(\mu(x,y)).$$

On voit bien que cette propriété reflète un résultat sur les couplages : il suffit que le couplage conserve la propriété qui définit $\hat{\mathcal{C}}$ pour que la propriété découle naturellement de l'existence de la composition. L'étude des couplages peut amener à se poser des questions sur la nature réelle de l'arité des fonctions. En particulier, on peut se poser la question de savoir si considérer une fonction d'arité 2 modifie ou non la perception que l'on peut en avoir, et ses propriétés. En particulier, la conservation des longueurs à travers un couplage

2. On note $c(a,b)$ la fonction de composition ($\phi_{c(a,b)} = \phi_a \circ \phi_b$).

n'est pas possible, comme le montre la proposition suivante :

Proposition 0.39 *Soit κ une bijection de Ξ^2 dans Ξ. Alors κ n'est pas sous-additive au sens où il n'existe pas de constante c telle que :*

$$\forall x, y \in \Xi, \ell(\kappa(x, y)) \leqslant \ell(x) + \ell(y) + c.$$

◇ *Preuve.* On compte les couples de mots dont la somme des longueurs est inférieure ou égale à une constante donnée. Le nombre de couples de mots ayant une longueur totale exactement égale à k est égal à la somme $\sum_{i=0}^{k} |\xi|^i |\xi|^{k-i}$, soit $\sum_{i=0}^{k} |\xi|^k$, soit exactement $(k+1)|\xi|^k$. Le nombre de tous les couples de mots de somme de longueur au plus égale à n est donc

$$\begin{aligned}\sum_{k=0}^{n}(k+1)|\xi|^k &= \frac{|\xi|^{n+1}((|\xi|-1)(n+1)-1)+1}{(|\xi|-1)^2} \\ &\geqslant n|\xi|^{n+1}/(|\xi|-1).\end{aligned}$$

Si une constante c vérifiant les conditions du théorème existait, alors tous les couples de longueur au plus n devraient se projeter via la bijection sur des mots de longueur au plus $n+c$, dont le nombre est de $\frac{|\xi|^{n+c+1}-1}{|\xi|-1}$. Or il existe au moins $\frac{n|\xi|^{n+1}}{|\xi|-1}$ tels couples. Pour $n \geqslant |\xi|^c$, on a une contradiction. □

Beaucoup plus proche de l'étude du théorème *s-m-n*, il est aussi possible de se demander si la combinaison de deux informations peut être possible tout en gardant la propriété préfixe. Une première réponse (d'autres seront abordées dans le chapitre suivant, voir la partie 4.2) est donnée par cette propriété de *s-m-n* interne.

3.3 Le cas des sous-classes préfixes

Les propriétés que l'on vient de définir posent les bases de l'étude de la calculabilité sur des sous-systèmes. Pour toutes les définitions de machines préfixes que l'on a pu voir, et pour les quatre sous-classes qui en découlent, essayons de répondre exactement à la question : la classe de machines $\hat{\mathcal{C}}$ possède-t-elle ou non la propriété ?

3.3.1 Projecteur

La première propriété, et la plus utile, que nous étudions est la représentabilité. En effet, on peut montrer pour les classes $\mathcal{MPC}$, $\mathcal{MPP}$, $\mathcal{MPG}$ et $\mathcal{MDPC}$ une propriété plus puissante que la propriété de **représentation effective**, qui est l'existence d'un *projecteur*, défini comme suit :

Définition 40 (projecteur) Soit une classe de machines $\hat{\mathcal{C}}$, et une fonction récursive α de $\mathbb{N}$ dans $\mathbb{N}$. Alors α est un projecteur pour la classe $\hat{\mathcal{C}}$ si et seulement si pour toute machine $\hat{\phi}_k$, $\hat{\phi}_{\alpha(k)}$ appartient à $\hat{\mathcal{C}}$, et que si $\phi_k \in \hat{\mathcal{C}}$, $\phi_k = \phi_{\alpha(k)}$. [3]

On peut remarquer qu'un projecteur est en particulier une fonction d'énumération effective. Mais la propriété supplémentaire est qu'une fonction est laissée invariante lorsqu'elle est calculée par une machine de la classe considérée. Attention : la machine elle-même est modifiée, c'est la fonction qu'elle calcule qui reste inchangée. Prouvons donc cette proposition pour chacune des classes.

Proposition 0.40 *Il existe un projecteur pour les classes $\mathcal{MPC}$, $\mathcal{MPP}$, $\mathcal{MPG}$ et $\mathcal{MDPC}$.*

◇ *Preuve.* On va transformer chaque machine de façon à laisser inchangée les fonctions vérifiant déjà les conditions préfixes propres à la classe considérée. Soit donc ϕ une machine, u et y deux mots. On va simuler le calcul de ϕ sur des entrées de plus en plus grandes à y constant pendant de plus en plus longtemps selon la procédure exposée plus loin, et lorsque l'on trouve un calcul convergent pour l'un des préfixes de u ou l'une de ses continuations, on décide dans une première phase que seul ce résultat peut être le bon. Dans une deuxième phase, on fait des vérifications supplémentaires jusqu'à trouver la convergence d'un préfixe du mot u (ou le mot u lui-même). On s'assure ainsi que toute continuation du deuxième candidat respecte les règles préfixes que l'on s'est fixé pour la classe considérée. Puisque les classes étudiées sont des fonctions à deux arguments (préfixes en le premier argument), mais que les propriétés à vérifier se limitent au premier argument, on se place à y constant.

Candidats La machine $\hat{\phi}_{\alpha(k)}$ va procéder de la façon suivante : elle va énumérer des *candidats*, qui sont des résultats de calculs convergents dans un ordre déterminé sur des entrées distinctes de la machine ϕ_k. On distinguera ensuite dans ces candidats les candidats *valides* des candidats *non valides*. Enfin, on arrête la construction de la liste des candidats valides à certaines conditions (variant selon la classe), et on en détermine tout de suite la convergence et le résultat éventuel du calcul. Un candidat est désigné par un triplet $(u, t) \to v$. Le u est le mot d'entrée ayant obtenu la convergence en au plus t pas, le résultat produit étant v. On désigne les candidats valides comme étant ceux qui vérifient que $t > \ell(u)$. Ainsi, si sur l'entrée 000110 l'arrêt de la machine est obtenu par la simulation de 4 pas de calcul et s'arrête sur le résultat 00, le candidat $(000110, 4) \to 00$ n'est pas valide, alors que $(000110, 6) \to 00$ l'est. L'ensemble des entrées examinée est restreint à l'ensemble des préfixes et continuations de u $\mathcal{L}_u$. Pour la classe $\mathcal{MDPC}$, l'ensemble examiné est modifié : on n'examine pas seulement l'ensemble des préfixes et des continuations, mais l'ensemble de tous les sous-mots de u et tous les mots dont u est un sous-mot. Le reste est inchangé. Cet ensemble comprend ainsi ε, $\mathsf{u}_{\langle 1\rangle}$, $\mathsf{u}_{\langle 1\rangle}\mathsf{u}_{\langle 2\rangle}$,..., u, et l'ensemble de tous les mots de la forme $u \cdot w$, avec $w \in \Xi$.

L'ordre d'examen des candidats est le suivant : on examinera toujours les candidats par ordre de temps de calcul simulé croissant. Pour un même temps de simulation, on classe par ordre longueur-lexicographique (les mots plus courts d'abord, puis pour une même longueur un ordre lexicographique arbitraire). On simule donc $\hat{\phi}_k$ pendant t pas sur l'entrée v pour t croissant et v parcourant l'ensemble $\{w \in \mathcal{L}_u, \ell(w) \leqslant t\}$. On part

3. Ou, autrement dit : si $\hat{f} \in \hat{\mathcal{C}}, \phi_{\alpha(\langle f\rangle)} = f$, et $\forall f, \hat{\phi}_{\alpha(\langle f\rangle)} \in \hat{\mathcal{C}}$.

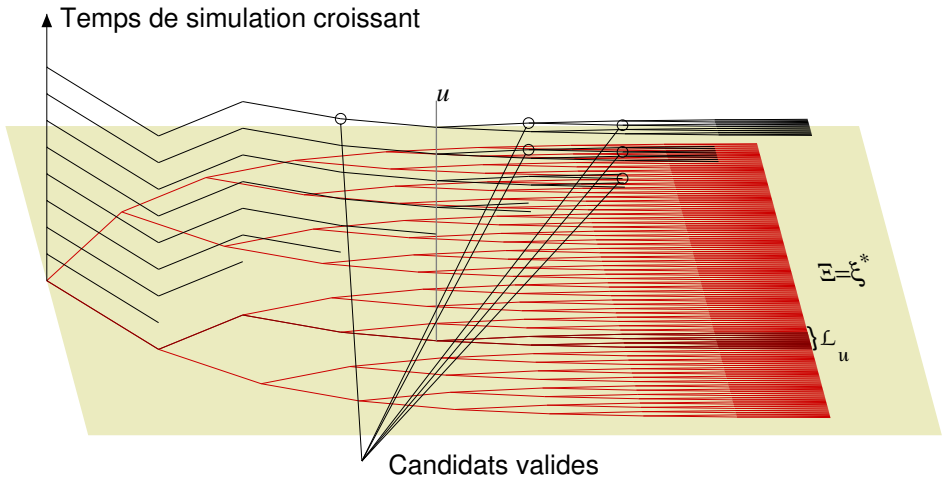


FIGURE 3.1 – Parcours de l'ensemble des candidats ($u = 1011$)

de $t = 0$, et on l'augmente dès que l'on a épuisé tous les mots possibles, comme montré figure 3.3.1. On établit ainsi une liste progressive de tous les calculs convergents. Une même entrée u est potentiellement réutilisée pour chaque t tel que $\ell(u) \leqslant t$.

Détermination de la convergence L'algorithme d'énumération se poursuit tant qu'un candidat de la forme requise par la classe préfixe considérée n'a pas été trouvé. On note chacun de ces candidats sous la forme $(x_i, t_i) \to y_i$:

(i) Si la classe est $\mathcal{MPC}$ ou $\mathcal{MDPC}$, alors l'énumération s'arrête dès que le premier candidat valide est trouvé ;

(ii) Si la classe est $\mathcal{MPP}$, alors l'énumération s'arrête dès qu'un candidat valide est trouvé sous la forme $(x_n, t_n) \to y_n$ avec x_n un préfixe de u (ou éventuellement u lui-même) ;

(iii) Si la classe est $\mathcal{MPG}$, alors l'énumération ne s'arrête que si un candidat $(u, t_n) \to y_n$ est trouvé.

Le résultat de la convergence est déterminée comme suit :

— S'il y a eu plusieurs candidats énumérés, on ne converge que si les y_i ont été tous identiques ;

— Si la première condition a été remplie, le résultat est donc le résultat commun à tous les candidats si le calcul est convergent ;

— Si la classe est $\mathcal{MPC}$ ou $\mathcal{MDPC}$, on converge si le premier (et le seul) candidat est u ;

— Si la classe est $\mathcal{MPP}$ ou $\mathcal{MPG}$, on converge si l'énumération finit par s'arrêter.

Garantie de la qualité préfixe Supposons que l'on ait u et v deux mots. Dans le cas géné-

ral, on veut garantir que si u et $u \cdot v$ convergent sur la machine $\phi_{\alpha(k)}$, alors la valeur est la même. Supposons que la convergence de u ait été décidée par le mot x_n et celle de $u \cdot v$ par x'_m. Supposons aussi que x_n et x'_m soient différents (sinon, le résultat est évident). x'_m est forcément une continuation de x_n : x_n étant un préfixe de u et x'_m de $u \cdot v$, le seul autre cas serait que x'_m soit un préfixe de x_n. Si c'était le cas, $(x_n, t_n) \to y_n$ étant un candidat valide pour $u \cdot v$ aussi bien que pour u et en étant aussi un préfixe, cela voudrait dire que $(x_n, t_n) \to y_n$ serait produit après $(x'_m, t'_m) \to y'_m$ lors du calcul mené pour $u \cdot v$; ce qui est contradictoire avec le fait qu'il est le premier candidat préfixe valide lors du calcul de u (la validité et l'ordre d'apparition des candidats est indépendant de l'entrée de $\hat{\phi}_{\alpha(k)}$). Les deux candidats finaux sont différents, c'est donc que x'_m est lui une continuation stricte de u, pour qu'il n'ait pas suffi à déclencher l'arrêt lors du calcul pour u mais qu'il ait arrêté le calcul pour $u \cdot v$. De plus, il doit se présenter avant dans l'ordre des candidats. Or, si les résultats y'_m et y_n avaient été différents, la règle *(3.3.1)* s'appliquerait, et il n'y aurait pas de convergence. Donc le résultat est bien le même. Donc, toute machine $\hat{\phi}_{\alpha(k)}$ appartient bien à $\mathcal{MPG}$. Dans le cas particulier de la classe $\mathcal{MPC}$, la règle *(3.3.1)* garantit que la convergence n'est que si l'entrée u est elle-même le premier candidat. Or ce n'est pas le temps de convergence que l'on regarde, mais aussi la longueur de l'entrée : une continuation de u qui est son propre premier candidat apparaît forcément comme candidat de tous ses préfixes avant eux-mêmes. Elle converge en effet plus vite qu'eux (leurs temps de convergence sont supérieurs à sa longueur), et ils ne pourront être candidats pour eux-mêmes qu'après l'avoir simulé pendant au moins son temps de convergence. Donc α est bien une projection sur la classe $\mathcal{MPC}$ (pour l'algorithme modifié comme indiqué). Dans le cas particulier de la classe $\mathcal{MPP}$, la règle *(3.3.1)* garantit aussi la projection : si un préfixe converge, alors on s'aligne sur sa valeur. Dans le cas particulier de la classe $\mathcal{MDPC}$, si deux mots u et v convergent et que u est un sous-mot de v, cela veut dire que chacun apparaît en premier dans sa liste de candidats. Or, u est inclus dans la liste des mots examinés par v, et v est inclus dans la liste des mots examinés par u. Un seul peut être le premier candidat à converger, donc u et v sont identiques.

Invariance de la classe On va vérifier que pour $\hat{\phi}_k \in \hat{\mathcal{C}}$, $\hat{\phi}_{\alpha(k)}$ appartient encore à $\hat{\mathcal{C}}$. $\hat{\mathcal{C}}$ peut être indifféremment $\mathcal{MPC}$, $\mathcal{MPG}$ ou $\mathcal{MPP}$. De par les inclusions, on peut déjà voir que le cas *(3.3.1)* n'est jamais atteint. Si pour une entrée u, $\phi_k(u)$ converge, comme le cas *(3.3.1)* n'est jamais atteint, alors la convergence se fait par l'autre règle, car on finira par trouver le candidat $(u, t_u) \to \phi_k(u)$. Pour la classe $\mathcal{MPC}$, la définition de la classe implique que le candidat u est forcément le premier ; pour la classe $\mathcal{MPG}$, la condition est automatiquement vérifiée (et la valeur est inchangée) ; enfin, pour la classe $\mathcal{MPP}$, ce peut être un préfixe qui décide de la convergence, mais par définition la fonction devait aussi converger pour u. La condition sur la classe $\mathcal{MDPC}$ est similaire à celle de la classe $\mathcal{MPC}$ et donne le même résultat. Enfin, si pour une entrée u, $\hat{\phi}_k(u)$ diverge, alors la condition d'arrêt n'est jamais atteinte (sauf éventuellement pour la classe $\mathcal{MPP}$, mais c'est absurde pour la raison évoquée ci-dessus). Donc les fonctions calculées par des machines appartenant à une classe préfixe sont conservées par α.

□

L'existence de ce projecteur montre donc que non seulement il est possible de donner une représentation effective de chacune de ces classes, mais aussi qu'il est possible de contraindre cette représentation à une forme de stabilité (du point de vue des fonctions calculées, elle est idempotente). En particulier, on peut tout de suite conclure :

Corollaire 0.41 *La propriété de* ***représentation effective*** *est vérifiée pour les classes de machines préfixes et doublement préfixes ($\mathcal{MPC}$, $\mathcal{MPP}$, $\mathcal{MPG}$, $\mathcal{MDPC}$).*

3.3.2 Universalité

L'universalité est en fait une des propriétés les plus importantes pour utiliser une classe de machines. Heureusement, elle est prouvable pour toutes les classes que l'on considère :

Proposition 0.42 *Il existe une machine universelle pour les classes de machines préfixes et doublement préfixes ($\mathcal{MPC}$, $\mathcal{MPP}$, $\mathcal{MPG}$, $\mathcal{MDPC}$).*

◇ *Preuve.* Le mécanisme de chacune des preuves est toujours le même : construire une fonction récursive f qui encode les deux données n et x selon un format qui permet de garder les propriétés de la classe, et exécuter la machine $\phi_{\alpha(n)}$ sur l'entrée x (et donc effectuer le décodage). Il faut donc essentiellement vérifier les propriétés de la classe pour la fonction considérée. Si la donnée en entrée n'est pas dans l'image de f, et qu'on ne peut pas la décoder, la machine diverge. Dans tous les cas, on pose donc la machine $\hat{U}_{\hat{c}}$ comme étant $\phi_{c(\langle U\rangle,c(\langle\kappa\rangle,\langle f^{-1}\rangle))}$, où c, κ, $\hat{f}^{-1}$ et $\hat{U}$ sont respectivement les fonctions de composition, de couplage standard, une machine calculant la fonction réciproque de f qui diverge si l'entrée n'est pas dans l'image de f et une machine universelle pour le calcul. On considérera α comme étant le projecteur pour la classe choisie comme défini dans la proposition 0.40.

$\underline{\mathcal{MPC}, \mathcal{MPP}, \mathcal{MPG}}$ On pose $f\langle n, x\rangle = \overline{n} \cdot x$. De par la définition de α, et parce que $n \to \overline{n}$ est un codage préfixe, la machine s'arrête sur un préfixe d'une autre valeur de convergence uniquement si la valeur de n est la même, c'est-à-dire si le n est identique. Comme ce qui est simulé est la machine $\hat{\phi}_{\alpha(n)}$ qui appartient à la classe, elle respecte donc bien la propriété d'origine (le α est différent pour chacune des classes).

$\underline{\mathcal{MDPC}}$ On pose $f\langle n, x\rangle = 111 \cdot (01)^{n+1} \cdot (10)^{x+1} \cdot 000$. La machine $\hat{U}_{\mathcal{MDPC}}$ n'accepte que les entrées de la forme définie par f. Montrons que si elle converge sur un sous-mot défini par (n, x), elle ne converge pas pour un autre mot qui serait un sous-mot propre (ou un sur-mot). Soit (m, y) un autre couple sur lequel la machine converge. Parce qu'il n'y a pas d'autre séquence 111 qu'au début, $f\langle n, x\rangle$ et $f\langle m, y\rangle$ commencent au même point. De même pour la séquence 0000 finale. Donc $\ell(f\langle n, x\rangle) = \ell(f\langle m, y\rangle)$; Ils sont identiques. Comme il est évident que f est récursive, la preuve est conclue.

□

De plus, la propriété de composabilité est bien vérifiée pour les classes de mahines préfixes :

Proposition 0.43 *La propriété de* ***composabilité*** *est vérifiée pour les classes de machines préfixes et doublement préfixes ($\mathcal{MPC}$, $\mathcal{MPP}$, $\mathcal{MPG}$, $\mathcal{MDPC}$).*

◇ *Preuve.* En fait, seule la condition sur la deuxième fonction est utile. Pour que $f \circ g(x)$ converge, il faut en effet que $g(x)$ converge. Donc si $f \circ g(x \cdot y)$ converge aussi, c'est le cas pour $g(x \cdot y)$, et les règles s'appliquent selon la classe de g. Comme de plus, la valeur rendue pour deux entrées préfixes l'une de l'autre est la même (dans le cas de $\mathcal{MPP}$ et $\mathcal{MPG}$), c'est la même valeur qui est donnée en entrée à f, donc aussi la même valeur comme résultat. □

Donc les quatre classes de machines préfixes vérifient toutes les principales propriétés de la calculabilité.

