INSTITUT POLYTECHNIQUE DE GRENOBLE

Thèse

pour l'obtention du titre de

DOCTEUR de l'INSTITUT POLYTECHNIQUE de GRENOBLE

Spécialité : Sciences Cognitives

Préparée au laboratoire : TIMC-IMAG

Dans le cadre de l'école doctorale : Ingénierie pour la Santé, la Cognition et l'Environnement

présentée et soutenue publiquement le 7 Décembre 2009 par

Vivien Robinet

Modélisation cognitive computationnelle de l'apprentissage inductif de chunks basée sur la théorie algorithmique de l'information

---

Directeurs de thèse :

Benoît Lemaire et **Mirta Gordon**

Composition du jury :

M	Jean-Paul Delahaye	Président	(LIFL, Lille)
M	Antoine Cornuéjols	Rapporteur	(LRI, Paris)
M	Fernand Gobet	Rapporteur	(Brunel University, Londres)
M	Robert French	Examinateur	(LEAD, Dijon)
Mme	Mirta Gordon	Directrice de Thèse	(TIMC, Grenoble)
M	Benoît Lemaire	Directeur de Thèse	(TIMC, Grenoble)

Remerciements

Je remercie chaleureusement Antoine Cornuéjols et Fernand Gobet d'avoir accepté d'être les rapporteurs de ma thèse, ainsi que Robert French qui a bien voulu être examinateur et Jean-Paul Delahaye être président de mon jury de thèse. Je remercie très sincèrement tous les membres de mon jury à la fois pour le temps qu'ils ont accepté de me consacrer mais également pour les questions aussi intéressantes que variées qui m'ont été posées durant ma soutenance.

Je tiens à remercier tout particulièrement mes directeurs de thèse, Benoît Lemaire et Mirta Gordon pour toute l'aide qu'ils m'ont apporté durant ma thèse. J'ai conscience qu'il s'agit d'une chance énorme que d'avoir eu des directeurs de thèse aussi ouverts et compréhensifs que Benoît et Mirta. Je viens de me rendre compte après un rapide calcul que sur la totalité de ma thèse, le temps que mes directeurs m'ont consacrés se compte en centaines d'heures.

Je remercie donc Benoît pour les centaines de fois où je suis allé le déranger dans son bureau et où je l'ai interrompu en plein milieu d'un mail, d'une ligne de code ou d'une idée. Je remercie également Mirta pour les heures que nous avons passé à discuter et grâce auxquelles j'ai pu remettre en question et clarifier de nombreuses idées. Durant ces trois ans, j'ai usé et abusé des conseils avisés de mes directeurs, sans lesquels je n'aurais jamais pu réaliser ma thèse telle qu'elle est. J'espère de tout coeur avoir la possibilité dans un avenir proche de pouvoir travailler à nouveau avec eux.

Je remercie également Gilles Bisson qui m'a offert un regard extérieur sur mon travail, une oreille attentive aux problèmes rencontrés et des suggestions simples et astucieuses. La grande force de persuasion de Gilles s'est notamment illustrée par le changement du système d'exploitation de mon ordinateur.

Je remercie également les autres doctorants de l'équipe : Viktoryia Semeshenko pour son « *Vivien, you are a disaster !* » , Fawad Hussain pour ses « *An other day at grad school !* » et Myriam Chanceaux pour son indéfectible capacité à trouver sur internet des réponses avant même la fin des questions.

Je remercie mes parents ainsi que David Souveton pour avoir impitoyablement traqué et éliminé les fautes en tous genres présentes dans cette thèse. Merci donc pour leur courage et leur acharnement. Je remercie également mes soeurs de ne pas avoir corrigé les fautes d'orthographe et donc de ne pas en avoir ajouté ... (aucune contribution n'est assez modeste pour ne pas être mentionnée).

Je remercie Welore Tamboura pour l'aide qu'elle m'a apporté dans la rédaction *in extremis* de mon dernier chapitre.

Résumé

Cette thèse présente un modèle cognitif computationnel de l'apprentissage inductif. Le modèle proposé est appelé MDLChunker car il se base d'une part sur le principe de *Minimum Description Length* (MDL), et d'autre part sur le mécanisme de *chunking*.

Le mécanisme de *chunking* est couramment utilisé en psychologie cognitive dans des domaines aussi variés que l'apprentissage de grammaires artificielles, la segmentation de mots ou la modélisation de la mémoire à court terme. Il est à la base de nombreux modèles cognitifs computationnels.

Le MDL est quant à lui utilisé comme formalisation du « principe de simplicité » ou « rasoir d'Occam » . Il permet d'implémenter la notion vague de simplicité grâce au concept clairement défini de taille de codage. Les résultats théoriques qui justifient le principe de simplicité peuvent être établis grâce à la théorie algorithmique de l'information dont le MDL fournit une approximation calculable.

Utilisant ces deux mécanismes (MDL et *chunking*), le MDLChunker est capable de générer automatiquement la représentation la plus courte (la plus simple) d'un ensemble de stimuli discrets. Les représentations ainsi produites sont comparées à celles créées par des participants humains confrontés aux mêmes stimuli. À travers le modèle et les expériences proposées, le but de cette thèse est d'évaluer à la fois les fondements théoriques et l'efficacité pratique du principe de simplicité dans le cadre de la modélisation cognitive.

Abstract

This thesis presents a computational cognitive model of inductive learning. Because it is based both on the Minimum Description Length principle (MDL), and on the chunking mechanism, the model is called MDLChunker.

The chunking process is broadly used in cognitive psychology in areas as diverse as artificial grammar learning, word segmentation and short term memory modeling. It is used as a basis for many computational cognitive models.

The MDL principle is a formalisation of the simplicity principle (Occam's razor). It implements the fuzzy notion of simplicity through the well-defined concept of codelength. The theoretical results justifying the simplicity principle are established through the algorithmic information theory. The MDL principle could be considered as a computable approximation of measures defined in this theory.

Using these two mechanisms (MDL and chunking) the MDLChunker automatically generates the shortest representation of discrete stimuli. Such a representation could be compared to those produced by human participants facing the same set of stimuli. Through the proposed model and experiments, the purpose of this thesis is to assess both the theoretical and the practical effectiveness of the simplicity principle for cognitive modeling.

Table des matières

Introduction

Problématique

Notre système cognitif doit faire face à un monde qui est à la fois extrêmement complexe mais également hautement régulier (Chater, 1999). Ce sont ces régularités qui permettent la compréhension, la prédiction et donc l'action. La formidable capacité du système cognitif à extraire les régularités du monde extérieur est un problème qui a été largement étudié et qui continue à l'être.

La présente thèse s'inscrit dans ce cadre et s'intéresse plus particulièrement au « principe de simplicité » (Chater, 1999 ; Chater & Vitanyi, 2003) selon lequel les motifs que le système cognitif est susceptible d'extraire sont ceux qui permettent la représentation la plus simple du monde extérieur. Les motifs les plus prégnants seraient alors ceux qui autorisent la meilleure compression des stimuli perçus (Chaitin, 2002). Nous proposons ici un modèle computationnel implémentant ce principe, dans le but de tester sa capacité à prédire correctement les représentations qu'un humain est capable de créer à partir d'un ensemble de stimuli donné.

Ce principe très général possède d'excellentes propriété théoriques que nous aborderons en détail. Il offre également la possibilité d'être implémenté rigoureusement, de façon à fournir des prédictions quantitatives très précises permettant de trancher entre diverses représentations concurrentes. Ce principe ne fait en revanche aucune hypothèse sur la nature des régularités pouvant être perçues. On sait par ailleurs que le système cognitif n'est en mesure d'extraire que certains types de régularités (Oaksford & Chater, 1993). En voici quelques exemples :

$$0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1 \tag{1}$$

$$1, 2, 4, 8, 16, 32, 64, 128, 256, 512 \tag{2}$$

$$0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1 \tag{3}$$

$$8, 8, 8, 8, 8, 8, 8, 8 \tag{4}$$

$$13, 24, 35, 46, 57, 68, 79 \tag{5}$$

$$1, 2, 1, 0, 0, 3, 7, 3, 0, 0, 1, 2, 1 \tag{6}$$

S'il était demandé de se souvenir des six séquences, il est probable que certaines seraient rappelées bien plus facilement que d'autres, parce qu'elles sont plus régulières, donc susceptibles de subir une bien meilleure compression. C'est notamment le cas de la séquence 4 pour laquelle il suffit de mémoriser le chiffre « 8 » ainsi que le fait qu'il s'agit à la fois de la longueur de la séquence et du chiffre répété.

Les séquences 2, 3 et 5 font appel à la notion d'addition et la séquence 6 à celle de symétrie qui sont des connaissances de relativement haut niveau. Dans cette thèse, nous nous intéresserons uniquement aux régularités semblables à celles qui sont observables dans les séquences 1 et 5. Ce type de régularités est appelé *chunk* et correspond à un regroupement d'éléments qui apparaissent fréquemment conjointement (Miller, 1956). Le repérage de ce type de régularités dans un environnement bruité n'est pas un tâche aussi simple qu'il y paraît sur ces exemples.

Approche envisagée

L'approche que nous avons choisi est de concevoir un modèle cognitif computationnel basé sur le principe de simplicité, dont le but est de prédire la création de chunks. Ce modèle appelé MDLChunker, est capable de prendre en entrée un ensemble de stimuli, puis de les comprimer en utilisant les chunks qu'ils contiennent, construisant ainsi une représentation compressée de l'environnement extérieur. Au cours de ce processus, le MDLChunker n'introduit ni ne retire aucune information des stimuli originaux, il se contente de modifier leur représentation en créant des chunks.

Nous verrons que cette compression se fait dans un cadre rigoureux basé sur la théorie de l'information (Shannon, 1948), qui, sous certaines conditions, permet de garantir l'optimalité de la compression obtenue. Mais l'objectif ultime du MDLChunker n'est pas d'effectuer cette compression dans le but de diminuer la taille de la mémoire utilisée, mais parce que sous cette forme compressée les représentations vont posséder une propriété fondamentale : la capacité de prédire les stimuli futurs avec le plus faible taux d'erreur (Vitanyi & Li, 2000). La représentation compressée des données est surtout intéressante car il s'agit de la plus prédictive. C'est ce qui justifie la plausibilité cognitive du principe de simplicité (Chater, 1999). Le fait que la représentation correspondante soit la plus courte apparaît comme un sous-produit de cette compression et non comme l'objectif principal.

Modélisation cognitive

C'est en grande partie pour cette raison que le principe de simplicité a été proposé par Chater et Vitanyi (2003) comme un principe fondamental en sciences cognitives. La notion de *chunk* qui constitue l'autre partie du modèle proposé,

possède elle aussi une longue tradition en psychologie cognitive. Elle a notamment été utilisée pour décrire certains mécanismes tels que l'apprentissage de grammaires artificielles (Servan-Schreiber & Anderson, 1990), l'acquisition du lexique (Perruchet & Vinter, 1998) et de la syntaxe (Dowman, soumis), l'apprentissage de la morphologie (Goldsmith, 2001), mais également pour décrire des mécanismes de plus haut niveau comme l'expertise au jeu d'échecs (de Groot, Gobet, & Jongman, 1996).

En associant les deux, le but poursuivi est de concevoir un modèle cognitif dont les principes de fonctionnement soient fondés au niveau théorique. Sur le plan pratique, nous verrons que le modèle proposé permet non seulement de prédire quels chunks des participants humains sont susceptibles de créer face à un ensemble de stimuli donnés, mais également qu'il permet de prédire le décours temporel de leur création.

Plan de la thèse

Cette thèse se divise en trois grandes parties : théories, modélisations et validations. La première concerne les différentes justifications théoriques du modèle, la seconde comprend la description du modèle proprement dit et sa comparaison avec des modèles similaires, et la troisième partie s'intéresse à la validation pratique du modèle dans différents domaines.

Nous discuterons tout d'abord brièvement des enjeux et des difficultés liés la modélisation cognitive computationnelle (chapitre 1), en discutant notamment de la plausibilité cognitive du principe de simplicité et du mécanisme de *chunking*.

Nous verrons (chapitre 2) qu'il existe un mesure universelle de la complexité (Chaitin, 1966 ; Kolmogorov, 1968 ; Solomonoff, 1960), et que bien qu'elle soit incalculable en pratique, cette mesure permet de définir un cadre formel très riche dans lequel il est possible d'exprimer le principe de simplicité.

Cette mesure fournit également certains outils théoriques indispensables à toute modélisation, et notamment une formalisation du dilemme biais-variance (chapitre 3). Nous verrons alors qu'une approximation calculable de cette mesure universelle peut être trouvée en utilisant la théorie de l'information.

Le fonctionnement du modèle créé à l'aide de ces outils sera ensuite décrit (chapitre 4). Ce modèle qui constitue le coeur de la thèse, implémente de façon *brute* les principes abordés dans la partie théorique. S'il a le mérite d'être simple, la charge de calcul qu'il impose est trop importante pour être plausible cognitivement.

Une version approchée de ce modèle sera donc proposée (chapitre 5) afin de tenir compte des limitations cognitives évoquées.

Une fois le modèle décrit, nous serons alors en mesure d'effectuer une comparaison avec d'autres modèles existants dans la littérature (chapitre 6).

Après avoir justifié les besoins de la création d'une nouvelle expérience, nous

procéderons à la validation du modèle. Pour cela, nous comparerons les résultats obtenus par le MDLChunker avec ceux obtenus par des participants humains soumis au même ensemble d'apprentissage (chapitre 7).

Puis nous évaluerons enfin la généralité du modèle en l'appliquant à des domaines pour lesquels il n'a pas été conçu : la segmentation de mots (chapitre 8) et la modélisation de la mémoire à court terme (chapitre 9).

Première partie

Théories

Chapitre 1

Modélisation cognitive computationnelle

Sommaire

The cognitive system can learn to deal with a remarkably broad range of challenges, both natural and artificial, from unicycling to backgammon to musical composition. It acquires, stores, and can flexibly retrieve, an immensely rich understanding of the everyday world. It seems plausible that, as for other biological structures, this success is not accidental - rather, the cognitive system seems more likely to be superbly adapted to serve practical and computational ends.

Chater and Oaksford 1999

1.1 Introduction

Ce premier chapitre joue essentiellement un rôle introductif. Il permet de décrire le contexte de cette thèse, « la modélisation cognitive computationnelle » , en insistant sur la plausibilité cognitive du principe de simplicité et du mécanisme de *chunking*.

1.2 Modularité de la cognition

Le type de modélisation que nous avons choisi s'inscrit dans la démarche classique utilisée en sciences expérimentales. Elle consiste à essayer d'isoler un phénomène donné afin de pouvoir l'étudier. Lorsque plusieurs phénomènes ont été modélisés avec précision, il est parfois possible de les regrouper au sein d'un même principe plus général.

Il existe une approche totalement différente dans le champ particulier des sciences cognitives. Il nous a semblé intéressant de la mentionner afin d'expliquer d'emblée les raisons qui nous on poussé à l'écarter. Cette approche proposée par Ohlsson notamment, consiste à supposer que la cognition est composée de plusieurs modules ayant leur fonctionnement propre mais ne pouvant être analysés séparément. L'étude du système cognitif doit alors porter sur l'intégralité de la chaîne de traitement, des mécanismes perceptifs jusqu'aux mécanismes de plus haut niveau. La différence entre les deux démarches porte sur la modularité de la cognition et sur l'indépendance des différents mécanismes impliqués. Nous détaillons maintenant cette seconde approche afin de clarifier les raisons pour lesquelles nous l'avons rejetée.

Selon Ohlsson (2008), pour expliquer l'acquisition de connaissances il est nécessaire de spécifier un répertoire de mécanismes d'apprentissage, chacun étant défini par ses conditions de déclenchement, et par la modifications des connaissances qui en résultent. Pour qu'un modèle computationnel puisse rendre compte du large panel de compétences du système cognitif humain, il doit intégrer l'ensemble le plus vaste possible de mécanismes d'apprentissage. C'est l'assemblage des ces différents mécanismes qui permet de rendre compte de l'acquisition de connaissances dans un contexte écologique.

Toujours selon Ohlsson (2008), il est improbable que la capacité du système cognitif à s'adapter à des situations différentes puisse découler d'un principe unique. Plus grave encore, il est illusoire de penser valider chaque mécanisme indépendamment des autres : le comportement d'un ensemble de N mécanismes étant plus que la somme des comportements de chacun d'eux. Il n'est donc pas possible de valider le N-ième mécanisme en cherchant à limiter l'influence des N-1 autres. La cognition doit alors être pensée comme un ensemble de mécanismes fonctionnant en parallèle, dont l'étude doit être faite dans sa globalité, aucun d'eux ne pouvant être étudié indépendamment des autres. La conclusion de l'auteur est que si un mécanisme isolé est capable de prédire avec succès un phénomène, alors il doit être rejeté, et ceci avec d'autant plus de conviction

que la prédiction est bonne. Si un mécanisme isolé produit de bons résultats, il est très probable qu'il en produise de mauvais en interaction avec d'autres mécanismes. Ce raisonnement se justifie par le fait qu'il y a toutes les raisons de penser que la cognition est composée de multiples mécanismes dont les fonctionnements sont hautement inter-dépendants (Ohlsson, 2008).

Cette approche s'oppose à celle traditionnellement utilisée consistant à diviser le problème colossal que représente la modélisation de la cognition dans son ensemble, en sous-problèmes indépendants : modélisation de la mémoire à court terme, modélisation du phénomène d'oubli, etc. Si l'approche proposée par Ohlsson a le mérite d'être plus générale, puisqu'elle englobe l'autre [1], l'espace de recherche considéré est bien trop grand pour pouvoir être exploré. La cognition étant considérée comme le fruit de N mécanismes en interaction, il suffit que la modélisation de l'un d'eux soit incorrecte pour que le modèle général le soit. En réponse à ce problème, Ohlsson (2008) propose une solution plus philosophique qu'expérimentale, qui consiste à accepter ou rejeter un modèle sur la seule intuition qu'on en a et non pas sur son accord ou désaccord avec les données expérimentales.

L'approche que nous avons choisi est plus traditionnelle. Au lieu de supposer qu'il existe plusieurs mécanismes différents ayant leur fonctionnement propre mais devant être étudiés conjointement, nous avons supposé l'existence d'un principe unique (décrit ci-après) dont découlent différents mécanismes, qui sont suffisamment indépendants pour pouvoir être validés isolément. Cette approche suppose une certaine modularité de la cognition.

La métaphore de la modularité de la cognition (Fodor, 1983) est de voir cette dernière comme un couteau suisse plutôt que comme une unique lame multi-usages (Cosmides & Tooby, 1995). La justification de la modularité est à la fois anatomique, avec l'existence de modules spécifiques (Damasio, 1995), et fonctionnelle dans la mesure où elle offre un avantage indiscutable dans le processus de sélection naturelle (Sun, 2004). Pour fixer les idées sur le type de découpage modulaire qui peut être fait, voici une liste non-exhaustive des modules fonctionnels proposés par Cosmides et Tooby (1995) :
– module de perception spatiale
– module de reconnaissance faciale
– module syntaxique
– module sémantique
– module *theory of mind*
La modularité de certaines fonctions n'est pas incompatible avec l'existence de mécanismes généraux trans-modulaires. En termes de modélisation cognitive, il est préférable selon Sun (2004) de supposer l'existence de mécanismes généraux, puis de les adapter en les spécialisant. C'est l'approche que nous avons suivie, et qui s'intègre dans le cadre proposé par l'analyse rationnelle de la cognition (Anderson & Bellezza, 1993). Dans ce cadre, un modèle doit à la fois dériver d'un principe général, et être formalisé de façon assez précise pour pouvoir être implémenté sous la forme d'un programme informatique dont la validité

1. Considérer les différentes modalités cognitives comme indépendantes est un cas particulier du problème qui consiste à les considérer comme dépendantes.

peut être testée expérimentalement [2]. L'utilisation d'un principe général limite la création de modèles *ad-hoc*, et la validation expérimentale supprime l'emploi d'hypothèses a priori dont la validité est invérifiable. Ce qui distingue cette approche de celle de Ohlsson (2008) est qu'un modèle ne peut être validé sur une simple croyance a priori.

1.3 Analyse rationnelle de la cognition

L'analyse rationnelle de la cognition trouve son origine dans l'article de Anderson et Milson (1989). Amplement reprise par la suite, cette approche s'oppose à l'analyse « mécanistique » de la cognition. Cette dernière étudie la cognition du point de vue des mécanismes mis en jeu, tandis que l'analyse rationnelle s'intéresse principalement aux buts et aux fonctions remplis par la cognition. Selon Chater et Oaksford (1999), l'analyse mécanistique tend à présenter une vision fragmentée de la cognition, dans laquelle le système cognitif est vu comme un assemblage de mécanismes arbitraires, chacun étant soumis à ses limitations propres, et dont la fonction générale n'apparaît pas clairement. À l'inverse, l'analyse rationnelle étudie les mécanismes cognitifs à la lumière de la fonction qu'il remplissent. Cela revient à supposer que le système cognitif est une solution quasi-optimale à un problème que l'on cherche à définir.

L'analyse rationnelle proposée par Anderson (1990) se décompose en six étapes :

1. **Buts :** spécification des buts à atteindre.
2. **Environnement :** définition des propriétés de l'environnement dans lequel le système va évoluer.
3. **Limitations :** spécification des limitations connues au niveau cognitif (limitations perceptives, computationnelles, etc.).
4. **Optimisation :** recherche de la solution optimale permettant d'atteindre les buts (1) dans l'environnement (2) avec les limitations (3).
5. **Validation :** validation sur des données empiriques des prédictions effectuées par le système.
6. **Itération :** répétition des cinq premières étapes en vue d'améliorer le système.

L'analyse rationnelle s'intéresse essentiellement à la solution optimale permettant d'atteindre le but fixé, en supposant que la cognition en est une approximation. Cela suppose implicitement que la solution sous-optimale atteinte par le système cognitif possède un fonctionnement suffisamment proche de celui de la solution optimale. Cette hypothèse est assez forte car rien ne garantit que deux solutions proches en performance soient nécessairement proches dans leur fonctionnement [3]. Pour plus de détail, le lecteur peut se reporter à Oaksford et Chater (1998).

2. Se reporter à Lane et Gobet (2003) pour une discussion sur l'intérêt des modèles computationnels pour effectuer les prédictions quantitatives et reproductibles

3. Selon la structure de l'espace, deux maxima locaux éloignés peuvent posséder des valeurs proches.

Dans une perspective évolutionnaire, l'approche envisagée par l'analyse rationnelle est séduisante car elle suppose que le système cognitif est le fruit d'une évolution lui ayant permis de s'adapter de façon quasi-optimale [4] à une fonction donnée.

Le travail présenté dans cette thèse s'inclut parfaitement dans le cadre de cette analyse rationnelle de la cognition. Si l'on voulait exprimer le travail réalisé à la lumière de ces six points, cela donnerait :

1. **But :** le système cognitif tend à créer les représentations les plus simples du monde extérieur car ce sont les représentations les plus prédictives (chapitres 1 et 2).
2. **Environnement :** le monde extérieur est en grande partie composé d'éléments apparaissant spatialement ou temporellement groupés. Ce type de régularités est aisément représentable sous forme de chunks (chapitre 1).
3. **Limitations :** la taille de la mémoire et le volume des calculs effectués par le système cognitif ne peuvent excéder une certaine limite (chapitre 5).
4. **Optimisation :** il est possible de montrer que la solution trouvée (le modèle proposé) est proche de l'optimum (chapitre 3 pour une discussion de l'optimum sur la plan mathématique et chapitre 4 pour l'implémentation correspondante).
5. **Validation :** la comparaison des prédictions du modèle et des résultats de participants humains sur une même tâche montre des comportements similaires (chapitre 7).
6. **Itération :** de légères modifications sont ajoutées au modèle pour l'appliquer à de nouveaux domaines (chapitres 8 et 9).

Comme cela est indiqué dans le point (1), l'idée sur laquelle va s'appuyer la suite de cette thèse est que le système cognitif crée des représentations en accord avec le principe de simplicité. Ce principe va maintenant être décrit plus en détail, en discutant notamment de sa plausibilité cognitive.

1.4 Principe de simplicité

Dans un article de référence, Chater et Vitanyi (2003) ont suggéré que de nombreux phénomènes cognitifs pouvaient être expliqués à partir d'un unique principe appelé « principe de simplicité » . Ce principe peut être résumé de la façon suivante : parmi toutes les représentations possibles qui peuvent être créées à partir du monde extérieur, le système cognitif tend à choisir systématiquement la plus simple.

Ce qui rend ce principe particulièrement intéressant est qu'il est susceptible de s'appliquer à divers niveaux de granularité : des processus sensoriels de bas niveau, où il reprend l'idée introduite par le *gestalt principle* (Wertheimer, 1922),

4. Nous employons le terme « quasi-optimal » de préférence à « sous-optimal » qui est souvent employé pour insister sur la mauvaise qualité de la solution trouvée.

jusqu'à des processus de plus haut niveau comme l'acquisition de la syntaxe (voir section 3.3). Ce principe pourrait notamment rendre compte de la capacité que possède le système cognitif à résoudre le problème de l'induction.

Problème de l'induction

Pour tout jeu de données de longueur finie, il est toujours possible de trouver une infinité d'explications possibles, *i.e.* qui soient compatibles avec le jeu de données. Par exemple, pour :

$$2, 4, 8, 16, 32 \tag{1.1}$$

les explications

$$x_i = 2^i \tag{1.2}$$

et

$$x_i = \begin{cases} 2^i & \text{si } i < 6 \\ 0 & \text{sinon} \end{cases} \tag{1.3}$$

sont toutes les deux possibles.

Parmi toutes les explications qui sont compatibles avec un phénomène observé, nous sommes en mesure d'en choisir une. Le système cognitif possède donc un critère permettant d'effectuer ce choix. On peut logiquement supposer qu'il ne s'agit pas d'un choix totalement aléatoire, puisque partant d'une même observation, tout être humain en arrive généralement à la même explication [5]. Pour quelle raison l'explication 1.2 apparaît-elle plus probable que l'explication 1.3 ?

Prédictions

Supposer que le système cognitif choisisse une explication plutôt qu'une autre au regard de sa simplicité permet d'apporter une solution au problème de l'induction (Chater & Vitanyi, 2003). Mais ce qui justifie cette solution particulière plutôt que toute autre, est d'une part que l'explication la plus simple est également la plus vraisemblable au regard des données (Chater, 1996)(discuté section 3.3), et d'autre part qu'elle est la plus prédictive (Vitanyi & Li, 2000). Ce dernier point est essentiel pour justifier la plausibilité cognitive du principe de simplicité (Chater, 1999).

Mesurer la simplicité

La difficulté majeure de cette approche est qu'il faut pouvoir être en mesure de quantifier la notion de simplicité. La manière la plus naturelle est sans doute

5. Ou du moins est capable d'évaluer qu'il n'est pas en mesure de fournir une explication convenable.

d'utiliser la relation d'inclusion : si deux explications sont également prédictives et que l'une est incluse dans l'autre, alors elle doit être privilégiée. C'est le principe connu sous le nom de « rasoir d'Occam » dont le but est d'éviter l'ajout d'hypothèses superflues. Cette mesure suffit à rejeter l'explication 1.3 au profit de l'explication 1.2.

Mais une mesure basée sur l'inclusion est beaucoup trop restrictive en pratique, et d'autres mesures ont été développées. Par exemple la dimension de Vapnick-Chervonenkis, qui représente le cardinal du plus grand ensemble de points qu'un classifieur est capable de pulvériser (séparer), et qui mesure donc la complexité d'un classifieur à son pouvoir de séparation (Vapnik & Chervonenkis, 1971).

Dans les années 1960, Chaitin (1966), Kolmogorov (1968) et Solomonoff (1960) ont défini une mesure, à la fois universelle et objective, de la complexité d'une chaîne binaire. Toute explication pouvant être mise sous cette forme (notamment un programme informatique), la mesure ainsi fournie possède toute la généralité souhaitée. Outre son incalculabilité pratique, qui a donné lieu à de nombreuses approximations pour la plupart basées sur la théorie de l'information, cette mesure de complexité est d'un intérêt limité dans le cadre de la modélisation cognitive. En effet, elle se base sur l'utilisation de machines de Turing (Turing, 1936) qui ont la capacité de capturer des régularités bien plus complexes que ne peut le faire un cerveau humain (Chater, 1996 ; Oaksford & Chater, 1993 ; van Rooij, 2008).

Au niveau de la modélisation cognitive, l'objectif est de partir des régularités que l'on sait le cerveau susceptible de capturer, afin de faire des hypothèses sur le type de représentations qui peuvent en découler. Lorsqu'il est soumis à un environnement extérieur, les perceptions d'un agent cognitif sont nécessairement contraintes par son système sensoriel (Sun, 2004), et les régularités qu'il est capable de percevoir sont limitées. Il est facile de percevoir la régularité dans la chaîne suivante :

$$1,1,0,1,1,0,1,1,0,1,1,0,1,1,0,1,1,0,1,1,0,1,1,0,1,1,0,1,1 \tag{1.4}$$

mais il est plus difficile de reconnaître la décomposition binaire de Pi dans celle-ci :

$$1,1,0,0,1,0,0,1,0,0,0,0,1,1,1,1,1,1,0,1,1,0,1,0,1,0,0,0,1 \tag{1.5}$$

Nous avons donc choisi de restreindre notre étude à un seul type de régularité : les *chunks*.

1.5 Mécanisme de *chunking*

Parmi les différentes régularités qu'un agent cognitif est susceptible de percevoir, la plus immédiate est sans doute la régularité de type « ET » , qui correspond à une simple association d'éléments, comme dans l'exemple 1.4 ci-dessus : 1 et 1 et 0. La représentation correspondant à ce type de régularité est appelée *chunk* et le mécanisme associé est appelé *chunking*. Ce mécanisme a été largement

utilisé et décrit en psychologie cognitive et sa plausibilité cognitive n'est plus à démontrer. Le lecteur peut se reporter à Gobet et al. (2001) pour une revue de la plausibilité cognitive du mécanisme de *chunking* et des modèles de *chunking*.

Un chunk est généralement défini en termes qualitatifs et non pas en termes quantitatifs. Gobet et al. (2001) définissent un chunk comme étant une collection d'éléments entretenant de fortes associations entre eux, et de faibles associations avec les autres éléments. Ainsi, un chunk ne représente pas nécessairement une quantité d'information fixe telle qu'elle pourrait être mesurée par un algorithme de compression, mais plutôt une quantité d'information cognitive, dépendante des connaissances possédées. Par exemple :

$$1 \tag{1.6}$$

est un chunk, au même titre que

$$14 \tag{1.7}$$

si l'on est capable d'y associer le département du Calvados, ou même

$$14071789 \tag{1.8}$$

pour ceux qui y reconnaîtront le jour de la prise de la Bastille.

Ainsi, Miller (1956) définissait la taille de la mémoire à court terme comme contenant 7 chunks [6], sans qu'il soit possible de quantifier précisément le contenu d'un chunk.

1.6 Simplicité et *chunking*

Bien que le choix de se focaliser uniquement sur des régularités de type ET (chunks) ait été fait pour des raisons de plausibilité cognitive uniquement, cela permet de résoudre également le problème évoqué précédemment quant à la difficulté de mesurer la simplicité. En supposant que les seules régularités possibles sont des chunks, cela signifie que, les chunks mis à part, il n'existe aucun lien de dépendance entre les différents éléments constitutifs d'un stimulus : les différents éléments sont supposés être des réalisations indépendantes et identiquement distribuées d'une même variable aléatoire. Dans ce cadre, une mesure de la complexité peut être donnée à l'aide d'un code de Shannon-Fano (Shannon & Weaver, 1949). C'est ce mécanisme qui sera utilisé par le MDLChunker (chapitre 4).

1.7 MDLChunker

La plausibilité cognitive du principe de simplicité d'une part, et du mécanisme de *chunking* d'autre part, a conduit à la création d'un modèle basé sur ces deux

6. Ce chiffre a depuis été revu à la baisse : 3 à 4 chunks selon Cowan (2005) et même 2 chunks selon Gobet et Clarkson (2004)

principes. Le MDLChunker prend en entrée une liste de stimuli en provenance du monde extérieur, et crée de nouveaux chunks lorsque ceux-ci permettent une description plus simple des stimuli (Fig. 1.1).

Des chunks comme le chunks A ou le chunk B de la figure 1.2 n'ont aucun intérêt car il n'apportent pas de compression supplémentaire par rapport aux chunks déjà possédés. En revanche, le chunk C est susceptible d'être créé car il permettrait une meilleure compression des stimuli 2 et 4 notamment.

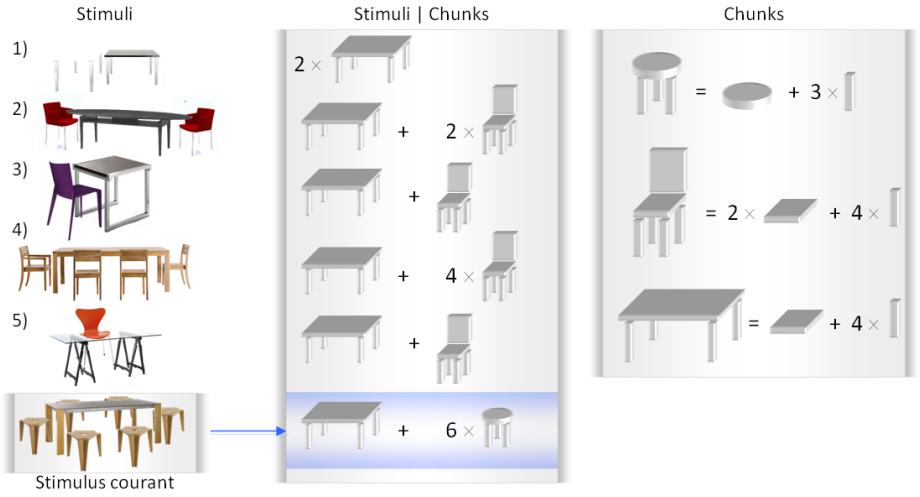


FIGURE 1.1 – Exemple illustrant le fonctionnement du MDLChunker. La partie chunks contient les chunks déjà créés au vu des régularités présentes dans les stimuli. Ici, trois chunks ont été créés. Ces chunks sont en retour utilisés pour encoder les stimuli (partie stimuli|chunks).

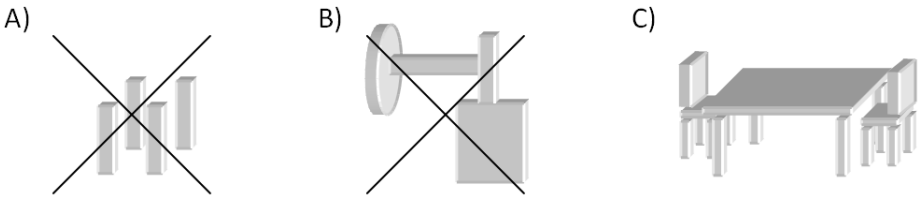


FIGURE 1.2 – Exemple de chunks. Le chunk A n'apporte rien de plus que le chunk « chaise » . Le chunk B n'apporte aucune information susceptible d'aider l'encodage des stimuli. Le chunk C en revanche pourrait permettre une meilleure description de l'environnement extérieur en introduisant la notion de « tablée » .

La difficulté pratique qui se pose et de déterminer à quel moment un chunk doit être créé. Par exemple, le chunk « tablée » de la figure 1.2 est-il assez intéressant pour être créé ? Une création trop lente des chunks risque de conduire à une représentation sous-optimale des stimuli. À l'inverse, une création trop rapide peut donner lieu à l'apparition de chunks qui ne seront pas réutilisés par la suite. Il s'agit du problème classique en sélection de modèles connu sous le nom de « dilemme biais-variance » .

1.8 Dilemme biais-variance

Le dilemme biais-variance permet de mettre un nom sur l'idée qu'un modèle, s'il est suffisamment complexe, c'est-à-dire qu'il comporte suffisamment de paramètres pouvant être ajustés a posteriori, est alors en mesure d'expliquer n'importe quel jeu de données. Si l'on prend l'exemple classique d'un modèle polynomial de degré n, on sait qu'il pourra rendre compte de n'importe quel ensemble comportant jusqu'à $n+1$ points. Chacune des données est utilisée pour fixer un des paramètres libres (ou degrés de liberté) du modèle.

Plus un modèle est complexe, plus il est susceptible d'expliquer finement les données (faible biais), mais plus il est sensible aux spécificités du jeu de données (forte variance). Dans notre exemple, il facile de voir que sa complexité augmente avec le degré du polynôme choisi. Nous verrons que caractériser la complexité d'un modèle dans le cas général n'est pas chose facile. Ce sera l'objet du chapitre 2.

Lorsqu'un modèle est trop complexe et qu'il ajuste parfaitement les données, sa capacité prédictive est généralement pauvre : c'est le phénomène de surajustement. Un tel modèle capture la structure des données, mais également le bruit qu'elles contiennent, et surtout ajoute de l'information qui n'est pas contenue initialement dans les données. Cette idée sera rendue plus claire au chapitre 3.

La qualité d'un modèle dépend de ses capacités prédictives, c'est pourquoi un modèle est généralement construit en minimisant l'erreur de prédiction (également appelée erreur de généralisation). Il est pour cela nécessaire de limiter artificiellement la complexité du modèle. Selon le type de modèle considéré (polynômes, réseaux de neurones formels, arbres de classification et régression, etc.), il existe des stratégies d'élagages spécifiques. En pratique cela revient généralement à séparer les données en un ensemble d'apprentissage permettant de fixer les paramètres du modèle, et un ensemble de test permettant d'estimer l'erreur de généralisation. L'erreur de généralisation quantifie l'incapacité du modèle à reproduire les données de l'ensemble de test (Fig. 1.3 page ci-contre).

Le paradigme classique de l'apprentissage est le suivant : on observe un jeu de données x généré par un processus S^* inconnu et que l'on souhaite modéliser. Soit parce qu'un a priori est justifié par la connaissance du domaine, soit pour des raisons calculatoires, on se restreint à un ensemble de modèles $\mathcal{S}$ pouvant ou non contenir S^*. Un bon modèle $S \in \mathcal{S}$ pour x est un modèle le plus simple possible pour lequel x est un élément typique. La notion de simplicité sera abordée au chapitre 2 et la notion de typicalité au chapitre 3. Nous verrons alors qu'un critère objectif peut être défini pour mettre en balance simplicité et typicalité, donnant lieu à la création d'une méthode générale appelée MDL (*Minimum Description Length*) permettant résoudre le dilemme biais-variance.

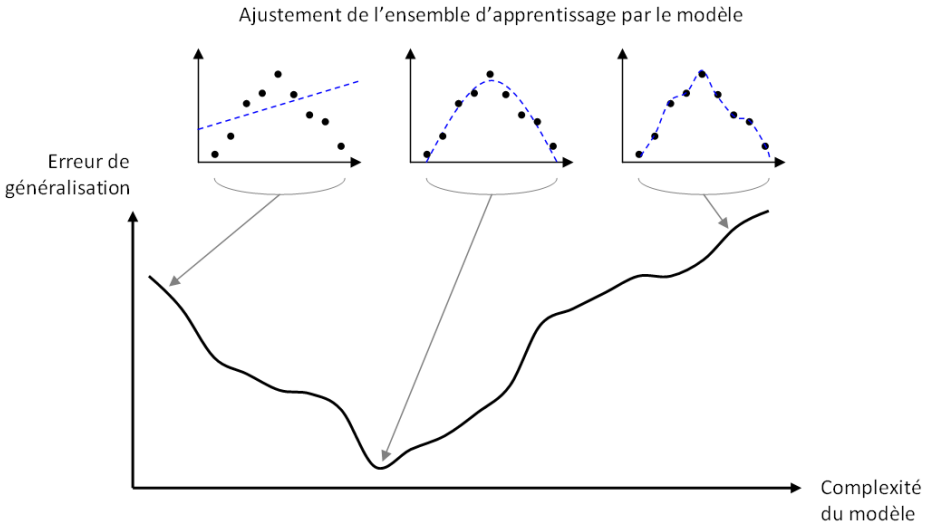


FIGURE 1.3 – Exemple typique d'évolution de l'erreur de généralisation en fonction de la complexité du modèle. Le modèle considéré est polynomial.

1.9 Conclusion

Dans ce chapitre, nous avons essayé de montrer la plausibilité cognitive du mécanisme de *chunking* et du principe de simplicité qui vont constituer le coeur du modèle qui sera décrit dans la suite de cette thèse. Avant d'aborder les aspects plus pratiques concernant le modèle, nous allons consacrer les deux prochains chapitres à décrire le cadre théorique dans lequel il s'inscrit. Il s'agit d'évaluer l'intérêt théorique du principe de simplicité (chapitre suivant) et de montrer quels types d'approximations peuvent être faites dans le but d'appliquer ce principe théorique à la modélisation cognitive.

Nous allons voir que les outils théoriques permettant de définir rigoureusement le principe de simplicité apportent également une meilleure compréhension du dilemme biais-variance, ainsi qu'une ébauche de solution. Les deux chapitres suivants, bien qu'ils correspondent à deux domaines différents, forment un tout qui va nous permettre d'aborder la question fondamentale de l'apprentissage : « quel-est le meilleur modèle qui puisse être créé pour expliquer un jeu de données ? » . La réponse théorique apportée à cette question est relative à l'apprentissage en général, mais s'applique parfaitement au cadre de la modélisation cognitive.

Chapitre 2

Complexité algorithmique

Sommaire

We discuss views about whether the universe can be rationnally comprehended [...], we defend the thesis that comprehension is compression, i.e., explaining many facts using few theoretical assumptions, and that a theory may be viewed as a computer program for calculating observations.

G. Chaitin

2.1 Introduction

Le concept de complexité algorithmique, également appelée « complexité de Kolmogorov » ou « entropie algorithmique » est défini dans le cadre de la « théorie algorithmique de l'information » . Cette dernière étend la « théorie de l'information » de Shannon (1948) en remplaçant la notion de variable aléatoire par celle plus générale d'algorithme. Elle ne doit pas être confondue avec la théorie de la complexité qui a trait à l'évaluation du niveau de complexité des algorithmes.

La présentation qui est faite dans ce chapitre a une double utilité. D'une part fournir une réponse théorique à la question fondamentale de l'apprentissage : quel est le meilleur modèle qui peut être créé pour expliquer un jeu de données ? En cela, ce chapitre étend les notions esquissées dans le précédent, mais dans un cadre plus général. D'autre part, il permet d'introduire le concept de simplicité et fournit une transition naturelle vers la notion de *Minimum Description Length* (MDL) qui sera développée au chapitre suivant. Nous verrons alors que le principe du MDL basé sur la théorie de l'information, peut être vu comme une approximation calculable de la complexité algorithmique d'un ensemble de données (Leung-Yan-Cheong & Cover, 1978).

L'objet de ce chapitre n'est pas de fournir une présentation formelle de la complexité algorithmique, mais d'en donner une intuition dans le but de mieux comprendre les limites inhérentes à toute modélisation. Le lecteur intéressé par le sujet pourra se reporter à l'ouvrage (Li & Vitanyi, 1997) qui fait office de référence en la matière.

2.1.1 Exemple introductif

Présentons à travers un exemple les notions de « complexité » , « régularité » et de « quantité d'information » , qui se rapportent en réalité au même concept (défini rigoureusement section 2.1.3). Prenons trois suites binaires de complexité croissante :

$$1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1 \tag{2.1}$$

$$1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0 \tag{2.2}$$

$$1,1,0,1,0,1,1,1,0,1,0,0,1,1,0,1,0,1,0,1,1,0,0,0,1,0,1,1,1,0 \tag{2.3}$$

La suite 2.1 est la moins complexe : elle possède une régularité cyclique de période 1. C'est donc celle qui véhicule la plus petite quantité d'information. Si l'on souhaite connaître la valeur d'un bit choisi au hasard dans la suite, aucune information additionnelle n'est nécessaire : il s'agit forcément d'un 1. Dans le cas de la suite 2.2 de période 2, la connaissance de la parité de l'indice est nécessaire pour déterminer la valeur du bit choisi. Cette suite contient une plus grande quantité d'information que la précédente. Dans le cas de la suite 2.3

générée aléatoirement, la période peut être vue comme infinie. Chaque bit est maximalement informatif, et la connaissance précise de la position du bit dans la suite est nécessaire pour en déduire sa valeur.

Régularités et quantité d'information : L'idée sous-jacente est que plus une suite possède de régularités, moins la quantité d'information véhiculée est importante et plus il est possible de la compresser. Ainsi, les 30 bits de la suite 2.1 peuvent être résumés par « 30 fois 1 » (9 caractères). La suite 2.2 légèrement plus complexe peut être résumée par « 15 fois 10 » (10 caractères) [1]. En revanche, la suite 2.3 qui ne comporte aucune régularité ne peut être résumée en quelque chose de plus court qu'elle-même : « 1 fois 11010111010011010101100010111 0 » ou « 110101110100110101011000101110 » (30 caractères). Cette incompressibilité est justement signe que la suite est aléatoire : chaque bit est imprévisible et l'information apporté par chacun ne peut se déduire des autres. En résumé, toute régularité peut être utilisée pour comprimer une suite et réciproquement toute compression dénote l'existence de régularités. Il y a équivalence entre les deux termes.

Nécessité d'un langage de description général : Comprimer une suite en utilisant comme ci-dessus le langage naturel (« n fois y ») est sous-optimal et ne permet pas d'obtenir la description la plus courte puisque la langue est elle-même fortement redondante. Le langage de description optimal (le plus court) se doit d'être exempt de toute redondance. Dans l'exemple, nous nous sommes restreint pour plus de simplicité aux régularités de type « cycle » . Mais le langage choisi doit être suffisamment expressif pour pouvoir décrire tous les types de régularités. Nous verrons au chapitre suivant que la théorie de l'information basée sur la notion de variable aléatoire permet de compresser les régularités fréquentielles. Le cadre sur lequel s'appuie la complexité algorithmique est celui plus général des machines de Turing, capables de capturer toutes les régularités pouvant être décrites par une procédure mécanique déterministe. Une telle procédure est appelée algorithme.

2.1.2 Complexité algorithmique et simplicité

Ce chapitre présente une méthode purement théorique permettant de transformer n'importe quelle suite binaire en sa description la plus courte. La complexité algorithmique a été développé indépendamment et pour des raisons différentes par Chaitin (1966), Kolmogorov (1968) et Solomonoff (1960). Cette notion a notamment été utilisée pour poser les bases d'une théorie générale de l'apprentissage inductif pouvant servir dans le cadre de la création de nouvelles théories scientifiques (Le terme « théorie » est ici employé dans le sens de « modèle

1. D'une façon rigoureuse et en utilisant une notation binaire dans la forme « n fois y » , le nombre de bits nécessaires pour coder n croit logarithmiquement avec le nombre de cycles tandis que le nombre de bits nécessaires pour coder y croit linéairement avec la longueur de chaque cycle. La représentation la plus courte en nombre de bits est celle qui favorise un maximum de cycles courts.

d'un phénomène observé »). L'approche que nous adopterons dans ce chapitre est proche de celle suivie par Solomonoff (1964). Elle peut se résumer de la façon suivante :

Tout phénomène perçu peut être représenté sous forme d'une suite binaire de longueur finie : c'est notamment le cas des observations scientifiques [2]. Créer un modèle expliquant un phénomène donné, revient à trouver une méthode capable d'une part de reproduire le phénomène observé et d'autre part de prédire, c'est-à-dire d'extrapoler la suite binaire par son prolongement le plus probable. Parmi tous les modèles reproduisant le phénomène observé, le plus court est celui qui contient uniquement la quantité d'information apportée par les données sans ajout d'information supplémentaire. C'est donc celui qui est le plus à même d'effectuer la meilleure prédiction des données futures (discuté ci-après).

Pour cela, la complexité algorithmique se place dans le cadre très général des machines de Turing où tout modèle peut être vue comme un programme. La complexité d'un modèle est alors donnée par la longueur de son programme.

Un programme est une suite binaire au même titre que les données observées. La différence de longueur entre les données et le programme représente la compression apportée par le modèle. Un modèle est d'autant plus explicatif que cette compression est grande. À l'extrême, la longueur du modèle le plus court représente la quantité d'information contenue dans les données après suppression de toute redondance. Cette formulation généralise l'idée très ancienne du rasoir d'Occam.

Rasoir d'Occam

Le rasoir d'Occam ou principe de parcimonie date du 14e siècle et aurait été formulé par le moine franciscain Guillaume d'Occam :

> *Entia on sunt multiplicanda praeter necessitatem.*
> Les entités ne doivent pas être multipliées au-delà de la nécessité.

Ce principe a largement été utilisé et réinventé par la suite. Notamment par Newton en 1687 dans *Philosophiae Naturalis Principia Mathematica* sous la forme :

> *Nous n'avons à accepter pas plus de causes des choses naturelles que celles qui sont à la fois vraies et suffisantes pour expliquer ces choses.*

Il a également été énoncé sous le nom de principe d'économie par Ernst Mach qui s'est intéressé au sens des lois en physique :

> *En réalité, une loi contient moins que le fait lui-même, parce qu'elle ne reproduit pas le fait dans son ensemble mais seulement dans son aspect qui est le plus important à nos yeux, le reste étant ignoré intentionnellement ou par nécessité.*

2. De façon rigoureuse, un phénomène continu ne peut être mis sous cette forme qu'après avoir été discrétisé avec une précision finie. Les mesures physiques effectuées à l'aide d'instruments numériques entrent dans cette catégorie.

Mach s'est également intéressé à la plausibilité cognitive du principe d'économie :

> *Quand l'esprit humain, avec son pouvoir limité, tente de reproduire en lui-même la riche vie du monde, dont il est lui-même une petite partie, et dont il ne peut jamais espérer s'extraire, il a toutes les raisons de procéder économiquement.*

Une formulation plus utile en termes de construction de théories scientifiques (modèles) est de considérer deux modèles A et B concurrents. Si A et B sont également prédictifs, alors le meilleur serait le plus simple des deux. L'application de ce principe se trouve limité par le fait qu'il ne fournit pas de méthode permettant de mesurer la complexité d'un modèle donné. Il n'est applicable que dans le cas ou l'un des modèles est inclus dans l'autre, en évitant l'ajout d'hypothèses *ad hoc* inutiles. Cette mesure est donc très limitée car l'inclusion est une relation d'ordre partielle sur l'ensemble des modèles.

En fournissant une méthode générale permettant de mesurer la complexité d'un modèle, la complexité algorithmique étend ce principe à tous les modèles pouvant être décrits par un algorithme (programme).

Quelques exemples

Un phénomène se trouve d'autant mieux expliqué qu'il peut être réduit à une description courte sans perte d'information. À l'extrême, un phénomène aléatoire ne peut être substantiellement compressé et donc expliqué (Fig. 2.1 à 2.4).

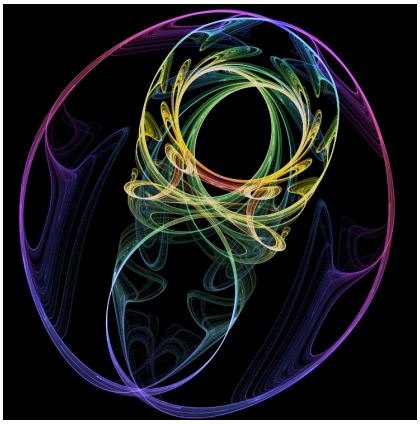


FIGURE 2.1 – Image d'une fractale. Une description brute de l'image nécessite plus de 10 millions de bits. Une description capable de prendre en compte la structure de l'image (équation de la fractale) serait beaucoup plus courte.

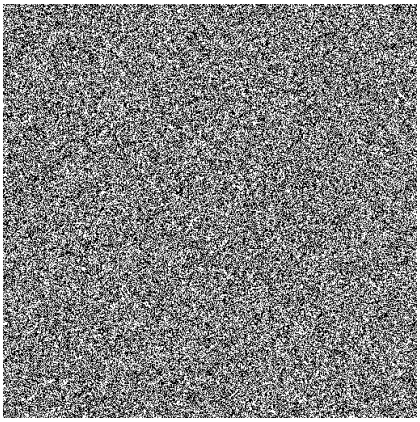


FIGURE 2.2 – Image de bits aléatoires. L'absence de structure dans l'image rend toute compression impossible.

Nos capacités à modéliser des phénomènes sont souvent testées à l'aide de suites de nombres à extrapoler (tests de QI). La prédiction jugée correcte est alors celle

1, 1, 2, 3, 5, 8, 13, 21, 34, 55

FIGURE 2.3 – Séquence de 10 nombres véhiculant la même quantité d'information que la description $x_{i+1} = x_i + x_{i-1}$ avec $x_1 = x_2 = 1$.

1, 5, 20, 4, 61, 8, 3, 41, 8, 44

FIGURE 2.4 – Séquence incompressible de 10 nombres. La quantité d'information véhiculée ne peut être réduite à moins que ces 10 nombres.

qui correspond à la description la plus simple.

La complexité algorithmique mesure la simplicité d'une description à la quantité d'information dont elle a besoin, c'est-à-dire au nombre de degrés de libertés qu'elle fixe dans l'espace de toutes les descriptions possibles. Une même quantité d'information peut prendre la forme de diverses descriptions. Par exemple, la suite

$$2, 3, 5, 9, 17, 33, 65, 129 \tag{2.4}$$

est compatible avec les trois descriptions suivantes

$$x_{i+1} = (3 - i) + \sum_{k=1}^{i} x_k \tag{2.5}$$

$$x_{i+1} = 2 \times x_i - 1 \tag{2.6}$$

et

$$x_{i+1} = 2^i + 1 \tag{2.7}$$

qui, bien que différentes en apparence, sont formellement équivalentes [3] et conduisent toutes aux mêmes prédictions $257, 513, 1025, \ldots$.

En revanche, la description suivante

$$x_{i+1} = \begin{cases} 2^i + 1 & \text{si } i < 9 \\ 0 & \text{sinon} \end{cases} \tag{2.8}$$

correspond à 2.7 à laquelle s'ajoute l'information $x_i = 0$ si $x \geq 9$. Cette description est elle aussi compatible avec la suite 2.4, mais l'ajout d'information n'étant pas motivé par les données, les prédictions effectuées sont moins probables que celles de 2.7. Elles sont conditionnelles à la justesse de l'hypothèse supplémentaire qui n'est pas justifiée par les données.

L'explication la plus courte est celle qui est la plus à même d'effectuer les meilleures prédictions car c'est celle qui fait les hypothèses minimales sur le phénomène à modéliser. En évitant l'introduction d'information superflue, l'explication la plus courte évite le surajustement (cf. section 1.8). Le phénomène de surajustement peut être vu comme l'ajout a priori d'une certaine quantité

3. Des modèles en apparence différents, s'ils conduisent exactement aux mêmes prédictions, véhiculent en réalité la même information.

d'information qui se révèle être incorrecte. Cet ajout d'information se fait généralement de manière implicite et non délibérée. Par exemple, la suite

$$1, 1, 2, 3, 5, 8, 13 \tag{2.9}$$

supporte les deux descriptions

$$x_{i+1} = x_i + x_{i-1} \tag{2.10}$$

et

$$x_i = \frac{1}{144}i^6 - \frac{41}{240}i^5 + \frac{241}{144}i^4 - \frac{395}{48}i^3 + \frac{1535}{72}i^2 - \frac{133}{5}i + 13 \tag{2.11}$$

qui conduisent à des prédictions différentes : $21, 34, \ldots$ pour 2.10 et $29, 85, \ldots$ pour 2.11. La première plus concise ne capture que la quantité d'information nécessaire. Le seconde en revanche ajoute implicitement de l'information non-contenue dans les données. Il était facile de voir que 2.7 $\subset$ 2.8 donc que 2.8 contenait nécessairement une plus grande quantité d'information que 2.7. Il est plus difficile de voir que 2.11 contient une plus grande quantité d'information que 2.10. Il s'agit d'un exemple où il est nécessaire de pouvoir évaluer la complexité de deux explications n'étant pas liées par une relation d'inclusion.

2.1.3 Définition de la complexité algorithmique

Cette introduction a permis de présenter la notion de quantité d'information (parfois abusivement appelée « information »), qui correspond à ce qu'il y a d'irréductible dans une suite x. Nous allons voir qu'elle peut se mesurer par la longueur de la plus courte description de x, encore appelée complexité algorithmique de x.

La notion vague de description est maintenant remplacée par la notion plus rigoureuse d'algorithme. Un algorithme est une suite d'instructions explicites pouvant être suivies pas à pas de façon non-ambiguë, et conduisant toujours au même résultat. Un algorithme doit être autosuffisant [4], c'est-à-dire qu'il contient en lui-même toute l'information nécessaire à son exécution et ne dépend pas des connaissances de la personne ou de la machine qui l'exécute. On emploiera indifféremment les termes « algorithme » et « programme » .

La complexité algorithmique fournit une définition absolue et objective de la quantité d'information véhiculée par une suite binaire x de longueur finie. Elle est absolue dans la mesure où elle se base sur l'utilisation d'une machine de Turing capable de compresser tout type de régularité. Elle est objective car elle ne fait aucune d'hypothèse sur la nature du processus ayant engendré x. La complexité d'une suite ne dépend que d'elle-même. Elle se définit de la façon suivante :

4. Un algorithme pour une machine de Turing universelle contient à la fois la table d'actions définissant le comportement de la machine, et l'algorithme proprement dit. Pour plus d'information se reporter à l'annexe A.1.1 page 2

La complexité algorithmique $K(x)$ d'une suite binaire x est la longueur du plus court programme autodélimitant p qui génère x sur une machine de Turing universelle U, puis s'arrête.

$$K(x) = \min_{p \in PA} \{|p| : U(p) = x\} \tag{2.12}$$

où $|p|$ est la longueur du programme p et PA l'ensemble des programmes autodélimitants.

Une formulation équivalente est :

$$K(x) = log_2 \left(\frac{1}{\sum_{p \in PA : U(p)=x} 2^{-|p|}} \right) \tag{2.13}$$

La formulation 2.13 est très intéressante car elle apporte une interprétation différente de la complexité, basée sur la notion de séparation. Elle permet d'établir le parallèle avec la théorie de l'information en faisant apparaître plus explicitement la probabilité de la suite x. La section suivante explique le cheminement permettant d'aboutir à la définition 2.13, et fournit une intuition de l'équivalence entre les formulations 2.12 et 2.13.

2.2 Présentation de la complexité algorithmique

Pour ne pas égarer l'attention du lecteur et dans la mesure où elles ne sont pas indispensables à la compréhension, les définitions formelles sont reléguées en annexe A.1.1. Le lecteur peut se représenter une machine de Turing universelle comme un ordinateur capable d'exécuter une suite d'instructions écrite dans un langage donné (C++, PASCAL, LISP, JAVA, etc.).

La complexité algorithmique d'une suite x représente la plus petite quantité d'information à mettre en oeuvre pour recréer x. C'est ce qu'il y a d'irréductible dans x, le reste pouvant être déduit par une procédure mécanique à partir de cette partie irréductible. La partie contenant l'information irréductible est appelée « programme » . La procédure mécanique déterministe permettant de recréer x à partir du programme est une « machine de Turing universelle » [5].

2.2.1 Mesure objective de la complexité

Le but est de définir une mesure de la complexité qui soit objective, c'est-à-dire qui dépende uniquement de la suite x sans faire d'hypothèses a priori sur le type de processus ayant pu engendrer x.

5. Une machine de Turing universelle est une machine abstraite permettant de réaliser n'importe quelle procédure mécanique (algorithme) (annexe A.1.1).

Ainsi la suite 2.1 page 16 est très régulière et possède une complexité faible. Cette complexité est indépendante du processus ayant généré la suite. Bien que cela soit improbable ($P = 2^{-30}$), il pourrait s'agir du résultat d'un tirage aléatoire. De même, la suite 2.3 page 16 qui ne peut être substantiellement réduite à moins qu'une trentaine de bits, pourrait avoir été générée par un phénomène non-aléatoire pour peu que sa complexité soit supérieure à 30 bits.

2.2.2 Cadre de travail

On se place dans l'espace X des suites binaires x de longueur finie :

$$x \in \{0,1\}^* \tag{2.14}$$

Se restreindre à l'utilisation de suites binaires n'a pas d'incidence sur la généralité de l'approche, toute donnée discrète pouvant être mise sous cette forme par simple traduction de n'importe quel alphabet n-aire en binaire. Comme nous allons le voir, la difficulté majeure provient du fait que cet espace contient un nombre infini d'éléments.

2.2.3 Caractériser c'est séparer

Caractériser une suite c'est fournir la quantité minimale d'information permettant de la séparer des autres. Afin de mieux comprendre cette idée fondamentale, nous effectuons le parallèle avec la théorie de l'information.

Notion de taille de codage

La théorie de l'information (Shannon, 1948) fournit un cadre permettant de déterminer la quantité d'information véhiculée par un objet x lorsque sa probabilité $P(x)$ est connue. x est supposé être la réalisation d'une variable aléatoire discrète de distribution $P(.)$. La quantité d'information ou taille de codage (notée $C(x)$) de x, correspond à la quantité minimale d'information nécessaire pour représenter x sous une forme tenant compte des régularités fréquentielles.

Dans le cas où cette taille de codage est exprimée en bits, elle représente le nombre minimal de questions binaires qu'il faut poser pour caractériser l'objet, c'est-à-dire le séparer des autres. Dans un espace contenant deux objet $\{x_1 x_2\}$ équiprobables, une question (1 bit) suffit pour séparer un objet de l'autre. Si un espace contient quatre objets équiprobables $\{x_1 x_2 x_3 x_4\}$, deux questions (2 bits) sont nécessaires pour isoler l'un d'eux (Fig. 2.5 page suivante).

La taille de codage de Shannon-Fano $C(x)$ d'un objet x dépend uniquement de sa probabilité $P(x)$, selon la relation :

$$C(x) = log_2\left(\frac{1}{P(x)}\right) \tag{2.15}$$

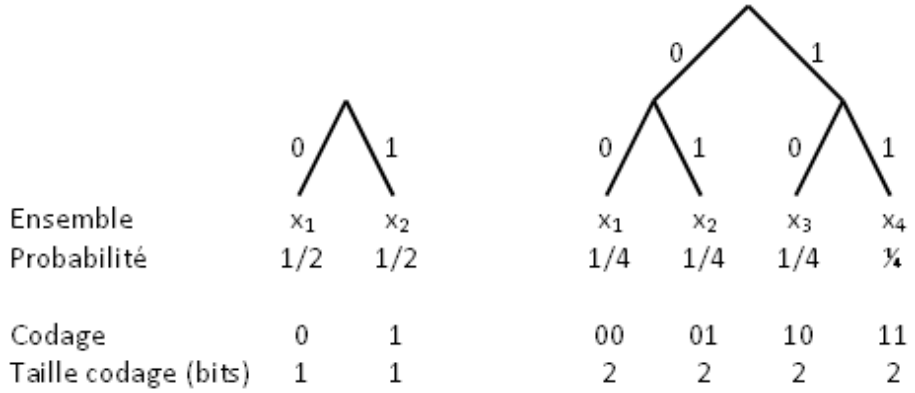


FIGURE 2.5 – Quantité d'information nécessaire (en bits) pour séparer soit 2 soit 4 objets équiprobables. Chaque noeud de l'arbre représente une question binaire et les branches qui en sont issues représentent les deux réponses possibles.

$C(x)$ est également appelée information propre de x et s'exprime dans la base du logarithme (ici en bits). Il n'est pas toujours possible d'associer à x un codage de taille optimale $C(x)$, mais il est possible de l'approcher par des algorithmes de recherche de codes optimaux (Huffman, 1952).

En supprimant l'équiprobabilité des x_i, par exemple ($P(x_1) = \frac{1}{2}; P(x_2) = \frac{1}{4}; P(x_3) = P(x_4) = \frac{1}{8}$), on obtient les tailles de codages présentées figure 2.6.

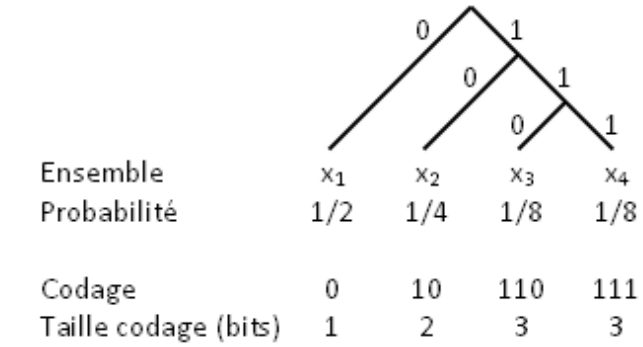


FIGURE 2.6 – Quantité d'information nécessaire (en bits) pour séparer 4 objets non-équiprobables.

La quantité d'information à fournir pour séparer un objet des autres dépend uniquement de sa probabilité. Dans l'espace X des suites binaires finies, si l'on pouvait associer une probabilité $P(x)$ à chaque élément x, alors $log_2\left(\frac{1}{P(x)}\right)$ fournirait une mesure de la complexité de x. C'est la quantité d'information nécessaire pour caractériser x, c'est-à-dire pour le séparer des autres éléments de X.

2.2.4 Absence d'hypothèses a priori

Le cadre de la théorie de l'information présenté ci-dessus est trop restrictif car il ne s'applique pas à un objet seul, mais à un objet en relation avec une liste d'alternatives. Connaître la probabilité d'une suite x nécessite la connaissance de toutes les suites de l'espace X. On veut une mesure de complexité qui soit objective et ne fasse pas d'hypothèses a priori sur la structure de l'espace. Toutes les suites binaires finies doivent être considérées comme également différentes. Il s'agit d'une formulation similaire au principe d'indifférence en probabilité, qui veut qu'en l'absence d'information sur la probabilité de plusieurs évènements il faille les considérer comme équiprobables. Si toutes les suites x sont équiprobables (également informatives), leur nombre étant infini, il n'est pas possible de définir de mesure de probabilité sur X : on aboutit à une impasse [6]. La solution à ce problème va consister à changer d'espace.

2.2.5 Espace des programmes

La complexité algorithmique va permettre d'associer implicitement une probabilité à chaque suite x de l'espace X. L'idée introduite par Solomonoff (1960) est qu'une suite binaire finie est d'autant plus probable qu'il existe de programmes capables de la générer sur une machine de Turing universelle. Un programme est également une suite binaire finie, mais représentant une succession d'instructions [7]. p est un programme pour x sur la machine U si :

$$U(p) = x \tag{2.16}$$

Il est en théorie possible de lister tous les programmes existants. Si on exécutait en parallèle tous ces programmes, certains ne s'arrêteraient jamais (problème de l'arrêt détaillé section 2.3.4 page 31), d'autres encore ne produiraient aucune sortie. Néanmoins toutes [8] les suites binaires possibles seraient produites. La subtilité de l'approche tient au fait que plus une suite est simple, plus il y a de programmes capable de la générer et plus ces programmes sont courts.

À suite simple, programme probable

Plus une suite est simple, plus elle contient de régularités pouvant être factorisées (comprimées). On comprend aisément comment une régularité de type « cycle » peut se factoriser :

$$1,0,0,1,0,0,1,0,0,1,0,0,1,0,0 \longrightarrow 5 \text{ fois } 100 \tag{2.17}$$

6. Il n'est pas non plus possible d'approcher la probabilité de x à partir de sa fréquence empirique puisque l'on dispose uniquement de x.

7. Quelles que soient les instructions machine de base, un programme peut toujours être mis sous forme d'une suite binaire.

8. En toute rigueur, chaque suite binaire est même produite par un nombre infini de programmes, comme cela sera expliqué dans la suite.

mais cela reste vrai pour des régularités plus complexes comme :

$$1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1 \longrightarrow \text{15 premiers bits de Pi} \tag{2.18}$$

La généralité de l'approche tient au fait que toutes les régularités pouvant être capturées par une procédure mécanique (algorithme) peuvent l'être par une machine de Turing universelle.

Plus une suite contient de régularités, plus il existe de façons de la compresser donc plus il existe de programmes dont elle est la sortie. Essayons d'en donner une idée intuitive à l'aide d'un exemple [9] (Fig. 2.7) qui se limite aux programmes de longueur inférieure à 38 caractères.

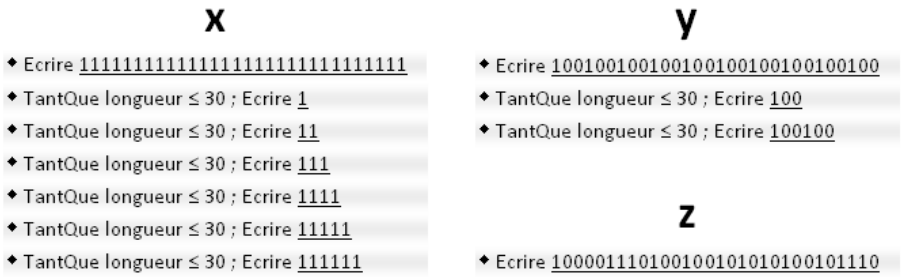


FIGURE 2.7 – Un seul programme de longueur inférieure à 38 caractères est capable de produire la suite z = 10000111010010010101010010110 dans laquelle aucune régularité ne peut être extraite. En revanche 3 programmes sont capables de produire en sortie la suite y = 10010010010010010010010010010 moyennement régulière, et 7 pour la suite x = 11111111111111111111111111111 très régulière.

En définissant une mesure de probabilité sur l'espace des programmes il va être possible d'induire une mesure de probabilité dans l'espace X des suite binaires finies.