3.3.3 Autres propriétés

Les autres propriétés que nous avons définies sont aussi vérifiables dans la plupart des cas. Ainsi, les propriétés du point fixe et de Rice sont toutes les deux vérifiables :

Proposition 0.44 *La propriété du* ***point fixe*** *est vérifiée pour toutes les classes de machines, qui sont définies par l'ensemble des machines calculant une classe de fonctions possédant au moins une énumération effective. En particulier, ceci est vérifié pour les classes de machines préfixes et doublement préfixes ($\mathcal{MPC}$, $\mathcal{MPP}$, $\mathcal{MPG}$, $\mathcal{MDPC}$).*

◇ *Preuve.* Soit α une énumération effective. En appliquant le théorème du point fixe à la fonction $\alpha \circ f$, on obtient n tel que $\phi_n = \phi_{\alpha(f(n))}$. Par définition de α, $\phi_{\alpha(f(n))} \in \hat{\mathcal{C}}$. Comme toutes les classes de machines considérées contiennent toutes les machines calculant une même fonction, cela veut dire que $\hat{\phi}_n$ est aussi dans $\hat{\mathcal{C}}$. □

Proposition 0.45 *La propriété de* ***Rice*** *est vérifiée pour les classes de machines préfixes et doublement préfixes ($\mathcal{MPC}$, $\mathcal{MPP}$, $\mathcal{MPG}$, $\mathcal{MDPC}$).*

◇ *Preuve.* Soit α défini par la proposition 0.40 pour la classe choisie. Supposons que l'ensemble $\left\{i, \phi_{\alpha(i)} \in F\right\}$ soit récursif. Cela veut dire qu'il existe une machine $\hat{M}$ telle que $M(x) = 1$ si $\phi_{\alpha(x)} \in F$, $M(x) = 0$ si $\phi_{\alpha(x)} \notin F$. Supposons maintenant que l'on ait deux entrées u et v, telles que $M(u) = 1$ et $M(v) = 0$, et que $\hat{\phi}_u$ et $\hat{\phi}_v$ soient dans $\hat{\mathcal{C}}$. Soit la machine $\hat{T}$ suivante : sur l'entrée $\langle a, b\rangle$, elle rend le résultat suivant :

$$T\langle a, b\rangle = \begin{cases} \phi_{\alpha(v)}(b) & \text{si } M(a) = 1 \\ \phi_{\alpha(u)}(b) & \text{si } M(a) = 0 \end{cases}.$$

Soit $\hat{S}$ la machine définie à l'aide du théorème de récursion telle que $\langle S\rangle$ étant le point fixe de la fonction $n \mapsto s(\langle T\rangle, n)$; on a alors $S(x) = T\langle\langle S\rangle, x\rangle$. Supposons que $M(\langle S\rangle) = 1$. Cela veut dire que pour tout x, $S(x) = \phi_{\alpha(v)}(x) = \phi_v(x)$. Ce qui signifie que $S \notin F$, et

$\hat{S} \in \hat{\mathcal{C}}$. Comme S est dans $\hat{\mathcal{C}}$, $\phi_{\alpha(\langle s \rangle)} = \phi_{\langle s \rangle}$. Donc $M(\langle S \rangle) = 0$, ce qui est absurde. De même, si $M(\langle S \rangle) = 0$, $S \in F$, et donc $M(\langle S \rangle) = 1$, ce qui est absurde. Supposons donc qu'il n'existe pas de u tel que $M(u) = 1$ et $\phi_u \in \hat{\mathcal{C}}$. Cela veut dire que soit F est vide, soit $\hat{\mathcal{C}}$ est contenu dans le complémentaire de F. Donc, cela veut dire que $\left\{i, \phi_{\alpha(i)} \in F\right\}$ est l'ensemble vide. Ou alors, s'il n'existe pas de v tel que $M(v) = 0$ et $\hat{\phi}_v \in \hat{\mathcal{C}}$, cela veut dire que $\left\{i, \phi_{\alpha(i)} \in F\right\}$ est égal à $\mathbb{N}$, et que $\hat{\mathcal{C}}$ contient F. □

Proposition 0.46 *La propriété de* **s-m-n *interne*** *est vérifiée pour les classes des machines préfixes ($\mathcal{MPC}$, $\mathcal{MPP}$, $\mathcal{MPG}$).*

◇ *Preuve.* Pour prouver cette proposition, nous prouvons qu'il existe des codages qui conservent la propriété préfixe. Pour cela, nous montrons qu'il existe μ, un couplage vérifiant la propriété suivante :

$$\exists f, \forall x, y, v \in \Xi, \mu(x, y \cdot v) = \mu(x, y) \cdot f(v). \tag{3.1}$$

Comme μ est un couplage, on en déduit aisément que $f(\varepsilon) = \varepsilon$. Donc $\mu(x, y) = \mu(x, \varepsilon \cdot y) = \mu(x, \varepsilon) \cdot f(y)$. Par ailleurs, $f(y) \neq \varepsilon$ si $y \neq \varepsilon$ (si $f(y) = \varepsilon$, pour tout x, $\mu(x, \varepsilon) = \mu(x, y)$). Il nous faut prouver qu'une telle bijection existe. Soit, d'une part, la fonction f de Ξ dans Ξ qui à chaque mot associe le même mot où on dédouble chaque lettre ($010 \mapsto 001100$). Soit, d'autre part, g la fonction qui à x associe le x-ième mot ne se terminant pas par une lettre dédoublée (avec un ordre d'énumération récursif). Ces deux fonctions sont récursives primitives, de même que leur concaténation. Alors $\mu(x, y) = g(x) \cdot f(y)$ vérifie la propriété (3.1). C'est bien une injection, car étant donné $w = \mu(x, y) = \mu(x', y')$, il existe une façon unique de couper w pour trouver la partie constituée uniquement de doublons et située à la fin du mot ($g(x)$ ne se termine jamais par un doublon), donc $y = y'$, et g est injective, donc $x = x'$. C'est aussi une surjection, puisque pour tout mot on peut séparer les doublons finaux de la partie initiale, et en trouver une interprétation. De plus μ est bien strictement monotone en chacun des arguments. Soit la fonction $s_\mu(a, x)$, égale à $s(c(a, \langle \mu' \rangle), x)$.[4] Étant donné la définition de c, on a $\phi_{s_\mu(a,x)}(y) = \phi_a(\mu(x, y))$. Donc $\phi_{s_\mu(a,x)}(y \cdot v) = \phi_a(\mu(x, y \cdot v) = \phi_a(\mu(x, y) \cdot f(v)$. En résumé, une continuation (stricte) de y dans le calcul de $\phi_{s_\mu(a,x)}(y \cdot v)$ donnera lieu au calcul de $\phi_a(\mu(x, y) \cdot f(v))$, qui est une continuation (stricte) de $\mu(x, y)$. Ainsi, si $\hat{\phi}_a$ est une machine préfixe (pour n'importe laquelle des classes de machines), $\phi_{s_\mu(a,x)}$ vérifiera la même propriété, et $\hat{\phi}_{s_\mu(a,x)}$ appartiendra à la même classe. □

3.3.4 La complexité de Kolmogorov

La complexité de Kolmogorov a fait très tôt l'utilisation de la complexité préfixe (cf. [Cha75]). Il est donc naturel que pour au moins l'une des classes considérées, on puisse répondre positivement sur cette propriété. Mais en fait pour toutes les variantes de la complexité préfixe simple, elle est vérifiée :

4. Il est également possible de démontrer directement l'existence d'une fonction *s-m-n* adéquate à partir de l'existence de c et du codage.

Proposition 0.47 (Additivité optimale) *La propriété* ***de l'additivité optimale*** *est vérifiée pour les classes des machines préfixes ($\mathcal{MPC}$, $\mathcal{MPP}$, $\mathcal{MPG}$).*

◇ *Preuve.* La preuve est exactement identique à celle du théorème 6. On construit la machine $\hat{M}$, en posant :

$$M\left\langle \overline{\beta}\cdot p, y\right\rangle = \phi_{\alpha(\beta)}\left\langle p, y\right\rangle .$$

La fonction α est celle qui a été définie par la proposition 0.40 pour la classe choisie. On ajoute à la définition de $\hat{M}$ que, si la donnée ne correspond pas à des valeurs correctement formatées (de la forme $\overline{\beta}\cdot p$), $\hat{M}$ diverge. On peut construire cette machine, car on peut séparer les trois données β, p et y puisque l'on a un code uniquement décodable pour β. On peut ensuite calculer $\langle p, y\rangle$, puis exécuter la machine universelle sur $\alpha(\beta)$ et sur cette donnée. On vérifie que $\hat{M}$ est bien une machine de la classe $\hat{\mathcal{C}}$, puisque $\hat{\phi}_{\alpha(\beta)}$ en fait partie aussi. Pour toute machine $\hat{\psi}\in\hat{\mathcal{C}}$, on a alors $M\left\langle \overline{\langle\psi\rangle}\cdot p, y\right\rangle = \psi\left\langle p, y\right\rangle$. Donc en particulier :

$$\forall x, y, \mathbf{K}_M(x|y) \leqslant \ell(\overline{\langle\psi\rangle}\cdot p) \leqslant \ell(\overline{\langle\psi\rangle}) + \ell(p).$$

Le théorème est donc prouvé avec $\hat{M}$ comme machine additivement optimale et $c_{\hat{\psi}} = \ell(\overline{\langle\psi\rangle})$. □

On peut définir ainsi trois classes de complexité : **KPC**, **KPP** et **KPG**. On peut même les comparer, car il n'apparaît pas évident qu'elles soient toutes égales, voire même reliées entre elles.

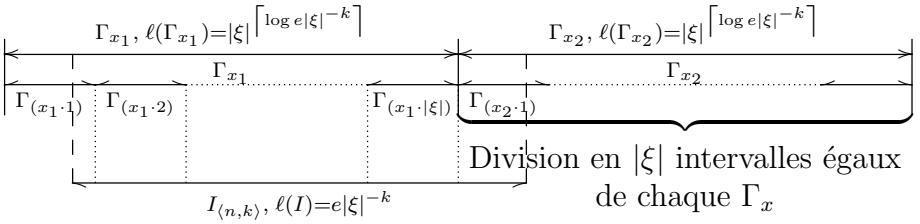
Théorème 15 *La relation suivante est vraie :*

$$\forall x, \quad \mathbf{KPG}(x) = \mathbf{KPP}(x) + \mathcal{O}(1) = \mathbf{KPC}(x) + \mathcal{O}(1)$$

◇ *Preuve.* On montre d'abord que **KPG** est inférieure à **KPP** et **KPC**. On montre ensuite que **KPC** est inférieure à **KPG**, puis que **KPP** est inférieure à **KPG**. La deuxième preuve se fait à l'aide d'un lemme intermédiaire.

$\underline{\mathbf{KPG}(x) \leqslant \mathbf{KPP}(x) + \mathcal{O}(1) \textbf{ et } \mathbf{KPG}(x) \leqslant \mathbf{KPC}(x) + \mathcal{O}(1)}$ Cette étape est évidente. Comme l'ensemble $\mathcal{MPG}$ inclut la classe $\mathcal{MPP}$ et la classe $\mathcal{MPC}$, la machine additivement optimale de ces deux dernières classes donne naissance à une complexité qui est forcément supérieure ou égale (à constante près) à celle de la classe $\mathcal{MPG}$. On a forcément une constante additive qui vient s'ajouter, comme lorsque l'on compare deux machines additivement optimales pour la même classe (comme précisé dans la proposition 0.2).

$\underline{\exists F \in \mathcal{MPC}, \mathbf{K}_F(x) + \mathcal{O}(1) \leqslant c(x)}$ Soit c une fonction qui vérifie la propriété suivante : la somme $\sum |\xi|^{-c(x)}$ est finie, et c est une fonction approximable par au-dessus (l'ensemble $\{\langle x, k\rangle, c(x) \leqslant k\}$ est récursivement énumérable). On veut montrer qu'il existe alors un codage préfixe récursif $\hat{F}$ de $\mathcal{MPC}$ tel que la fonction c soit minorée (à constante additive



Les lettres de ξ sont représentées par les caractères 1, 2,. . . ,$|\xi|$. *On a représenté sous forme de flèches les intervalles associés à tous ces mots.*

FIGURE 3.2 – Inclusion de Γ_x dans I. $F(x_{1,i}) = n$, pour i de 2 à $|\xi|$

près) par la complexité selon cette machine. Soit donc l'énumération de l'ensemble $C = \{\langle x, k\rangle, c(x) \leqslant k\}$. Prouvons que la somme $\sum_{\langle n,k\rangle\in C} |\xi|^{-k}$ est finie. Pour cela, on peut réarranger les termes :

$$\begin{aligned}\sum_{\langle n,k\rangle\in C} |\xi|^{-c(k)} &= \sum_n \sum_{k\geqslant c(n)} |\xi|^{-k} \\ &\leqslant \sum_n \sum_{k\geqslant 0} |\xi|^{-(k+c(n))} \\ &\leqslant (|\xi|/(|\xi|-1)) \sum_n |\xi|^{-c(n)}.\end{aligned}$$

Étant données les conditions initiales sur c, cette somme est donc bien finie. Soit e une constante telle que cette somme soit inférieure à 1. On peut associer chaque élément de C à un sous-intervalle fermé de $[0, 1]$ de longueur $e|\xi|^{-k}$ (la borne supérieure du précédent devenant la borne inférieure du suivant). Chacune des bornes d'intervalle est calculable. Faisons maintenant l'association entre les intervalles fermés de $[0, 1]$ et Ξ, en associant à chaque mot x l'intervalle $\Gamma_x = \left[\sum \mathsf{x}_{\langle i\rangle}|\xi|^{-i}, \sum \mathsf{x}_{\langle i\rangle}|\xi|^{-i} + |\xi|^{-\ell(x)}\right]$. On peut ainsi définir la machine $\hat{F}$ en disant que $F(x) = n$ s'il existe un segment I associé à un couple $\langle n, k\rangle$, tel que $\Gamma_x \subset I$, et que Γ_x soit maximal pour cette propriété. Pour chaque couple $\langle n, k\rangle$, il existe donc x tel que $F(x) = n$, et tel que $\ell(x) \leqslant \left\lceil \log_{|\xi|} e|\xi|^{-k} \right\rceil + 1 \leqslant k + \mathcal{O}(1)$. Donc F vérifie bien $\mathbf{K}_F(x) + \mathcal{O}(1) \leqslant c(x)$. On vérifie aussi aisément que $\hat{F} \in \mathcal{MPC}$, car on ne fait converger x que si Γ_x est maximal, et F est bien p.p.r. (car C est récursivement énumérable).

$\mathbf{KPC}(x) \leqslant \mathbf{KPG}(x) + \mathcal{O}(1)$ On peut conclure le lemme précédent en constatant que $\mathbf{KPG}(x)$ vérifie les conditions posées sur c dans le lemme. La condition d'approximabilité par dessus provient de la remarque qui suit la proposition 0.5. De plus, pour toute machine $\hat{M}$ dans $\mathcal{MPG}$, dénotons par X l'ensemble des mots de Ξ qui sont un plus court codage pour un certain y. La somme $\sum_{y\in\Xi} |\xi|^{-\mathbf{K}_M(y)}$ est en fait inférieure ou égale à $\sum_{x\in X} |\xi|^{-\ell(x)}$. Or l'ensemble X est un code préfixe, puisqu'un élément et une de ses prolongations donnent

le même résultat, ils ne peuvent être un plus court codage tous les deux. Donc, d'après le théorème 1, la somme $\sum_{y\in\Xi} |\xi|^{-\mathbf{K}_M(y)}$ est bien finie. Ceci étant vrai en particulier pour la machine additivement optimale pour la classe $\mathcal{MPG}$, on peut donc bien appliquer le lemme précédent. Donc, en raison de la proposition 0.47, il existe $\hat{F} \in \mathcal{MPC}$ telle que $\mathbf{KPC}(x) \leqslant \mathbf{K}_F(x) + \mathcal{O}(1) \leqslant \mathbf{KPG}(x) + \mathcal{O}(1)$.

$\underline{\mathbf{KPP}(x) \leqslant \mathbf{KPG}(x) + \mathcal{O}(1)}$ On va montrer que pour toute machine $\hat{M} \in \mathcal{MPG}$, il existe une machine $\hat{M'} \in \mathcal{MPP}$ telle que $\mathbf{K}_M = \mathbf{K}_{M'}$ (de plus, la fonction $\langle M\rangle \mapsto \langle M'\rangle$ est récursive). En appliquant ceci à la machine additivement optimale pour la classe $\mathcal{MPG}$, puis en appliquant la proposition 0.47, on obtient le résultat attendu. Soit $\hat{M}$ une machine préfixe, et $\hat{M'}$ la machine qui exécute l'algorithme suivant : simuler la machine $\hat{M}$ en alternance sur l'entrée et tous ses préfixes, en augmentant le nombre de pas de calculs. Le résultat donné par la machine $\hat{M'}$ est alors le premier résultat donné par l'arrêt de la machine. La machine $\hat{M'}$ appartient bien à $\mathcal{MPP}$: elle reste préfixe parce que $\hat{M} \in \mathcal{MPG}$, et si elle converge sur un mot x, elle convergera alors sur tous les mots qui sont une continuation de x. La transformation est bien récursive. Vérifions maintenant que $\mathbf{K}_M = \mathbf{K}_{M'}$. Soit $x \in \Xi$, et p un plus court programme tel que $M(p) = x$. On a alors automatiquement $M(p) = M'(p)$, donc $\mathbf{K}_{M'} \leqslant \mathbf{K}_M$. Par ailleurs, on ne peut pas avoir non plus de chaîne p' plus courte que p telle que $M'(p') = x$, car sinon il en existerait un préfixe tel que $M(p'') = x$. Donc $\mathbf{K}_M = \mathbf{K}_{M'}$, et la preuve est terminée.

□

3.3.5 Puissance de calcul et expressivité

Il est légitime de se demander s'il y a des différences entre les classes de machines évoquées, puisque la complexité de Kolmogorov résultante est la même. Il ne peut être question de comparer directement les fonctions, encore que l'on sache que $\mathcal{MPC}$ et $\mathcal{MPP}$ sont inclus dans $\mathcal{MPG}$. Par exemple, il n'est pas possible de comparer $\mathcal{MPC}$ et $\mathcal{MPP}$, car les deux ensembles sont disjoints. Dans le cas des machines préfixes, il existe une caractéristique plus représentative :

Définition 41 (Frontière) Soit $\hat{M}$ une machine appartenant à $\mathcal{MPG}$, de domaine D. On dit que x appartient à la frontière de M si et seulement s'il n'existe pas de préfixe propre de x dans D. On note $\mathcal{F}(\hat{M})$ l'ensemble des mots qui appartiennent à la frontière de M. On note par $\mathcal{F}_{\hat{\mathcal{C}}}$ l'ensemble des frontières de toutes les machines de la classe $\hat{\mathcal{C}}$. [5]

Si on considère l'ensemble Ξ ordonné par l'ordre préfixe, l'ensemble frontière d'une machine représente l'ensemble des premières occurrences d'un point de convergence de $\hat{M}$, et $\mathcal{F}_{\hat{\mathcal{C}}}$ représente tous les « dessins » (voir figure 3.3.5) que l'on peut construire à l'aide d'une fonction de $\hat{\mathcal{C}}$. Ils ne représentent pas strictement la puissance de calcul au sens de la hiérarchie habituelle, mais ont un rapport avec l'expressivité de la classe.

5. La notion étant indépendante de la machine qui est choisie, on peut parler de la frontière d'une machine plutôt que de la fonction qu'elle calcule.

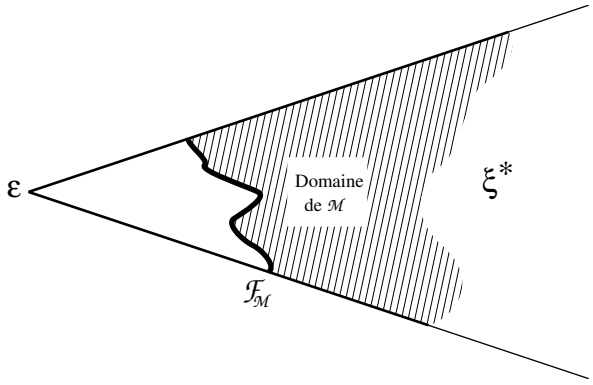


FIGURE 3.3 – Représentation de l'ensemble frontière ($M \in \mathcal{MPP}$)

Il est possible de comparer ces ensembles. Soit la notion d'inclusion récursive suivante. L'ensemble des frontières d'une classe A est récursivement inclus dans l'ensemble des frontières d'une classe B si et seulement si toute machine $\hat{M}$ appartenant à A peut être transformé récursivement en une machine $\hat{M}'$ appartenant à B tel que $\mathcal{F}(\hat{M}) = \mathcal{F}(\hat{M}')$. On le note alors $\mathcal{F}_A \lesssim \mathcal{F}_B$. Si $\mathcal{F}_A \lesssim \mathcal{F}_B$ et $\mathcal{F}_B \lesssim \mathcal{F}_A$, alors $\mathcal{F}_A$ et $\mathcal{F}_B$ sont dit récursivement équivalentes ($\mathcal{F}_A \sim \mathcal{F}_B$). Si deux ensembles ne sont pas récursivement équivalents, ils peuvent être égaux quand même. Par contre, s'ils ne sont pas égaux, ils ne peuvent pas être récursivement équivalents. De même, l'inclusion récursive implique en particulier l'inclusion.

Proposition 0.48 *On a les équivalences suivantes :*
- $\mathcal{F}_{\mathcal{MPG}} \not\subset \mathcal{F}_{\mathcal{MPC}}$;
- $\mathcal{F}_{\mathcal{MPC}} \lesssim \mathcal{F}_{\mathcal{MPG}}$;
- $\mathcal{F}_{\mathcal{MPP}} \sim \mathcal{F}_{\mathcal{MPG}}$.

Arrêtons nous sur ce résultat avant de le prouver. Cela peut apparaître comme étonnant au premier abord, puisque la complexité de Kolmogorov engendrée par chacune de ces classes est identique. Il est expliqué de façon informelle par la raison suivante : il est plus difficile de construire un ensemble « fin » qui capture toutes les finesses d'un dessin (ici, la fonction caractéristique de $\mathbb{K}$) que si l'on a la possibilité de faire la même chose en moins précis.

◇ *Preuve.*

$\underline{\mathcal{F}_{\mathcal{MPP}} \lesssim \mathcal{F}_{\mathcal{MPG}} \textbf{ et } \mathcal{F}_{\mathcal{MPC}} \lesssim \mathcal{F}_{\mathcal{MPG}}}$ Ce sens d'inclusion est évident, car $\mathcal{MPP}$ et $\mathcal{MPC}$ sont inclus dans $\mathcal{MPG}$. Donc toute frontière engendrée par une machine préfixe creuse ou pleine est dans l'ensemble des fonctions préfixes généralisées.

$\underline{\mathcal{F}_{\mathcal{MPG}} \lesssim \mathcal{F}_{\mathcal{MPP}}}$ Soit $\hat{M} \in \mathcal{MPG}$, et $\hat{M}'$ la machine qui exécute l'algorithme suivant : simuler la machine $\hat{M}$ en alternance sur l'entrée et tous ses préfixes, en augmentant le nombre de pas de calcul. Le résultat donné par la machine $\hat{M}'$ est alors le premier résultat donné par l'arrêt de la machine. On note γ cette transformation $\langle M \rangle \mapsto \langle M \rangle'$ (récursive). La

machine $\hat{M'}$ appartient bien à $\mathcal{MPP}$: elle reste préfixe parce que $\hat{M} \in \mathcal{MPG}$, et si elle converge sur un mot x, elle convergera alors sur tous les mots qui sont une continuation de x. La transformation est bien récursive. De plus, la frontière de M' est égale à la frontière de M. Donc, pour toute frontière d'une machine $\hat{M} \in \mathcal{MPG}$, il existe une machine $\hat{M'} = \hat{\phi}_\gamma(\langle M \rangle) \in \mathcal{MPP}$, telle que $\mathcal{F}(M) = \mathcal{F}(M')$.

$\underline{\mathcal{F}_{\mathcal{MPG}} \not\subset \mathcal{F}_{\mathcal{MPC}}}$ Comme l'inclusion n'est pas vérifiée, on peut remarquer qu'en particulier les deux ensembles ne sont pas récursivement équivalents.

Soit $\mathbb{K} = \{n \in \xi^\star, \phi_n(n) \downarrow\}$. $\overline{\mathbb{K}}$ n'est pas énumérable, mais $\mathbb{K}$ l'est. Soit $\hat{M}$ une machine de $\mathcal{MPG}$ répondant à la définition suivante : $\hat{M}$ diverge sur toute entrée qui n'est pas de la forme 1^n01 or 1^n0. Sur une entrée de la forme 1^n01, la machine donne le résultat 1, et sur l'entrée 1^n0 la machine répond 1 si et seulement si $n \in \mathbb{K}$, divergeant sinon. C'est possible puisque $\mathbb{K}$ est énumérable.

L'ensemble frontière de cette machine, qui appartient à $\mathcal{F}_{\mathcal{MPG}}$, est l'ensemble $F = \{1^n0, n \in \mathbb{K}\} \cup \{1^n01, n \notin \mathbb{K}\}$. Supposons qu'il existe une machine $\hat{M'}$ dans $\mathcal{MPC}$ qui ait le même ensemble frontière. Pour une machine préfixe creuse, l'ensemble frontière et le domaine sont confondues. À partir de $\hat{M'}$, on peut construire une machine qui décide de l'appartenance à $\mathbb{K}$: cette nouvelle machine $\hat{P}$ simule en alternance, et sur des durées de plus en plus longues, la machine $\hat{M'}$ sur l'entrée 1^n01 et 1^n0. Selon que l'une ou l'autre est acceptée, ce qui se fait toujours en temps fini, on peut ainsi décider si l'entrée n de $\hat{P}$ est dans $\mathbb{K}$ ou pas. Ceci est contradictoire avec de nombreux résultats connus sur $\mathbb{K}$, et prouve donc que $\hat{M}$ n'a pas d'équivalent non-récursif.

□

3.3.6 Machines double-préfixes

Le modèle des machines doublement préfixes a jusqu'ici suivi toutes les propriétés des machines préfixes, sauf pour la propriété de ***s-m-n* interne**. Toutefois, il n'est pas possible d'utiliser directement la même démonstration à propos de la propriété d'additivité optimale que pour les autres classes de machines. Dans la preuve apparaît en effet un codage dont on ne peut pas être certain qu'il ne va pas faire converger la machine dans des conditions qui garantissent la qualité doublement préfixe. En fait, on peut même prouver le contraire, à l'aide du théorème suivant :

Théorème 16 *Les deux équations suivantes sont vérifiées :*

$$\forall \epsilon > 0, \exists \hat{f} \in \mathcal{MDPC}, \exists c, \forall x \in \Xi, \quad \mathbf{K}_f(x) \leqslant (1+\epsilon)\mathbf{KS}(x) + c \tag{3.2}$$

$$\forall \hat{f} \in \mathcal{MDPC}, \exists \epsilon > 0, \exists c, \forall x \in \Xi, \quad \mathbf{K}_f(x) \geqslant (1+\epsilon)\mathbf{KS}(x) + c. \tag{3.3}$$

◇ *Preuve.*

$\underline{\mathbf{K}_f(x) \leqslant (1+\epsilon)\mathbf{KS}(x) + c}$ Soit ϵ un réel strictement positif, et N le plus petit entier strictement supérieur à $1/\epsilon$. On note $\hat{U}_{\mathbf{KS}}$ la machine additivement optimale choisie pour la

KS-complexité. On considère, pour cette preuve seulement, les notations suivantes :

$x^\star$	Un mot tel que $U_{\mathbf{KS}}(x^\star) = x$ et $\ell(x^\star)$ minimal
$\overline{x}$	$01 \cdot \mathsf{x}_{\langle 0\rangle} \dots \mathsf{x}_{\langle 2N-1\rangle} \cdot 01 \cdot \mathsf{x}_{\langle 2N\rangle} \dots \mathsf{x}_{\langle 4N-1\rangle} \cdot 01 \cdot \mathsf{x}_{\langle 4N\rangle} \dots \mathsf{x}_{\langle \ell(x)-1\rangle} \cdot \underbrace{0 \dots 0}_{\text{complète la longueur à un multiple de } (2N+2)}$
$[\![n]\!]$	$1^{n+1}0^{2N-1-n}$, pour $0 \leqslant n < 2N$

On examine la machine $\hat{f}$ qui diverge si l'entrée n'est pas de la forme suivante, *i.e.* telle qu'il existe v tel que :

$$u = 0^{2N+2} \cdot \underline{\overline{[\![\ell(v) \bmod 2N]\!] \cdot v}} \cdot 0 \cdot 1^{2N+2}.$$

Dans le cas contraire, le résultat du calcul est alors le résultat de l'exécution de $\hat{U}_{\mathbf{KS}}$ sur v. Il est possible de retrouver v car le code employé est uniquement décodable. Vérifions que $\hat{f}$ appartient bien à $\mathcal{MDPC}$. Soit u dans le domaine de f. D'après la définition de $\hat{f}$, il existe v tel que $u = 0^{2N+2} \cdot \underline{\overline{[\![\ell(v)]\!] \cdot v}} \cdot 0 \cdot 1^{2N+2}$. Mais écrivons aussi $u = u' \cdot w \cdot u''$, avec w dans le domaine de $\hat{f}$. Si $u' \neq \varepsilon$, il existe une séquence de $2N+3$ 0 dans u, puisque w est un sous-mot de u et que tout mot du domaine de f commence par $2N+3$ 0. Or, ce ne peut être tout au début puisque $u' \neq \varepsilon$, et qu'il y a forcément un 1 en position $2N+4$. Ce n'est donc pas possible, à cause de la définition de $\overline{x}$, puisqu'on a au plus $2N+1$ 0 consécutifs dans le codage utilisé. Donc $u' = \varepsilon$. De même, à cause de la définition de $\overline{x}$, on ne peut pas avoir plus que $2N+2$ 1 successifs, comme c'est le cas à la fin de chaque mot. Donc $u'' = \varepsilon$, et $\hat{f}$ appartient bien à $\mathcal{MDPC}$. Vérifions l'inégalité (3.2). On a bien $x = f(y)$, où $y = 0^{2N+2} \cdot \underline{\overline{[\![\ell(x^\star) \bmod 2N]\!] \cdot x^\star}} \cdot 0 \cdot 1^{2N+3}$ donne bien x comme résultat. La longueur d'un codage possible pour x par $\hat{f}$ est alors :

$$\begin{aligned}
\ell(y) &= 2N + 2 + \ell\left(\underline{\overline{[\![\ell(x^\star) \bmod 2N]\!] \cdot x^\star}}\right) + 2N + 3 \\
&= 4N + 5 + (2N+2)\left\lceil \frac{\ell\left([\![\ell(x^\star) \bmod 2N]\!]\right) + \ell(x^\star)}{2N} \right\rceil \\
&= 4N + 5 + (2N+2)\left\lceil \frac{2N + \mathbf{KS}(x)}{2N} \right\rceil \\
\ell(y) &\leqslant 6N + 7 + (2N+2)\left(\frac{\mathbf{KS}(x)}{2N} + 1\right) \\
\mathbf{K}_f(x) &\leqslant 8N + 9 + (1+\epsilon)\mathbf{KS}(x).
\end{aligned}$$

Donc en prenant $c = 8N + 9$, l'inégalité est bien vérifiée.

$\underline{\mathbf{K}_f(x) \geqslant (1+\epsilon)\mathbf{KS}(x) + c}$ Nous allons maintenant prouver l'inégalité (3.3). Soit une machine $\hat{f}$ quelconque appartenant à $\mathcal{MDPC}$. On suppose que son domaine n'est pas vide (auquel cas la vérification est immédiate), et on note u l'un des mots du domaine de longueur minimale. Soit $n = \ell(u)$. L'appartenance à $\mathcal{MDPC}$ garantit que quelque soit le mot v

dans le domaine de $\hat{f}$, u n'est pas un sous-mot de v. Soit x un mot tel que $\mathbf{K}_f(x)$ soit fini, et $x^\star$ un mot tel que $\ell(x^\star) = \mathbf{K}_f(x)$ et $f(x^\star) = x$. Prenons maintenant l'écriture de $x^\star$, vu comme un mot de $(\xi^n)^\star$. Pour $\mathbf{K}_f(x)$ suffisamment grand, cette écriture est compressible : en effet, une lettre de ξ^n, à savoir le mot u, n'est jamais utilisé. Notons $\ddot{x}$ la réécriture de $x^\star$ en utilisant tout l'alphabet disponible. On a l'existence d'une constante c, telle que pour tout x, $\ell(\ddot{x}) \leqslant \frac{\log(|\xi|^n-1)}{\log(|\xi|^n)}\ell(x^\star) + c$, où $|\xi|$ est le cardinal de ξ. Posons ϵ vérifiant $1/(1+\epsilon) = \frac{\log(S^n-1)}{\log(|\xi|^n)}$. Donc il existe une machine $\hat{g}$, telle que pour tout x, $\mathbf{K}_g(x) \leqslant (1+\epsilon)\mathbf{K}_f(x) + c'$. Donc, a fortiori, il existe c'' et ϵ, vérifiant l'inégalité (3.3).

□

Corollaire 0.49 *Il n'existe pas de machine additivement optimale pour la classe des machines doublement préfixe.*

◇ *Preuve.* Supposons qu'il existe $\hat{f}$, additivement optimale pour $\mathcal{MDPC}$. Selon le théorème 16, l'équation (3.3) s'applique à $\hat{f}$:

$$\exists \epsilon > 0, \exists c, \forall x \in \Xi, \mathbf{K}_f(x) \geqslant (1+\epsilon)\mathbf{KS}(x) + c. \tag{3.3'}$$

Pour ce même ϵ, il existe une machine $\hat{g}$ qui vérifie (3.2) (appliquée à 2ϵ) :

$$\exists \hat{g} \in \mathcal{MDPC}, \exists c', \forall x \in \Xi, \mathbf{K}_g(x) \leqslant (1+2\epsilon)\mathbf{KS}(x) + c'. \tag{3.2'}$$

En combinant (3.2)' et (3.3)', on obtient que

$$\forall x \in \Xi, \mathbf{K}_g(x) \leqslant \mathbf{K}_f(x) + c' - c + \epsilon\mathbf{KS}(x),$$

ce qui est incompatible avec l'équation fondamentale (1.9). □

Chapitre 4 Modélisation du calcul

De nombreuses définitions de la notion de calcul et d'algorithme ont été données et ont essayé d'en caractériser tous les aspects. Ces définitions font souvent la part belle à la description des caractéristiques finies du calcul (comme la simulation par agrégats dans les machines de Kolmogorov-Uspensky [US93a]). L'un des modèles les plus classiques est la machine de Turing. Ce modèle présente une particularité que l'on retrouve dans d'autres modèles : c'est une modélisation dont l'espace de simulation est infini (même s'il en existe une représentation finie). C'est cet aspect infini de la machine de Turing et des problèmes liés à la simulation que nous abordons ici.

4.1 Étude du modèle de Turing à une variable

4.1.1 Étude générale

Lorsque l'on regarde comment est définie la machine de Turing, de nombreuses imprécisions restent dans les définitions. Si la définition que l'on en donne est complète, les démonstrations principales utilisent des hypothèses — parfois implicites — qui ne sont pas *a priori* justifiées par autre chose que la difficulté à les contourner.

L'une des hypothèses les plus marquantes de ce point de vue est le choix de l'alphabet de la machine. Les modèles les plus classiques, qui calculent les fonctions récursives sur un alphabet ξ, utilisent le plus souvent un alphabet pour la machine considérée qui n'est pas ξ, mais $\xi \cup \{B\}$, où B est une « lettre distinguée » qui sert de caractère blanc.

Le modèle de la machine de Turing est un modèle fini. On le justifie par l'argument suivant : « à tout moment du calcul, il est possible de donner une représentation finie du mot sur lequel on est en train de calculer ». Mais le modèle de la machine de Turing est aussi un modèle infini : les espaces dans lesquels évolue la machine sont des espaces infinis. Pour concilier ces deux visions, il faut regarder d'un peu plus près la nature de ce qui permet de passer du modèle infini au modèle fini. La clé du problème est dans le mot « représentation » : on donne une représentation finie d'un état du calcul, donc c'est un élément qui appartient au domaine du fini. On retrouve la même ambiguïté qu'entre un nombre décimal, comme 2,5 par exemple, qui est aussi un nombre comportant une

infinité de décimales (toutes nulles). La question se complique pour le nombre 2/7. Il est bien « fini », puisqu'on peut l'écrire de façon non ambiguë, en utilisant une quantité finie de caractères. Pourtant, toutes ses décimales ne sont pas nulles. De même, π est aussi un nombre « fini », car on peut en donner une *représentation* finie ; le cas est similaire, bien que plus compliqué, pour des nombres comme la constante d'Euler (qui est $\lim_{n\to\infty} \log n - 1 - 1/2 - \cdots - 1/n$). Cependant, dans le cas qui nous occupe, la représentation du calcul n'est pas une opération complètement banale. Elle peut changer la façon dont on envisage le calcul.

Attardons-nous sur un premier problème : la machine de Turing à deux états. On considère donc une machine de Turing classique, mais dont l'alphabet de codage est $\{0, 1\}$.

Ces machines sont aussi puissantes que toutes les autres : par groupage des cellules du ruban, il est possible d'y encoder tout ce que peut faire un modèle à quatre caractères, ce qui est suffisant pour obtenir la puissance des machines de Turing usuelles. Cette puissance est obtenue par simulation à ratio de temps borné, mais pas en « temps réel » (chaque étape de la nouvelle machine représente deux ou trois pas de l'ancienne machine). [1]

On peut également simuler toutes les fonctions calculables sur un alphabet à une seule lettre (la deuxième servant alors de caractère blanc distingué). Cette puissance est toutefois inférieure à la puissance des machines usuelles, car il y a un changement fondamental dans les données quantitatives de la représentation. La représentation d'un objet parmi un grand nombre se fait en taille linéaire par rapport au nombre d'objets possibles, alors qu'avec au moins deux caractères significatifs, on reste dans le domaine du polynomial.

Nous venons d'utiliser de façon implicite la possibilité de faire un « groupage », sans en justifier sa légitimité. Une proposition nous permet ainsi de tempérer cette observation sur la puissance de ce modèle de calcul. Nous allons regarder comment sur un même ensemble de machines, nous pouvons faire varier la notion de calcul, et de là la puissance de ces machines. On peut intervenir d'abord sur le résultat. Examinons la modélisation suivante : nous définissons la fonction f calculée par une machine M, comme la fonction qui à un mot x associe un mot y si et seulement si pour tout mot infini $\mathbf{w}$, la machine de Turing M pour laquelle l'état initial du ruban est le mot $x \wr \mathbf{w}$, converge vers le mot $\overline{y} \wr \mathbf{w}'$ ($\mathbf{w}'$ est un mot infini quelconque), avec $y \mapsto \overline{y}$ un codage préfixe choisi au préalable. [2] Le résultat du calcul est alors y. La machine peut comporter un nombre fini quelconque de rubans, mais nous nous contenterons de deux rubans pour la simulation.

L'origine de cette conception du calcul est simple : on cherche un modèle de calcul préfixe, et indépendant du bruit qu'il peut y avoir sur le ruban d'entrée. En un sens, toutes les entrées possibles ayant un préfixe commun sont susceptibles de représenter, dans cette modélisation du calcul, une unique entrée — mais pour une machine. Une machine différente peut distinguer plusieurs entrées distinctes dans ce groupe.

Cette modélisation du calcul présente des avantages : notamment, elle utilise des machines bien connues, changeant seulement la notion de convergence. De plus, les fonctions

1. Toutefois, certains auteurs définissent ceci comme étant du « calcul en temps réel », un facteur multiplicatif étant autorisé entre un modèle et sa simulation [Rab63, Ros67].

2. On rappelle que la notation $x \wr \mathbf{w}$ note le mot infini $\mathbf{w}$ dont les premières lettres sont remplacées par le mot fini x. Les règles d'associativité sont $x \wr y \wr \mathbf{z} = x \wr (y \wr \mathbf{z})$.

calculées restent calculées sur des entrées finies, mais ce que l'on rajoute au-delà de ces entrées finies n'a pas d'importance. L'imprécision peut rester sans ambiguïté. Hélas, cette modélisation du calcul n'est pas Turing-équivalente (c'est-à-dire universelle pour le calcul au même sens que la modélisation usuelle) : on peut calculer strictement moins de fonctions, comme le montre la proposition suivante.

Proposition 0.50 *Dans la modélisation du calcul précédemment définie, les fonctions f calculées sont exactement les fonctions calculables préfixes pleines de $\Xi \to \Xi$, sauf celles qui vérifient la propriété suivante (0 est une des lettres de ξ) :*

$$\exists x \notin \mathrm{Dom}\, f, \forall y \neq \varepsilon, x \cdot y \in \mathrm{Dom}\, f \text{ et } f(x \cdot y) = f(x \cdot 0). \tag{4.1}$$

Un corollaire évident nous montre que le choix de la lettre 0 n'est pas important.

Attardons-nous un instant sur la condition (4.1). On a donc des fonctions préfixes non calculables *dans notre modélisation*. En effet, il est impossible de distinguer la longueur d'un mot et de le différencier de ses suffixes. Ainsi, la fonction préfixe pleine qui vaut 1 si la longueur du mot est plus grande qu'une constante, et 0 sinon, n'est calculable (dans notre cadre) par aucune machine, puisqu'il est impossible de « voir » la fin d'un mot.

◇ *Preuve.* Une telle modélisation ne calcule que des fonctions préfixes pleines de $\Xi \to \Xi$. Supposons qu'une entrée donne un résultat convergent sur x ; elle converge sur l'état initial $x \wr \mathbf{w}$ pour tout $\mathbf{w} \in \Omega$. En particulier, elle converge sur l'état initial $x \wr ((x \cdot u) \wr \mathbf{w})$, pour tout $u \in \Xi$ et tout $\mathbf{w} \in \Omega$. Or, $x \wr (x \cdot u) \wr \mathbf{w} = (x \cdot u) \wr \mathbf{w}$. Donc, $x \cdot u$ est aussi une entrée convergente qui donne le même résultat que x. Donc, la fonction calculée appartient à $\mathcal{MPP}$.

De même, les fonctions calculées ne pourront jamais vérifier la propriété (4.1). Cette propriété caractérise le fait que le domaine de convergence doit « s'écrouler ». Expliquons cet abus de langage : si tous les mots ayant un préfixe commun (disons x) ont le même résultat, alors notre modélisation ne peut pas distinguer x comme entrée indépendante, car il y a forcément quelque chose derrière la donnée de x, et que quelque soit la lettre suivante, le résultat est le même. Alors, le domaine de définition « s'écroule », et x doit être un point de convergence. Plus précisément, supposons qu'une telle fonction soit calculée par ce modèle, et soit x un mot vérifiant cette propriété. Sur un ruban initial infini $(x \cdot y) \wr \mathbf{w}$ pour y non vide et $\mathbf{w}$ quelconque, l'itération de la machine de Turing converge vers une unique valeur z quel que soit y. On en déduit que x vérifie la propriété suivante : pour tout $\mathbf{w}$, l'itération de la machine de Turing sur le ruban initialement à $x \wr \mathbf{w}$ s'arrête, avec une unique valeur de convergence. Donc $x \in \mathrm{Dom}\, f$.