Mesure de probabilité sur l'espace des programmes

L'espace des programmes est lui aussi un ensemble infini de suite binaires finies, mais sur lequel le principe d'indifférence prend une forme particulière : la probabilité d'un programme p ne dépend que de sa longueur. Une mesure de probabilité a priori sur l'espace des programmes a été proposée par Solomonoff (1960). Elle permet d'associer une probabilité plus élevée aux programmes les plus courts :

$$P(p) = 2^{-|p|} \tag{2.19}$$

où $|p|$ est la longueur (en bits) du programme p.

9. Il faut voir cet exemple comme purement illustratif. L'alphabet choisi n'est pas binaire, les fonctions ne sont pas définies en terme d'instructions machine, etc.

Pour que cette probabilité soit définie, il est nécessaire d'ajouter une contrainte supplémentaire : les programmes doivent être autodélimitants. Sous cette condition essentielle (détaillée annexe A.1.2), on a

$$\sum_{p \in PA} 2^{-|p|} = 1 \tag{2.20}$$

où PA est l'ensemble des programmes autodélimitants.

La probabilité d'une suite $x \in X$ peut donc être définie comme la probabilité qu'un programme p choisi aléatoirement selon 2.19 produise x sur une machine de Turing universelle U :

$$P(x) = \sum_{p \in PA : U(p) = x} 2^{-|p|} \tag{2.21}$$

Cette mesure de probabilité sur l'espace X, induite par la mesure de probabilité sur PA porte le nom de probabilité algorithmique (Solomonoff, 1960).

Comme expliqué précédemment, la complexité algorithmique $K(x)$ de x se déduit alors naturellement de la probabilité algorithmique $P(x)$ d'après

$$K(x) = log_2 \left(\frac{1}{P(x)} \right) \tag{2.22}$$

Pour résumer, on passe de l'espace contenant toutes les suites binaires finies X sur lequel on ne peut pas définir de mesure de probabilité, à l'espace des programmes autodélimitants PA sur lequel cela devient possible moyennant l'hypothèse que la probabilité d'un programme dépend uniquement de sa longueur.

2.2.6 Programme le plus court ?

Telle qu'elle est définie ci-dessus, la complexité algorithmique $K(x)$ représente la plus petite quantité d'information nécessaire pour séparer les programmes générant x des autres. C'est-à-dire l'information minimale à fournir pour caractériser un programme capable de produire x (formulation 2.13).

La formulation 2.12 plus classique est de voir $K(x)$ comme étant la taille du plus court programme permettant de générer x. Une formulation équivalente est de voir $K(x)$ comme le nombre minimal de degrés de libertés (binaires) à fixer dans l'espace des programmes pour être assuré d'obtenir [10] un programme générant x. Toujours de manière informelle, la figure 2.8 page suivante représente sous forme d'expressions régulières la forme que pourraient prendre le plus petit jeu de contraintes d'un programme produisant les suites x et y de la figure 2.7.

L'équivalence entre la taille du plus court programme (2.12) et le logarithme de l'inverse de la probabilité algorithmique (2.13) est appelée *Coding Theorem* (Levin, 1974). Cette équivalence est vraie à une constante additive près.

10. Cela revient à dire qu'en explorant des degrés de libertés laissés libres on obtient tous les programmes générant x et uniquement ceux-là.

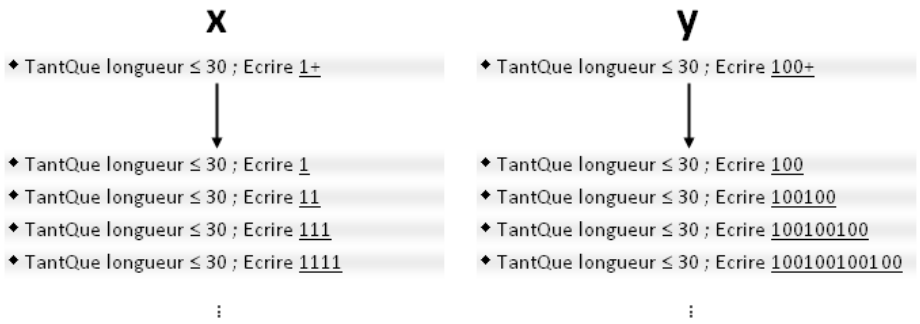


FIGURE 2.8 – Le symbole + des expressions régulières signifie « une fois ou plus » et s'applique aux caractères situés avants. En explorant les degrés de libertés laissés libres par l'expression régulière, on obtient les programmes situés dessous. Leur nombre est infini.

2.2.7 Résumé et hypothèses

Effectuons un bref résumé de cette section et des hypothèses implicites qui ont été faites.

Résumé

Soit PA l'espace infini des programmes autodélimitants de longueur finie. Les éléments p de PA étant autodélimitants (annexe A.1.2 page 3), une mesure de probabilité P peut être définie sur PA :

$$P(p) = 2^{-|p|} \tag{2.23}$$

Cette mesure de probabilité sur l'espace des programmes permet d'induire une mesure de probabilité sur l'espace des suites binaires finies X :

$$P(x) = \sum_{p \in PA : U(p) = x} 2^{-|p|} \tag{2.24}$$

Il en découle que la complexité algorithmique $K(x)$ d'une suite x est :

$$K(x) = log_2 \left(\frac{1}{P(x)} \right) \tag{2.25}$$

Qui correspond bien à l'équation 2.13 que l'on cherchait à expliquer.

Hypothèses implicites

Les hypothèses faites sont toutes faibles et facilement justifiables, mais il importe néanmoins de les identifier clairement.

– Les suites x de l'espace X sont binaires et de longueur finie. Si leur longueur était infinie, considérer les programmes qui produisent x puis s'arrêtent n'est plus possible. Une généralisation de la complexité algorithmique s'appliquant aux suites de longueur infinie a cependant été proposée par Poland (2004) et Schmidhuber (2002). Nous avons vu que les suites pouvaient être n-aires mais il doit impérativement s'agir de suites discrètes d'éléments discrets issus d'un alphabet fini puisqu'une machine de Turing est un automate à états finis (donc a fortiori discrets) (annexe A.1.1 page 1).
– La seconde hypothèse est le choix effectué (bien que très naturel) pour la distribution de probabilité sur l'espace des programmes autodélimitants.
– La dernière hypothèse est en réalité une conséquence du principe d'indifférence. Il est nécessaire que l'espace X contienne **toutes** les suites binaires de longueur finie. La mesure de complexité n'est donc valable que si l'on ne dispose d'aucune information sur la structure de X.

2.3 Résultats

Maintenant que la notion de complexité algorithmique a été détaillée, nous énonçons brièvement les principaux résultats qui en découlent.

2.3.1 Borne supérieure

La plus courte description $K(x)$ d'une suite x ne peut être substantiellement plus grande que la suite elle-même :

$$\forall x, K(x) \leq |x| + C \tag{2.26}$$

Il est toujours possible de reproduire la suite x à l'identique à l'aide d'un programme de type $ecrire(x)$. La constante C peut alors être vue comme la longueur de la fonction $ecrire()$. À l'extrême lorsque $K(x) \geq |x|$, la suite x est dite aléatoire (au sens de (Martin-Löf, 1966)).

2.3.2 Théorème d'invariance

La généralité de cette approche tient au fait que la complexité algorithmique est indépendante de la machine de Turing considérée. Cette propriété appelée théorème d'invariance (*invariance theorem*) a été montrée indépendamment par (Chaitin, 1966 ; Kolmogorov, 1968 ; Solomonoff, 1964). Si $T1$ et $T2$ sont deux machines de Turing, alors les complexités correspondantes K_{T1}, K_{T2} sont égales à une constante additive prêt, indépendante de x :

$$\exists C_{T1\to T2}, \forall x, K_{T2}(x) = K_{T1}(x) + C_{T1\to T2} \tag{2.27}$$

La constante $C_{T1\to T2}$ dépend uniquement de $T1$ et $T2$. Elle peut être vue comme la taille de l'émulateur permettant d'exprimer les instructions du langage $T1$ dans le langage $T2$.

2.3.3 Incalculabilité

Le résultat le plus important est l'incalculabilité de la complexité algorithmique (Kolmogorov, 1968 ; Zvonkin & Levin, 1970). Il n'existe pas de programme général capable de prendre en entrée une suite x et de renvoyer sa complexité $K(x)$. La méthode naïve qui consiste à générer des programmes de plus en plus grands jusqu'à trouver le premier produisant la suite x, ne permet pas d'aller au-delà du premier programme qui boucle indéfiniment. L'incapacité de prédire si un programme s'arrêtera ou non rend cette solution impossible (problème de l'arrêt détaillé ci-après). La complexité algorithmique peut être réduite au problème de l'arrêt : les deux problèmes sont dit « Turing-équivalents » .

L'incalculabilité de la complexité algorithmique peut être montrée par l'absurde. Supposons l'existence d'une fonction $K(x)$ retournant la complexité de toute suite x placée en argument. Ce programme a une longueur $|K|$ fixe. Le programme $genererChaineDeComplexite(n)$ (Algo.1) permet de générer une suite de complexité au moins n. Sa longueur $|genererChaineDeComplexite()|$ est fixe et celle de sont argument est $|n| = log_2(n)$.

Avec une quantité d'information totale $|K|+|genererChaineDeComplexite|+|n|$ croissant en $O(log_2(n))$, il est possible de générer une suite de complexité n. Pour n suffisamment grand on obtient une contradiction : il est possible de générer une suite de complexité supérieure au programme l'ayant générée.

Algorithme 1

```
FONCTION genererChaineDeComplexite(n)
1: pour tout x ∈ {0,1}^n faire
2:    si K(x) ≥ n alors
3:       retourner x
4:    finsi
5: fin pour
```

Cette incalculabilité peut se comprendre de façon intuitive. Elle fait partie de la définition même de K. Comme un programme de taille minimale est par définition incompressible, chaque bit est maximalement imprévisible et il ne peut y avoir de méthode générale « intelligente » de recherche du programme le plus court. La seule exploration possible est une exploration exhaustive de l'ensemble des programmes. Ce dernier étant infini, l'incalculabilité de K est inévitable.

2.3.4 Problème de l'arrêt

Soit un programme $prgm$ prenant en argument une suite x. Le problème de l'arrêt consiste à déterminer si $prgm(x)$ s'arrêtera ou s'il bouclera indéfiniment. De nombreux problèmes se ramènent au problème de l'arrêt (calcul de K, problème du castor affairé, etc.). L'indécidabilité du problème de l'arrêt provient du fait qu'il n'existe pas de procédure générale permettant de prédire le résultat de l'exécution d'un programme (notamment s'il s'arrête) sans l'exécuter [11].

L'indécidabilité du problème de l'arrêt peut être montrée par l'absurde. Supposons l'existence d'un programme $arret(prgm, x)$ qui renvoie 1 si le programme $prgm(x)$ s'arrête et renvoie 0 sinon. Soit le programme *probleme* (algorithme 2) prenant en argument un programme x :

Algorithme 2

```
FONCTION probleme(x)
1: si arret(x, x) = 1 alors
2:     bouclage
3: sinon
4:     arret
5: finsi
```

Dans le cas où $x(x)$ s'arrête *probleme* boucle indéfiniment, et inversement il s'arrête si $x(x)$ boucle indéfiniment. Une incohérence émerge lorsque l'on considère *probleme*(*probleme*). Si *probleme*(*probleme*) s'arrête, alors par définition (algorithme 2) *probleme*(*probleme*) ne s'arrête pas. Inversement, si *probleme*(*probleme*) ne s'arrête pas, alors il s'arrête (algorithme 2). Supposer l'existence du programme *arret* conduit donc à une incohérence.

2.4 Discussion

L'incalculabilité de la complexité algorithmique en fait un cadre essentiellement théorique. Ce dernier, à la fois général et objectif est néanmoins indispensable pour comprendre les limites de la modélisation.

Généralité

Les situations triviales dans lesquelles deux modèles sont liés par une relation d'inclusion (polynômes de degrés n $\subset$ polynômes de degrés n+1) sont rares. Dans tous les autres cas il n'existe aucune mesure naturelle de la complexité relative de deux modèles concurrents. La généralité de l'approche tient au fait qu'elle s'applique à n'importe quelle suite binaire de longueur finie, donc à n'importe

11. La solution triviale consistant à exécuter $prgm(x)$ puis à renvoyer 1 s'il s'arrête ne fonctionne pas dans le cas d'un bouclage infini. Un programme qui boucle ne peut renvoyer 0 et il n'est pas possible d'utiliser le nombre de caractères écrits comme détection d'un bouclage infini : c'est le problème du castor affairé (Rado, 1962).

quel modèle, et qu'elle se base sur l'utilisation d'une machine de Turing universelle capable de reproduire toute procédure mécanique déterministe, donc capable de capturer tous types de régularités.

Appliquée à une suite binaire x représentant un modèle, la longueur $|p|$ du plus court programme générant x mesure sa complexité, permettant ainsi de trancher entre deux modèles concurrents. Appliquée à une suite binaire x représentant un phénomène observé, le plus court programme p générant x correspond au meilleur modèle pouvant être construit pour décrire ce phénomène. En ce sens, la complexité algorithmique fournit une réponse théorique à la question « quel est le meilleur modèle pouvant être construit pour expliquer un jeu de données ? » (Chaitin, 1987).

Objectivité

La mesure ainsi construite est objective car elle s'applique à toute suite binaire isolée sans formuler d'hypothèse sur la source dont elle est issue ni sur les autres suites de l'espace.

Simplicité et prédiction

Ce chapitre a permis de discuter de l'objectivité et de l'universalité de la complexité algorithmique comme mesure de la simplicité d'un modèle. Cependant les liens entre simplicité et prédiction n'ont pas été discutés. L'hypothèse de base généralement admise est que le modèle le plus simple d'un jeu de données est également le meilleur prédicteur possible en l'absence d'information extérieure (Grunwald, Myung, & Pitt, 2005 ; Rissanen, 1978). C'est l'idée véhiculée par le rasoir d'Occam, par le principe d'indifférence de Laplace, par le maximum d'entropie (Jaynes, 1982) ou encore par le MDL (chapitre suivant). Une justification plus formelle de cette idée intuitive peut être trouvée dans Li et Vitanyi (1997) et Vereshchagin et Vitanyi (2002). Nous verrons également au chapitre suivant que minimiser la taille de codage du modèle est équivalent à maximiser la probabilité du modèle sachant les données.

La notion de simplicité n'est pas intéressante en soi mais seulement dans la mesure où le modèle le plus simple est celui qui a la plus grande probabilité d'être le plus prédictif. La section suivante explique comment la simplicité permet d'apporter (au moins partiellement) une solution au dilemme biais-variance en éliminant une cause possible de variance : le surajustement. L'autre cause de variance (le surapprentissage) sera discutée au chapitre suivant.

Simplicité et dilemme biais-variance

Les modèles « tout est vrai » ($x \in \{0,1\}^*$) et « uniquement ce jeu de données est vrai » ($x \in \{x\}$) sont tous deux compatibles avec n'importe quel jeu de données. L'un est exagérément général et ne comprime pas les données (fort

biais), l'autre est exagérément spécifique et n'est en mesure de comprimer aucune donnée nouvelle (forte variance). La réponse apportée à ce dilemme par la complexité algorithmique est que le meilleur modèle est celui qui est en mesure d'extraire toute l'information contenue dans les données sans en ajouter de superflue. Il s'agit de la plus petite quantité d'information capable de regénérer les données. Le plus court programme contient à la fois toute l'information (compression sans perte) et seulement l'information (c'est le plus court) contenue dans les données. Toute adjonction d'information supplémentaire (explicitement ou non) n'est pas justifiée au vu des seules données : il s'agit d'une hypothèse a priori. Cette hypothèse est alors fortement sujette à être incompatible avec toute nouvelle donnée même si cette dernière n'apporte aucune information supplémentaire : c'est le phénomène de surajustement. La figure 2.9 illustre sur un exemple les phénomènes de sousajustement et surajustement en fonction de la quantité d'information. L'exemple limite les modèles au cas simple des fonctions polynomiales non bruitées.

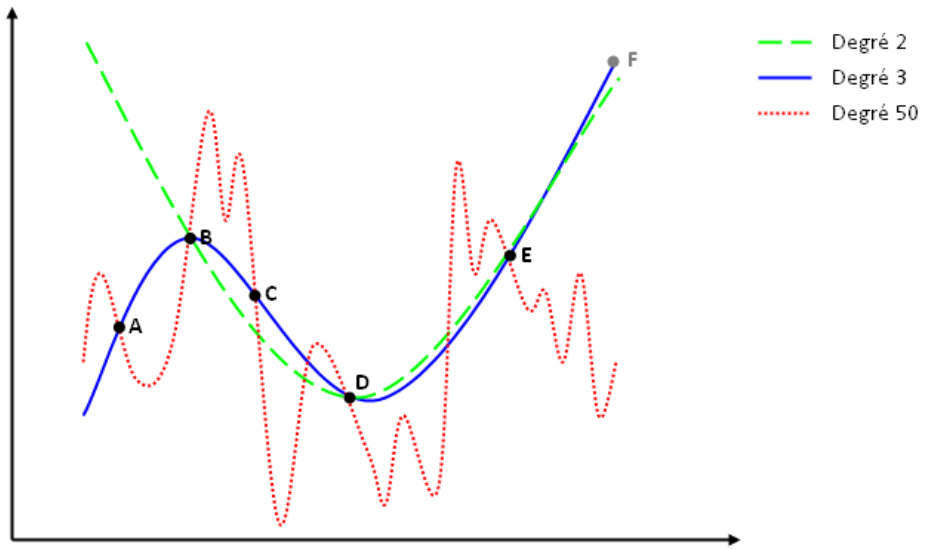


FIGURE 2.9 – Les cinq points A, B, C, D et E représentant les données sont générés par un polynôme de degré 3. Il est donc possible à un polynôme de degré 3 de capturer intégralement l'information apportée par ces cinq points (ajustement parfait). Un polynôme de degré 2 ne peut capturer que l'information apportée par trois des cinq points (sousajustement). Un polynôme de degré 50 capture toute l'information des cinq points, mais en ajoute implicitement (surajustement). Un nouveau point F se trouve sousajusté par le polynôme de degré 2 (manque d'information). Il est correctement ajusté par le polynôme de degré 3, mais n'est pas compatible avec l'information supplémentaire apportée par le polynôme de degré 50. Il s'agit ici d'un exemple où le surajustement est explicite puisqu'il y plus de paramètres à fixer (51) que de données (5).

Incalculabilité

L'incalculabilité de la complexité algorithmique est inhérente à la façon dont elle est définie. L'absence d'hypothèse sur l'espace des données empêche toute exploration heuristique, la recherche du plus court programme devant alors se faire « en aveugle » . Chaque bit du plus court programme étant maximalement informatif, cette recherche doit être exhaustive sur l'espace infini des programmes pour que l'optimalité du résultat puisse être garantie.

Nécessité d'un biais d'apprentissage

Sachant qu'il est vain d'espérer un jour produire un algorithme capable de trouver de façon générale le meilleur modèle (plus court programme) associée à un jeu de données : un biais d'apprentissage (hypothèse a priori) est nécessaire. Un biais d'apprentissage est une information introduite a priori dans le but de favoriser certains modèles au détriment d'autres, modifiant ainsi l'espace de recherche. En réduisant suffisamment cet espace, la recherche du plus court modèle peut être envisagée. La performance du modèle trouvé est alors conditionnelle à la justesse de l'a priori : c'est ce que l'on appelle le *no free lunch theorem*.

No free lunch theorem

Le *no free lunch theorem* (Wolpert & Macready, 1997) formalise l'idée qu'un algorithme (modèle) ne peut être meilleur qu'un autre sur toutes les classes de problèmes. S'il est meilleur sur une classe de problèmes, c'est que l'on y a ajouté un a priori favorisant cette classe. La performance est alors d'autant plus élevée que l'a priori est fort, c'est-à-dire que la quantité d'information qu'il apporte est élevée. Il est dans ce cas d'autant moins performant sur toutes les classes exclues par le biais d'apprentissage (classes pénalisées par l'a priori). Whitley et Watson (2005) résument l'idée de Wolpert et Macready de la façon suivante :

> *Pour une mesure de performance donnée, aucun algorithme de recherche ne peut être meilleur qu'un autre lorsque sa performance est moyennée sur l'ensemble de toutes les fonctions discrètes possibles.*

Un algorithme ne peut donc pas être meilleur qu'un autre en absolu, mais uniquement sur une classe définie de problèmes qu'il importe d'identifier clairement.

Modélisation cognitive

Selon le problème envisagé, différents bais d'apprentissage peuvent être choisis. Dans le cas de la modélisation cognitive, la classe des problèmes intéressants est celle des problèmes pouvant être résolus par le système cognitif. Nous savons que ce dernier n'est pas en mesure d'extraire la structure sous-jacente de n'importe quelle donnée (Chater, 1996 ; Oaksford & Chater, 1993 ; van Rooij, 2008). Retrouver Pi à partir de sa décomposition binaire ou l'équation d'une fractale

à partir de son image en sont des exemples parmi d'autres. Si l'on met de côté cette « forme ultime de l'intelligence » que supposerait la capacité à extraire toutes les régularités possibles, pour se restreindre à celles que l'on sait pouvoir être capturées par le système cognitif, alors le problème d'incalculabilité peut être surpassé. Les limitations inhérentes au système cognitif vont fournir le biais d'apprentissage cherché.

Chapitre 3

MDL
Minimum Description Length

Sommaire

How does one decide among competing explanations of data given limited observation ? This is the problem of model selection. It stands out as one of the most important problems of inductive and statistical inference.

P. Grünwald

3.1 Introduction

Ce chapitre présente la notion de MDL (*Minimum Description Length*), qui va constituer le coeur du modèle qui se sera décrit dans la suite de cette thèse. Dans une première partie, nous verrons les propriétés théoriques du MDL à travers les fonctions de structures de Kolmogorov (Kolmogorov, 1974 ; Vitanyi, 2005). Ce MDL idéal se fonde sur la notion de complexité algorithmique abordée au chapitre précédent. Dans une seconde partie, nous aborderons les aspects plus pratiques du MDL. Notamment le *two part coding* (Rissanen, 1978) utilisant la théorie de l'information comme approximation calculable de la complexité algorithmique.

3.2 MDL théorique

3.2.1 MDL et complexité algorithmique

Nous avons évoqué dans le chapitre précédent (section 2.4 page 32) que la complexité algorithmique en tant que formalisation du principe de simplicité permettait d'éviter le surajustement. Néanmoins, la réponse apportée au dilemme biais-variance n'est que partielle. L'utilisation du plus court programme permet d'éviter l'ajout d'information superflue pour décrire les données, ces dernières pouvant encore contenir de l'information incompressible assimilable à du bruit. Reprenons l'exemple des polynômes (Fig. 2.9 page 33) en supposant maintenant que les données comportent une part de bruit (Fig. 3.1 page ci-contre).

Le programme le plus court tel qu'il a été présenté au chapitre précédent sur la complexité algorithmique ne sépare pas explicitement la structure du bruit. Il prévient contre le surajustement (capture d'information superflue), mais pas contre le surapprentissage (capture d'information aléatoire ou bruit). Ce programme le plus court p peut être séparé en deux programmes (notés p_S et $p_{x|S}$) sans ajout substantiel [1] d'information :

$$|p| = |p_S| + |p_{x|S}| + O(1) \tag{3.1}$$

p_S contenant la structure de x et $p_{x|S}$ le bruit.

Illustrons cette idée de séparation structure/bruit par la donnée x suivante comprenant 100 bits :

$$0,0,0,1,0,1,0,1,0,0,0,1,\ldots\ldots,0,1,0,0,0,0,0,1,0,0,0,1,0,1 \tag{3.2}$$

Les bits impairs valent 0 (structure) tandis que la valeur des bits pairs a été tirée aléatoirement (bruit).

Le plus court programme p pour x est de la forme

1. Toutes les équations de cette section sont valables à une constante additive près notée O(1), car toute information indépendante de la longueur $|x|$ des données peut être considérée comme négligeable. Ici O(1) représente la taille de la ligne $ecrire(S)$ qui doit être ajoutée à la fin de p_S.

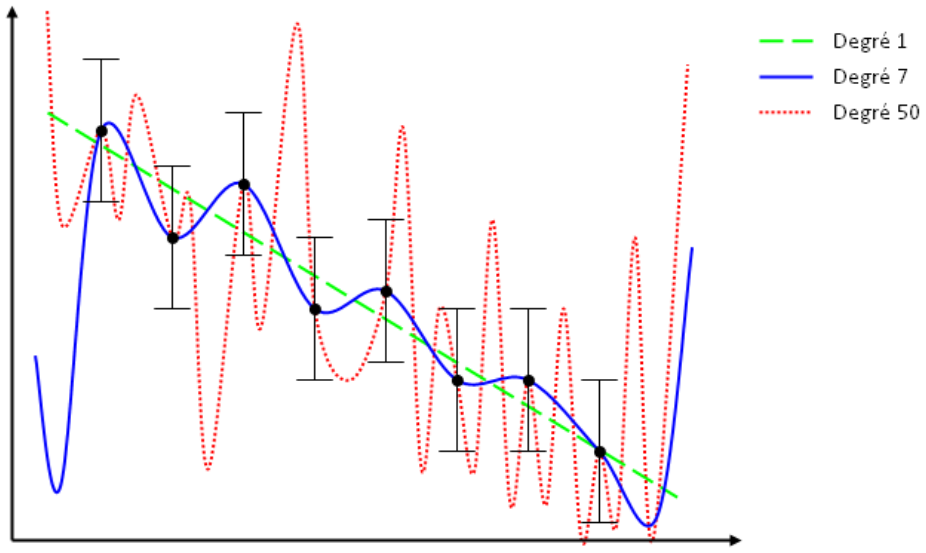


FIGURE 3.1 – Les données ont été générées par un polynôme de degré 1 auquel on a ajouté du bruit (représenté par les barres d'erreur). La structure des données peut être capturée par un polynôme de degré 1, la partie bruitée restant alors inexpliquée (la droite passe à l'intérieur des barres d'erreur, mais pas par les points). Les données peuvent être ajustées par un polynôme de degré 7, capturant à la fois la structure et le bruit (surapprentissage). Tout polynôme de degré encore supérieur ajoute alors de l'information qui n'est pas contenue dans les données (surajustement).

Algorithme 3 Plus court programme p pour x

pour tout $i \in 1 \ldots 50$ **faire**
    $x[2i-1] = 0$
fin pour
$x[2] = 0$
$x[4] = 1$
$x[6] = 1$
$x[8] = 1$
$x[10] = 0$
$\vdots$
$x[100] = 1$
$ecrire(x)$

Il peut être séparé en deux programmes : p_S comprend les trois premières lignes qui permettent de fixer les 50 bits impairs, et $p_{x|S}$ les cinquante lignes suivantes qui permettent de fixer les 50 bits pairs.

Toute donnée x peut être séparée en une partie régulière S appelée « modèle » et une partie aléatoire $x|S$ appelée « bruit » . Cela revient à décomposer le plus court programme p pour x en deux programmes : un programme p_S listant les éléments de S et un programme $p_{x|S}$ prenant en entrée les éléments de S et produisant x. Cette séparation peut être caractérisée de façon formelle en utilisant les fonctions de structure de Kolmogorov (Kolmogorov, 1974) qui vont faire une transition naturelle entre la complexité algorithmique du chapitre précédent et la notion de MDL détaillée dans ce chapitre. Les principaux résultats concernant ces fonctions sont décrits dans Gacs, Tromp, et Vitanyi (2001), Vereshchagin et Vitanyi (2002) et Vereshchagin et Vitanyi (2004). Les notations utilisées sont empruntées à Vitanyi (2005).

3.2.2 Fonctions de structure de Kolmogorov

Les fonctions de structure de Kolmogorov (*Kolmogorov structure functions*) servent de fondement théorique au MDL. Elles établissent les critères permettant de déterminer pour toute donnée x, quelle portion d'information doit être considérée comme régulière et donc capturée par le modèle S, et quelle portion d'information doit être considérée comme aléatoire ($x|S$) et exclue du modèle. On considère uniquement des données de taille finie $x \in S \subseteq \{0,1\}^*$.

Définitions

K(S) : La complexité algorithmique $K(S)$ du modèle S est la taille du plus court programme p_S capable de lister les éléments de S. On note x_i les différents éléments de $S = \{x_1, \ldots, x_m\}$, dont x fait partie. Il est indispensable pour la suite de remarquer que le modèle S est défini de façon équivalente par l'ensemble des ses éléments $\{x_1, \ldots, x_m\}$ ou par son programme [2] p_S. p_S est le plus court programme capable d'isoler les éléments $\{x_1, \ldots, x_m\}$ des autres chaînes binaires possibles. Par exemple, les trois premières lignes du programme 3 page précédente définissent S de la même façon que la liste des 2^{50} suites binaires de longueur 100, qui possèdent des zéros en positions impaires.

K(x|S) : La complexité algorithmique $K(x|S)$ des données sachant le modèle est la taille du plus court programme $p_{x|S}$ capable de produire x en prenant en entrée la liste des éléments de S.

2. Un tel programme ou toute autre description est aussi appelé « statistique de S » .

$\log_2|S|$: On note $|S|$ le nombre d'éléments [3] de S. $log_2|S|$ représente alors la quantité d'information nécessaire pour séparer un élément x_i de S des autres. Chaque $x_i \in S$ pouvant être caractérisé de façon unique par son indice i, $log_2|S|$ peut être vu comme le nombre de bits nécessaires pour spécifier l'indice i du $x_i = x$. Par exemple, 50 bits sont nécessaires pour isoler un $x_i \in S$ des 2^{50} autres éléments. En particulier pour isoler le $x \in S$ de l'exemple 3.2 page 38, les 50 bits nécessaires correspondent aux 50 dernières lignes du programme 3 page 39.

Randomness deficiency

S étant connu, $log_2|S|$ représente la quantité d'information nécessaire à une description brute de la donnée x, et $K(x|S)$ à une description compressée tenant compte des régularités. La description compressée ne peut être significativement plus longue que la description brute :

$$K(x|S) \leq log_2|S| + O(1) \tag{3.3}$$

Dans la mesure où $log_2|S|$ dépend uniquement du cardinal de S et non du contenu des $x_i \in S$, il fournit une borne supérieure à $K(x|S)$, ce dernier étant d'autant plus court que $x|S$ comporte de régularités. Si x est aléatoire au regard des autres éléments de S, alors la quantité d'information nécessaire pour caractériser x connaissant S est équivalente à la quantité d'information pour caractériser l'indice de x dans S. Dans ce cas, la différence entre les deux termes est négligeable :

$$K(x|S) \geq log_2|S| + \beta \tag{3.4}$$

On dit alors que x est aléatoire pour S au niveau β, ce qui est le critère cherché pour distinguer l'information aléatoire de l'information structurée. Dans ce cas, x est un élément typique [4] [5] de S, toute sa structure étant contenue dans S. Les degrés de libertés laissés vacants par S sont alors maximalement informatifs pour caractériser x et doivent tous être fixés par le programme $p_{x|S}$ sans qu'aucun ne puisse être déduit des autres. x ne peut être isolé des autres éléments de S qu'en utilisant une quantité d'information équivalente à la longueur (en bits) de l'indice maximal des éléments $x_i \in S$.

L'absence de typicalité de x pour le modèle S se mesure donc par la différence :

$$\delta(x|S) = log_2|S| - K(x|S) \tag{3.5}$$

appelée *randomness deficiency* de x pour S.

3. Appliqué à un ensemble, l'opérateur |.| renvoie son cardinal et appliqué à une suite binaire il renvoie sa longueur.

4. Un élément x est typique de S si S capture toute l'information de x, l'information restante étant aléatoire. Le x de l'exemple 3.2 page 38 est typique du modèle S constitué des trois première lignes du programme 3 page 39. En revanche $x = 0, 0, 0, 0, \ldots, 0, 0, 0$ ne l'est pas.

5. La notion de typicalité est similaire au principe du maximum d'entropie (Jaynes, 1982) qui tend à maximiser l'incertitude en dehors des contraintes connues (exprimées dans ce cas en termes de moments d'une distribution inconnue).

Définition des fonctions de structure de Kolmogorov

Les trois fonctions de structure de Kolmogorov ont pour but de trouver le modèle S de complexité α fixée, capturant le maximum d'information de la donnée x.

La première fonction de structure de Kolmogorov (*minimal randomness deficiency function*) $\beta_x(\alpha)$ encore appelée estimateur de meilleur ajustement, minimise la *randomness deficiency* :

$$\beta_x(\alpha) = \min_S \{\delta(x|S) : x \in S, K(S) \leq \alpha\} \tag{3.6}$$

où α est la complexité maximale autorisée pour le modèle S. La fonction $\beta_x(\alpha)$ possède des propriétés intéressantes (Vereshchagin & Vitanyi, 2002) qui ne sont pas détaillées ici. Elle n'est pas calculable en pratique, sa valeur ne pouvant être approchée avec une précision suffisante (Vitanyi, 2005).

La seconde fonction de structure de Kolmogorov $h_x(\alpha)$ ou estimateur du maximum de vraisemblance minimise le nombre d'éléments du modèle S, donc tend à maximiser la quantité d'information capturée $|x| - log_2(S)$:

$$h_x(\alpha) = \min_S \{log_2|S| : x \in S, K(S) \leq \alpha\} \tag{3.7}$$

la complexité maximale de de S étant limitée par α.

La troisième fonction de structure de Kolmogorov $\lambda_x(\alpha)$ appelée estimateur du MDL (*Minimum Description Length estimator*) minimise la taille totale du modèle et des données sachant le modèle :

$$\lambda_x(\alpha) = \min_S \{K(S) + log_2|S| : x \in S, K(S) \leq \alpha\} \tag{3.8}$$

Afin de mieux appréhender les propriétés de ces trois fonctions de structure, nous avons décidé d'aborder leurs différences sous un angle particulier : en terme de surajustement et de surapprentissage.

3.2.3 Surajustement et surapprentissage

Différence entre surajustement et surapprentissage

Parce qu'ils conduisent tous deux à une augmentation de l'erreur de généralisation, les termes « surajustement » (*overfitting*) et « surapprentissage » (*overtraining*) sont généralement employés comme synonymes. Cela tient au fait qu'il est généralement impossible de distinguer la cause réelle de l'augmentation de l'erreur de généralisation.

Dans la suite, nous réserverons le terme « surajustement » dans le sens utilisé au chapitre précédent (section 2.4), c'est-à-dire pour désigner un ajout d'information superflue pour caractériser x. Le plus court programme p pour x étant remplacé par deux programmes (p_S pour le modèle et $p_{x|S}$ pour le bruit), le

terme « surapprentissage » désigne quant à lui une mauvaise séparation entre modèle et bruit : le modèle capturant du bruit. Illustrons la différence entre surajustement et surapprentissage à l'aide d'un schéma (Fig. 3.2 page suivante).

Pour chacun des quatre cas, la quantité d'information représentant la donnée x est séparée en une partie régulière notée A (information compressible) et une partie bruitée notée B (information incompressible [6]). La partie C représente de l'information qui n'est pas nécessaire pour caractériser x. Le meilleur modèle (cas (4)) est celui séparant parfaitement A de B sans ajout d'information superflue C.

En abscisse est portée la quantité d'information. Le haut de chaque schéma représente la donnée x découpée sous forme d'un modèle S et d'une partie $x|S$. Le bas de chaque schéma représente la longueur des plus courts programmes associés p_S et $p_{x|S}$. La séparation entre S et $x|S$ est matérialisée par un trait vertical.

Cas (1), sousapprentissage : Le modèle S ne capture pas toute l'information régulière A, mais seulement une partie (A1). Il s'agit du cas où $K(x|S) < log_2|S| + O(1)$, x n'est donc pas un élément typique de S.

Cas (2), surapprentissage : Le modèle S capture toute l'information régulière A, ainsi qu'une partie du bruit (B1). L'information restante étant incompressible ($K(x|S) \geq log_2|S| + \beta$), x est typique de S.

Cas (3), surajustement : Le modèle S capture de l'information (C1) qui n'est pas nécessaire pour caractériser x. Dans ce cas on a $K(S) + K(x|S) > K(x) + O(1)$. La quantité d'information ajoutée vaut $K(C1)$.

Cas (4), plus petite statistique suffisante : Le modèle S capture toute l'information compressible et uniquement celle-là, laissant le bruit à la partie $x|S$. Il n'y a ni sousajustement (non représenté), ni surajustement (3), ni sousapprentissage (1), ni surapprentissage (2).

Exemple de surajustement et de surapprentissage

Différents exemples possibles de surajustement et surapprentissage pour l'exemple 3.2 page 38 sont résumés par le tableau 3.1 page 45.

6. Les notions de compressibilité et d'incompressibilité dépendent du seuil β choisi (équation 3.4 page 41).

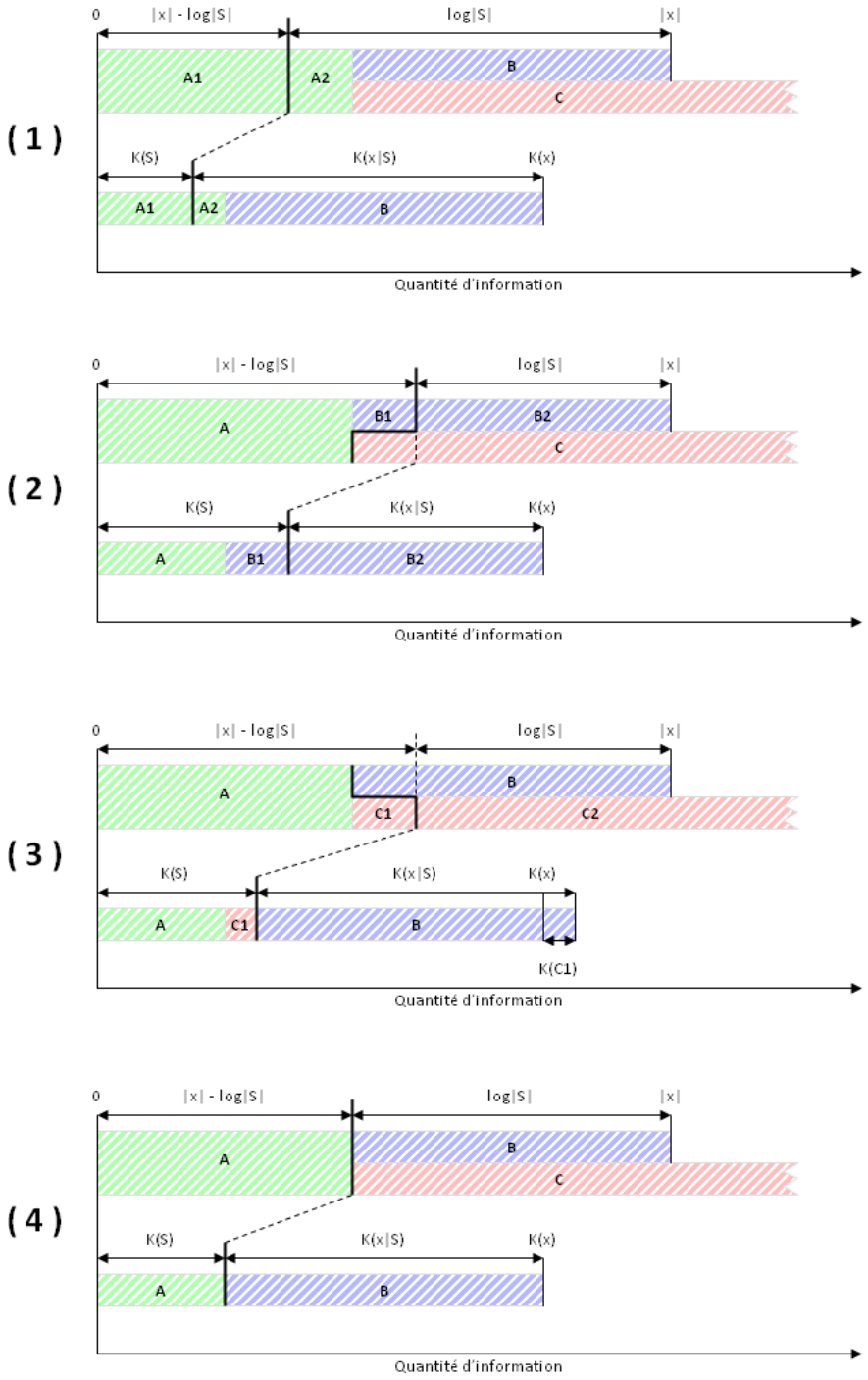


FIGURE 3.2 – Représentation de l'information, compressible (A), incompressible (B) et superflue (C), dans différents cas possibles de séparations entre modèle S et données sachant modèle $x|S$. Cas (1) : sousapprentissage. Cas (2) : surapprentissage. Cas (3) : surajustement. Cas (4) : plus petite statistique suffisante.

	p_S	$p_{x\|S}$	$K(S)$	$K(x\|S)$	$log_2\|S\|$
Plus petite statistique suffisante (4)	bits impairs nuls	$x[2] = 0$, $x[4] = 1$, $x[6] = 1$, ·····, $x[100] = 1$	10 [a]	50	50
Surapprentissage (2)	bits impairs nuls et $x[2] = 0$, $x[4] = 1$, $x[6] = 1$, ·····, $x[20] = 0$	$x[22] = 1$, $x[24] = 0$, $x[26] = 0$, ·····, $x[100] = 1$	20 [b]	40	40
Sousapprentissage (1)	1 bit sur 4 nul	$x[2] = 0$, $x[3] = 0$, $x[4] = 1$, $x[6] = 1$, $x[7] = 0$, $x[8] = 1$, ·····, $x[100] = 1$	5 [c]	55	75
Surajustement (3)	les bits impairs sont les 50 bits du développement binaire de Pi [d] à partir de la décimale m_1.	$x[2] = 0$, $x[4] = 1$, $x[6] = 1$, ·····, $x[100] = 1$	70 [e]	50	50
Surajustement et surapprentissage (2) et (3)	100 bits du développement binaire de Pi à partir de la décimale m_2.		120 [f]	0	0

a. Valeur vérifiant $K(S) + K(x|S) = K(x) + O(1)$.
b. 10 bits pour définir les bits impairs, et 10 bits pour les dix premiers bits pairs.
c. Valeur deux fois moindre que $K(S)$ du le cas (4).
d. Suppose que Pi est un nombre univers, c'est-à-dire que son développement binaire passe par toutes les séquences possibles de longueur finie.
e. 20 bits pour générer Pi (valeur arbitraire) et 50 bits pour spécifier l'indice m_1.
f. 20 bits pour générer Pi (valeur arbitraire) et 100 bits pour spécifier l'indice m_2.

TABLE 3.1 – Information véhiculée par les programmes p_S et $p_{x|S}$ dans différents cas de surapprentissage et de surajustement. La donnée x est celle de l'exemple 3.2 page 38, avec $|x| = 100bits$ et $K(x) = 60bits$.

.

Surajustement et surapprentissage pour les fonctions de structure

Il est maintenant possible de comparer le comportement des trois fonctions de structure de Kolmogorov au regard de leurs capacités à éviter le surajustement et le surapprentissage (Fig. 3.3).

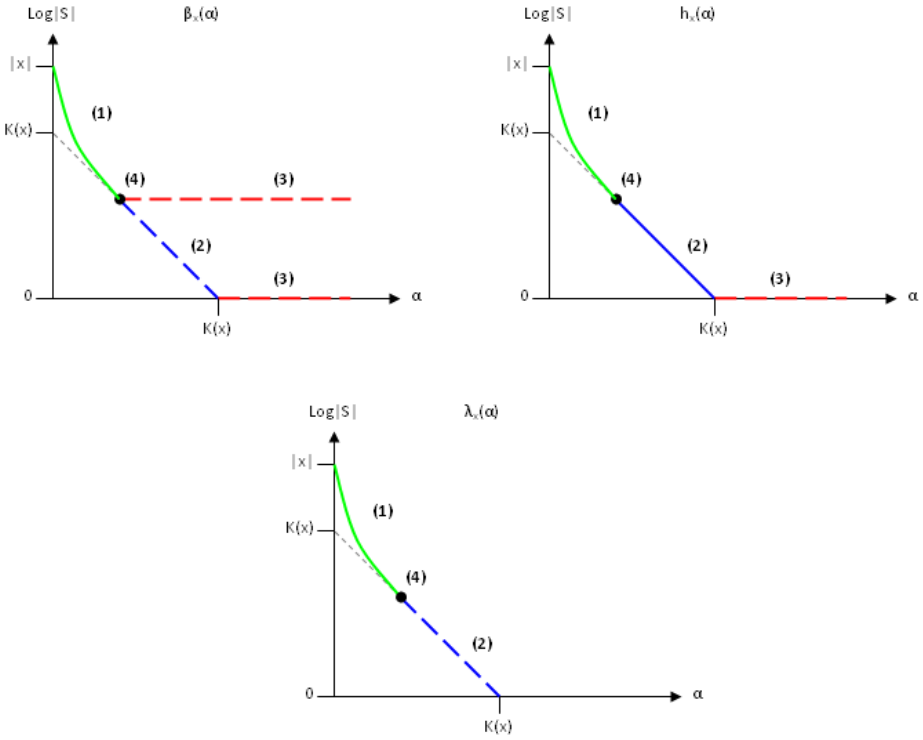


FIGURE 3.3 – Evolution de la quantité $log_2|S|$ en fonction de α pour les trois fonctions $\beta_x(\alpha)$, $h_x(\alpha)$ et $\lambda_x(\alpha)$. Les portions pointillées sont celles pour lesquelles la valeur de la statistique est constante (cf. figure 3.4 page suivante). Les portions de courbe notés (1) à (4) correspondent aux quatre cas envisagés figure 3.2 page 44. La ligne pointillée (ligne de suffisance) qui se confond avec la portion (2) matérialise $K(x|S)$.

Cas (1) : Pour $\alpha = 0$, $S = \varnothing$ donc $log_2|S| = |x|$. Comme les trois estimateurs tendent à minimiser le nombre d'éléments du modèle, l'augmentation de α s'accompagne nécessairement d'une diminution au moins équivalente de $log_2|S|$. Chaque bit ajouté au modèle ne pouvant apporter significativement moins qu'un bit d'information, la dérivée en tout point de la courbe (1) est inférieure à -1. Comme les trois estimateurs tendent à capturer un maximum d'information, la portion (1) de la courbe est convexe. L'écart entre la courbe (1) et la ligne de suffisance représente la quantité $\delta(x|S) = log_2|S| - K(x|S) \geq 0$.

Cas (2) : La portion de droite (2) se confondant avec la ligne de suffisance a un coefficient directeur de -1 : chaque bit ajouté au modèle diminue d'autant la

quantité $log_2|S|$, donc $\delta(x|S) = 0$.

Cas (3) : La portion de droite (3) correspond à l'ajout d'information superflue pour caractériser x et ne modifie pas le nombre d'éléments du modèle.

Cas (4) : La plus petite statistique suffisante se situe à l'interface entre sous-sapprentissage (1) et surapprentissage (2).

L'estimateur $\lambda_x(\alpha)$ est le seul à ne pas pouvoir surajuster les données (3). Un surapprentissage est néanmoins possible (2), mais comme pour $\beta_x(\alpha)$ il reste limité dans la mesure où cela n'améliore pas la valeur de la statistique. À l'inverse l'estimateur du maximum de vraisemblance possède une tendance naturelle au surapprentissage (Fig. 3.4).

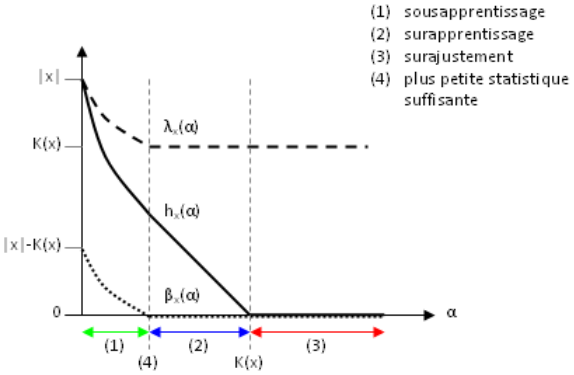


FIGURE 3.4 – Evolution de $\beta_x(\alpha)$, $h_x(\alpha)$ et $\lambda_x(\alpha)$ en fonction de α.

Comme le suggère le titre de ce chapitre, la statistique qui va maintenant être utilisée est celle du MDL, notamment pour ses capacités à éviter le surajustement et à limiter le surapprentissage. Le cadre théorique très général des machines de Turing que nous venons de décrire va être abandonné car il pose à la fois le problème de l'incalculabilité (discuté section 2.3.3 page 30), ainsi que celui de la plausibilité cognitive (discuté section 2.4 page 34). Pour ces deux raisons, nous allons maintenant nous limiter à un cadre beaucoup plus restreint : celui de la théorie de l'information dans lequel seules les régularités fréquentielles peuvent être capturées.

3.3 MDL pratique

Le MDL pratique peut être justifié à partir de la loi de Bayes (Vitanyi & Li, 2000) :

$$P(S|x) = \frac{P_1(x|S) \times P_2(S)}{P_1(x)} \tag{3.9}$$

donc

$$log_2 \frac{1}{P(S|x)} = log_2 \frac{1}{P_1(x|S)} + log_2 \frac{1}{P_2(S)} - log_2 \frac{1}{P_1(x)} \tag{3.10}$$

où P_1 est la mesure de probabilité sur l'espace $\mathcal{X}$ des données, et P_2 sur l'espace $\mathcal{S}$ des modèles. P est induite par P_1 et P_2. Le modèle le plus vraisemblable au vu des données maximise $P(S|x)$, donc minimise :

$$S = \arg\min_{S \epsilon \mathcal{S}} \left(log_2 \frac{1}{P_1(x|S)} + log_2 \frac{1}{P_2(S)} \right) \tag{3.11}$$

où $log_2 \frac{1}{P_1(x)}$ a été retiré car il est indépendant de S.

En interprétant l'équation 3.11 dans le cadre des machines de Turing, c'est-à-dire en termes de probabilité algorithmique, cela équivaut à :

$$S = \arg\min_{S \epsilon \mathcal{S}} \left(K(x|S) + K(S) \right) \tag{3.12}$$

Exprimée dans le cadre de la théorie de l'information, c'est-à-dire en terme de tailles de codage, on obtient :

$$S = \arg\min_{S \epsilon \mathcal{S}} \left(C_1(x|S) + C_2(S) \right) \tag{3.13}$$

où $C(.)$ est le codage de Shannon-Fano introduit au chapitre précédent. Cela revient à définir de façon a priori deux langages de description pour S et $x|S$, puis à compresser ces deux descriptions par un codage de Shannon-Fano. Les probabilités $P_1(x|S)$ et $P_2(S)$ sont alors dépendantes des tailles de codages $C_1(x|S)$ et $C_2(S)$, qui sont elles-mêmes implicitement définies à partir du choix des langages de description de $x|S$ et S.

Dans le cadre restreint où une chaîne x est composée de réalisations indépendantes et identiquement distribuées (i.i.d.) d'une variable aléatoire discrète (v.a.d.), il a été montré par Leung-Yan-Cheong et Cover (1978) que la taille de codage $C(x)$ donnée par un code de Shannon-Fano ne pouvait différer significativement de la complexité algorithmique $K(x)$.

L'approche envisagée par le MDL pratique est de diminuer l'expressivité du langage de représentation, donc des régularités pouvant être capturées, afin que la plus courte description des données dans ce langage soit calculable. Diminuer l'expressivité du langage introduit le biais d'apprentissage nécessaire pour surmonter le problème de l'incalculabilité discuté précédemment.

MDL *two-part coding*

La version la plus simple du MDL, appelée *MDL two-part coding*[7] (Rissanen, 1978) se base sur l'équation 3.13, l'apprentissage se faisant en minimisant la taille totale des deux représentations S et $x|S$. Elle se fonde sur l'existence de

7. Cette approche est similaire au *Minimum Message Length* (Wallace & Boulton, 1968).

régularités fréquentielles uniquement, les deux parties S et $x|S$ étant représentées à l'aide de chaînes de caractères supposées issues de deux v.a.d. dont les réalisations sont i.i.d. Dans ce cadre restreint, la théorie de l'information permet de calculer la taille de codage totale des deux parties. L'apport du MDL est d'utiliser cette taille de codage totale comme critère de sélection de modèles.

Exemple 1

Illustrons le fonctionnement du MDL *two-part coding* à travers un exemple simple : les abréviations dans la langue française. L'idée sous-jacente à l'utilisation d'abréviations est que tout ce qui n'apporte pas d'information peut-être supprimé. Certains suffixes, par exemple « ment » et « tion » sont couramment abrégés en ^{mt} et $^{t^\circ}$ ($genti^{mt}$, $manipula^{t^\circ}$). L'information portée par quatre lettres est en quelque sorte redondante et peut être compressée en deux lettres seulement.

Plus un groupe de lettre est fréquent, plus il est prévisible, donc moins sa présence apporte d'information et plus il peut être compressé : c'est l'idée générale de la théorie de l'information. Ainsi dans une langue, seuls les mots fréquents sont abrégés (« sans » → « ss » pour un gain de 2 caractères) alors que d'autres mots moins fréquents ne le sont jamais (« renard » → « rnrd » pour un gain de 2 caractères).

Abréger a un coût car il est nécessaire de mémoriser la correspondante entre l'abréviation et le mot. La charge ajoutée pour se rappeler que « rnrd » = « renard » est de loin supérieure au gain apporté par l'utilisation de « rnrd » plutôt que « renard » . Il s'agit là de l'idée véhiculée par le MDL. Une abréviation du type « rnrd » = « renard » n'est intéressante que si la charge nécessaire à sa définition (10 caractères) est inférieure au gain apporté par son utilisation (2 caractères par utilisation). Par exemple :

- $S = \varnothing$ *(0 caractères)*
- $x|S = sans, sans, sans, sans, sans, renard, renard$ *(32 caractères)*

nécessite un total de 32 caractères,

- $S = sans \rightarrow ss$ *(6 caractères)*
- $x|S = ss, ss, ss, ss, ss, renard, renard$ *(22 caractères)*

abrège « sans » en « ss » et conduit à un total de 28 caractères, et

- $S = sans \rightarrow ss, renard \rightarrow rnrd$ *(16 caractères)*
- $x|S = ss, ss, ss, ss, ss, rnrd, rnrd$ *(18 caractères)*

abrège à la fois « sans » et « renard » , pour un total de 34 caractères.

Dans le MDL *two-part coding*, les notions de « charge ajoutée » et de « gain » sont définies rigoureusement sous forme de tailles de codage, mais le principe de séparation entre S et $x|S$ reste identique.

Exemple 2

L'exemple suivant est tiré de Dowman (soumis) qui a utilisé le MDL *two-part coding* pour rendre compte de l'acquisition de règles syntaxiques. Le système développé permet d'apprendre de façon inductive le plus petit nombre de règles permettant de reproduire un ensemble donné de phrases. L'ensemble d'apprentissage x contient les 19 phrases suivantes :

John hit Mary
Mary hit Ethel
Ethel ran
John ran
Mary ran
Ethel hit John
Noam hit John
Ethel screamed
Mary kicked Ethel
John hopes Ethel thinks Mary hit Ethel
Ethel thinks John ran
John thinks Ethel ran
Mary ran
Ethel hit Mary
Mary thinks John hit Ethel
John screamed
Noam hopes John screamed
Mary hopes Ethel hit John
Noam kicked Mary

Une grammaire peut être facilement représentée sous forme d'un arbre. Ici, le langage de représentation choisi par Dowman (soumis) est un arbre binaire. La partie grammaire S contient les règles de type ET, dont certains degrés de libertés restent vacants (OU). Par exemple (Fig. 3.7 page 52), la règle $1 : S \rightarrow NP\,VP$ indique qu'une phrase se compose d'un groupe nominal [8] suivi d'un groupe verbal (ET), le groupe nominal pouvant prendre les valeurs $John$, $Ethel$, $Mary$ ou $Noam$ (OU). Recréer le jeu de données x à partir de la grammaire S nécessite de fixer les degrés de libertés laissés vacants par les règles grammaticales (partie $x|S$).

Initialement, le système commence avec une grammaire excessivement générale (Fig. 3.5 page suivante) acceptant toute suite de mots comme grammaticalement correcte. Un mécanisme d'exploration permet de générer des grammaires voisines de la grammaire courante. Les opérations élémentaires pouvant être appliquées à la grammaire courante sont : l'ajout d'une nouvelle règle aléatoire, la suppression d'une règle existante, la modification d'un symbole ou de l'ordre des symboles d'une règle existante, et la construction de deux règles $X \rightarrow A$, $A \rightarrow Z$ à partir des règles existantes $X \rightarrow Y$, $Y \rightarrow Z$. La nouvelle grammaire ainsi générée remplace la précédente avec une probabilité dépendant de sa performance (recuit simulé). Le critère de performance utilisé (MDL) est la taille de codage totale des deux parties S et $x|S$.

Pour calculer la taille de codage de S, les règles constituant la grammaire sont préalablement concaténées sous forme d'une chaîne de caractères unique. Pour éviter l'utilisation de séparateurs, les règles sont binarisées en ajoutant le symbole *null* aux règles unaires. Par exemple, la grammaire de la figure 3.7 page

8. Le système n'ayant aucune connaissance grammaticale a priori, les symboles utilisés sont nécessairement arbitraires. L'utilisation des symboles conventionnels S, NP, VP, Vt, Vs est faite a posteriori pour faciliter la compréhension de la grammaire.

suivante est mise sous la forme :

$$S, NP, VP, VP, ran, null, VP, screamed, null, \ldots \tag{3.14}$$

La taille de codage de S est ensuite calculée en associant à chaque symbole de la chaîne un codage de Shannon-Fano de taille optimale :

$$C(symbole) = log_2 \frac{1}{P(symbole)} \tag{3.15}$$

qui ne dépend que de la fréquence empirique $P(symbole)$ du symbole dans la chaîne. $C(S)$ se calcule alors comme étant la somme des tailles de codage de chaque symbole de la chaîne [9].

De façon similaire, la partie $x|S$ de la figure 3.7 page suivante est mise sous la forme d'une chaîne de caractères

$$1, 10, 4, 6, 12, 1, 12, 4, 6, 11, 1, 11, 2, \ldots \tag{3.16}$$

où la taille de codage associée à chaque symbole dépend de la probabilité de règle correspondante sachant la grammaire. Comme seule la règle 1 commence par S, la probabilité de déclencher cette règle sachant que l'on est à la racine de l'arbre est de 1, et la taille de codage correspondante est nulle. De la même façon, la taille de codage du symbole 10 sachant que la règle attend un groupe nominal (règle 1) est $log_2 \frac{1}{P(John|NP)} = log_2 \frac{36}{11}$.

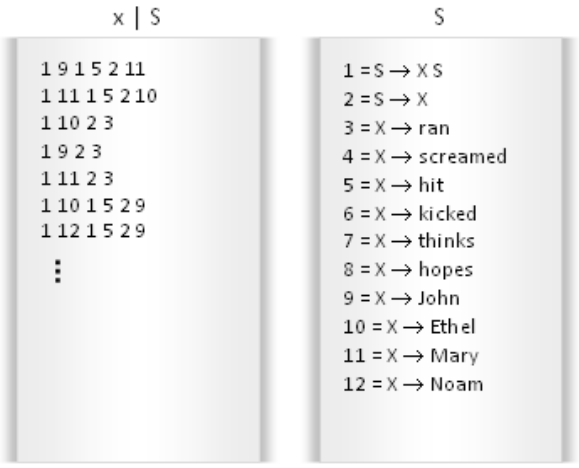


FIGURE 3.5 – Etat du système au début de l'apprentissage. La grammaire S très générale a une taille de codage faible mais ne capture aucune régularité. L'information nécessaire pour reconstruire x à partir de S est proche de celle véhiculée par x : la taille de codage de $x|S$ est donc élevée.

9. On rappelle que chaque symbole de la chaîne est supposé être la réalisation d'une v.a.d.i.i.d. La somme se fait bien sur les réalisations (*tokens*) et non sur les évènements (*types*).

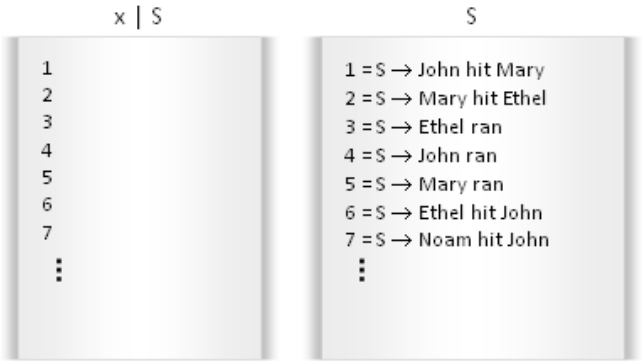


FIGURE 3.6 – Cas extrême dans lequel la grammaire contient presque toute l'information, conduisant à une taille de codage élevée pour S et faible pour $x|S$.

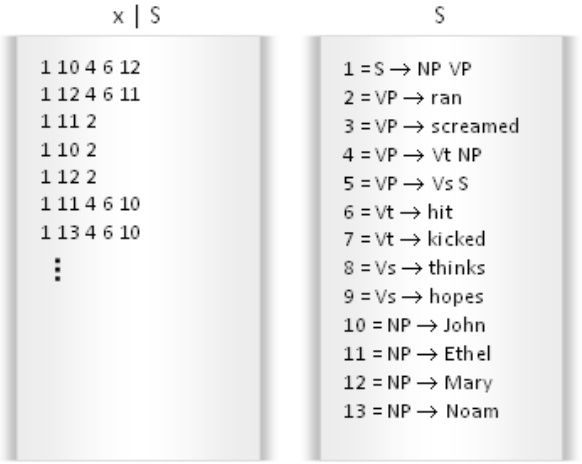


FIGURE 3.7 – Cas minimisant la taille de codage totale ($C(S) + C(x|S)$). La grammaire ainsi produite n'est pas sans rappeler la grammaire anglaise.

MDL et dilemme biais-variance

Le MDL est souvent présenté comme une méthode d'apprentissage inductif permettant de résoudre le problème de sélection de modèles. En effet, le *two-part coding* prend explicitement en compte le dilemme biais-variance introduit au chapitre 1. La question de l'importance relative accordée aux deux facteurs biais et variance se trouve alors résumée en un seul problème : trouver l'encodage le plus court ($C(S) + C(x|S)$) des données. Sous cette forme de codage bipartite, un trop fort biais comme une trop forte variance conduisent à une taille de codage totale élevée. Ce sont les deux cas extrêmes présentés figure 3.5 page 51 (fort biais) et figure 3.6 page précédente (forte variance). Le cas intermédiaire de la figure 3.7 page ci-contre est celui qui factorise au mieux l'information, la partie S contenant la structure et la partie $x|S$ ce qui est assimilable à du bruit au vu du langage de représentation utilisé.

Langage de représentation et objectivité

L'objectivité de la méthode est illusoire dans la mesure où l'on s'éloigne du MDL idéal décrit section 3.2. Dans le MDL théorique la probabilité algorithmique du modèle ($P(S)$) est une mesure de probabilité universelle, indépendante de la machine de Turing considérée (cf. Théorème d'invariance section 2.3.2 page 29). Dans le MDL pratique en revanche, la probabilité a priori $P_2(S)$ se trouve implicitement fixée par le langage choisi pour S. De même, $P_1(x|S)$ dépend du langage choisis pour $x|S$. L'importance relative donnée à S et $x|S$, donc la solution apportée au dilemme biais-variance, dépend des langages utilisés.

Ensemble d'apprentissage et ensemble de test

Le paradigme classique en sélection de modèles est d'entraîner chaque modèle sur une partie des données (ensemble d'apprentissage), puis d'évaluer leur erreur de généralisation sur le reste des données (ensemble de test). Il nous semble que le MDL puisse être vu comme une manière élégante de faire jouer à chaque donnée le rôle d'ensemble d'apprentissage et d'ensemble de test. Au lieu de fixer un seuil pour l'erreur de généralisation, les langages de description choisis pour S et $x|S$ contraignent implicitement le poids relatif des deux parties. La séparation entre ensemble d'apprentissage et ensemble de test n'est plus explicite.

3.4 Conclusion

À travers ce chapitre et le précédent, nous avons vu que la complexité algorithmique fournissait un cadre permettant de quantifier de façon objective et universelle la simplicité d'une chaîne binaire. Bien que cette mesure soit incalculable en pratique, elle fournit une vision générale des problèmes liés à l'apprentissage inductif. Le but de la modélisation est de minimiser l'erreur de prédiction, c'est-

à-dire de capturer uniquement la structure des données sans surajustement ni surapprentissage.

Ce cadre théorique permet notamment d'aborder de façon différente le problème posé par le dilemme biais-variance. Elle montre qu'il existe une solution, au moins dans le cas où le langage utilisé possède l'expressivité d'une machine de Turing, et que cette solution ne nécessite pas la traditionnelle séparation entre ensemble d'apprentissage et ensemble de test. En pratique, l'utilisation d'un langage moins expressif peut conduire à une situation différente de celle présentée figure 3.2 page 44 et plus proche de celle de la figure 3.8. Par exemple, dans le cas où des données sont générées à partir d'une fonction sinus à laquelle s'ajoute du bruit, il est impossible en utilisant uniquement un modèle polynomial de capturer toute la structure des données (A) sans capturer de bruit (B) ni d'information superflue (C).

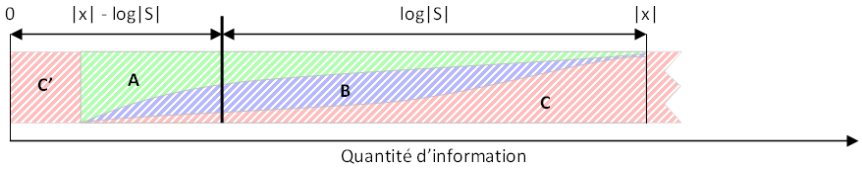


FIGURE 3.8 – Représentation schématique équivalente à la figure 3.2 dans le cas pratique où le langage utilisé ne possède pas l'expressivité d'une machine de Turing. D'une part, le choix du langage introduit un biais d'apprentissage susceptible d'ajouter une information non-contenue dans les données (C'). D'autre part, l'impossibilité du langage à pouvoir représenter tous les types de régularités peut conduire à une situation où la plus courte description des données ne peut être produite dans ce langage. En augmentant la complexité du modèle on est donc susceptible de capturer à la fois du bruit (B) et de l'information superflue (C).

Le cadre que nous venons de décrire va servir de fondement théorique au MDLChunker. Comme évoqué brièvement à la fin du chapitre 2, l'incalculabilité de la complexité algorithmique n'est pas problématique dans la cadre de la modélisation cognitive puisque les performances du système cognitif sont inférieures à celle d'une machine de Turing (van Rooij, 2008). Le problème auquel il faut faire face en revanche, est de déterminer le type de régularités que les humains sont capables d'extraire de leur environnement. Le MDLChunker qui va maintenant être décrit en détail se limite aux deux types de régularités les plus simples : les régularités fréquentielles et les régularités de type ET.

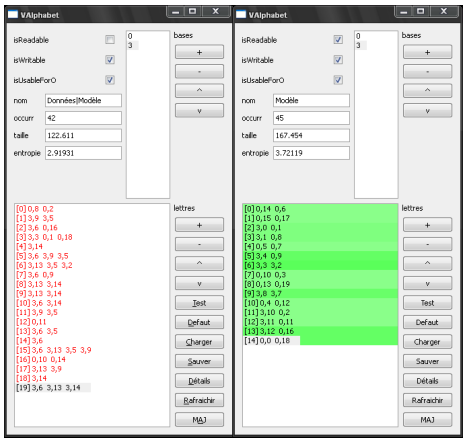
Deuxième partie

Modélisations

Chapitre 4

MDLChunker

Sommaire



4.1 Introduction

Ce chapitre décrit le fonctionnement du MDLChunker. Il est assez détaillé pour permettre au lecteur intéressé de pouvoir ré-implémenter entièrement le modèle s'il le souhaite. Une présentation générale est faite dans l'introduction, puis les différents points abordés sont repris et discuté en détail dans la suite du chapitre. Pour cette raison, le chapitre peut paraître fragmenté et parfois redondant, car certaines informations sont reprises pour être approfondies. Avant de décrire le MDLChunker en détail, il nous a semblé utile de le replacer dans le cadre de la modélisation cognitive. Le but recherché est avant tout de concevoir un modèle cognitivement plausible et non un modèle performant au niveau algorithmique. Un soin tout particulier a notamment été apporté à ce que le MDLChunker ne contienne aucun paramètre.

4.1.1 Modélisation cognitive

Analyse rationnelle de la cognition

Nous avons présenté au chapitre 1 le cadre de l'analyse rationnelle de la cognition proposée par Anderson (1989). Les six étapes de cette analyse sont : définir **le but** que cherche à atteindre le système cognitif, puis **l'environnement** dans lequel il évolue ainsi que ses **limitations**. Le système cognitif étant le produit d'une longue évolution il est possible de supposer qu'il a développé une stratégie **optimale** (ou presque) pour atteindre ce but dans cet environnement et soumis à ces limitations. Nous allons chercher à faire de même à l'aide d'un modèle computationnel, puis à **valider** ensuite expérimentalement les résultats obtenus. Une amélioration du modèle est alors possible en **itérant** à nouveau ces cinq étapes.