Il ne reste plus qu'à montrer que l'on peut calculer toutes les autres fonctions préfixes. Choisissons une machine qui calcule une fonction préfixe sur une machine de Turing habituelle (avec blancs), qui ne vérifie pas (4.1). On transforme ensuite cette machine en utilisant une fonction très similaire au projecteur α décrit pour la classe des machines préfixes pleines. On obtient ainsi une machine calculant la même fonction, et dont le mode opératoire consiste à simuler la machine de départ sur des entrées de longueur croissante.

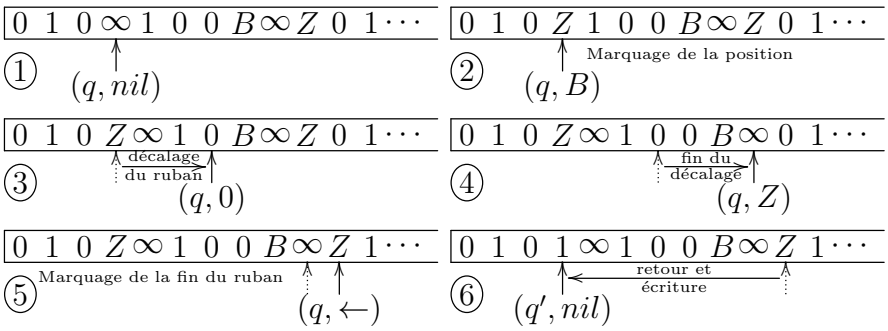


FIGURE 4.1 – Écriture d'un 1 sur un ∞ (représenté en macro-lettres)

La première étape consiste à modifier la machine à simuler de façon à ce qu'elle lise et écrive des groupements de trois caractères binaires. On utilise la table de transformation suivante :

$$0 \to 000 \qquad 1 \to 111 \qquad B \to 101 \qquad \infty \to 010.$$

Il est facile de transformer automatiquement l'automate fini qui contrôle une machine de Turing pour faire cet effet. La transformation du ruban initial, qui ne sera pas codé par des macro-lettres, sera expliquée plus loin. En fait, le ruban initial sera recopié petit à petit sur le ruban de travail.

Le ruban de travail comportera plusieurs plages de travail, composées de 0, de 1 et de B. Aux extrémités de chacune des plages, il y a un caractère ∞ qui représente une infinité de caractères blancs. Pour simuler le ruban de travail, on exécute la procédure suivante : lorsqu'un caractère ∞ est lu et qu'on doive écrire à sa place, ou qu'on veuille se déplacer vers sa droite, un groupement inutilisé (par exemple 100), que l'on peut noter symboliquement Z, est posé à la place du ∞ (figure 4.1). On décale les caractères à partir du ∞ vers la droite, en retenant le caractère suivant dans l'état interne au préalable. L'opération est recommencée jusqu'à ce que le caractère lu soit un symbole Z à nouveau. Ce Z est alors réécrit une macro-lettre plus loin (donc trois lettres plus loin qu'il n'était au départ). La tête de lecture revient alors sur le caractère Z qui avait été posé au départ, et le remplace par la macro-lettre qu'il fallait écrire depuis le départ. Le déroulement continue alors de façon normale.

Ce qui est fait par ces deux opérations consiste à se munir de cinq caractères (les trois de la machine que l'on veut simuler, un caractère ∞ plus un caractère Z supplémentaire), et de représenter par une portion de ruban, un ruban de la machine simulée terminée à droite par uniquement des caractères blancs. En fait, il peut y avoir autre chose au-delà de notre portion de ruban (des informations complémentaires, un autre ruban, etc.) mais si on devait en venir à vouloir écrire dessus (à augmenter l'espace de calcul, en fait), une sous-procédure décale tout le reste du ruban de trois cellules vers la droite (soit la taille d'une macro-lettre) jusqu'au premier Z rencontré. Comme nous le verrons plus loin, l'algorithme garantira que sur le ruban de travail, il y aura toujours un Z à la fin de la partie significative du ruban de travail, et ce sera la seule fois où cette macro-lettre sera utilisée sur ce ruban en dehors de la procédure de décalage du ruban. On utilise le

caractère ∞ pour être certain que le caractère B peut être utilisé exactement de la même façon qu'il l'était dans la machine d'origine M.

L'algorithme utilisé par la machine est le suivant :

(i) On commence par mémoriser les trois premières lettres d'entrée sur le ruban d'entrée, et on les recouvre par une macro-lettre Z (soit 100). On écrit sur le ruban de travail ∞Z, c'est-à-dire le groupement 010100.

(ii) On écrit le premier bit sur le début du ruban de travail sous forme d'une macro-lettre (ce faisant, on décale donc la séquence ∞Z vers la droite, puisqu'on efface le ∞ en faisant cela).

(iii) On revient sur le ruban d'entrée, et on décale le symbole Z d'*une* cellule vers la droite, en mémorisant auparavant le quatrième bit de l'entrée. On a maintenant en mémoire dans les états internes de la tête le deuxième, le troisième et le quatrième bit de l'entrée.

(iv) Par des techniques classiques, on va maintenant simuler la machine initiale sur le mot décrit par ce bout de ruban et tous ses préfixes pendant un temps égal à sa longueur. Pour cela, on peut par exemple dépasser le ∞ et écrire d'abord un compteur de longueur borné à la longueur désirée (en unaire), un ∞, un compteur de temps (en unaire), un ∞, et un bout de ruban qui servira de ruban de travail pour la simulation. On clôt le tout par un Z.

(v) On simule la machine modifiée comme expliqué avant cet algorithme, avec comme configuration initiale les n premiers caractères de l'entrée (selon l'état du compteur de préfixe), et en faisant suivre chaque pas de calcul de l'interruption suivante : mémoriser la position de la tête par une macro-lettre Z, aller incrémenter le compteur de temps.

(vi) Quand le compteur de temps est plein, si la machine n'a pas convergé, on efface le ruban de travail (donc jusqu'au Z, non inclus), et on augmente de 1 la longueur dans le compteur préfixe. On recommence à l'étape *(v)*, en remettant au préalable à zéro le compteur temps.

(vii) Quand le compteur de préfixes est plein, on retourne à l'étape *(ii)*, et on récupère un bit supplémentaire de l'entrée (écriture sur le ruban de travail sur le premier blanc suivi de la séquence ∞Z, puis décalage du « curseur » Z sur le ruban d'entrée).

(viii) Lorsque la simulation donne un calcul convergent à l'étape *(vi)*, on écrit sur le ruban de travail/sortie (selon que c'est un ruban spécialisé ou non) la sortie du modèle simulé dans le codage préfixe choisi (qui peut être par exemple le codage en macro-lettres utilisé dans cet algorithme avec un blanc final). On se met alors dans l'état d'arrêt, et on rend le résultat correspondant à la simulation effectuée (qui a provoqué l'arrêt).

Il faut prouver que cette machine calcule bien la même fonction que la machine de départ. La preuve est très similaire à la preuve de l'existence du projecteur, mais on se place ici dans un cas très simple : on sait que la fonction de départ est préfixe pleine. Si on admet

que les suppositions énoncées dans les petits sous-algorithmes (format du ruban de travail en particulier) seront respectées, ce qui est très facile à constater, on voit que sur chaque préfixe du mot, puis sur des suffixes qui dépendent de la configuration initiale choisie, la machine de départ va être simulée et rendre son résultat. Si c'est un préfixe du mot qui converge le premier, le fait que la fonction soit préfixe pleine garantit que l'on a rien transformé ; si c'est un suffixe, on se repose sur la notion de convergence employée dans le modèle de calcul.

On sait donc que pour l'entrée x de notre nouvelle machine, le résultat du calcul de $x \cdot y$ a été choisi avec une continuation infinie $x \cdot y \wr \mathbf{w}$, avec $y \neq \varepsilon$, et que celui-ci n'est pas le même que celui de x. Deux cas sont possibles : « la valeur de la fonction pour x est différente de celle de $x \cdot y$ » ou « x n'appartient pas au domaine de la fonction, mais $x \cdot y$ si ».

Le premier cas peut vite être éliminé car la fonction d'origine est préfixe pleine. Le cas problématique s'élimine par le fait que la fonction n'a pas la propriété (4.1). Trois sous-cas se présentent : il existe au moins deux continuations ayant un résultat différent ($x \cdot z$ et $x \cdot t$), il existe une infinité de continuations qui ne sont pas dans le domaine de f mais aucune ne donne de résultats différents, ou encore il en existe un nombre fini non nul.

Dans le premier sous-cas, parce que f est préfixe pleine, $x \cdot z \wr \mathbf{w}$ donnera le même résultat pour tout $\mathbf{w}$. Or $x \cdot z \wr \mathbf{w} = x \wr (x \cdot z) \wr \mathbf{w}$. En appliquant le même résultat pour t, on voit que deux mots infinis dans lesquels on plonge x donnent lieu à un résultat du calcul différent, ce qui suffit à dire que x ne converge pas dans notre nouvelle machine (avec notre nouvelle définition de « converger »). Donc la différence de résultat entre x et $x \cdot y$ pour un autre $\mathbf{w}$ n'a pas d'importance.

Dans le deuxième sous-cas, le lemme de Köenig nous dit qu'il existe un mot infini $x \wr \mathbf{w}$ dont tout préfixe fini n'appartient pas au domaine de f. Donc, sur l'entrée infinie $x \wr \mathbf{w}$, le calcul de la nouvelle machine n'est pas convergent, donc x n'est pas dans son domaine.

Dans le troisième sous-cas, nous prenons la continuation de longueur maximale. $x \cdot z$ n'appartiendra pas au domaine de f, et donc pas au domaine de la nouvelle machine (en effet, seuls les deux premiers sous-cas peuvent se présenter pour ce mot). Donc en particulier, il existe un mot infini $(x \cdot z) \wr \mathbf{w}$ qui ou bien est tel que la nouvelle machine converge vers un résultat différent d'un autre continuation, ou bien est tel qu'elle ne converge pas du tout (l'un des deux sous-cas s'appliquant). Or, ces mots infinis commencent en particulier par le mot x. La raison qui fait que $x \cdot z$ ne soit pas dans le domaine de la machine est donc aussi une raison pour que x n'y appartienne pas.

Donc, si on introduit sur certains points une convergence qui n'aurait pas dû exister avec certains $\mathbf{w}$, il en existe toujours au moins un qui propose un résultat différent ou une divergence. Donc, la fonction calculée (dans notre nouvelle modélisation du calcul) est bien la même qu'avec la machine de départ. □

Il est possible de modifier cet énoncé pour utiliser un seul ruban bi-infini. On obtient alors un modèle avec des conditions minimales (dans un certain sens), où les machines ont quand même une capacité de calcul plus grande qu'un automate à pile.

Corollaire 0.51 *Soit $\xi = \{0, 1\}$. Il existe une variante de la machine de Turing à un seul ruban bi-infini qui calcule exactement les fonctions préfixes pleines de $\Xi \to \Xi$, sauf celles qui vérifient la propriété* (4.1).

◇ *Preuve.* On utilise la preuve précédente, sauf que l'on appelle « ruban de travail » la partie gauche du ruban, lue à l'envers. On marque la séparation des deux rubans par un symbole Z que l'on écrit à gauche de la position de départ de la tête de lecture dès le début du calcul. Le reste de la preuve est inchangé. □

Cette proposition nous prouve qu'un des premiers éléments constitutifs du calcul, en dehors du mécanisme d'itération du calcul, est de définir dans une modélisation du calcul la façon de déterminer le résultat du calcul.

Avant de revenir aux conditions qui permettent de définir le calcul, regardons ce qui se passe dans le cas de la machine de Turing avec deux caractères, en fonction de la façon dont on interprète ces caractères. Il est possible de considérer d'autres versions de la machine de Turing à deux caractères, où l'on n'utilise pas de codage préalable de l'entrée, et où l'on n'a aucun caractère supplémentaire (pas de blanc). Il s'agit en fait des machines de Turing qui calculent des fonctions sur un alphabet à un seul caractère.

Par exemple, on peut prendre le modèle conventionnel (avec A la lettre signifiante, le mot d'entrée étant alors A^n) et un caractère blanc B. On peut considérer que le blanc est pour la machine un caractère comme les autres. On dissocie alors complètement l'alphabet de travail de la machine et l'alphabet de codage de l'entrée (puisqu'on utilise un caractère ordinaire), ce qui diminue la puissance du calcul. Dans ce cas, l'ensemble des fonctions calculables par ce modèle est l'ensemble des fonctions qui sont de la forme $x \mapsto f(\ell(x))$, avec f une fonction récursive quelconque [3] de $\mathbb{N} \to \mathbb{N}$. On retombe de toute façon dans le cadre de la figure 4.3, qui est exposé plus loin. En effet, on ne peut pas distinguer deux entrées autrement que par leur longueur, d'où la notation $f \circ \ell$ donnée ci-après pour les fonctions qui n'agissent que sur la longueur de l'entrée.

Proposition 0.52 *Dans le modèle de calcul à un caractère avec écriture de blanc, l'ensemble des fonctions calculées est exactement l'ensemble des fonctions $f \circ \ell$, avec f p.p.r.*

◇ *Preuve.* Puisque les entrées ne sont distinguables que par leur longueur, toute fonction calculée est forcément de la forme $f \circ \ell$, avec f une fonction récursive (on peut identifier les fonctions agissant sur x et sur $\ell(x)$). La notation $f \circ \ell$ est une notation adoptée pour les fonctions récursives sur un alphabet unaire.

Il reste à prouver que toute fonction de la forme précédente peut être effectivement calculée. Pour cela, on recourt à un même système de macro-lettres que précédemment, et on commence par transformer l'écriture de l'entrée. On simule alors une machine à deux caractères et un blanc de la façon la plus directe qui soit.

La transformation de l'entrée du unaire en une écriture en macro-lettres est possible de la façon suivante : la tête de lecture se déplace jusqu'au premier B rencontré. Elle

3. Il reste toutefois le problème du codage de la sortie, point qui est évoqué plus loin dans ce chapitre.

inscrit alors $ABAB$ à cet emplacement, puis se décale sur le dernier A de l'entrée (quatre mouvements à gauche).

On commence l'itération suivante : tant que le caractère courant est un A (et que l'on n'est pas en début de ruban dans le cas d'un semi-ruban), on le fait suivre d'un B. On écrit alors des A vers la droite jusqu'à ce qu'on ait lu deux B au total (le ruban est alors de la forme $A \dots ABA \dots AB$). On remplace ce B par le groupement AB.

À la fin de chaque itération, on revient au caractère précédent le premier B du ruban. On a alors décalé le premier groupement AB d'une cellule vers la gauche, et le deuxième d'une cellule vers la droite. Ainsi, $A^n \star ABA^{2k}AB$ devient $A^{n-1} \star ABA^{2(k+1)}AB$ (l'étoile marque la position de la tête de lecture).

Après toutes les itérations, on a réécrit l'entrée sous formes de macro-lettres. En utilisant BB et BA comme autres macro-lettres, on a maintenant une machine de Turing à trois caractères et un blanc (AB). □

Il existe aussi d'autres modèles intéressants, comme le modèle à « blancs connexes ». Dans cet autre modèle, et bien que ce ne soit pas une propriété décidable, on exige que le caractère B soit spécial, et qu'une machine, au cours de son exécution ait le droit de l'écrire uniquement si on laisse connexe les ensembles de caractères blancs et non-blancs. La puissance est alors diminuée à celle d'un automate à une pile unaire, comme le montre la proposition 0.53.

Dans ce modèle qui, en spécialisant le blanc, s'enlève un « vrai caractère », la puissance de calcul diminue encore plus :

Proposition 0.53 *Dans le modèle de calcul à un caractère avec écriture de blancs connexes uniquement, la puissance de calcul est celle d'un automate fini (transducteur).*

◇ *Preuve.* On utilise un résultat de [Har78], qui prouve l'énoncé suivant : l'ensemble des langages algébriques à un caractère est l'ensemble des langages rationnels à un caractère.

On utilise en plus une simulation de cette catégorie particulière de machines de Turing par un automate à une pile unaire (dont la capacité de reconnaissance de langage est celle d'un automate fini).

Étant donnée une machine à simuler utilisant q états internes, on fait une analyse *a priori* de son comportement sur un ruban comportant une infinité de A. Par finitude, le comportement de la machine est ultimement périodique. On peut pour chaque état de départ définir un cycle dans le comportement de la tête. Pour chaque état appartenant à un tel cycle, on peut définir un déplacement de la tête (borné par q) et une « amplitude » correspondant en valeur absolue aux cases visitées avant de revenir à l'état initial. On définit donc une table correspondant à ces deux valeurs en fonction de l'état de départ. Il est à noter que au plus $q - 1$ itérations suffisent dans un parcours où on ne lit que des A, à trouver un état appartenant à un cycle. On calcule aussi le p.p.c.m. de tous les déplacements (pris en valeur absolue).

L'état de notre automate à pile sera le produit cartésien de plusieurs caractéristiques (voir la figure 4.1.1). Une partie du ruban sera codée directement dans les états internes,

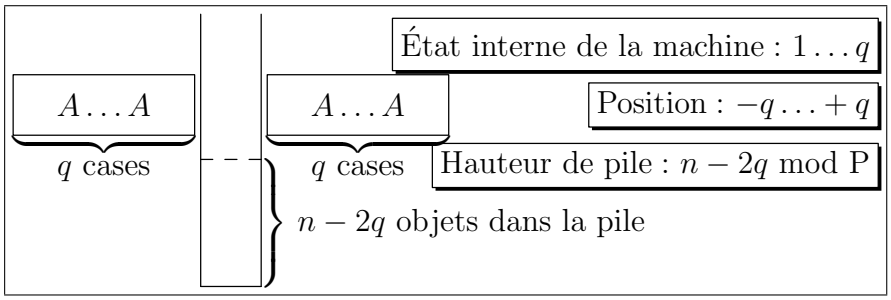


FIGURE 4.2 – Automate à pile unaire simulant une machine de Turing unaire

avec un état spécial au milieu. On code ainsi les q premiers éléments du ruban, et les q derniers (le reste étant codé par la hauteur de la pile). On ajoute un compteur de hauteur de pile, qui est comptée modulo le p.p.c.m. précédemment trouvé.

On ajoute un compteur de position de la tête de lecture/écriture de la machine de Turing. On ne dépile jamais le fond de pile.

La simulation se fait de la manière suivante : l'automate à pile commence par dépiler son entrée (ou la lire, selon les détails du modèle adopté pour l'automate à pile). Lorsque l'on dépasse les $2q$ états mémorisés dans les états internes, on empile le reste (l'entrée est forcément de la forme A^n). On initialise le compteur modulaire de hauteur de pile, et les bouts de piles simulés directement dans les états internes. On initialise aussi à 1 le compteur de position (on peut noter $1 \dots q$ les q premières positions du ruban, 0 le reste, jusqu'aux q dernières, notées $-q \dots -1$).

On simule la machine de Turing de façon normale, tant que l'on reste sur les q cases du ruban situées à une extrémité ou une autre, entièrement par l'automate fini. Si on augmente la taille de la partie non-blanche du ruban, on empile un objet sur la pile ; si on la diminue, on dépile de même. Et si la tête de lecture doit traverser la partie du ruban qui est représentée par la pile, elle est forcément dans un cycle homogène et on peut précalculer directement sur quelle case et dans quel état la tête de lecture arrivera de l'autre côté. Ainsi, on simule la machine par un automate à une pile. Un peu plus de détails : si la tête de lecture essaye de s'aventurer à gauche du ruban, on va changer les codage des q premiers états du ruban en les décalant, et empiler un A (et on maintient à jour le compteur de hauteur de pile). De même si on essaye de s'aventurer à droite du ruban déjà inscrit. On sait que si on dépasse le nombre de q blancs à gauche ou à droite (et que l'on aurait voulu empiler un blanc dans notre pile), on est dans un comportement cyclique — on refuse d'accepter le mot. Si on se déplace à l'intérieur du ruban et que dans les états mémorisés il y a des blancs (et que la pile n'est pas vide), on décale la partie mémorisée dans le sens adéquat et on dépile un élément.