Les deux chapitres précédents nous ont permis de fournir une justification théorique au principe de simplicité. Nous allons maintenant l'implémenter sous forme d'un modèle computationnel afin de tester l'hypothèse selon laquelle le but du système cognitif est en réalité de construire les représentations les plus simples cohérentes avec ses perceptions du monde extérieur (Chater, 1999). Les représentations pouvant être extrêmement complexes, nous allons restreindre notre étude aux régularités de type ET, dont les représentations sont généralement appelées « chunks » . Les raisons de ce choix sont expliquées au chapitre 1. Nous supposerons que l'environnement ne contient que des régularités de type ET, et des régularités fréquentielles.

Dans un premier temps, aucune limitation n'est imposée au modèle. Ce sera l'objet du chapitre suivant. Le modèle considéré a pour but de créer les chunks permettant la représentation la plus simple des stimuli en provenance de l'environnement extérieur. Le comportement du MDLChunker est quasi-optimal pour cette tâche.

Optimalité

Si la seule information disponible est de type fréquentielle, un stimulus donné

$$a, a, b, a, c, a, b, d \tag{4.1}$$

peut être compressé de façon optimale à l'aide d'un code de Shannon-Fano

$$1 = a \tag{4.2}$$
$$01 = b \tag{4.3}$$
$$001 = c \tag{4.4}$$
$$000 = d \tag{4.5}$$
$$\tag{4.6}$$

ce qui donne

$$11011001101000 \tag{4.7}$$

Si à cette information fréquentielle s'ajoutent uniquement des régularités de type ET

$$xyz, xyz, vw, xyz, c, xyz, vw, d \tag{4.8}$$

alors un mécanisme capable d'extraire les chunks

$$a = xyz \tag{4.9}$$
$$b = vw \tag{4.10}$$

nous ramène dans la situation précédente

$$a, a, b, a, c, a, b, d \tag{4.11}$$

pour laquelle le codage est optimal. Si ce mécanisme d'extraction de chunks est lui-même optimal, alors l'algorithme général est optimal.

Principe de simplicité

L'utilisation des chunks permet une compression des régularités de type ET, et l'utilisation d'un code de Shannon-Fano permet une compression des régularités fréquentielles. Pour être conforme au principe de simplicité, le but est de trouver la meilleur compression de l'ensemble des stimuli, la taille de codage totale étant utilisée comme critère permettant de déterminer quels chunks doivent être créés.

4.1.2 Présentation informelle

Cette section effectue une présentation rapide du modèle permettant d'avoir un aperçu des mécanismes mis en jeu. Les détails de son fonctionnement sont expliqués de façon approfondie section 4.2.

Données : Les données sur lesquelles s'effectuent l'apprentissage doivent être discrètes à la fois dans l'espace et dans le temps. Les raisons pour lesquelles l'approche choisie est incompatible avec les données continues seront discutées en section 4.4. On entend par données spatialement discrètes, des données dont les valeurs sont quantifiées, par exemple $a, b, c, \ldots, 1, 2, 3, \ldots$ ou $rouge, vert, bleu, \ldots$. Cela correspond typiquement aux données pouvant être obtenues par une représentation de type attribut-valeur. Au niveau temporel, le modèle impose deux contraintes. Une contrainte forte : chaque donnée doit être clairement distincte de la précédente. Et une contrainte faible qui sera discutée section 4.4 : les données sont présentées par paquets à l'intérieur desquels l'ordre n'a pas d'importance. Nous verrons là aussi section 4.4 que la contrainte d'absence d'ordre peut-être relaxée au prix de modifications mineures. Dans la suite, ces paquets de données sans ordre seront systématiquement représentés par des listes de nombres [1]. On leur préférera le nom plus concis de « stimuli » , qui, en outre, a le mérite d'indiquer clairement leurs fonctions : il s'agit de données brutes en provenance de l'extérieur qui vont servir de base à la création de représentations de plus haut niveau.

Principe : Les stimuli sont présentés un par un au modèle, qui va être capable d'extraire certaines des régularités statistiques qu'ils contiennent, construisant ainsi une représentation interne « compressée » des stimuli externes. La discussion des arguments théoriques motivant cette recherche d'une représentation compressée a fait l'objet des chapitre 2 et 3. Comme expliqué dans la section précédente, le modèle va être en mesure d'extraire et de compresser deux types de régularités : les régularités de type ET (caractères qui apparaissent conjointement dans les stimuli), et les régularités de type fréquentielles (un caractère aura une taille faible s'il est fréquent). Les premières sont compressées sous forme de chunks (*chunking*), les secondes le sont lors du calcul des tailles de codage grâce à la théorie de l'information (code de Shannon-Fano).

Architecture : Le modèle est composé de deux parties : une partie Chunks et une partie Stimuli|Chunks (Fig. 4.1 page ci-contre). La partie Chunks contient les chunks créés. La partie Stimuli|Chunks (stimuli sachant chunks) contient les stimuli ré-exprimés sous une forme dans laquelle la redondance a été supprimée. En extrayant les régularités des stimuli, la partie Chunks permet de factoriser l'information redondante. La partie Stimuli|Chunks se contentant de faire référence aux régularités déjà définies dans la partie Chunks. L'information contenue dans les deux parties Chunks et Stimuli|Chunks est identique à celle initialement contenue dans les stimuli. Dans les stimuli, cette information est sous forme développée, dans le modèle elle est sous forme factorisée.

Partie Stimuli|Chunks : La partie Stimuli|Chunks contient les stimuli factorisés grâce aux chunks. Par exemple, le quatrième stimulus de la figure 4.1 admet C1 C2 C3 C4 C5 comme forme développée (non représentée) et C9 C7

1. L'utilisation de nombres plutôt que de lettres ou de n'importe quels autres symboles et arbitraire et n'enlève rien à la généralité de l'approche. Il s'agit en quelque sorte de l'alphabet utilisé.

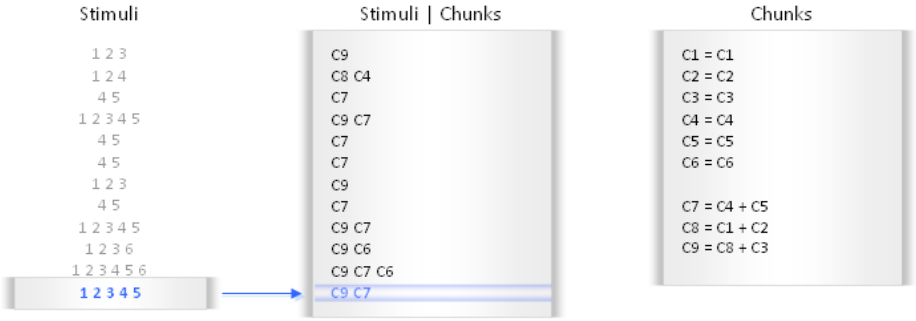


FIGURE 4.1 – Exemple d'état des deux parties Chunks et Stimuli|Chunks au cours de l'apprentissage. 12 stimuli ont déjà été perçus (partie Stimuli|Chunks) et 3 chunks ont été créés en plus des 6 chunks-canoniques présents au départ (partie Chunks).

comme forme factorisée. Cette factorisation est possible car C7 et C9 sont définis dans la partie Chunks.

Partie Chunks : La partie Chunks comprend deux types de chunks. Les chunks-canoniques (C1 à C6 sur la figure 4.1) qui ne sont pas de vrai chunks à proprement parler, puisqu'ils ne regroupent pas d'autres chunks. Ils sont définis comme étant égaux à eux-mêmes ($C1 = C1$, $C2 = C2$). Ils jouent le rôle d'alphabet. Tout stimulus peut être représenté uniquement à l'aide des chunks-canoniques : c'est la représentation développée évoquée au paragraphe précédent. Le stimulus 1 2 3 4 5 peut être représenté C1 C2 C3 C4 C5. Les chunks-canoniques définissent l'espace des stimuli qui peuvent être représentés. Ils sont créés au moment de l'initialisation et leur nombre reste fixe durant l'apprentissage. La seconde catégorie est celle des chunks non-canoniques (C7, C8 et C9 sur la figure 4.1). Ils sont créés au cours de l'apprentissage. Chacun correspond à une régularité observée dans les stimuli. Par exemple, la régularité "4 et 5 apparaissent souvent ensemble dans les stimuli" est désigné par le chunk $C7 = C4 + C5$.

Mécanismes utilisés : Le modèle fonctionne grâce à trois mécanismes. Le premier est le calcul des tailles de codage de tous les chunks à partir de leur fréquence d'apparition. Ensuite vient la factorisation du stimulus perçu sous la forme la plus concise possible au moment où il est ajouté à la partie Stimuli|Chunks. Et enfin l'optimisation, c'est-à-dire la création de nouveaux chunks dans la partie Chunks lorsque cela conduit à une diminution de la taille totale des deux parties Stimuli|Chunks et Chunks. La factorisation et l'optimisation sont donc directement dépendantes du mécanisme de calcul des tailles de codage. Si l'on fait un parallèle avec la programmation informatique, l'optimisation consiste à écrire des bibliothèques contenant des portions de code générales destinées à être couramment utilisées. La factorisation consiste à écrire le programme proprement dit en utilisant au mieux les bibliothèques disponibles.

L'optimisation et la factorisation possèdent néanmoins leurs logiques propres que nous allons présenter maintenant de façon succincte afin de donner une intuition de leur fonctionnement. Lorsque le stimulus (1 2 3 4 5 Fig.4.1) est perçu, il est stocké dans la partie Stimuli|Chunks sous sa forme la plus concise : la forme développée C1 C2 C3 C4 C5 comportant 5 caractères peut alors être avantageusement factorisée en C9 C7 comportant seulement 2 caractères. L'ajout de C9 C7 à la partie Stimuli|Chunks (Fig.4.1) fait alors passer à 4 le nombre de cooccurrences de C9 et C7. Il est donc possible d'optimiser la taille totale des deux parties en ajoutant le chunk $C10 = C9 + C7$ à la partie Chunks. Cela fait augmenter de 3 caractères la taille de la partie Chunks, mais fait diminuer de 4 caractères la taille de la partie Stimuli|Chunks, puisque les 4 cooccurrences C9 C7 sont remplacées par C10. La taille totale a diminué d'un caractère. Dans le modèle c'est la taille de codage et non le nombre de caractères qui guide la factorisation et l'optimisation. Il ne sera dans la suite plus jamais question de nombre de caractères. Cette présentation « incorrecte » avait pour seul but d'introduire les trois mécanismes qui vont être détaillés maintenant.

Taille de codage : Basée sur la théorie de l'information, la taille de codage est plus intéressante que le simple nombre de caractères, car elle intègre en plus la notion de fréquence. Par exemple, sur la figure 4.1, le chunk C6 apparaît 4 fois (2 dans la partie Stimuli|Chunks et 2 dans la partie Chunks) et le chunk C9 apparaît 8 fois (7 dans la partie Stimuli|Chunks et 1 dans la partie Chunks). C9 plus fréquent va avoir une taille de codage plus faible que C6, bien que tous deux soient représentés par un seul caractère.

La taille de codage $C(chunk)$ d'un chunk (en bits) dépend uniquement de sa fréquence $P(chunk)$, selon la relation :

$$C(chunk) = log_2\left(\frac{1}{P(chunk)}\right) \tag{4.12}$$

Pour l'exemple de la figure 4.1, il y a 40 occurrences de chunks (19 dans la partie Stimuli|Chunks et 21 dans la partie Chunks). La taille de codage de C6 est donc $C(C6) = log_2\left(\frac{40}{4}\right) = 3.3$ bits, et celle de C9 est $C(C9) = log_2\left(\frac{40}{8}\right) = 2.3$ bits. Pour plus de clarté, nous utiliserons dans la suite le terme « taille » plutôt que « taille de codage » [2] ainsi que « taille totale » plutôt que « taille totale de la réunion des deux parties Stimuli|Chunks et Chunks » .

Illustrons ce processus par un exemple concret. Les figures 4.2, 4.3 et 4.4 présentent l'évolution des tailles de codage au début de l'apprentissage.

Optimisation : L'optimisation est le coeur du modèle. C'est le mécanisme consistant à créer de nouveaux chunks lorsque cela conduit à une diminution de la taille totale. L'optimisation se sépare en deux sous-phases. La création du chunk proprement dit et la mise à jour de la partie Stimuli|Chunks avec le nouveau chunk. L'ajout figure 4.4 du dernier stimulus fait augmenter le nombre

2. Le lecteur déjà familier avec le formalisme utilisé, voudra bien créer le chunk : « taille = taille + de + codage » . (- ;

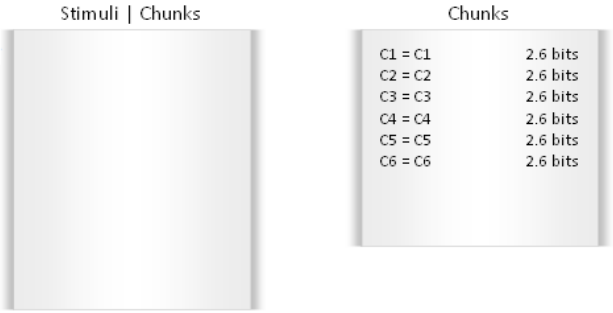


FIGURE 4.2 – Le système ne contient initialement que 6 chunks-canoniques de fréquences égales donc de tailles égales. La partie Stimuli|Chunks est vide car aucun stimuli n'a encore été perçu.

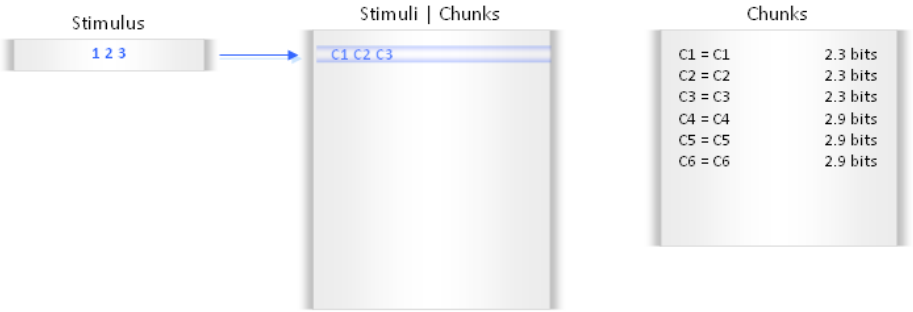


FIGURE 4.3 – Le premier stimulus est ajouté à la partie Stimuli|Chunks, modifiant les fréquences de C1, C2 et C3. Les tailles de codage correspondantes diminuent et celles des autres chunks augmentent car leur fréquence relative diminue.

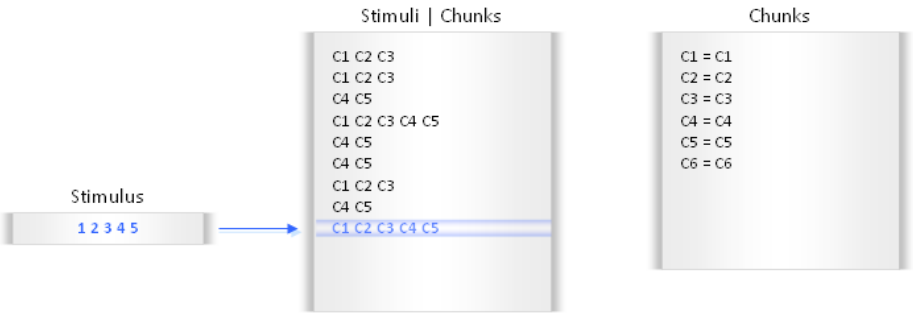


FIGURE 4.4 – Etat du système après la perception du neuvième stimulus. Aucun chunk n'a encore été créé. Les représentations utilisées dans la partie Stimuli|Chunks sont développées, car aucune factorisation n'est encore possible en l'absence de chunks.

d'apparitions conjointes de C4 et C5. La création du chunk correspondant est maintenant rentable et conduit à une diminution de la taille totale (Fig. 4.5).

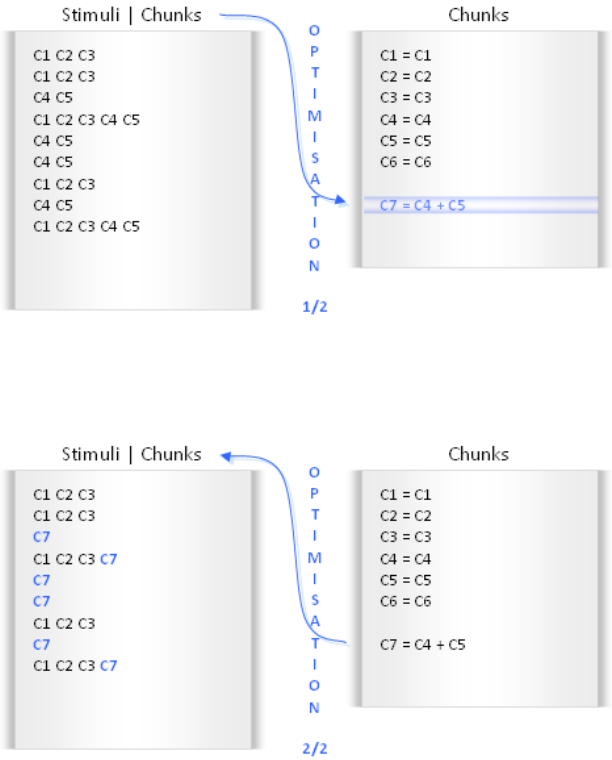


FIGURE 4.5 – La phase 1/2 correspond à la création du chunk C7 regroupant C4 et C5. La phase 2/2 correspond au remplacement de C4 et C5 par C7 dans la partie Stimuli|Chunks. Le gain de taille qui en découle est supérieur à la perte de taille due à l'ajout de $C7 = C4 + C5$ dans la partie Chunks.

Factorisation : La factorisation consiste à trouver la représentation la plus courte pour chaque stimuli, en utilisant les chunks de la partie Chunks. Jusqu'à présent seuls les chunks-canoniques étaient disponibles dans notre exemple. La forme factorisée était identique à la forme développée. La création du chunk C7 va permettre de factoriser le stimulus suivant (Fig. 4.6 page ci-contre).

Cette présentation informelle des différents mécanismes mis en jeu par le modèle a permis d'établir leurs rôles respectifs. La section suivante détaille leur fonctionnement interne.

4.2 Fonctionnement

Cette partie ne revient pas sur la description de l'architecture utilisée, le lecteur pourra se reporter à la section précédente ou au chapitre 3 pour plus de détail.

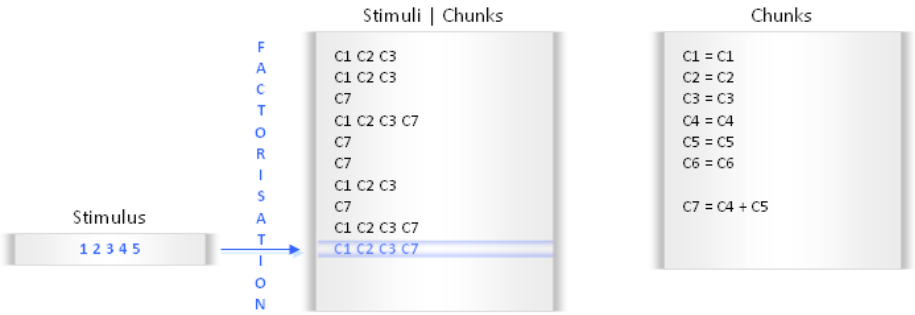


FIGURE 4.6 – Factorisation du stimulus 1 2 3 4 5. Deux factorisations sont possibles : C1 C2 C3 C4 C5 qui correspond à la forme développée, et C1 C2 C3 C7 qui utilise le chunk C7. La taille de codage de C7 étant plus faible que celle de $C4 + C5$ (tailles non représentées), c'est la seconde factorisation qui est choisie.

Cette section décrit de façon approfondie les algorithmes utilisés par les trois mécanismes constitutifs du modèle : le calcul des tailles de codage, la factorisation des stimuli et l'optimisation.

4.2.1 Tailles de codage

La taille de codage d'un chunk dépend uniquement de sa fréquence (Eq. 4.12 page 62). Cette fréquence est une simple division du nombre d'apparitions du chunk par le nombre total de chunks dans le système (Fig. 4.7 page suivante).

La taille de codage ainsi calculée correspond à celle qui pourrait être obtenue avec un codage optimal de type Shannon-Fano. Il n'est pas forcément possible en pratique d'obtenir un tel codage, bien que certains algorithmes de compression (l'algorithme d'Huffman (1952) ou ses raffinements) permettent de s'en approcher. Cela n'est pas utile dans notre cas, car la taille de codage est utilisée uniquement comme critère de sélection permettant de déterminer entre deux états du système, lequel a la plus faible taille.

4.2.2 Factorisation

Présentation du problème

Le mécanisme de factorisation consiste à représenter le stimulus courant sous sa forme la plus concise (en terme de taille de codage). Cette étape peut être vue comme une forme de compression du stimulus grâce aux chunks. Comme nous l'avons vu précédemment, bien qu'il soit toujours possible de représenter un stimulus uniquement à l'aide des chunks-canoniques, l'utilisation des chunks non-canoniques permet une diminution de la taille de codage (exemple figure 4.8 page suivante). Nous conserverons cet exemple tout au long de la section.

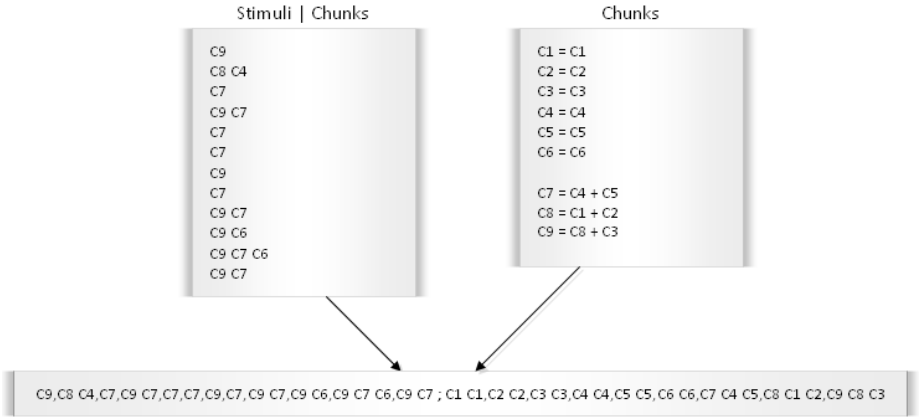


FIGURE 4.7 – Le système total constitué des deux parties Stimuli|Chunks et Chunks est mis sous forme d'une simple chaîne de caractères (cela n'entraîne aucune perte d'information). La fréquence d'un chunk est alors donnée par son nombre d'apparitions divisé par la taille de la chaîne. Par exemple, $P(C4) = \frac{4}{40}$ pour une taille de codage de $C(C4) = log_2\left(\frac{40}{4}\right) = 3.3$ bits. La taille totale du système est la somme des tailles de codages de tous les chunks de la chaîne : $C(C9) + C(C8) + C(C4) + \cdots + C(C8) + C(C3) = 120$ bits.

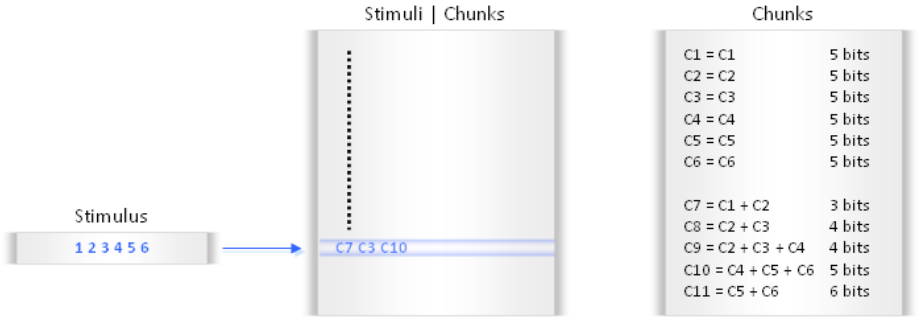


FIGURE 4.8 – Exemple de factorisation. La partie Chunks comporte 6 chunks-canoniques et 5 chunks non-canoniques. Les tailles de codage des chunks sont indiquées à droite. Pour simplifier la figure, les stimuli déjà factorisés ne sont pas représentés. Le stimulus courant (1 2 3 4 5 6) peut être trivialement représenté C1 C2 C3 C4 C5 C6 pour une taille de 6×5 bits= 30 bits, mais une factorisation sous la forme C7 C3 C10 permet d'obtenir une taille de codage plus faible : 3 bits +5 bits +5 bits = 13 bits.

Exprimer un stimulus en terme de chunks-canoniques est immédiat car à chaque caractère du stimulus (1, 2, 3, ...) correspond un et un seul chunk-canonique (C1, C2, C3, ...). Le problème devient plus difficile lorsque la factorisation met en jeu des chunks non-canoniques, car ces derniers ne sont pas nécessairement disjoints. La factorisation d'un stimulus n'est alors plus unique. Dans l'exemple (Fig.4.8), C9 et C10 ne sont pas disjoints (ils possèdent tous deux C4), choisir

d'utiliser l'un des deux pour factoriser le stimulus exclut l'utilisation de l'autre [3].

D'une façon générale, différentes factorisations possèdent différentes tailles de codages. Le mécanisme de factorisation a pour rôle de trouver la meilleure. Sur l'exemple de la figure 4.8 le stimulus aurait pu être représenté par C7 C3 C4 C11 (19 bits) ou encore par C1 C9 C11 (15 bits), mais c'est C7 C3 C10 (13 bits) qui est retenue. Sur cet exemple simple, il est envisageable de produire toutes les factorisations possibles, puis de choisir la plus courte. Mais dans un cas plus complexe, l'espace de recherche devient vite trop important. Une exploration exhaustive de type recherche en largeur d'abord (*breadth-first search*) (Pearl, 1984), n'est plus possible. Il faut alors se limiter à une exploration partielle de l'espace des factorisation possibles.

Pour cela, nous utilisons un algorithme de recherche par meilleur d'abord (*best-first search*) (Pearl, 1984) avec retours arrière (*backtracking*). Les figures 4.9, 4.10 et 4.11 page suivante illustrent la différence entre les deux algorithmes et l'intérêt du retour arrière.

Dans notre cas, une recherche en meilleur d'abord sans retours arrière serait obtenue en choisissant systématiquement le chunk de plus forte utilité (plus faible taille). Cela n'assure pas d'obtenir la meilleure factorisation. Si C9 (4 bits) est choisi à la place de C10 (5 bits), la factorisation optimale C7 C3 C10 ne peut plus être trouvée. La limitation provient du fait qu'en retenant un chunk particulier (C9 par exemple) cela exclut l'utilisation des chunks non-disjoints (C7, C8 et C10).

Pour pallier à ce problème, la factorisation effectue une recherche en meilleur d'abord avec retours arrière. L'algorithme choisi est de type A* (Pearl, 1984). Cela permet d'obtenir une solution acceptable en effectuant une exploration seulement partielle de l'espace de recherche.

Algorithme A*

A* est un algorithme de recherche de chemin le plus court entre deux noeuds d'un graphe. L'idée qui sous-tend cet algorithme est de diminuer l'espace d'exploration en développant en priorité les noeuds de plus forte utilité (meilleur d'abord). Il s'agit d'une recherche guidée par la distance au but. Comme la distance au but ne peut être connue avec précision puisque le chemin qui y mène n'est pas connu, il faut en fournir une estimation. La performance de l'algorithme dépend de la précision de cette estimation. À l'extrême, assigner un nombre aléatoire à chaque distance au but revient à chercher le but en aveugle. La première factorisation trouvée ne sera en moyenne pas plus performante qu'une factorisation au hasard. Un noeud se voit affecté d'une utilité élevée si la distance déjà parcourue plus l'estimation de la distance restant à parcourir est faible (figures 4.12 et 4.13 page 69).

3. Avec l'addition choisie, $C_i + C_i = 2C_i$, donc C2 C3 C4 + C4 C5 C6 = C2 C3 2×C4 C5 C6, qui n'est pas le résultat souhaité. Ce choix d'addition est naturel pour la représentation choisie, mais un choix différent est possible, par exemple $C_i + C_i = C_i$ mieux adaptée à la représentation par attributs/valeurs.

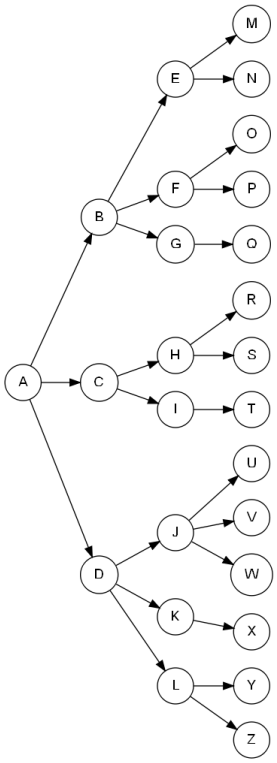


FIGURE 4.9 – Exemple de recherche en largeur d'abord sur une profondeur de 3 niveaux. Toutes les solutions possibles sont envisagées. Puis la meilleure est choisie (X par exemple).

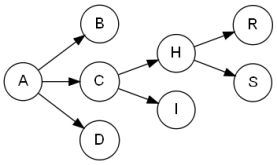


FIGURE 4.10 – Exemple de recherche par meilleur d'abord sur les 3 niveaux de la figure précédente. Le noeud le plus intéressant est systématiquement développé : A puis C puis H puis S. Seule une partie de l'espace de recherche est explorée, mais la solution S trouvée est sous-optimale.

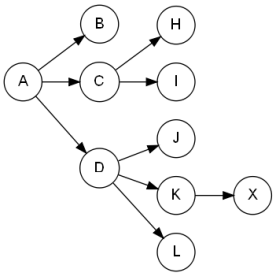


FIGURE 4.11 – Exemple de recherche par meilleur d'abord avec retours arrière possibles. Les retours arrière permettent une meilleure exploration de l'espace de recherche qu'en 4.10. C est d'abord développé, puis D (retour) car il est plus intéressant que H ou I, puis enfin K puis X. La solution X finalement trouvée est meilleure que S trouvée précédemment, sans que l'exploration soit exhaustive.

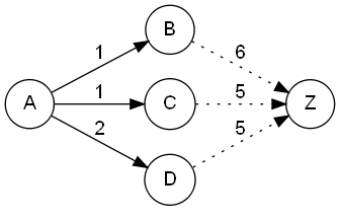


FIGURE 4.12 – Recherche du chemin le plus court entre le noeud A (départ) et le noeud Z (but). Les arcs pleins représentent la distance déjà parcourue et les arcs pointillés une estimation de la distance restant à parcourir. Le noeud C a la plus forte utilité car la plus faible distance totale (*parcourue*+*restante* = 1+5 = 6).

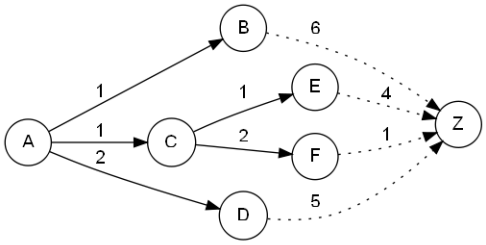


FIGURE 4.13 – Le chemin est ensuite développé à partir du noeud le plus intéressant (noeud C). Le noeud de plus forte utilité est maintenant F (distance parcourue : 1 + 2 = 3 ; estimation de la distance restant à parcourir : 1). L'algorithme continue ainsi jusqu'à ce que le but soit atteint.

Le départ de la recherche est une liste vide, et le but est la représentation canonique du stimulus à factoriser (C1 C2 C3 C4 C5 C6 pour l'exemple de la figure 4.8 page 66). Chaque nouveau noeud correspond à l'ajout d'un chunk, ainsi de suite jusqu'au but (voir figure 4.14 page 71 pour un exemple de l'application de A* à notre problème). La distance parcourue correspond tout naturellement aux tailles de codage des chunks utilisés. Toute la difficulté consiste à fournir une estimation correcte de la distance au but. L'optimalité (admissibilité) de l'algorithme A* n'est garantie que s'il est possible de minimiser cette distance au but (Pearl, 1984). Mais les performances de l'algorithme dépendent de la qualité du minorant trouvé. Un mauvais minorant (zéro par exemple) revient à ne pas utiliser le but pour guider la recherche. Il s'agit dans ce cas d'une recherche en largeur d'abord qui revient à envisager toutes les factorisations possibles. Pour la situation qui nous occupe, il n'existe pas de minorant évident de la distance au but. En revanche, l'utilisation des chunks-canoniques en fournit une estimation [4] relativement naturelle. Il est en effet toujours possible de rejoindre le but à partir de n'importe quel noeud en utilisant uniquement les chunks-canoniques. Le mérite de cette approche est que chaque noeud représente une solution viable. Le chemin du départ jusqu'au noeud est composé de chunks, canoniques ou non, le chemin restant étant composé uniquement de chunks-canoniques.

Le choix d'utiliser une estimation de la distance au but qui ne soit pas un minorant rend l'algorithme plus rapide mais sous-optimal. Afin d'augmenter les chances d'approcher la solution optimale, la condition d'arrêt n'est pas de stopper dès que le but est atteint, mais de poursuivre l'exploration tant que la place

4. L'estimation peut être vue comme un majorant de la distance au but, car la solution réellement trouvée, en mettant en jeu des chunks non-canoniques ne pourra être que meilleure que l'estimation qui est restreinte à l'utilisation des chunks-canoniques.

mémoire occupée ne dépasse pas un seuil fixé. Une autre condition d'arrêt souvent utilisée est de limiter le temps de calcul. Il s'agit dans les deux cas d'un moyen artificiel permettant d'explorer une portion plus vaste de l'espace de recherche. La portion de l'espace exploré est fonction de la contrainte imposée sur le temps de calcul : une contrainte forte est équivalente à une simple recherche en meilleur d'abord guidée par le but, tandis qu'une absence de contrainte est équivalente à une recherche exhaustive.

Le lecteur attentif pourrait objecter de façon très juste que l'ajout ou non du retour ne modifie en rien le résultat dans la mesure ou l'algorithme ne revient jamais en arrière. Puisque l'estimation de la distance au but est en quelque sorte un majorant, l'utilité d'une branche ne peut qu'augmenter en la développant. Cette remarque n'est plus valable lors de l'ajout de la négation (NON) qui sera discuté section 4.3.2. L'algorithme est présenté ici de façon à posséder toute la généralité nécessaire aux améliorations présentées plus loin.

Afin de mieux comprendre le fonctionnement de l'algorithme, la section suivante détaille les étapes de factorisation du stimulus (1 2 3 4 5 6) de l'exemple de départ (Fig. 4.8 page 66). Pour simplifier la lecture des schémas, les chunks ne sont pas représentés par leur nom (C9), mais par leur définition (C2C3C4).

Exemple

Le rôle de la factorisation est de chercher la meilleure solution entre le départ (vide) et le but à représenter (C1 C2 C3 C4 C5). Cette recherche s'effectue en développant à chaque itération le noeud de plus forte utilité. Pour cela, l'algorithme applique tous les chunks existants à la fois inclus dans le but et disjoints du noeud courant.

L'exemple présenté est celui de la figure 4.8 page 66. Il existe 5 chunks non-canoniques dont les tailles de codage sont : $C(C1C2)$ = 3 bits, $C(C2C3)$ = $C(C2C3C4)$ = 4 bits, $C(C4C5C6)$ = 5 bits et $C(C5C6)$ = 6 bits ; ainsi que 6 chunks-canoniques de 5 bits. Le chunk correspondant à chaque noeud est noté entre crochets. Le coût du noeud est indiqué en précisant à chaque fois quelle est la distance déjà parcourue (premier terme) et l'estimation de la distance restante (second terme). Par exemple, C2C3C4 coûte 4 bits, et la distance restante si l'on n'utilise que les chunks-canoniques est C1 C5 C6 = 15 bits, pour un total de 19 bits. Afin de simplifier les schémas, certains noeuds ont été volontairement omis.

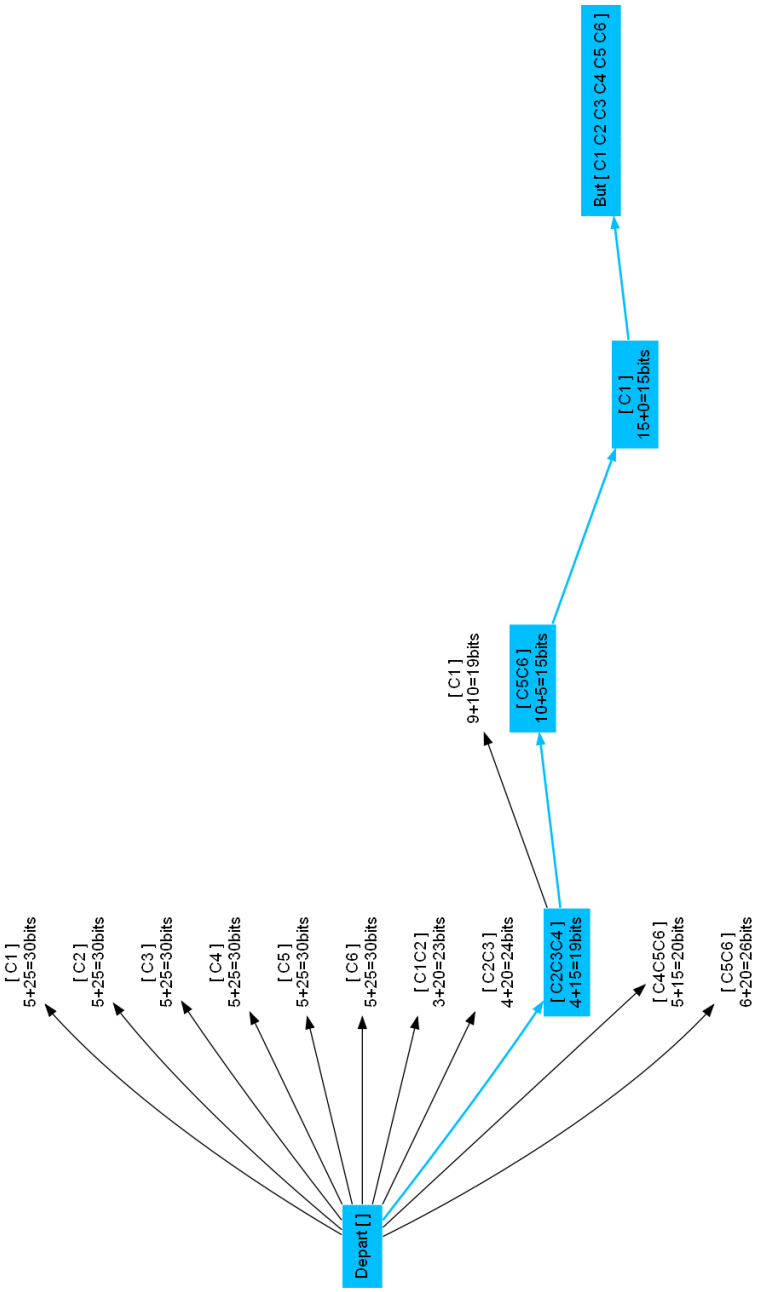


FIGURE 4.14 – Première factorisation trouvée par A*. À la première itération, les 11 chunks sont appliqués car tous sont contenus dans le but et disjoints du départ (vide). Le meilleur noeud est [C2C3C4] avec 19 bits. Il est développé. Le meilleur est maintenant [C5C6] (15 bits) puis enfin [C1] (15 bits). La factorisation trouvée est surlignée [C2C3C4 C5C6 C1] et coûte 15 bits.

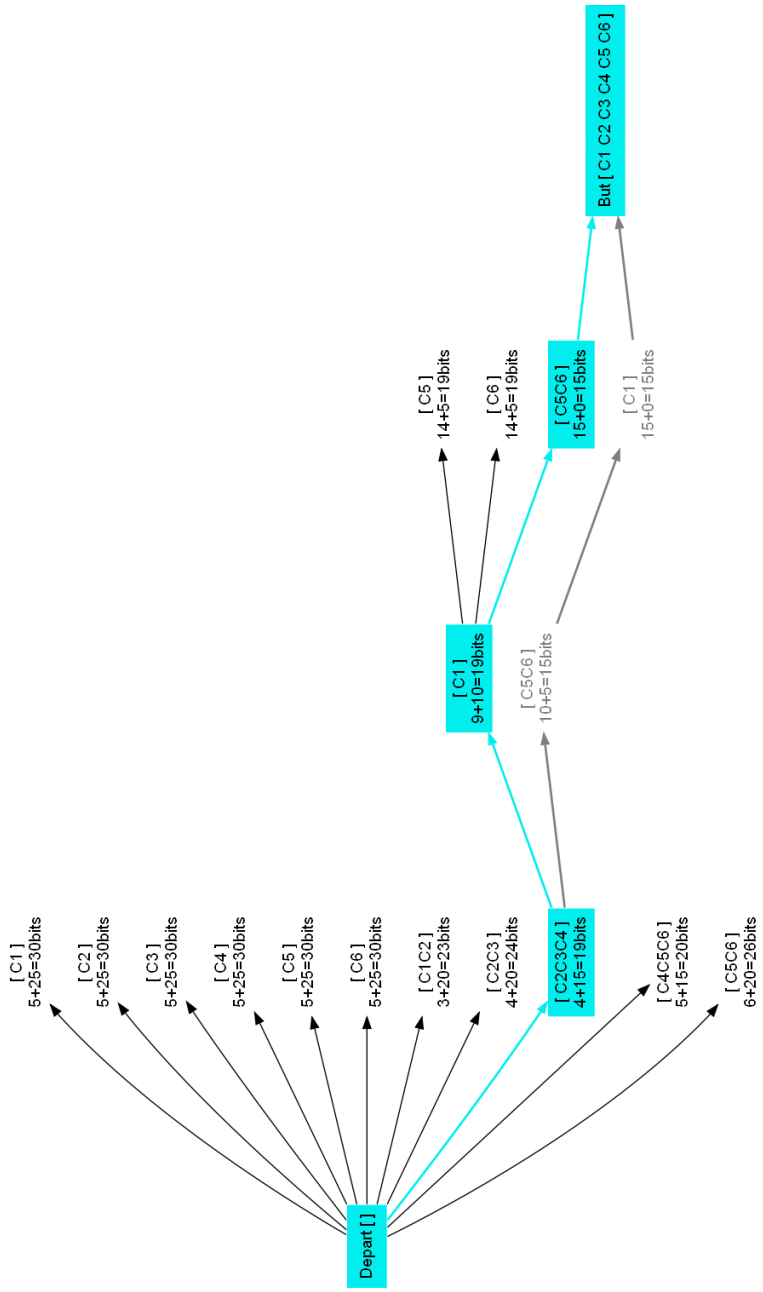


FIGURE 4.15 – Seconde factorisation trouvée par A*. La solution précédente est grisée. Sur la figure précédente, le meilleur chemin développable est [C2C3C4 C1] avec 19 bits. Il est développé ainsi que [C5C6] qui est le meilleur des trois nouveaux noeuds. La factorisation trouvée est surlignée [C2C3C4 C1 C5C6], elle est identique à la précédente (seul l'ordre change).

FIGURE 4.16 – Les factorisations 3 et 4 sont surlignées. Leur coût est de 19 bits. Les flèches pointillées indiquent l'existence de noeuds intermédiaires non-représentés.

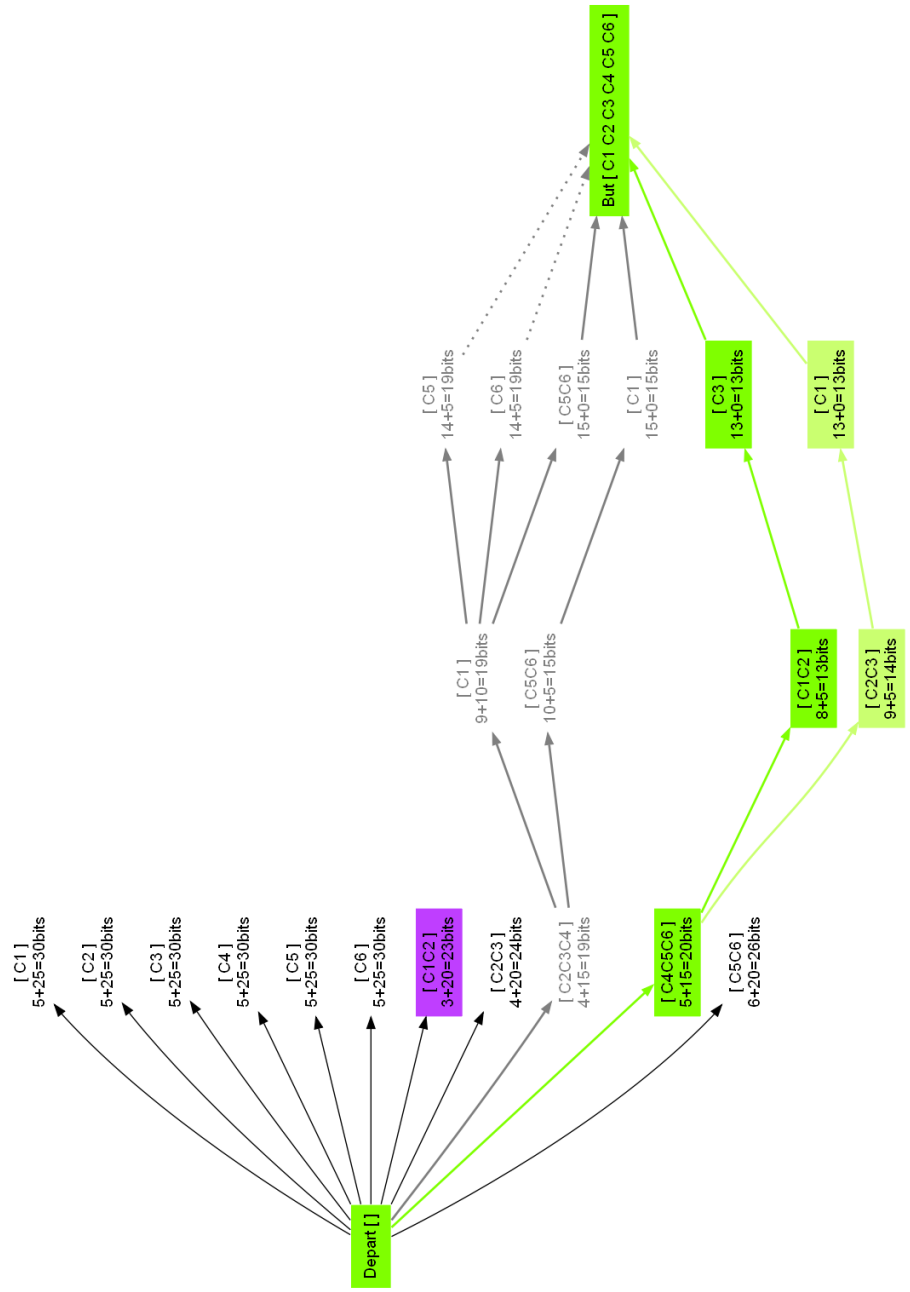


FIGURE 4.17 – Les factorisation 5 et 6 sont les deux meilleures : [C4C5C6 C1C2 C3] et [C4C5C6 C2C3 C1] (en vert). Elles sont toutes les deux optimales et valent 13 bits, tandis que les quatre premières solutions envisagées valaient respectivement 15 bits, 15 bits, 19 bits et 19 bits. L'algorithme poursuit ainsi jusqu'à ce que la condition d'arrêt soit atteinte. Le noeud suivant à être développé est [C1C2].

Conclusion sur la factorisation

Le mécanisme de factorisation choisi est basé sur l'algorithme A*. Il pourrait être remplacé par n'importe quel algorithme remplissant la même fonction (recuit simulé par exemple) sans que cela affecte le reste du modèle. Les choix effectués concernant l'estimation de la distance au but ne garantissent pas l'optimalité des solutions trouvées, mais réduisent suffisamment l'espace de recherche pour obtenir des temps de calcul raisonnables. Nous présenterons dans le chapitre suivant une version simplifiée de cet algorithme de factorisation, qui au prix d'une légère dégradation des performances, permet une diminution substantielle de la quantité de calculs.

4.2.3 Optimisation

L'optimisation est le mécanisme qui permet d'extraire les régularités présentes dans les stimuli. Une régularité est intéressante si elle permet une diminution de la taille de codage de l'ensemble du système (Stimuli|Chunks + Chunks). La performance de la compression dépend de la nature des régularités extraites. Dans le MDLChunker, les régularités envisagées sont de type ET (conjonction de chunks apparaissant fréquemment de façon conjointe). Ce sont les chunks créés dans la partie Chunks.

Il est possible de décrire le mécanisme d'optimisation de façon indépendante du type de régularités recherchées. On cherche l'ensemble des états adjacents, c'est-à-dire les états qui peuvent être atteint par une transformation élémentaire à partir de l'état actuel du système. Le critère de sélection du meilleur état adjacent est le MDL (taille totale du système). L'état dont la taille de codage totale est la plus faible, c'est-à-dire celui permettant la meilleure compression, se voit assigné la plus forte utilité. Le meilleur état adjacent devient alors l'état courant et le processus continue tant que l'utilité d'au moins un des états adjacent est supérieure à celle de l'état courant. (Fig. 4.18).

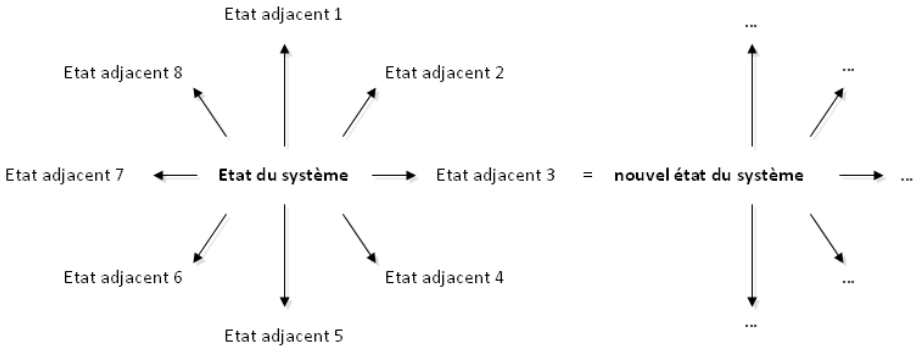


FIGURE 4.18 – Exploration des états adjacents. L'état adjacent de plus forte utilité (n°3) devient le nouvel état du système.

Dans le cas qui nous occupe, une transformation élémentaire correspond à la

création d'un chunk. La suppression de chunks n'est pas implémentée, mais les mécanismes sont les mêmes que pour la création. Le nombre d'états adjacents à envisager est donc égal au nombre de chunks qui peuvent être créés. Si les chunks envisagés sont n-aires, leur nombre devient vite trop important. Un stimulus dont la forme factorisée est de longueur n contient potentiellement

$$\sum_{i=2}^{n} C_n^i = 2^n - n - 1 \tag{4.13}$$

chunks n-aires. Par exemple $C2C4C9$ ($n = 3$) contient 4 chunks susceptibles d'être créés : $C2C4$, $C2C9$, $C4C9$ et $C2C4C9$. Tandis que pour $n = 20$, il y a déjà plus d'un million d'états adjacents possibles.

Pour une raison évidente de dimension de l'espace de recherche, nous nous limiterons dans la suite à la création de chunks binaires (Fig. 4.19). Les chunks n-aires restent représentables comme somme de chunks binaires ($C10 = C1 + C2 + C3$ peut être exprimé sous la forme $C10 = C1 + C2$ et $C11 = C10 + C3$).

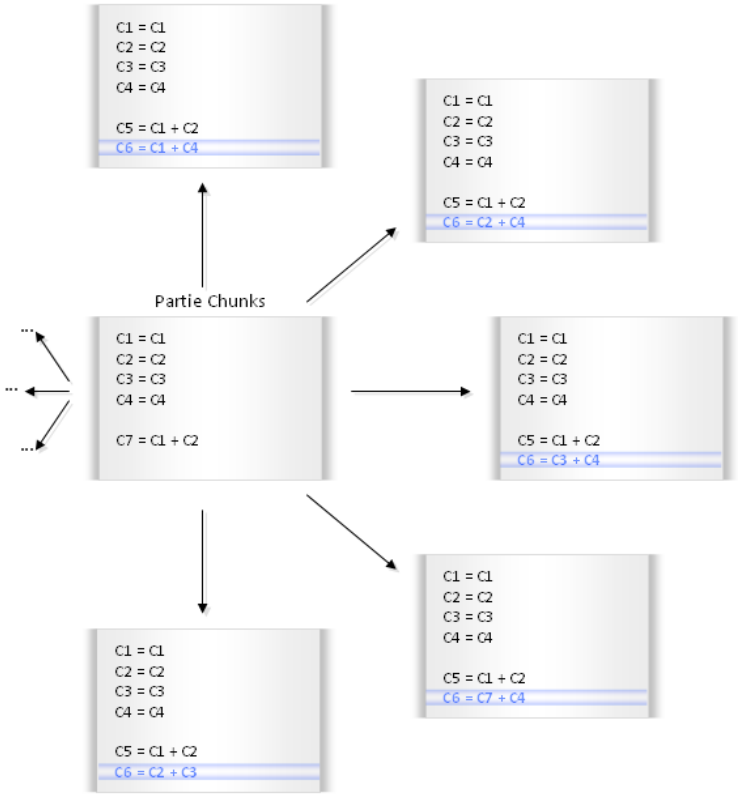


FIGURE 4.19 – Exploration des états adjacents dans le cas de chunks binaires.

L'approche décrite ci-dessus est sous-optimale pour deux raisons. La première est due à l'utilisation de chunks binaires uniquement. Un chunk ternaire par exemple ne peut être créé que si au moins l'un des chunks binaires qu'il contient l'a été. Dans le cas où l'utilité du chunk ternaire est positive sans que celle de ses chunks binaires le soit, le chunk ternaire ne pourra être créé. La seconde

cause de sous-optimalité provient du mode d'exploration des états adjacents. L'algorithme décrit choisit systématiquement l'état de plus forte utilité sans possibilité de retour en arrière. Nous avons déjà vu dans la partie factorisation que ce type d'exploration est sujette à converger vers un optimum qui soit local.

L'optimisation se fait en trois phases. La recherche des chunks binaires susceptibles d'être créés (états adjacents). Puis pour chacune de ces paires, le calcul du gain qu'induirait la création du chunk correspondant (utilité de l'état adjacent). Et enfin la création du meilleur chunk et mise à jour de la partie Stimuli|Chunks (l'état adjacent de plus forte utilité devient l'état courant).

Recherche des cooccurrences

Pour savoir quels chunks binaires sont susceptibles d'être créés, il suffit de parcourir chaque ligne de la partie Stimuli|Chunks en listant les paires de chunks qui sont apparues conjointement (cooccurrences). En reprenant l'exemple introductif (Fig. 4.5 page 64), les cooccurrences sont :

Paire de chunks	Nombre de cooccurrences	Gain de taille de codage
C1-C2	5	-0.6 bits
C1-C3	5	-0.6 bits
C2-C3	5	-0.6 bits
C1-C4	2	-7.3 bits
C1-C5	2	-7.3 bits
C2-C4	2	-7.3 bits
C2-C5	2	-7.3 bits
C3-C4	2	-7.3 bits
C3-C5	2	-7.3 bits
C4-C5	6	+0.8 bits

TABLE 4.1 – Nombre de cooccurrences et gain induit par la création du chunk correspondant pour la partie Stimuli|Chunks de l'exemple introductif.

Calcul du gain

Le calcul du gain est effectué pour chaque paire de chunks. Cette opération doit donc être rapide car il n'est pas envisageable de créer réellement chaque chunk afin de regarder si cela conduit à une augmentation ou à une diminution de la taille totale du système. Les différents états adjacents ne sont jamais construits, on se contente de calculer leur utilité (gain). Cette dernière peut être calculée de façon exacte.

Si $\#A$ est le nombre d'occurrences d'un chunk A, $\#B$ le nombre d'occurrences d'un chunk B, et $\#AB$ le nombre de cooccurrences de A et B, alors de gain de taille de codage qu'induirait la création du chunk $C = A + B$ est donné par l'équation 4.14 page suivante. Pour simplifier, on introduit les notations $\#A\bar{B} = \#A - \#AB$ (nombre d'occurrences de A sans B), $\#\bar{A}B = \#B - \#AB$

(nombre d'occurrences de B sans A) et $\#\bar{A}\bar{B} = \#N - \#A - \#B + \#AB$ (nombre d'occurrences d'autres chunks que A et B). $\#N$ est le nombre total d'occurrences de chunks dans le système.

$$
\begin{aligned}
Gain = & - \#AB \cdot \left[log_2 \left(\frac{\#N + 1 + 2 - \#AB}{\#AB + 1} \right) - log_2 \left(\frac{\#N}{\#A} \right) - log_2 \left(\frac{\#N}{\#B} \right) \right] \\
& - \#A\bar{B} \cdot \left[log_2 \left(\frac{\#N + 1 + 2 - \#AB}{\#A\bar{B} + 1} \right) - log_2 \left(\frac{\#N}{\#A} \right) \right] \\
& - \#\bar{A}B \cdot \left[log_2 \left(\frac{\#N + 1 + 2 - \#AB}{\#\bar{A}B + 1} \right) - log_2 \left(\frac{\#N}{\#B} \right) \right] \\
& - log_2 \left(\frac{\#N + 1 + 2 - \#AB}{\#AB + 1} \right) - log_2 \left(\frac{\#N + 1 + 2 - \#AB}{\#A\bar{B} + 1} \right) - log_2 \left(\frac{\#N + 1 + 2 - \#AB}{\#\bar{A}B + 1} \right) \\
& - \#\bar{A}\bar{B} \cdot log_2 \left(\frac{\#N + 1 + 2 - \#AB}{\#N} \right)
\end{aligned}
\tag{4.14}
$$

Les différents termes de cette équation sont expliqués ci-après.

$\mathbf{\#N + 1 + 2 - \#AB}$ **:** Il s'agit du nombre total d'occurrences de chunks après création du nouveaux chunk $C = A + B$. Il y a une occurrence supplémentaire, celle du nouveau chunk C et deux occurrences supplémentaires pour sa définition $A + B$. Les $\#AB$ cooccurrences de A et B sont remplacées par C, supprimant ainsi $\#AB$ occurrences.

$\mathbf{log_2 \left(\frac{\#N}{\#A} \right)}$ **:** Ancienne taille de codage de A.

$\mathbf{log_2 \left(\frac{\#N+1+2-\#AB}{\#A\bar{B}+1} \right)}$ **:** Nouvelle taille de codage de A.

Ligne 1 : Valeur toujours positive. Gain de taille de codage due au remplacement de A et B par C dans la partie Stimuli|Chunks.

Ligne 2 : Valeur toujours négative. Représente l'augmentation de taille de codage due au fait que les occurrences de A sans B vont maintenant avoir une taille plus importante. Comme toutes les occurrences de A avec B sont remplacées par C, le nombre d'occurrences de A a diminué, ce qui a fait augmenter sa taille de codage.

Ligne 3 : Même chose que la ligne 2, mais pour B.

Ligne 4 : Valeur toujours négative. Taille de codage nécessaire pour définir le nouveau chunk $C = A + B$.

Ligne 5 : Valeur toujours positive. Diminution de la taille de codage de tous les autres chunks du système due à l'augmentation de leur fréquence relative puisque le nombre total de chunks a diminué.

Le gain de taille de codage obtenu avec l'équation 4.14 pour chaque paire de chunk est donné dans le tableau 4.1 page 77. Le chunk auquel correspond le plus fort gain positif est alors créé (ici $C6 = C4 + C5$).

Mise à jour de la partie Stimuli|Chunks

D'un point de vue algorithmique, cette étape est instantanée. Toutes les cooccurrences de A et B sont remplacées par C.

Dans notre exemple, avant la création du chunk C6, la taille du système est de 67.4 bits ($Stimuli \mid Chunks$ = 44.8 bits et $Chunks$ = 22.6 bits). Après la création de C6, la taille du système est de 66.6 bits ($Stimuli \mid Chunks$ = 34.4 bits et $Chunks$ = 32.2 bits). On a bien la diminution de 0.8 bits prévue. La taille de la partie Chunk a augmenté puisqu'un chunk a été créé. Cette augmentation est néanmoins plus faible que la diminution correspondante de taille de la partie Stimuli|Chunks après remplacement des cooccurrences de C4 et C5 par C6.

4.2.4 Exemple de fonctionnement

4.3 Améliorations

Les deux améliorations qui sont présentées ici n'ont pas été décrites avant pour ne pas compliquer inutilement la description du modèle. Leur ajout n'entraîne que des modifications mineures.

Pour les besoins de certaines expériences, il est nécessaire d'apporter deux améliorations au modèle. La première est l'introduction d'un ordre entre les chunks lorsque la notion d'ordre existe dans les stimuli (jusqu'à présent les stimuli sont des paquets de chunks non-ordonnés). Typiquement pour les données linguistiques, comme nous le verrons dans le chapitre 8 qui applique le modèle à la segmentation de mots, il est naturel de décrire les mots comme une suite ordonnée de lettres. La seconde amélioration consiste à introduire la négation de chunks ('un tabouret est une chaise sans dossier'). Un chunk C (tabouret) peut ainsi être décrit comme un chunk plus grand A (chaise), auquel on retranche un chunk B (dossier) ($C = A - B$).

4.3.1 Ordre

Différentes notions d'ordre

Une même chaîne de caractères ordonnés « A B C » peut être représentée de différentes manières : « A » suivi de « B » suivi de « C » ; « A » est avant « B » et avant « C » , et « B » est avant « C » ; « C » est à une distance de 2 après « A » , et « B » est entre « A » et « C » ; « A » est en première position dans la chaîne, « B » en seconde et « C » en troisième, etc. À chaque représentation correspond un type de régularité que l'on peut extraire.

Dans notre cas, la notion d'ordre est nécessaire afin de pouvoir appliquer le modèle à la segmentation de mots (cf. chapitre 8). La première représentation est parfaitement adaptée pour cette tâche. L'ordre est défini entre paires de caractères adjacents uniquement. Un tout autre choix serait à faire si l'on souhaitait par exemple pouvoir extraire des association préfixes/suffixes au sein des mots (associations de type A .+ B) [5]. Choisir cette définition de l'ordre plutôt qu'une autre n'est pas une limitation du modèle puisque ce dernier n'a pas vocation à extraire tous les types de régularités, celles-ci sont liées à l'ordre comme les autres.

Modification des algorithmes

En introduisant l'ordre, on restreint grandement l'espace de recherche qu'il est nécessaire d'explorer. Cela simplifie à la fois l'optimisation et la factorisation d'un facteur $n!$ (ou n est la longueur du stimulus).

Optimisation : L'optimisation se trouve simplifiée puisque seule les paires de chunks contiguës peuvent donner naissance à un nouveau chunk. Pour un stimulus de taille n il n'y a plus que $n-1$ chunks qui peuvent être créés (au lieu de C_n^2 dans le cas non-ordonné).

Factorisation : Lors de la factorisation d'un stimulus ordonné, l'ajout des différents chunks se fait dans le même ordre que le but. Le facteur de branchement est donc beaucoup plus faible. Lors de la factorisation de $C1C2C3C4C5C6$ une branche ayant déjà construit $C1C2C3$ pourra envisager $C4$ ou encore $C4C5C6$, mais pas $C5$ ni $C6$. La figure 4.20 page suivante donne l'ensemble des factorisations possibles pour l'exemple 4.8 page 66.

5. Association du préfixe A avec le suffixe B. L'expression régulière « .+ » représente n'importe quelle chaîne de caractères de longueur non-nulle.

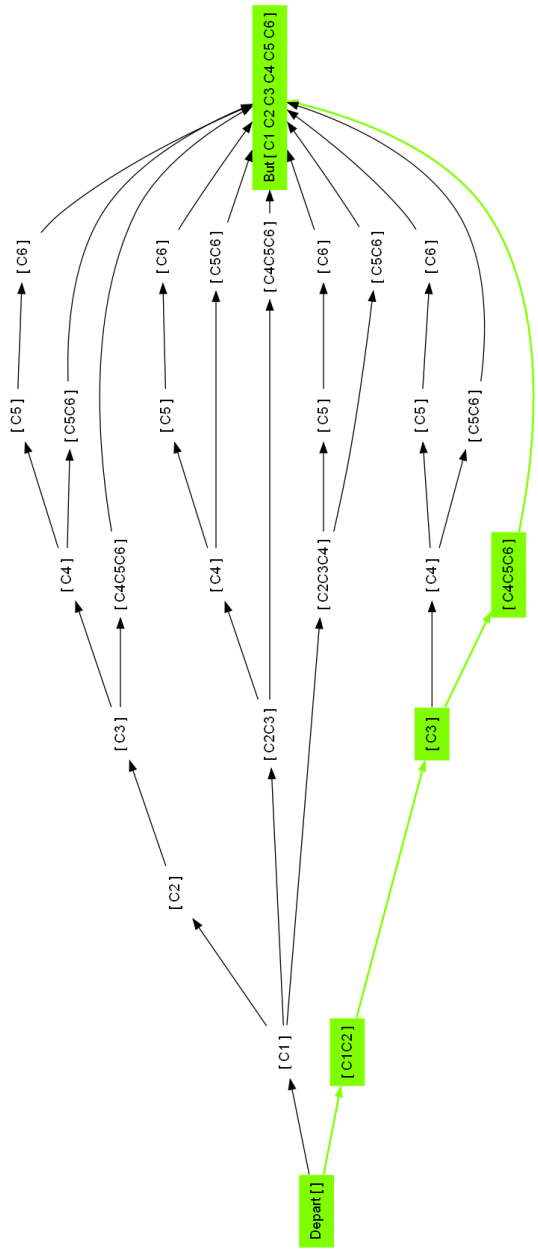


FIGURE 4.20 – Ensemble des chemins possibles du départ vers le but pour une factorisation avec ordre. Toutes les solutions peuvent être tracées car le facteur de branchement est très faible. La solution optimale $C1C2C3C4C5C6$ est surlignée.

4.3.2 Négation

La négation se fait en ajoutant à chaque chunk-canonique existant (C1, C2, C3) son pendant négatif (-C1, -C2, -C3) ; puis en définissant la règle suivante :

$$\forall i, C_i + (-C_i) = \varnothing \tag{4.15}$$

Pour chaque chunk créé par l'étape d'optimisation ($C_i = C_j + C_k$), on crée également sa négation ($-C_i = -C_j - C_k$). La factorisation demeure inchangée.

Nous ne motiverons pas ici l'intérêt de ces deux améliorations. Cela sera expliqué dans les chapitres correspondants de la partie expériences.

4.4 Discussion

4.4.1 Conséquences de l'implémentation choisie

Cette partie a pour but d'évaluer les conséquences implicites des choix réalisés dans l'implémentation. Il s'agit d'établir une liste des effets de bord inévitables liés à l'implémentation retenue : c'est-à-dire ce qui n'est pas contenu dans le principe d'association du MDL et du *chunking* et qui pourrait avoir une incidence sur les résultats obtenus. Les différents points sont listés ci-dessous sans transition.

Partie Stimuli|Chunks

Tous les stimuli perçus sont stockés sans perte d'information dans la partie Stimuli|Chunks. Cela suppose une mémoire infinie, ce qui est discutable d'un point de vue cognitif. Dans certains cas cette hypothèse d'une mémoire infinie est trop forte, le modèle doit alors être adapté en y ajoutant un paramètre limitant la taille de la mémoire. Nous verrons sur l'exemple du chapitre 8 comment cette taille de mémoire peut être ajustée pour réaliser une tâche de segmentation de mots.

Ce qui pose problème d'un point de vue de la plausibilité cognitive n'est pas tant la taille de mémoire infinie que le fait que les représentations ne soient pas fixes. La forme factorisée d'un stimulus est susceptible d'évoluer au cours du temps puisqu'à chaque optimisation la partie Stimuli|Chunks est mise à jour. Cela suppose implicitement que chaque création d'un nouveau chunk a non seulement des répercussions sur les représentations futures, mais également sur les représentations passées. Nous verrons qu'il est possible de retrouver des résultats similaires sans qu'il soit nécessaire de mettre à jour les représentations passées et sans supposer l'existence d'une mémoire infinie. C'est l'objet du chapitre suivant.

Chunks binaires

L'utilisation de chunks binaires n'est pas un paramètre intrinsèque au modèle. Il s'agit d'une astuce d'implémentation pour limiter l'espace des optimisations possibles. La représentation reste générale car des chunks n-aires peuvent être envisagés comme sommes de chunks binaires. La taille de codage n'est cependant pas la même car des chunks intermédiaires sont nécessaires. Dans le cas d'un chunk ternaire $C11 = C2 + C2 + C3$, représentable par $C10 = C1 + C2$ et $C11 = C10 + C3$, cela ajoute deux occurrences du chunk C10. Il serait techniquement possible de fusionner a posteriori les différents chunks binaires pour former un unique chunk n-aire lorsque cela diminue la taille de codage. Mais même sous cette forme, la difficulté évoquée dans la section 4.2.3 persiste. Le processus d'optimisation n'est en mesure de créer un chunks n-aire que si au moins un de ses chunks (n-1)-aires a déjà été créé. Il n'est pas possible d'envisager les rares cas ou un chunk n-aire a une utilité positive alors que tous ses chunks (n-1)-aires possèdent une utilité négative.

Suppression de chunks

Le MDLChunker permet la création de chunks, mais leur suppression n'est pas envisagée. Pour que la suppression d'un chunk soit intéressante, il faut que la taille nécessaire pour définir le chunk soit supérieure à celle gagnée par son utilisation. Cette situation n'est possible que si lors de la factorisation le chunk est délaissé au profit d'un autre plus court qui n'existait pas lors de sa création.

Ajouter un mécanisme de suppression des chunks ne pose en théorie aucun problème, mais cela est susceptible d'introduire des incohérences si le chunk à supprimer est utilisé dans la définition d'autres chunks. La taille de codage des différents chunks nous a semblé être un indicateur de leur utilité suffisamment performant pour qu'il ne soit pas nécessaire de supprimer explicitement les chunks peu utilisés.

Algorithme A*

L'utilisation d'un algorithme de type A* permet une exploration plus vaste de l'espace des factorisations possibles. En pratique l'exploration est assez vaste pour que l'on puisse considérer la solution trouvée comme optimale. On peut en revanche s'interroger sur la capacité des humains à effectuer une exploration aussi large. Il est probable que le MDLChunker soit en mesure de factoriser certains stimuli de façon plus efficace que les humains. Les stimuli utilisés pour l'expérience de chapitre 7 ne sont pas assez complexes pour permettre d'observer cette différence.

Dans le cas probable où l'algorithme A* serait « trop » performant, il pourrait être envisageable de restreindre l'espace de recherche en faisant varier la limite imposée à la taille de mémoire utilisée pour la recherche. Cela aurait pour conséquence d'introduire un paramètre dans le modèle, ce qui est peu souhaitable.

Codage autodélimitant

Les tailles de codages sont calculées à l'aide d'un code de Shannon-Fano qui n'est pas autodélimitant. Cela n'a aucune importance dans la mesure où les tailles de codage servent de critère à la création de chunks et n'ont pas pour but d'effectuer une réelle compression des données. Le codage en lui-même n'existe pas, et seule sa taille théorique est utile. Le calcul des tailles pour un codage autodélimitant pourrait se faire en ajoutant $2Log_2(n)$ bits à tout code de taille n bits.

one-part coding / two-part coding

Dans le MDL *two-part coding* tel qu'il est présenté au chapitre 3, la séparation du modèle et des données sachant le modèle permet de distinguer la structure de ce qui est assimilable à du bruit. Au-delà de cette simple séparation, les langages de description utilisés pour les deux parties sont généralement différents. Dans l'exemple de (Dowman, soumis) du chapitre précédent, le langage de description de la partie modèle est un arbre binaire ET/OU, et le langage de la partie données|modèle consiste à fixer les degrés de libertés laissés vacants par les disjonctions de l'arbre. L'expressivité des deux langages est complètement différente : l'un capture les ET, et l'autre les OU. Dans le cas du MDLChunker, le langage de représentation utilisé dans les deux parties est identique et pourrait être considéré comme un *one-part coding*. La séparation est purement formelle et les parties chunks et stimuli|chunks pourraient être regroupées sans que cela n'entraîne de modifications.

Nous avons vu dans le chapitre 3 que dans le poids relatif accordé à chacune des parties était implicitement fixé par le langage de représentation choisi pour chacune d'elles. Dans notre cas, les langages utilisés étant identiques, cette pondération appairait donc assez naturelle.

Disjonction

Le langage actuel a une expressivité permettant de décrire les conjonction (ET) ainsi que les négations (NON) grâce à l'amélioration proposée section 4.3.2. Il est naturel de se demander s'il est possible d'exprimer la disjonction (OU), afin d'avoir un langage possédant les trois opérations logiques de base (ET, OU, NON). Au niveau de la représentation, la disjonction revient à autoriser plusieurs chunks à avoir le même nom (C5' = C1 + C2 et C5" = C3 + C4). La taille de codage nécessaire pour choisir C5' ou C5" sachant C5 est alors plus faible que celle nécessaire pour coder directement C5' ou C5". Le problème qui se pose est que l'espace de recherche est beaucoup plus vaste que pour la conjonction puisqu'il s'agit du nombre de partitions possibles d'un ensemble de chunks. Le but étant de rester proche des régularités qu'un humain est susceptible de capturer, la conjonction semble pour le moment plus intéressante puisqu'elle permet notamment de comparer les résultats avec ceux obtenus grâce à d'autres modèles de *chunking* : l'implémentation de la disjonction n'est donc pas une priorité. Ce-

pendant, en l'absence de disjonction, la représentation de grammaires (Dowman, soumis) n'est pas possible.

Nature discrète ou continue des stimuli

Le MDLChunker est capable de traiter des stimuli temporellement et spatialement discrets. La généralisation à l'utilisation de données temporellement continues est possible. Cette modification sera présentée au chapitre 8 en même temps que l'expérience correspondante. Dans ce cas, les données apparaissent sous la forme d'un flux continu d'éléments (Fig. 4.21).

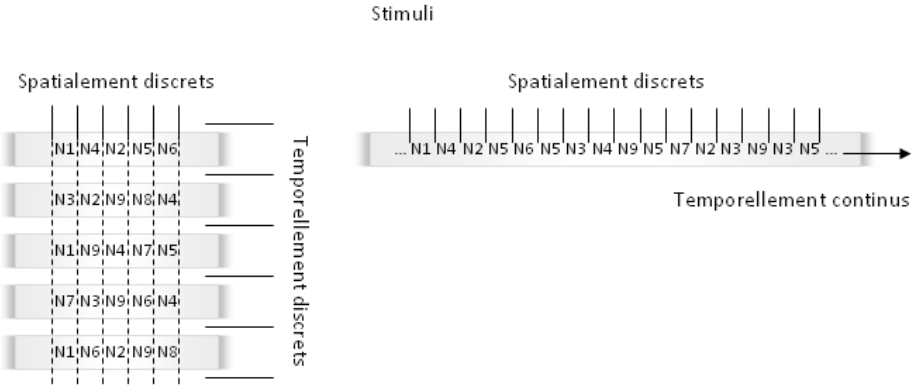


FIGURE 4.21 – Présentation de la différence entre les stimuli temporellement discrets et temporellement continus.

En revanche, la généralisation du MDLChunker pour permettre l'utilisation de stimuli spatialement continus est totalement incompatible avec l'utilisation de la théorie de l'information qui suppose des réalisations discrètes de variables aléatoires discrètes. Une discrétisation spatiale est toujours possible, mais elle doit également s'accompagner d'une discrétisation des valeurs que peuvent prendre les éléments des stimuli puisque l'alphabet utilisé doit être discret. Cette discrétisation supprime la relation d'ordre naturelle qui existe entre les valeurs que peut prendre une variable continue.

Le MDLChunker a été utilisé sur les données présentées dans l'article de French, Mareschal, Mermillod, et Quinn (2004) visant à modéliser et tester l'acquisition chez de jeunes enfants des catégories « chien » et « chat » à partir d'images. Les caractéristiques permettant de distinguer les deux animaux étant continues (longueur des oreilles, distance entre les yeux, largeur du nez, etc.), une discrétisation a été nécessaire. Plusieurs pas de discrétisation ont été testés sans que le MDLChunker soit en mesure de reproduire l'effet d'asymétrie décrit par les auteurs : sur ces données, les enfants créent une catégorie chien qui inclue les chats[6], mais une catégorie chat qui exclue les chiens. Cet effet s'explique par une différence dans les distributions des caractéristiques (continues) selon la catégorie chien ou chat. Leur discrétisation a supprimé cette information, rendant

6. Un chat est un chien comme un autre.

l'effet impossible à reproduire.

Une des différence du MDLChunker avec un réseau de neurones formels, porte justement sur l'aspect discret de l'alphabet utilisé. Dans le cas d'un réseau de neurones les stimuli sont temporellement et spatialement discrets, mais les valeurs d'entrée du réseau (l'alphabet utilisé) peuvent être continues.

4.4.2 Conclusion

Généralité de l'approche choisie

Au-delà des deux principales limitations que sont l'absence de disjonction, et l'impossibilité de représenter des stimuli spatialement continus, le MDLChunker possède une assez grande généralité. Les stimuli peuvent être ordonnés ou non, et être exprimés à l'aide de n'importe quel alphabet discret de longueur finie. Cela permet de pouvoir utiliser le MDLChunker dans différents domaines et à différents niveaux de granularité.

L'atout principal du MDLChunker est son absence de paramètres. Le résultat qu'il fournit est totalement indépendant de celui produit par des participants humains : aucun ajustement a posteriori n'est nécessaire. Les prédictions effectuées dépendent donc uniquement des stimuli et pas des connaissances expertes qui auraient pu être ajoutées dans le but de reproduire un type particulier de données. Cela permet un meilleur contrôle de l'information introduite dans le modèle. Le MDLChunker ayant un comportement déterministe, il fournit un résultat et un seul, comme le ferait un participant humain. Dans les différents expériences, il est considéré comme un participant virtuel.

Le MDLChunker implémente le principe de simplicité sous la forme d'une compression des stimuli, basée sur l'utilisation d'un code de Shannon-Fano. Le MDLChunker effectue une compression sans perte des stimuli. Partant d'une représentation canonique redondante, il construit sa propre représentation des stimuli en supprimant certaines régularités. Deux types de régularités sont susceptibles d'être perçues et compressées : les régularités fréquentielles et les régularités de type ET.

Le MDLChunker peut être vu comme un algorithme de réduction de dimension, au même titre qu'une analyse en composantes principales (ACP). La différence est que le MDLChunker fonctionne sur des données discrètes et de façon itérative, et que les composantes de la base de représentation ne sont pas nécessairement indépendantes (orthogonales pour l'ACP, disjointes pour le MDLChunker). Les stimuli initiaux sont décrit dans un alphabet choisi pour sa pertinence mais potentiellement très redondant. Chaque stimulus perçu est alors factorisé de façon à tenir compte des connaissances déjà extraites, ce qui revient à une projection sur la base des chunks possédés. La difficulté d'effectuer cette projection tient justement au fait que les chunks ne sont pas nécessairement disjoints.

Principe général et implémentation spécifique

D'une façon plus générale, l'implémentation du MDLChunker qui est proposée dans ce chapitre vise à tester l'intérêt du principe de simplicité dans la modélisation du processus de *chunking*. L'implémentation proposée est un moyen d'atteindre ce but et non une fin. On souhaite tester un principe et pas l'implémentation qui en est faite. Il est nécessaire d'implémenter ce principe afin de pouvoir le valider en le comparant à des données fournies par des participants humains. Mais en testant la plausibilité cognitive du principe au travers d'une implémentation, on ne prétend pas pour autant proposer l'implémentation elle-même comme cognitivement plausible. Certains mécanismes, comme la factorisation des stimuli à l'aide de l'algorithme A*, sont même cognitivement très improbables.

Une implémentation cognitivement très improbable peut cependant fournir d'excellentes prédictions, pour peu qu'elle découle d'un principe qui, lui, peut être cognitivement justifié. Un exemple simple est la modélisation qui est faite du traitement post-rétinien en utilisant une transformée en séries de Fourier. Il a été établi que les premières aires cérébrales impliquées dans la vision effectuaient un traitement similaire à une transformée de Fourier de l'image renvoyée par la rétine (Hérault, 2009). Il n'a pourtant jamais été soutenu que le cerveau possédait la connaissance mathématique nécessaire pour effectuer un tel calcul comme nous le ferions à l'aide d'un ordinateur. Néanmoins, la meilleure modélisation du traitement effectué avant les premières aires visuelles reste à ce jour une transformée de Fourier.

À travers un exemple similaire, Perruchet et Gallego (2006) ont montré que le problème bien connu des élèves de lycée, consistant à appliquer le calcul de dérivées pour déterminer un trajet optimal, pouvait être résolu avec succès par un chien pour peut qu'on lui fournisse la motivation nécessaire (Fig. 4.22). Bien que les connaissances du chien en matière de calcul de dérivées soient relativement sommaires, le meilleur modèle permettant de prédire le point d'entrée dans l'eau consiste en un calcul de dérivée suivi d'une résolution d'équation.

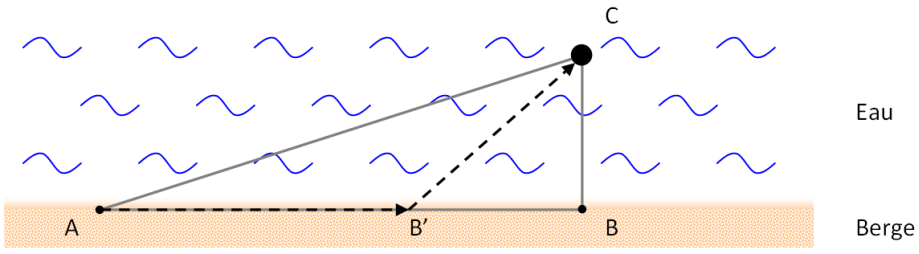


FIGURE 4.22 – L'expérience se passe sur la berge d'un lac. Le chien se situe au point A et doit aller chercher sa balle tombée dans l'eau au point C. La vitesse de déplacement est plus rapide sur la berge que dans l'eau. Le point B' d'entrée du chien dans l'eau correspond à celui qui assure le temps de trajet minimal.

À travers ces deux exemples, notre but est de montrer qu'un modèle, même cognitivement improbable, peut être un excellent prédicteur, pour peut qu'il

implémente un principe qui soit cognitivement justifié.

Unicité du principe et multiplicité des implémentations

En choisissant une catégorie de modèles, dans notre cas les modèles de *chunking* basés sur le principe de simplicité, on restreint le nombre de degrés de liberté dans l'espace des modèles possibles. Cette restriction n'est cependant pas suffisante pour qu'il n'y ait qu'une seule implémentation possible. Pour passer d'un principe général à une implémentation particulière capable de tourner sur un ordinateur, il est nécessaire de fixer d'autres degrés de liberté. Pour fixer ces degrés de liberté, des choix d'implémentation doivent être faits. Et en faisant ces choix, on apporte de l'information qui n'était pas contenue dans le principe de départ.

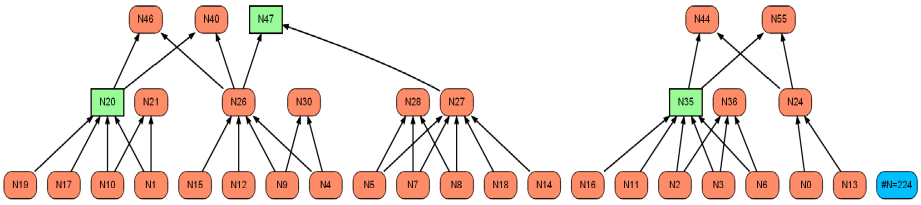
L'implémentation retenue contient nécessairement plus d'information que le principe de départ. Il est donc légitime de se demander dans quelle mesure les résultats obtenus dépendent du principe initial que l'on souhaite valider et non pas de l'implémentation qui est faite de ce principe. La seule validation possible serait de générer l'ensemble des implémentations possibles du principe afin de comparer les résultats obtenus, c'est-à-dire trouver toutes les façons possibles de fixer les degrés de liberté restants. Cela est bien évidemment impossible à réaliser en pratique. Néanmoins, dans un souci de validation [7] du principe et non de l'implémentation, le chapitre suivant propose une implémentation différente du principe étudié. Cette difficulté empirique de privilégier une implémentation plutôt qu'une autre (*algorithmic level*) lorsque toutes deux découlent d'un même principe (*computational level*) a notamment été discutée par Chater (1996).

7. Il s'agit bien sûr d'une tentative dérisoire au regard du nombre d'implémentations possibles.

Chapitre 5

MDLChunker-approché

Sommaire



5.1 Introduction

Le présent chapitre est à mettre en parallèle avec le chapitre précédent. Il s'intéresse au principe de simplicité basé sur l'association du MDL et du *chunking*, mais en propose une autre implémentation. Cette implémentation se base sur une architecture différente autorisant uniquement les calculs locaux. Cette nouvelle implémentation est appelée MDLChunker-approché dans la mesure où les calculs effectués sont une approximation de ceux réalisés par le chunker du chapitre 4.

En comparaison du précédent, ce chapitre est d'une importance relative moindre. L'implémentation présentée, si elle peut paraître cognitivement plus plausible, n'a pas vocation à remplacer celle du chapitre 4. Elle vise au contraire à l'étendre en montrant que le principe étudié n'est pas lié à une architecture donnée, et n'est pas conditionné à l'utilisation d'algorithmes nécessitant un volume de calculs cognitivement impossible. L'objectif poursuivi à travers ce chapitre est de proposer une implémentation locale capable de produire des résultats proches de ceux du MDLChunker.

Il existe nécessairement d'autres implémentations locales possibles (cf. section 4.4) et celle qui est décrite n'est sans doute pas la plus performante. L'important ici est simplement de montrer qu'une telle implémentation existe, et non de trouver la meilleure. Il est également très probable que certains raffinements des algorithmes proposés permettraient une augmentation substantielle des performances, mais ce n'est pas le but recherché.

5.1.1 Motivations

Le chapitre 4 présente avant tout un principe, non une implémentation. Le principe testé est que le MDL associé au *chunking*, permet dans les limites présentées, de prédire quels concepts sont susceptibles d'être créés à partir d'un ensemble donné de stimuli. Pour confronter ce principe aux données expérimentales, il a cependant été nécessaire de l'implémenter afin d'effectuer les simulations indispensables à sa validation. Nous avons choisi l'implémentation la plus naturelle possible afin que les résultats obtenus découlent du principe testé et non des choix faits dans l'implémentation. Si l'architecture choisie est simple et relativement facile à décrire, elle fait néanmoins appel à des algorithmes d'exploration nécessitant une importante charge de calculs.

Nous présentons dans ce chapitre une méthode permettant de remplacer les calculs complexes effectués par le MDLChunker en calculs locaux et simples qui ne soient pas cognitivement impossibles. Par simples, nous entendons qu'ils soient en $O(\#fils + \#parents)$ pour chaque élément de la hiérarchie de chunks. Par locaux, nous entendons que chaque élément fonctionne indépendamment des autres en utilisant uniquement l'information fournie par ses fils et ses parents directs, sans avoir une connaissance omnisciente de l'ensemble de la hiérarchie de chunks. Chaque élément est mis à jour de façon asynchrone, son état influençant en retour l'état de ses fils et celui de ses parents.

Cette hypothèse de calculs à la fois locaux et parallèles est une hypothèse souvent retenue dans les modèles cognitifs. C'est notamment les cas des réseaux de neurones formels dont l'architecture est intrinsèquement parallèle. Dans notre cas, cela revient à paralléliser les algorithmes utilisés dans le MDLChunker en les simplifiant pour qu'ils ne dépendent que d'informations locales. L'autre différence majeure est que l'architecture proposée ne suppose plus l'existence d'une mémoire infinie. Il ne s'agit pas d'une simple limitation de la taille de la mémoire (cf. chapitre 8), mais d'une disparition de la représentation explicite des stimuli perçus (partie Stimuli|Chunks du MDLChunker). La quantité de mémoire utilisée reste fixe au cours de l'apprentissage.

Le but de ce chapitre est double. D'une part, montrer que les résultats du chapitre 4 sont robustes et relativement indépendants de l'implémentation choisie en testant le même principe sur une architecture totalement différente. D'autre part, montrer que les résultats du MDLChunker ne sont pas conditionnés par des calculs complexes cognitivement impossibles. Ce point est nécessaire afin de vérifier que la modélisation proposée (MDL et *chunking*) n'est pas incompatible avec ce que l'on sait être la puissance calculatoire du cerveau (Oaksford & Chater, 1993 ; van Rooij, 2008).

Trois objections majeures peuvent être faites sur la plausibilité cognitive des processus décrits dans le chapitre 4. Dans l'implémentation proposée, l'ensemble des stimuli perçus sont stockés en mémoire. Ce stockage est nécessaire car chaque création de chunk est suivie d'un réencodage de tous les stimuli permettant de tenir compte du nouveau chunk. Cela suppose à la fois une mémoire infinie, mais surtout la possibilité d'effectuer régulièrement des modifications sur l'ensemble du contenu de cette mémoire. La seconde objection concerne la dimension de l'espace de recherche exploré lors de la factorisation d'un stimulus. L'algorithme A* utilisé pour trouver la meilleure décomposition du stimulus suppose une quantité de calculs importante puisque tous les chemins de forte utilité sont développés jusqu'au but (section 4.2.2 page 65). La troisième objection porte sur le nombre de cooccurrences envisagées dans l'étape d'optimisation. Afin de savoir quels chunks sont susceptibles d'être créés, il est nécessaire de connaître les chunks qui apparaissent fréquemment ensemble dans les stimuli. Le nombre de cooccurrences de chunks qu'il faut envisager est élevé : c'est le nombre de façons de choisir deux chunks parmi tous les caractères du stimulus.

Comme il a été indiqué, le modèle proposé au chapitre 4 sera dans la suite intitulé MDLChunker, dans la mesure où il s'agit de l'implémentation la plus brute possible du principe étudié, nécessitant des calculs complexes. L'implémentation décrite dans ce chapitre sera appelée MDLChunker-approché, en référence à la sous-optimalité des algorithmes utilisés

5.1.2 Présentation informelle du MDLChunker-approché

Avant de décrire plus en détail les algorithmes utilisés, effectuons une présentation intuitive du fonctionnement du MDLChunker-approché.

Architecture : L'architecture choisie pour le MDLChunker-approché est un réseau constitué de noeuds et d'arcs orientés. Cela permet de paralléliser l'information ainsi que les calculs, qui se retrouvent distribués au niveau des noeuds et des arcs. Les noeuds sont organisés en couches. Ceux de la couche la plus basse correspondent aux chunks-canoniques présents dans les stimuli. Ceux situés dans les couches supérieures regroupent les noeuds des couches inférieures pour former les chunks (Fig. 5.1).

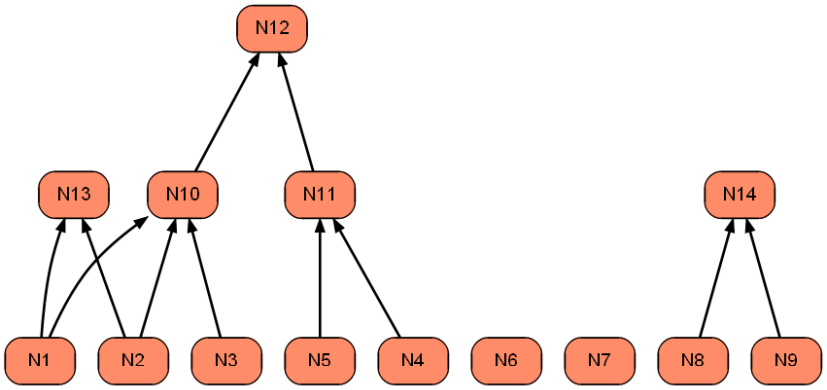


FIGURE 5.1 – Exemple de représentation d'un réseau comportant 5 chunks N10 à N14 et 9 chunks-canoniques N1 à N9.

De même façon que dans le MDLChunker, le MDLChunker-approché fonctionne grâce à trois opérations : le calcul des tailles de codage des différents éléments, la factorisation du stimulus courant et l'optimisation du réseau.

Factorisation : La factorisation est l'étape permettant de trouver la représentation la plus courte (en terme de taille de codage) du stimulus courant. Lorsqu'un stimulus est perçu, les noeuds-canoniques correspondants sont activés (Fig. 5.2 page suivante).

La factorisation proprement dite intervient ensuite par propagation des activations le long des arcs. Un noeud s'active lorsque l'ensemble de ses fils sont activés. Il peut y avoir ambiguïté dans la factorisation lorsque deux noeuds ont des fils activés en commun (par exemple N13 et N10). Il est alors nécessaire d'introduire la notion d'utilité d'un noeud, qui représente la taille de codage que l'on gagne à activer un noeud plutôt que l'ensemble de ses fils. Un noeud d'utilité positive s'active en inhibant ses fils (Fig. 5.3 page ci-contre noeud 14). Un noeud ne peut s'activer si son utilité est négative, c'est-à-dire s'il est moins intéressant que ses fils ou si tous ses fils ne sont pas activés (Fig. 5.3 page suivante noeud 11). La notion d'utilité est surtout importante pour déterminer la factorisation la plus intéressante lorsque plusieurs sont possibles. Par exemple figure 5.3, les noeuds N1, N2, N3 peuvent se factoriser en N13 et N3 ou simplement en N10. Le noeud choisi est alors celui de plus forte utilité (ici N10).

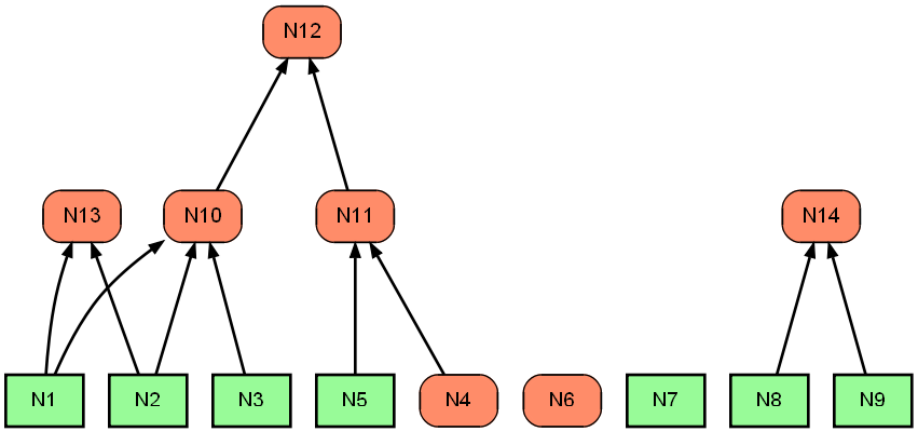


FIGURE 5.2 – Etat du réseau lors de la perception du stimulus 1 2 3 5 7 8 9 et avant sa factorisation. Les noeuds-canoniques N1 N2 N3 N5 N7 N8 N9 sont activés (rectangles verts).

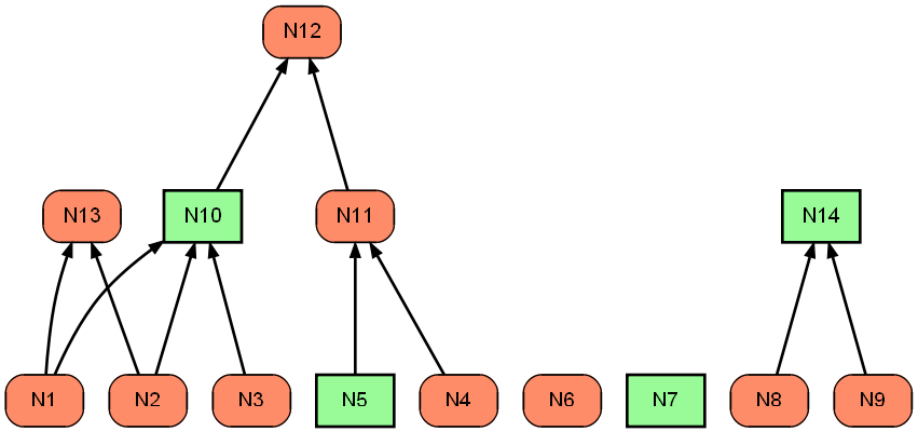


FIGURE 5.3 – Etat du réseau de la figure 5.2 après factorisation. Les noeuds N8 et N9 sont factorisés en N14. Les noeuds N1, N2 et N3 sont factorisés en N10 qui a une utilité plus forte que N13.

L'algorithme de factorisation nécessite que les noeuds possèdent deux attributs : **activé** (booléen) ainsi que l'**utilité du noeud** (réel).

Tailles de codage : Pour calculer les tailles de codage des noeuds du réseau, il est nécessaire de connaître le nombre d'activations de chaque noeud. Les noeuds activés à l'issue de l'étape de factorisation voient leur nombre d'activations incrémenté de 1. Les tailles de codage sont mises à jour en conséquence.

L'algorithme de calcul des tailles de codage introduit deux nouveaux attributs

pour les noeuds : le **nombre d'activations** du noeud (entier), et sa **taille de codage** proprement dite (réel).

Optimisation : L'étape d'optimisation du réseau est plus délicate. Elle consiste à créer de nouveaux arcs lorsque cela fait diminuer la taille de codage totale du réseau. Initialement, chaque noeud du réseau est connecté à l'ensemble des noeuds de la couche supérieure par des arcs non-créés. Les arcs non-créés existent [1] mais pour plus de lisibilité, soit ils ne sont pas représentés, soit ils sont représentés en pointillés lorsque cela est nécessaire pour une meilleure compréhension (Fig. 5.4). Créer un nouvel arc revient à créer un nouveau chunk ou à modifier un chunk existant (augmenter son arité). Cela correspond bien à l'optimisation telle qu'elle est effectuée dans le MDLChunker.

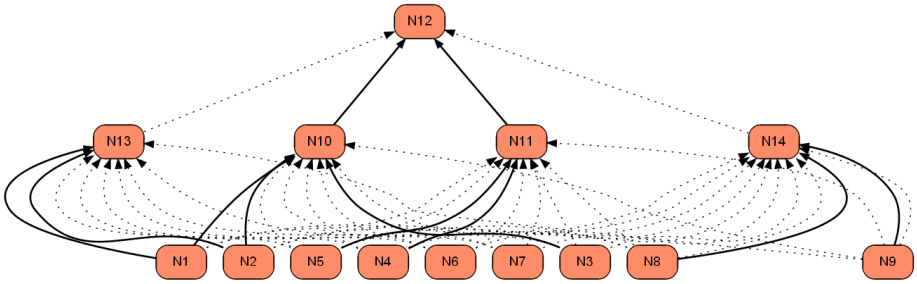


FIGURE 5.4 – Réseau de la figure 5.1 page 92 dans lequel les arcs non-créés sont représentés en pointillés.

Comme évoqué dans l'introduction, la différence fondamentale entre le MDLChunker et le MDLChunker-approché est que ce dernier ne possède pas de représentation explicite des stimuli perçus (partie Stimuli|Chunks du MDLChunker). L'information relative aux cooccurrences entre noeuds doit être extraite au fur et à mesure de l'apprentissage. Elle est stockée par les arcs du réseau. Chaque arc a pour but de comptabiliser le nombre de fois où son père et son fils apparaissent conjointement (cooccurrences).

Connaissant le nombre de cooccurrences des différents noeuds et le nombre d'occurrences de chaque noeud, de la même façon que dans le MDLChunker, il est possible de déduire l'utilité de chaque arc : c'est le gain de taille de codage qu'induirait la création de l'arc. L'étape d'optimisation consiste à créer tous les arcs d'utilité positive. Supposons que l'arc $N6 \rightarrow N13$ figure 5.5 page suivante ait une utilité positive. Il est alors créé, transformant le chunk binaire $N13 = N1 + N2$ en un chunk ternaire $N13 = N1 + N2 + N6$.

Dans la phase d'optimisation ci-dessus, un détail a été volontairement passé sous silence. Le nombre d'occurrences des noeuds du réseau, indispensable au calcul de l'utilité des arcs, ne correspond pas au nombre d'activations. En effet, le nombre d'activations représente le nombre de fois où un noeud a été activé durant la phase de factorisation. Un noeud qui n'est jamais activé doit pouvoir

1. Les arcs non-créés ont un rôle de stockage d'information : ils contiennent les mêmes attributs que les arcs créés. Ils sont susceptibles d'être créés durant l'étape d'optimisation.

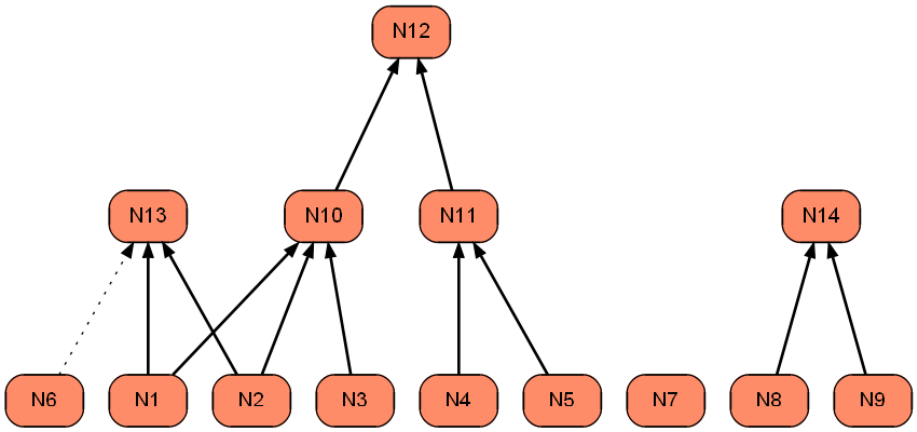


FIGURE 5.5 – Etat du réseau avant optimisation. L'arc $N6 \rightarrow N13$ n'est pas encore créé (pointillés), mais son utilité est positive.

donner lieu à une optimisation dès lors que ses fils sont apparus dans les stimuli. Le nombre d'occurrences d'un noeud n'est donc pas le nombre d'activations mais le nombre de fois où le noeud aurait pu être activé (sous entendu, si sa taille de codage avait été plus faible). Les figures 5.6 et 5.7 illustrent la différence entre le nombre d'occurrences et le nombre d'activations.

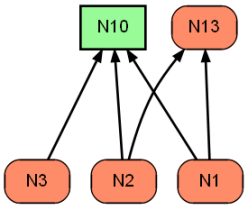


FIGURE 5.6 – Factorisation possible du stimulus 1 2 3 si l'utilité du noeud N10 est plus élevée que celle du noeud N13. Le nombre d'activations de N10 est incrémenté, celui de N13 reste inchangé. Le nombre d'occurrences des deux noeuds est incrémenté car tous deux auraient pu être activés (tous leurs fils étaient activés).

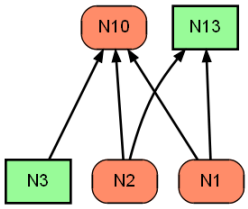


FIGURE 5.7 – Factorisation possible du stimulus 1 2 3 si l'utilité du noeud N13 est plus élevée que celle du noeud N10. Seul le nombre d'activations de N13 est incrémenté. En revanche, les deux noeuds voient leur nombre d'occurrences incrémenté. Le nombre d'occurrences est identique dans ce cas et dans le précédent. Il ne dépend pas du résultat de la factorisation.

Le nombre d'occurrences d'un noeud est incrémenté dès que ses fils sont activés, qu'il soit lui-même activé ou non. Dans l'exemple figures 5.6 et 5.7, les noeuds N1 N2 N3 sont activés, le nombre d'occurrences des deux noeuds N10 et N13 est donc incrémenté. En résumé, le nombre d'occurrences dépend uniquement

de la structure du réseau et non des tailles de codage. Il est indépendant des noeuds activés par l'étape de factorisation.

Pour fonctionner, l'étape d'optimisation requiert donc un attribut supplémentaire pour les noeuds : le **nombre d'occurrences** du noeud (entier). Elle requiert également trois attributs pour les arcs : **créé** (booléen), le **nombre de cooccurrences** des noeuds père et fils (entier), ainsi que l'**utilité de l'arc** (réel).

Après cette présentation rapide et succincte du MDLChunker-approché, il est d'ores et déjà possible d'apprécier quelles vont être les similitudes et différences avec le MDLChunker.

5.1.3 Similitudes et différences avec le MDLChunker

Certaines parties du MDLChunker se retrouvent sous une forme équivalente dans le MDLChunker-approché. D'autres en revanche sont simplifiées afin d'être compatibles avec une architecture n'autorisant que les calculs locaux et simples sans mémoire infinie.

Dans le MDLChunker-approché, la parallélisation de l'information et des calculs entraîne une dégradation des performances. Cela conduit à la fois à une perte d'information (compression avec perte) et à une convergence des algorithmes de calcul vers des solutions sous-optimales en comparaison de celles du le MDLChunker. La différence majeure de cette nouvelle architecture est l'absence de stockage explicite des stimuli perçus (partie Stimuli|Chunks du MDLChunker). Lorsqu'un stimulus est traité, les régularités qui sont pertinentes à cette phase de l'apprentissage sont extraites, les autres sont perdues. Pour dire les choses autrement, en l'absence d'une mémoire infinie, l'état actuel du réseau définit quelles informations pourront être extraites du stimulus courant. Cela modifie l'état du réseau et donc la perception des stimuli futur, sans qu'il soit possible de revenir sur les stimuli passés pour ré-extraire l'information. Passons rapidement en revue ce qui est commun aux deux MDLChunker avant de nous intéresser plus en détail à leurs différences.

Les similitudes avec le MDLChunker

La partie Chunks du MDLChunker se retrouve stockée sans perte d'information dans l'organisation des noeuds et des arcs du MDLChunker-approché. A chaque chunk du MDLChunker correspond un noeud dans les couches supérieures du MDLChunker-approché. La définition d'un chunk est donnée par la liste de ses noeuds fils (Fig.5.8).

Le calcul des tailles de codage fournit les mêmes résultats pour les deux MDLChunkers. Les calculs locaux à chaque noeud du MDLChunker-approché correspondent aux calculs des tailles de codages des chunks du MDLChunker.

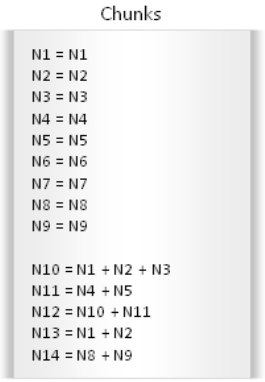


FIGURE 5.8 – Partie Chunks du MDLChunker correspondant à l'architecture du MDLChunker-approché présentée Fig.5.1.

Les différences avec le MDLChunker

La différence majeure est que le MDLChunker-approché ne stocke qu'une partie de l'information contenue dans les stimuli. Cela a un impact sur les deux opérations principales, factorisation et optimisation, dont les performances sont inférieures à celles de leurs homologues du MDLChunker.

La Partie Stimuli|Chunks n'est pas stockée explicitement dans le MDLChunker-approché. Cette information est stockée de manière partielle dans les poids portés par les sommets et les arcs du réseau (occurrences et cooccurrences).

Dans le MDLChunker, les stimuli perçus sont intégralement stockées sous une forme compressée grâce aux chunks, sans altération de leur contenu. Il s'agit d'une compression sans perte. Le MDLChunker-approché, bien qu'utilisant les mêmes principes, ne stocke pas explicitement les stimuli et ne permet donc pas de recréer les données originales. En résumé, le MDLChunker-approché possède une partie Chunks, mais pas de partie Stimuli|Chunks.

Le mécanisme de factorisation est le mécanisme utilisé pour encoder chaque stimulus sous sa forme la plus concise grâce aux chunks disponibles à cette étape du traitement. L'algorithme A* utilisé dans le MDLChunker est remplacé par une version dégradée interdisant les retours en arrière. Lors de l'exploration, un chemin choisi n'est plus remis en question, les chemins abandonnés n'étant jamais réévalués. Le chemin entre le départ et le but est trouvé en choisissant systématiquement la branche de plus forte utilité. La factorisation d'un nouveau stimulus se fait donc naturellement par propagation des noeuds activés le long des arcs.

Le mécanisme d'optimisation proprement dit est identique dans les deux MDLChunkers puisqu'il dépend uniquement de l'estimation du gain de taille

qu'induirait la création d'un nouveau chunk. En revanche dans le MDLChunker-approché, chaque création de chunk entraîne une perte d'information due à l'absence de représentation explicite des données passées (partie Stimuli|Chunks). Le nombre de cooccurrences qu'il y a eu dans le passé entre le chunk nouvellement créé et les autres noeuds n'est plus accessible.

Dans le MDLChunker, lors de la création d'un chunk AB, toutes les cooccurrences de A et B dans les stimuli passés se trouvent remplacées par AB. Le nombre d'apparitions conjointes de AB avec C par exemple est connu. Le MDLChunker-approché ne stockant pas explicitement les Stimuli|Chunks, accéder aux stimuli passés n'est plus possible. Les figures 5.9 et 5.10 présentent ce problème sur un exemple.

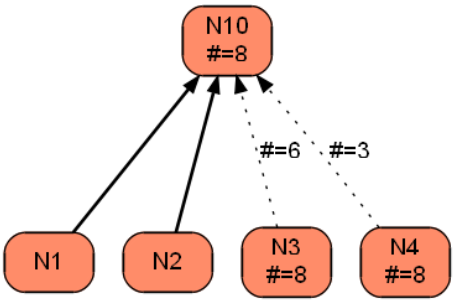


FIGURE 5.9 – Réseau comportant un chunk N10=N1+N2. Les arcs en pointillés ne sont pas encore créés, indiquant que N3 et N4 ne font pas partie du chunk N10. Les # sur les noeuds indiquent le nombre d'occurrences du noeud et les # des arcs, le nombre de cooccurrences des noeuds père et fils. N3 est apparu 8 fois, dont 6 avec N1 et N2. N4 est apparu 8 fois dont 3 avec N1 et N2. N10=N1+N2 est apparu 8 fois.

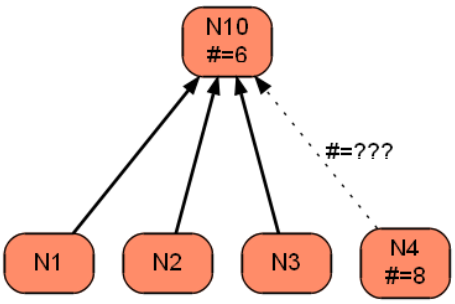


FIGURE 5.10 – Le noeud N3 a été ajouté au chunk N10 de la Fig.5.9. Il n'est plus possible de connaître avec exactitude le nombre de cooccurrences de N4 avec le nouveau N10=N1+N2+N3 (cela reste vrai même si les cooccurrences de N3 avec N4 étaient connues). Cette information n'a pas été extraite des données passées et n'est plus disponible.

Après cette introduction sur le MDLChunker-approché et le rapide aperçu des similitudes et différences qu'il entretient avec le MDLChunker, nous allons dé-

crire plus en détail son fonctionnement.

5.2 Fonctionnement

Cette section explique de façon détaillée le fonctionnement du MDLChunker-approché. Le niveau de détail a été poussé assez loin afin que ce chapitre suffise à le reprogrammer entièrement. Pour simplifier la présentation, les algorithmes nécessaires à la programmation sont présentés en pseudo-code et les explications fournies en langage naturel.

5.2.1 Principe

Le MDLChunker-approché fonctionne sur le même principe que le MDLChunker et utilise les deux mêmes types d'opérations : factorisation et optimisation. La factorisation est l'opération consistant à ré-exprimer chaque stimulus sous la forme la plus concise possible (partie Stimuli|Chunks), en utilisant les chunks déjà créés. L'optimisation est la phase de création des nouveaux chunks dans le réseau (partie Chunks). Comme dans le MDLChunker, ces deux phases alternent au cours de l'apprentissage et font appel à des parties différentes du réseau. La factorisation modifie et utilise principalement l'information contenue dans les noeuds et l'optimisation celle contenue dans les arcs. La distinction n'est pas aussi stricte, mais fourni une idée générale assez claire de l'architecture utilisée.

5.2.2 Architecture

Organisation

La structure générale du réseau demeure fixe au cours de l'apprentissage, seuls les informations contenues dans les arcs et les noeuds évoluent. Afin d'éviter la formation de cycles, le réseau est organisé en couches. Chaque couche contient autant de noeuds que de chunks-canoniques nécessaire à la représentation des stimuli. Chaque noeud de la couche N est relié à l'ensemble des noeuds de la couche $N + 1$ par des arcs non-créés (sauf un arc noté créé). La Fig. 5.11 page suivante présente l'état d'un réseau au début de l'apprentissage.

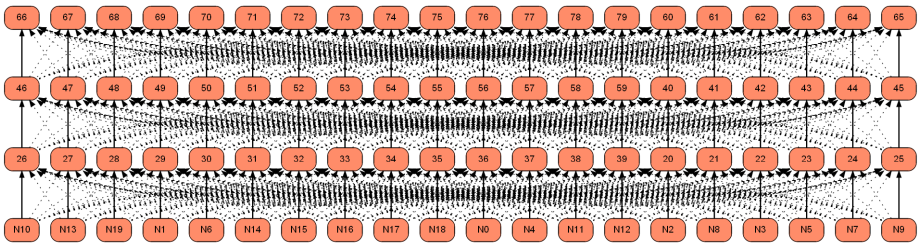


FIGURE 5.11 – Exemple de structure initiale de réseau pour 20 caractères-canoniques et 4 couches. Les arcs créés sont en traits pleins et les arcs non-créés en pointillés.

Noeuds

Chaque noeud du réseau contient six attributs (Fig. 5.12 page ci-contre) :
- **activé** (booléen) : indique si le noeud est utilisé pour factoriser le stimulus.
- **nombre d'activations** (entier) : nombre de fois où le noeud a été activé.
- **filsActivés** (booléen) : indique si tous les fils du noeud sont activés.
- **nombre d'occurrences** (entier) : nombre de fois où filsActivés est vrai.
- **utilité** (réel) : performance [2] du noeud pour encoder le stimulus.
- **taille de codage** (réel) : taille de codage du noeud.

Les attributs **activé**, **nombre d'activations** et **utilité** sont mis en jeu uniquement par l'étape de factorisation et les attributs **filsActivés** et **nombre d'occurrences** par l'étape d'optimisation.

Arcs

Chaque arc du réseau contient trois attributs (Fig.5.12) :
- **créé** (booléen) : indique si l'arc est créé [3] ou non.
- **nombre de cooccurrences** (entier) : nombre de fois où le père et le fils de l'arc sont apparus dans un même stimulus.
- **utilité** (réel) : performance [4] de l'arc.

Les attributs **nombre de cooccurrences** et **utilité** servent uniquement pour l'étape d'optimisation.

2. Taille de codage que l'on gagne à utiliser ce noeud plutôt que ses fils pour factoriser le stimulus courant.
3. Un arc qui n'est pas créé contient les mêmes informations qu'un arc créé, mais n'est pas pris en compte dans le chunk.
4. Gain en terme de taille de codage qu'induirait la création de l'arc pour optimiser le réseau.

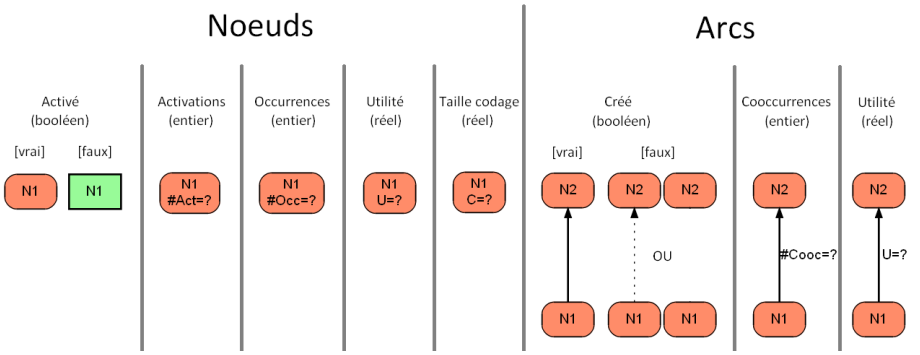


FIGURE 5.12 – Nomenclature utilisée pour représenter les différents attributs des noeuds et des arcs.

5.2.3 Calculs

Tous les calculs sont effectués en parallèle à partir des informations disponibles localement. Chaque calcul est réalisé de manière asynchrone dès que ses conditions d'application sont vérifiées. La taille de codage par exemple est mise à jour en permanence, même si cela n'a d'impact que lorsque le nombre d'activations est modifié.

Tailles de codage (Algo.4)

Le calcul des tailles de codage est en tout point identique à celui qui est réalisé par le MDLChunker. La taille de codage d'un noeud est notée $C(noeud)$. Elle dépend uniquement du nombre d'activations du noeud $\#Act(noeud)$, et du nombre total d'activations depuis le début de l'apprentissage $\#N$ (Algo. 4).

Algorithme 4 Mise à jour de la taille de codage du noeud (calcul local à chacun des noeuds)

1: $C(noeud) \leftarrow log_2\left(\frac{\#N}{\#Act(noeud)}\right)$

Ce calcul est effectué de façon simultanée par chacun des noeuds. Cela nécessite la connaissance de $\#N$ qui est une information globale à l'ensemble du réseau. Déterminer localement $\#N$ peut se faire par ajout d'un noeud supplémentaire connecté à l'ensemble des noeuds du réseau, dont le but est de comptabiliser le nombre total d'activations (Fig. 5.13 page suivante).

Les tailles de codage des noeuds du réseau sont mises à jour en permanence. Elles évoluent à chaque modification de $\#N$. La taille de codage de chaque noeud est supposée connue à tout moment. Pour simplifier les schémas, $\#N$ ne sera pas systématiquement représenté.

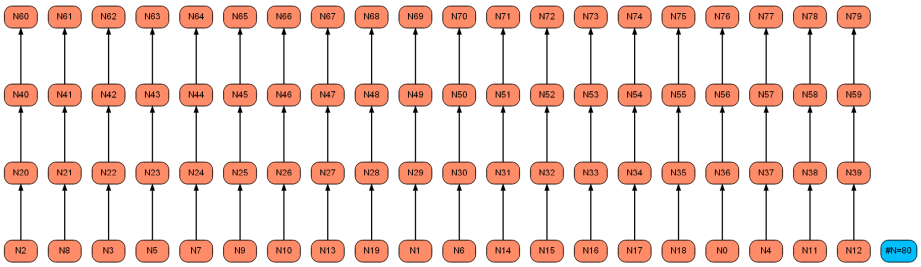


FIGURE 5.13 – Ajout d'un noeud supplémentaire (bleu) comptabilisant le nombre d'activations dans le réseau. Chaque nouvelle activation d'un noeud met à jour $\#N$, qui en retour induit une modification des tailles de codage de chacun des noeuds. Les arcs reliant $\#N$ aux autres noeuds ne sont pas représentés.

Initialisation : le calcul de la taille de codage met en jeu un logarithme. Le nombre d'activations d'un noeud ne peut donc être nul. Initialement, tous les noeuds sont activés une fois. Il s'agit du même procédé que celui utilisé pour le MDLChunker. On a $\#N = 80$ par exemple pour le réseau de la Fig. 5.13, juste après son initialisation.

Vers un calcul de taille de codage plus local : Une autre façon de voir le calcul de taille de codage est la suivante : lorsqu'un noeud du réseau est activé, tous les noeuds voient leur taille de codage augmenter d'une quantité $\Delta 1$

$$\Delta 1 = log_2\left(\frac{\#N+1}{\#N}\right)$$

et le noeud A voit en plus sa taille de codage diminuer de $\Delta 2$ [5].

$$\Delta 2 = log_2\left(\frac{\#Act(A)+1}{\#Act(A)}\right)$$

Seul $\Delta 1$ dépend de l'information globale $\#N$. Sous l'hypothèse où le nombre de noeuds activés par unité de temps peut être considéré comme relativement constant, $\Delta 1$ pourrait être déduit à partir d'un calcul local dépendant uniquement du temps. L'astuce consistant à utiliser un noeud spécial pour comptabiliser $\#N$ pourrait être évitée. Ce n'est pas cette solution mais bien la précédente, moins élégante mais plus juste, qui est retenue pour la suite.

5. La taille de codage d'un noeud A est $log_2\left(\frac{\#N}{\#Act(A)}\right)$. Si un autre noeud est activé, la taille de codage de A devient $log_2\left(\frac{\#N+1}{\#Act(A)}\right) = log_2\left(\frac{\#N}{\#Act(A)} \cdot \frac{\#N+1}{\#N}\right)$. Si c'est le noeud A qui est activé, sa taille de codage devient $log_2\left(\frac{\#N+1}{\#Act(A)+1}\right) = log_2\left(\frac{\#N}{\#Act(A)} \cdot \frac{\#N+1}{\#N} \cdot \frac{\#Act(A)}{\#Act(A)+1}\right)$. On retrouve bien $\Delta 1$ et $\Delta 2$.

Factorisation (Algos. 5 6 7 et 8)

L'opération de factorisation consiste à trouver la représentation la plus concise d'un stimulus connaissant sa représentation canonique. Dans le MDLChunker, cette opération est réalisée par un algorithme de type A* (détaillé section 4.2.2 page 65). L'algorithme utilisé dans le MDLChunker-approché en est une approximation : lors de l'exploration, les branches laissées de côté ne sont pas réévaluées. Ce qui revient à choisir systématiquement la meilleure branche et à s'arrêter dès que le but est atteint.

Dans de nombreuses situations, les factorisations obtenues avec le MDLChunker-approché sont identiques à celles obtenues avec le MDLChunker. La figure 5.14 présente un cas fictif pour lequel les deux modèles produiraient des factorisations différentes.

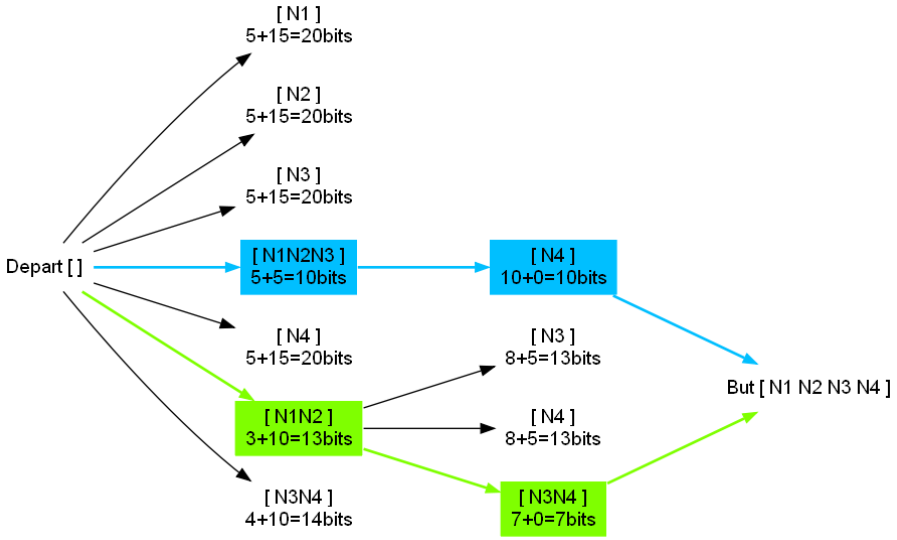


FIGURE 5.14 – Exemple de situation pour laquelle les factorisations produites par le MDLChunker et le MDLChunker-approché sont différentes. Pour factoriser le stimulus 1 2 3 4, le MDLChunker trouve les deux solutions (bleue [en haut] puis verte [en bas]) car il est capable de revenir en arrière pour développer une branche qui semble au premier abord moins intéressante. Il choisit finalement la solution verte (taille de codage = 7 bits), tandis que le MDLChunker-approché ne trouve que la solution bleue (taille de codage = 10 bits), préférable au premier abord, mais finalement moins performante. Le contenu de chaque solution est trouvé en concaténant les solutions partielles entre crochets du départ jusqu'au but. La taille de codage indiquée pour chaque solution partielle comprend la taille du chemin parcouru à laquelle s'ajoute une estimation de la taille du chemin restant à parcourir lorsque seuls les chunks-canoniques sont utilisables. Les tailles de codages sont : $C(N1) = C(N2) = C(N3) = C(N4) = C(N1N2N3) = 5$ bits, $C(N1N2) = 3$ bits, $C(N3N4) = 4$ bits.

Lors de la recherche de la meilleure factorisation possible, l'algorithme implé-

menté dans le MDLChunker-approché est plus facilement sujet à tomber dans un minimum local que celui du MDLChunker. Dans MDLChunker-approché, la factorisation est réalisée par quatre algorithmes (Algos. 5 6 7 et 8).

Initialisation : l'utilité des noeuds-canoniques correspondant au stimulus à factoriser est notée positive. L'utilité des autres noeuds est notée négative (Algo.5).

Algorithme 5 Initialisation des noeuds-canoniques

si $noeud \in NoeudsCanoniques$ **alors**
  si $noeud \in Stimulus$ **alors**
    $U(noeud) \leftarrow \epsilon$ // ϵ : valeur positive arbitrairement petite
  sinon
    $U(noeud) \leftarrow -\infty$
  finsi
finsi

Mise à jour de l'utilité : si tous les fils du noeud ont une utilité positive, alors l'utilité du noeud est calculée, sinon elle est notée négative (Algo.6). L'utilité d'un noeud représente la taille de codage que l'on gagne à utiliser le noeud plutôt que ses fils pour encoder le stimulus courant (Fig. 5.15 page ci-contre). L'utilité est donnée par l'écart entre la taille de codage du noeud et la taille de codage de ses fils, auquel s'ajoute le gain de taille de codage apporté par les fils eux mêmes (utilité des fils).

Algorithme 6 Mise à jour de l'utilité du noeud (calcul local à chacun des noeuds)

si $\forall fils, U(fils) > 0$ **alors**
  $U(noeud) \leftarrow \left(\sum_{fils} C(fils)\right) - C(noeud) + \sum_{fils} U(fils)$
sinon
  $U(noeud) \leftarrow -\infty$
finsi

Activation des noeuds encodant le stimulus : le noeud du réseau ayant la plus forte utilité est activé et son nombre d'activations $\#Act(noeud)$ est incrémenté. L'activation des noeuds-canoniques correspondants (descendants) est notée négative afin que tous les noeuds qui partagent au moins un noeud-canonique avec le noeud activé ne puisse être activés. En rendant négative l'utilité des noeuds-canoniques descendants, l'utilité des noeuds parents issus de ces noeuds-canoniques devient également négative (Algo.6). Les figures 5.16 et 5.17 page 106 illustrent cette activation des noeuds de plus forte utilité sur un exemple.

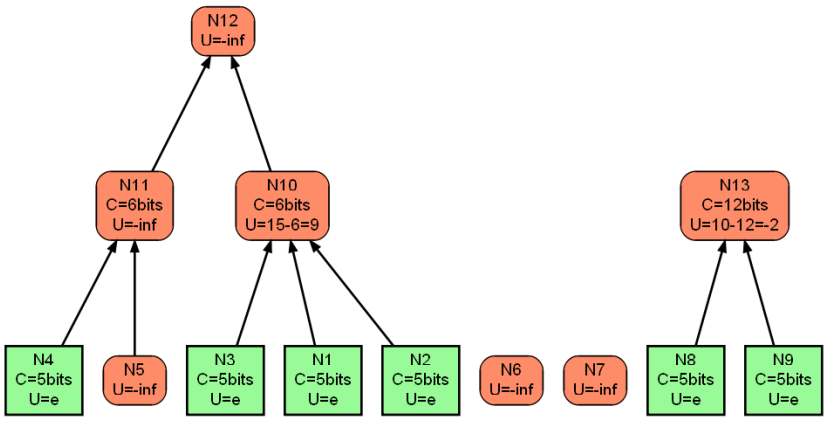


FIGURE 5.15 – Exemple de mise à jour des utilités à partir des noeuds-canoniques du stimulus : 1 2 3 4 8 9 (rectangles vert avec $U = \epsilon > 0$). N10 a une utilité positive car sa taille de codage $C(N10) = 6$ bits est inférieure à la somme des tailles de codage de ses fils $C(N1) + C(N2) + C(N3) = 15$ bits. A l'inverse, N13 a une utilité négative, ainsi que N11 dont un fils a une utilité négative.

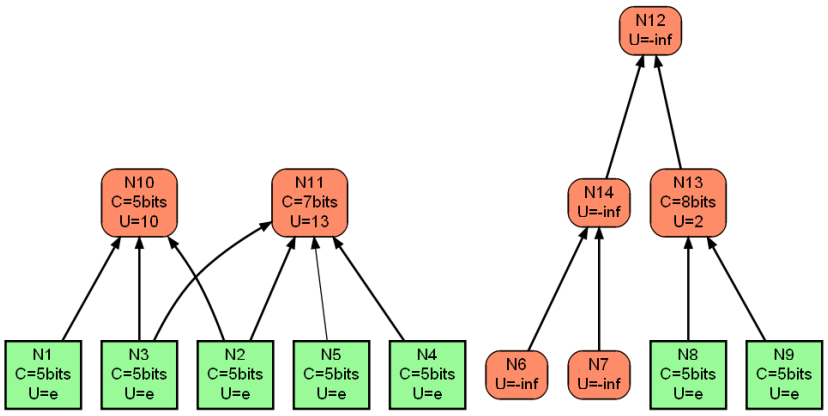


FIGURE 5.16 – Exemple d'un réseau avant la factorisation. Le stimulus courant est composé des noeuds 1 2 3 4 5 8 et 9 dont l'utilité est notée positive ($U = \epsilon > 0$). L'utilité des autres noeuds est calculée en conséquence.

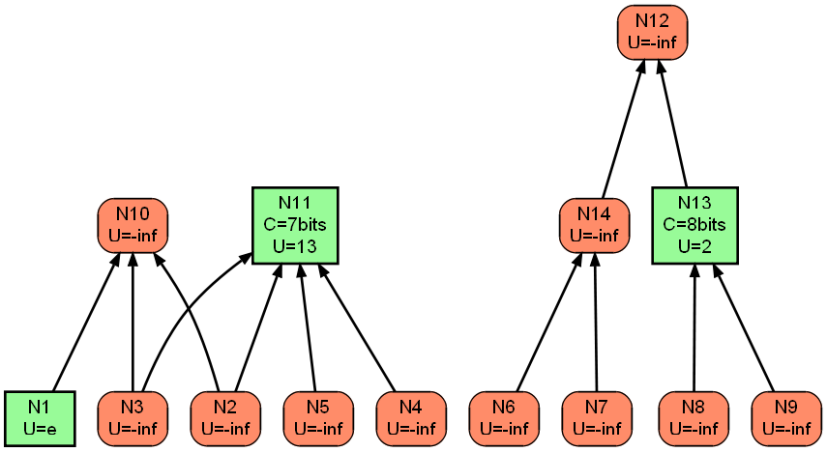


FIGURE 5.17 – Le noeud de plus forte utilité N11 est activé (rectangle). L'utilité des noeuds-canoniques correspondants (N2 N3 N4 N5) est notée $-\infty$, faisant passer à $-\infty$ l'utilité de N10 incompatible avec N11. Le noeud de plus forte utilité est maintenant N13 qui s'active à son tour. La représentation après factorisation est N1, N11, N13.

Présentés ainsi, les deux processus décrits ci-dessus n'ont rien de locaux : trouver le noeud du réseau ayant la plus forte utilité, puis noter négative l'utilité des noeuds-canoniques descendants du noeud activé.

Pour rendre locale la recherche du noeud de plus forte utilité, il suffit d'utiliser une astuce temporelle. Un noeud est activé au bout d'un temps inversement proportionnel à son utilité. Cela suffit pour que le noeud de plus forte utilité soit activé en premier (Algo.7). Lorsque deux noeuds sont incompatibles (ils possèdent au moins un noeud-canonique en commun), le noeud de plus forte utilité s'active au bout d'un temps plus court et inhibe l'activation de l'autre noeud en rendant son utilité négative (Fig. 5.17).

Si l'utilité du noeud est encore positive après le délai d'attente (Alg.7 lignes 2 et 3), c'est qu'aucun noeud incompatible ne possède une utilité plus forte. Le noeud s'active donc et incrémente son nombre d'activations (Alg.7 lignes 4 et 5).

Algorithme 7 Mise à jour du nombre d'activations du noeud (calcul local à chacun des noeuds)

1: **si** $U(noeud) > 0$ **alors**
2: $\quad attendre\left(\frac{1}{U(noeud)} \cdot \mu s\right)$
3: $\quad$ **si** $U(noeud) > 0$ **alors**
4: $\quad\quad Act(noeud) \leftarrow vrai$
5: $\quad\quad \#Act(noeud) \leftarrow \#Act(noeud) + 1$
6: $\quad\quad \forall fils$, envoyer(signalUtilitéNégative) // Désactive les fils
7: $\quad$ **finsi**
8: **finsi**

Donner une utilité négative aux noeuds-canoniques descendants du noeud activé peut être fait également à l'aide de calculs locaux. Le noeud activé impose à ses fils d'avoir une utilité négative, qui eux même imposeront une utilité négative à leurs fils récursivement jusqu'aux noeuds-canoniques (Algo.7 ligne 6 et Algo.8).

Algorithme 8 Mise à jour de l'utilité du noeud (calcul local à chacun des noeuds)

```
1: si reçu(signalUtilitéNégative) alors
2:    U(noeud) ← −∞
3:    ∀fils, envoyer(signalUtilitéNégative)        // Désactive ses fils
4: finsi
```

Cette implémentation de la factorisation est une approximation de l'implémentation réalisée par le MDLChunker.

Optimisation (Algos. 9 10 11 12 13)

L'optimisation du réseau est la phase consistant à créer de nouveaux noeuds afin de diminuer la taille de codage de l'ensemble du système : Stimuli|Chunks + Chunks. Les noeuds créés correspondent aux chunks du MDLChunker. De façon similaire au MDLChunker, l'optimisation se fait en deux phases. D'abord lister quels sont les noeuds souvent présents conjointement. Puis évaluer le gain en terme de taille de codage qu'entraînerait la création d'un nouveau chunk regroupant ces noeuds.

Le MDLChunker-approché ne stocke pas explicitement la partie Stimuli|Chunks. Les occurrences et cooccurrences des noeuds sont extraites au fur et à mesure de l'apprentissage grâce aux attributs **occurrences** et **cooccurrences** des noeuds et des arcs. L'information relative aux apparitions conjointes de deux noeuds est portée par l'attribut **cooccurrences** des arcs. Cette information est à mettre en relation avec le nombre d'apparition de chaque noeud (attribut **occurrences** des noeuds). Cela permet d'en déduire le gain associé à la création d'un chunk regroupant deux noeuds. Le calcul du gain est en tous points identique à celui réalisé par le MDLChunker.

Le processus d'optimisation induit une perte d'information à chaque création d'un nouveau chunk (Fig.5.9 et 5.10 page 98). Le nombre de cooccurrences du nouveau chunk avec les autres noeuds n'ayant pas été extrait des données passées, il n'est plus disponible pour guider les optimisations futures. Cette différence majeure avec le MDLChunker est uniquement due à l'absence de stockage de la partie Stimuli|Chunks. Cela conduit à une formation plus lente des chunks (section 5.3 page 117).

Le processus d'optimisation du MDLChunker-approché fonctionne de la façon suivante. Les attributs **occurrences** (Algos. 9 et 10) et **cooccurrences** (Algo.11) sont mis à jour. Cela permet de connaître les noeuds qui apparaissent fréquemment de façon conjointe et donc les chunks qui sont susceptibles d'être créés. L'utilité de chaque chunk est ensuite évaluée (Algo.12) : il s'agit du gain

en terme de taille de codage qu'induirait la création du chunk. Puis le chunk de plus forte utilité positive est créé (Algo.13).

Mise à jour des occurrences : si tous les fils d'un noeud sont activés, alors l'attribut **filsActivés** [6] passe à vrai et son nombre d'occurrences est incrémenté. (Algo.9). Connaître le nombre d'occurrences d'un noeud est une information indispensable à la création des chunks.

Algorithme 9 Mise à jour des occurrences du noeud (calcul local à chacun des noeuds)

1: **si** $\forall fils, Act(fils) = vrai$ **et** $noeud \notin NoeudsCanoniques$ **alors**
2: $filsActive(noeud) \leftarrow vrai$
3: $\#Occ(noeud) \leftarrow \#Occ(noeud) + 1$
4: **sinon**
5: $filsActive(noeud) \leftarrow faux$
6: **finsi**

Le présent paragraphe concerne une remarque d'algorithmique secondaire. Les noeuds-canoniques n'ayant pas de fils, le calcul de leur nombre d'occurrences est différent (cf. Algo.9 ligne 1). L'attribut **filsActivé** des noeuds-canoniques est noté vrai pour les noeuds contenus dans le stimulus. Le nombre d'occurrences est incrémenté en conséquence (Algo.10 venant compléter l'algorithme 5 page 104).

Algorithme 10 Initialisation des noeuds-canoniques

1: **si** $noeud \in NoeudsCanoniques$ **alors**
2: **si** $noeud \in Stimulus$ **alors**
3: $filsActive(noeud) \leftarrow vrai$
4: $\#Occ(noeud) \leftarrow \#Occ(noeud) + 1$
5: **sinon**
6: $filsActive(noeud) \leftarrow faux$
7: **finsi**
8: **finsi**

Mise à jour des cooccurrences : si le père d'un arc est présent et son fils activé, alors le nombre des cooccurrences père-fils est incrémenté (Algo.11). Le nombre de cooccurrences portées par un arc comptabilise le nombre d'apparitions conjointes du père et du fils de l'arc. Père et fils cooccurrent si le fils est activé et que le père a tous ses fils activés (**filsActivés** = vrai). Il est en effet impossible que le fils et le père soient tous deux activés simultanément. Le nombre de cooccurrences représente le nombre de fois où l'on aurait pu économiser la taille de codage du fils de l'arc si l'arc avait été créé. Cette information est indispensable pour la création de chunks.

6. Avoir tous ses fils activés est une condition plus faible qu'être soi même activé. Pour être activé, un noeud doit non seulement avoir tous ses fils activés, mais en plus être choisi par l'étape de factorisation.

Algorithme 11 Mise à jour des cooccurrences de l'arc (calcul local à chacun des arcs)

1: **si** $filsActive(pere) = vrai$ **et** $Act(fils) = vrai$ **alors**
2: $\quad \#Cooc(arc) \leftarrow \#Cooc(arc) + 1$
3: **finsi**

Mise à jour de l'utilité de l'arc : il s'agit là du coeur de l'optimisation. L'utilité d'un arc représente le gain de taille de codage qu'induirait la création de cet arc. Tout arc dont l'utilité est positive sera créé, donnant lieu à l'apparition d'un nouveau chunk.

De par sa structure, l'algorithme d'optimisation du MDLChunker-approché autorise naturellement la création de chunks n-aires, alors que seuls les chunks binaires sont envisagés par le MDLChunker. Il s'agit d'une différence de forme et non de fond, puisque la fonction de gain utilisée est identique dans les deux cas. À un chunk ternaire du type (A B C), correspondent dans le MDLChunker deux chunks binaires du type ((A B) C). La taille de codage des deux chunks ((A B) C) est exactement la même que celle du chunk (A B C). La représentation sous forme de chunks n-aires est donc purement graphique.