Quand la pile devient vide, bien sûr, on remplit tous les états internes en trop par des blancs, et on se comporte différemment (jusqu'à ce que le ruban dépasse la longueur $2q$). Sinon, les déplacements sont gérés par une simulation plus directe (ainsi que l'écriture d'un blanc ou d'un A sur l'un des deux bords), sauf dans le cas où on atteint la position q ou $-q$. On sait alors qu'on est dans un état cyclique, et que l'on va se déplacer ainsi

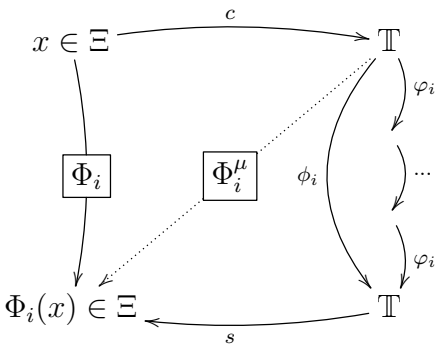


FIGURE 4.3 – Calcul avec codage de l'entrée

jusqu'à l'autre partie du ruban simulée par la pile. En utilisant la donnée de la hauteur de la pile modulo le p.p.c.m. et le déplacement mesuré au départ, on peut savoir sur quelle case du ruban on va arriver dans le même état que l'état de départ au-delà de la partie du ruban qui est simulée par la pile. On peut même utiliser l'amplitude pour économiser une partie du calcul. □

4.1.2 Utilisation d'un codage pour l'entrée

À la lumière des constatations précédentes, il devient évident qu'en essayant de réduire le modèle de calcul aux composantes les plus élémentaires possibles, comme une taille d'alphabet et un nombre de rubans minimaux, si l'on veut garder l'universalité du calcul (au sens de la thèse de Church, c'est-à-dire l'ensemble des fonctions calculables au sens habituel), on ne peut pas se contenter d'utiliser de façon immédiate le même alphabet pour coder l'entrée et les calculs. L'une des solutions consiste à ajouter des caractères, dont la sémantique est différente (ils ne peuvent pas être utilisés dans l'entrée) ; une autre solution consiste à utiliser un codage pour l'entrée. C'est fréquemment le cas lorsque l'on fait des simulations d'autres modélisations du calcul. Si l'on s'autorise à faire un codage pour l'entrée, on arrive à une structure de modélisation plus compliquée.

La propriété la plus importante que l'on doit récupérer par rapport à la modélisation de la proposition 0.50 est de faire agir la machine sur une seule entrée infinie et de pouvoir en déduire les résultats du calcul. L'entrée n'est pas donnée sous sa forme brute, mais via un codage récursif.

Toutefois, on ne sait caractériser qu'un codage récursif fini (la récursivité appliquée aux ensembles infinis n'est pas utilisable en temps fini). Or, les configurations initiales ne peuvent être qu'infinies dans les variantes du modèle de la machine de Turing. En autorisant un codage quelconque, la question apparaît naturellement, de savoir si la puissance de nos machines est modifiée.

Le schéma général que nous proposons pour le calcul est porté sur la figure 4.3. Le calcul,

sous cette forme, est décomposé en trois étapes. On procède d'abord à un *plongement*, une transformation d'un mot fini en un mot infini (ou une autre structure, voir plus loin) qui sert d'entrée à un *processus de calcul*. Le processus de calcul peut déboucher sur deux résultats : ou bien il diverge, auquel cas le mot d'entrée ne fait pas partie du domaine de la fonction calculée, ou bien il converge. Dans ce dernier cas, on applique alors une fonction d'extraction au résultat du processus de calcul. La fonction d'extraction transforme donc une des structures utilisées par le processus de calcul en un mot fini.

La fonction ϕ s'obtient alors à travers la définition d'une transition $\mathbb{T} \xrightarrow{\varphi} \mathbb{T}$ (ou *fonction de transition*). Le calcul est convergent par le fait qu'un point fixe de la fonction de transition φ est atteint en un nombre fini d'itérations. L'objet mathématique associé à φ, qui à un mot de $\mathbb{T}$ associe le point fixe (s'il existe) de φ, est la fonction globale calculée par la machine ayant cette fonction de transition. Une machine est définie à un niveau encore au-delà : on donne une description (un mot, un entier), et par le biais d'une fonction de codage (ou d'interprétation) on en déduit une fonction de transition. Par exemple, imaginons une machine de Turing à 28 états, et la même machine avec 29 états, dont le 29$^{\text{e}}$ reste inusité. C'est la même fonction qui est calculée, même dans la façon dont le processus de calcul opère, et quelque soit l'entrée ; mais ce n'est pas la même machine.

Définition 42 (Modélisation du calcul) Étant donné un alphabet σ, une modélisation du calcul est la donnée d'un *espace de calcul* $\mathbb{T}$, d'une fonction d'interprétation (voir ci-après), d'une fonction de $\Xi \to \mathbb{T}$ appelée *plongement* et d'une fonction de $\mathbb{T} \to \Xi$ appelée *extraction*. La fonction d'interprétation associe à un mot ou un entier i (le code d'une machine) une fonction de transition $\mathbb{T} \xrightarrow{\varphi_i} \mathbb{T}$ qui gouverne le fonctionnement de la *machine* $\hat{\phi}_i$. La *fonction globale* de la machine est ϕ_i, obtenue comme le point fixe de φ_i. La *fonction calculée* par la machine $\hat{\phi}_i$ est $s \circ \phi_i \circ c$. La fonction d'interprétation n'est pas une injection, seul le code (ou numéro) spécifie uniquement la machine.
Cette modélisation du calcul est dite bonne si la liste des programmes $\left\{\hat{\phi}_i\right\}_{i\in\mathbb{N}}$ forme un système acceptable de programmation.

Dans l'espace $\mathbb{T}$, on doit regrouper tout ce qui est nécessaire pour faire le calcul, et qui peut être dépendant de l'entrée. Ainsi, dans le cas Turing, on doit y mettre l'état initial de la machine de Turing si celui-ci dépend de l'entrée.

L'expérience des modélisations précédentes nous apprend que ce schéma correspond à des cas déjà vus. Ainsi, le modèle de calcul utilisé à la proposition 0.52 rentre tout à fait dans ce modèle, et est universel pour le calcul. Le plongement adopté est alors $c(x) = 1^x 0^\omega$, et l'extraction est la fonction qui à $\mathbf{y}$ associe un codage préfixe quelconque (mais choisi au préalable), comme par exemple l'écriture en macro-lettres.

Comme indiqué sur la figure 4.3, il peut être intéressant de comparer à la fois la puissance de calcul d'un modèle à travers les fonctions calculées Φ et à travers l'expressivité de son processus de calcul Φ^μ. C'est ce qui a été fait plus haut pour le cas des modèles de calcul à deux caractères : modulo le codage, les machines à deux caractères sont universelles pour le calcul — on peut dire *Turing-universelles* — mais si l'on n'a pas le bon codage, les processus de calcul Turing avec deux caractères donnent lieu à une modélisation du

calcul qui n'est pas bonne. Il est possible de le voir de façon duale : les processus de calculs peuvent être vus comme universels pour le calcul puisqu'*il existe* un codage qui leur permet de calculer toutes les fonctions.

Réfléchissons aux conditions que l'on veut garder sur les fonctions d'extraction et de plongement. On peut fabriquer des modèles de calcul où toute la complexité du calcul peut être mise dans l'une ou l'autre de ces fonctions. Par exemple, le cas où l'on cache dans la fonction de codage le résultat de chacune des fonctions calculables sur l'entrée considérée. On code l'entrée par l'écriture de tous ces résultats sur un ruban. Une machine peut ainsi dépasser la puissance de calcul Turing, en allant lire ce ruban. Voire, la seule action de la machine pourrait être de positionner la tête de lecture au bon endroit. On peut aussi cacher plus subtilement le résultat dans la sortie : le codage d'une entrée consiste alors à répéter à l'infini cette entrée, et l'action d'une machine est d'écrire son numéro. On code alors dans l'extraction le calcul, en disant que l'interprétation d'un tel ruban est le résultat de l'application du numéro marqué sur le ruban à la valeur périodique située à la fin du ruban. Le temps de calcul est nul ou polynomial pour tout calcul, ce qui n'est pas satisfaisant.

On introduit maintenant une notion qui limite nos modélisations du calcul mais les assure d'avoir une justification plus forte. On suit en cela les principes des machines de Kolmogorov-Uspensky, en restreignant l'étude aux machines dont les structures de données sont arrangeables de façon à être représentées par un mot infini (la généralisation est possible, mais le but de notre étude est en particulier les variantes des machines de Turing) :

Définition 43 Une modélisation du calcul est une *modélisation par remplacement initial* si l'espace de calcul $\mathbb{T}$ est en bijection avec les mots infinis sur un alphabet ς et si la fonction de transition φ est la prolongation (définie comme suit) d'une fonction $\tilde{\varphi}$ de $\varsigma^\star$ dans $\varsigma^\star$ vérifiant :

$$\forall \mathbf{u} \in \varsigma^{\mathbb{N}}, \exists! v \in \varsigma^\star, \begin{cases} \tilde{\varphi}(v) \text{ est définie.} \\ \mathbf{u} \in \lhd v \end{cases} \tag{4.2}$$

On a alors $\varphi(\mathbf{u}) = \tilde{\varphi}(v) \wr \mathbf{u}$.

Expliquons cette définition : on regarde ici des modélisations qui peuvent être définies uniquement par des substitutions sur un préfixe ; il s'agit d'un cas très général. Elle ne définit pas uniquement de *bonnes* modélisations du calcul. Elle ne recouvre pas non plus toutes les bonnes modélisations du calcul, en particulier les systèmes de Post. En revanche, elle représente bien les systèmes de calcul usuellement employés, et en particulier les variantes des machines de Turing. L'alphabet ς est le produit cartésien de plusieurs sources dans le cas Turing : celui-ci est le produit cartésien des alphabets de chaque ruban, et les autres caractéristiques des machines seront notées en unaire (position des têtes de lecture/écriture, et état interne).

On peut remarquer que l'espace $\mathbb{T} = \varsigma^{\mathbb{N}}$ obtenu est strictement plus grand que ce qui est nécessaire pour notre machine. Par exemple, si on code en unaire la position de la tête, un certain nombre de configurations ne sont pas valides. On peut faire une analogie avec une machine physique sur laquelle les pièces ne seraient pas montées dans le bon sens.

La fonction de transition φ nécessaire pour décrire le fonctionnement d'une machine

de Turing est la prolongation d'une fonction de « substitution initiale » : pour faire une itération de calcul, il n'est pas nécessaire de regarder plus loin que la tête de lecture la plus éloignée pour calculer quel est l'état suivant. Grâce à la notation en unaire de la position de chaque tête, il est possible de repérer exactement cette position, et d'assurer le respect de (4.2). La fonction $\tilde{\varphi}$ est alors définie par un schéma général, et des substitutions faites au niveau de chaque tête de lecture/écriture. Il existe des préfixes de taille quelconque, mais générés en fait par translation à partir d'un nombre fini de mots (c'est ce qui se passe dans le cas Turing, mais pas dans les systèmes de Post).

Dans le cas d'une machine simple, qui se contente de remplacer tous les 1 d'un mot par un 0 jusqu'à rencontrer un B, les préfixes sont donnés de la façon suivante — on code par $a^n b^\omega$ la position de la tête sur la $(n+1)$-ième case :

$$\begin{bmatrix}0\\a\end{bmatrix}^n \begin{bmatrix}B\\b\end{bmatrix} \to \begin{bmatrix}0\\a\end{bmatrix}^n \begin{bmatrix}B\\b\end{bmatrix} \qquad \begin{bmatrix}0\\a\end{bmatrix}^n \begin{bmatrix}0\\b\end{bmatrix} \to \begin{bmatrix}0\\a\end{bmatrix}^{n+1} \qquad \begin{bmatrix}0\\a\end{bmatrix}^n \begin{bmatrix}1\\b\end{bmatrix} \to \begin{bmatrix}0\\a\end{bmatrix}^{n+1}$$

Le premier remplacement traduit la règle d'arrêt de la machine (si la tête est sur un blanc, s'arrêter). Les deux autres traduisent le décalage vers la droite de la tête, et l'écriture éventuelle d'un 0. Cet exemple a été choisi par commodité, car il n'y a pas d'état interne à représenter.

Toutes nos modélisations du calcul utilisant des mots finis sur un alphabet fini, agissant par dérivations successives, peuvent bien évidemment être considérées comme étant à remplacement initial (on ajoute un caractère blanc que l'on répète à l'infini), comme les systèmes de Post et les systèmes RAM.

L'important dans une modélisation du calcul à remplacement initial, c'est le déterminisme de l'application de la transition. Pour tout calcul convergent, on peut ainsi définir une zone examinée, et une zone modifiée. À chaque itération, un unique préfixe d'un mot infini de $\mathbb{T}$ est candidat à substitution par $\tilde{\varphi}$ (par l'application de (4.2)). Comme la convergence se fait en un nombre d'itérations finies, on peut effectivement borner la taille de la zone examinée (qui est la longueur maximale du v préfixe pour chaque itération), et la taille de la zone réellement substituée (inférieure à la première valeur).

Il est possible que certaines machines convergent dans le sens que nous avons donné, et ne convergent pas au sens habituel. C'est le cas où la configuration atteinte est un point fixe lui-même, sans que l'état considéré soit un état d'arrêt. Ce n'est pas important, car c'est une propriété décidable, et une réécriture de la table de transition de la machine par un automate fini, avec l'ajout d'un état spécial, permet de changer cet unique comportement (en ajoutant par exemple un état puits, et en l'associant de façon indifférente à un décalage à gauche d'une tête de lecture).

Revenons sur les limitations dont nous avons déjà parlé page 106. S'il ne faut pas que plongement et extraction cachent toute la puissance de calcul du modèle, il ne faut pas non plus qu'ils la diminuent de façon rédhibitoire. Ainsi, un plongement ne doit pas impliquer que le modèle de calcul, quelque soit le choix de la machine, ne puisse plus distinguer deux entrées. On peut ainsi donner cette définition d'un modèle de calcul qui garde au moins cette propriété :

Définition 44 Une modélisation du calcul munie d'un plongement c sépare les données en entrée si et seulement si quelque soit $u \in \Xi$, il existe une machine τ_u vérifiant que $\tau_u(c(u))$ existe et pour tout $v \neq u$, $\tau_u(c(v))$ est indéfini.

Cette distinction nous semble naturelle. Le modèle de calcul, si l'on veut qu'il soit universel, doit pouvoir distinguer chaque entrée individuellement. Remarquons tout de suite que dans les modélisations classiques, il existe plusieurs familles de machines qui peuvent jouer ce rôle.

Quant à la fonction d'extraction, il faut formaliser ce qui a été explicité plus haut sur la possibilité de cacher dans cette extraction une partie de la puissance du calcul. Le moins que l'on puisse demander (même si ce n'est pas suffisant pour n'obtenir que de bonnes modélisations du calcul) est que cette fonction d'extraction puisse répondre systématiquement, lorsque la condition d'arrêt est vérifiée, à la question « Quelle est le mot représenté sur le ruban de sortie ? ». Si le plus simple est de demander à ce que la fonction d'extraction soit totale, ceci exclut des codages de sortie pourtant simples, comme le codage en unaire, dans certains cas où le codage en entrée porte à l'infini le caractère de l'entrée.

Pour formaliser cette dépendance entre le codage initial et le décodage final, on se contentera de supposer vraie la condition suivante :

Définition 45 Une modélisation du calcul produit des sorties valides si pour tout processus de calcul ϕ sa fonction d'extraction est totale sur les mots qui sont dans l'image de $\phi \circ c$.

4.1.3 L'entrée vue comme un oracle

La notion de codage utilisée pour l'entrée est problématique. Certains codages peuvent faire apparaître des notions d'entrée qui sont inhabituelles, et qui font appel à des données non-connexes sur le ruban. On peut imaginer par exemple, de coder un entier sur les cases dont le numéro est premier, ou alors dont le numéro est celui d'une machine de Turing qui s'arrête sur l'entrée vide. Le plongement à l'infini empêche d'utiliser les notions habituelles de récursivité, puisqu'il n'est pas possible en toute généralité de capturer un mot infini par des processus qui doivent converger en temps fini. Il est désirable de pouvoir en quelque sorte « lire l'entrée » dans le ruban infini, et savoir en isoler une partie qui définit cette entrée. Même si chaque entrée est unique, il faut savoir s'il existe une façon pour le processus de calcul de décider quelle est l'entrée qui lui est donnée.

Il existe une solution pour les modèles de calcul agissant par remplacement initial qui séparent les données en entrée, c'est-à-dire pour lesquels il existe un filtre permettant de choisir précisément une entrée, et de rejeter toutes les autres : il est possible en fait d'ignorer une partie infinie de l'entrée de façon déterministe. C'est le reste, cette partie finie, que nous allons ensuite pouvoir catégoriser.

Proposition 0.54 *Si une modélisation du calcul à remplacement initial sépare les données en entrée, alors le plongement s'écrit sous la forme* $\Gamma \odot \Omega$*, où* Γ *est un code préfixe sur* ς *et* Ω *est une fonction de* Ξ *dans* $\varsigma^{\mathbb{N}}$*.*

Il peut être intéressant de remarquer que ceci équivaut à dire que le plongement s'écrit sous la forme $\Gamma \wr \Omega'$. Toutefois, si Ω est uniquement déterminée par la donnée de Γ, Ω' reste partiellement indéterminée (puisque les premières lettres sont « écrasées » par Γ). D'une façon ou d'une autre, l'entrée est immergée via un code préfixe dans le reste de la configuration initiale correspondante.

◇ *Preuve.* Soit u un mot de Ξ. On utilise l'existence de τ_u (voir définition 44). Le résultat d'un calcul dépend d'un nombre fini de cellules au cours de la dérivation. On va noter m_u le rang le plus élevé des cellules examinées au cours du calcul de τ_u sur $c(u)$. Soit $\Gamma(u)$ les m_u premières cellules de $c(u)$. De par la nature déterministe de la méthode de calcul, il est évident que le calcul reste identique si les m_u premières cellules de $c(v)$ sont les mêmes que celles de $c(u)$. Ainsi, il existe dans $\varsigma^{\mathbb{N}}$ un cylindre $\lhd w$ tel que le calcul s'arrête dans le même temps sur toute entrée. On note $\Gamma(u) = w$. Supposons qu'il existe v tel que $\Gamma(v)$ soit une continuation de $\Gamma(u)$. $\Gamma(v)$ appartiendrait alors au cylindre $\lhd w$, et donc il est impossible que $\tau_u(v)$ diverge. □

On a réussi à séparer deux données fondamentales dans le plongement, et on associe ainsi un code préfixe à tout plongement. On peut donc essayer maintenant de caractériser un « plongement récursif ». En particulier, on regarde si ce code est récursif ou ne l'est pas. Imaginons par exemple le plongement suivant : $c(x) = (0 \cdot \overline{x}) \odot B^\omega$ si $x \in \mathbb{K}$, et $c(x) = (1 \cdot \overline{x}) \odot B^\omega$ sinon. Ce plongement transforme le modèle habituel des machines de Turing en modèle qui sait résoudre le problème de la halte, et perturbe donc l'étude du modèle de calcul en lui-même. C'est l'étude du processus de calcul qui est importante, dans ses différents paramètres. On s'intéresse donc par la suite uniquement au cas où ce codage préfixe est récursif. [4] On peut donc poser cette définition modulo le choix d'une famille de séparation :

Définition 46 Pour toute modélisation du calcul munie d'un plongement c à remplacement initial séparant les données en entrée, on définit Γ comme étant le code préfixe défini par la famille $\{\tau_u\}_{u\in\Xi}$ qui intervient dans la proposition 0.54. On définit Ω tel que $c = \Gamma \odot \Omega$.

Il est à noter que cette définition n'est pas unique, car il peut exister plusieurs familles de fonctions $\{\tau_u\}_{u\in\Xi}$. On ne considérera que le cas où il existe une famille telle que Γ soit un code préfixe récursif (c'est-à-dire récursif d'inverse récursif).

Ainsi, on peut modifier le schéma précédemment construit figure 4.3, pour obtenir un schéma plus évolué, comme sur la figure 4.4.

4.1.4 L'oracle prolongeant les entrées

On peut maintenant se poser la question de savoir à quoi peut servir la partie Ω du plongement. Dans la littérature classique, Ω prend en général une valeur limitée à B^ω ou 0^ω. Mais imaginons que l'on ait le même Ω pour toutes les entrées. La machine va pouvoir

4. La décision d'exclure les codages non récursifs n'est pas définitive : selon le choix des machines, il est possible qu'un codage préfixe non récursif mène quand même à une bonne modélisation du calcul. Dans le cas précédent, si tous les processus de calcul ignorent le premier caractère, on n'ajoute rien.

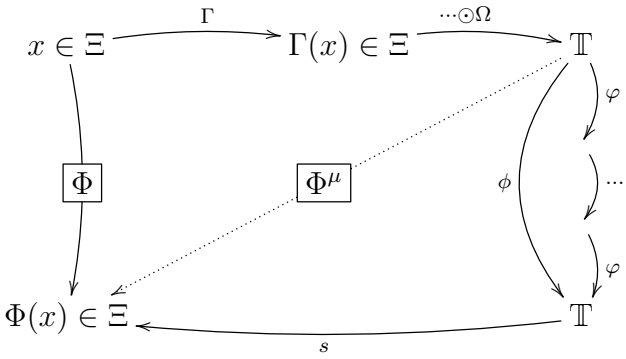


FIGURE 4.4 – Décomposition du plongement

utiliser les informations de Ω comme une aide à la décision. Ω représente un *pseudo-oracle*, qui altère la capacité de calcul de la machine par un phénomène extérieur.

Rappelons la définition classique d'une machine à oracles (telle que l'on peut la lire par exemple dans [HU79]). L'oracle est donné par un langage sur un alphabet σ. La machine possède de plus trois états distingués q_Q, q_O et q_N. La transition de q_Q est étiquetée de façon spéciale : on lit le contenu d'un ruban spécial [5], et une transition spéciale se produit vers q_O ou q_N selon que le mot écrit dessus appartienne ou non à un langage pré-determiné. Le mot écrit est une question, et la transition la réponse (booléenne).

Nous nous posons donc la question, dans cette partie, de savoir si cette modélisation des oracles par une utilisation de la partie infinie du ruban est consistante avec la notion traditionnelle d'oracle (et dans quelles conditions), et nous regardons aussi finalement les valeurs que l'on peut donner au pseudo-oracle en fonction de l'entrée.

Pour représenter ces pseudo-oracles, on va désormais noter sous la forme $\mathbb{1}_A$ la représentation sous forme d'un mot infini d'un ensemble : étant donné une partition de ς en deux ensembles, et on note le fait que le $n-$ième mot de Ξ appartienne ou non au langage A en choisissant la n-ième lettre de $\mathbb{1}_A$ dans le second ensemble au lieu du premier (c'est l'écriture de A sous forme de fonction caractéristique). On utilisera le mot pseudo-oracle pour désigner l'usage d'un ensemble dans notre modélisation du calcul comme source d'information, par opposition au modèle classique des machines relativisées où on parlera alors d'oracle.

Dans un premier temps, nous allons démontrer qu'en donnant le pseudo-oracle avec le plongement $c(x) = \Gamma(x) \wr \mathbb{1}_A$, il existe des oracles qui ne peuvent pas être simulés par notre modèle dans le cas particulier où ς n'a que deux caractères. Par contre, pour peu que ς ait au moins quatre caractères, et qu'il existe une « redondance ascendante » (ce terme sera défini exactement plus loin) de l'oracle A, alors on peut construire un modèle de calcul

5. Ou encore, dans certaines modélisations, la partie droite du ruban de travail à partir de la position actuelle de la tête de lecture.

correct avec le même plongement, qui simule les machines de Turing à oracle relativisées avec A.

On va montrer ensuite que dans le cas général où l'on considère un modèle de calcul possédant des propriétés raisonnables, et quelle que soit la dépendance entre $\Omega(x)$ et l'entrée x, alors on peut faire une réduction-Turing de n'importe quel $\Omega(x)$ à n'importe quel $\Omega(y)$, ce qui donne (entre autres) un positionnement de ces pseudo-oracles dans la hiérarchie arithmétique.

La vision la plus simple de cette simulation est « écrivons notre ensemble, sous forme de fonction caractéristique sur le ruban, mettons l'entrée par dessus » et de voir si on est bien équivalent à une machine de Turing relativisée par l'ensemble en question. Montrons donc qu'il n'est pas possible de simuler une machine de Turing avec un oracle quelconque simplement en le donnant sur la partie infinie du ruban. L'idée de la preuve est de construire un ensemble dont des parties irrécupérables vont être effacées par l'écriture de l'entrée.

Théorème 17 *Soient un ensemble A, et le plongement $c(x) = \Gamma(x) \wr \mathbb{1}_A$. Alors il existe A tel qu'un langage Turing-équivalent à A n'est reconnu par aucune machine de Turing avec le plongement c.*

◇ *Preuve.* On va supposer pour cette preuve que le résultat rendu par la machine de Turing n'a qu'une importance booléenne, c'est-à-dire qu'une machine de Turing accepte un mot, le rejette, ou diverge. Cette simplification ne change rien à la notion de réduction Turing. De même, on va reconnaître non pas le langage A, mais le langage 0^A défini ainsi : pour chaque entier n, on met dans le langage 0^A le plus court mot de la forme 0^m qui vérifie que $\ell(\Gamma(0^m))$ soit de longueur au moins n. Les deux langages sont bien Turing-équivalents (Γ et Γ^{-1} sont supposées récursives), et donc on prouve que si 0^A ne peut pas être reconnu par nos pseudo-oracles, alors elle est bien moins puissante que la relativisation usuelle.

On va construire A par portions finies croissantes. Pour cela, prenons les machines de Turing une par une. On suppose que l'on en est à la i-ième machine et que l'on a déjà déterminé un ensemble A_i fini (c'est-à-dire que pour un nombre fini d'entiers j, on sait si le j-ième mot est dans A_i ou pas). Cet ensemble A_i a la propriété qu'aucune machine ayant un numéro strictement plus petit que i ne reconnaît un langage 0^A pour A contenant A_i. Soit n le plus petit entier non déterminé dans A_i, et soit m le plus petit entier tel que $\Gamma(m)$ soit au moins de longueur n. Regardons tous les calculs possibles de la i-ième machine de Turing sur toutes les extensions possibles de A_i sur l'entrée m. Notons bien que le plongement choisi effaçe la n-ième lettre de $\mathbb{1}_A$, qui n'a pas encore été déterminée[6], et que quelque soit la valeur d'appartenance de n à A, les calculs seront les mêmes.

Si tous ces calculs divergent, alors de toute façon la machine de Turing i ne peut pas reconnaître l'appartenance de 0^m à 0^A, et on pose une valeur arbitraire pour cette appartenance pour compléter A_i en A_{i+1} ($n \in A_{i+1}$, par exemple).

Si au moins un calcul est convergent, on en choisit un et on va le « contrarier » : si 0^n a été accepté, on pose $n \notin A$ et vice-versa. On regarde alors l'ensemble de toutes les

6. Il est possible que le codage préfixe Γ soit contractant pour certaines entrées particulières. C'est pourquoi on utilise un entier plus grand.

cases parcourues lors du calcul qui a été choisi, et on leur donne la valeur qu'elles avaient dans A_{i+1}. Ce nombre de cases étant fini (nous sommes dans l'hypothèse de calcul par substitution initiale), A_{i+1} est bien déterminé en un nombre fini d'entiers. Tout langage qui contiendra ce A_{i+1} sera au moins incapable de reconnaître un mot avec une machine de Turing de numéro inférieur ou égal à i, ce qui conclut la preuve par récurrence : tout langage contenant $\cup_{i\in\mathbb{N}} A_i$ a au moins un langage Turing-équivalent qui n'est pas reconnaissable par une machine de Turing de notre modélisation. □

On peut même étendre le résultat, en regardant non plus la reconnaissance, mais l'énumérabilité :

Corollaire 0.55 *Il existe A et B Turing-équivalents tels que B n'est pas énumérable avec le pseudo-oracle A.*

◇ *Preuve.* On considère B comme étant l'ensemble précédemment défini 0^A. La preuve est identique au théorème précédent, sauf qu'on regarde juste la notion de convergence/divergence (pas de refus ou d'acceptation). Il faut transformer les réactions précédentes pour tenir compte de cela : si tous les calculs divergent, on va choisir systématiquement l'appartenance de n à A_{i+1}, et si au moins un calcul converge, on va par contre la refuser. □

Après ces considérations, il est donc normal de se demander si notre modèle, avec un plongement adéquat, peut quand même, dans certains cas, être assimilé à un modèle relativisé usuel.

Dans un premier temps, nous allons considérer le même plongement que précédemment, qui a un défaut intrinsèque : une partie du pseudo-oracle est effacé *ab initio* par l'entrée, quelque soit le langage choisi. On va donc ajouter la condition de redondance suivante :

Définition 47 Un langage est *effectivement redondant* si et seulement s'il existe une fonction récursive $\mathcal{S}kip$ telle que $\mathcal{S}kip(n) \in A \Leftrightarrow n \in A$ et que $\mathcal{S}kip(n) > n$ pour tout n.

Cette condition n'est pas triviale ; il existe évidemment des langages qui ne la vérifient pas, et des langages vérifient cette propriété. On prend par exemple $\mathbb{K}$, et pour $\mathcal{S}kip$ l'opération qui consiste à rajouter des instructions sans effet à la fin du code de la machine n ; cette fonction est naturellement récursive, croissante (pour peu qu'on l'itère suffisamment), et la convergence d'une machine n'étant pas changé par cette transformation syntaxique, l'appartenance de n à $\mathbb{K}$ est la même que celle de $\mathcal{S}kip(n)$.

Remarque sur le ruban auxiliaire Par un jeu de simulations, il est équivalent de considérer une machine à quatre caractères ou une machine avec un ruban auxiliaire, chacun des rubans ayant deux caractères. Les quatre caractères peuvent être vus comme le produit cartésien de deux fois deux caractères. Le ruban auxiliaire étant supposé initialement inutilisé, on peut le distendre (n'utiliser que les caractères pairs pour faire le calcul), et utiliser les caractères impairs pour marquer l'emplacement de la « tête de lecture » simulée sur le ruban principal. Lorsqu'il y a besoin de la déplacer ou de l'utiliser, on utilise alors la case impaire pour noter la position actuelle de la tête de lecture sur le ruban auxiliaire, on se déplace jusqu'à trouver le (seul autre) marqueur qui est la tête sur le ruban principal,

on l'utilise, la déplace éventuellement, et on revient à la position préalablement marquée. Le temps supplémentaire n'est pas important pour notre propos.

On garde donc le plongement $c(x) = \Gamma(x) \wr \mathbb{1}_A$. On prouve alors la proposition suivante :

Proposition 0.56 *Notre modélisation du calcul, munie du plongement $c(x) = \Gamma(x) \wr \mathbb{1}_A$ est équivalent au modèle des machines de Turing à oracles avec l'oracle A, si l'ensemble A est effectivement redondant, et si au moins une des deux conditions suivante est remplie :*

(i) On dispose d'un ruban de travail auxiliaire ;

(ii) $\mathcal{S}kip$ est constructible en unaire dans un espace égal à sa valeur.

◇ *Preuve.* Il y a deux questions auxquelles on doit répondre pour cette preuve : démontrer qu'il est possible de simuler correctement la « consultation d'oracle » à l'aide du pseudo-oracle, et montrer que l'on a gardé l'existence de la composition (le cas de l'universalité se traitant de façon similaire). Nous proposons donc une construction systématique et récursive, correcte étant donné la fonction $\mathcal{S}kip$ pour remplacer l'oracle par le psudo-oracle.

Dans un premier temps, supposons que l'on ait un ruban auxiliaire. L'action d'une de nos machines est de simuler la machine ayant le même code sur le ruban auxiliaire. Au début de la simulation, la quantité $\Gamma(x)$ est recopiée du ruban d'entrée sur le ruban auxiliaire, et le ruban d'entrée n'est jamais effacé. Lorsqu'une question est posé à l'oracle, on lit le contenu de la question (en utilisant uniquement le ruban auxiliaire). On va itérer la fonction $\mathcal{S}kip$ suffisamment de fois pour dépasser la longueur de l'entrée. On peut alors aller lire sur le ruban d'entrée la valeur d'appartenance du mot de la question à l'ensemble A (codé comme un pseudo-oracle par le mot infini $\mathbb{1}_A$ sur le ruban d'entrée), et passer dans l'état q_O ou q_N selon cette valeur. Ainsi, on simule correctement le questionnement de l'oracle A par nos machines.

On peut procéder de la même façon si la fonction $\mathcal{S}kip$ est constructible en unaire dans son propre espace. Lors de la simulation de la machine, on borne la zone de travail de façon auto-délimitée. Lorsque la question est posée, on commence à construire $\mathcal{S}kip$ à gauche de la zone de travail, en décalant celle-ci peu à peu. Lorsqu'on décale la zone de travail, on reporte de l'autre côté les valeurs du ruban (non précédemment touchées par le calcul, donc intactes). Une fois que l'on a calculé $\mathcal{S}kip$ (en itérant éventuellement suffisamment de fois pour avoir décalé la zone de travail d'au moins sa propre longueur), on a donc pu mémoriser la valeur du pseudo-oracle à cet endroit, et on peut en déduire le résultat de la question à l'oracle. On ajoute alors la partie du ruban qui a servi à calculer $\mathcal{S}kip$ en unaire à la zone de travail.

L'universalité ne pose pas plus de problèmes que la composition. Regardons comment on peut faire la fonction de composition : on cherche un numéro de machine k, récursif en i et j qui calcule exactement $\phi_i \circ \phi_j$. La machine ϕ_k va d'abord simuler de façon tout à fait directe la machine ϕ_j sur l'entrée qui lui est donnée. Si le résultat est divergent, il n'y a rien à faire de plus. Il faut ensuite executer ϕ_i en prenant en compte tout le calcul déjà précédemment effectué, et non pas considérer que l'on commence sur une entrée avec un oracle intacte. Ainsi, lorsqu'une question est posée à l'oracle, l'emplacement recherché ne

doit pas être au-delà de la longueur du calcul actuel, mais bien de la longueur du calcul total. Comme on a pas touché au ruban d'entrée au-delà de cette limite, les données y sont toujours disponibles. On peut en quelque sorte reconstituer la portion d'oracle qui aurait été donnée en plus, puisqu'elle est forcément présente, de manière calculable, plus loin sur le ruban d'entrée (qui n'a pas été effacé). □

Bien évidemment, si cette preuve nous satisfait sur l'adéquation partielle de notre modélisation, elle n'est pas satisfaisante sur d'autres points, car elle ne donne pas des conditions minimales, ni valables sur tous les oracles. Elle ne donne pas non plus d'éléments pour juger *a priori*, étant donné un plongement qui fait figure d'oracle, s'il est possible de construire un modèle de calcul correct, équivalent aux machines de Turing relativisées.

La problématique peut toutefois être étendue à d'autres types de plongement. En particulier, le cas où le plongement n'efface pas le pseudo-oracle semble plus abordable, si l'on suppose que $c(x) = \Gamma(x) \odot \mathbb{1}_A$. Si l'on dispose d'un ruban de travail auxiliaire, il est en tout cas clair que l'on peut utiliser les mêmes techniques que précédemment pour ne pas détruire le pseudo-oracle et simuler les machines de Turing à oracles avec un oracle quelconque. Par contre, le problème reste ouvert dans le cas où l'on a pas de ruban auxiliaire.

Il est possible de regarder ce qui se passe si on suppose que l'oracle est différent pour chaque entrée. Étendons d'abord notre définition de système acceptable de programmation de façon naturelle aux machines à oracles. Il suffit par rapport à la définition 4 donnée 23, d'enlever la condition « la machine universelle est Turing-calculable ». Ceci modifie en retour la définition donnée plus haut d'une bonne modélisation du calcul, en enlevant une contrainte similaire, ce qui donne une bonne modélisation du calcul relativisée.

Supposons qu'il est possible d'avoir un ruban de travail sans effacer le pseudo-oracle. On prouve alors le théorème suivant :

Théorème 18 *Pour toute bonne modélisation du calcul relativisée, pour tout x et tout y, $\Omega(x)$ est Turing-réductible à $\Omega(y)$ (ils sont donc Turing-équivalents).*

◇ *Preuve.* Pour cette preuve, il faut donc pouvoir calculer de façon récursive $\Omega(x)$ à partir de $\Omega(y)$. Il existe une famille récursive de machines de Turing à oracles $\{X_z\}_{z\in\mathbb{N}}$ qui sur l'entrée x donne le z-ième bit de $\Omega(x)$ (de par le théorème *s-m-n*, et parce que $\Omega(x) \leqslant_T \Omega(x)$). On passe du calcul de la fonction caractéristique de $\Omega(x)$ par une seule machine au calcul par une famille récursive de machines, dans le modèle habituel des machines de Turing à oracles. Cette famille se transcrit en une famille récursive de machines de Turing de notre modélisation du calcul où Ω sert d'oracle.

Donc, en réappliquant le théorème *s-m-n*, mais dans notre modélisation dépendant de s et de c, on peut transformer cette machine en une famille de machines qui procèdent de façon identique, mais à partir de l'entrée y voulue, et non plus à partir de l'entrée x : si on appelle G la machine qui écrit x sur le ruban, et F_z la machine qui donne le z-ième bit de $\Omega(x)$ sur l'entrée x, $F_z \circ G$ est une machine qui sur n'importe quelle entrée calcule le z-ième bit de $\Omega(x)$. En particulier, chacune de ces machines se transforme récursivement en une machine à oracles Y_z qui avec l'oracle $\Omega(y)$ donne le z-ième bit de $\Omega(x)$. En utilisant une dernière fois le théorème *s-m-n*, mais dans le modèle de calcul ordinaire des machines

de Turing à oracle, on en déduit qu'il existe une unique machine de Turing, avec l'oracle $\Omega(y)$, qui calcule la fonction caractéristique de $\Omega(x)$. □

Lorsque l'on définit une machine de Turing comme utilisant un oracle, on peut en même temps classer le langage comme faisant partie d'un ensemble de langages, d'indécidabilité croissante (Σ_0 étant les langages décidables, Σ_1 contenant la solution au problème de la halte, etc.). On définit de la même façon une série croissante de langages complémentaires (Π_0 et suivants). On peut alors définir $\Delta_n = \Sigma_{n+1} \cap \Pi_{n+1}$. En particulier, les ensembles contenus dans Δ_0 sont les ensembles récursifs. Ces ensembles forment un treillis pour l'ordre induit par l'inclusion. On peut trouver plus de détails sur cette hiérarchie, appelée la hiérarchie arithmétique dans [Rog67].

Corollaire 0.57 *Pour toute bonne modélisation du calcul relativisée, s'il existe k et y tel que $\Omega(y) \in \Delta_k$, alors pour tout x, $\Omega(x) \in \Delta_k$.*

◇ *Preuve.* Suivant le théorème de Post, si un pseudo-oracle $\Omega(y)$ est dans un certain niveau de la hiérarchie arithmétique Δ_k, alors $\Omega(y) \leqslant_T \emptyset^{(k)}$ (en utilisant les notations de [Rog67]). Tous les oracles $\Omega(x)$ sont donc aussi Turing-réductible à $\emptyset^{(k)}$. La réciproque du théorème de Post étant aussi vraie, on a donc $\forall x, \Omega(x) \in \Delta_k$. □

En fait, cette proposition statue fortement sur la nature du plongement : pour avoir un système de calcul cohérent, qui soit capable de simuler un système acceptable de programmation entre autres, on doit donc avoir forcément un plongement qui est un code préfixe, suivi d'un langage éventuellement indexé par l'entrée, mais toujours dans le même niveau de la hiérarchie arithmétique. Car si on a donné ici une forme particulière à Ω en le codant de façon binaire, il est possible d'utiliser le même alphabet ς utilisé pour Γ (à condition qu'il reste un ruban vierge). Si l'on n'a pas de ruban vierge, il reste encore le problème de savoir si un pseudo-oracle est suffisamment redondant (et de façon récursive pour que l'on puisse retrouver les résultats du calcul).

Une autre remarque : si $\Omega(\varepsilon)$ est un oracle récursif, alors tous les oracles sont récursifs. On n'ajoute alors aucune puissance de calcul. Toutefois, il faut remarquer que ceci n'est pas suffisant pour garantir la cohérence, et donc l'existence, de cette modélisation du calcul. Imaginons par exemple la modélisation où l'on fait suivre x d'un blanc B et de 1^ω si $x \in \mathbb{K}$, et 0^ω sinon. Tous les oracles sont bien récursifs, mais ce système n'est pas garanti de vérifier le théorème *s-m-n* (on peut voir aisément qu'il existe alors une machine H telle que $s \circ H \circ c$ résout le problème de la halte, en lisant le premier caractère après un blanc ; par l'usage de *s-m-n*, on montre qu'une autre machine résout le problème de la halte pour une permutation de toutes les lettres de x, ce qui impliquerait que si l'on peut savoir si une permutation de x appartient à $\mathbb{K}$, alors on saurait si $x \in \mathbb{K}$ ou pas de façon récursive, ce qui n'est pas vrai en général).

Par la suite, on supposera que le plongement c ne change pas la puissance du modèle de calcul, et que l'on a une bonne modélisation du calcul.

4.1.5 Étude détaillée de la complexité après plongement

Il est finalement normal de considérer que toute entrée doit être écrite de façon préfixe, quelque soit le modèle de calcul qui sous-tend le calcul que l'on va employer. Lorsque l'on décrit un objet, il apparaît alors naturel de faire apparaître cette difficulté dans la définition (qui mène d'ailleurs à un certain nombre de termes logarithmiques lorsque l'on utilise la **KS**-complexité, qui disparaissent dès lors que l'on utilise la **KP**-complexité).

La définition originelle de la complexité s'appuie sur la formule (1.8). On peut la modifier pour essayer de refléter cette intuition :

$$ {}^{c}_{s}\mathbf{K}_{\phi}(x) = \min\{\ell(\Gamma(p)), s \circ \phi \circ c(p) = x\}. \tag{4.3}$$

Il est logique alors de voir si le théorème de Solomonoff-Kolmogorov est valable pour cette définition de la complexité, d'abord en faisant varier les machines ϕ, et ensuite en parcourant tous les codages préfixes Γ possibles.