Lorsque deux noeuds A et B cooccurrent, le gain de taille de codage induit par la création du chunk AB est donné par la fonction de gain du MDLChunker (équation 4.14 page 78). Cette fonction dépend de :

1. $\#A$: le nombre d'apparitions de A.
2. $\#B$: le nombre d'apparitions de B.
3. $\#AB$: le nombre de apparitions conjointes de A et B.
4. $\#N$: le nombre total d'apparitions.
5. $\#A\bar{B}$: le nombre d'apparitions de A sans B.
6. $\#\bar{A}B$: le nombre d'apparitions de B sans A.
7. $\#\bar{A}\bar{B}$: le nombre d'apparitions d'autres noeuds que A et B.

Ces sept éléments sont déductibles des quatre premiers uniquement [7]. Ils correspondent tous à des informations possédées localement. $\#AB$ est le nombre de cooccurrences porté par l'arc $A \rightarrow B$: $\#Cooc(A \rightarrow B)$. $\#A$ est le nombre d'occurrences de A : $\#Occ(A)$. Et $\#B$ est le nombre d'activations de B : $\#Act(B)$. $\#N$ est connu de chacun des noeuds du réseau (Fig.5.13). L'utilité d'un arc $A \rightarrow B$ (gain de taille de codage à créer le chunk AB) peut donc être calculée uniquement à partir d'informations locales (Algo. 12 page suivante).

Création de l'arc : l'arc $B \rightarrow A$ de plus grande utilité est créé. Cela correspond à la création du chunk AB (figures 5.18 et 5.19 page suivante) (Algo.13 ligne 4). Les cooccurrences de A avec B sont alors remplacées par le nouveau chunk AB (Algo.13 lignes 5, 6 et 7).

La création de l'arc de plus forte utilité fait appel à la même astuce temporelle que pour la factorisation. Un temps d'attente inversement proportionnel à

7. On a $\#A\bar{B} = \#A - \#AB$, $\#\bar{A}B = \#B - \#AB$ et $\#\bar{A}\bar{B} = \#N - \#A - \#B + \#AB$.

Algorithme 12 Mise à jour de l'utilité de l'arc (calcul local à chacun des arcs)

1: $\#AB \leftarrow \#Cooc(arc)$
2: $\#A \leftarrow \#Occ(pere)$
3: $\#B \leftarrow \#Act(fils)$
4: $\#A\bar{B} \leftarrow \#A - \#AB$
5: $\#\bar{A}B \leftarrow \#B - \#AB$
6: $\#\bar{A}\bar{B} \leftarrow \#N - \#A - \#B + \#AB$
7: $P1 \leftarrow \#AB \cdot \left[log_2\left(\frac{\#N+1+2-\#AB}{\#AB+1}\right) - log_2\left(\frac{\#N}{\#A}\right) - log_2\left(\frac{\#N}{\#B}\right)\right]$
8: $P2 \leftarrow \#A\bar{B} \cdot \left[log_2\left(\frac{\#N+1+2-\#AB}{\#A\bar{B}+1}\right) - log_2\left(\frac{\#N}{\#A}\right)\right]$
9: $P3 \leftarrow \#\bar{A}B \cdot \left[log_2\left(\frac{\#N+1+2-\#AB}{\#\bar{A}B+1}\right) - log_2\left(\frac{\#N}{\#B}\right)\right]$
10: $P4 \leftarrow log_2\left(\frac{\#N+1+2-\#AB}{\#AB+1}\right) + log_2\left(\frac{\#N+1+2-\#AB}{\#A\bar{B}+1}\right) + log_2\left(\frac{\#N+1+2-\#AB}{\#\bar{A}B+1}\right)$
11: $P5 \leftarrow \#\bar{A}\bar{B} \cdot log_2\left(\frac{\#N+1+2-\#AB}{\#N}\right)$
12: $U(arc) \leftarrow -(P1 + P2 + P3 + P4 + P5)$

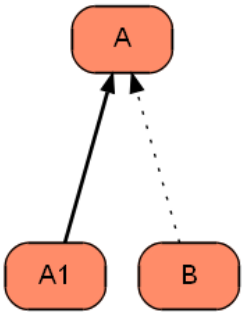


FIGURE 5.18 – Exemple de formation d'un chunk binaire. La création du lien $B \rightarrow A$ entraîne la création d'un chunk A1 B.

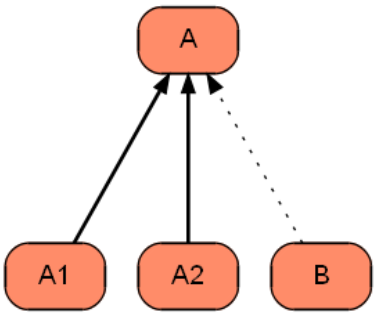


FIGURE 5.19 – Exemple de formation d'un chunk ternaire. La création du lien $B \rightarrow A$ entraîne la création d'un chunk A1 A2 B.

l'utilité de l'arc précède sa création (Algo.13 ligne 2).

Une fois le chunk AB créé, les cooccurrences de A avec B sont remplacées par AB. En utilisant les notations de l'algorithme 12 page précédente, cela donne les lignes 5, 6 et 7 de l'algorithme 13. Le nouveau chunk (noeud A) se trouve activé $\#AB$ fois : chaque fois que l'ancien noeud A et le noeud B sont apparus conjointement. Le noeud B est activé $\#\bar{A}B$ fois : chaque fois qu'il est apparu sans l'ancien noeud A. Chaque noeud Ai fils de A est activé : $\#Act(Ai) + \#Act(A) - (\#AB + 1)$ fois : chaque fois qu'il est apparu sans les autres Ai ou avec les autres Ai mais sans B [8]. De la même façon que dans le MDLChunker, on ajoute une occurrence aux noeuds Ai, B et A pour définir le chunk $A = \sum Ai + B$ (Algo.13 lignes 5, 6 et 7). Dans le MDLChunker-approché il n'est pas nécessaire de définir les chunks comme cela est fait dans la partie Chunks du MDLChunker, mais cela permet d'utiliser les mêmes fonctions de gain dans les deux MDLChunkers.

Algorithme 13 Création de l'arc (calcul local à chacun des arcs)

1: **si** $U(arc) > 0$ **alors**
2: $attendre\left(\frac{1}{U(arc)} \cdot \mu s\right)$
3: **si** $U(arc) > 0$ **alors**
4: $Cree(arc) \leftarrow vrai$
5: $\forall i, \#Act(Ai) \leftarrow \#Act(Ai) + \#Act(A) - (\#AB + 1) + 1$
6: $\#Act(B) \leftarrow \#\bar{A}B + 1$
7: $\#Act(A) \leftarrow \#AB + 1$
8: $\#Occ(A) \leftarrow \#AB + 1$
9: **finsi**
10: **finsi**

Les figures 5.20 et 5.21 page suivante présentent l'algorithme 13 sur un exemple de chunk ternaire et les figures 5.22 et 5.23 sur un exemple de chunk binaire. Cela illustre en particulier le fonctionnement de la ligne 5 de l'algorithme 13, ainsi que la nécessité de l'attribut **nombre d'occurrences** des noeuds.

Les algorithmes 9 à 13 permettent de remplir la fonction d'optimisation du réseau. Comme évoqué dans l'introduction de ce chapitre, la plus grosse approximation du MDLChunker-approché intervient durant cette phase d'optimisation. Il s'agit de la perte d'information concernant le nombre de cooccurrences entre les noeuds. Le nombre de cooccurrences entre un chunk nouvellement créé et les autres noeuds du réseau ne peut être déduit à partir des informations disponibles localement [9].

8. $\#Act(Ai)$ est le nombre d'apparitions de Ai sans les autres Ai dans l'ancien chunk. $\#Act(A) - (\#AB + 1)$ est le nombre de fois où le chunk A était activé, mais ne le sera plus car B n'est pas présent : c'est Ai qui sera activé.

9. Pour que les informations du réseau restent néanmoins cohérentes, toute augmentation/diminution du nombre d'activations d'un noeud s'accompagne d'une augmentation/diminution proportionnelle des cooccurrences portées par les arcs pères et fils du noeud. Les détails de ce processus ne seront pas présentés pour ne pas compliquer la description de l'algorithme 13

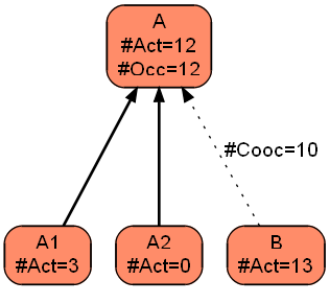


FIGURE 5.20 – Etat du réseau avant la création du chunk AB. L'arc $B \rightarrow A$ n'est pas encore créé. Les noeuds B, A1 et A2 sont apparus 10 fois conjointement. A1 et A2 sont apparu 12 fois conjointement et A1 est apparu 3 fois sans A2.

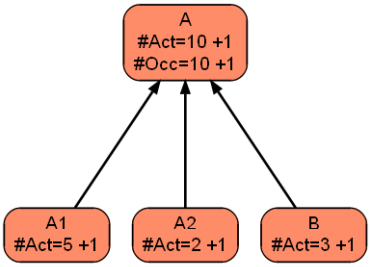


FIGURE 5.21 – Etat du réseau après création du chunk AB. L'arc $B \rightarrow A$ est maintenant créé. A1, A2 et B sont apparus 10 fois conjointement ; A1 5 fois sans A2 et B ; A2 2 fois sans A1 et B ; B 3 fois sans A1 et A2. +1 a été ajouté aux occurrences des noeuds intervenant dans la définition du chunk.

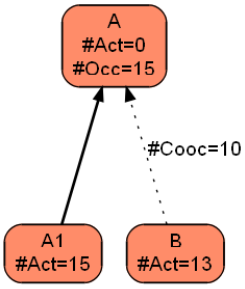


FIGURE 5.22 – Etat du réseau avant la création du chunk AB. L'arc $B \rightarrow A$ n'est pas encore créé. Les noeuds B et A1 sont apparus 10 fois conjointement. Bien que le noeud A n'ait encore jamais été activé, son nombre d'occurrences est de 15.

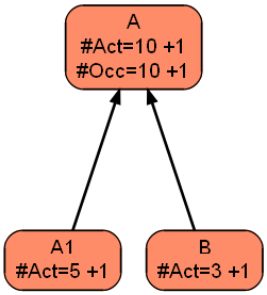


FIGURE 5.23 – Etat du réseau après création du chunk AB. L'arc $B \rightarrow A$ est maintenant créé. A1 et B sont apparus 10 fois conjointement ; A1 5 fois sans B ; B 3 fois sans A1. +1 a été ajouté aux occurrences des noeuds intervenant dans la définition du chunk.

5.2.4 Exemple de fonctionnement

Afin de mieux comprendre les mécanismes mis en jeu par le MDLChunker-approché, nous présentons son fonctionnement sur un exemple concret. Des stimuli pouvant contenir jusqu'à 7 caractères sont générés à partir de la grammaire présentée figure 5.24, puis présentés au MDLChunker-approché. La liste détaillée des stimuli est donnée en annexe B.1.

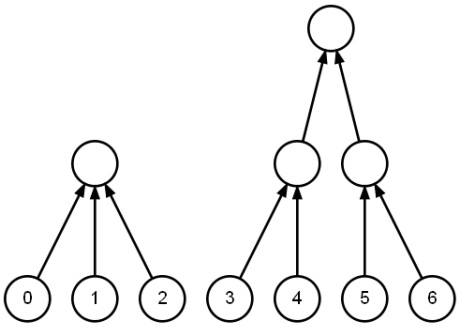


FIGURE 5.24 – Grammaire ayant servi à générer l'ensemble d'apprentissage.

Le réseau doit comporter au moins 3 couches afin de pouvoir représenter les 3 niveaux de la grammaire. Chaque couche comprend 7 noeuds : autant que de caractères-canoniques. Initialement, le réseau est dans l'état de la figure 5.25. Pour permettre le calcul des tailles de codage (Algo.4) aucun effectif ne doit être nul : chaque noeud est donc activé une fois. La valeur initiale de $\#N$ est égale au nombre de noeuds du réseau.

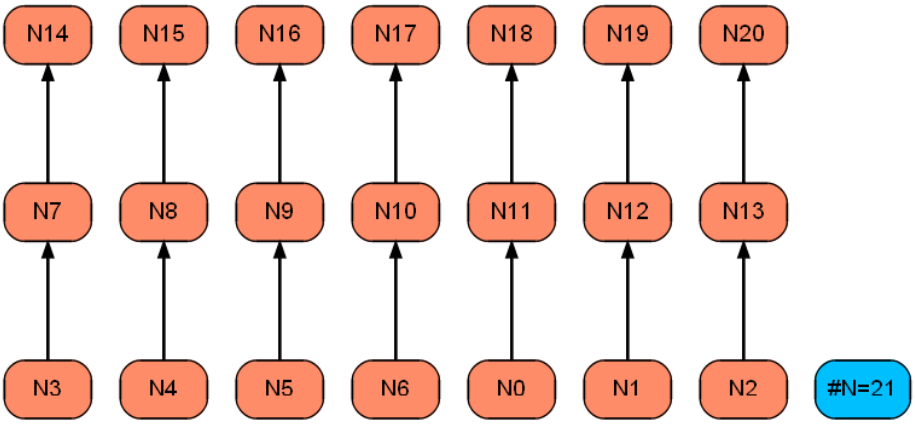


FIGURE 5.25 – Etat initial du réseau comportant 3 couches et 7 caractères-canoniques. Les 21 noeuds sont tous activé une fois : $\#N = 21$

Le premier stimulus est 0, 1, 2. L'utilité des noeuds correspondants (N0, N1, N2) est notée positive (Algo.5). Ils sont donc activés (Algo.7) et leur nombre

d'activations est incrémenté (il passe à 2), faisant augmenter de 3 le nombre total d'activations. Leur taille de codage est localement mise à jour par l'algorithme 4 et vaut $C(N0) = log_2 \frac{28}{2} = 3.8$ bits (Fig. 5.26).

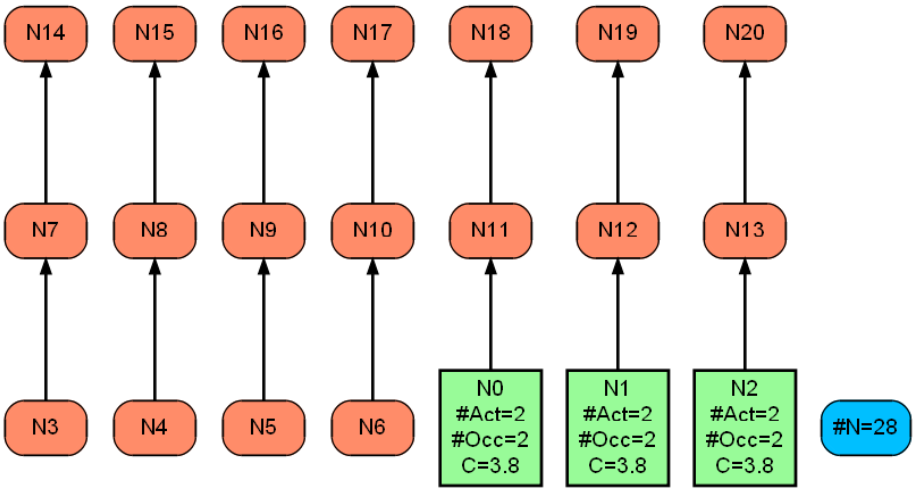


FIGURE 5.26 – Etat du réseau après perception du premier stimulus : 0, 1, 2.

Chaque nouvelle perception d'un stimulus fait évoluer les tailles de codage de l'ensemble des noeuds du réseau. Ceux qui sont activés voient leur taille de codage diminuer tandis que ceux qui ne le sont sont pas voient leur taille de codage augmenter.

La création du premier chunk ($N7 = N3 + N4$) apparaît à la dixième itération. Juste avant la création de l'arc $N4 \rightarrow N7$, le réseau est dans l'état présenté figure 5.27 page ci-contre. Les algorithmes 9, 10 et 11 permettent de calculer le nombre d'occurrences et de cooccurrences des différents noeuds. L'algorithme 12 permet d'en déduire que l'arc $N4 \rightarrow N7$ a une utilité positive.

La figure 5.28 page 116 montre l'état du réseau après création de l'arc $N4 \rightarrow N7$ et factorisation du stimulus. La création de l'arc est effectuée par l'algorithme 13. Le nombre d'activations du noeud N7 passe à 6 (6 = nombre de cooccurrences de N3 et N4 + 1 pour la définition du chunk). Le nombre activations de N3 et N4 passe à 2 (2 = nombre d'occurrences de l'un des noeuds sans l'autre + 1 pour la définition du chunk). Le nombre des cooccurrences portées par les arcs fils de N7 devient nécessairement nul. Le nombre de cooccurrences portées par les arcs pères et fils de N3 et N4 (par exemple $N4 \rightarrow N8$) diminuent/augmentent proportionnellement à la diminution/augmentation du nombre d'activations de N3 et N4 [10].

La modification du nombre d'activations des différents noeuds entraîne une remise à jour de $\#N$ ainsi que des tailles de codage (Algo.4). Les noeuds-

10. Ici, le nombre d'activations de N4 passe de 6 à 2, le nombre de cooccurrences porté par l'arc $N4 \rightarrow N8$ passe donc de 5 à 2 ($5 \cdot \frac{2}{6} = 1.7$ arrondi à 2).

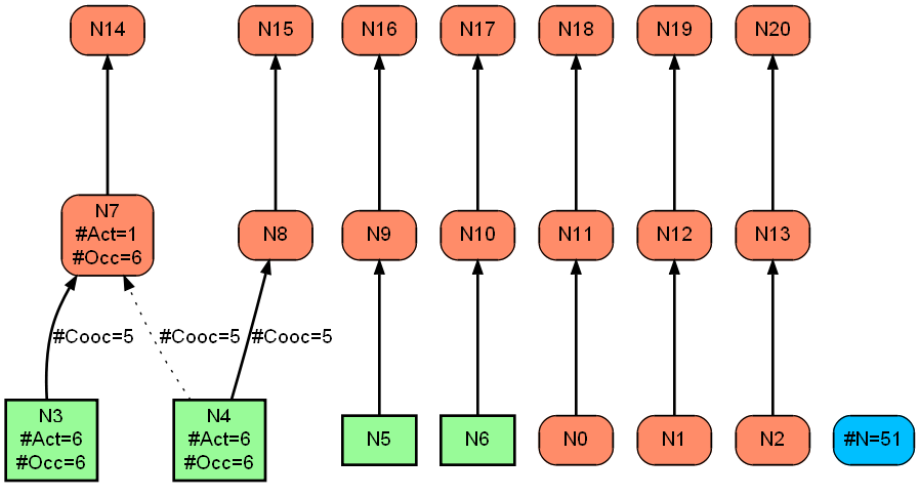


FIGURE 5.27 – Etat du réseau avant la création de l'arc $N4 \rightarrow N7$. Les noeuds N3 et N4 sont apparus chacun 6 fois ($\#Act(N3) = \#Act(N4) = 6$) dont 5 conjointement ($\#Cooc(N4 \rightarrow N7) = 5$).

canoniques N3 et N4 sont factorisés en N7 pour un gain de taille de codage de $4.8 + 4.8 - 3.0 = 6.6$ bits.

De manière similaire, le chunk $N9 = N5 + N6$ est créé a l'issu de la perception du douzième stimulus (Fig. 5.29).

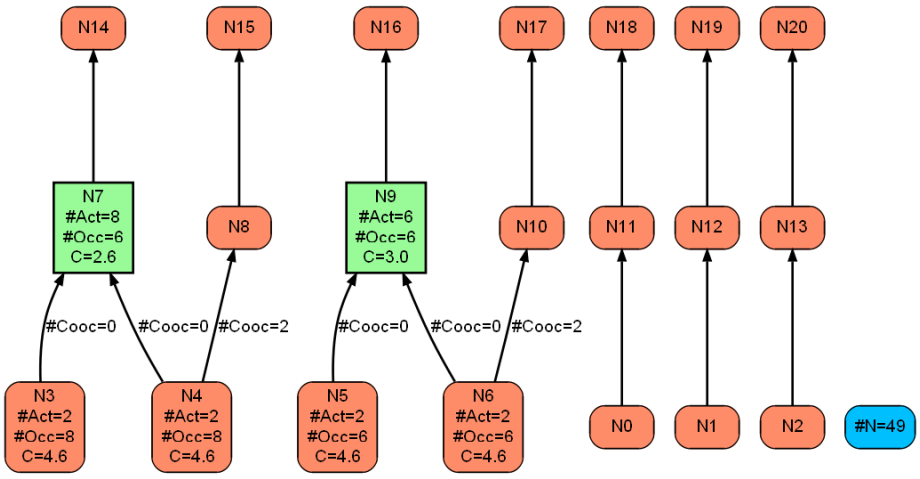


FIGURE 5.29 – Etat du réseau après création du chunk binaire $N7 = N3 + N4$.

Le chunk ternaire $N11 = N0 + N1 + N2$ est créé lors de la perception du treizième stimulus (Fig. 5.30 page suivante). Comme expliqué dans la section sur l'optimisation, un chunk ternaire est en réalité constitué de deux chunk bi-

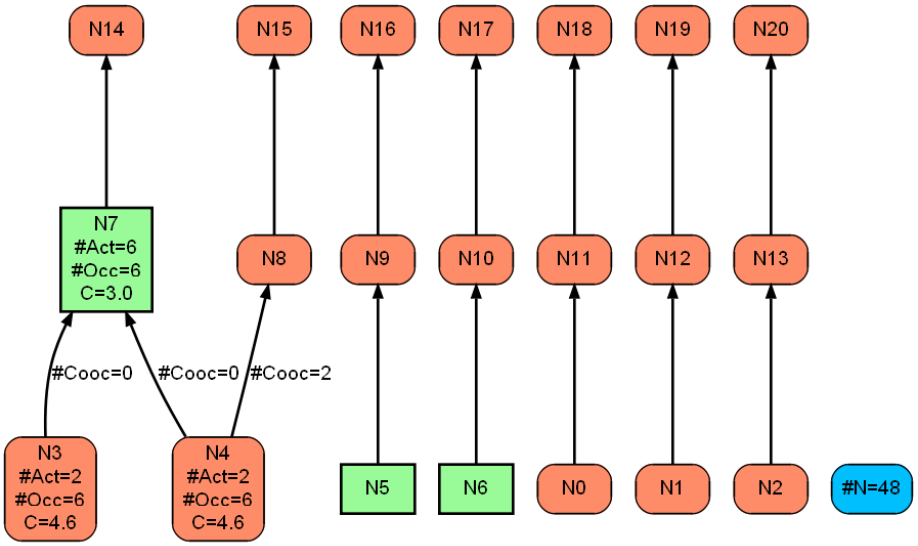


FIGURE 5.28 – Etat du réseau après création du chunk $N7 = N3 + N4$. Les deux noeuds N3 et N4 se trouvent factorisés en un seul noeud N7.

naires successifs. Dans notre cas $N11 = ((N0 + N1) + N2)$, ce qui explique que le nombre d'activations de N2 soit inférieur au nombre d'activations de N0 et N1. En effet, N2 n'apparaît que dans la définition du « second » chunk tandis que N0 et N1 apparaissent dans les deux définitions : $N11 = N0 + N1$ puis $N11 = N0 + N1 + N2$.

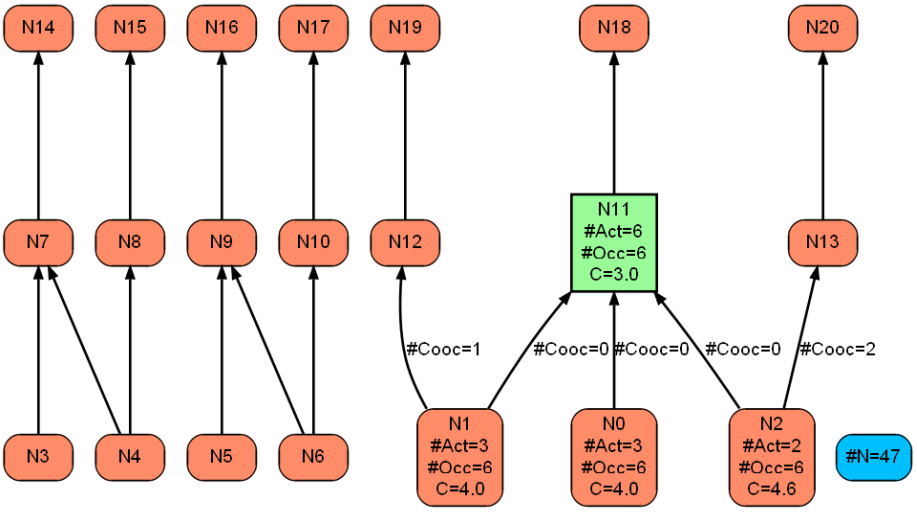


FIGURE 5.30 – Etat du réseau après création du chunk ternaire $N11 = N0 + N1 + N2$.

Le chunk de second niveau $N14 = N7 + N9$ est créé lors de la perception du

stimulus 24 (Fig. 5.31). On retrouve ainsi la grammaire initiale (Fig. 5.32).

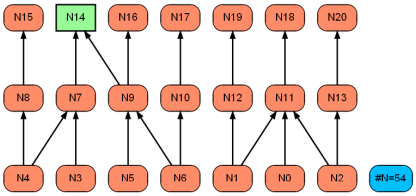


FIGURE 5.31 – Etat du réseau après création du chunk $N14 = N7 + N9$.

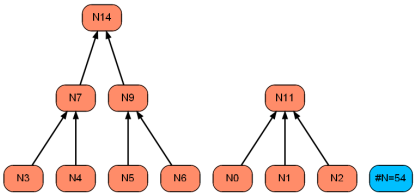


FIGURE 5.32 – Réseau de la figure de gauche dans lequel les noeuds ne possédant qu'un fils ne sont pas représentés.

5.3 Comparaison avec le MDLChunker

Comme cela a été expliqué, la différence majeure entre les deux chunkers réside dans l'absence de représentation explicite de la partie Stimuli|Chunks dans le MDLChunker. La conséquence de cette différence est que chaque création de chunk entraîne une perte d'information. Le nombre exact de cooccurrences du nouveau chunk avec les autres noeuds du réseau ne peut être connu. Cette information n'a pas été extraite des données passées et n'est plus disponible à cause de l'absence de représentation explicite de la partie Stimuli|Chunks.

La seconde différence provient du fait que les calculs doivent être effectués à partir d'informations uniquement locales, l'algorithme de factorisation (A*) du MDLChunker doit donc être dégradé. Il est remplacé par une version simplifiée choisissant la solution de plus forte utilité à chaque pas de temps sans possibilité de retour en arrière (Fig. 5.14 page 103).

En résumé, les deux opérations principales que sont l'optimisation et la factorisation se trouvent dégradées par rapport à leurs homologues du MDLChunker. Cela conduit à des factorisations sous-optimales [11] dans certains cas, ainsi qu'une création systématiquement plus lente des chunks.

Afin d'illustrer ces différences, les résultats du MDLChunker-approché sont comparées avec ceux du MDLChunker sur l'exemple de la section 5.2.4. L'ensemble d'apprentissage est constitué de 25 stimuli (liste détaillée en annexe B.1). Les 4 chunks qui constituent la grammaire initiale (Fig. 5.24 page 113) sont créés par les deux chunkers. Aucun autre « mauvais » chunk n'est créé. Le tableau 5.1 page suivante présente les tailles de codage de ces 4 chunks au cours de l'apprentissage.

11. Il s'agit de sous-optimalité par rapport aux algorithmes du MDLChunker qui eux-mêmes convergent dans certains cas vers des solutions sous-optimales.

		Stimuli							
Modèle	Chunk	10	11	12	13	14	15	16	17
MDLChunker	(0 1 2)	···	···	···	1.8	1.8	1.9	1.9	2.0
	(3 4)	1.7	1.6	1.5	1.5	1.4	1.5	1.4	1.4
	(5 6)	···	···	1.8	1.8	1.7	1.6	1.5	1.5
	((3 4)(5 6))	···	···	···	···	···	···	···	···
MDLChunker-approché	(0 1 2)	···	···	···	3.0	3.0	3.1	3.1	3.2
	(3 4)	3.0	2.8	2.6	2.6	2.4	2.5	2.4	2.3
	(5 6)	···	···	3.0	3.0	2.8	2.6	2.5	2.4
	((3 4)(5 6))	···	···	···	···	···	···	···	···

		Stimuli							
Modèle	Chunk	18	19	20	21	22	23	24	25
MDLChunker	(0 1 2)	1.9	1.9	1.9	2.0	1.8	1.9	1.9	1.9
	(3 4)	2.1	2.1	2.1	2.2	2.2	2.2	2.2	2.2
	(5 6)	2.3	2.1	2.1	2.2	2.2	2.2	2.2	2.2
	((3 4)(5 6))	1.5	1.5	1.4	1.4	1.4	1.3	1.3	1.2
MDLChunker-approché	(0 1 2)	3.2	3.2	3.3	3.3	3.1	3.2	2.9	3.0
	(3 4)	2.2	2.2	2.2	2.1	2.1	2.1	4.8	4.8
	(5 6)	2.3	2.2	2.2	2.1	2.1	2.1	4.8	4.8
	((3 4)(5 6))	···	···	···	···	···	···	2.6	2.5

TABLE 5.1 – Comparaison du décours temporel de la création des chunks et de leur taille pour le MDLChunker et le MDLChunker-approché. Les valeurs indiquées dans le tableau sont les tailles de codages des 4 chunks de la grammaire initiale.

Décours temporel de la création des chunks

Chunks de premier niveau hiérarchique (0 1 2), (3 4) et (5 6) : Les deux chunks binaires (3 4) et (5 6) sont respectivement créés aux itérations 10 et 12 et de façon simultanée par les deux chunkers. Il en va de même pour le chunk ternaire (0 1 2) créé à l'itération 13 dans les deux cas. Le décours temporel de la création des chunks du premier niveau hiérarchique est donc identique pour les deux chunkers car la fonction de gain est identique.

Chunk du second niveau hiérarchique ((3 4)(5 6)) : Le chunk ((3 4)(5 6)) est créé plus tard par le MDLChunker-approché (itération 24) que par le MDLChunker (itération 18). La création plus lente des chunks est la limitation majeure du MDLChunker-approché. Lors de l'apparition des chunks (3 4) et (5 6), le nombre de cooccurrences de chacun de ces chunks avec les autres noeuds du réseau n'est pas connu par le MDLChunker-approché (en particulier les co-occurrences de (3 4) avec (5 6)). L'association de (3 4) avec (5 6) est apprise uniquement à partir des stimuli perçu après la création de ces deux chunks : les stimuli 12 à 24. Les chunks de second niveau apparaissent donc plus tard que pour le MDLChunker qui a la possibilité d'utiliser les informations contenues dans les 12 premiers stimuli. (3 4) et (5 6) apparaissent exactement 4 fois dans les 12 premiers stimuli, il manque donc 4 cooccurrences de ces chunks pour compenser ce manque : il y a bien 4 cooccurrences de (3 4) et (5 6) entre les stimuli 12 et 24.

Tailles de codage

L'effet d'initialisation : Le nombre de noeuds étant initialement différent [12] entre les deux chunkers et le nombre d'occurrences de chaque noeud du réseau étant initialisé à 1, les tailles de codages des noeuds impliqués dans les premiers stimuli sont différentes. Ces différences (1.7 contre 3.0 pour (3 4) à l'itération 10) tendent à s'estomper progressivement au cours des itérations (1.4 contre 2.3 à l'itération 17).

Chunk du second niveau hiérarchique : La taille de codage élevée du chunk ((3 4)(5 6)) de second niveau (2.5 contre 1.2 à l'itération 25) s'explique par le fait que ce chunk apparaît moins souvent pour le MDLChunker-approché puisqu'il est construit uniquement à partir des cooccurrences postérieures à l'itération 12.

12. Initialement le MDLChunker-approché comprend $nbNoeudsCanoniques \times nbCouches$ noeuds, tandis que le MDLChunker ne contient que les noeuds-canoniques.

5.4 Conclusion

Ce chapitre plus technique présente en détail une nouvelle implémentation du principe visant à associer MDL et *chunking*. Cette implémentation diffère de celle du chapitre 4 pour plusieurs raisons.

Au niveau de son fonctionnement, les calculs sont effectués localement et en parallèle. Les performances du MDLChunker-approché s'en trouvent dégradées par rapport à celles du MDLChunker. Les deux différences majeures sont la création plus lente des chunks durant la phase d'optimisation, ainsi qu'une sous-optimalité de l'algorithme de factorisation due à l'absence de représentation explicite des données.

D'un point de vue théorique, les buts poursuivis sont différents. Il s'agit d'une tentative pour établir la robustesse du principe étudié vis-à-vis de son implémentation, en montrant qu'une implémentation basée sur une architecture totalement différente permet de reproduire des résultats similaires. Un second objectif est de montrer que la forte complexité des algorithmes utilisés au chapitre 4 n'est pas une nécessité intrinsèque au principe étudié. Une implémentation n'effectuant que des calculs locaux peut être trouvée : le MDLChunker-approché en est une (parmi de nombreuses autres). Au travers de l'implémentation, c'est la plausibilité cognitive du principe que l'on cherche à valider. Le MDLChunker-approché doit donc conserver une place de second ordre par rapport au MDLChunker qui est une implémentation plus « directe » du principe étudié.

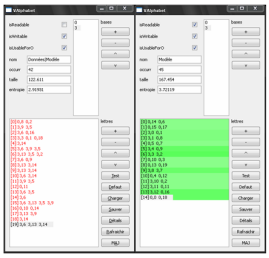
Une comparaison plus systématique est nécessaire. Il est pour cela indispensable d'établir une mesure de distance entre les résultats fournis par les deux MDLChunkers. Ceci afin que les performances puissent être testées sur différentes grammaires.

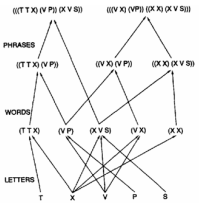
Le problème est que, quelle que soit la mesure de distance choisie, elle n'aura de signification qu'en comparaison des résultats obtenus par des participants humains. Les performances des deux MDLChunkers seront comparables si les distances entre leurs résultats sont du même ordre que les distances les séparant des résultats obtenus par des participants humains. Le décours temporel de création des chunks sur d'autres grammaires est étudié plus en détail dans le chapitre 7. C'est ce chapitre qui va permettre de valider expérimentalement le fonctionnement des deux MDLChunkers.

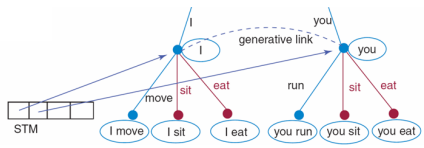
Chapitre 6

Comparaison théorique avec des modèles existants

Sommaire







6.1 Introduction

Avant d'entreprendre la validation expérimentale du MDLChunker, il est nécessaire d'effectuer une comparaison avec d'autres modèles existants, considérés dans la littérature comme des références en termes de performance et de plausibilité cognitive. La comparaison effectuée dans ce chapitre est essentiellement théorique. Il ne s'agit pas d'appliquer le MDLChunker aux données utilisées pour la validation des autres modèles (ce sera l'objet du chapitre 8), mais de comparer les similitudes et différences existant entre les architectures utilisées. Six modèles de référence ont été choisis : trois modèles de *chunking* et trois modèles basés sur le MDL. Ces modèles dont les champs d'application vont de la segmentation de mots jusqu'à l'expertise dans le domaine du jeu d'échecs, ont tous pour but la modélisation d'une, ou plusieurs tâches cognitives spécifiques.

La plupart des modèles décrits dans ce chapitre sont le fruit de nombreuses années de travail et sont trop riches pour pouvoir être décrits en détail. Une compréhension approfondie de leur architecture nécessiterait notamment d'illustrer leur fonctionnement par des exemples. C'est pourquoi la comparaison qui est faite avec le MDLChunker porte sur les principes généraux de leur fonctionnement plutôt que sur l'implémentation technique qui en est faite.

Pour certains de ces modèles, une comparaison détaillée avec le MDLChunker a été ajoutée en annexe A. Cette comparaison porte sur des détails techniques et s'adresse uniquement au lecteur ayant une connaissance approfondie du fonctionnement de ces modèles. Elle a pour but de répondre à d'éventuelles questions qui pourraient se poser. Pour plus de simplicité et pour faciliter la comparaison des différents modèles, le vocabulaire utilisé est systématiquement celui du MDLChunker.

Les six modèles choisi sont :
- Le *Competitive Chunker* (Servan-Schreiber & Anderson, 1990) qui est un modèle de *chunking* que la simplicité de fonctionnement rend aisément comparable au MDLChunker.
- PARSER (Perruchet & Vinter, 1998) qui est également un modèle de référence en matière de *chunking* et dont certains résultats seront analysés en détail au chapitre 8.
- CHREST (Gobet, 1993) qui est le troisième modèle de *chunking* présenté et qui a été choisi pour la richesse de son architecture. Il s'agit d'un modèle beaucoup plus riche et complexe que le MDLChunker, ce qui permet de mieux appréhender les limites de ce dernier.
- Le modèle de Brent et Cartwright (1996), qui comme PARSER, est appliqué à la segmentation de mots, mais qui utilise le MDL comme principe de fonctionnement.
- Le modèle de Goldsmith (2001) qui utilise le MDL pour extraire les suffixes les plus fréquents que l'on peut observer dans une langue.
- Le modèle *Simplicity and Power* (Wolff, 2006) qui de tous est celui qui se rapproche le plus du MDLChunker. Bien que les motivations générales des deux modèles soient quelque peu différentes, il est possible que faire quelques rapprochements techniques intéressants.

Les présentations qui sont faites de chacun de ces modèles sont indépendantes. Le lecteur peut donc à sa guise se reporter aux modèles de son choix et ne lire qu'une partie du chapitre. Il est néanmoins conseillé de lire la description de PARSER (section 6.3) avant la lecture du chapitre 8.

6.2 Competitive Chunker

Description générale

Le modèle computationnel proposé par Servan-Schreiber et Anderson (1990) porte le nom de *Competitive Chunker* (CC). Il s'inspire de l'idée de Miller (1958) selon laquelle des participants sont capables de mémoriser une grammaire en groupant les parties fréquentes, puis en les recodant (*group and recode*) : idée qui trouve son origine dans la notion de *chunk* introduite par Miller (1956). L'hypothèse testée par CC est que les contraintes grammaticales peuvent être représentées sous la forme d'une hiérarchie de chunks.

L'expérience proposée pour la validation consiste à entraîner le modèle et des participants humains en leur demandant de mémoriser un ensemble de phrases issues d'une grammaire artificielle. Les phrases produites par la grammaire sont du type TPPPTS, TPPTXVPS, TPPTXXVS, TPTXXVPS (voir figure 7.1 page 151 pour plus de détail). À l'issue de cette phase d'entraînement, participants et modèle sont testés sur une tâche de discrimination comportant des phrases grammaticales et des phrases générées aléatoirement. Cette tâche a pour but de vérifier l'hypothèse selon laquelle une phrase paraît d'autant plus familière qu'elle contient un nombre élevé de chunks connus.

Fonctionnement

Chaque stimulus perçu par CC est décomposé en chunks. Par exemple, si CC possède les chunks TPP, XX et VS, le stimulus TPPTXXVS est décomposé en TPP T XX VS. En cas d'ambiguïté dans la décomposition (si par exemple CC possède également le chunk XXV) la factorisation se fait de façon probabiliste, chaque chunk ayant une certaine probabilité d'être choisi. La probabilité d'un chunk est définie au moment de sa création. Elle dépend des utilités de ses deux chunks fils (voir annexe A.2 pour plus de détail).

Pour chaque nouveau stimulus perçu, un chunk binaire associant deux chunks adjacents est créé. Dans notre exemple, trois chunks sont candidats à la création TPP T, T XX et XX VS. Le chunk effectivement créé est celui dont la probabilité est la plus élevée. En supposant qu'il s'agisse de XX VS, alors le chunk ((XX) (VS)) est ajouté et le processus recommence avec un nouveau stimulus.

Comparaison avec le MDLChunker

L'architecture générale de CC et celle du MDLChunker sont similaires. Dans les deux cas, il s'agit d'une hiérarchie de chunks binaires dans laquelle chaque chunk connaît ses fils directs. Dans les deux cas, un certain nombre de chunks-canoniques (*elementary chunks*) définissent un langage de description et sont supposés connus dès le début de l'apprentissage.

La décomposition de chaque stimulus en chunks est un mécanisme probabiliste pour CC, tandis que son équivalent pour le MDLChunker (la factorisation) est déterministe. La décomposition opérée par CC est plus simple que la factorisation du MDLChunker. Elle nécessite en retour de moyenner les résultats sur plusieurs simulations afin de diminuer le bruit introduit par la nature probabiliste de la décomposition.

La seconde différence concerne le mécanisme de création de chunks qui est systématique dans le cas de CC : un chunk et un seul est créé pour chaque nouveau stimulus perçu. Cette création systématique des chunks est en partie compensée par un mécanisme d'oubli qui supprime les chunks trop peu fréquents (voir annexe A.2 pour plus de détail). Les chunks existant à un instant donné sont fortement sujets à disparaître. Ils sont très dépendants des derniers stimuli perçus, et ne sont donc pas nécessairement de bon prédicteurs des chunks possédés quelques itérations plus tard. À l'inverse dans le MDLChunker, un chunk n'est créé que s'il apporte une contribution notable à la description des stimuli (cette contribution est matérialisée par un gain en terme de taille de codage). Un chunk correspond dans ce cas à une connaissance relativement stable dans le temps : il est peu sujet à variations et n'est susceptible de disparaître que si un changement important intervient dans le processus observé.

Le nombre de paramètres du modèle est une autre différence entre CC et le MDLChunker. L'information nécessaire pour fixer les paramètres de CC provient des résultats des participants. À l'inverse, l'absence de paramètres du MDLChunker lui permet de ne pas utiliser cette information. Comme CC est validé sur sa capacité à reproduire les résultats des participants, son pouvoir explicatif devrait être amputé de la quantité d'information utilisée pour ajuster ses paramètres. Nous verrons dans l'introduction du chapitre 7 que la quantité d'information recueillie dans notre expérience est justement faible en comparaison de celle qui est introduite à travers les paramètres du *Competitive Chunker*. Une comparaison plus systématique des points communs et différences entre CC et le MDLChunker est présentée en annexe A.2.

6.3 PARSER

Description générale

PARSER est un modèle computationnel créé par Perruchet et Vinter (1998) pour rendre compte de l'acquisition des mots à partir du langage oral. Ce der-

nier peut être considéré en première approximation comme un flux continu de syllabes, et bien que l'acquisition des mots puisse être en partie guidée par la présence d'indices prosodiques ou phonologiques, Saffran, Newport, et Aslin (1996) ont montré que la présence de tels indices n'étaient pas indispensable à la segmentation, et que l'extraction de mots pouvait se faire sur la base de régularités statistiques uniquement. C'est pour modéliser ce phénomène que PARSER a été conçu.

L'expérience utilisée pour la validation de PARSER sera décrite plus en détail au chapitre 8. Elle consiste en une reprise de l'expérience de Saffran et al. (1996) dans laquelle des participants adultes sont soumis à une suite ininterrompue de syllabes lues par un synthétiseur vocal. Cette suite de syllabes correspond en réalité à une concaténation aléatoire de mots artificiels (*patubi*, *pidabu*, *dutaba*, etc.). La capacité des participants et de PARSER à segmenter cette suite en mots est évaluée par une tâche de discrimination forcée dans laquelle sont proposés simultanément des mots de la grammaire et des concaténations aléatoires de syllabes (*patubi* contre *bubuta*, etc.).

Fonctionnement

Le fonctionnement du MDLChunker comme celui de PARSER se base sur l'utilisation de chunks organisés sous forme hiérarchique. L'architecture générale de PARSER est relativement différente de celle du MDLChunker, mais il est possible d'établir une correspondance assez fine entre les mécanismes utilisés par les deux modèles.

PARSER segmente le flux continu de syllabes en stimuli de longueur variable, et crée un nouveau chunk pour chaque stimulus non familier (Fig. 6.1 page suivante). Un mécanisme d'oubli fait décroître en permanence l'utilité des chunks. À cela s'ajoute un mécanisme d'interférence pénalisant les chunks de faible longueur qui sont inclus dans des chunks plus longs. Par exemple *pa* et *tubi* sont pénalisés à chaque utilisation de *patubi*. Le diagramme résumant le fonctionnement de PARSER est présenté figure 6.2 page suivante.

Comparaison avec le MDLChunker

La combinaison des trois mécanismes (création, oubli, interférence) a un effet similaire au mécanisme d'optimisation du MDLChunker. Le couple création-oubli permet de ne maintenir à un niveau d'utilité suffisant que les chunks dont la fréquence est suffisamment élevée. Les chunks trop peu fréquents sont créés puis oubliés après quelques itérations. La valeur du paramètre fixant le taux d'oubli relativement au taux de réactivation d'un chunk, permet implicitement de définir la fréquence minimale à laquelle un chunk doit être réactivé pour pouvoir perdurer. De façon relativement similaire, le mécanisme d'optimisation du MDLChunker crée un chunk lorsque sa fréquence est suffisamment élevée pour compenser le coût en terme de taille de codage qu'induit la création du chunk. Le mécanisme d'interférence de PARSER qui pénalise les chunks de

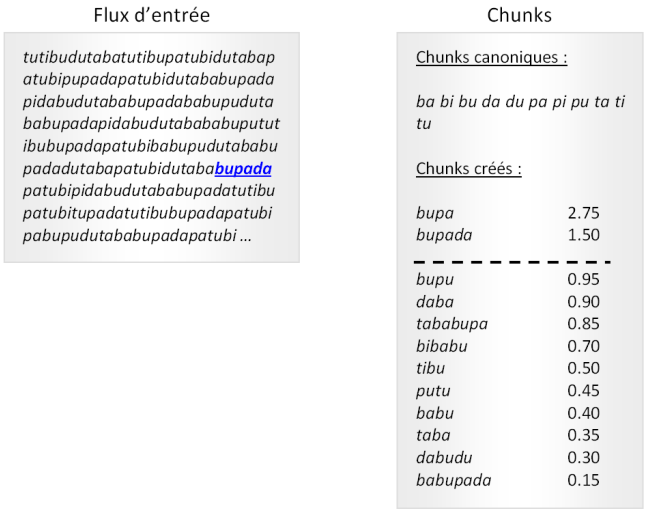


FIGURE 6.1 – Représentation de l'architecture de PARSER. Sur la partie gauche est représenté le flux continu de syllabes utilisé en entrée. Le stimulus courant est souligné. Sur la partie droite sont représentés les différents chunks créés. Le seuil de familiarité est matérialisé par un trait pointillé séparant les chunks de faible utilité (<1) des chunks de forte utilité (>1).

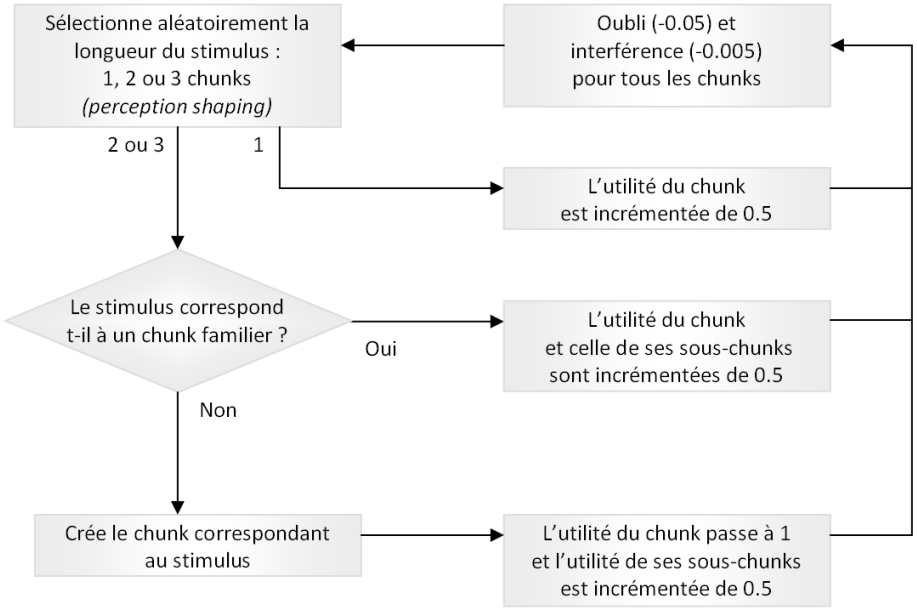


FIGURE 6.2 – Diagramme du fonctionnement de PARSER tiré de Perruchet et Vinter (1998).

faible taille au profit des chunks qui les contiennent est naturellement intégré au mécanisme d'optimisation du MDLChunker : lorsqu'un chunk ABC est créé, toutes les occurrences de AB C ou A BC sont remplacées par ABC. Les chunks de petite taille tendent ainsi à disparaître au profit des plus grands qui les contiennent.

De façon très simple, PARSER propose une conjonction de trois mécanismes (création, oubli, interférence) dont l'association est sensiblement équivalente au mécanisme d'optimisation du MDLChunker. L'optimisation réalisée par le MDLChunker se fonde sur un principe théorique cognitivement plausible (principe de simplicité), alors que le mécanisme de création-oubli-interférence de PARSER se fonde sur une implémentation cognitivement plus plausible, car moins coûteuse en mémoire comme en calculs. La similarité des résultats obtenus par les deux modèles (voir chapitre 8), suggère que ces mécanismes sont sensiblements équivalents en pratique, ce qui fournit une implémentation possible des principes du MDLChunker et permet de justifier de façon théorique les mécanismes utilisés par PARSER.

PARSER utilise également un mécanisme de *perception shaping* qui tient compte des chunks déjà créés pour guider la segmentation du flux continu de syllabes. De façon similaire, le mécanisme de factorisation du MDLChunker permet de découper le stimulus courant en chunks, sur la base des chunks déjà créés. La différence réside essentiellement dans le critère utilisé : PARSER segmente l'entrée en choisissant systématiquement les chunks les plus longs, et le MDLChunker factorise le stimulus de façon à ce que sa taille de codage soit minimale.

Il y a une très forte adéquation entre les mécanismes utilisés par les deux modèles. Ceux du MDLChunker possèdent une justification théorique simple, mais nécessitent en pratique une quantité de calculs importante. La situation se trouve inversée dans le cas de PARSER. La plausibilité cognitive (théorique et pratique) des deux modèles se trouve renforcée par la similarité de leurs comportements. Une comparaison plus détaillée des différents points communs et différences entre PARSER et le MDLChunker est présentée en annexe A.3.

6.4 CHREST

Description générale

Contrairement au *Competitive Chunker*, à PARSER et au MDLChunker, CHREST (Gobet, 1993) est beaucoup plus qu'un modèle de *chunking*. Il s'agit d'une architecture complète intégrant une modélisation de la mémoire à long terme, de la mémoire à court terme ainsi que de certains paramètres physiologiques comme les temps de traitement associés à certaines tâches. CHREST (*Chunk Hierarchy and REtrieval STructures*) est une amélioration du modèle EPAM (*Elementary Perceiver And Memorizer*) (Feigenbaum & Simon, 1984).

Basé sur cette architecture très complète, CHREST a pu être appliqué à des domaines aussi différents que l'acquisition du vocabulaire (Jones, Gobet, & Pine,

2000), l'acquisition des catégories syntaxiques (Croker, Pine, & Gobet, 2000) ou la perception et la mémoire de joueurs d'échecs (de Groot et al., 1996). La comparaison du MDLChunker avec CHREST est surtout intéressante dans la mesure où elle permet de mieux voir la nature des améliorations qui pourraient être apportées au MDLChunker, de la même façon que CHREST a permis de faire évoluer les principes de base utilisés dans EPAM.

Fonctionnement

L'architecture de CHREST est basée sur l'utilisation de réseaux de discrimination. Un réseau de discrimination est une hiérarchie de tests permettant de classer différents objets. À chaque noeud du réseau est associé un test de façon à ce qu'un objet partant de la racine subisse une succession de tests lui permettant d'être assigné à la feuille qui lui correspond (Fig. 6.3). Il s'agit d'un principe similaire à celui des arbres de classification (Breiman, Friedman, Olshen, & Stone, 1984), la différence étant que les tests peuvent porter sur n'importe quelle propriété de l'objet et pas nécessairement sur des variables numériques. Comme cela est illustré figure 6.3, les tests réalisés pour passer d'un noeud à son noeud fils peuvent être n-aires [1].

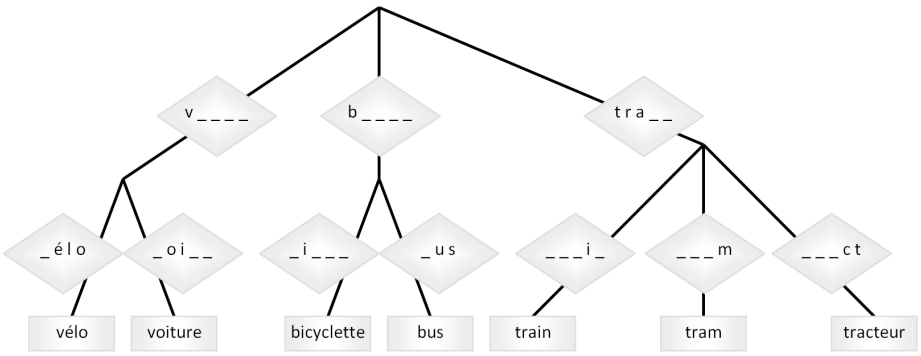


FIGURE 6.3 – Exemple d'un réseau de discrimination dans le cas où les objets sont des mots. Les tests sont matérialisés par des losanges et les feuilles par des rectangles. L'objet « tractopelle » par exemple serait classé dans la même catégorie que « tracteur » .

Le rôle d'un réseau de discrimination est de matérialiser sous forme de tests l'information nécessaire pour séparer différents objets. À l'inverse, l'information présente dans les noeuds du réseau est celle qui n'est pas utile pour séparer les objets. Cette information correspond aux chunks. On retrouve là sous une forme totalement différente la notion de compression à base de chunks. Par exemple, le suffixe « inter » des quatre mots « international » , « interopérabilité » , « intermédiaire » et « interstice » n'apporte pas d'information pour les séparer. Elle peut donc être « chunkée » . C'est le cas également de la première lettre des quatre mots « chien » , « chat » , « crapaud » et « chimpanzé »

1. C'est une des améliorations apportées par CHREST par rapport à EPAM.

. Les deux réseaux de discrimination sont similaires [2] dans les deux cas, que le chunk contienne une ou quatre lettres.

Afin que les chunks créés puissent contenir un maximum d'information redondante (c'est le rôle d'un chunk), il est nécessaire que les tests n'utilisent que le minimum d'information nécessaire à la séparation des différents objets présentés au réseau. En tenant compte de cette contrainte supplémentaire de minimisation de l'information utilisée par les tests, le réseau de la figure 6.3 devient celui présenté figure 6.4.

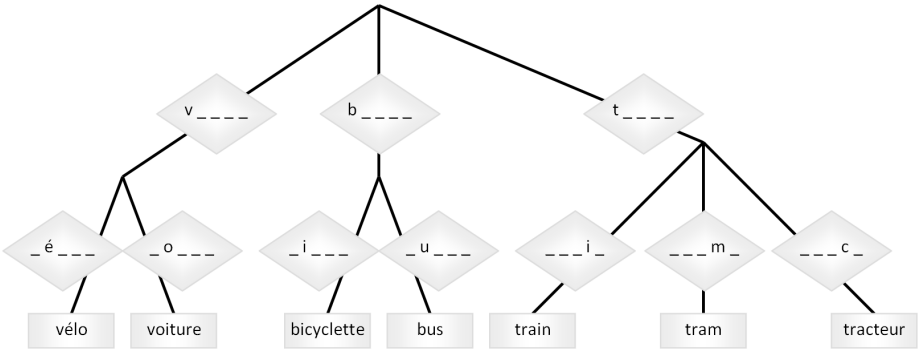


FIGURE 6.4 – Réseau de la figure 6.3 utilisant les contraintes minimales permettant la séparation des objets

Dans CHREST, chaque noeud du réseau comprend à la fois un test et une « image » (Fig. 6.5 page suivante). Une image est un objet partiel compatible avec les tests situés en amont. Cet objet partiel peut contenir plus ou moins d'information que l'information minimale compatible avec la série de tests menant au noeud. Le contenu d'une image est susceptible d'évoluer au cours du temps en fonction des différents objets [3] utilisés lors de l'apprentissage du réseau.

Le réseau de discrimination de CHREST est créé de façon itérative. Cette évolution du réseau se fait sous la gouverne de deux mécanismes. Si un objet passe un certain nombre de tests et se retrouve classé avec une image (objet partiel) qui est incompatible avec lui, alors c'est que la contrainte imposée par les tests est trop lâche, et un nouveau test est ajouté pour discriminer l'objet de l'image. C'est le mécanisme de discrimination. À l'inverse, si l'objet est classé avec une image qui est compatible avec lui, à moins d'être identique à l'image, l'objet apporte nécessairement une information supplémentaire qui ne joue aucun rôle dans la discrimination, cette information est donc ajoutée à l'image. C'est le mécanisme de familiarisation (Fig. 6.6 page suivante).

Dans l'architecture générale de CHREST (Fig. 6.7 page 131) les réseaux de discrimination sont utilisés pour modéliser la mémoire à long terme. Chaque modalité sensorielle (visuelle, auditive) se voit associer un réseau de discrimination. CHREST comporte également une modélisation de la mémoire à court

2. Ils sont similaires au niveau de leur arité. Dans le premier cas les tests pourraient être _____n, _____o, _____m, _____s, et dans le second __i__, __a__, _r___, _h___.

3. La notion d'objet de CHREST correspond à ce que nous avons appelé « stimuli » dans le MDLChunker.

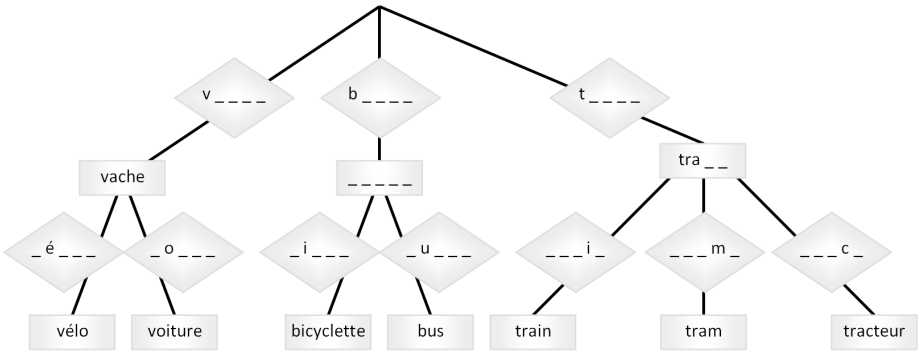


FIGURE 6.5 – Réseau de la figure 6.4 dans lequel les images des noeuds sont représentées (rectangles). L'image tra_ _ contient plus que l'information minimale imposée par les tests en amont, et l'image _ _ _ _ _ _ en contient moins.

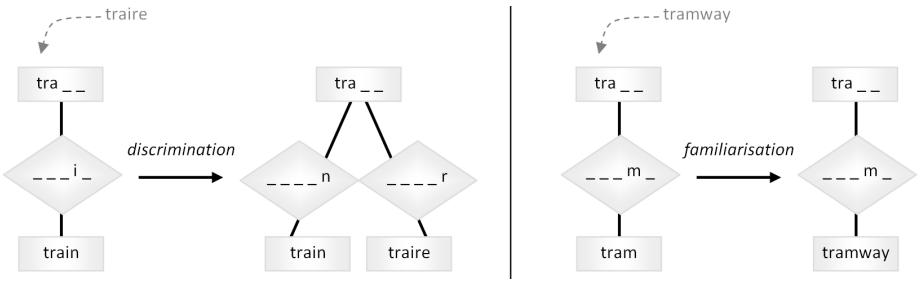


FIGURE 6.6 – Exemple de mécanisme de discrimination et de familiarisation pour le réseau de la figure 6.5. L'ajout de l'objet « traire » conduit à l'ajout d'un test sur la cinquième lettre afin de séparer « traire » de « train ». L'ajout de l'objet « tramway » conduit à une familiarisation de « tram » en « tramway ».

terme inspirée de la séparation proposée par Baddeley et Hitch (1974). La partie correspondant à la modalité auditive est modélisée par une boucle phonologique et la partie correspondant à la modalité visuelle par un calepin visuo-spatial.

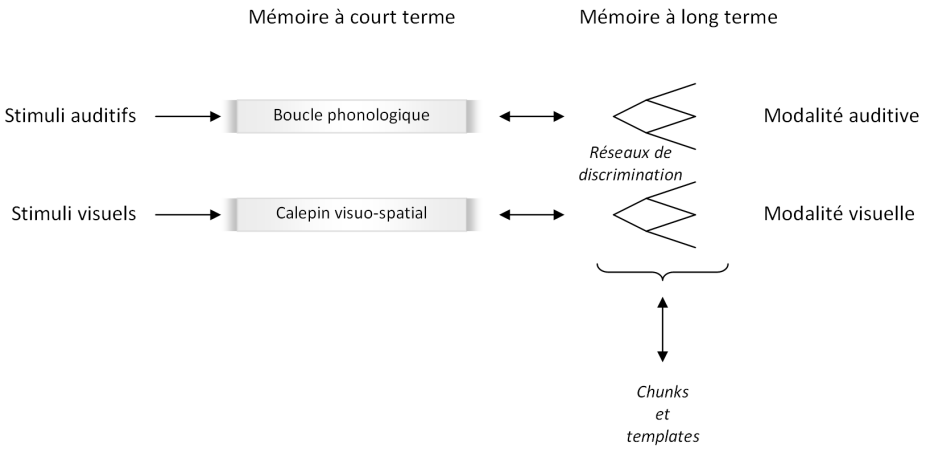


FIGURE 6.7 – Architecture de CHREST tirée de Gobet (2001). La liste des modalités sensorielles modélisées n'est pas exhaustive : il existe notamment un réseau de production (non représenté) associé à la modalité motrice.

L'apport majeur de CHREST consiste à permettre la création d'associations sémantiques grâce à des « liens latéraux » reliant certains noeuds dans le réseau de discrimination (Gobet, 1996). Un lien latéral est susceptible d'être créé entre deux noeuds apparaissant ensemble en mémoire à court terme. De tels liens permettent de matérialiser à l'intérieur du réseau la proximité temporelle ou spatiale entre noeuds. Cette information est mise en relation avec l'image du noeud menant ainsi à la création de divers types de liens (Fig. 6.8 page suivante) :

- **Les liens d'équivalence** permettent de relier les différents noeuds dont les images sont identiques.
- **Les liens de similarité** [4] permettent d'exprimer la proximité entre deux noeuds d'un même réseau dont les images sont similaires, c'est-à-dire qu'elles partagent un certain nombre de propriétés en commun [5].
- **Les liens génératifs** permettent d'exprimer la similitude des tests effectués pour accéder aux fils du noeud.
- **Les liens de production** permettent d'exprimer une relation causale (lien orienté) entre une situation observée et une action.

Les trois premiers types de liens permettent un rapprochement des informations similaires disséminées au sein du réseau. Ils sont à la base de la création des *templates*, qui peuvent être vus comme l'équivalent informatique de la notion de schéma en psychologie. Si un chunk est une conjonction de propriétés d'un objet, alors un *template* est une conjonction de disjonctions de propriétés (Fig. 5.5 page 95). Pour dire les choses autrement, dans un chunk l'information

4. Qui semblent être également appelés liens de redondance.
5. Ce lien correspond à un relâchement des contraintes utilisées pour le lien d'équivalence.

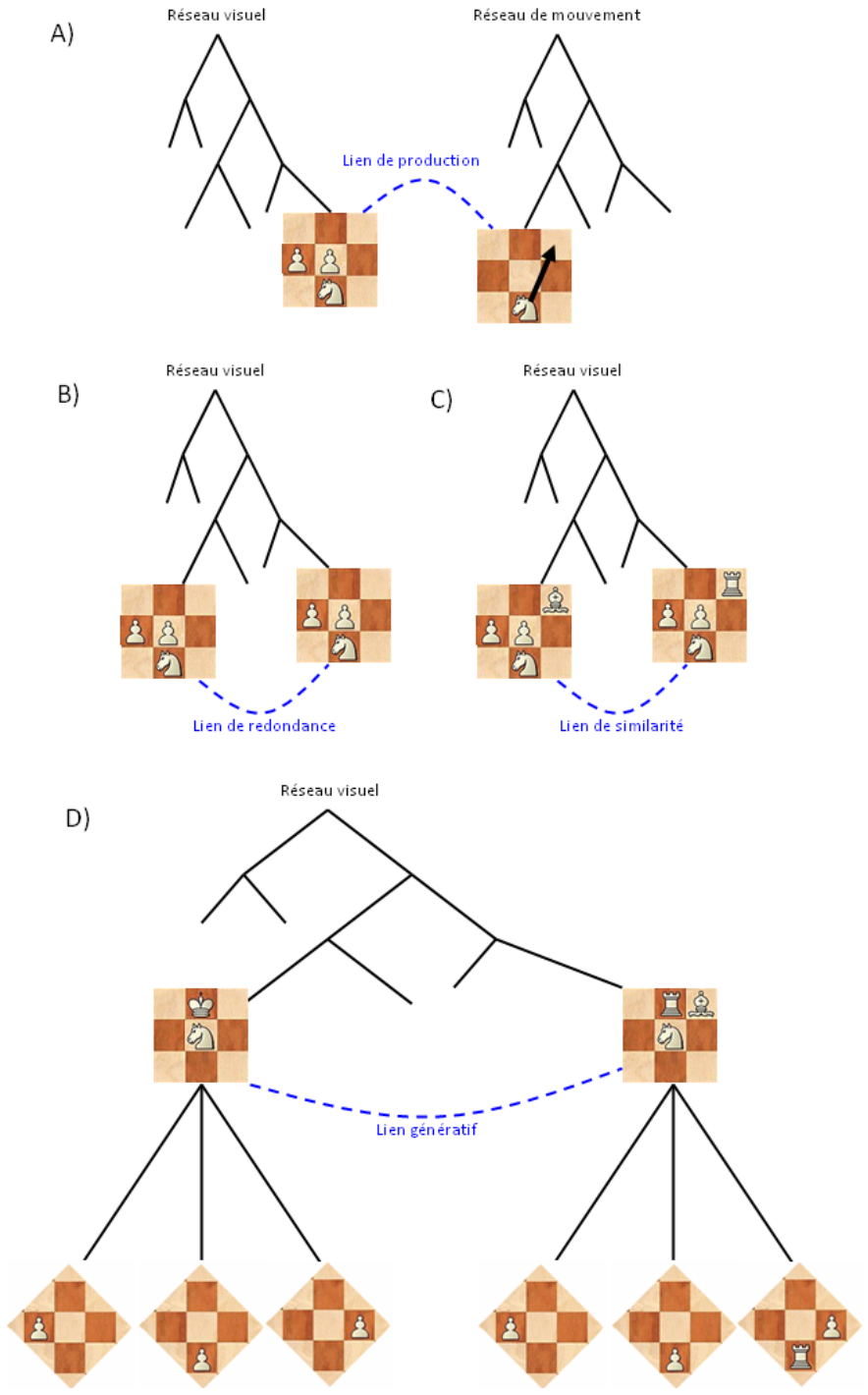


FIGURE 6.8 – Figure inspirée de Gobet et al. (2001) illustrant les différents types de liens latéraux dans le cas du jeu d'échecs. Les images des noeuds sont représentés par des rectangles et les tests par des losanges.

est fixe, tandis que dans un *template* une partie de l'information (les disjonctions) nécessite d'être instanciée. Un *template* peut ensuite être référencé en mémoire à court terme comme un chunk.

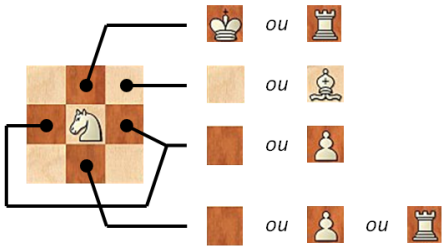


FIGURE 6.9 – Exemple de *template* correspondant au lien génératif de la figure 6.8.

Comparaison avec le MDLChunker

Le MDLChunker n'intégrant qu'une faible portion des fonctionnalités de CHREST, la comparaison entre les deux modèles est difficile. Les deux atouts principaux de CHREST par rapport au MDLChunker est qu'il intègre une modélisation de la mémoire à court terme ainsi qu'une modélisation (non présentée ici) du temps nécessaire à l'exécution de certaines tâches : création d'un chunk, encodage d'un chunk en mémoire à court terme, etc. Ces deux atouts ont permis une validation de CHREST dans des situations écologiques bien plus complexes que celle proposée pour le MDLChunker au chapitre suivant.

L'expressivité du langage de description de CHREST est supérieure à celle du MDLChunker grâce à l'utilisation des liens latéraux. Pour que le MDLChunker soit en mesure de créer l'équivalent des *templates*, il serait nécessaire d'implémenter la disjonction (section 4.4.1) avec toutes les difficultés que cela comporte.