Si c est tel que la puissance de calcul du modèle est la puissance Turing, alors le théorème de Solomonoff-Kolmogorov est valable à c fixé s'il existe un couplage (non bijectif) qui permet de concaténer deux informations, en ne grandissant que linéairement en fonction de l'un des deux.

Définition 48 (Additivité optimale pour une modélisation) La propriété **de l'additivité optimale** est vérifiée pour une bonne modélisation du calcul avec un plongement c et une extraction s, si et seulement s'il existe une fonction M calculable par la modélisation vérifiant :

$$\forall \Phi, \exists c_{\Phi}, \forall x \in \Xi, \quad {}^{c}_{s}\mathbf{K}_{\phi}(x) \leqslant {}^{c}_{s}\mathbf{K}_{m}(x) + c_{\Phi}.$$

Proposition 0.58 *Étant donnée une modélisation du calcul, s'il existe une fonction κ de $\Xi^2 \to \Xi$ telle que :*

$$\ell(\Gamma(\kappa(u,v))) \leqslant \ell(\Gamma(u)) + c_v, \tag{4.4}$$

alors il existe une machine additivement optimale pour cette modélisation du calcul.

◇ *Preuve.* De la même façon que précédemment, on pose la fonction suivante :

$$M(z) = \begin{cases} s \circ \phi_b \circ c(a) = \Phi_b(a) & \text{si } z = \kappa(a,b) \\ \perp & \text{sinon} \end{cases}$$

La preuve est fondamentalement identique à celle du théorème 3. La machine m est capable de simuler n'importe quelle autre machine, en commençant par décoder l'entrée pour récupérer deux arguments, puis en simulant la machine correspondant au deuxième argument sur l'entrée étant le premier argument.

Étant donnés Φ_a et p tels que $s \circ \phi_a \circ c(p) = x$, avec p de longueur minimale, on a donc $s \circ m \circ c(\kappa(p,a)) = x$. D'après l'équation (4.4), on peut dire que $\ell(\Gamma(\kappa(p,a))) \leqslant \ell(\Gamma(p)) + c_a$. Comme ${}^{c}_{s}\mathbf{K}_{m}(x) \leqslant \ell(\Gamma(\kappa(p,a)))$ et ${}^{c}_{s}\mathbf{K}_{\phi_a}(x) = \ell(\Gamma(p))$, on obtient le résultat voulu ${}^{c}_{s}\mathbf{K}_{m}(x) \leqslant {}^{c}_{s}\mathbf{K}_{\phi_a}(x) + c_a$. □

Ainsi, on peut définir toute une famille de complexités pour les codages préfixes vérifiant des propriétés de conservation de longueur. C'est le cas en particulier pour des codages fréquemment utilisés, comme par exemple $\Gamma(x) = 1^{\ell}(x)0x$ (avec le couplage $\kappa(u,v) = \Gamma(v)u$), ainsi que d'autres codes préfixes (avec le même couplage), obtenus par usage récursif de ce codage dans la formule $\Gamma_{n+1}(x) = \Gamma_n(\ell(x))x$. On note ces complexités ${}^{c}_{s}\mathbf{K}$.

Il n'est pas possible d'étudier transversalement tous les codes préfixes pour lesquels cette complexité relative à une modélisation du calcul est définie. On va le voir dans la proposition suivante. Ce problème est lié en effet à l'impossibilité d'énumérer effectivement les codes préfixes récursifs.[7] Chacune de ces modélisations se limite en effet à un unique code préfixe, et en tant que telle, reste plus grande que la **KP**-complexité, comme nous l'avons définie dans les chapitres précédents. Il existe un codage récursif additivement optimal, mais celui-ci n'est pas récursif (c'est le codage induit par la **KP**-complexité). S'il en existait un qui soit récursif, il serait au plus à distance bornée de la fonction $x \mapsto \mathbf{KP}(x)$, ce qui n'est pas possible (voir le théorème 0.17).

Proposition 0.59 *Soit U une fonction additivement optimale pour la classe des fonctions préfixes ($\mathbf{K}_U = \mathbf{KP}$). Soit $\Gamma^\star$ le code préfixe (non récursif) correspondant à l'association $x \mapsto x^\star$, où $U(x^\star) = x$ et $x^\star$ de longueur $\mathbf{KP}(x)$. Soit $c^\star$ le plongement dérivé avec un oracle constant quelconque. Alors ${}^{c^\star}_{s}\mathbf{K} = \mathbf{KP}$, et pour toute modélisation du calcul normale pour laquelle la propriété de Solomonoff-Kolmogorov est vérifiée, on a ${}^{c}_{s}\mathbf{K} \geqslant {}^{c^\star}_{s}\mathbf{K} + \mathcal{O}(1)$.*

◇ *Preuve.* La preuve se fait en interprétant la complexité comme étant calculée de façon normale, mais en composant avec l'inverse d'un code préfixe Γ.

Tout code préfixe est une injection ; donc a un inverse défini, qui est aussi p.p.r. (il suffit d'essayer un nombre croissant d'entrées durant des temps croissants jusqu'à trouver l'élément dont on cherche à calculer Γ^{-1}). $\Gamma^\star$ a aussi un inverse p.p.r. (il s'agit de la fonction U).

Soit p une entrée minimale pour une machine additivement optimale pour un codage Γ et pour l'égalité (4.3). On sait que ${}^{c}_{s}\mathbf{K}(x) = \ell(\Gamma(p))$ et que $s \circ \phi \circ c(p) = x$. Soit $\widetilde{\Phi} = \Phi \circ \Gamma^{-1}$. $\widetilde{\Phi}$ est une fonction p.p.r., et qui sur l'entrée $\Gamma(p)$ donne x. Donc $\mathbf{KP}(x) \leqslant \ell(\Gamma(p))$, et $\mathbf{KP}(x) \leqslant {}^{c}_{s}\mathbf{K}(x)$.

Dans le sens inverse, une entrée minimale pour décrire x avec le code $\Gamma^\star$ est x, qui est de longueur une fois codée $\mathbf{KP}(x)$. Donc l'égalité ${}^{c^\star}_{s}\mathbf{K} = \mathbf{KP}$ est vraie, et on en déduit qu'il existe bien un codage préfixe optimal pour notre mesure de la complexité, mais celui-ci est non récursif. □

On a donc une vision duale de la complexité préfixe. Ou bien on utilise uniquement des machines qui sont préfixes, ou bien on utilise une mesure de la complexité qui tient compte de l'alphabet de travail et du codage initial de l'entrée. Ainsi, utiliser un caractère supplémentaire pour le codage fait bien diminuer la complexité, mais il existe alors une

7. Si on pouvait les énumérer effectivement sous forme d'une suite f_n, on trouverait un paradoxe en définissant $g(n) = \overline{n}f_n(n)$ qui est f_k pour un certain k, et en cherchant $g(k)$. Cette impossibilité est liée au fait que **KP** n'est pas calculable.

façon plus astucieuse d'utiliser ce caractère blanc. Il en existe d'ailleurs autant que l'on veut dès lors que l'on se restreint à savoir associer de façon calculable l'entrée à sa représentation sur la machine. On ne peut trouver des représentations optimales qu'à travers un certain modèle — une représentation plus condensée étant toujours possible.

Or, c'est finalement cette représentation en machine qui est importante : on ne veut pas savoir quelle est la représentation qui possède des propriétés d'être préfixe, mais quelle est celle qui occupe le moins d'espace physique lors du procédé de calcul.

4.2 Dépendance de la deuxième variable

4.2.1 Complexité de Kolmogorov relative à un couplage

Il a été fait très tôt le choix d'utiliser des modèles de calcul où l'on considérait le contexte (le $|y$) indépendamment, comme un deuxième argument. Mais les modèles les plus naturels sont ceux qui ne présentent qu'un seul argument, où le codage a déjà été effectué. Il est possible de donner des comparaisons entre ces deux modèles distincts. On distinguera ces deux énumérations par l'arité des machines énumérées. On notera en général ϕ une machine d'arité 2, et ψ une machine d'arité 1.

Considérons donc un instant la définition suivante : on choisit deux systèmes acceptables de programmation distincts, l'un étant défini de $\Xi^2 \to \Xi$, et l'autre étant simplement défini de $\Xi \to \Xi$. Considérons maintenant un codage κ de $\Xi^2 \to \Xi$, qui est utilisé de façon universelle pour transformer les machines de la seconde énumération en machines d'arité 2.

On va poser les conditions suivantes sur κ : c'est un codage récursif, dont la réciproque est aussi p.p.r. On appelle une telle fonction un couplage récursif.

Définition 49 On définit la complexité (conditionnelle) de Kolmogorov de x sachant y selon la machine ψ d'arité 1 par la formule suivante :

$$\mathbf{K}^{\kappa}_{\psi}(x|y) = \min\{\ell(p), \psi(\kappa(p, y)) = x\}. \tag{4.5}$$

On peut d'abord en déduire un premier corollaire : sur ces deux modèles, les complexités de Kolmogorov d'arité 1 et d'arité 2 sont équivalentes. En effet, pour tout ϕ d'arité 2, la composée $\phi \circ \kappa^{-1}$ aura la même complexité. Donc en particulier pour une machine additivement optimale, on obtient que $\mathbf{KS} \geqslant \mathbf{KS}^{\kappa}$. Comme de même, la composée $\psi \circ \kappa$ est une machine d'arité 2 qui est donc dans l'énumération correspondante, $\mathbf{KS}^{\kappa} \geqslant \mathbf{KS}$.

Il y a donc équivalence si on considère l'ensemble de toutes les machines. Une question naturelle est de savoir ce qu'il en est de chacune des autres catégories, comme les machines préfixes, par exemple.

En effet, pour être qualifiée de préfixe, une machine d'arité 2 doit vérifier une propriété de préfixité sur le premier argument (p). Si κ est telle que l'ajout d'un suffixe à une entrée modifie non pas p, mais y, la composée $\phi \circ \kappa^{-1}$ n'a pas de raison particulière d'être préfixe, même si ϕ est préfixe.

Il est possible de poser des restrictions supplémentaires sur κ, ce qui pourra résoudre un premier problème, à savoir l'existence d'une possibilité de coder des machines d'arité 2 en des machines d'arité 1 qui gardent consistantes la théorie de Kolmogorov.

Définition 50 (Couplage préfixant) Étant donné un couplage récursif de $\Xi^2 \to \Xi$, on dit qu'il est préfixant pour le premier argument s'il vérifie la propriété suivante :

$$\kappa(x \cdot u, y) = \kappa(x, y) \cdot u.$$

À cette condition, et si l'on prouve qu'un tel couplage existe, alors il est possible d'établir une équivalence entre complexité préfixe quelque soit l'arité du modèle de calcul sous-jacent.

Proposition 0.60 *Il existe des couplages récursifs préfixant pour le premier argument. En conséquence, il y a équivalence entre les modèles d'arité* 1 *et les modèles d'arité* 2 *en général.*

◇ *Preuve.* Le couplage récursif défini par $\kappa(x, y) = \ell(y) \cdot y \cdot x$ est préfixant pour le premier argument.

Pour toute machine préfixe ψ d'arité 1, $\psi \circ \kappa$ est une machine préfixe d'arité 2 de la même classe. Par exemple, pour $\psi \in \mathcal{MPC}^\kappa$, $\psi \circ \kappa(x, y) \cdot u$ est égal à $\psi(\kappa(x \cdot u, y))$. Si $\psi \circ \kappa(x, y)$ et $\psi(x \cdot u, y)$ convergent, alors $y = \varepsilon$, et donc $\psi \circ \kappa$ est bien dans $\mathcal{MPC}$. Les démonstrations se font de même dans l'autre sens, et pour les autres classes de préfixe $\mathcal{MPP}$ et $\mathcal{MPG}$. □

Corollaire 0.61 *Les couplages préfixants sont les couplages vérifiant $\kappa(x, y) = f(y) \cdot x$, avec f un codage préfixe.*

◇ *Preuve.* Pour tout mot $x \in \Xi$, par récurrence immédiate, $\kappa(x, y) = \kappa(\varepsilon) \cdot x$. Soit $f(y) = \kappa(\varepsilon, y)$. Supposons que f ne soit pas un codage préfixe, il existe u, v et w (avec $w \neq \varepsilon$) tels que $f(v) = f(u) \cdot w$. Cela veut dire en particulier que $\kappa(\varepsilon, v) = \kappa(w, u)$, ce qui veut dire $v = u$, ce qui est absurde. □

On peut par contre, pour le cas général, montrer qu'il existe des systèmes acceptables de programmation qui ne vérifient pas cette transition.

Proposition 0.62 *La fonction à deux arguments $\mathbf{KP}(x|y)$ est différente de la fonction $\mathbf{KP}^\kappa(x|y)$ pour le couplage $\kappa(x) = \overline{x}y$, même à une constante près.*

◇ *Preuve.* Supposons que ces fonctions soient égales à une constante près. Cela veut dire en particulier que l'on peut choisir une machine optimale pour la classe $\mathcal{MPP}^\kappa$ des machines préfixes pleines préfixes à un argument. Pour cette machine, $\forall y, \phi(\overline{x} \cdot \varepsilon) = \phi(\overline{x} \cdot y)$. On va montrer en fait que cette fonction ne peut pas refléter la propriété $\mathbf{KP}(x|x) = \mathcal{O}(() 1)$. Ceci est vrai dans le cas du calcul d'arité 2, parce qu'il suffit de prendre une fonction qui recopie le deuxième argument sur la sortie — elle est alors préfixe en le premier argument. Dans notre cas, cela ne peut pas être vrai. Soit une constance c, et S l'ensemble des mots de longueur inférieure ou égale à c. L'ensemble des mots tel que $\phi(\overline{p}x)$ soit convergent

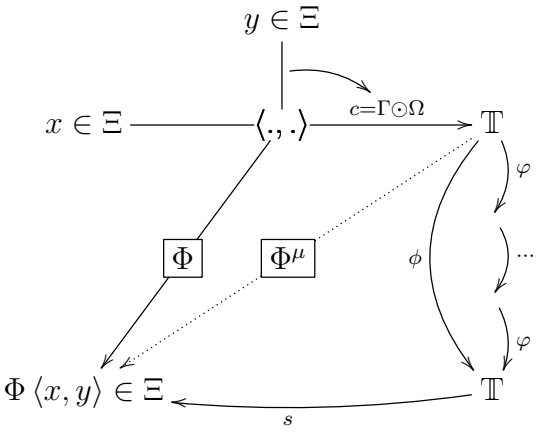


FIGURE 4.5 – Schéma du calcul à deux arguments

avec $p \in S$ vérifie la condition de Kraft pour chaque p distinct. Donc l'ensemble des mots calculés $F_p = \{\overline{p}x \downarrow\}$ est un code préfixe et vérifie en particulier :

$$\sum_{x \in F_p} |\xi|^{-\ell(x)} \leqslant 1 \quad \text{pour un même } p,$$

$$\sum_{p \in S} \sum_{x \in F_p} |\xi|^{-\ell(x)} \leqslant |S| \leqslant \frac{|\xi|^{c+1} - 1}{|\xi| - 1}.$$

Or cet ensemble de mots devrait être tous les mots de Ξ pour un certain c, qui ne vérifient pas cette inégalité (quelque soit c). Donc il y a une différence importante entre $\mathbf{KP}^\kappa$ et $\mathbf{KP}$. □

4.2.2 Schématisation du processus de calcul à deux arguments

Puisqu'il y a une différence entre avoir un ou deux arguments dans le cas général, il peut être intéressant de modifier notre schématisation du calcul pour y introduire le rôle du contexte, tel qu'utilisé dans la mesure de la quantité d'information, ou encore comme une forme d'oracle initial.

Le paragraphe précédent, et en particulier la proposition 0.62, montrent qu'il n'est pas simplement possible de considérer une machine à deux arguments comme préfixée par un couplage quelconque. Le choix du couplage est une étape importante. Des propriétés comme la notion de préfixe qui sont naturelles dans une représentations (par exemple, préfixe par rapport au premier argument, dans le cas du couplage $\kappa(x, y) = \overline{y}x$) deviennent non-naturelles dans une autre représentation ($\kappa(x, y) = \overline{x}y$). Donc ce couplage doit être figuré et spécifié dans la modélisation du calcul choisie.

Si on s'attache en particulier à la propriété préfixe du premier argument, on peut représenter la situation comme indiquée sur la figure 4.5. Si on reprend la décomposition

énoncée dans la proposition 0.54, on va trouver un résultat similaire. Ainsi on peut décomposer la transformation de x et y en un codage préfixe de $\langle x, y\rangle$. Pour avoir une vision plus dynamique qui rend compte de la différence constatée plus haut, on peut aussi dire que l'on a une transformation de x conditionnée par y (qui vérifie d'autres propriétés). Si cette transformation vérifie des conditions d'uniformité (par exemple, être préfixant), alors la propriété préfixe usuelle peut être identifiée à la propriété préfixe sur le modèle intermédiaire.

Il est donc intéressant de voir y comme étant un facteur qui influence le plongement, tout en s'y mêlant. Cette intervention externe se retrouve d'ailleurs dans plusieurs théorèmes sur la complexité, où l'on voit que des mesures de dénombrement se font à y constant (et sont indépendantes par ailleurs), comme la proposition 0.7.

4.3 Analogies du système d'exploitation

La notion de préfixe est donc importante pour la notion de calcul, car c'est ce qui caractérise ce que le système de calcul doit supporter. Bien qu'un ordinateur ne soit qu'un gigantesque automate fini (soumis éventuellement à des interactions continues), un système d'exploitation se doit d'être extensible et dans l'absolu, être organisé comme si les ressources à gérer étaient infinies. Ce qui nous amène tout naturellement au modèle de calcul qui été développé dans ce qui précède : un système d'exploitation peut être vu comme une modélisation du calcul (certes compliquée), arrangeant le fonctionnement. On néglige en cela l'aspect interactif de l'ordinateur, pour se concentrer sur son aspect purement fonctionnel.

À différents niveaux dans les systèmes d'exploitations interviennent des données à conserver sur disque (sur un ou plusieurs média), et dans la *mémoire vive*. Ces différentes données forment l'espace de travail de l'ordinateur. On expose ici en quoi les notions qui se sont dégagées de notre analyse de la théorie de l'information algorithmique viennent justifier les choix techniques qui ont souvent été pris.

La gestion de la mémoire La gestion de la mémoire est très certainement ce qui utilise le moins un codage préfixe dans un système d'exploitation. C'est également parmi les choses les plus difficilement extensibles.

De nos jours encore, une grande partie des ordinateurs sont au format « PC compatible IBM », utilisant une architecture à plusieurs étages pour la gestion de la mémoire. En particulier, l'accès à celle-ci est gérée après le démarrage par le BIOS, un système d'entrées-sorties conçu dans les années 70. Le BIOS permet en particulier l'adressage de la mémoire sur 640 kO, soit $10 \cdot 2^{19}$ bits. Même si ensuite d'autres mécanismes permettent d'exploiter la totalité de la mémoire (qui est plus de l'ordre de 2^{30} bits), les mécanismes de démarrage des machines sont toujours soumis à ces limitations.

Par ailleurs, le modèle de programmation impératif, encore très employé actuellement, utilise une représentation des liens entre objets sous forme d'un pointeur, très souvent de taille fixe, bornant ainsi la taille de la mémoire adressable. De plus en plus, les modèles

de programmation (objet et/ou fonctionnel) s'éloignent de la représentation physique d'un pointeur, ne le manipulant plus que dans l'abstrait.

Si on regarde la façon dont sont donc maintenant agencées les données, les indications sont beaucoup plus organisées conceptuellement qu'avec le modèle des adresses fixes. Ainsi, la majorité des programmes peuvent fonctionner (modulo une recompilation) sans modifications sur des systèmes disposant de plus de mémoire. Avec certains nouveaux systèmes de pseudo-code, comme utilisés pour des langages comme JAVA et OCAML, et de compilation au dernier moment, le même programme peut d'ailleurs servir directement sur des ordinateurs différents, la représentation des objets manipulés s'étant développée au point de s'abstraire du support.

Le jeu d'instructions du processeur Si la technique qui permet d'affirmer l'optimalité d'une machine (basée sur la machine universelle) peut sembler impraticable, une technique similaire est pourtant appliquée dans des processeurs très récents. Ainsi, le processeur Crusoë™ de TRANSMETA fonctionne grâce à un micro-code interne programmable, qui est capable d'interpréter d'autres jeux d'instructions (en fait, même pour utiliser le sien propre, il est nécessaire d'utiliser une forme de traduction). Une partie de la machinerie interprète comme une machine universelle du code, spécifié au préalable. Cela ressemble aux codages $\overline{n}p$ qui reproduisent le codage p le plus court fait par ϕ_n pour la machine additivement optimale. De même, les instructions sont codées de façon à être les plus courtes et denses possibles : le processeur analyse les premiers bits de chaque instruction d'abord, et dans certains cas seulement d'autres instructions. On distingue ainsi une structure « numéro de machine » (parmi un nombre très limité) suivi des arguments de cette machine.