L'utilisation par CHREST d'un réseau de discrimination comme processus de *chunking* utilise une architecture totalement différente du mécanisme de création ascendante de chunks proposée par PARSER, par CC ou par le MDLChunker. À l'inverse de ces trois chunkers qui capturent la partie régulière des données, le réseau de discrimination proprement dit capture la partie irrégulière (Fig. 6.10 page suivante). La partie régulière apparaît par contraste dans les images qui ne subissent pas le mécanisme de discrimination. Plus les stimuli comportent d'irrégularités, plus le réseau de discrimination de CHREST est complexe, donc plus les chunks sont nombreux, courts et peu fréquents. À l'inverse, les autres chunkers ont une structure d'autant plus complexe que les stimuli sont réguliers.

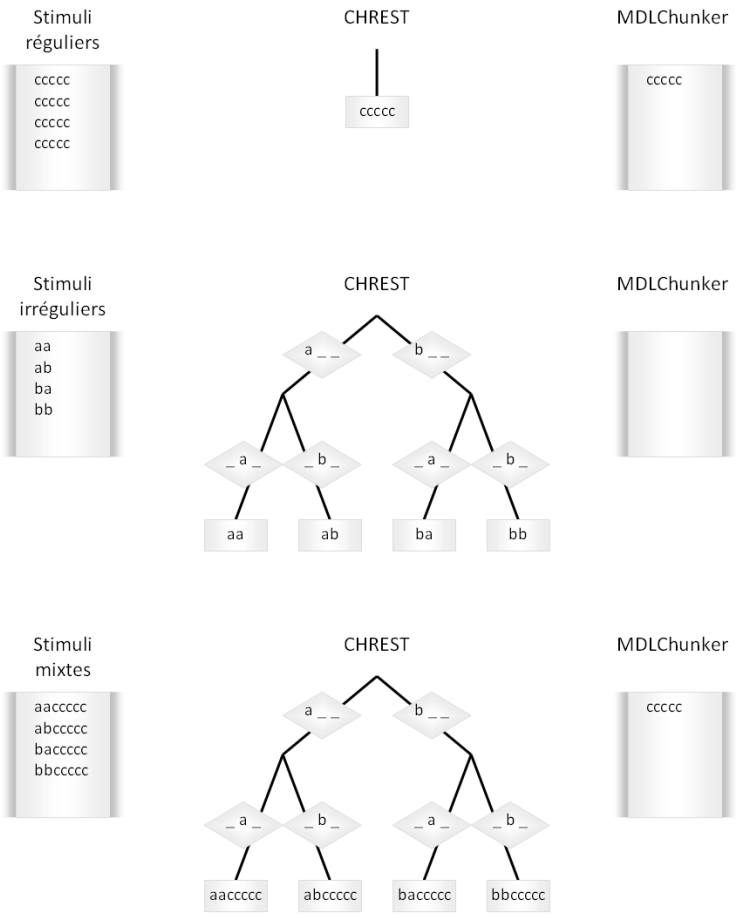


FIGURE 6.10 – Situation extrême présentant les différence de comportement entre un réseau de discrimination (type CHREST) et un modèle de *chunking* ascendant (type MDLChunker) en fonction de la régularité des stimuli. Trois cas sont présentés, le cas régulier, le cas irrégulier et le cas mixte.

6.5 Modèle de Brent et Cartwright

Description générale

Comme PARSER, le modèle de Brent et Cartwright (1996) a été conçu pour rendre compte de la segmentation d'un flux de syllabes en mots. Ce modèle utilise le *MDL two-part coding* comme critère de sélection de la meilleure segmentation possible. Bien qu'en apparence très proche du MDLChunker, ce modèle comporte en réalité de nombreuses différences, notamment dans les mécanismes d'exploration et dans la façon dont est représentée l'information.

La validation du modèle a été effectuée sur sa capacité à reproduire les mots de la langue anglaise. Le corpus utilisé (CHILDES (MacWhinney & Snow, 1985)) comprend la transcription phonétique de la production verbale de 9 mères à leur enfant de 18 mois (âge légèrement supérieur aux premiers stades d'acquisition du lexique).

Fonctionnement

Le modèle de Brent et Cartwright (1996) se divise en deux parties. Une partie que nous appellerons chunks (pour plus de cohérence avec le vocabulaire utilisé pour le MDLChunker) et qui contient la liste des mots extraits. Chaque mot est représenté par un indice suivi de la liste des phonèmes qu'il contient. La seconde partie du *two part coding* (stimuli|chunks) contient la liste des phrases du corpus, chaque mot étant représenté par son indice dans la partie chunks (Fig. 6.11). La segmentation finalement choisie est celle qui minimise la taille de codage totale.

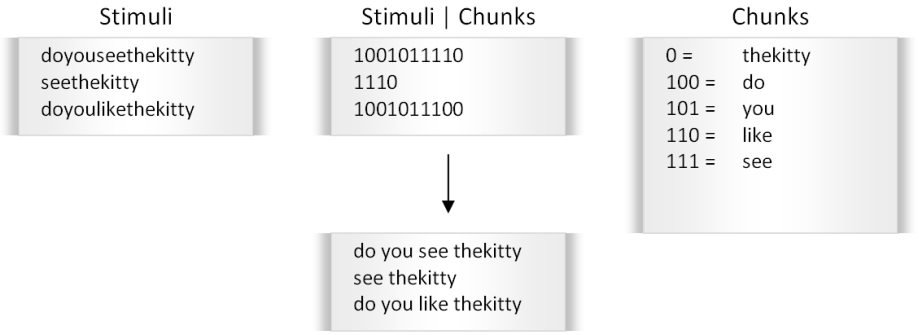


FIGURE 6.11 – Architecture du modèle de Brent et Cartwright (1996). La partie chunks présente les 5 mots extraits des 3 stimuli. L'indice du mot est donné en binaire. Pour une meilleure compréhension de la figure, les mots sont représentés à l'aide de lettres et non de phonèmes. Le codage utilisé pour les indices est autodélimitant. La partie stimuli|chunks correspond donc à la simple concaténation des indices des mots correspondants.

Comparaison avec le MDLChunker

La différence majeure entre ce modèle et le MDLChunker réside dans les mécanismes d'exploration utilisés. Le MDLChunker fonctionne de façon itérative en optimisant la taille de codage du système après perception de chaque nouveau stimulus. Le modèle de Brent et Cartwright (1996) fonctionne quant à lui en « batch » sur l'intégralité des stimuli. La création des chunks par le MDLChunker se fait de « bas en haut » , chaque nouveau chunk étant un regroupement de deux chunks plus petits. À l'inverse les chunks du modèle de Brent et Cartwright (1996) sont créés de « haut en bas » par segmentations successives des stimuli. Ces deux approches supposent des mécanismes cognitifs totalement différents : *bracketing strategy* contre *clustering strategy* (Swingley, 2005).

Le mécanisme exploratoire qui en résulte est différent. Il est exhaustif dans le cas de Brent et Cartwright (1996) : toutes les segmentations possibles sont envisagées, ce qui conduit à une solution optimale. Les auteurs sont conscients du manque de plausibilité cognitive d'une telle approche, et laissent ouverte la question d'une approche différente qui permettrait d'obtenir des résultats similaires. Pour le MDLChunker, cette recherche d'une certaine plausibilité cognitive (en termes de volume des calculs) a nécessité la conception d'un mécanisme d'exploration partielle conduisant à des solutions sous-optimales (voir chapitre 4 pour plus de détail).

Pour des raisons qui n'apparaissent pas clairement, le calcul des tailles de codage du modèle de Brent et Cartwright (1996) est différent pour la partie stimuli|chunks et pour la partie chunks. Dans le premier cas, il est similaire à celui du MDLChunker, mais dans le second cas la fréquence des différents phonèmes n'est pas prise en considération et une taille de codage identique est assignée à chacun d'eux. Les raisons de ce choix ne sont hélas pas discutées dans l'article. Assigner une taille de codage identique à chaque phonème suppose implicitement que leur distribution est uniforme. La taille de codage de chaque phonème se trouve alors en moyenne augmentée de la divergence de Kullback-Leibler (Kullback & Leibler, 1951) entre la distribution réelle des phonèmes et la distribution uniforme. La conséquence est que la taille de codage de la partie chunks est nécessairement plus élevée que pour le MDLChunker.

Il est possible que cet effet soit en partie compensé par le fait que le codage utilisé pour la partie stimuli|chunks ne soit pas un code de Shannon-Fano, comme celui utilisé par le MDLChunker. Pour le MDLChunker, les tailles trouvées sont théoriques et servent uniquement de critère de sélection, le but n'étant pas de compresser réellement les données. Dans le modèle de Brent et Cartwright (1996), les tailles de codage sont calculées grâce à un code de type Huffman (1952) qui est autodélimitant est permet une réelle [6] compression des données. La taille de codage résultante de la partie stimuli|chunks est donc nécessairement plus élevée. Bien qu'il soit possible que les deux phénomènes se compensent, il est difficile de justifier l'utilisation de deux codages différents pour les parties chunks et stimuli|chunks.

6. Cela introduit un effet de quantification des tailles de codage qui diminue la précision des résultats.

Le modèle d'Argamon, Akiva, Amir, et Kapah (2004) s'inspire de celui de Brent et Cartwright (1996). Il vise non pas à rendre compte de la segmentation de phrases en mots, mais à rendre compte de la segmentation de mots en affixes (par exemple inter-nation-al-iste). L'unité de référence n'est plus le phonème mais la lettre. La différence majeure entre les modèles réside dans la façon d'envisager les segmentations possibles. Ce modèle souffre du même problème que le précédent en ce qui concerne le calcul des tailles de codage de la partie chunks. Un paramètre supplémentaire a d'ailleurs été ajouté pour fixer arbitrairement la taille de codage des lettres, ce qui limite considérablement l'intérêt d'utiliser le MDL (la solution au dilemme bais-variance est fixée par ce paramètre). Les auteurs proposent d'ailleurs un modèle beaucoup plus simple, basé sur le nombre des affixes créés, qui obtient des résultats similaires à ceux obtenus avec le MDL.

6.6 Modèle de Goldsmith

Description générale

Le modèle de Goldsmith (2001) permet de rendre compte de la segmentation de mots en affixes de façon similaire au modèle d'Argamon et al. (2004). Le but de ce modèle est double. D'une part, il peut servir de base à la création d'un analyseur morphologique pouvant fonctionner sur de larges corpus de textes. Un tel générateur pourrait permettre, par exemple, d'étudier l'évolution de la morphologie d'une langue au cours des siècles ou au cours des divers stades d'acquisition du langage. D'autre part, il peut être utilisé pour pré-traiter les données d'un système de création automatique de grammaire, en permettant notamment de faire correspondre à chaque mot sa catégorie grammaticale. Il est en effet possible d'utiliser les suffixes (*ed*, *ing*, *s*) d'un mot (*jump*) pour avoir des indications sur sa catégorie grammaticale [7] (verbe).

Ce modèle a été appliqué à des textes en anglais, français, allemand, espagnol, italien, hollandais, latin et russe. Des résultats satisfaisants ont été produits pour des corpus allant de 5 000 à 500 000 mots. Les performances obtenues sont remarquables puisque le traitement du corpus de 500 000 a nécessité seulement cinq minutes. Ces performances sont en grande partie dues au fait que ce modèle a été entièrement conçu pour s'adapter de façon efficace à cette tâche très particulière. Les différences avec le MDLChunker étant tellement nombreuses, nous n'effectuerons pas de comparaison avec ce modèle.

Fonctionnement

Le fonctionnement du modèle de Goldsmith (2001) est relativement simple dans son principe. Il s'adapte parfaitement à la tâche de segmentation d'un mot sous la forme « radical » + « suffixe » . Le modèle est composé de trois parties :

7. Même en ignorant que les mots suivis de *ed*, *ing* et *s* sont des verbes, savoir que *jump*, *laugh* et *walk* appartiennent à la même catégorie est une information très précieuse.

une partie radical contenant les différents radicaux extraits, une partie suffixe qui de façon similaire contient les suffixes extraits et une partie signature qui associe un ensemble de suffixes à un ensemble de radicaux (Fig. 6.12).

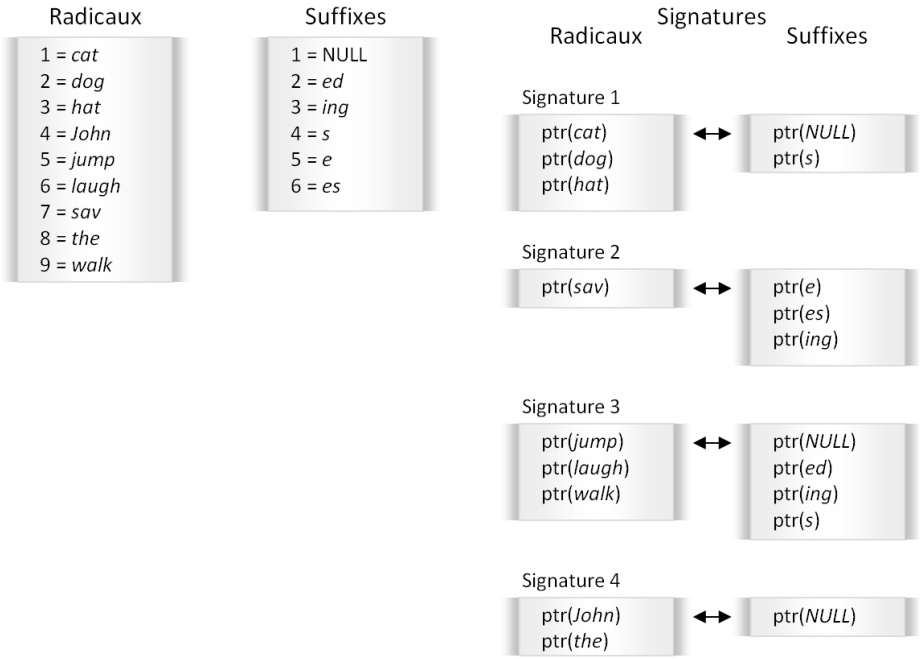


FIGURE 6.12 – Exemple tiré de Goldsmith (2001). La partie signatures est constituée de deux listes. Une liste de pointeurs vers les radicaux à laquelle est associée une liste de pointeurs vers les suffixes. Les mots couverts sont : *cat*, *dog*, *hat*, *cats*, *dogs*, *hats*, *save*, *saves*, *saving*, *jump*, *laugh*, *walk*, *jumped*, *laughed*, *walked*, *jumping*, *laughing*, *walking*, *jumps*, *laughs*, *walks*, *John* et *the*.

Chaque pointeur vers les suffixes ou les radicaux est en réalité un identifiant permettant de retrouver le radical ou le suffixe correspondant. La taille de cet identifiant dépend uniquement de la fréquence de l'élément considéré : elle est donnée par un code de Shannon-Fano. La meilleure segmentation de chaque mot est trouvée en envisageant tous les suffixes possibles comportant de une à cinq lettres. La longueur des suffixes envisagés est susceptible d'être modifiée selon le type de langue. Pour le mot *jumping* les suffixes successivement envisagés sont : *NULL*, *g*, *ng*, *ing*, *ping* et *mping*. La segmentation effectivement retenue est celle qui minimise la taille de codage totale des trois parties : radicaux, suffixes et signatures.

L'exemple présenté est un peu restrictif car le modèle permet en réalité la création de plusieurs affixes en donnant la possibilité aux pointeurs de la partie radicaux de pointer également sur des signatures. Un radical peut être soit un élément de la partie radicaux, soit être un groupement radical + suffixe, et ainsi de suite de façon récursive (Fig. 6.13 page ci-contre).

Comme évoqué au début de cette section, la différence majeure entre le MDL-Chunker et ce modèle est la forme très particulière donnée à la partie signatures,

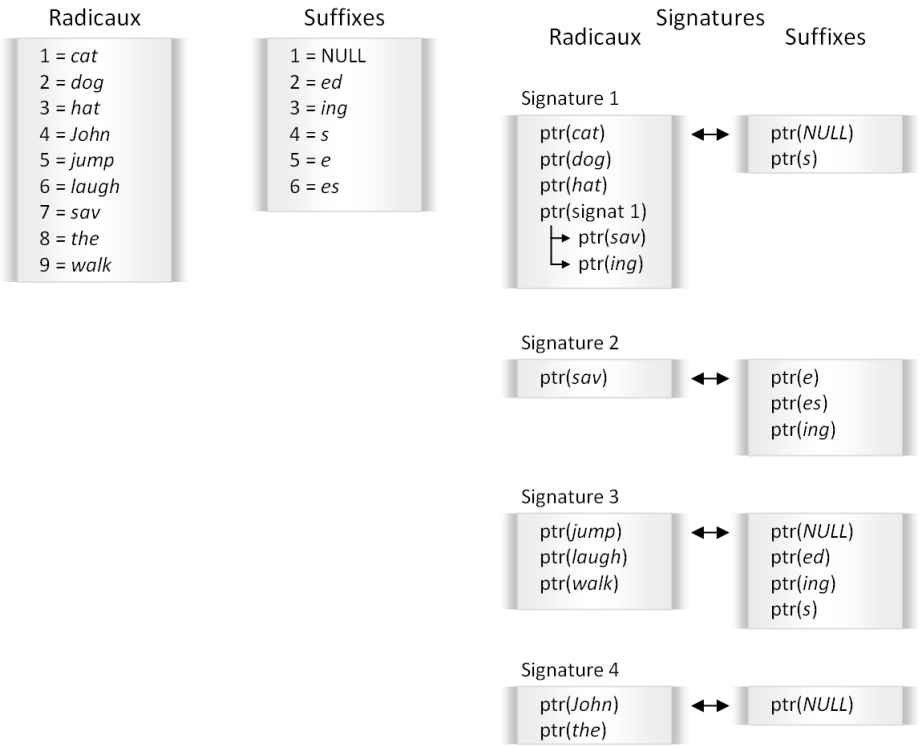


FIGURE 6.13 – Exemple tiré de Goldsmith (2001). Reprise de la figure 6.12 dans laquelle un radical complexe a été ajouté à la première signature. Ce radical contient un pointeur vers la seconde signature et prend en argument les deux pointeurs nécessaires à son instanciation. Cela permet d'ajouter le mot *savings*.

qui en fait un modèle exclusivement dédié à la séparation des mots en radical + suffixe.

6.7 Modèle *Simplicity and Power*

Description générale

Le modèle *Simplicity and Power* (SP) (Wolff, 2006) est basé sur l'idée qu'un ensemble d'observations donné doit être « simplifié » de façon à supprimer le plus de redondance possible, tout en préservant au maximum son « pouvoir » descriptif. La notion de simplicité étant évaluée par un code de Shannon-Fano, il s'agit en réalité d'une formulation différente du MDL, dans laquelle on retrouve de façon très explicite le dilemme biais-variance discuté au chapitre 3. Ce modèle se propose d'unifier différents domaines de la cognition et a donc vocation à être assez général pour pouvoir être appliqué à des tâches très différentes. Il a notamment été appliqué à la segmentation de mots (Wolff, 2000), au raisonnement probabiliste (Wolff, 1999) et au diagnostic médical (Wolff, 2006).

Fonctionnement

Le modèle SP apparaît également sous le nom ICMAUS (*Information Compression by Multiple Alignment, Unification and Search*) (Wolff, 2003), qui a le mérite de faire apparaître plus explicitement les mécanismes utilisés. La notion de simplicité qui est au coeur du modèle est évaluée en terme de « compression d'information » , basée sur l'utilisation d'un code de Shannon-Fano. Le terme d' « alignement multiple » fait référence au processus permettant d'exprimer un nouveau stimulus à partir des stimuli déjà perçus : chaque nouveau stimulus est aligné avec les parties les plus fréquentes des anciens stimuli. Il s'agit du mécanisme similaire à celui de *perception shaping* (Perruchet, Vinter, Pacteau, & Gallego, 2002) évoqué précédemment, qui consiste à utiliser les perceptions passées pour guider la perception courante. Le terme « recherche » fait référence au processus responsable de trouver le meilleur alignement pour le stimulus. Ce processus est loin d'être trivial. Finalement, la notion d' « unification » renvoie de façon détournée à la notion de chunk. En résumé, le modèle SP est un modèle de *chunking* utilisant le MDL comme critère d'alignement du stimulus courant avec les stimuli passés.

La notion d'alignement multiple utilisée dans SP est similaire à celle utilisée en bio-informatique pour aligner les séquences d'ADN. Lorsque le stimulus courant est perçu, il est aligné avec les stimuli déjà perçus. Chacun des stimuli possède une taille de codage qui dépend uniquement de sa fréquence. Un mécanisme complexe permet de tester plusieurs alignements, et de retenir celui dont la taille de codage est la plus faible. L'alignement n'est pas nécessairement total. Deux paramètres permettent de choisir si tous les symboles du stimulus courant doivent être appariés et si tous les symboles des stimuli utilisés pour l'alignement doivent l'être également (Fig. 6.14 page suivante).

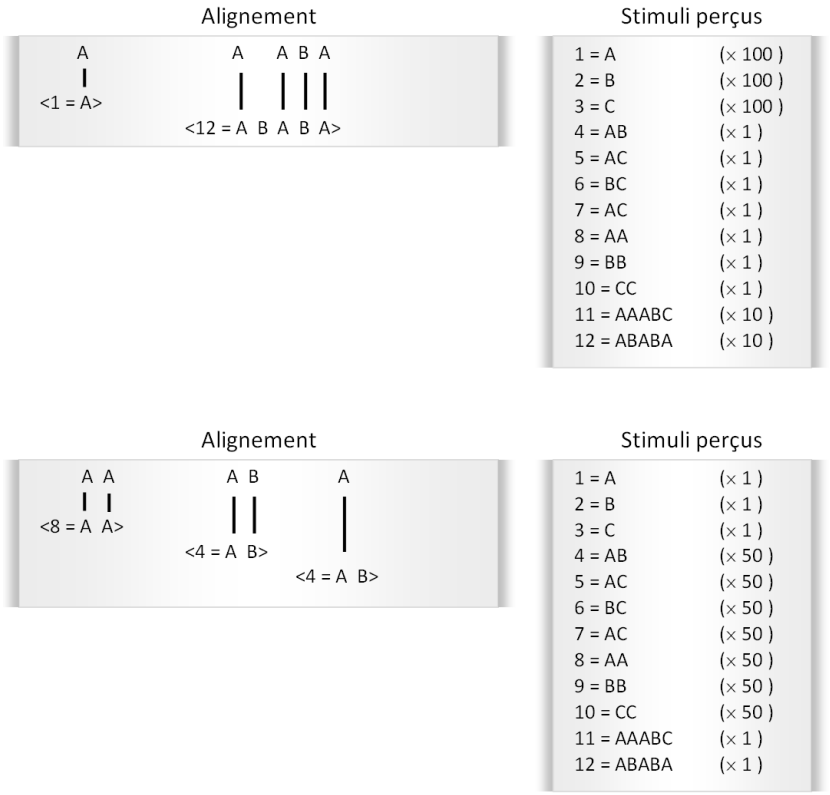


FIGURE 6.14 – Exemple tiré de Pothos et Wolff (2006) montrant deux alignements différents d'un même stimulus AAABA. Chaque alignement utilise prioritairement les stimuli perçus de plus forte fréquence (indiquée entre parenthèses), donc de taille de codage faible. Le premier appariement utilise les stimuli 1 et 12 et le second les stimuli 1, 4 et 4. Dans les deux cas, l'alignement n'est pas parfait car un symbole n'est pas apparié.

À l'issue de l'alignement, chaque portion de séquence qui a pu être appariée est ajoutée aux stimuli perçus. Il en va de même des portions non-appariées. Un exemple de ce mécanisme est présenté figure 6.15.

FIGURE 6.15 – Exemple inspiré de Wolff (2003). Le stimulus courant « itsplaytimenow » est apparié avec le stimulus « itsbedtimenow » déjà perçu auparavant. Cela conduit en retour à la création des trois stimuli représentés en gras.

Même s'il n'est pas présenté sous cette forme, le mécanisme d'association des symboles adjacents peut être vu comme une forme de *chunking*. Une présentation plus détaillée du modèle peut être trouvée dans Wolff (2006).

Comparaison avec le MDLChunker

De tous les modèles évoqués dans ce chapitre, SP est incontestablement celui qui est le plus proche du MDLChunker. Non seulement les deux processus de base que sont le MDL et le *chunking* sont similaires, mais surtout l'un comme l'autre ont une justification théorique solide et ont comme but de pouvoir être appliqués à des domaines variés. Certaines différences importantes existent néanmoins. Elle proviennent essentiellement de ce que les objectifs ne sont pas tout à fait les mêmes. La notion la plus importante pour SP est la familiarité du stimulus, qui représente une mesure de proximité entre le stimulus et les stimuli déjà possédés par le système. À l'inverse, ce qui est le plus central pour le MDLChunker est la notion de chunk, la familiarité du stimulus n'apparaissant que comme sous-produit des chunks créés.

Pour SP, les stimuli doivent pouvoir être stockés sous forme d'un tableau ordonné de symboles à une ou plusieurs dimensions. À ce niveau, la différence majeure entre SP et le MDLChunker est la notion d'ordre. Pour le MDLChunker, un stimulus peut être soit ordonné soit non-ordonné, ce qui est notamment utile pour une représentation de type attribut-valeur. Dans le cas où il est ordonné, la notion d'ordre choisie porte actuellement sur une dimension seulement, chaque symbole ne pouvant avoir que deux symboles adjacents. Cependant, rien dans l'architecture du MDLChunker ne s'oppose à ce que l'ordre entre symboles porte sur plusieurs dimensions. L'absence d'ordre correspond d'ailleurs à la situation où chaque symbole est adjacent à tous les autres. À l'inverse, supprimer la notion d'ordre dans SP n'est pas possible, car la notion même d'alignement perdrait tout son sens.

L'alignement est le processus central de SP. Il est identique dans sa fonction à l'étape de factorisation du MDLChunker : les deux processus ont pour but d'exprimer le stimulus courant en fonction des stimuli passés. L'alignement réalisé

par SP est intimement lié à la notion d'ordre entre les symboles et pour cette raison, il est plus riche que la factorisation du MDLChunker. Dans la version standard du MDLChunker, la factorisation se fait sans tenir compte de l'ordre. Cette version a par la suite été améliorée pour les besoins du chapitre 8, et un ordre a été introduit entre les symboles. Mais dans la version avec ordre, l'utilisation de la négation a été supprimée. Pour que le MDLChunker soit capable d'effectuer une factorisation similaire à l'alignement de SP, il faudrait introduire à la fois l'ordre et la négation. En reprenant l'exemple de la figure 6.15 page précédente, la factorisation de « itsplaytimenow » donnerait :

« itsbedtimenow » (pos 1) - « bed » (pos 4) + « play » (pos 4)

où les indications entre parenthèses représentent la position à laquelle doit s'effectuer l'ajout ou la suppression.

La différence majeure entre le MDLChunker et SP concerne la représentation de l'information. Dans le MDLChunker, l'information est « compilée » , tandis que dans SP elle est ré-extraite à chaque nouvel alignement. Les différents stimuli perçus par SP sont stockés en l'état après chaque alignement. Lors de l'alignement suivant, tous les stimuli sont alors candidats. À l'inverse, le MDLChunker compile cette information sous forme de chunks. Seuls les chunks qui contribuent à diminuer la taille de codage du système sont créés. L'information qui n'était pas assez pertinente pour donner lieu à la création d'un chunk est alors ignorée lors de la factorisation suivante (bien qu'elle soit conservée dans la partie stimuli|chunks pour être extraite plus tard si nécessaire). Le pendant positif de cette « non-compilation de l'information » est que SP est en mesure de stocker des relations plus complexes que des chunks qui sont de simple conjonctions. C'est notamment le cas des disjonctions (Fig. 6.15 page ci-contre) dont l'impact sur la taille de codage du système est plus difficile à prévoir (discuté section 4.4.1).

6.8 Conclusion

Les modèles computationnels présentés dans ce chapitre ont tous été choisis pour les similitudes qu'ils entretiennent avec le MDLChunker. Cela a permis de constater que, bien qu'il existe de nombreuses similitudes entre le MDLChunker et les six modèles présentés, l'architecture du MDLChunker reste assez différente de celle des autres modèles sur de nombreux points qui ne sont pas tous des détails. La raison de ces différences est en grande partie due au fait que les buts recherchés ne sont souvent pas les mêmes. C'est d'ailleurs ce qui rend la comparaison ardue. Une comparaison détaillée des mécanismes employés n'a pu être donnée que pour PARSER et pour le *Competitive Chunker* et a été placée en annexe pour ne pas ennuyer le lecteur avec des détails techniques. La comparaison avec les quatre autres modèles a surtout porté sur les principes généraux de leur fonctionnement.

Nous avons pu voir (s'il était encore besoin de le montrer) que le mécanisme de *chunking* utilisé en psychologie cognitive est à la base de plusieurs modèles

cognitifs computationnels. À travers les quelques implémentations décrites dans ce chapitre, il est aisé que constater que le *chunking* n'est pas uniquement un principe théorique, mais également un principe permettant de réaliser des prédictions quantitatives précises, vérifiables expérimentalement. De même nous avons vu que plusieurs modèles computationnels utilisent le MDL comme critère de sélection afin de fixer les degrés de liberté laissés libres par le modèle de façon à minimiser la taille de codage de la représentation des données.

Les deux principes de base du MDLChunker (MDL et *chunking*) ont tous deux prouvé leur utilité dans le cadre de la modélisation cognitive. Ce chapitre a montré que l'architecture du MDLChunker possédait de nombreux points communs avec d'autres modèles existants, mais également des différences qui font son originalité. Cette comparaison a également permis de montrer certaines des limites du MDLChunker qui sont en grande partie liées à la simplicité de son architecture et à l'absence de paramètres. La partie suivante concerne la validation du MLDLChunker et se divise en trois chapitres. Le premier est destiné à sa validation proprement dite, au travers d'une nouvelle expérience conçue dans le but de valider le décours temporel de la création des chunks par le MDLChunker. Les deux suivants ont pour but de montrer la généralité du MDLChunker en l'appliquant à deux domaines différents : la segmentation de mots et la modélisation de la mémoire à court terme. De la même façon que CHREST étend le modèle EPAM, vous verrons dans ces deux derniers chapitres comment le MDLChunker basique supporte l'ajout de modules supplémentaires et notamment une limitation de la taille de sa mémoire.

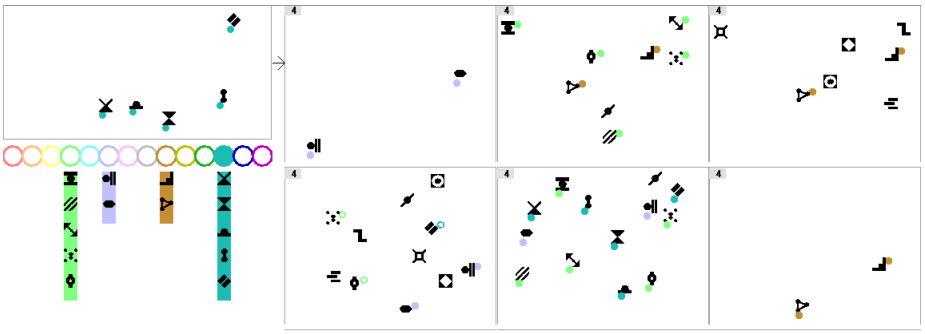
Troisième partie

Validations

Chapitre 7

Validation expérimentale

Sommaire



7.1 Introduction

Nous avons établi dans la première partie de cette thèse les fondements théoriques des principes utilisés par le MDLChunker, montrant ainsi qu'il ne s'agit pas simplement d'un modèle *ad hoc* conçu pour ajuster un type particulier de données. L'application du principe de simplicité et du mécanisme de *chunking* à la modélisation cognitive a notamment été discuté. Selon Chater (1999) cette « justification normative » (théorique) est la première étape indispensable à la validation d'un modèle cognitif ; la seconde étant naturellement l'adéquation des prédictions du modèle aux données expérimentales : le modèle doit être « descriptivement correct » . Cette seconde étape est l'objet du présent chapitre qui a pour but la validation du MDLChunker. Les chapitres suivants sont destinés à montrer la généralité de l'approche en appliquant le MDLChunker à d'autres domaines (segmentation de mots, modélisation de la mémoire à court terme).

Ce chapitre présente une nouvelle expérience conçue pour valider le modèle en comparant les représentations créées par le MDLChunker et celles créées par les participants lorsqu'ils sont entraînés sur le même ensemble de stimuli. Le MDLChunker ne comportant aucun paramètre ajustable, son exécution est totalement indépendante des résultats obtenus par les participants. Les participants et le MDLChunker sont soumis exactement au même jeu de données sans ajout d'information supplémentaire. Le MDLChunker peut être considéré comme un participant informatique que l'on souhaite comparer aux participants humains.

La présente introduction vise à justifier la création d'une nouvelle expérience pour valider le MDLChunker. La suite du chapitre est consacrée à la description de l'expérience proprement dite puis à l'interprétation des résultats obtenus, par le MDLChunker puis par le MDLChunker-approché.

7.1.1 Différences entre participants et MDLChunker

Il est nécessaire que les participants et le MDLChunker soient comparés sur la même base. Or le MDLChunker est un modèle très simple qui n'est sensible qu'aux régularités fréquentielles et aux chunks. Il ne contient aucun a priori sur le matériel utilisé. Pour qu'une comparaison soit possible, il est nécessaire que les a priori ainsi que les mécanismes mis en jeu soient identiques pour les participants et le MDLChunker. La première difficulté vient du fait que les participants humains ne sont pas nécessairement sans a priori quant au matériel utilisé (par exemple les symboles suivants composés de deux traits : « + » , « × » et « = » sont naturellement perçus comme étant des symboles mathématiques). Pour ce qui est des mécanismes d'apprentissage, les participants humains sont capables de percevoir des régularités beaucoup plus complexes que de simples chunks (1, 2, 4, 8, 16, 32, 64 en est un exemple). Comme les participants sont testés uniquement sur leur capacité à créer des chunks, cela signifie que l'expérience ne doit pas inciter la création de représentations plus complexes.

Différences au niveau des a priori

Si l'on considère l'hypothèse raisonnable selon laquelle notre perception présente dépend en partie des représentations passées, alors il est nécessaire de contrôler ces représentations dans le cadre d'une expérience rigoureuse. Dans le cas où ces représentations sont des chunks, ce phénomène a été décrit sous le nom de *perception shaping* (Perruchet et al., 2002).

La première solution pourrait consister à approcher les chunks possédés par les participants, puis à les incorporer au MDLChunker comme connaissance a priori. Si incorporer des chunks a priori au MDLChunker ne pose aucune difficulté théorique, cette solution est totalement inapplicable en pratique. Il n'est pas possible d'approcher les chunks possédés par les participants avec assez de précision pour que l'erreur commise (*i.e.* le bruit introduit) soit négligeable devant les effets que l'on cherche à mesurer. De plus, les connaissances a priori n'étant pas nécessairement identiques d'un participant à l'autre, une évaluation des chunks de chacun des participants serait nécessaire.

Une autre solution consiste à faire en sorte que ni le MDLChunker, ni les participants n'aient d'a priori sur les chunks mis en jeu dans l'expérience. Cette solution peut se décliner de deux façons.

La première (French et al., 2004) consiste à utiliser comme participants de jeunes enfants dont on peut facilement être sûr que leurs connaissances a priori du domaine sont négligeables, et que les différences inter-individuelles sont faibles. Cette solution possède l'avantage de ne pas tester les participants sur une tâche totalement artificielle pour laquelle se pose le problème de l'adéquation des comportements observés en laboratoire avec les situations de la vie réelle.

La seconde déclinaison consiste justement à utiliser une tâche artificielle pour laquelle les participants n'ont aucune connaissance a priori. C'est notamment le cas de l'apprentissage de grammaires artificielles (*Artificial Grammar Learning* ou AGL). C'est cette solution qui a été retenue pour l'expérience décrite dans ce chapitre.

Différences au niveau des mécanismes d'apprentissage

Comme cela a été discuté dans le chapitre 1, la modélisation choisie dans cette thèse s'oppose à celle défendue notamment par Ohlsson (2008) qui suppose une très forte dépendance des différentes modalités cognitives et vise à modéliser l'intégralité de la chaîne de traitement, des processus perceptifs jusqu'aux fonctions cognitives de haut niveau. Une telle approche a l'inconvénient de produire des modèles très complexes, mais possède l'indiscutable avantage de pouvoir être appliquée à un panel d'expériences plus large, dans la limite où les fonctions cognitives impliquées dans l'expérience sont modélisées. L'approche que nous avons choisie est plus proche de celle adoptée par les sciences expérimentales en général. Elle consiste à se concentrer sur la modélisation d'une seule modalité cognitive (le mécanisme de *chunking*) en cherchant à limiter l'influence des

autres. En contrepartie, un soin tout particulier doit être apporté à la conception des expériences afin de limiter les inévitables effets dûs aux modalités cognitives qui ne sont pas modélisées (effets d'oubli, effets d'ordre ou de proximité spatiale, etc.).

Il est donc nécessaire de placer les participants humains et le MDLChunker dans des situations identiques, et faire en sorte que les régularités perçues puissent être exprimées dans le même formalisme. Le type de régularités perçues par le MDLChunker ainsi que le formalisme de représentation utilisé est parfaitement connu. La difficulté est de définir pour les participants un protocole expérimental permettant d'une part d'imposer un format particulier aux représentations [1] et d'autre part d'éviter que des effets non modélisés n'introduisent un bruit trop important dans les résultats.

7.1.2 Vue d'ensemble de l'expérience

L'expérience choisie est de type *Artificial Grammar Learning* (AGL) et consiste à présenter une succession de phrases issues d'une grammaire inconnue. Les symboles utilisés ne correspondent à aucun symbole connu. Cela permet de limiter les effets d'interférence liés aux représentations a priori qui sont associées aux symboles : effets que le MDLChunker ne peut prédire dans la mesure où tous les symboles sont considérés comme équivalents.

Le MDLChunker ne capturant que les régularités de type « chunks » et les régularités fréquentielles, la grammaire a été conçue de façon à éviter la présence d'autres régularités qui pourraient être perçues par les participants humains sans pouvoir l'être par le MDLChunker. C'est le cas notamment de la proximité spatiale entre symboles qui est contrôlée pour éviter l'association privilégiée de symboles spatialement proches. Afin de limiter l'influence de la mémoire (non modélisée), l'interface comporte également un historique des différentes phrases présentées ainsi qu'un historique des chunks rapportés par le participant. Il est explicitement demandé aux participants de rapporter les régularités observées, et ceci pour chaque nouvelle phrase présentée. Le but est de pouvoir connaître l'évolution temporelle des régularités perçues.

7.1.3 Les raisons d'une nouvelle expérience

La décision de concevoir une nouvelle expérience se justifie par une forte exigence de termes de validation, ce que ne permet pas le paradigme classique utilisé en AGL. À titre de comparaison, reprenons l'expérience utilisée pour valider le *Competitive Chunker* décrit au chapitre 6.

1. Il ne s'agit bien sûr pas de contraindre les représentations elles-mêmes, mais la façon dont elles sont rapportées par les participants (se reporter à la description de l'expérience pour plus de détail).

The Reber task

Le protocole expérimental utilisé par Servan-Schreiber et Anderson (1990) est plus connu sous le nom de *Reber task* (Reber, 1967). Il s'agit d'une tâche de mémorisation, suivie d'une tâche de discrimination. Dans un premier temps, on demande aux participants de mémoriser des chaînes de caractères issues d'une grammaire qui leur est inconnue (Fig. 7.1). Une fois l'apprentissage terminé, on leur révèle l'existence d'une grammaire sous-jacente, puis différentes chaînes de caractères leurs sont présentées. Les participants doivent alors discriminer les chaînes grammaticales des chaînes non-grammaticales sur la base de leur expérience passée. Le *Competitive Chunker* est validé sur cette tâche de discrimination d'après sa capacité à prédire le même pourcentage de chaînes grammaticales que celui obtenu par les participants humains.

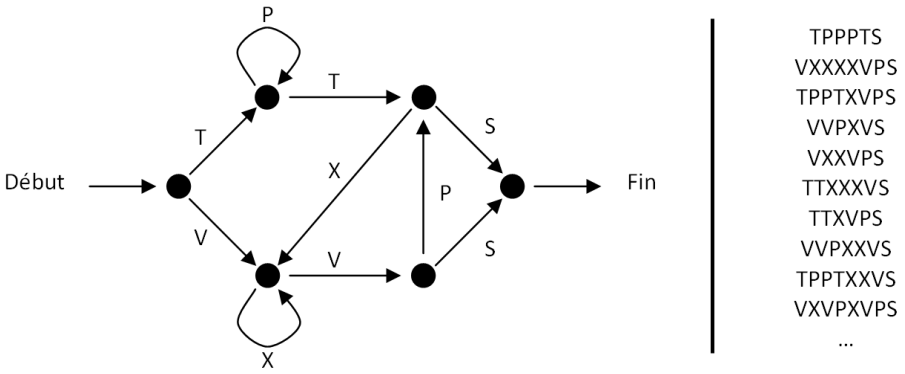


FIGURE 7.1 – Grammaire utilisée pour la *Reber task*. Le choix des symboles T, V, X, P et S est arbitraire et peut varier d'une expérience à l'autre. Quelques exemples de chaînes de caractères sont donnés à droite.

Apprentissage implicite et influence de la nature consciente ou inconsciente des représentations sur la validation du modèle

La principale difficulté qui est discutée maintenant est l'accès aux chunks créés par les participants humains : le fait qu'un individu soit capable d'abstraire des régularités ne signifie pas qu'il soit capable de les formuler explicitement (Cleeremans & Jiménez, 2002). En effet, le protocole expérimental mis en jeu dans la *Reber task* a comme but l'étude de l'apprentissage implicite. Bien que la nature consciente ou inconsciente des représentations qui découlent d'un apprentissage implicite ne fasse pas consensus (Cleeremans, Destrebecqz, & Boyer, 1998 ; Perruchet, Vinter, & Pacton, 2007), la tâche de discrimination utilisée ne nécessite pas de la part des participants d'expliciter les raisons de leur choix. La validation se fait alors uniquement sur les effets observables de cet apprentissage implicite tels qu'ils sont capturés par la tâche de discrimination. Les résultats obtenus à l'issue de la tâche de discrimination consistent en une simple suite binaire des réponses concernant la grammaticalité des exemples présentés. Cette information relativement pauvre est encore fortement dégradée en la réduisant

à une simple moyenne. Or un très grand nombre de modèles différents sont compatibles avec la faible quantité d'information véhiculée par un nombre réel (voir chapitre 2 pour une discussion approfondie sur ce point). La validation d'un modèle sur sa capacité à reproduire une moyenne correspond à un faible niveau d'exigence.

En utilisant ces mêmes résultats, une validation plus fine consisterait à prédire la grammaticalité de chacun des exemples présentés durant la tâche de discrimination. Cela permettrait de valider le modèle sur sa capacité à reproduire l'intégralité de la séquence binaire des réponses plutôt que la moyenne.

Cependant, dans le cadre de l'apprentissage de grammaires artificielles (AGL), une étude de Dulany, Carlson, et Dewey (1984) a montré que les participants étaient en mesure d'expliciter durant la tâche de discrimination les raisons de leur choix, en soulignant les lettres (chunks) qui à leur sens rendaient la phrase grammaticale. En termes d'exigences de validation, estimer la performance d'un modèle sur sa capacité à reproduire les chunks explicités par les participants est de loin bien supérieure à l'approche précédente.

Une validation encore plus fine consisterait à déterminer quels chunks sont possédés par les participants après avoir appris la première phrase de la tâche de mémorisation, puis après la seconde, la troisième et ainsi de suite pour toutes les phrases. Cela permettrait d'évaluer, non pas la performance asymptotique du modèle, mais sa capacité à prédire correctement les chunks créés à chaque itération (*i.e.* à chaque nouvelle phrase). En utilisant le protocole expérimental de Dulany et al. (1984) cela nécessiterait de répliquer l'expérience autant de fois qu'il y a de phrases utilisées dans la tâche de mémorisation. Au problème pratique qui se pose pour trouver une telle quantité de participants différents [2], s'ajoute également celui de l'apprentissage durant la tâche de discrimination. En effet, Redington et Chater (1996) ont montré que le pourcentage de discriminations correctes (significativement au-dessus du seuil de chance) qui est classiquement attribué à un apprentissage implicite, peut en partie s'expliquer par un apprentissage de la grammaire durant la phase de discrimination [3]. Dans le cas d'une répétition d'expériences de type Dulany et al. (1984), ce phénomène se trouve encore accentué par le fait que les premières itérations comportent nécessairement un faible nombre de phrases dans la tâche de mémorisation en comparaison du nombre de tests effectués durant la tâche de discrimination.

La solution alternative qui a été choisie ici consiste à concevoir une nouvelle expérience dans laquelle les participants sont informés dès le début de l'expérience de l'existence d'une grammaire sous-jacente sans que soient données d'indications sur sa structure. Cela permet de rendre explicite l'apprentissage ainsi que la création de chunks. Disparaissent ainsi les problèmes liés aux phénomènes d'apprentissage durant la tâche de discrimination. Cela introduit néanmoins certaines difficultés.

2. Un même participant ne peut participer à plusieurs expériences, car indépendamment de la connaissance qu'il a acquise de la grammaire, il a conscience de l'existence d'une grammaire sous-jacente, comme cela lui a été révélé avant la tâche de discrimination.

3. Cet effet peut être montré grâce à l'ajout d'un groupe contrôle (Perruchet & Pacteau, 1990).

Difficultés liées à une tâche d'apprentissage explicite

Il a été montré que l'explicitation des représentations et notamment la verbalisation (Stanley, Mathews, Buss, & Kotler-Cope, 1989) pouvait conduire à une amélioration substantielle des performances (*the synergy effect* (Sun, Merrill, & Peterson, 2001). Ce résultat est à nuancer car Berry et Broadbent (1988) ont également montré que dans une tâche de recherche explicite, les performances pouvaient être soit améliorées soit dégradées selon la saillance des régularités recherchées. Cet effet n'est pas limitant dans la mesure où le MDLChunker n'a pas vocation à analyser les liens entre l'apprentissage implicite et la nature consciente ou inconsciente des représentations qui en découlent. Néanmoins, les résultats obtenus sont intimement liés au caractère explicite des représentations testées.

Bien évidemment, en demandant aux participants d'expliciter les chunks créés, on se limite aux chunks conscients. Il est alors possible que certains chunks existent de façon inconsciente, ou simplement que les participants omettent de les rapporter. Le MDLChunker, qui ne fait pas de distinction entre chunks conscients ou inconscients, risque simplement de surestimer le nombre des chunks rapportés par les participants. L'existence de tels chunks créés par les participants et le MDLChunker mais non rapportés par les participants, seront considérés comme des prédictions incorrectes de la part du MDLChunker. Cela va dans le sens du rejet des prédictions du MDLChunker ; il n'y a donc pas de risque de valider à tort les prédictions réalisées.

Une nouvelle expérience

Le protocole expérimental choisi a pour but de valider le modèle sur la base des représentations conscientes plutôt que sur la base des conséquences observables de représentations potentiellement inconscientes. Cela permet d'être beaucoup plus strict en terme de validation et nous a semblé justifier la création d'une nouvelle expérience. On peut résumer les différents types de validation possibles dans le cas de l'apprentissage de grammaires artificielles (AGL) de la façon suivante (classés par exigences de validation croissantes) :

1. Validation du modèle sur sa capacité à prédire le pourcentage de réponses correctes obtenu par l'ensemble des participants sur la tâche de discrimination.
2. Validation du modèle sur sa capacité à prédire la séquence (chaîne binaire) des réponses obtenues par les participants sur la tâche de discrimination.
3. Validation du modèle sur sa capacité à prédire les chunks créés par les participants à l'issue de la tâche de mémorisation.
4. Validation du modèle sur sa capacité à prédire les chunks créés par les participants pour chaque itération (phrase) de la tâche de mémorisation, c'est-à-dire sur sa capacité à prédire le décours temporel de la création des chunks.

La quantité d'information recueillie dans la quatrième situation est sans commune mesure avec celle recueillie dans la première. Le nombre de modèles com-

patibles avec les résultats de la quatrième situation est donc bien plus faible que pour la première. L'expérience décrite maintenant entre dans cette quatrième situation.

7.2 Matériel

L'expérience a été conçue de façon à pouvoir connaître les chunks créés par les participants, ainsi que leur évolution au cours du temps. L'interface proposée (Fig. 7.2 page ci-contre) vise à limiter au maximum l'influence de facteurs annexes telles que la mémoire ou les prérequis conceptuels, susceptibles d'accroître les différences inter-individuelles.

Symboles

Pour ce faire, un alphabet composé de 20 symboles a été créé. Chaque symbole a été sélectionné pour son absence de signification évidente, afin de limiter l'association de symboles sur la base d'associations entre leurs signifiants. Certains des symboles employés proviennent de Fiser et Aslin (2001).

Ordre

Des « phrases » sont composées à partir de symboles de cet alphabet. Dans sa version basique, le MDLChunker considère chaque stimulus comme un ensemble non-ordonné d'éléments. Les phrases sont donc générées à partir d'une grammaire n'imposant pas de contrainte d'ordre entre les symboles.

Les phrases créées apparaissent dans une zone rectangulaire située en haut à gauche de l'interface (Fig. 7.2 page suivante). Pour éviter de privilégier l'association entre symboles adjacents, les symboles constituant la phrase ne sont pas présentés sous forme d'une liste (A C B E D), mais sont positionnés aléatoirement dans la zone d'apparition selon une loi uniforme. Une vérification est faite pour éviter le chevauchement de symboles. Afin de moyenner les effets dus à l'association privilégiée entre symboles spatialement proches, les symboles sont repositionnés aléatoirement pour chacun des participants.

Explicitation des chunks créés

Afin de s'assurer que le participant accorde un minimum d'attention à tous les symboles de chaque chunk, la consigne est de déplacer chaque symbole par glisser/déposer, de la zone d'apparition des phrases (en haut à gauche) vers la zone située à sa droite (Fig. 7.2 page ci-contre). Cette tâche répétitive et relativement fastidieuse a également pour but d'inciter le participant à expliciter les chunks qu'il a créé. Chaque chunk ainsi créé peut ensuite être déplacé dans

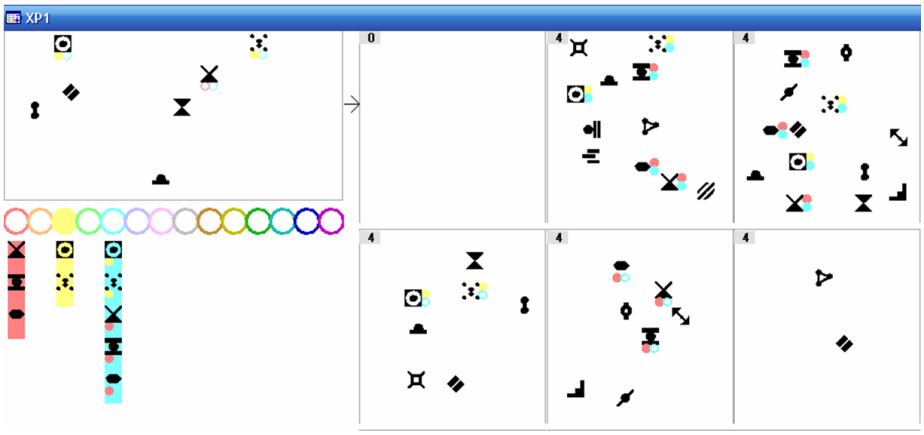


FIGURE 7.2 – Description de l'interface utilisée. La phrase courante est composée de 7 symboles (en haut à gauche). 3 chunks ont été créés (en bas à gauche) par le participant : un rouge composé de 3 symboles, un jaune composé de 2 symboles et un turquoise composé des deux chunks précédents (5 symboles). Le symbole situé le plus en haut à gauche se trouve impliqué dans le chunk jaune et dans le chunk turquoise, comme l'indiquent les deux cercles colorés qui lui sont accolés. Le cercle jaune est plein, indiquant que le chunk jaune est complet (ses 2 symboles apparaissent dans la zone). Le cercle turquoise est évidé car le chunk turquoise est incomplet (2 symboles sur 5 sont manquants). La zone de droite représente l'historique des phrases vues. Seulement 6 sont représentés, mais l'interface complète en possède 25 (interface complète présentée annexe B.3). La première case de l'historique (celle qui est vide) est celle vers laquelle les symboles doivent être déplacés.

son intégralité, ce qui facilite la fastidieuse tâche de déplacement. Les 14 cercles situés sous la zone d'apparition des phrases (en bas à gauche), permettent au participant d'expliciter jusqu'à 14 chunks.

La création d'un chunk se fait par glisser/déposer des différents symboles vers l'un des 14 cercles de la zone des chunks (en bas à gauche). Un symbole ajouté par erreur à un chunk peut être retiré par un simple clic-droit. À chaque chunk est associée une couleur permettant de le repérer. Chaque symbole de la zone d'apparition des phrases qui est impliqué dans un chunk, se voit accoler un cercle de la couleur du chunk. Si le chunk est complet (tous les symboles qui le constituent sont présents), alors le cercle de couleur correspondant est plein. Dans le cas contraire, il est évidé. Un symbole impliqué dans plusieurs chunks se voit accoler autant de cercles colorés que de chunks le contenant (Fig. 7.2 page précédente).

Oubli

Afin de diminuer les différences inter-individuelles liées aux inévitables effets d'oubli, et parce que la version basique du MDLChunker n'intègre pas de modélisation de la mémoire, un historique des phrases passées a été ajouté à l'interface (Fig. 7.2 page précédente partie droite). L'historique contient les 25 dernières phrases vues. Le marquage coloré des symboles, utilisé pour la zone d'apparition des phrases, est repris dans l'historique.

Grammaire

75 phrases sont présentées successivement au participant, sans aucune limite de temps : chaque phrase remplaçant la précédente dès qu'elle a été déplacée. Les 75 phrases ont été générées à partir de la grammaire présentée figure 7.3, et sont identiques pour tous les participants ainsi que le modèle. Afin de moyenner les effets dus à d'éventuelles associations basées sur les caractéristiques morphologiques des symboles, l'appariement de chaque symbole de la grammaire avec sont apparence visuelle est tiré aléatoirement. Cette permutation aléatoire de l'apparence visuelle des 20 symboles intervient au début de chaque expérience.

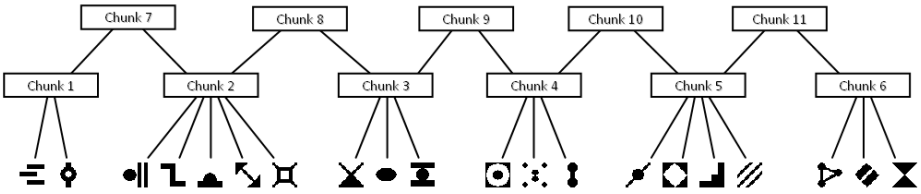


FIGURE 7.3 – Grammaire utilisée pour générer les 75 phrases de l'expérience. Les 20 symboles de la grammaire sont répartis en 11 groupes (chunks) organisés selon une hiérarchie à deux niveaux.

Chaque phrase est générée en choisissant aléatoirement un ou plusieurs chunks

dans la hiérarchie. La probabilité qu'un chunk soit choisi pour apparaître dans la phrase est fixée à 0.1, ce qui conduit à une moyenne de 7 symboles par phrase ($\mu = 7, \sigma = 4$), sous la condition qu'une phrase contienne au moins un symbole. Afin d'être plus proche d'une situation réelle, un mécanisme de génération aléatoire de bruit provoque de temps à autre l'ajout ou la suppression d'un symbole. Le niveau de bruit est constant au cours de l'expérience et uniforme sur l'ensemble des symboles. La probabilité d'erreur pour chaque symbole est de 0.05, ce qui provoque en moyenne une erreur (insertion ou délétion) toutes les 2.5 phrases.

7.3 Participants

18 participants d'âge moyen 25 ans ont été recrutés pour cette expérience sur la base du bénévolat. L'expérience consistait en la présentation successive de 75 phrases sur l'interface décrite précédemment. Aucune contrainte de temps n'était donnée et la durée approximative de chaque expérience était de 45 minutes.

Les participants était avertis de l'existence de régularités dans les phrases proposées. Aucune indication précise n'était donnée sur la nature de la grammaire (nombre de chunks, nombre de niveaux dans la hiérarchie, etc.). Il était explicitement demandé aux participants d'indiquer les symboles « ayant tendance à apparaître conjointement » . Afin de motiver les participants à le faire, il leur était spécifié que créer les « bons regroupements » permettait de gagner des points et qu'un classement final serait établi entre les participants. La création de chunks incorrects ou incomplets était bien sûr autorisée, et aucun retour n'était donné aux participants quant à la justesse des chunks créés.

Afin de se rendre compte de la nature difficile de la tâche et de la concentration qu'elle demande, le lecteur est invité à se rendre annexe B.4 où sont proposés les 25 premiers stimuli. L'apparence visuelle des symboles a été permutée par rapport à celle de la figure 7.3 page ci-contre afin de permettre au lecteur de créer quelques chunks qu'il pourra à sa guise comparer avec les résultats des autres participants et du MDLChunker présentés dans la suite.

7.4 Résultats du MDLChunker

Le MDLChunker ne comportant aucun paramètre, les résultats qu'il fournit ne dépendent en rien des résultats des participants. Nous avons fait passer l'expérience simultanément aux 18 participants et au MDLChunker. Le fonctionnement de ce dernier étant déterministe, il n'a été lancé qu'une seule fois sur le jeu de données. Les traces récupérées chez les participants contiennent le décours temporel de la création des différents chunks. Le MDLChunker fournit la même information, ainsi que les tailles de codages associées à chaque chunk. Dans le souci de comparer le MDLChunker et les participants sur la même base, nous

n'avons pas utilisé l'information relative aux tailles de codage. Dans la suite, le MDLChunker est considéré de la même façon que les autres participants.

7.4.1 Positionnement multi-dimensionnel

Tous participants confondus (MDLChunker inclus), 99 chunks différents ont été créés au cours des 75 itérations (phrases). Afin d'évaluer la position relative du MDLChunker et des participants pour chacune des 75 itérations, chaque participant ainsi que le MDLChunker peut être représenté comme un point dans l'espace des 99 chunks possibles. La grande dimension de cet espace rend impossible toute visualisation directe. La figure 7.4 page suivante utilise un positionnement multi-dimensionnel (Borg & Groenen, 2005) pour projeter [4] les différents points sur le plan de plus forte inertie.

Sur cette représentation, le MDLChunker occupe une position relativement centrale au cours des itérations : il semble se comporter comme un participant moyen. Nous allons maintenant essayer de caractériser rigoureusement cette notion de « position centrale » .

7.4.2 Détail des chunks créés

Chunks créés par les participants

On observe de fortes différences inter-individuelles pour les chunks créés. Les 99 chunks créés se répartissent comme suit :
– 75 chunks sont possédés par un participant seulement
– 5 chunks sont possédés par deux participants
– 8 autres chunks sont possédés par moins d'un tiers des participants
– 11 chunks sont possédés par plus d'un tiers des participants
Un grand nombre de chunks (88%) sont possédés par moins d'un tiers des participants, dont la plupart par un seul participant. Dans la suite, ces chunks assimilables à du bruit sont considérés comme non-représentatifs dans la mesure où ils sont soumis à de très fortes variations inter-individuelles. Le MDLChunker doit être validé sur sa capacité à être un bon prédicteur, c'est-à-dire à reproduire l'information structurée et non pas le bruit. La séparation entre chunks non-représentatifs et chunks représentatifs (fixée à un tiers) est assez nette pour que les résultats soient peu sensibles à la valeur du seuil choisi : ils sont identiques que le seuil soit fixé à un tiers ou à la moitié des participants (Fig. 7.5 page 160).

Les chunks non-représentatifs sont pour la plupart des dérivations de chunks représentatifs, c'est-à-dire des chunks représentatifs auxquels il manque des symboles ou auxquels des symboles supplémentaires ont été ajoutés. Ils peuvent être vus comme des chunks représentatifs erronés ou bruités.

4. Le positionnement multi-dimensionnel ne prend pas en entrée les coordonnées des différents points, mais une matrice de dissimilarités entre paires de points. Nous avons utilisé une distance de Manhattan comme mesure de dissimilarité.

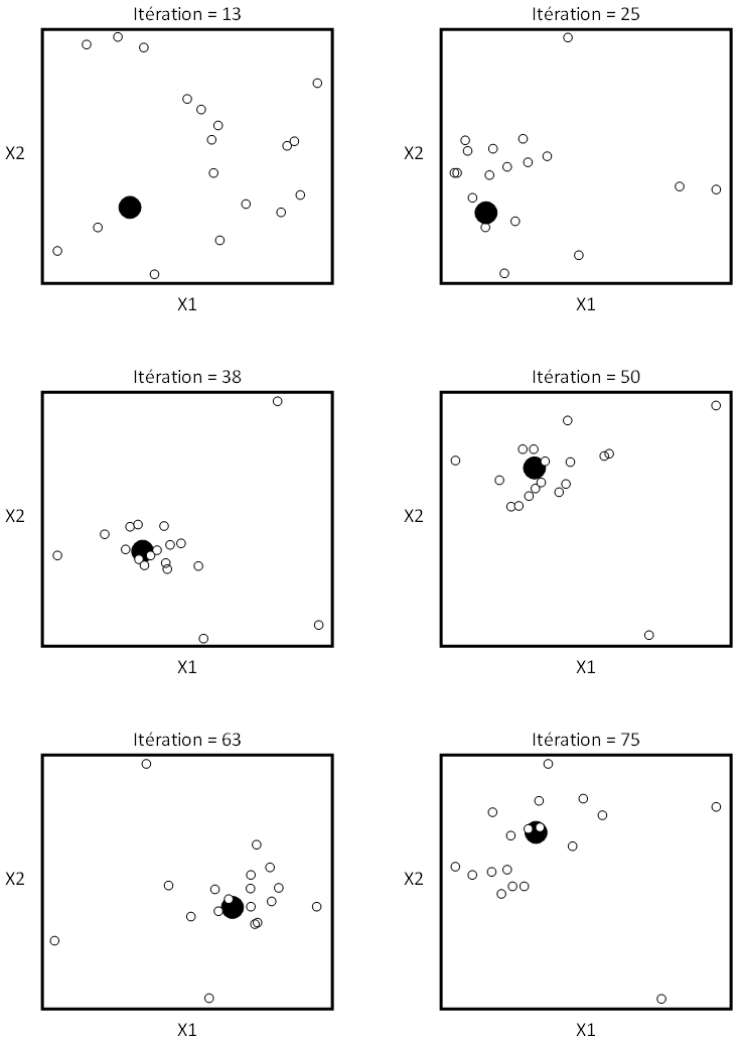


FIGURE 7.4 – Positionnement bi-dimensionnel présentant l'évolution de la position relative du MDLChunker et des 18 participants au cours des 75 itérations (par pas de 1/6=12.5 itérations). Les participants sont représentés par des cercles évidés et le MDLChunker par un cercle plein. Les axes de plus forte inertie (X1 et X2) ne sont pas nécessairement identiques au cours des itérations.

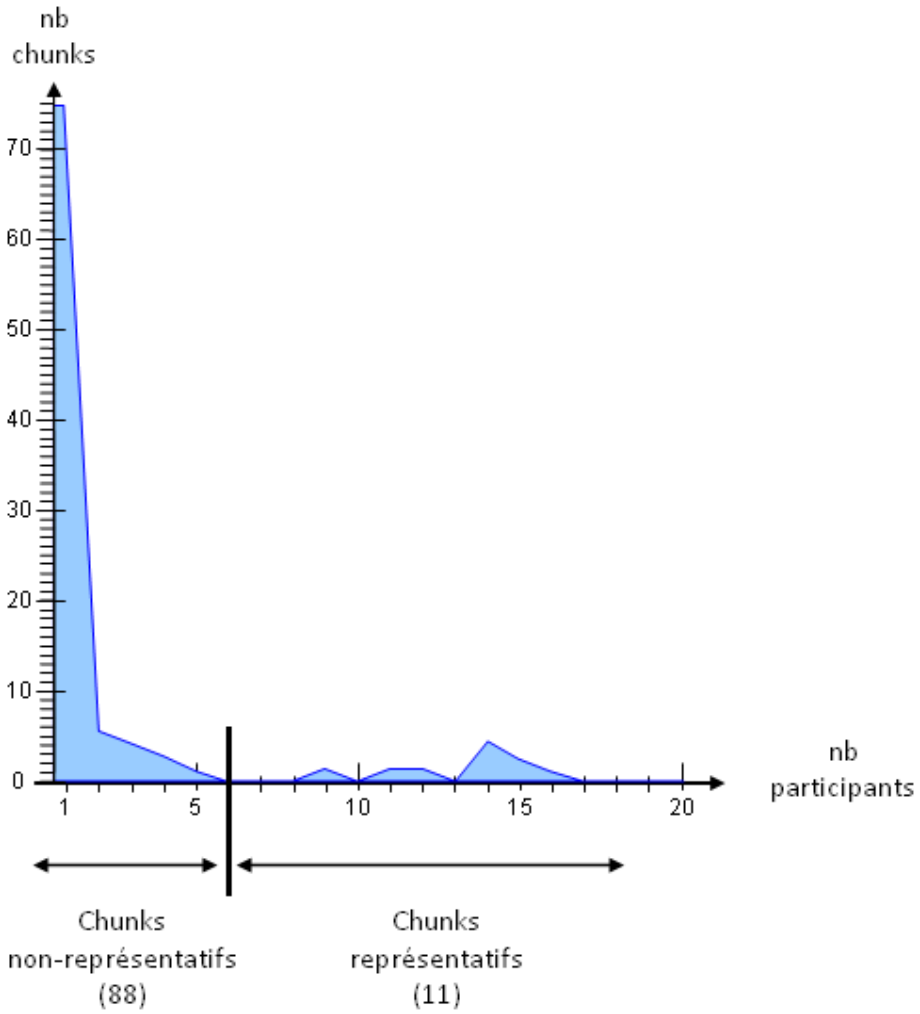


FIGURE 7.5 – Répartition des 99 chunks en fonction du nombre de participants qui les possèdent. Le premier mode de cette distribution bi-modale correspond aux 88 chunks non-représentatifs qui sont possédés par moins d'un tiers des participants. Le second mode correspond aux 11 chunks représentatifs qui sont possédés par la majorité des participants.

Chunks de la grammaire initiale

Parmi les 11 chunks initialement contenus dans la grammaire, 9 ont été créés par la majorité des participants et 2 ne l'ont pas été. Il s'agit du chunk n°8 (créé par aucun participant) et du chunk n°10 (créé par deux participants). Deux chunks qui n'étaient pas contenus dans la grammaire ont été créés par une majorité de participants. Ils sont notés 3' et 4' car ils correspondent aux dérivations des chunks 3 et 4, auxquels un symbole a été ajouté (resp. retiré).

Chunks créés par le MDLChunker

Tous les chunks représentatifs créés par les participants (11 chunks) ont également été créés par le MDLChunker. Cela inclut les deux chunks 3' et 4' qui n'étaient pas présents dans la grammaire de départ. Le fait que le MDLChunker produisent les mêmes « erreurs » que les participants est important dans la mesure où le MDLChunker n'est pas validé sur sa capacité à retrouver la grammaire initiale mais bien sur sa capacité à se comporter comme un participant moyen. Les chunks 8 et 10 de la grammaire (ceux qui n'ont pas été créés par les participants), n'ont pas non plus été créés par le MDLChunker. Deux chunks supplémentaires notés 3" et 6' (dérivations des chunks 3 et 6) ont été créés par le MDLChunker sans avoir été créés par la majorité des participants. Le chunk 3" n'a été créé que par 2 participants et le chunk 6' par 4 participants (Tab. 7.1).

Chunk n°	1	2	3	3'	3"	4	4'	5	6	6'	7	8	9	10	11
Grammaire	✓	✓	✓	.	.	✓	.	✓	✓	.	✓	✓	✓	✓	✓
Participants	✓	✓	✓	✓	.	✓	✓	✓	✓	.	✓	.	✓	.	✓
MDLChunker	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	.	✓	.	✓

TABLE 7.1 – Tableau récapitulatif des différents chunks contenus dans la grammaire ainsi que ceux créés par les participants (chunks représentatifs uniquement) et l'ensemble de ceux créés par le MDLChunker.

Décours temporel de la création des chunks

À l'issue des 75 itérations, les chunks créés par le MDLChunker sont une bonne prédiction des chunks créés par la majorité des participants : tous les chunks majoritairement créés par les participants ont été créés par le MDLChunker et seulement 2 chunks supplémentaires ont été ajoutés par le MDLChunker (contre 5 en moyenne par participant). On souhaite maintenant évaluer si le MDLChunker permet de prédire correctement le moment (itération) de création de chaque chunk. Pour chaque chunk du tableau 7.1, la figure 7.6 page suivante présente l'itération à laquelle le chunk a été créé par chacun des 18 participants ainsi que par le MDLChunker. Bien que l'évaluation de la performance du modèle à travers cette représentation soit essentiellement qualitative, c'est la seule qui soit réellement objective dans la mesure où elle contient l'intégralité de l'information recueillie.

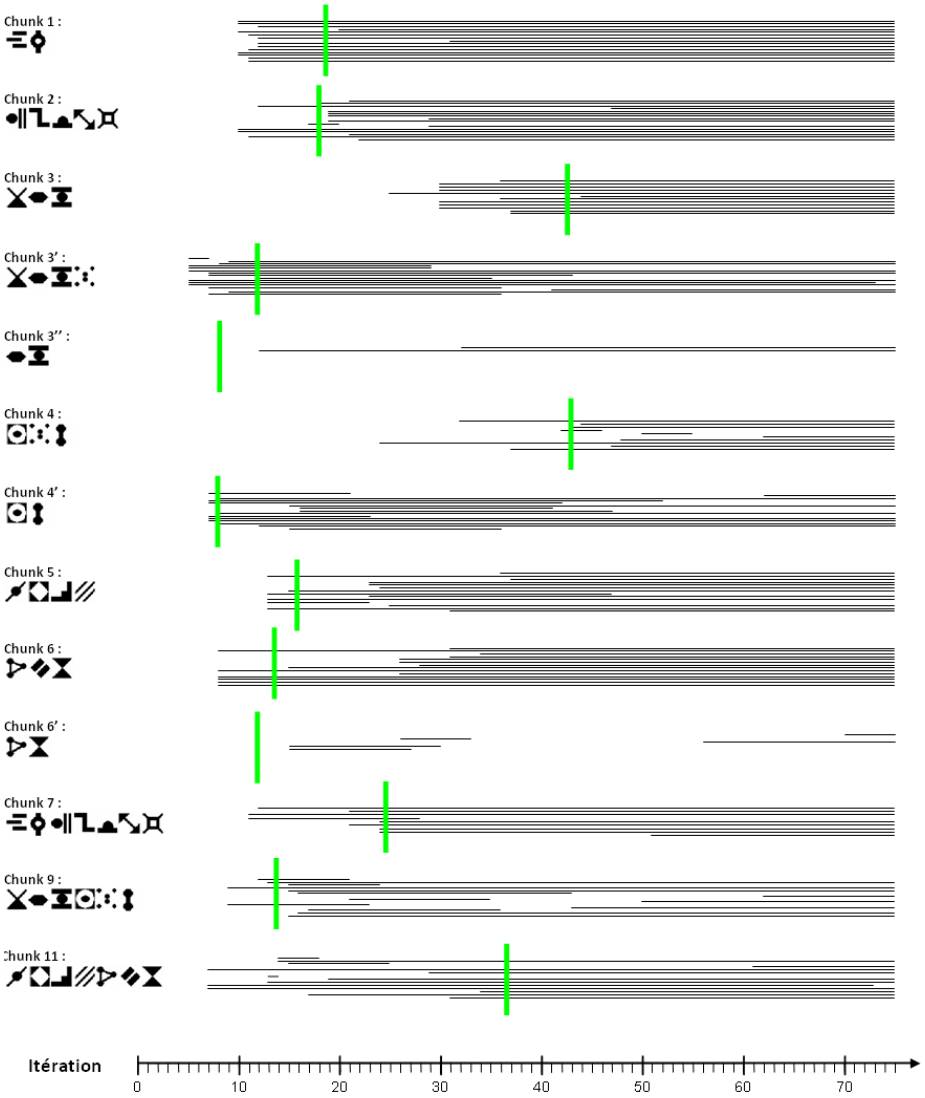


FIGURE 7.6 – Figure présentant pour chaque chunk, l'itération à laquelle il a été créé par chacun des 18 participants. Chaque ligne horizontale représente un participant. Le début de la ligne correspond à la date de création du chunk et la longueur de la ligne représente sa durée (nombre d'itérations pendant lesquelles le chunk est demeuré inchangé). Tous les chunks créés par le MDLChunker sont représentés (13 chunks). 11 de ces chunks sont possédés par une majorité de participants (chunks 1, 2, 3, 3', 4, 4', 5, 6, 7, 9 et 11) et 2 par une minorité de participants (chunks 3" et 6'). L'itération à laquelle le chunk est créé par le MDLChunker est matérialisé par une barre verticale.

7.4.3 Validation

La position centrale qu'occupe le MDLChunker par rapport aux autres participants dans l'espace des chunks peut également être évaluée de façon quantitative. Pour chacune des 75 itérations, il est possible de calculer la distance [5] moyenne de chaque participant à l'ensemble des autres (le MDLChunker étant considéré comme un participant supplémentaire). Le participant dont la distance moyenne à tous les autres est la plus faible est celui qui occupe la position la plus centrale dans l'espace des chunks. La figure 7.7 présente un classement des différents participants, de celui qui occupe la position la plus centrale à celui qui occupe la position la plus périphérique, pour chacune des 75 itérations

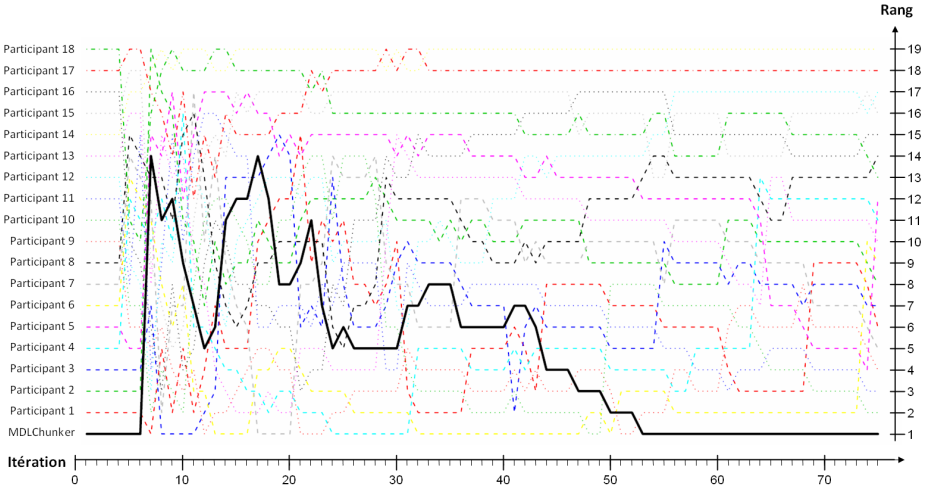


FIGURE 7.7 – Rang des différents participants au cours des 75 itérations. Le rang le plus faible correspond au participant ayant la position la plus centrale (distance moyenne aux autres participants la plus faible). Le MDLChunker est représenté en trait plein.

À une itération donnée, le participant dont le rang est le plus faible est le meilleur représentant de l'ensemble des participants. Il est possible d'utiliser le rang moyen comme indicateur de la performance d'un participant au cours des 75 itérations. Selon ce critère, le MDLChunker occupe la troisième place derrière les participants 6 et 9, avec un rang moyen de 4.6. La moyenne et l'écart-type des rangs moyens pour l'ensemble des participants sont $\mu = 10$, $\sigma = 4.7$. La moyenne des distances du MDLChunker aux autres participants au cours des 75 itérations vaut 10.7 ($\mu = 14.2$ [6], $\sigma = 4.5$).

En résumé, le MDLChunker peut être considéré comme le troisième meilleur représentant des participants à l'expérience. C'est un bon prédicteur du comportement moyen des participants.

5. On utilise une distance de Manhattan, chaque participant étant représenté par un vecteur dans l'espace des 99 chunks.

6. Il s'agit de la moyenne sur les 19 participants, de la moyenne sur les 75 itérations, de la moyenne des distances de chaque participant à tous les autres.

7.5 Résultats du MDLChunker-approché

Dans cette section, on s'intéresse à la validation des performances du MDLChunker-approché. Les conditions de l'expérience et les indicateurs utilisés pour la validation sont en tous points identiques à ce qui a été décrit pour le MDLChunker.

7.5.1 Détail des chunks créés

Les résultats obtenus par le MDLChunker-approché sont moins bons que ceux du MDLChunker. 2 des chunks créés par la majorité des participants n'ont pas été créés par le MDLChunker-approché. Il s'agit du chunk 6 (premier niveau hiérarchique) et du chunk 11 (second niveau hiérarchique). Les 9 autres chunks qui sont majoritaires chez les participants ont bien été créés par le MDLChunker-approché, mais 11 chunks non-majoritaires ont également été créés, ce qui rend les prédictions du MDLChunker-approché bien moins bonnes que celles du MDLChunker. Ces 11 chunks non-majoritaires comprennent notamment les 2 chunks non-majoritaires créés par le MDLChunker (chunks 3" et 6')

Pour ce qui est du décours temporel de la création des chunks, les résultats obtenus par le MDLChunker-approché sont en revanche très acceptables au vu des approximations réalisées. Ces résultats sont présentés figure 7.8 page suivante. Afin de savoir quels sont les 11 chunks non-majoritaires créés, la figure 7.9 page 166 représente l'évolution de la structure interne du MDLChunker-approché par pas de 10 itérations.

7.5.2 Validation

De même que pour le MDLChunker, la validation peut être faite sur l'évolution du rang qu'occupe le MDLChunker-approché. Le rang le plus faible correspond à la position la plus centrale et le rang le plus fort à la position la plus périphérique. Les résultats sont présentés figure 7.10 page 167.

Au cours des 75 itérations, le rang moyen occupé par le MDLChunker-approché vaut 15.1 (μ = 10, σ = 4.7 pour les participants). Ce résultat le place à la 17^e position sur 19, devant le participant 14 et le participant 17. En termes de distance, la moyenne des distances du MDLChunker-approché aux autres participants au cours des 75 itérations vaut 20.6 (μ = 15.2, σ = 4.5).

Les résultats obtenus par le MDLChunker-approché sont nettement moins bons que ceux obtenus par le MDLChunker. Néanmoins, ces derniers restent acceptables. Contrairement au MDLChunker, le MDLChunker-approché n'occupe pas une position centrale dans le nuage de points [7]. Mais sa position, bien que relativement périphérique reste incluse dans le nuage (Fig. 7.11 page 168). Le MDLChunker n'étant pas le participant le plus périphérique (deux participants

7. Il s'agit du nuage de points évoqué précédemment, représentant les différents participants comme des points dans l'espace des chunks.

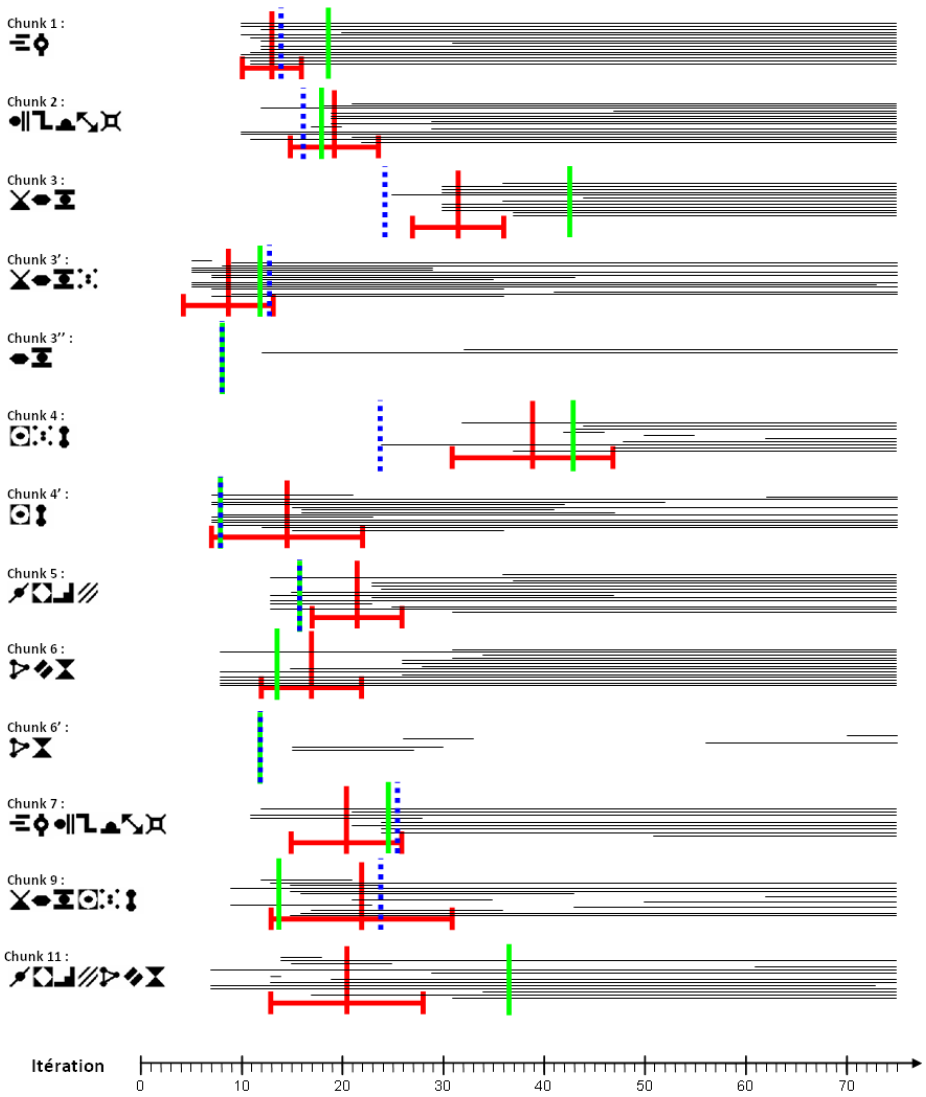


FIGURE 7.8 – Reprise de la figure 7.6 page 162 à laquelle sont ajoutés les résultats du MDLChunker-approché matérialisés par une barre verticale pointillée. 9 chunks supplémentaires ont été créés par le MDLChunker-approché et ne sont pas représentés ici. Pour plus d'information, voir la figure 7.9 page suivante. Pour une meilleure comparaison des performances, l'itération moyenne de création de chaque chunk par les participants est matérialisée par une barre verticale et l'écart-type correspondant par une barre horizontale afin de visualiser des différences inter-individuelles.

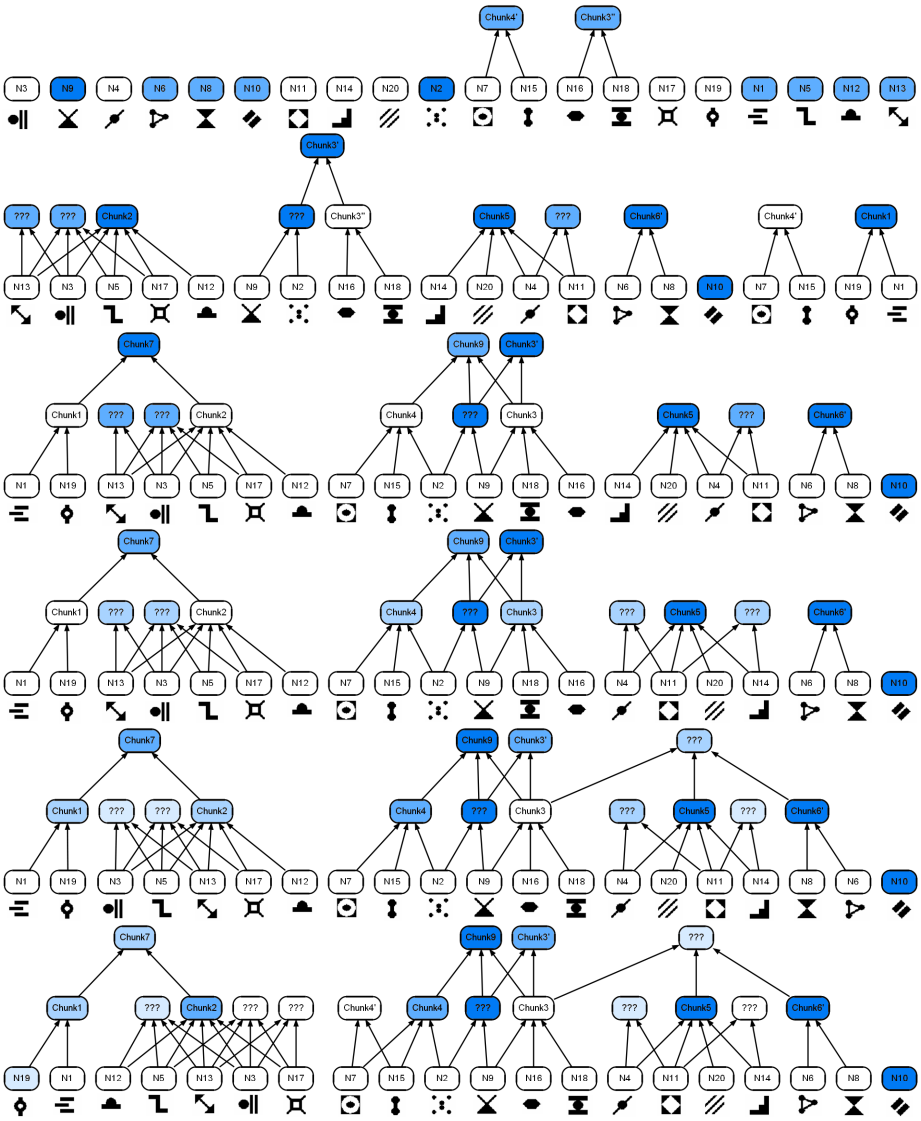


FIGURE 7.9 – Etat du MDLChunker-approché aux itérations 10, 20, 30, 40, 50 et 60 (de haut en bas). La coloration des noeuds du réseau est d'autant plus importante que la taille de codage des noeuds est faible. La taille de codage permet d'estimer qualitativement la pertinence du chunk, information qui manque à la figure 7.8 page précédente. Les 9 chunks créés par le MDLChunker-approché qui ne sont pas créés de façon majoritaire par les participants ou par le MDLChunker sont notés « ? ? ? » . Ils ne sont pas tous présents à la dernière itération, 3 d'entre eux ayant disparu au cours de l'apprentissage (notamment le chunk 3").

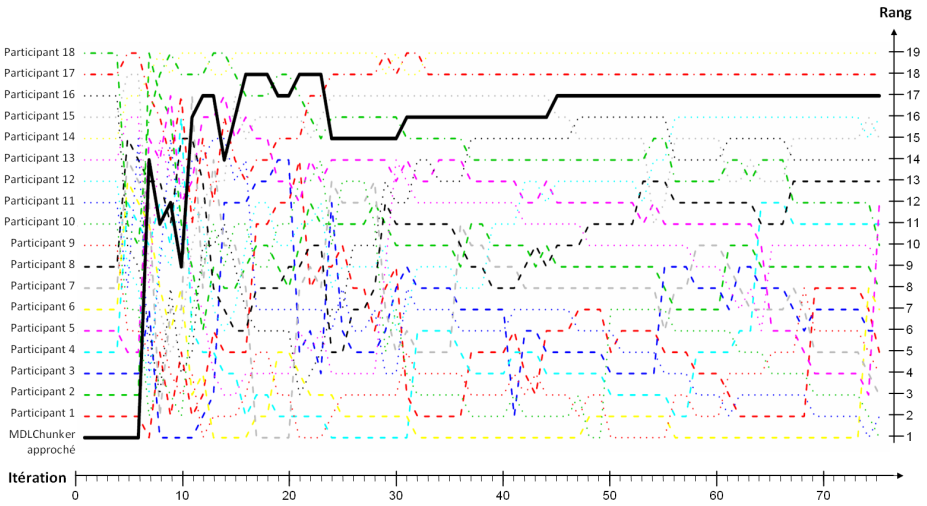


FIGURE 7.10 – Rang des différents participants au cours des 75 itérations. Le rang le plus faible correspond au participant ayant la position la plus centrale (distance moyenne aux autres participants la plus faible). Le MDLChunker-approché est représenté en trait plein.

ont fait pire), les résultats qu'il obtient restent comparables à ceux des participants.

7.6 Conclusion

Nouvelle expérience

Nous avons vu que la création d'une nouvelle expérience permettait d'augmenter les exigences de validation par rapport au paradigme classique utilisé en AGL (*Reber task*). Cela permet de passer de la validation d'un modèle sur sa capacité à prédire une moyenne, à la validation d'un modèle sur sa capacité à prédire non seulement les chunks créés, mais également le décours temporel de cette création. Le prix à payer pour atteindre ce niveau d'exigence de validation est de passer d'un paradigme d'apprentissage implicite à un paradigme d'apprentissage explicite. La conséquence est qu'un certain nombre de chunks risquent de ne pas être accessibles consciemment, augmentant ainsi la distance entre les chunks rapportés par les participants et les prédictions du modèle. Ce problème n'est pas apparu de façon prononcée puisque seuls deux chunks ont été prédits par le MDLChunker sans avoir été créés par une majorité des participants. Dans l'éventualité inverse, il aurait été nécessaire d'utiliser l'information fournie par le MDLChunker sur la taille de codage des chunks. Cela aurait nécessité l'ajout d'un paramètre au modèle afin de séparer les chunks « inconscients » des chunks « conscients » [8]. Ajouter un paramètre n'est pas souhaitable car les prédictions

8. La notion de chunks « conscients » ou « inconscients » n'a évidemment pas de signification réelle pour le modèle : le terme « inconscient » s'appliquant aux chunks dont la taille de

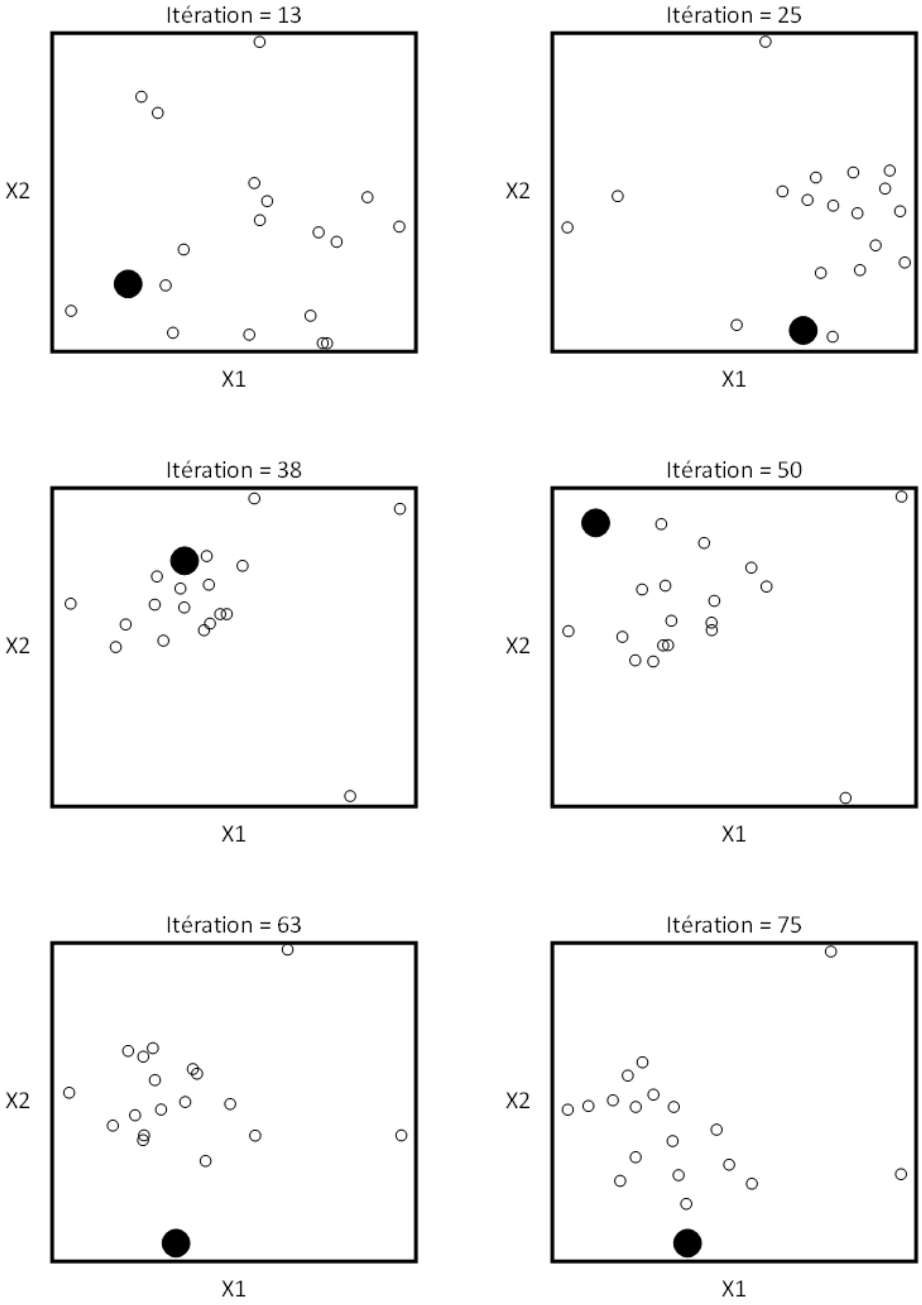


FIGURE 7.11 – Positionnement bi-dimensionnel présentant l'évolution de la position relative du MDLChunker-approché et des 18 participants au cours des 75 itérations.

du modèle deviendraient alors dépendantes de l'ajustement de ce paramètre qui ne peut se faire qu'a posteriori. Le modèle ne serait plus indépendant des résultats des participants.

Difficultés liées à cette expérience

La principale difficulté liée à cette expérience est qu'elle demande de la part des participants une concentration importante. C'est d'ailleurs pourquoi le recrutement des participants a été fait exclusivement sur la base du bénévolat. Pour cette raison, il ne paraît pas réaliste d'augmenter le nombre de symboles de la grammaire ou le nombre de phrases présentées sans par ailleurs intégrer au MDLChunker une modélisation de la baisse d'attention au cours du temps.

Dans le souci de concevoir une interface et une grammaire visant à éviter que les participants n'utilisent de connaissances a priori ou de mécanismes non-modélisées par le MDLChunker, c'est l'inverse qui s'est produit. Le MDLChunker qui était en mesure de capturer des régularités que les participants ne pouvaient rapporter. L'introduction de bruit dans la grammaire était susceptible de supprimer un symbole dans un chunk (A B C D E → A C D E) sans que le participant ayant créé le chunk correct (A B C D E) ne soit en mesure de déplacer en une fois le chunk incomplet (A C D E). L'interface créée ne permettait pas aux participants d'exprimer la négation (A C D E = A B C D E - B) tandis que le MDLChunker était en mesure de le faire. Nous avons vérifié a posteriori qu'aucune négation n'avait été utilisée par le MDLChunker.

D'une façon générale, la volonté de prédire le décours temporel de la formation des chunks a rendu la conception de l'expérience plus délicate. Il a été nécessaire de chercher à limiter au maximum l'influence des effets non modélisés. Pour résumer les principales causes de divergence entre les prédictions du modèle et les résultats des participants, on peut les regrouper en trois parties :

- La première est l'existence inévitable de connaissances a priori pouvant influencer la création de certains chunks.
- La seconde est qu'en l'absence d'une organisation évidente, les participants imposent la leur. Servan-Schreiber et Anderson (1990) citent l'exemple de la simple énumération des lettres de l'alphabet qui se fait de façon discontinue : abc def gh ij kl mn op qr st uvw xyz. Cet effet a été notamment décrit par Tulving (1962) sous le nom de *subjective organisation*.
- La dernière est une conséquence du phénomène de *perception shaping* (Perruchet et al., 2002) qui est susceptible de devenir une cause très forte de divergence à long terme. Les chunks existants imposant une contrainte sur la perception de nouveaux stimuli, ils influencent en retour la formation de nouveaux chunks. Par exemple l'existence des chunks XKM et RN impose la structure B C XKM T T RN J T au stimulus BCXKMTTRNJT favorisant ainsi la création des chunks BC, TT et JT au détriment d'autres comme MTT ou NJ. On comprend ainsi aisément comment une faible erreur sur les premiers chunks créés par le modèle peut vite conduire à une forte divergence après

codage serait supérieure à un seuil fixé. Dans cette éventualité, seuls les chunks « conscients » du modèle seraient comparés aux chunks conscients rapportés par les participants.

quelques itérations. Cela vient du fait que le modèle n'a pas la possibilité de se recaler sur les résultats des participants.

Résultats

Les résultats obtenus par le MDLChunker le positionnent comme étant le troisième meilleur représentant des 19 participants pour l'ensemble des 75 phrases présentées. Le MDLChunker apparaît comme un excellent prédicteur du comportement moyen des participants humains. Il prédit l'intégralité des chunks majoritaires créés des participants en n'ajoutant qu'un bruit relativement faible (2 chunks). Une représentation du MDLChunker et des 18 participants dans l'espace des chunks confère au MDLChunker une position relativement centrale au cours des 75 itérations.

En comparaison, les résultats du MDLChunker-approché apparaissent nettement moins convaincants (17^e position sur 19). En effet, de la même façon que certains participants, le MDLChunker-approché a une certaine tendance à créer des chunks qu'il est le seul à posséder. Le MDLChunker-approché a un comportement proche de celui d'un participant réel (produisant un grand nombre de chunks qui lui sont propres), tandis que le MDLChunker a un comportement proche de celui qu'aurait un participant imaginaire idéal qui posséderait uniquement les chunks majoritaires.

Expériences futures

Bien que les résultats obtenus soient prometteurs, plusieurs problèmes restent à l'étude. C'est le cas notamment du niveau de bruit qui a arbitrairement été fixé pour cette expérience à 0.05 et dont l'influence n'a pas encore été testée. C'est également le cas de la grammaire utilisée. Une expérience est actuellement en cours sur quatre grammaires différentes pour lesquelles le niveau de bruit a été ramené à 0. Les différents paramètres de la grammaire qui ont été modifiés sont le nombre de chunks, l'arité des chunks et le nombre de niveaux dans la hiérarchie. Pour information, les grammaires utilisées sont données figure 7.12 page suivante, ainsi que les résultats obtenus pour la première grammaire (Fig. 7.13 page 172).

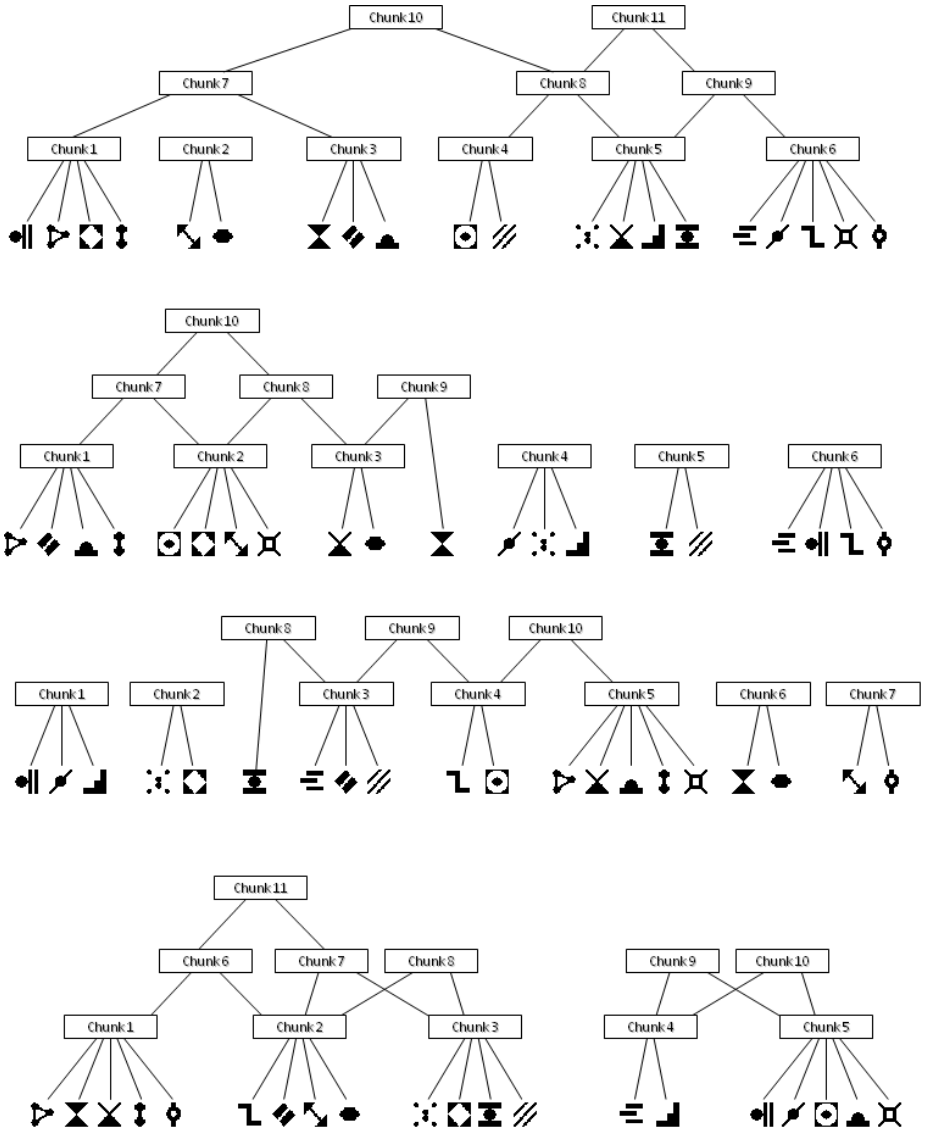


FIGURE 7.12 – Structure des quatre grammaires utilisées pour la nouvelle expérience.

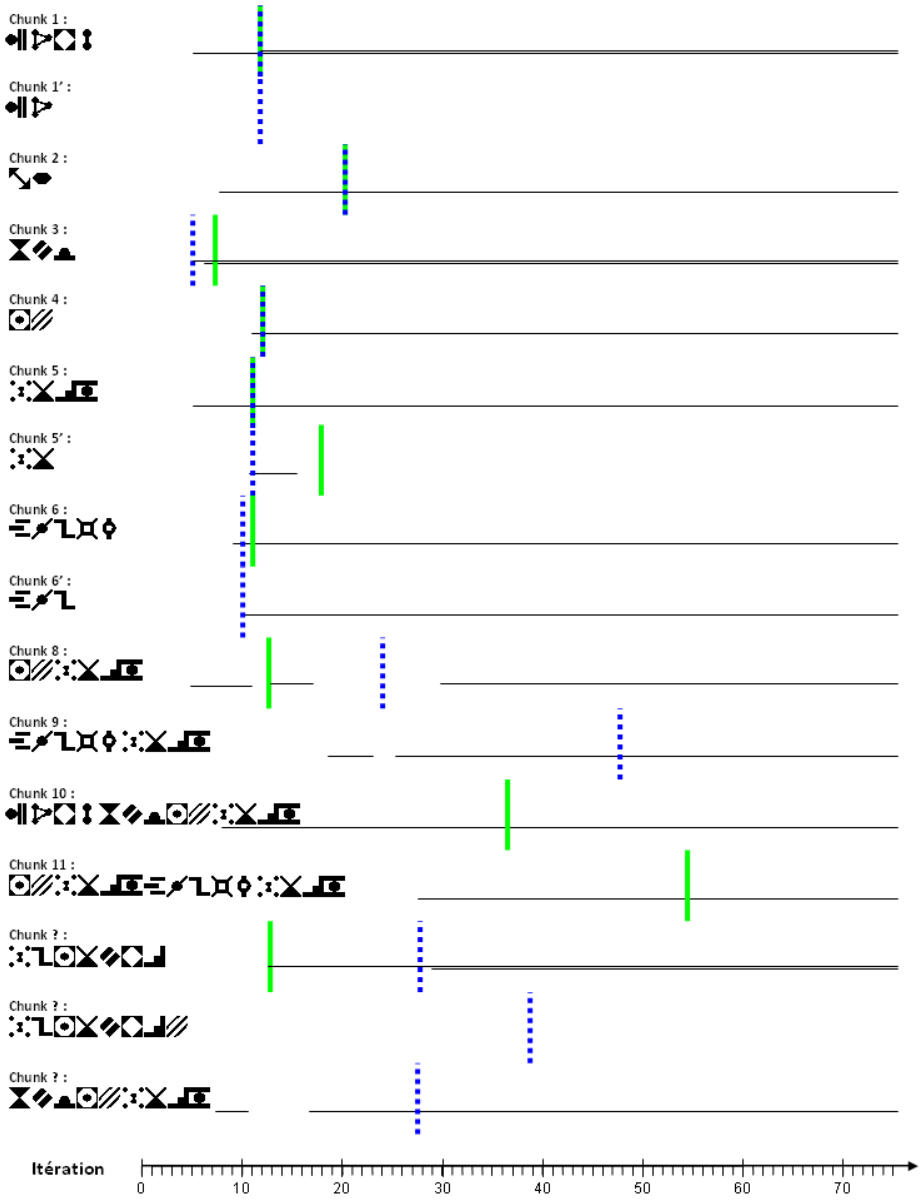
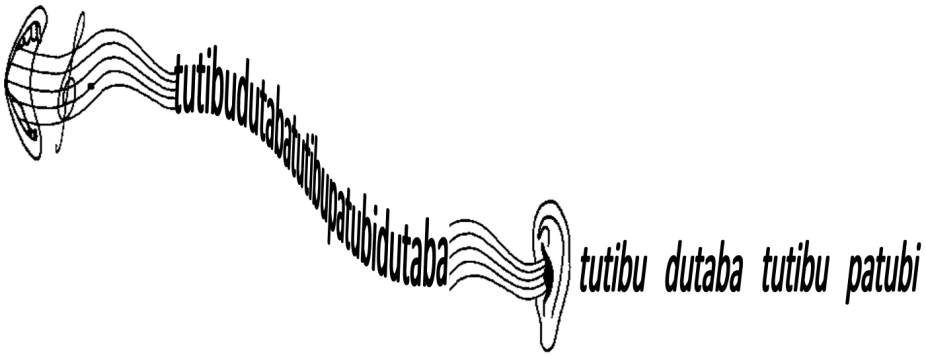


FIGURE 7.13 – Ensemble des chunks créés par le MDLChunker (trait vertical plein) et le MDLChunker-approché (trait vertical pointillé) pour la première grammaire de la figure 7.12. Les résultats préliminaires (deux participants) ont également été représentés. Les dérivations de chunks existants sont notés « X' » et les chunks nouveaux sont notés « ? » .

Chapitre 8

Application à la segmentation de mots

Sommaire



8.1 Introduction

Le but de ce chapitre est de tester la généralité du MDLChunker ainsi que sa capacité à être appliqué à un domaine pour lequel il n'a pas été conçu. Le MDLChunker est un modèle dont le fonctionnement est relativement simple et qui ne prétend pas modéliser d'autres phénomènes que le mécanisme de *chunking*. Aucune modélisation de la mémoire notamment, n'a été réalisée. Il s'agit comme nous l'avons vu (chapitre 1) d'un choix délibéré, mais qui limite le champ d'application aux expériences ne mettant pas en oeuvre de façon trop importante d'autres mécanismes cognitifs que le *chunking*. Dans ce chapitre, le MDLChunker est appliqué à la segmentation de mots. Pour que cela soit possible, trois modifications ont été apportées.

La première modification consiste à introduire une notion d'ordre entre les différents éléments d'un stimulus. Par défaut, le MDLChunker considère chaque stimulus comme un ensemble non ordonné de chunks-canoniques. Introduire un ordre entre ces chunks-canoniques ne pose aucune difficulté car cela consiste à n'envisager que les associations entre chunks adjacents (se reporter au chapitre 4 section 4.3.1 page 80 pour une discussion approfondie sur l'ordre).

La seconde modification consiste à introduire une modélisation du focus attentionnel afin de limiter la perception du flux de données. Les données utilisées pour la segmentation consistent en un flux continu de syllabes. Or dans sa version standard le MDLChunker prend en entrée un ensemble discret de stimuli. La modélisation du focus attentionnel va permettre une segmentation de ce flux continu en stimuli discrets.

La dernière modification, qui est également la plus importante, consiste à modéliser le phénomène d'oubli. Par défaut, le MDLChunker possède une mémoire infinie des stimuli perçus. Une limitation de la mémoire en terme de taille de codage est proposée, ainsi qu'une estimation de cette taille en nombre de bits.

À travers ces différentes modifications, on souhaite tester la capacité du MDLChunker à supporter l'ajout de modules supplémentaires, ainsi que sa capacité à être appliqué à de nouvelles tâches. Ce chapitre reprend sous une forme plus étendue le contenu décrit dans Robinet et Lemaire (2009).

Segmentation de mots

La capacité à percevoir les mots comme entités propres peut sembler naturelle à toute personne sachant lire, mais cette capacité est bien antérieure à l'apprentissage de la lecture. Les enfants sont capables de segmenter un flux de paroles en mots sans posséder d'indications claires sur ce qui peut marquer le début ou la fin d'un mot. Si l'utilisation d'indices prosodiques ou phonologiques peut jouer un rôle dans la reconnaissance des mots, les informations qu'ils fournissent sont essentiellement de nature probabiliste. Saffran et al. (1996) ont montré que de telles informations n'étaient pas indispensables pour expliquer la capacité de participants adultes à segmenter correctement un langage artificiel.

Soumis à un tel flux continu de syllabes, PARSER (décrit section 6.3 page 124) est en mesure de retrouver les mots constitutifs du langage artificiel utilisé (Perruchet & Vinter, 1998). Nous verrons que le MDLChunker en est également capable.

Effet d'évanouissement des sous-chunks

Nous nous intéresserons tout particulièrement à un effet décrit par Giroux et Rey (2009) et que nous appellerons « effet d'évanouissement des sous-chunks » . Cet effet est mis en avant par les auteurs pour comparer deux types de stratégies de segmentation : *bracketing strategy* et *clustering strategy*[1] (Swingley, 2005). Toutes deux permettent de rendre compte de la capacité à extraire les mots d'un flux de syllabes, mais seule la seconde est en mesure de reproduire l'effet d'évanouissement des sous-chunks observé chez les participants (se reporter à Perruchet et Pacton (2006) pour une comparaison des deux stratégies). Cet effet correspond au fait que certains chunks (appelés sous-chunks) sont d'abord appris avant d'être supplantés par des chunks englobants plus complexes. Il sera décrit en détail dans la section 8.3.

Les auteurs ont montré qu'il était possible de reproduire cet effet en utilisant PARSER comme modèle de type *clustering strategy*. Nous avons également pu le reproduire en utilisant le MDLChunker.

8.2 Segmentation de mots

Expérience

L'expérience originale proposée par Saffran et al. (1996) puis reprise par Perruchet et Vinter (1998) consiste à utiliser un langage artificiel composé de 6 mots de 3 syllabes :

- *pidabu*
- *patubi*
- *tutibu*
- *babupu*
- *bupada*
- *dutaba*

Ces mots sont ensuite concaténés dans un ordre aléatoire, en évitant les répétitions consécutives du même mot, puis lus par un synthétiseur vocal afin de supprimer tout indice prosodique susceptible de guider la segmentation :

1. La *bracketing strategy* consiste à trouver les bornes des mots et la *clustering strategy* consiste à créer des chunks de plus en plus longs (bigrammes, trigrammes, ...) jusqu'à obtenir des mots.

...bitutibudutabatutibupatubidutabapatubibupadapatubipidabututibupa...

Après 21 minutes d'écoute de cette séquence (4536 syllabes), une tâche de discrimination forcée est proposée dans laquelle les participants doivent déterminer entre deux mots lus, lequel semble le plus proche du langage artificiel. Chaque paire de mots est composée d'un mot correct (issu du langage artificiel) et d'un mot créé en concaténant aléatoirement 3 syllabes utilisées dans le langage (par exemple *patubi* contre *butiba*).

Résultats des participants

Les résultats montrent que les performances obtenues par les participants sur la tâche de discrimination (entre 65% et 75% de réponses correctes) sont significativement supérieures à la chance (Saffran et al., 1996). Ces résultats restent valables (59% de réponses correctes) dans le cas où il n'est pas demandé aux participants de se concentrer sur l'écoute, mais sur une autre tâche (création d'illustrations) (Saffran, Newport, Aslin, Tunick, & Barrueco, 1997).

Résultats de PARSER

Les résultats obtenus avec PARSER (Perruchet & Vinter, 1998) montrent que ce dernier est également capable d'extraire les 6 mots du langage artificiel. Après un apprentissage sur 4536 syllabes, les performances obtenues avec PARSER sur la tâche de discrimination surpassent celles des participants humains. Les auteurs définissent deux critères de convergence pour PARSER. Le critère faible, qui est atteint lorsque les 6 chunks de plus forte utilité correspondent aux 6 mots du langage (Tab. 8.1). Il est atteint en moyenne [2] au bout de 2064 syllabes (entre 735 et 4479 selon les simulations).

Chunk	Utilité
pidabu	21.57
dutaba	21.10
babupu	17.39
bupada	16.39
tutibu	14.60
patubi	13.75
bu	13.68
tu	5.74
tuti	4.87
pa	1.88
tutibupatubi	1.00

TABLE 8.1 – Utilité des différents chunks lorsque PARSER vérifie le critère de convergence faible. Tiré de Perruchet et Vinter (1998).

Le critère de convergence forte est atteint lorsque tous les chunks dont l'utilité

2. Moyenne effectuée sur 100 simulations

est supérieure à 1 sont soit des mots du langage, soit un groupement de mots du langage (Tab. 8.2). Des chunks correspondant à des groupements de mots du langage sont créés en permanence et disparaissent en permanence, la durée d'existence de chacun pouvant être très courte (se reporter au chapitre 6 pour une comparaison du mécanisme de création-oubli de PARSER avec le mécanisme d'optimisation du MDLChunker). Ce critère est atteint en moyenne au bout de 3837 syllabes (entre 1380 et 8796 selon les simulations).

Chunk	Utilité
dutaba	48.53
bupada	42.78
babupu	39.09
pidabu	37.58
tutibu	37.24
patubi	36.65
tutibupatubi	1.16
bupadapidabu	1.00

TABLE 8.2 – Utilité des différents chunks lorsque PARSER vérifie le critère de convergence forte. Tiré de Perruchet et Vinter (1998).

En négligeant les chunks regroupant plusieurs mots (dont l'utilité et la durée de vie sont très faibles), après 3837 syllabes en moyenne, PARSER est capable de retrouver les mots constitutifs du langage artificiel. En comparaison, la vitesse de convergence du MDLChunker est bien plus rapide.

Résultats du MDLChunker

Les résultats du MDLChunker ont été obtenus en introduisant une contrainte d'ordre comme cela est décrit au chapitre 4, et en segmentant arbitrairement le flux continu de syllabes en stimuli de longueur fixe [3]. Une longueur courte favorise la rupture des mots du langage et ralentit l'apprentissage. La longueur des stimuli est arbitrairement fixée à 10 syllabes pour ne pas être un multiple de la longueur des mots (3 syllabes). Les 6 chunks correspondant aux 6 mots du langage artificiel sont créés au bout de 18 stimuli (180 syllabes). Aucun autre chunk n'est créé. Le fonctionnement du MDLChunker étant déterministe, une moyenne sur plusieurs simulations n'est pas nécessaire.

De façon peu surprenante, le MDLChunker est en mesure de retrouver les différents mots du langage artificiel. L'absence de bruit, notamment, simplifie grandement la tâche. De même que pour PARSER, la convergence est plus rapide que pour les participants humains. Mais dans les cas du MDLChunker, l'absence de modélisation de l'oubli fait que l'ordre de grandeur obtenu (180 syllabes) est sans commune mesure avec celui observé chez les participants.

Nous allons maintenant décrire un autre effet lié à la même tâche de segmentation de mots. Cet effet, que PARSER ainsi que le MDLChunker sont en mesure

3. Dans la suite de ce chapitre sera présentée une modélisation du focus attentionnel permettant une segmentation du flux de syllabes sur la base des chunks créés par le MDLChunker.

de reproduire, suppose une organisation particulière de la hiérarchie de chunks et du mécanisme d'apprentissage. Par exemple, le mécanisme d'apprentissage du *Competitive Chunker* décrit au chapitre 6 ne permet pas d'expliquer cet effet.

8.3 Effet d'évanouissement des sous-chunks

Expérience

L'expérience proposée par Giroux et Rey (2009) pour montrer l'effet d'évanouissement des sous-chunks s'inspire de celle de Saffran et al. (1996). Un langage composé de 6 mots est artificiellement créé. Deux de ces 6 mots sont trisyllabiques (représentés formellement par ABC et DEF) et les 4 autres sont dissyllabiques (GH, IJ, KL et MN). Toutes les syllabes utilisées (A, B, C, D, ...) sont différentes. Les 6 mots ainsi créés sont concaténés aléatoirement en évitant les répétitions consécutives du même mot. La séquence de mots produite est ensuite lue par un synthétiseur vocal. Une séquence d'apprentissage différente issue du même langage artificiel est donnée à chaque participant (seul l'ordre des mots change).

Les participants sont séparés en deux groupes. Le premier est soumis à la tâche d'écoute durant 2 minutes (400 syllabes) et le second groupe durant 10 minutes (2000 syllabes). Deux tâches de discrimination forcée composées de 8 tests sont ensuite proposées aux participants. La première teste la performance des participants à reconnaître les 4 mots dissyllabiques du langage (GH, IJ, KL et MN). Chaque mot dissyllabique est testé contre une paire de syllabes (appelée par la suite « non-mot ») qui est composée d'une syllabe de la fin d'un mot et d'une syllabe du début d'un autre (Tab. 8.3).

test n°	mot (binaire)	non-mot (binaire)
1	GH	CK
2	IJ	FM
3	KL	CG
4	MN	FI
5	GH	FM
6	IJ	CK
7	KL	FI
8	MN	CG

TABLE 8.3 – Première tâche de discrimination utilisée par Giroux et Rey (2009) testant les 4 mots binaires contre des non-mots.

La seconde tâche de discrimination teste la performance des participants à reconnaître les sous-mots de deux syllabes (par exemple AB ou EF), contenus à l'intérieur des 2 mots trisyllabiques (ABC et DEF). Chaque sous-mot de deux syllabes est testé contre un non-mot (Tab. 8.4 page ci-contre).

test n°	sous-mot (binaire)	non-mot (binaire)
1	AB	HI
2	DE	JK
3	BC	LM
4	EF	NG
5	AB	JK
6	DE	HI
7	BC	NG
8	EF	LM

TABLE 8.4 – Seconde tâche de discrimination utilisée par Giroux et Rey (2009) testant les 4 sous-mots binaires contre des non-mots.

Résultats des participants

Les résultats obtenus par Giroux et Rey (2009) montrent qu'à l'issue d'une tâche d'écoute de 10 minutes, la capacité des participants à reconnaître les mots binaires est significativement plus élevée que leur capacité à reconnaître les sous-mots binaires. En revanche, aucune différence significative n'est observée à l'issue d'une tâche d'écoute de 2 minutes. Ces résultats suggèrent que l'apprentissage de sous-chunks (par exemple AB) est une étape nécessaire à la création de chunks plus longs (ABC), les sous-chunks disparaissant par la suite au profit des chunks qui les contiennent. C'est ce que nous appellerons de façon concise « l'effet d'évanouissement des sous-chunks » . Les auteurs ont reproduit des résultats identiques à l'aide de PARSER en utilisant les mêmes paramètres que ceux proposés dans Perruchet et Vinter (1998).

Résultats du MDLChunker

Le MDLChunker est entrainé sur le même langage artificiel afin de voir s'il est en mesure de reproduire cet effet d'évanouissement des sous-chunks. De même que pour l'expérience précédente, il est nécessaire de segmenter le flux de syllabes en stimuli de longueur finie. La longueur des stimuli est arbitrairement fixée à 5 syllabes.

Afin de moyenner les résultats, 1000 simulations du MDLChunker sont effectuées (correspondant à 1000 participants virtuels). Le MDLChunker ayant un fonctionnement déterministe, chaque simulation est lancée à partir d'une séquence d'apprentissage différente. Il s'agit du même protocole que celui utilisé pour les participants (décrit ci-avant) ainsi que pour PARSER. La figure 8.1 page suivante illustre l'évolution des tailles de codage des différents mots, sous-mots et non-mots utilisés par les deux tâches de discrimination.

Les résultats montrent qu'au début de l'apprentissage, il n'y a aucune différence entre les tailles de codage des mots dissyllabiques, celles des sous-mots et celles des non-mots. De façon peu surprenante, les tailles des non-mots deviennent rapidement supérieures aux autres. Au-delà d'une centaine de syllabes, les tailles des mots trisyllabiques deviennent inférieures à celles des sous-mots. Ces der-

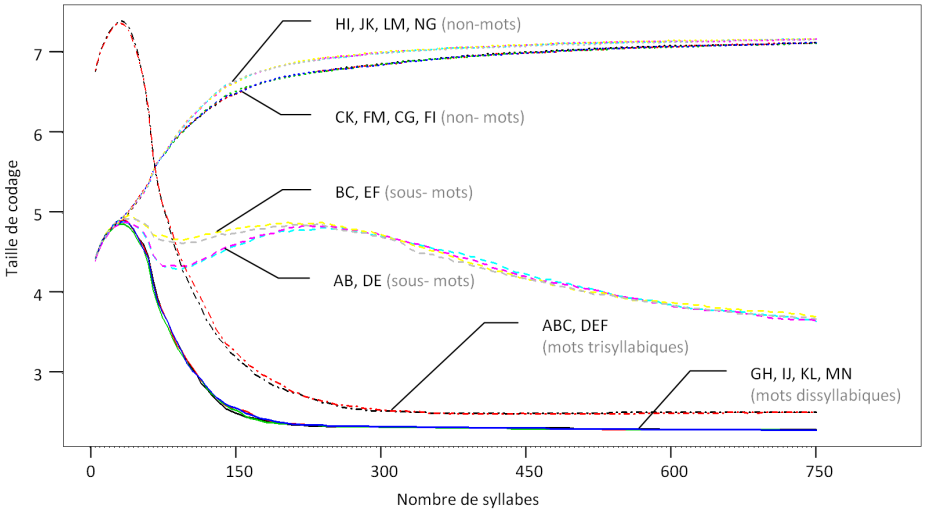


FIGURE 8.1 – Graphique présentant l'évolution de la taille de codage (en bits) obtenue avec le MDLChunker au cours de la perception des 750 premières syllabes. Les tailles de codages présentées correspondent aux tailles de codages moyennes obtenues pour 1000 exécutions du MDLChunker. Ces tailles de codage sont données pour les chunks correspondant aux 8 non-mots (HI, JK, LM, NG, CK, FM, CG, FI), aux 4 sous-mots (AB, BC, DE, EF), aux 4 mots dissyllabiques (GH, IJ, KL, MN) ainsi qu'aux 2 mots trisyllabiques (ABC et DEF). Les chunks correspondant aux mots trisyllabiques ont trivialement une taille de codage 50% plus élevée que ceux des mots disyllabiques à cause de la syllabe supplémentaire. L'opération de segmentation du flux d'entrée en stimuli de longueur finie conduit à une rupture plus fréquente des mots trisyllabiques que des mots disyllabiques. Les syllabes isolées issues de mots trisyllabiques sont donc plus fréquentes et leur taille de codage plus faible. Ainsi, les tailles des non-mots incluant une syllabe d'un mot trisyllabique sont légèrement plus faibles que celles des autres non-mots. La différence observable entre les tailles de BC, EF et celles de AB, DE s'explique par le fait que lorsque deux chunks binaires sont suceptibles d'être créés, le MDLChunker crée systématiquement le premier. Ainsi AB et DE sont créés plus fréquemment que BC et DE. Ces derniers ne sont créés que si le processus de segmentation de l'entrée en stimuli coupe ABC en A et BC. Cette situation est moins fréquente que celle consistant à trouver ABC ou AB et C.

nières commencent alors à augmenter, tandis que celles des mots continuent à diminuer : c'est l'effet d'évanouissement des sous-chunks.

Résultats sur la tâche de discrimination

Le MDLChunker est ensuite testé sur les deux tâches de discrimination conçues par Giroux et Rey (2009). Les résultats obtenus par le MDLChunker (1000 participants virtuels [4]) sont présentés figure 8.2 page suivante, ainsi que ceux obtenus par Giroux et Rey (2009) pour PARSER (32 participants virtuels) et pour les 32 participants humains ayant passé l'expérience.

Dans les trois cas, la performance obtenue sur les deux tâches de discrimination n'est pas significative après un court apprentissage, puis le devient par la suite. Les durées ont été fixés à 2 et 10 minutes pour les participants, ce qui correspond à 400 et 2000 syllabes pour PARSER. En conservant un rapport de cinq entre les deux durées, ce résultat est obtenu pour 15 et 75 syllabes avec le MDLChunker (Tab. 8.5).

Participants	2 minutes	10 minutes
p-value (test de Fisher)	0.56	0.01

PARSER	400 syllabes	2000 syllabes
p-value (test de Fisher)	*n.s.*	0.01

MDLChunker	15 syllabes	75 syllabes
p-value (test de Fisher)	0.41	10^{-16}

TABLE 8.5 – p-values obtenues avec le test de Fisher pour les participants humains, PARSER et le MDLChunker.

En utilisant le protocole expérimental de Giroux et Rey (2009), le MDLChunker est capable de reproduire l'effet d'évanouissement des sous-chunks. Néanmoins, en l'absence d'une modélisation du phénomène d'oubli, la vitesse d'apprentissage obtenue est bien supérieure à celle observée chez les participants humains. L'utilisation de stimuli de longueur fixe est également discutable, car en plus d'ajouter un paramètre, cela introduit des effets parasites comme nous l'avons vu figure 8.1. Une modélisation de l'oubli et du focus attentionnel ont donc été ajoutées.

4. Le choix d'utiliser 1000 participants virtuels pour le MDLChunker et non pas 32 est délibéré. Cela permet une plus grande confiance dans les résultats obtenus.

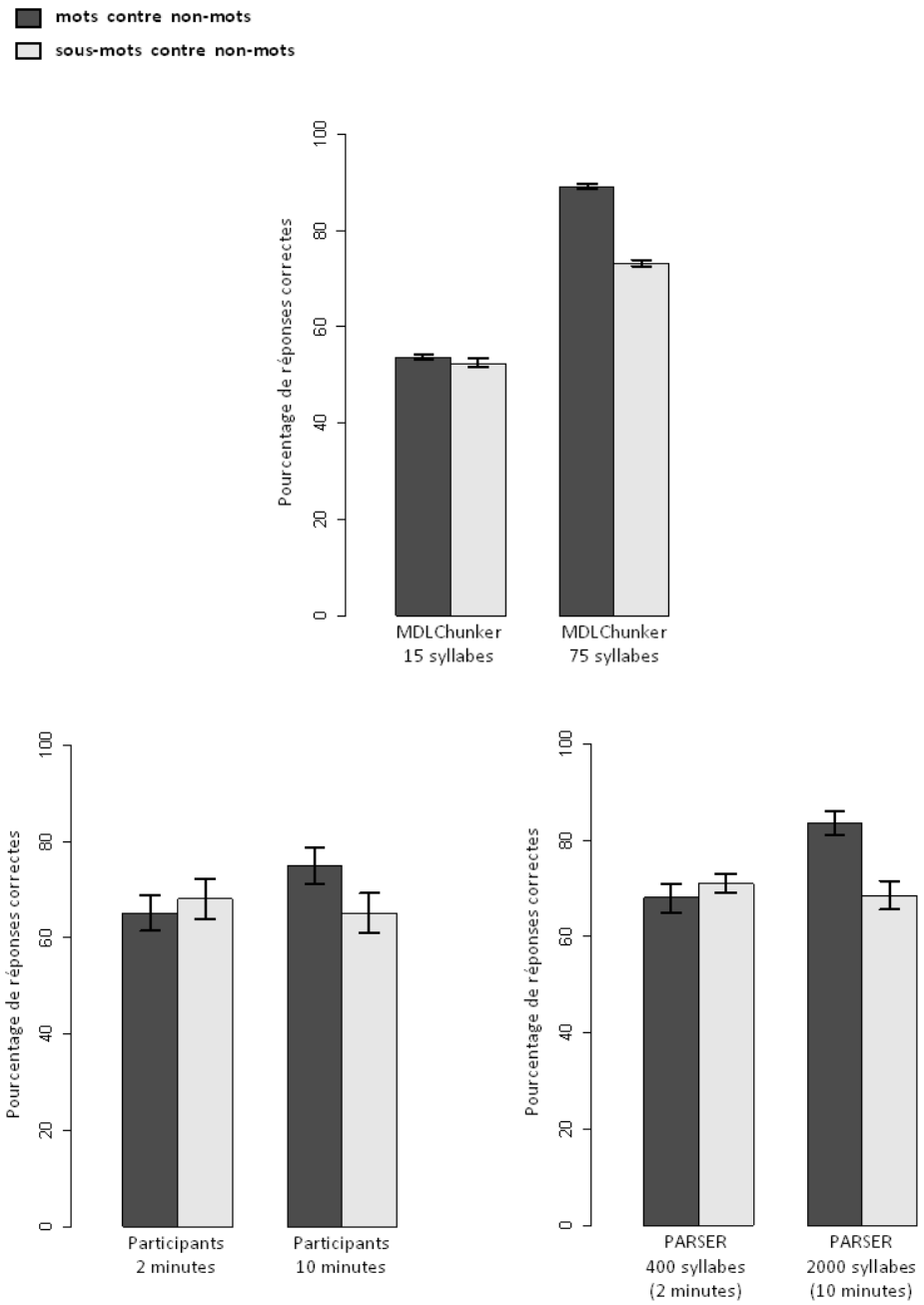


FIGURE 8.2 – Pourcentages de réponses correctes obtenues sur les deux tâches de discrimination pour les 32 participants humains, les 32 participants virtuels PARSER et les 1000 participants virtuels MDLChunker. Les résultats sont présentés après un apprentissage de 2 minutes (400 syllabes) et après un apprentissage de 10 minutes (2000 syllabes) pour les participants humains et PARSER. Ils sont présentés pour 15 et 75 syllabes pour le MDLChunker dont la vitesse d'apprentissage apparaît près de 30 fois supérieure.

8.4 Modifications apportées au MDLChunker

Modélisation du focus attentionnel

La modélisation du focus attentionnel présentée maintenant permet une segmentation cognitivement plus plausible du flux de syllabes. La segmentation du flux de syllabes n'est plus de longueur arbitraire mais dépend des chunks possédés à un moment donné par le MDLChunker. Cette approche est similaire au *perception shaping*[5] (Perruchet et al., 2002) utilisé par PARSER.

L'entrée courante (focus attentionnel) est factorisée sous forme de chunks de façon à minimiser sa taille de codage. La longueur du focus attentionnel est fixée à 7 syllabes afin de s'assurer que l'entrée contienne au moins deux mots complets (2 fois 3 syllabes plus 1). La factorisation de l'entrée en chunks fournit une segmentation naturelle du flux continu de syllabes. Le premier chunk joue alors le rôle de stimulus (Fig. 8.3). Il est retiré du flux de syllabes puis ajouté à la partie stimuli|chunks, comme dans le fonctionnement standard du MDLChunker.

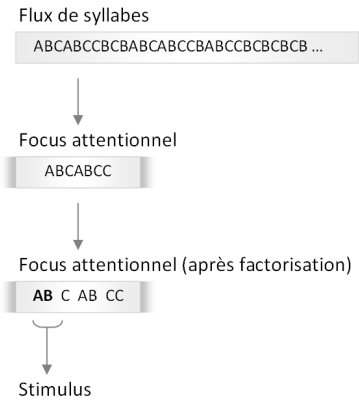


FIGURE 8.3 – Figure illustrant le fonctionnement du focus attentionnel. Le focus attentionnel porte sur les 7 premières syllabes du flux (ABCABCC), qui sont ensuite factorisées (AB C AB CC) en utilisant les chunks possédés. Cette factorisation suppose l'existence des chunks AB et CC (non représentés).

La modélisation du focus attentionnel a également une influence sur la création de nouveaux chunks (étape d'optimisation). Seuls les mots à l'intérieur du focus attentionnel sont candidats à la création d'un nouveau chunk. Le seul chunk dont la création est envisagée est celui associant les deux premiers mots du focus attentionnel. Il n'est effectivement créé que si cela entraîne une diminution de la taille de codage de l'ensemble du système.

5. Le *perception shaping* a été discuté aux chapitres précédents. Il s'agit du phénomène par lequel la perception d'un nouveau stimulus est affectée par les autres stimuli déjà perçus.

Modélisation de l'oubli

La modélisation de l'oubli qui est proposée consiste en une simple limitation de la taille de codage allouée à la partie stimuli|chunks. Cette limitation revient à considérer la partie stimuli|chunks comme une mémoire de taille fixe. Le fonctionnement du MDLChunker étant entièrement basé sur la notion de taille de codage, il est normal de faire porter cette limitation sur la taille de codage et non pas sur la longueur de la partie stimuli|chunks. Le fonctionnement de l'oubli est illustré par les figures 8.4 et 8.5 page suivante.

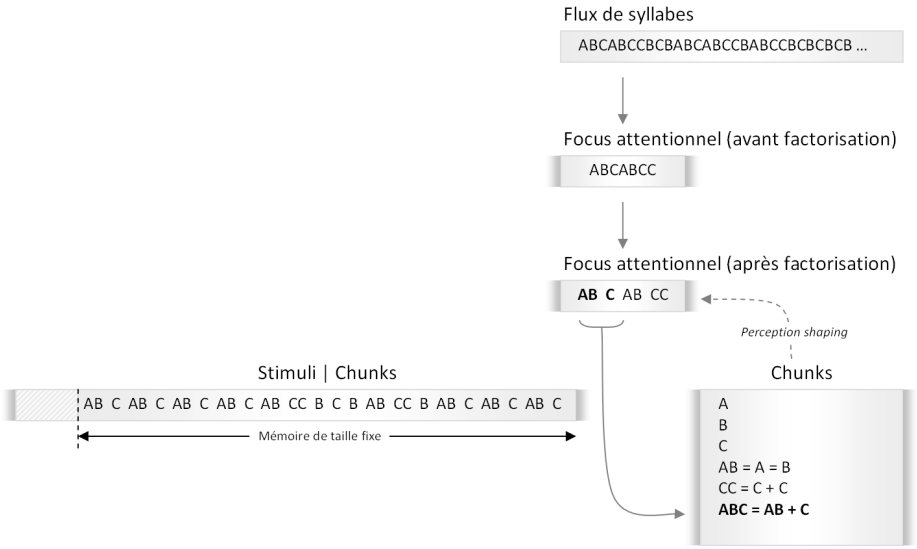


FIGURE 8.4 – Etat du MDLChunker avant perception du stimulus courant. Les chunks possédés (AB et CC) influencent la factorisation du focus attentionnel (*perception shaping*). Le chunk ABC regroupant AB et C est créé car la taille du système résultant (63.4 bits) est inférieure à celle qui serait obtenue sans ce chunk (65.5 bits).

Estimation de la taille de la mémoire

En choisissant de limiter la taille de la mémoire utilisée pour la partie stimuli|chunks, un nouveau paramètre (la taille de cette mémoire) a été ajouté au MDLChunker. Il est maintenant nécessaire de fournir une estimation de ce paramètre et voir dans quelle mesure les résultats précédents peuvent être reproduits.

L'utilisation d'une mémoire de grande taille (par exemple 1000 bits) revient à la situation décrite précédemment pour laquelle la vitesse d'apprentissage du MDLChunker est bien supérieure à celle observée chez les participants humains. À l'inverse, l'utilisation d'une mémoire de petite taille (par exemple 100 bits) empêche tout apprentissage dans la mesure où une trop faible quantité d'information est présente en mémoire pour que des régularités puissent être extraites. Différentes limites de mémoire ont été testées par pas de 10 bits, dans le but

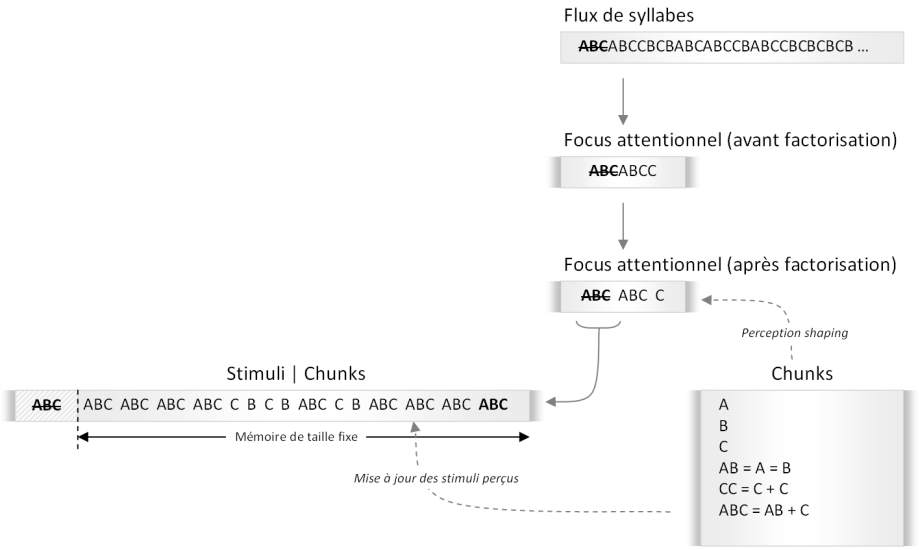


FIGURE 8.5 – La factorisation du focus attentionnel ainsi que celle de la partie stimuli|chunks est mise à jour pour tenir compte du nouveau chunk ABC créé. Le chunk est alors retiré du flux d'entrée puis ajouté à la partie stimuli|chunks. Les chunks les plus anciens dépassant la taille de mémoire autorisée sont supprimés (ici ABC).

d'ajuster la vitesse d'apprentissage du MDLChunker à celle des participants humains. La meilleure correspondance entre les deux a été obtenue pour une mémoire de 150 bits. Pour cette valeur, le MDLChunker est en mesure de reproduire à la fois l'effet d'évanouissement des sous-chunks, mais également le décours temporel de l'apprentissage. Après un entraînement de 2 minutes (400 syllabes), aucune différence significative ($p = 0.64$) n'est obtenue pour les performances du MDLChunker sur les deux tâches de discrimination. Cette différence devient significative ($p = 10^{-12}$) après 10 minutes (2000 syllabes).