Dans un autre registre, sur un processeur de type `Risc`, par exemple, on trouvera des opérations générales utilisant un grand espace de codage (l'addition de deux registres avec stockage dans un troisième nécessite des arguments plus longs, puisqu'il y en a 3, de même que le saut long à une autre adresse mémoire, qui doit avoir le plus de bits possibles pour éviter de décomposer un saut très long en petits sauts), alors que d'autres ont un codage des instructions plus long parce que le nombre d'arguments possibles est plus petit (le saut court est codé sur 24 bits d'instructions et 8 bits de déplacement, car il ne définit que des changements codables sur 8 bits). On utilise donc une compression optimale. À l'inverse, sur un processeur de type `Pentium`, les instructions vont avoir une syntaxe de longueur variable. On est contraint à une lecture linéaire des instructions, car le décodage n'est pas possible *a priori*.

Le système de fichiers Plus que la mémoire, qui est plutôt apparentée au nombre d'états de l'automate, il est possible de comparer les disques (de nature diverse) aux rubans des machines de Turing. La taille de ces périphériques de stockage de masse changeant rapidement, il est important d'avoir un système souple, permettant de gérer des tailles quelconques. Pour des raisons techniques, la structure logicielle n'a jamais été adaptée à une extension quelconque de la taille des disques. Toutefois, une partie des mécanismes qui résultent de décennies de développement marquent une tendance à rendre ces limites

indépendantes. Il faut dire qu'un système de fichiers doit répondre à des contraintes d'extensibilité d'échelle, mais aussi à des contraintes de rapidité d'accès et de redondance en cas de perte des informations qui font que la compression de ces données n'est pas une chose souhaitable. Les représentations les plus compactes sont donc souvent abandonnées ; il est même des cas où de la redondance explicite (recopie d'informations) est utilisée et dupliquée.

Par exemple, dans au moins un système de fichiers utilisé couramment (le format `ext2`, utilisé entre autres sous Linux), la structure arborescente des fichiers est codée par un double système. Le disque est divisé en blocs indivisibles de taille fixe, et en nombre borné. Mais outre cette limitation, l'indexation est effectuée par des blocs spéciaux contenant des informations qui codent cette structure (des *i-nœuds*). Chacun de ces i-nœuds a une taille fixe de 128 octets, et sert à pointer des zones de donnée de taille quelconque, éventuellement répartie dans des endroits séparés du disque.

Un i-nœud n'est en définitive qu'une liste de numéros de blocs. Toutefois, la taille fixe de ces i-nœud empêche de faire des fichiers de taille quelconque si l'on adopte pas un codage plus fin. C'est ce qui est fait par un système de référencement. Les 12 premiers blocs des fichiers sont indiqués directement (leur numéro est inclus dans les 128 octets, en plus d'informations de nature administrative sur le fichier considéré). Ensuite, viennent de 0 à 3 numéros de blocs, ayant la particularité de pouvoir coder des fichiers de taille de plus en plus grande en prenant le moins de place possible pour les informations internes. Le premier numéro pointe sur un bloc de références, qui est un bloc occupé uniquement par des numéros de blocs du fichier. Le deuxième numéro est un bloc de référencement double, c'est-à-dire que chaque bloc pointé ne contient lui-même que des listes de numéros de blocs de référencement. Enfin, le troisième numéro contient le numéro d'un bloc dont le contenu est interprété comme étant une liste de blocs de référencement double.

Ainsi, pour des blocs de 1024 octets, chacun peut contenir des références à 256 blocs. Un i-nœud sans référence peut pointer un fichier de taille 12 blocs au plus. Avec référence, on passe à 268 blocs. Avec double référence, on passe à $256^2+256+12 = 65804$ blocs. Avec la triple référence, on passe à 16 843 020, ce qui fait des fichiers de l'ordre de 16 Go. Avec des blocs de taille supérieure, on multiplie au cube la taille maximale des fichiers. Jusqu'à 12 ko, la place occupée par le fichier est d'exactement 128 octets plus sa taille arrondie par dessus à un multiple de 1024 octets. Au-delà, et jusqu'à 268 ko, il faut arrondir la taille aux 1024 octets supérieurs, et ajouter 1152 octets. Après, il faut ajouter encore 1024 octets, et pour chaque 256 ko supplémentaires, il faut ajouter 1024 autres octets, soit une taille de $128+1024+1024+\lceil(x-268)/256\rceil+1024*x$, où x est le nombre de kilo-octets du fichier (par excès), etc.

On peut encore bien améliorer ce système, par exemple d'un point de vue taille. On peut s'approcher théoriquement autant que l'on veut de $\mathbf{KP}(x)$. Mais les systèmes d'exploitations ont d'autres contraintes, notamment de temps de réponse, qui rendent impraticables l'usage de codes complexes. On pourrait aussi supprimer la possibilité de fragmentation, mais la dynamique des données impose la possibilité de cette fragmentation. Si on admet que la séparation en blocs est une nécessité, alors ce schéma de codage est très près de l'optimal.

Par ailleurs, la structure du système de fichiers est basée sur un arbre dont les nœuds sont des *répertoires*. Ces répertoires sont indexés comme étant des fichiers à part entière, chacun désigné par un i-nœud. On stocke le contenu des répertoires comme un fichier particulier. Les répertoires sont une suite de notations « numéro d'i-nœud – longueur de l'entrée – longueur du nom – nom du fichier ». Un code préfixe est donc utilisé. Ce n'est pas le seul code préfixe utilisé dans le système d'exploitation, mais celui-ci est remarquable parce qu'il n'utilise pas le caractère blanc (code ASCII 0) comme dans la majorité du système, mais une longueur du nom puis le nom.

Par ailleurs, comme expliqué plus haut, la compression n'est pas maximale : les informations essentielles à la structure du système de fichiers sont dupliquées régulièrement (racine de l'arbre des répertoires, et autres informations sur la taille du disque et d'autres informations système) pour permettre un contrôle en cas de perte de données accidentelles. La compression ne peut pas s'arranger de l'erreur, puisque la concentration des entrées valides empêcherait l'usage de codes correcteurs.

De plus, le code préfixe employé est simple. D'autres systèmes de fichiers utilisent des structures de données plus complexes, codant en particulier des *B-tree* pour la structure des répertoires, permettant un accès direct.

Conclusion

Il est possible de continuer ces recherches dans plusieurs directions, car plusieurs questions sont restées ouvertes au cours de notre exposé.

Dans le chapitre 3, nous avons examiné plusieurs codages possibles pour le mot de calcul, une fois qu'il est plongé dans l'espace de la machine. La première chose qui ait été utilisée est le codage simple, où la machine peut identifier facilement les deux extrémités du mot. La tête de lecture est, dans l'état initial, disposée sur une des extrémités, et on peut reconnaître l'autre extrémité au fait qu'elle est le passage d'un ensemble de lettres de codage à un caractère blanc spécial.

Dans un deuxième temps, nous avons modifié les connaissances de la machine sur sa propre entrée, au prix de l'augmentation de la redondance du codage utilisé : nous avons utilisé un codage préfixe. D'une certaine façon, la machine ne tient plus son entrée que par un seul point, et doit déduire la position de la fin du mot. Nous avons vu dans le chapitre 4 que sans modifier le codage, mais en diminuant les informations, on diminuait la puissance de calcul de la machine. Cette notion de « poignée » transparaît déjà dans l'analyse des modèles de calcul faite par Uspensky dans [US93a].

Dans un troisième temps, nous avons encore modifié les connaissances de la machine sur sa propre entrée. Au début, la machine ne connaît qu'une seule lettre du mot, sans en connaître aucune extrémité. La solution à ce problème paraît simple : augmenter encore la redondance du codage de l'entrée de façon à reconstituer les données manquantes, à savoir trouver les extrémités du mot. Mais l'usage de ces codes (appelés *comma free* dans [BP85]) mène à l'impossibilité de pouvoir mesurer la quantité d'information présente dans un objet, puisqu'il est impossible de vérifier alors la propriété d'additivité optimale pour cette classe de machines.

Toutefois, il existe des codes moins contraignants que les codes *comma free* mais porteurs de plus d'informations que les codes préfixes, comme les codes bi-préfixes (à ne pas confondre avec les codes précédents). Dans ceux-ci, l'information que l'on peut retrouver est une des deux extrémités, quelle que soit celle qui est donnée au départ. Cela reste un problème ouvert de savoir s'il est possible de construire une mesure d'information (variante de la complexité de Kolmogorov) sur la base de ces codages.

Un autre problème ouvert est la mesure à utiliser dans le chapitre 2. Lorsque l'alphabet de codage a été modifié pour passer de deux caractères (dans la théorie usuelle) à un alphabet de taille quelconque, pour garder la propriété de correspondance entre complexité de Kolmogorov et L-test universel (théorème 12), la probabilité d'obtenir un mot de longueur déterminée n a été changée d'une exponentielle de base $1/2$ à une exponentielle en base $1/|\xi|$. Il n'est pas encore très clair de voir quelle est l'influence réelle de la forme de cette

normalisation : il est certain qu'elle autorise moins de choses, puisque même si le gradient de « notes de rareté » autorisé est plus large, les exceptions aux règles de rareté doivent être moins fréquentes.

Également, il reste beaucoup de modèles à explorer du point de vue spécifique de la complexité de Kolmogorov. Si l'on sort du cadre classique de la calculabilité, on peut par exemple se demander s'il est possible de mettre à jour des caractéristiques spécifiques des modèles de calcul à partir desquelles on pourrait déduire une variante de la complexité de Kolmogorov sur ces modèles. C'est le cas dans le cadre du modèle quantique en particulier, du *DNA computing*, ou encore d'autres modèles. Dans ce cadre on peut faire rentrer l'étude générale qui a été faite sur les notions d'entropie ou « modes de description », tels que décrits dans [US96]. L'étude d'autres modèles dont la puissance est en dessous de la puissance des machines de Turing est aussi envisageable.

Enfin, la notion de codage préfixe utilisée est fortement linéaire. Que devient-elle lorsqu'on passe d'un modèle de calcul manipulant non plus des mots, mais des structures multi-dimensionnelles ? Quelle est l'expression naturelle de la complexité pour une forme disposée sur une grille 2D, par exemple, comme les triangles de calcul donnés au chapitre 2. On pourra par exemple, sur les formes en deux dimensions, regarder les travaux de [BN88].

Bibliographie

[BCFQ95] Braga (G.), Cattaneo (G.), Flocchini (P.) et Quaranta Vogliotti (C.), « Pattern growth in elementary cellular automata », *Theor. Comp. Sci.*, vol. 45, 1995, p. 1–26.

[BN88] Beauquier (D.) et Nivat (M.), « Codicity and simplicity in the plane ». Rapport technique n° 88–65, LITP, 1988.

[Bor12] Émile Borel, « Le calcul des intégrales définies », *Journal de Mathématiques pures et appliquées*, vol. 8, n° 2, 1912, p. 159–210.

[BP85] Berstel (J.) et Perrin (A.), *Theory of codes*. Academic Press, 1985.

[Cal93] Calude (C.), *Information and Randomness, an Algorithmic Perspective*. Springer-Verlag, 1993.

[CFMM99] Cattaneo (G.), Formenti (E.), Manzini (G.) et Margara (L.), « Ergodicity and regularity for cellular automata over Z_m », *Theor. Comp. Sci.*, vol. 233, n° 1-2, 1999, p. 147–164.

[Cha75] Chaitin (G. J.), « A theory of program size formally identical to information theory », *J. Assoc. Comput. Mach.*, vol. 22, 1975, p. 329–340.

[ČPY89] Čulik (K.), Pachl (J.) et Yu (S.), « On the limit set of cellular automata », *SIAM J. on Comp.*, vol. 18, 1989, p. 167–175.

[DDF00] Dubacq (J.-C.), Durand (B.) et Formenti (E.), « Kolmogorov complexity and cellular automata classification », *Theor. Comp. Sci.*, 2000. À paraître.

[Del99] Delorme (M.), « An introduction to cellular automata », dans Delorme et Mazoyer [DM99], p. 5–49.

[DM99] Delorme (M.) et Mazoyer (J.), éditeurs, *Cellular Automata, a Parallel Model*, coll. « Mathematics and Its Applications ». Kluwer Academic Publishers, 1999.

[Dub94] Dubacq (J.-C.), « Different kinds of neighborhood-varying cellular automata ». Rapport technique, Computer Science Departement, Indian Institute of Technology, Madras, 1994.

[Dub95] Dubacq (J.-C.), « How to simulate Turing Machines by invertible one-dimensional cellular automata », *International Journal for Foundation of Computer Science*, vol. 6, n° 4, decembre 1995, p. 395–402.

[For98] Formenti (E.), *Cellular automata and chaos : from topology to Kolmogorov complexity*. Thèse de doctorat, École normale supérieure de Lyon, LIP, 46

allée d'Italie, 69364 Lyon Cedex 07, octobre 1998. Téléchargeable sur le serveur `ftp.ens-lyon.fr`, répertoire `pub/LIP/Rapports/PhD/PhD1998/`, fichier `PhD1998-07.ps.Z`.

[Gác74] Gács (P.), « On the symmetry of algorithmic information », *Soviet Math. Dokl.*, vol. 15, 1974, p. 1477–1480.

[Gác78] Gács (P.), *Komplexität und Zufälligkeit*. PhD thesis, Mathematics Department, J. W. Goethe Universität, Frankfurt am Main, 1978.

[Gal68] Gallager (R. G.), *Information Theory and Reliable Communication*. Wiley, 1968.

[Har78] Harrison (M. A.), *Introduction to Formal Language Theory*. Addison-Wesley, 1978.

[Hed69] Hedlund (G. A.), « Endomorphism and automorphism of the shift dynamical system », *Mathematical System Theory*, vol. 3, 1969, p. 320–375.

[HU79] Hopcroft (J. E.) et Ullman (J. D.), *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[Hur90] Hurley (M.), « Attractors in cellular automata », *Erg. Theory & Dyn. Sys.*, vol. 10, 1990, p. 131–140.

[Kur97] Kurka (P.), « Languages, equicontinuity and attractors in cellular automata », *Erg. Theory & Dyn. Sys.*, vol. 17, 1997, p. 417–433.

[Lan90] Langton (C.), « Computation at the edge of chaos : phase transitions and emergent computation », *Physica D*, vol. 42, 1990, p. 12–37.

[Lev74] Levin (L. A.), « Laws of information conservation (non-growth) and aspects of the foundation of probability theory », *Soviet Math. Dokl.*, vol. 10, 1974, p. 206–210.

[lev76] « Various measures of complexity for finite objects (axiomatic description) », *Soviet Math. Dokl.*, vol. 17, 1976, p. 522–526.

[LV97a] Li (M.) et Vitányi (P.), *An Introduction to Kolmogorov Complexity and its Applications*, coll. « Graduate texts in computer science ». Springer-Verlag, 2e édition, 1997.

[LV97b] Li (M.) et Vitányi (P.), *An Introduction to Kolmogorov Complexity and its Applications*, chap. 3.9, p. 233, dans [LV97a], 2e édition, 1997.

[MK76] Maruoka (A.) et Kimura (M.), « Conditions for injectivity of global maps for tessellation automata », *Information & Control*, vol. 32, 1976, p. 158–162.

[MK79] Maruoka (A.) et Kimura (M.), « Injectivity and surjectivity for parallel maps for CA », *J. Comp. and Sys. Sci*, vol. 18, 1979, p. 47–64.

[ML] Martin-Löf (P.). « On the oscillation of the complexity of infinite binary sequences ». Travail non publié (en russe).

[ML66] Martin-Löf (P.), « The definition of random sequences », *Inform. Contr.*, vol. 9, 1966, p. 602–619.

[ML71] Martin-Löf (P.), « Complexity oscillations in infinite binary sequences », *Zeitschrift für Wahrscheinlichkeitstheorie und Verwandte Gebiete*, vol. 19, 1971, p. 225–230.

[MM97] Manzini (G.) et Margara (L.), « A complete and efficiently computable topological classification of linear cellular automata over Z_m », dans *24th International Colloquium on Automata, Languages and Programming (ICALP '97)*, coll. « Lecture Notes in Computer Science ». Springer-Verlag, 1997.

[Moo62] Moore (E. F.), « Machine models of self-reproduction », *Proc. Symp. Appl. Math.*, vol. 14, 1962, p. 13–33.

[MSU96] Muchnik (A. A.), Semenov (A. L.) et Uspensky (V. A.), « Mathematical metaphysics of randomness ». Rapport technique, Institute of New Technologies, Moscow 109004, Russia, 1996.

[MY78] Machtey (M.) et Young (P.), *An introduction to the general theory of algorithms*. Elsevier, 1978.

[Myh63] Myhill (J.), « The converse to Moore's garden-of-eden theorem », *Proc. Am. Math. Soc.*, vol. 14, 1963, p. 685–686.

[Pé67] Péters (R.), *Recursive functions*. Akadémiai Kiadó, Budapest, 1967.

[Rab63] Rabin (M.), « Real time computation », *Israel journal of mathematics*, vol. 1, n° 4, 1963, p. 203–211.

[Rog67] Rogers (H. Jr), *Theory of Recursive Functions and Effective Computability*. McGraw-Hill, 1967.

[Ros67] Rosenberg (A.), « Real-time definable languages », *Journal of the Association for Computing Machinery*, vol. 14, n° 4, 1967, p. 645–662.

[Sch71] Schnorr (C. P.), *Zufälligkeit und Wahrscheinligkeit ; Eine algorithmische Begründung der Wahrscheinlichkeitstheorie*, vol. 218 (coll. *Lecture Notes in Maths*). Springer-Verlag, 1971.

[US93a] Uspensky (V. A.) et Semenov (A. L.), *Algorithms : Main Ideas and Applications*. Kluwer Academic Publishers, 1993.

[US93b] Uspensky (V. A.) et Shen' (A. K.), « Relations between varieties of Kolmogorov complexities ». Rapport de recherche, CWI (Amsterdam), avril 1993.

[US96] Uspensky (V. A.) et Shen' (A. K.), « Relations between varieties of Kolmogorov complexities », *Mathematical Systems Theory*, vol. 29, 1996, p. 271–291.

[Usp94] Uspensky (V.), « Gödel's incompleteness theorem », *Theor. Comp. Sci.*, vol. 130, n° 2, août 1994, p. 239–319.

[Usp96] Uspensky (V. A.), « Kolmogorov complexity : Recent research in Moscow », dans *MFCS'96*, coll. « Lecture Notes in Computer Science ». Springer-Verlag, 1996.

[USS90] Uspensky (V. A.), Semenov (A. L.) et Shen' (A. K.), « Can an individual sequence of zeros and ones be random ? », *Russian Math. Surveys*, vol. 45, n° 1, 1990, p. 121–189.

[Wat92] Watanabe (O.), *Kolmogorov Complexity and Computational Complexity*. Springer-Verlag, 1992.

[Wol84] Wolfram (S.), « Computation theory of cellular automata », *Comm. in Math. Phys.*, vol. 96, 1984, p. 15–57.

[Wol86] Wolfram (S.), *Theory and Applications of Cellular Automata*. World Scientific, Singapore, 1986.

[ZL70] Zvonkin (A. K.) et Levin (L. A.), « The complexity of finite objects and the development of the concepts of information and randomness by means of the theory of algorithms », *Russian Math. Surveys*, vol. 25, n° 6, 1970, p. 83–124.

Résumé

La complexité de Kolmogorov donne une interprétation de la notion d'aléatoire pour les mots sur un alphabet fini. Ces notions ont conduit à l'explicitation de classes de fonctions calculables possédant des caractéristiques provenant de la théorie des codes : la classe des machines préfixes — machines dont le domaine est codé de façon préfixe, c'est-à-dire qu'un mot du domaine n'est jamais le préfixe d'un autre mot du domaine.

Outre la résolution du problème de l'aléatoire dans les mots infinis, cette classe de machine est stable vis-à-vis de la théorie de la calculabilité. On compare ici trois définitions distinctes de la notion de machine préfixe, mais remarquablement similaires. On étend ensuite les notions proposées aux codes *comma free* (sans facteurs). Certaines propriétés fondamentales sont alors non-vérifiées, comme l'existence d'une machine additivement optimale.

Enfin, on étudie la façon dont la notion de préfixe intervient dans la théorie de la calculabilité. On regarde en particulier les machines dont on limite le nombre de caractères (sans caractères blancs) ou encore la modélisation des modèles finis plongés dans des espaces de calcul infinis (ce qui est le cas de la machine de Turing, dotée d'un ruban infini).

Mots-clés : Complexité de Kolmogorov, Modèles de calcul, Codes préfixes, Aléatoire.

Abstract

Kolmogorov complexity theory gives a definition of randomness for words on a finite alphabet. The notions involved led to the description of a subclass of computable machines : the prefix computable machines, whose domain is a prefix code (no word in the domain is prefix of another one).

Beyond the matter of defining randomness for infinite words, this subclass has remarkable properties regarding computability theory. Three different definitions are given and compared. The case of comma free codes is also examined, but it doesn't yield anymore an additively optimal machine.

The notion of prefix also interacts with the computational models. An examination of machines with no blank characters or of finite models with an infinite calculus space (such as the Turing machine, which uses an infinite tape) reveals a strong influence of the prefix notion on computational processes.

Keywords : Kolmogorov complexity, Computability, Prefix codes, Randomness, Computation models.