8.5 Conclusion

L'application du MDLChunker à la segmentation de mots, puis à la reproduction de l'effet d'évanouissement des sous-chunks, nous a permis d'évaluer la généralité des principes utilisés. Cela nous a également permis de tester la capacité du MDLChunker à supporter l'ajout de modules additionnels. Trois modifications ont été introduites : l'ajout d'un ordre au sein des stimuli, la modélisation du focus attentionnel pour permettre le traitement d'un flux continu de données, et la modélisation de l'oubli comme étant une limitation du volume des données présentes en mémoire. Une estimation de cette taille a été proposée, et la valeur obtenue (150 bits) nous a permis de reproduire l'effet d'évanouissement des sous-chunks.

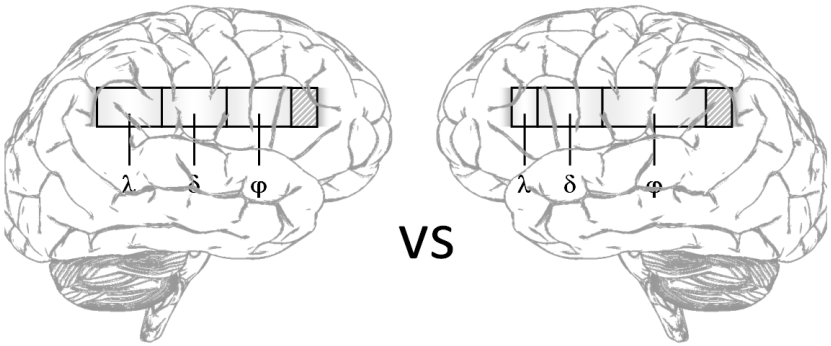
La limitation de la mémoire en terme de quantité d'information plutôt qu'en

terme de nombre de chunks (Miller, 1956) a servi de motivation à l'expérience du chapitre suivant. En effet, le MDLChunker a jusqu'ici été validé uniquement sur la nature des chunks et sur le décours temporel de leur création. Bien que la notion de taille de codage soit à la base du MDLChunker, cette information n'a pour le moment pas été exploitée. La raison principale est la difficulté de valider expérimentalement les résultats obtenus en terme de taille de codage. Le chapitre suivant tente de dépasser cette difficulté en utilisant les tailles de codage des chunks pour prédire la quantité de chunks pouvant être stockés en mémoire à court terme.

Chapitre 9

Application à la modélisation de la mémoire à court terme

Sommaire



9.1 Introduction

Les résultats qui sont présentés dans ce chapitre sont sujets à être complétés car il s'agit d'une piste encore à l'étude pour le moment. Il nous a néanmoins paru intéressant de les exposer dans cette thèse car ils s'intègrent parfaitement dans la continuité de ce qui a déjà été fait. Ce chapitre doit être pris pour ce qu'il est, c'est-à-dire une tentative de généralisation du MDLChunker à autre domaine. Les conclusions présentées visant à considérer la capacité de la mémoire à court terme comme contenant une certaine quantité d'information sont insuffisantes pour qu'il soit possible d'en tirer une conclusion définitive. Il s'agit simplement de montrer qu'une approche basée sur le principe de simplicité est également envisageable dans ce domaine et qu'elle reste cohérente avec les résultats expérimentaux obtenus.

La norme généralement admise est de considérer la mémoire à court terme comme comportant un nombre fixe de chunks. Différentes estimations de cette capacité ont d'ailleurs été données (Cowan, 2005 ; Gobet & Clarkson, 2004 ; Miller, 1956). La façon la plus naturelle d'évaluer cette capacité consiste à saturer la mémoire puis à utiliser une tâche de rappel afin de déterminer le nombre d'éléments qui peuvent être rapportés.

Le but de ce chapitre est de tester l'hypothèse selon laquelle la capacité de la mémoire à court terme peut être exprimée en terme de quantité d'information plutôt qu'en terme de nombre de chunks. Cette idée n'est pas nouvelle et avait déjà été évoquée et rejetée par Miller (1956). Nous verrons que les raisons invoquées par Miller pour écarter cette idée supposent l'utilisation d'un codage naïf des données très improbable au niveau cognitif.

L'idée de considérer la taille de la mémoire à court terme comme contenant une quantité d'information donnée et non pas un nombre d'éléments s'inscrit dans la continuité des chapitres précédents. Si comme nous l'avons supposé, les représentations en mémoire à long terme utilisent les régularités de l'environnement pour permettre une compression de l'information, alors il est tout naturel de penser que la mémoire à court terme tire, elle aussi, parti de cette compression. Cette remarque semble d'autant plus vraie pour la mémoire à court terme dont on sait que la capacité est limitée. De plus, il est normal de supposer que les représentations utilisées en mémoire à court terme ne sont pas différentes de celles créées en mémoire à long terme. Comme dans notre cas les représentations en mémoire à long terme sont définies par leur taille de codage, il apparaît relativement naturel de supposer que c'est cette taille de codage qui limite la mémoire à court terme.

Considérer les représentations en mémoire à court terme comme une liste de pointeurs vers les éléments correspondants situés en mémoire à long terme est une vision relativement classique (Cowan, 2005 ; Perruchet et al., 2007). C'est notamment celle qui est utilisée par CHREST (Gobet et al., 2001) (voir chapitre 6). Les représentations en mémoire à long terme du MDLChunker sont de la forme de celles présentées figure 9.1 page ci-contre. Ce qui est appelé taille du chunk est en réalité la taille nécessaire pour faire référence au chunk c'est-à-dire la taille d'un pointeur vers le chunk. Les tailles de codage associées à ces

pointeurs sont donc déjà connues du MDLChunker.

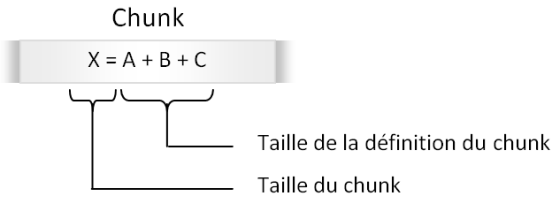


FIGURE 9.1 – Représentation des chunks pour les MDLChunker. La taille de la définition du chunk correspond à l'information qu'il contient. La taille du chunk correspond à l'information nécessaire pour retrouver le chunk en mémoire.

Il est important de noter que l'idée selon laquelle un chunk peut représenter une quantité arbitrairement grande d'information n'est pas remise en cause. La dissociation entre le chunk et son contenu persiste. La différence avec l'idée d'un nombre fixe de chunks proposée par Miller tient uniquement au fait que tous les chunks n'occupent pas nécessairement une taille identique en mémoire. Cela pourrait notamment expliquer pourquoi l'estimation du nombre de chunks en mémoire à court terme semble varier selon les tâches demandées : 7 pour Miller (1956), 4 pour Cowan (2005) et 2 pour Gobet et Clarkson (2004).

L'idée que nous souhaitons tester ici est que plus un chunk est fréquent, plus sa taille de codage est faible donc plus le chunk devrait être facile à mémoriser. L'exemple que l'on pourrait donner est la première paire de chiffres d'un numéro de téléphone. Tous les numéros de téléphones commençant par « 0 » , la quantité de mémoire nécessaire à la mémorisation du premier chiffre est négligeable. Pour la même raison, le second chiffre est également plus facile à retenir qu'un autre car seulement cinq chiffres sont possibles en seconde position (1, 2, 3, 4 ou 5 vers des téléphones fixes). De même si l'on souhaite mémoriser le contenu d'une boîte à outils donné, notre hypothèse est que la présence d'un marteau, d'une scie ou d'un tournevis étant très probable cette information est moins coûteuse à mémoriser que la présence d'un pointeau, d'un trusquin ou d'une platoire (Fig 9.2).

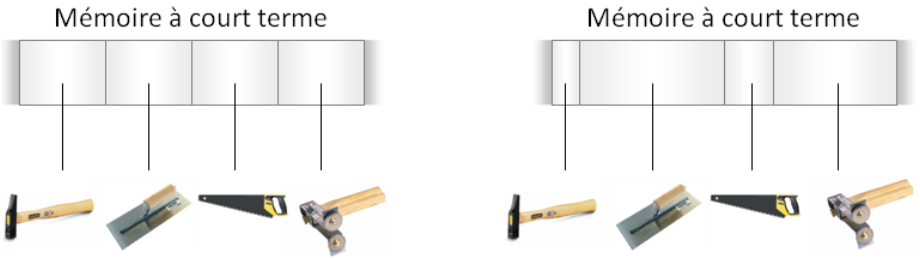


FIGURE 9.2 – Exemple très schématique illustrant la différence pouvant exister entre l'encodage de quatre chunks selon que leur taille soit supposée fixe ou qu'elle soit dépendante de la taille de codage des représentations impliquées.

Nous allons essayer de voir dans quelle mesure la quantité d'information peut être un meilleur indicateur de la capacité de la mémoire à court terme que le

nombre de chunks.

9.2 Nombre de chunks ou quantité d'information ?

Argument de Miller

La raison pour laquelle la notion de quantité d'information peut sembler un mauvais indicateur de la capacité de la mémoire à court terme est qu'elle apparaît très sensible au nombre d'éléments impliqués dans une expérience de rappel (Miller, 1956). Si on considère une tâche de mémoration-rappel de chiffres dans un alphabet contenant les chiffres 1, 2, 3 et 4, alors la quantité d'information théorique nécessaire pour encoder chaque élément est de 2 bits. Si les chiffres considérés vont maintenant de 1 à 8, alors le rappel de chacun d'eux nécessitera 3 bits d'information. En revanche, l'utilisation d'un alphabet contenant uniquement les nombres 2, 17, 316 et 1000 nécessitera elle aussi un codage de seulement 2 bits par élément.

Dans les deux cas présentés ci-dessus, le codage de l'information est un codage naïf qui ne tient pas compte des présupposés très forts que les participants peuvent avoir sur les nombres. Le fait que dans une expérience donnée la fréquence du chiffre 4 soit de $1/4$ et celle du chiffre 5 soit nulle ne signifie pas nécessairement que les présupposés des participants concernant ces deux chiffres disparaissent et que le codage du chiffre 4 puisse se faire sur seulement 2 bits. Il en va de même pour 2, 17, 316 et 1000. Si chacun d'eux devait être considéré comme un chunk (ce qui est purement hypothétique), utiliser un codage de 2 bits par élément suppose un a priori uniforme sur leur fréquence, ce qui n'est probablement pas le cas dans un contexte écologique.

La remarque faite ici porte uniquement sur l'a priori que les participants sont susceptibles de posséder sur la fréquence des différents chunks mis en jeu dans le cadre d'une expérience. Le codage optimal correspondant à l'alphabet restreint utilisé dans l'expérience n'est pas nécessairement un bon indicateur de la taille de codage réellement occupée par les chunks en mémoire.

Dissociation chunk / contenu

Vogel, Woodman, et Luck (2001) ont montré que des participants humains étaient capable de rappeler jusqu'à quatre objets indépendamment de leur complexité. Les performances de rappel obtenues apparaissent similaires, que les objets considérés contiennent une ou quatre caractéristiques (couleur, orientation). Cette expérience suggère l'existence d'une dissociation entre les chunks et leur contenu.

En réponse à cette expérience, Alvarez et Cavanagh (2004) ont testé le rappel de différents types d'objets. Les résultats semblent montrer que pour des objets simples comme les couleurs ou les objets de la vie de tous les jours (peigne,

stylo, brosse à dent) le taux de rappel est meilleur que pour des objets plus complexes comme des caractères chinois ou des polygones aléatoires. Les auteurs en concluent que la capacité de la mémoire à court terme gagne à être exprimée en terme de quantité d'information stockée plutôt qu'en nombre d'objets. Si leurs conclusions sont proches des nôtres, les raisons qui les motivent sont totalement différentes.

La difficulté principale qui se pose dans le cas de Alvarez et Cavanagh (2004) est que les objets choisis ne permettent pas facilement de se voir associer une taille de codage au sens de Shannon et Weaver (1949). Les auteurs estiment donc la complexité des objets à l'aide d'une quantité qu'il appellent « information visuelle » et qui correspond au temps passé à explorer visuellement chaque objet.

La conclusion que les auteurs tirent de leurs résultats est que la mémoire à court terme contient à la fois une limitation en nombre d'objets (la limite de cinq objets n'est jamais dépassée) et une limitation en quantité d'information visuelle. Cette dernière permet de rendre compte des capacités moindres à rappeler des objets complexes. À notre sens, ces résultats ne sont pas incompatibles avec ceux de Vogel et al. (2001) présentés ci-dessus. En effet, un glissement semble s'être opéré entre la notion initialement étudiée, qui est celle de chunk, et la notion d'objet qui est au centre de l'expérience de Alvarez et Cavanagh (2004). Rien n'assure qu'un objet, c'est-à-dire un ensemble de pixels visuellement contigus, soit nécessairement perçu sous la forme d'un chunk unique (Fig. 9.3). À cause de cette interprétation différente qui peut être faite des résultats, nous ne remettrons pas en cause l'idée généralement acceptée d'une dissociation entre le chunk et son contenu.

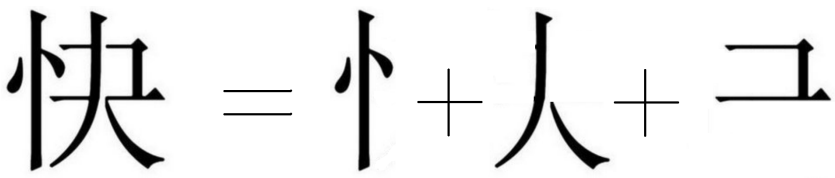


FIGURE 9.3 – Exemple de décomposition qui pourrait être faite d'un objet (caractère chinois) sous la forme de trois chunks déjà connus : un « arbre » , un « Y inversé » et un « U retourné » .

Expérience de Brady, Konkle et Alvarez

Le protocole expérimental que nous avons choisi reprend partiellement celui employé par Brady, Konkle, et Alvarez (2008). Les différences entre les deux sont expliquées dans la section suivante. L'expérience proposée par Brady et al. (2008) consiste à présenter un certain nombre de points colorés sur un écran d'ordinateur, puis à tester le rappel des participants pour un point particulier (Fig. 9.4 page suivante). Les points sont présentés par paires. Contrairement à l'expérience d'Alvarez et Cavanagh (2004) la fréquence d'apparition des paires de points est contrôlée ce qui permet d'estimer leur complexité en leur associant

une taille de codage (Shannon, 1948).

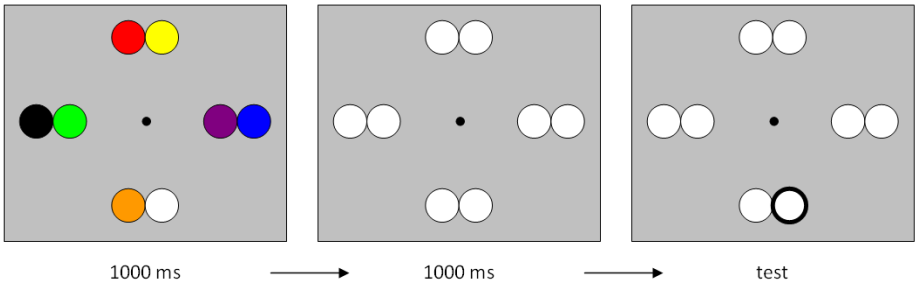


FIGURE 9.4 – Protocole expérimental utilisé par Brady et al. (2008). Huit points colorés sont présentés à l'écran autour du point de fixation pendant une durée de 1000 ms. Ils disparaissent ensuite durant 1000 ms avant qu'il soit demandé au participant de rappeler la couleur d'un point particulier.

Dans une première expérience, les huit couleurs apparaissent à des positions totalement aléatoires, rendant inutile l'association par paires. Les résultats obtenus montrent que les participants sont capables de mémoriser en moyenne 3.4 couleurs. Dans une seconde expérience, les couleurs ont tendance à apparaître par paires (rouge et jaune, noir et vert, etc.) dans 80% des cas. Dans cette situation les participants obtiennent une moyenne de 5.1 couleurs rappelées correctement. Les résultats obtenus dans les deux expériences sont compatibles avec une taille de mémoire à court terme de 10.1 bits.

Les participants étant en mesure de rappeler 5 couleurs dans le cas où elles sont appariées (au lieu de 3 dans le cas non-apparié) poussent les auteurs à en conclure que la capacité de la mémoire à court terme doit être pensée en terme de quantité d'information plutôt qu'en terme de nombre d'objets.

La remarque faite pour l'expérience de Alvarez et Cavanagh (2004) s'applique également ici. Le nombre d'objets distincts à l'écran n'est pas forcément un bon indicateur du nombre de chunks créés par les participants. Pour l'expérience de Alvarez et Cavanagh (2004), nous avons émis l'hypothèse qu'un symbole à l'écran pouvait contenir plusieurs chunks (Fig. 9.5 page ci-contre). Pour celle-ci, il y a tout lieu de penser qu'un seul chunk puisse regrouper deux couleurs. Les résultats obtenus peuvent être expliqués très simplement en supposant que les participants aient créé en moyenne deux chunks de type « rouge-jaune » ou « noir-vert » . En considérant pour la mémoire à court terme une capacité fixe de trois chunks, on rend à la fois compte des 3 chunks rappelés dans la première expérience et des 5 chunks rappelés dans la seconde.

L'hypothèse d'une limitation de la mémoire à court terme est compatible avec les résultats obtenus, mais elle n'apparaît pas indispensable pour les expliquer. La limitation en nombre de chunks de Miller (1956) est également compatible avec ces résultats.

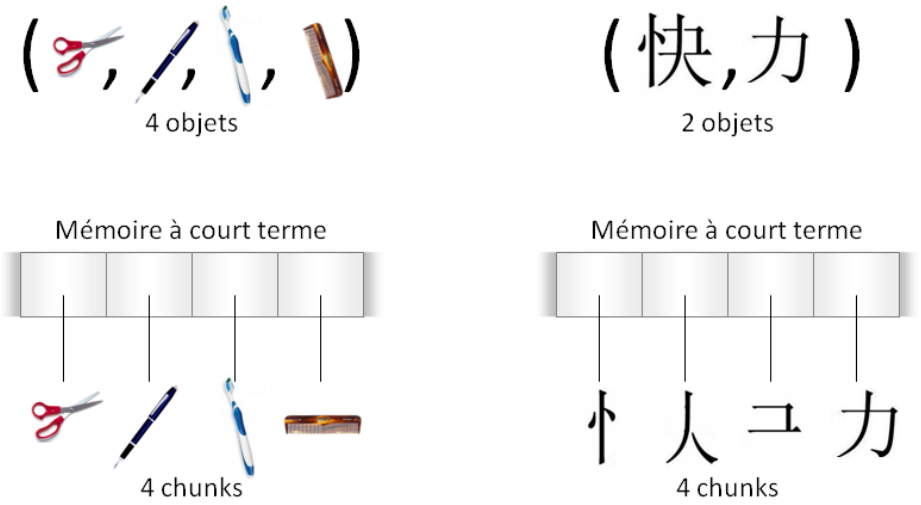


FIGURE 9.5 – Situation pour laquelle un objet est composé de plusieurs chunks.

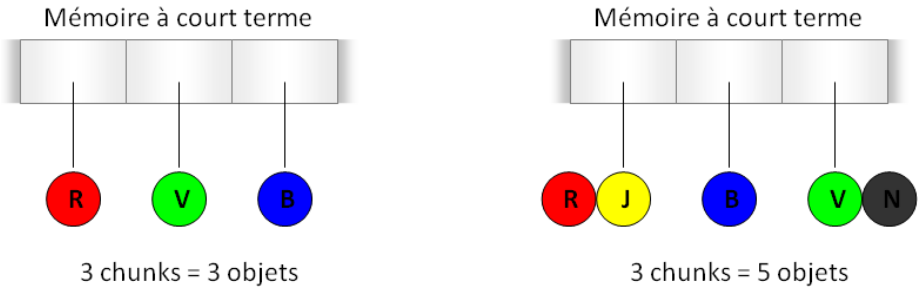


FIGURE 9.6 – Situation pour laquelle un chunk est composé de plusieurs objets.

Difficulté de distinguer le cas « quantité d'information » du cas « nombre de chunks »

Il apparaît que la tâche consistant à trouver une expérience qui permette de déterminer si la capacité de la mémoire à court terme doit être pensée en terme de quantité d'information ou en terme de nombre de chunks n'est pas aussi triviale qu'elle y paraît. La difficulté provient de l'incertitude sur la nature des chunks possédés à un moment donné. Il pourrait être envisageable d'utiliser le MDLChunker comme prédicteur des chunks créés. Nous avons vu au chapitre 7 que les prédictions effectuées peuvent être très correctes si les régularités ne sont pas trop complexes et que les participants font preuve de concentration. Le problème qui se pose dans le cas de la mémoire à court terme est que les performances inter-individuelles et même intra-individuelles sont extrêmement bruitées, et de loin supérieures à l'effet que l'on cherche à mesurer. Les résultats obtenus doivent donc être moyennés sur un grand nombre d'observations et ceci pour un grand nombre de participants.

On cherche à définir un protocole expérimental permettant de distinguer le cas « quantité d'information » du cas « nombre de chunks » . Pour cela il est nécessaire de contrôler les chunks créés par les participants, ainsi que leur fréquence. En faisant varier cette fréquence, il devient alors possible de tester si la mémoire à court terme est en mesure de contenir plus de chunks fréquents que de chunks peu fréquents. Si la mémoire contient un nombre fixe de chunks le fait d'augmenter la fréquence de l'un d'eux ne devrait pas avoir d'influence sur le nombre de chunks qui peuvent être rappelés. L'expérience décrite dans ce chapitre utilise ce principe (Fig. 9.7 page suivante).

Cela suppose de s'assurer que le chunk $\alpha\beta\gamma$ existe bien dès le départ et que l'effet d'augmentation du rappel ne correspond pas en réalité à la création progressive du chunk : $\alpha \rightarrow \alpha\beta \rightarrow \alpha\beta\gamma$. Plutôt que de faire varier la fréquence du chunk $\alpha\beta\gamma$, nous avons préféré définir trois chunks A, B et C de fréquences différentes mais fixes au cours du temps. Cela permet de moyenner plus facilement les taux de rappel obtenus.

9.3 Protocole expérimental

Le protocole expérimental qui va être décrit maintenant est encore largement perfectible. Il nous semble cependant qu'il permet d'éviter certains des problèmes liés aux expériences de Alvarez et Cavanagh (2004) et Brady et al. (2008). Contrairement aux expériences classiques qui consistent à supprimer la création de chunks et à contrôler la boucle phonologique (Cowan, 2005 ; Cowan, Chen, & Rouder, 2004 ; Cowan, Morey, Chen, Gilchrist, & Saults, 2008) notre but est de supprimer la boucle phonologique et de contrôler les chunks créés.

Le matériel se base sur l'utilisation de smileys comportant trois caractéristiques (les cheveux, les yeux , la bouche) et quatre modalités par caractéristique (Fig. 9.8 page 196) donnant lieu à 64 smileys possibles. Il y a six raisons

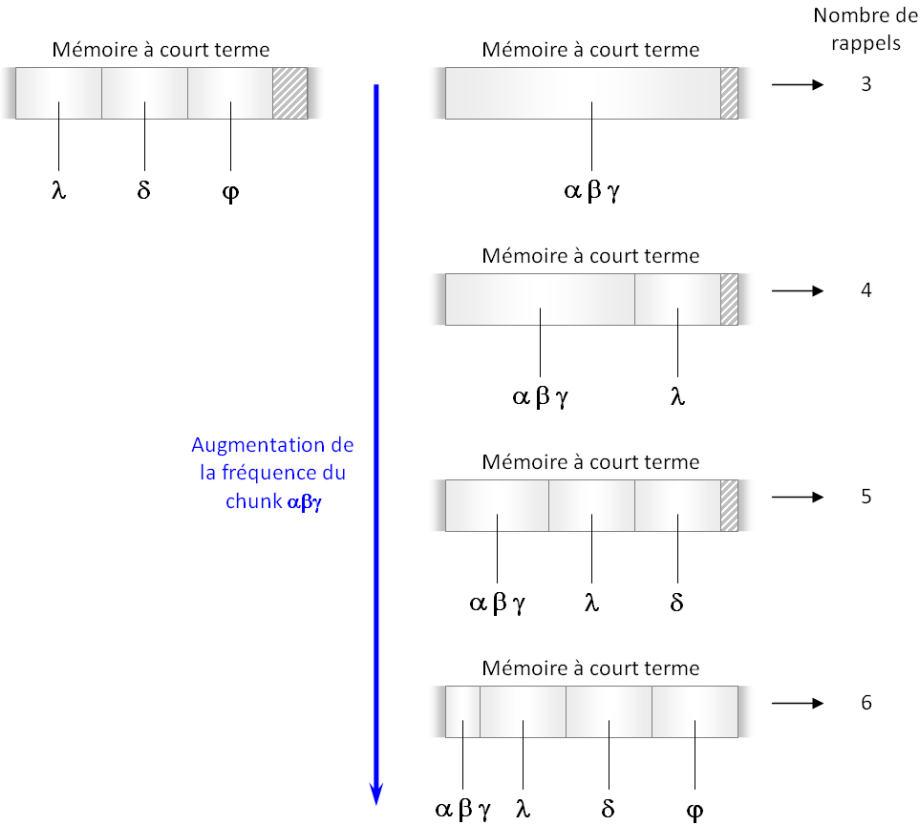


FIGURE 9.7 – Le principe de l'expérience consiste à posséder d'une part des chunks de taille fixe (λ, δ, ϕ), ainsi qu'un chunk ($\alpha\beta\gamma$) dont la fréquence augmente. La diminution de la taille de codage de ($\alpha\beta\gamma$) doit permettre un stockage plus important en mémoire et donc un meilleur rappel.

à ce choix (Fig. 9.9 page suivante). Tout d'abord, il est nécessaire que les représentations mentales associées aux différents objets utilisés soient immédiates et encodées sous forme d'un chunk unique. Il est peu probable que la bouche (ou même les yeux qui sont séparés) soient encodés comme deux chunks. Les résultats obtenus semblent d'ailleurs le confirmer.

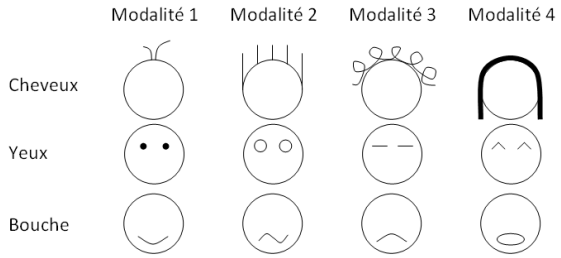


FIGURE 9.8 – Présentation des quatre modalités possibles pour chacune des trois caractéristiques des smileys.

La seconde raison est que, bien que la représentation de chaque élément du smiley soit immédiate, elle ne doit pas pouvoir être décrite par un mot court afin de ne pas inciter les participants à utiliser la boucle phonologique. C'est notamment pour cette raison que l'utilisation de couleurs a d'emblée été rejetée. Un contrôle rigoureux de la boucle phonologique a d'ailleurs été ajouté.

La troisième raison à ce choix est qu'il est nécessaire que les différentes modalités associées à chacune des caractéristique des smileys aient un poids similaire. Cela signifie que les participants ne doivent pas posséder un a priori trop fort risquant de favoriser un type de chevelure ou un type de bouche par rapport à un autre.

Il faut que la notion de chunk soit relativement naturelle afin qu'il ne soit pas nécessaire de demander explicitement aux participants de chercher des chunks. Il est assez logique de considérer les différents parties d'un visage (cheveux, yeux, bouche) comme faisant partie d'un tout et donc de chercher à les associer.

Il ne faut pas que les participants possèdent de présupposés trop forts quant aux chunks qui peuvent être formés. Si une association de deux ou trois modalités constitue déjà un chunk connu, alors cela facilite le rappel et introduit un bruit supplémentaire.

La dernière condition est que l'ordre des éléments ne doit pas avoir d'importance. On cherche uniquement à tester la capacité de rappel et non pas l'aptitude à associer un élément à une position donnée. Dans un smiley, l'ordre est intrinsèquement inexistant et il n'y a pas de risque de confusion entre la position de la bouche et celle des cheveux. Cela n'introduit donc pas de difficulté particulière liée à l'ordre et n'incite pas le recours à la boucle phonologique qui est particulièrement indiquée pour mémoriser une séquence ordonnée (Cowan, 2005).

Pour les six raisons qui viennent d'être évoquées, nous avons choisi de baser la tâche de mémorisation sur des smileys. La section suivante décrit en détail le matériel utilisé pour cette expérience.

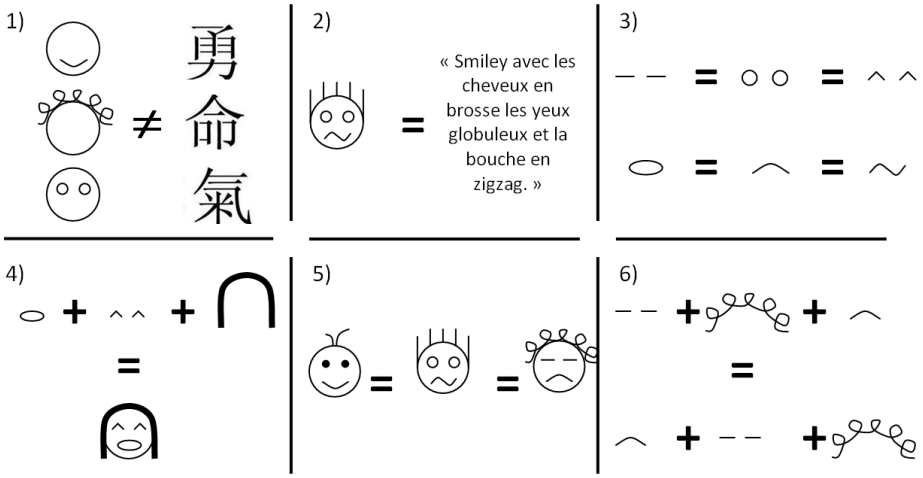


FIGURE 9.9 – Schéma résumant les six raisons qui justifient l'utilisation de smileys.

9.4 Matériel

Le but de cette expérience est d'inciter les participants à créer certains chunks dont la fréquence est soigneusement contrôlée afin de pouvoir en déduire les tailles de codage associées. L'expérience se déroule en deux phases : une phase d'apprentissage consistant à faire apprendre les chunks, et une phase de test dont le but est d'évaluer le nombre d'éléments rappelés en fonction de la taille de codage des chunks impliqués.

Phase d'apprentissage

La phase d'apprentissage vise à faire apprendre trois chunks ayant des tailles de codage différentes. Pour cela, trois smileys sont présentés successivement dans un ordre aléatoire et avec des fréquence d'apparition déterminées (Fig. 9.10). L'apprentissage se compose de 30 planches chacune présentant l'un des trois smileys (nommés A, B et C) pendant une durée de 10 secondes. Afin de s'assurer d'un minimum de concentration de la part des participants, une tâche de rappel est demandée après chacune des planches (Fig. 9.11 page 199, première partie).

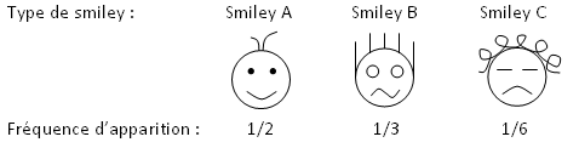


FIGURE 9.10 – Exemple de smileys utilisés durant la phase d'apprentissage. Leur fréquence d'apparition est indiquée au dessous.

On vérifie l'efficacité de l'apprentissage en révélant aux participants que cette

tâche était en réalité composée de trois smileys et en leur demandant de les rappeler. Le rappel est précédé d'une attente d'une minute destinée à s'assurer que les trois chunks rappelés sont bien en mémoire à long terme et non pas rappelés à partir de la mémoire à court terme.

Phase de test

Les participants passent ensuite à la phase de test qui comporte 60 planches et se déroule de la même façon que la phase d'apprentissage. Un certain nombre de smileys apparaissent à l'écran durant 10 secondes et à l'issue de ce délai les participants doivent en rappeler le maximum (Fig. 9.11 page ci-contre, seconde partie). La différence avec la tâche d'apprentissage réside dans le fait que le nombre de smileys présenté est bien supérieur à ce qu'il est possible de rappeler le but étant de saturer la mémoire à court terme. Le nombre de smileys présenté a été arbitrairement fixé à quatre [1].

Sept types de planches différentes sont présentées :

1. Planche contenant quatre smileys aléatoires (*i.e.* ne correspondant pas à l'un des trois smileys de la phase d'apprentissage).
2. Planche contenant un smiley de type C (fréquence d'apparition de 1/6) suivi de trois smileys aléatoires.
3. Planche contenant un smiley de type B (fréquence d'apparition de 1/3) suivi de trois smileys aléatoires.
4. Planche contenant un smiley de type A (fréquence d'apparition de 1/2) suivi de trois smileys aléatoires.
5. Planche contenant un smiley de type B, un smiley de type C suivi de deux smileys aléatoires.
6. Planche contenant un smiley de type A, un smiley de type C suivi de deux smileys aléatoires.
7. Planche contenant un smiley de type A, un smiley de type B suivi de deux smileys aléatoires.

Les smileys sont toujours présentés dans l'ordre indiqué ci-dessus, c'est-à-dire en débutant par les smileys A, B et C vus durant la phase d'apprentissage. Les smileys aléatoires qui les suivent jouent un rôle de remplissage de la mémoire à court terme. Les résultats attendus sont que la moyenne de rappel doit croître avec le type de planche.

Les planches de types 5, 6 et 7 sont présentées à la fin de la phase de test afin d'éviter la création de chunks du type AB, AC ou BC. Les planches de types 1 à 4 sont présentées dans un ordre aléatoire de façon à ce que les fréquences d'apparitions des smileys A, B et C soient identiques à celles de la phase d'apprentissage. Le nombre d'apparitions des planches 1, 2, 3 et 4 est entièrement déterminé par la fréquence des chunks A, B et C, qui doit demeurer constante au cours de l'expérience. Cette précaution est indispensable afin de s'assurer

1. Pour information, le meilleur rappel effectué les participants a été de trois smileys complets.

que l'inévitable apprentissage durant la phase de test ne modifie pas les tailles de codage relatives de ces trois chunks [2].

Le nombre de smileys aléatoires est suffisamment important (61) pour que le risque de création inopinée d'un chunk durant la phase de test puisse être considéré comme négligeable. Un contrôle additionnel est effectué afin de s'assurer qu'un même smiley n'apparaît pas deux fois sur la même planche ni à moins de cinq planches d'intervalle.

À l'issue de la phase de test, il est demandé de rappeler les trois chunks A, B et C de la même façon qu'après la phase de d'apprentissage.

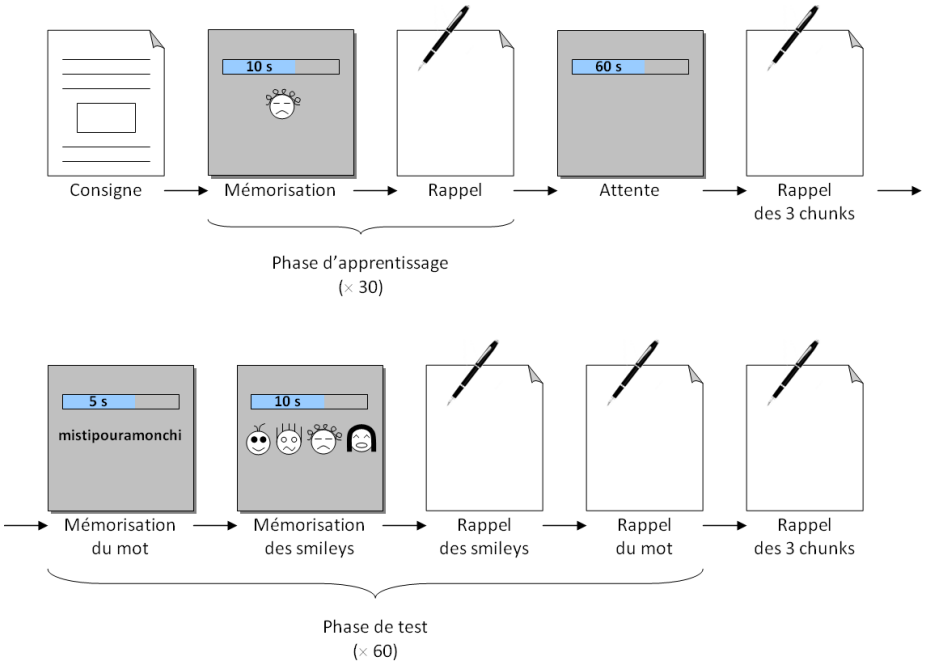


FIGURE 9.11 – Protocole expérimental utilisé.

Boucle phonologique

Afin de supprimer tout emploi de la boucle phonologique, chaque planche de test est précédée par un mot qui s'affiche à l'écran durant 5 secondes. Le mot doit être mémorisé puis rappelé à juste après la phase de rappel des smileys. Les mots présentés sont tous différents et ne sont pas des mots du dictionnaire afin d'éviter qu'ils puissent être traités comme des chunks. Ils sont composés de six syllabes afin d'être suffisamment longs pour épuiser les deux secondes de la boucle phonologique.

2. C'est la raison pour laquelle les planches 5, 6 et 7 sont présentées uniquement à la fin de la phase de test. Cela permet par exemple d'éviter qu'un participant mémorise le chunk A de la planche 7 sans mémoriser le chunk B, ce qui conduirait à une modification de la fréquence subjective de B par rapport à A.

9.5 Participants

En raison de la grande concentration qu'exige cette expérience, les participants ont uniquement été recrutés sur la base du bénévolat. Seulement 6 participants (de 7 à 27 ans) ont été choisis pour cette expérience préliminaire.

Il était spécifié aux participants que l'ordre de rappel des smileys n'avait aucune importance, mais qu'un smiley ne pouvait être rappelé que si ceux qui le précédaient l'étaient également [3]. Afin d'éviter les rappels incertains, les participants étaient informés qu'une erreur dans le rappel entraînait la nullité de la réponse.

9.6 Résultats

Phase d'apprentissage

En ce qui concerne la phase d'apprentissage, aucune erreur n'a été faite par les participants sur le rappel demandé à l'issue de chacune des 30 planches. La durée approximative de la tâche d'apprentissage a été de dix minutes. Après une minute d'attente les participants ont été informés de l'existence de trois smileys et il leur a été demandé de les rapporter. Les trois smileys ont systématiquement été reportés sans erreur. Cela suggère que les chunks correspondant ont été mémorisés correctement en mémoire à long terme. En l'absence de réponse correcte pour l'un des trois smileys, il aurait été nécessaire de retirer le participant de l'expérience.

Phase de test

En ce qui concerne la phase de test, les rappels erronés ont systématiquement été retirés de l'analyse. Un rappel est considéré comme erroné lorsque l'un des symboles rappelé est incorrect ou lorsqu'un smiley est rappelé sans que l'ensemble des smileys qui le précèdent soient complets. Le dernier smiley rappelé est donc le seul à pouvoir être incomplet. Un rappel est également considéré comme erroné si le mot à mémoriser n'est pas rappelé correctement. Quelques fautes d'orthographe sont cependant autorisées.

D'une façon générale, les mots ont été rappelés correctement. En revanche presque un quart des résultats a été retiré pour cause d'erreur dans le rappel. Les erreurs sont apparues indifféremment sur les sept types de planches mais sont beaucoup plus importantes chez certains participants que chez d'autres (du simple au quintuple). Ces erreurs semblent imputables au manque de concentration.

3. Il s'agit pas là d'éviter le cas improbable où un participant choisirait de rappeler un smiley aléatoire plutôt qu'un chunk (smiley de type A, B ou C), ce qui aurait pour conséquence de modifier la fréquence du chunk.

Le taux de rappel selon le type de planche est présenté section suivante. Le rappel des trois chunks A, B et C qui est demandé à la fin de l'expérience conduit à un résultat intéressant. Les trois smileys ont été rapportés correctement, sauf pour deux participants qui ont commis une erreur lors du rappel du smiley C. Le chunk C aurait donc partiellement disparu chez ces deux participants durant la phase de test. Cela permet notamment une meilleure compréhension du taux de rappel obtenu (section suivante) pour les planches de type 2 impliquant le smiley C.

Discussion

Les résultats obtenus durant la phase de rappel sont conformes à ce qui était attendu (Fig. 9.12). Le nombre moyen de rappels semble bien augmenter avec le type de planche considéré. Le taux de rappel est donc plus élevé lorsque les chunks impliqués sont de faible taille de codage. Ces résultats sont encore préliminaires, et vu le faible nombre de participants dont nous disposons, il est nécessaire de répliquer l'expérience à plus grande échelle avant de pouvoir songer à effectuer des tests statistiques plus probants.

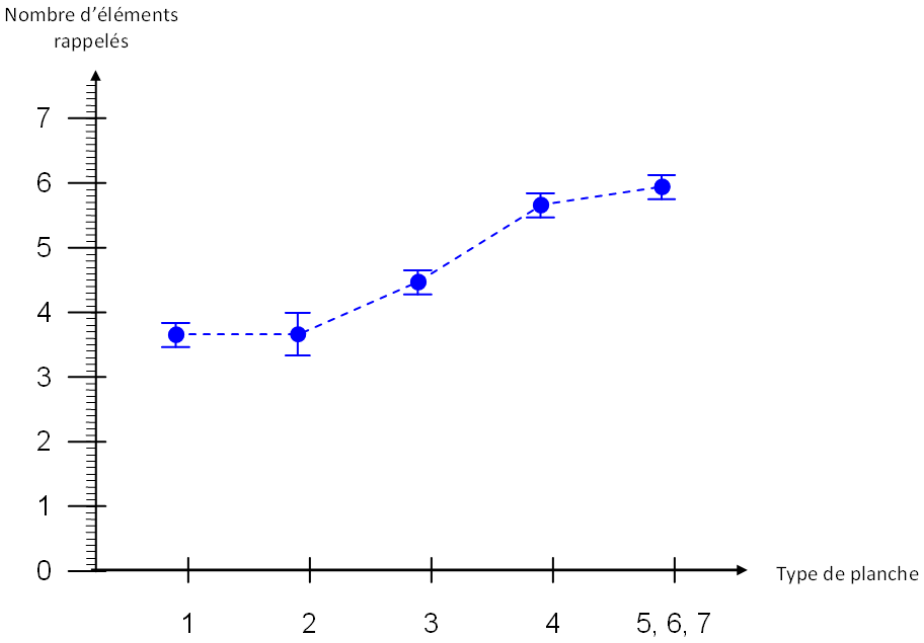


FIGURE 9.12 – Nombre moyen de rappels obtenus pour chacun des types de planche. Les planches 5, 6 et 7 ont été fusionnées en raison de leur faible effectif. Les barres verticales représentent l'erreur standard.

Néanmoins, l'augmentation du taux de rappel avec la fréquence des chunks semble militer en faveur d'une limitation de la mémoire en terme de quantité d'information. S'ils sont insuffisants, les résultats obtenus paraissent encourageants et méritent d'être approfondis.

9.7 Conclusion

Si les résultats obtenus sont confirmés, cela pourrait avoir un impact important en ce qui concerne la validation du MDLChunker. En effet, si les tailles de codage des chunks ont un impact sur le rappel de la mémoire à court terme, alors cela accrédite la thèse selon laquelle les représentations en mémoire à long terme sont basées sur l'utilisation de taille de codage. La nature des représentations en mémoire à court terme est particulièrement intéressante dans la mesure où elle reflète le type des représentations utilisées en mémoire à long terme. Les limitations de la mémoire à court terme fournissent une opportunité d'infirmer ou confirmer les hypothèses qui peuvent être faites concernant la mémoire à long terme.

En supposant que ces résultats se voient confirmés par le passage de l'expérience à une plus grande échelle, la poursuite naturelle de ce travail consisterait à utiliser les prédictions du MDLChunker afin d'estimer la quantité d'information pouvant être stockée en mémoire à court terme. Dans l'expérience présente, la prédiction des chunks possédés par les participants ne nécessite pas réellement l'utilisation du MDLChunker, mais cela peut être vu comme un atout pour des expériences futures. Si nous sommes en mesure d'une part d'évaluer la taille de la mémoire à court terme (perspective du présent chapitre), et d'autre part d'estimer correctement les chunks créés (chapitre 7), tout en intégrant une modélisation du focus attentionnel (chapitre 8), alors cela pourrait donner lieu à la création d'un système relativement polyvalent fondé sur le seul principe de simplicité.

Conclusion et perspectives

Problématique générale

Dans cette thèse nous avons cherché à appliquer le principe de simplicité (Chater & Vitanyi, 2003) au mécanisme de *chunking* à travers un modèle computationnel appelé MDLChunker. L'approche choisie était de concevoir un modèle s'intégrant dans le cadre de l'analyse rationnelle proposée par Anderson et Milson (1989). Dans ce cadre un modèle se doit de posséder deux propriétés essentielles : il doit être à la fois normativement justifié et descriptivement correct (Chater, 1999). Nous avons cherché à satisfaire ces deux critères en détaillant dans une première partie le cadre théorique dans lequel il s'inscrivait, puis en procédant à sa validation sur des données expérimentales recueillies auprès de participants humains.

La volonté de satisfaire ces deux critères nous a poussé à nous intéresser à la fois aux problèmes théoriques dont découlent les principes utilisés, mais également aux problèmes de modélisation et aux difficultés plus pratiques liées à la validation expérimentale. Ce choix nous a conduit à réaliser une thèse largement interdisciplinaire. Bien que le problème central soit la modélisation, nous avons abordé des domaines aussi divers que la complexité algorithmique, le mécanisme de *chunking*, le MDL, la segmentation de mots ou encore le fonctionnement de la mémoire à court terme. Le pendant négatif à cette interdisciplinarité est que les différents domaines abordés n'ont pas tous pu être approfondis avec autant de détail qu'ils auraient pu l'être. Il est notamment probable que certains des protocoles expérimentaux décrits dans cette thèse puissent être améliorés à l'avenir.

Revenons sommairement sur les trois grandes parties de la thèse et les principales conclusions qu'il est possible d'en tirer.

Partie théorique

Les deux objectifs majeurs de cette partie théorique étaient d'une part de fournir une meilleure compréhension des enjeux de la modélisation en général et d'autre part de détailler les outils nécessaires à l'implémentation du principe de simplicité.

En ce qui concerne les enjeux de la modélisation, nous avons essayé d'établir au niveau théorique quelle était la différence entre le surajustement et le surapprentissage ainsi que les raisons pour lesquelles cette différence n'apparaissait pas toujours clairement en pratique. Nous avons également vu que le cadre théorique lié à la complexité algorithmique permettait de définir les propriétés d'un modèle idéal pour un ensemble d'apprentissage donné, mais également d'introduire le *no-free-lunch theorem* et l'impossibilité de posséder un algorithme performant sur toutes les classes de problèmes possibles.

Nous avons vu que le principe de simplicité pouvait être implémenté dans le cadre du MDL et que ce dernier apportait une solution au dilemme biais-variance. Cela nous a permis d'en déduire certaines des propriétés liées au principe de simplicité et notamment que le modèle le plus simple était également le plus prédictif.

Munis de ces résultats théoriques et des outils permettant l'implémentation pratique du principe de simplicité, il a été possible de décrire le fonctionnement du MDLChunker.

Partie modélisation

Le but du premier modèle qui a été proposé était d'appliquer à la lettre les principes de l'analyse rationnelle (Oaksford & Chater, 1998). Le modèle ainsi produit correspondait donc à une application « brute » du principe de simplicité au mécanisme de *chunking*. Ce dernier était en mesure de générer une représentation quasi-optimale des stimuli lorsque ceux-ci étaient issus d'un environnement comportant des régularités fréquentielles et des régularités de type ET.

Le MDLChunker n'intégrant aucune limitation en terme de volume de calcul, un autre modèle a été proposé afin de rendre compte des mêmes principes de façon cognitivement plus plausible. Le but recherché à travers la création de ce second modèle était de montrer que les principes étudiés n'étaient pas liés à une architecture donnée.

Le grand avantage de ces deux modèles est d'une part leur généralité, puisqu'ils n'imposent que peu de contraintes sur le format des stimuli, et d'autre part leur absence de paramètres. Ce dernier point permet de leur accorder un plus grand crédit au niveau théorique puisque les modèles contiennent en eux-même toute l'information nécessaire à leur fonctionnement. De façon pratique, cela signifie que l'exécution et donc les prédictions des modèles dépendent uniquement des stimuli et ne nécessitent aucun ajustement à partir des résultats obtenus par des participants humains.

Partie validation

Au vu du nombre important de modèles de *chunking* existant déjà dans la littérature, il nous a paru important d'exiger de la part du MDLChunker qu'il

soit validé à partir de contraintes très strictes. Nous n'avons pas cherché à reproduire les valeurs d'une statistique particulière réalisée sur les données, ni même à valider le modèle sur sa capacité à reproduire un effet qui soit une conséquence indirecte des représentations possédées en mémoire. Nous avons cherché à valider le modèle sur sa capacité à reproduire les chunks possédés par les participants ainsi que le décours temporel de la création de ces chunks la performance étant alors évaluée à chaque pas de temps et non pas de façon asymptotique.

Nous avons ensuite cherché à appliquer le modèle à des domaines différents afin de tester sa généralité ainsi que sa capacité à supporter l'ajout de contraintes supplémentaires. Une modélisation de l'ordre, de l'oubli et du focus attentionnel ont ainsi été ajoutées afin de rendre compte de la segmentation de flux de syllabes en mots. Une tentative d'application des principes du MDLChunker à la modélisation de la mémoire à court terme a également été proposée. Bien qu'encore en cours d'exploration, cette piste pourrait permettre de valider ou d'invalider le type de représentation utilisé par le MDLChunker. À terme, une modélisation de la mémoire à court terme n'est pas non plus exclue et pourrait permettre d'appliquer le MDLChunker à un panel de tâches plus large.

Domaine d'application

À l'origine, le MDLChunker avait été conçu dans le but d'être intégré à un environnement d'apprentissage. L'objectif aurait été de pouvoir prédire les représentations qu'un élève était susceptible de créer face à un ensemble de stimuli donné (l'apprentissage d'équations par exemple). L'intérêt d'un tel système aurait été de pouvoir tester très rapidement différentes séquences d'apprentissage afin de déterminer la meilleure. Bien que l'apprentissage d'équations ne se résume pas uniquement à la création de chunks, la difficulté majeure qui est apparue est le problème de la modélisation des connaissances a priori du domaine. En d'autres termes, l'impossibilité de connaître avec précision les chunks déjà possédés empêche de savoir comment un stimulus donné sera perçu et donc quelles représentations sont susceptibles d'être créées.

Bien qu'il soit possible à très long terme que ce type de modèle (MDLChunker ou MDLChunker-approché) puisse être appliqué dans un contexte écologique, cet objectif a pour le moment été abandonné au profit d'une étude des mécanismes cognitifs impliqués dans la création de nouveaux concepts. Les perspectives que nous envisageons pour le moment sont uniquement du domaine de la recherche.

Perspectives

Il est possible de regrouper les perspectives en deux catégories : les perspectives en terme de modélisation et les perspectives en terme de validation. À moyen terme, il pourrait être envisageable de chercher à accroître l'expressivité du langage de représentation du MDLChunker. Ce dernier contient actuellement la négation (NON) et la conjonction (ET). Il serait en théorie possible d'ajouter

la disjonction (OU), mais outre les difficultés techniques que cela comporte, cet ajout pourrait poser des problèmes en termes de plausibilité cognitive (Feldman, 2003). Le travail restant à réaliser sur la conjonction nous semble suffisamment important pour que l'étude de la disjonction soit momentanément mise de côté.

Les perspectives immédiates concernent prioritairement la validation de la conjonction. Deux pistes sont en ce moment à l'étude : étudier la création de chunks en fonction du niveau de bruit, et étudier l'influence des chunks déjà créés sur la perception des stimuli. La première piste consiste simplement à faire varier le niveau de bruit afin de comparer la sensibilité des humains et du modèle. La seconde piste est étudiée en modifiant la grammaire au cours de l'expérience afin de voir si l'influence des anciens chunks sur la création des nouveaux chunks reste prédictible par le MDLChunker. Selon les résultats obtenus pour cette seconde expérience, il est possible qu'il soit nécessaire de dégrader les performances du mécanisme de factorisation en diminuant l'exploration effectuée par l'algorithme A*.

La dernière piste de recherche importante concerne l'obtention de résultats probants pour l'expérience relative à la mémoire à court terme. Si la poursuite sur cette voie s'avérait fructueuse, cela pourrait permettre de donner un plus grand crédit à l'utilisation du principe de simplicité dans le cadre de la modélisation cognitive. Au-delà de cet aspect « théorique » de la validation, cela donnerait également la possibilité au MDLChunker de modéliser des phénomènes impliquant la mémoire à court terme. Son champ d'application s'en trouverait alors nettement accru sans que sa complexité soit augmentée.

Références

Alvarez, G., & Cavanagh, P. (2004). The capacity of visual short-term memory is set both by visual information load and by number of objects. *Psychological Science*, *15*(2), 106-111.

Anderson, J. (1989). A rational analysis of human memory. In H. L. Roediger & F. I. Craik (Eds.), *Varieties of memory and consciousness* (p. 195-210). L. Erlbaum Associates.

Anderson, J. (1990). *The adaptive character of thought*. L. Erlbaum Associates.

Anderson, J., & Bellezza, F. (1993). *Rules of the mind*. L. Erlbaum Associates.

Anderson, J., & Milson, R. (1989). Human memory : An adaptive perspective. *Psychological Review*, *96*(4), 703-719.

Argamon, S., Akiva, N., Amir, A., & Kapah, O. (2004). Efficient unsupervised recursive word segmentation using minimum description length. In *Proceedings of the 20th International Conference on Computational Linguistics (Coling04)*.

Baddeley, A., & Hitch, G. (1974). Working memory. In *The psychology of learning and motivation* (p. 47-89). New York : GA, Bower.

Berry, D., & Broadbent, D. (1988). Interactive tasks and the implicit-explicit distinction. *British Journal of Psychology*, *79*(2), 251-272.

Borg, I., & Groenen, P. (2005). *Modern multidimensional scaling : Theory and applications*. Springer Verlag.

Brady, T., Konkle, T., & Alvarez, G. (2008). Efficient coding in visual Short-Term memory : Evidence for an Information-Limited capacity. In *Proceedings of the 30th annual conference of the cognitive science society* (p. 887-892).

Breiman, L., Friedman, J., Olshen, R., & Stone, C. (1984). Classification and regression trees. *Wadsworth Int*, *358*, 43-49.

Brent, M., & Cartwright, T. (1996). Distributional regularity and phonotactic constraints are useful for segmentation. *Cognition*, *61*(1-2), 93-125.

Chaitin, G. (1966). On the length of programs for computing finite binary sequences. *Journal of the ACM (JACM)*, *13*(4), 547-569.

Chaitin, G. (1987). *Information, randomness & incompleteness : papers on algorithmic information theory*. World Scientific Pub Co Inc.

Chaitin, G. (2002). On the intelligibility of the universe and the notions of simplicity, complexity and irreducibility. In *Philosophy congress on limits and transcending limits*. Bonn.

Chaitin, G. (2006). *Meta math ! : the quest for Omega*. Vintage Books.

Chater, N. (1996). Reconciling simplicity and likelihood principles in perceptual

organization. *Psychological Review*, *103*, 566-581.

Chater, N. (1999). The search for simplicity : A fundamental cognitive principle ? *The Quarterly Journal of Experimental Psychology A*, *52*, 273-302.

Chater, N., & Oaksford, M. (1999). Ten years of the rational analysis of cognition. *Trends in Cognitive Sciences*, *3*(2), 57-65.

Chater, N., & Vitanyi, P. (2003). Simplicity : a unifying principle in cognitive science ? *Trends in Cognitive Sciences*, *7*(1), 19-22.

Cleeremans, A., Destrebecqz, A., & Boyer, M. (1998). Implicit learning : News from the front. *Trends in cognitive sciences*, *2*(10), 406-416.

Cleeremans, A., & Jiménez, L. (2002). Implicit learning and consciousness : A graded, dynamic perspective. In R. M. French & A. Cleeremans (Eds.), *Implicit learning and consciousness : An empirical, computational and philosophical consensus in the making* (p. 1-40). Psychology Press.

Cosmides, L., & Tooby, J. (1995). Beyond intuition and instinct blindness : Toward an evolutionarily rigorous cognitive science. In J. Mehler & S. Franck (Eds.), *Cognition on cognition* (p. 69-105). Elsevier Science Publishers.

Cowan, N. (2005). *Working memory capacity*. Psychology Press.

Cowan, N., Chen, Z., & Rouder, J. (2004). Constant capacity in an immediate serial-recall task. *Psychological Science Cambridge*, *15*, 634-640.

Cowan, N., Morey, C., Chen, Z., Gilchrist, A., & Saults, J. (2008). Theory and measurement of working memory capacity limits. In *The psychology of learning and motivation : Advances in research and theory* (p. 49). Elsevier.

Croker, S., Pine, J., & Gobet, F. (2000). Modelling optional infinitive phenomena : A computational account of tense optionality in children's speech. In *Proceedings of the 3rd International Conference on Cognitive Modeling* (p. 78-85). Universal Press.

Damasio, R. (1995). *L'erreur de Descartes*. Odile Jacob.

de Groot, A., Gobet, F., & Jongman, R. (1996). *Perception and memory in chess : Studies in the heuristics of the professional eye*. Van Gorcum.

Dowman, M. (soumis). Minimum description length as a solution to the problem of generalization in syntactic theory.

Dulany, D., Carlson, R., & Dewey, G. (1984). A case of syntactical learning and judgment : How conscious and how abstract ? *Journal of Experimental Psychology : General*, *113*(4), 541-555.

Feigenbaum, E., & Simon, H. (1984). EPAM-like models of recognition and learning. *Cognitive Science*, *8*(4), 305-336.

Feldman, J. (2003). the simplicity principle in human concept learning. *Current Directions in Psychological Science*, *12*(6), 227-232.

Fiser, J., & Aslin, R. (2001). Unsupervised statistical learning of Higher-Order spatial structures from visual scenes. *Psychological Science*, *12*(6), 499-504.

Fodor, J. (1983). *The modularity of mind*. MIT press Cambridge, MA.

French, R., Mareschal, D., Mermillod, M., & Quinn, P. (2004). The role of bottom-up processing in perceptual categorization by 3-to 4-month-old infants : Simulations and data. *Journal of Experimental Psychology : General*, *133*(3), 382-397.

Gacs, P., Tromp, J., & Vitanyi, P. (2001). Algorithmic statistics. *IEEE Transactions on Information Theory*, *47*(6), 2443-2463.

Giroux, I., & Rey, A. (2009). Lexical and sublexical units in speech perception. *Cognitive Science*, *33*, 260-272.

Gobet, F. (1993). A computer model of chess memory. In *Proceedings of the Fifteenth Annual Conference of the Cognitive Science Society* (p. 463-468). L. Erlbaum Associates.

Gobet, F. (1996). Discrimination nets, production systems and semantic networks : Elements of a unified framework. In *Proceedings of the 1996 International Conference on Learning Sciences* (p. 398-403). International Society of the Learning Sciences.

Gobet, F. (2001). Réseaux de discrimination en psychologie : L'exemple de CHREST. *Swiss Journal of Psychology*, *60*(4), 264-277.

Gobet, F., & Clarkson, G. (2004). Chunks in expert memory : Evidence for the magical number four, or is it two ? *Memory*, *12*(6), 732-747.

Gobet, F., Lane, P., Croker, S., Cheng, P., Jones, G., Oliver, I., et al. (2001). Chunking mechanisms in human learning. *Trends in Cognitive Sciences*, *5*(6), 236-243.

Goldsmith, J. (2001). Unsupervised learning of the morphology of a natural language. *Computational Linguistics*, *27*(2), 153-198.

Grunwald, P., Myung, I., & Pitt, M. (2005). *Advances in Minimum Description Length : Theory and applications*. The MIT Press.

Hérault, J. (2009). *Vision : Signals, Images and Neural Networks* (Allan Murray, Ed.). World Scientific Publishers.

Huffman, D. (1952). A method for the construction of minimum-redundancy codes. In *Proceedings of the inst. radio engineers* (Vol. 40, p. 1098-1101).

Jaynes, E. (1982). On the rationale of maximum-entropy methods. *Proceedings of the IEEE*, *70*(9), 939-952.

Jones, G., Gobet, F., & Pine, J. (2000). A process model of children's early verb use. In *Proceedings of the 22nd Meeting of the Cognitive Science Society* (p. 723-728). L. Erlbaum Associates.

Kolmogorov, A. (1968). Three approaches to the quantitative definition of information. *International Journal of Computer Mathematics*, *2*(1), 157-168.

Kolmogorov, A. (1974). Complexity of algorithms and objective definition of randomness. *Uspekhi Mat. Nauk*, *29*(4), 155.

Kullback, S., & Leibler, R. (1951). On information and sufficiency. *The Annals of Mathematical Statistics*, *22*, 79-86.

Lane, P., & Gobet, F. (2003). Developing reproducible and comprehensible computational models. *Artificial Intelligence*, *144*(1-2), 251-263.

Leung-Yan-Cheong, S., & Cover, T. (1978). Some equivalences between Shannon entropy and Kolmogorov complexity. *Information Theory, IEEE Transactions on*, *24*(3), 331-338.

Levin, L. (1974). Laws of information conservation (nongrowth) and aspects of the foundation of probability theory. *Problemy Peredachi Informatsii*, *10*(3), 30-35.

Li, M., & Vitanyi, P. (1997). *An introduction to Kolmogorov complexity and its applications* (Springer éd.). Springer.

MacWhinney, B., & Snow, C. (1985). The Child language data exchange system. *Journal of Child Language*, *12*(2), 271.

Martin-Löf, P. (1966). The definition of random sequences. *Information and Control*, *9*(6), 602-619.

Miller, G. (1956). The magical number seven, plus or minus two : Some limits on our capacity to process information. *Psychological Review*, *63*(2), 81-97.

Miller, G. (1958). Free recall of redundant strings of letters. *Journal of Experimental Psychology*, *56*, 485-491.

Oaksford, M., & Chater, N. (1993). Reasoning theories and bounded rationality. In K. I. Manktelow & D. E. Over (Eds.), *Rationality : Psychological and philosophical perspectives* (p. 31-60). International Library of Psychology.

Oaksford, M., & Chater, N. (1998). *Rational models of cognition.* Oxford University Press.

Ohlsson, S. (2008). Computational models of skill acquisition. In R. Sun (Ed.), *The Cambridge Handbook of Computational Psychology* (p. 359-395). The MIT Press.

Pearl, J. (1984). *Heuristics : intelligent search strategies for computer problem solving.* Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA.

Perruchet, P., & Gallego, J. (2006). Do dogs know related rates rather than optimization ? *College Mathematics Journal*, *37*(1), 16.

Perruchet, P., & Pacteau, C. (1990). Synthetic grammar learning : Implicit rule abstraction or explicit fragmentary knowledge. *Journal of Experimental Psychology : General*, *119*(3), 264-275.

Perruchet, P., & Pacton, S. (2006). Implicit learning and statistical learning : one phenomenon, two approaches. *Trends in Cognitive Sciences*, *10*(5), 233-238.

Perruchet, P., & Vinter, A. (1998). PARSER : a model for word segmentation. *Journal of Memory and Language*, *39*(2), 246-263.

Perruchet, P., Vinter, A., Pacteau, C., & Gallego, J. (2002). The formation of structurally relevant units in artificial grammar learning. *The Quarterly Journal of Experimental Psychology Section A*, *55*(2), 485-503.

Perruchet, P., Vinter, A., & Pacton, S. (2007). La conscience auto-organisatrice : Une alternative au modèle dominant de la psychologie cognitive. *Education et Didactique*, *1*(3), 7-34.

Poland, J. (2004). A coding theorem for enumerable output machines. *Information Processing Letters*, *91*(4), 157-161.

Pothos, E., & Wolff, J. (2006). The Simplicity and Power model for inductive inference. *Artificial Intelligence Review*, *26*(3), 211-225.

Rado, T. (1962). On non-computable functions. *Bell System Technical Journal*, *41*(3), 877-884.

Reber, A. (1967). Implicit learning of artificial grammars. *Journal of Verbal Learning & Verbal Behavior. Vol*, *6*(6), 855-863.

Redington, M., & Chater, N. (1996). Transfer in artificial grammar learning : A reevaluation. *Journal of experimental psychology. General*, *125*, 123-138.

Rissanen, J. (1978). Modeling by shortest data description. *Automatica*, *14*(5), 465-471.

Robinet, V., & Lemaire, B. (2009). MDLChunker : a MDL-based model of word segmentation. In N. T. . H. van Rijn (Ed.), *Proceedings of the 31th Annual Conference of the Cognitive Science Society (CogSci 2009).* Amsterdam, Netherland : Cognitive Science Society.

Saffran, J., Newport, E., & Aslin, R. (1996). Word segmentation : The role of distributional cues. *Journal of Memory and Language*, *35*(4), 606-621.

Saffran, J., Newport, E., Aslin, R., Tunick, R., & Barrueco, S. (1997). Incidental

language learning : Listening and learning out of the corner of your ear. *Psychological Science*, *8*(2), 101-105.

Schmidhuber, J. (2002). Hierarchies of generalized Kolmogorov complexities and nonenumerable universal measures computable in the limit. *International Journal of Foundations of Computer Science*, *13*(4), 587-612.

Servan-Schreiber, E., & Anderson, J. (1990). Learning artificial grammars with competitive chunking. *Journal of Experimental Psychology : Learning, Memory, and Cognition*, *16*(4), 592-608.

Shannon, C. (1948). A mathematical theory of communication. *Bell System Tech. Journal.*, *27*(3), 379-423.

Shannon, C., & Weaver, W. (1949). *Mathematical theory of communication.* University of Illinois Press.

Solomonoff, R. (1960). A preliminary report on a general theory of inductive inference. *Zator Co*, *Report V-131*, 1.

Solomonoff, R. (1964). A formal theory of inductive inference. parts I and II. *Information and Control*, *7*(2), 224-254.

Stanley, W., Mathews, R., Buss, R., & Kotler-Cope, S. (1989). Insight without awareness : On the interaction of verbalization, instruction and practice in a simulated process control task. *The Quarterly Journal of Experimental Psychology Section A*, *41*(3), 553-577.

Sun, R. (2004). Desiderata for cognitive architectures. *Philosophical Psychology*, *17*(3), 341-373.

Sun, R., Merrill, E., & Peterson, T. (2001). From implicit skills to explicit knowledge : A bottom-up model of skill learning. *Cognitive Science : A Multidisciplinary Journal*, *25*(2), 203-244.

Swingley, D. (2005). Statistical clustering and the contents of the infant vocabulary. *Cognitive Psychology*, *50*(1), 86-132.

Tulving, E. (1962). Subjective organization in free recall of unrelated words. *Psychological Review*, *69*, 344.

Turing, A. (1936). On computable numbers : With an application to the entscheidungsproblem. *Proceeding of the London Mathematical Society*, *2*, 230-265.

van Rooij, I. (2008). The tractable cognition thesis. *Cognitive Science*, *32*(6), 939-984.

Vapnik, V., & Chervonenkis, A. (1971). On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability and its Applications*, *16*, 264-280.

Vereshchagin, N., & Vitanyi, P. (2002). Kolmogorov's structure functions with an application to the foundations of model selection. In *Proc. 47th IEEE symp. found. comput. Sci.(FOCS02).*

Vereshchagin, N., & Vitanyi, P. (2004). Kolmogorov's structure functions and model selection. *IEEE Transactions on Information Theory*, *50*(12), 3265-3290.

Vitanyi, P. (2005). Algorithmic statistics and Kolmogorov's structure function. In *Advances in Minimum Description Length : Theory and Applications* (p. 151-174). The MIT Press.

Vitanyi, P., & Li, M. (2000). Minimum description length induction, bayesianism, and Kolmogorov complexity. *Information Theory, IEEE Transactions on*, *46*(2), 446-464.

Vogel, E., Woodman, G., & Luck, S. (2001). Storage of features, conjunctions,

and objects in visual working memory. *Journal of Experimental Psychology. Human Perception and Performance*, *27*(1), 92-114.

Wallace, C., & Boulton, D. (1968). An information measure for classification. *The Computer Journal*, *11*(2), 185.

Wertheimer, M. (1922). Untersuchungen zur lehre von der gestalt. *Psychological Research*, *1*(1), 47-58.

Whitley, D., & Watson, J. (2005). Complexity theory and the no free lunch theorem. In E. K. Burke & G. Kendall (Eds.), *Search methodologies : Introductory tutorials in optimization and decision support techniques* (p. 317-339). Springer.

Wolff, J. (1999). Probabilistic reasoning as information compression by multiple alignment, unification and search : an introduction and overview. *Journal of Universal Computer Science*, *5*(7), 418-462.

Wolff, J. (2000). Syntax, parsing and production of natural language in a framework of information compression by multiple alignment, unification and search. *Journal of Universal Computer Science*, *6*(8), 781-829.

Wolff, J. (2003). Information compression by multiple alignment, unification and search as a unifying principle in computing and cognition. *Artificial Intelligence Review*, *19*(3), 193-230.

Wolff, J. (2006). Medical diagnosis as pattern recognition in a framework of information compression by multiple alignment, unification and search. *Decision Support Systems*, *42*(2), 608-625.

Wolpert, D., & Macready, W. (1997). No free lunch theorems for optimization. *Evolutionary Computation, IEEE Transactions on Evolutionary Computation*, *1*(1), 67-82.

Zvonkin, A., & Levin, L. (1970). The complexity of finite objects and the development of the concepts of information and randomness by means of the theory of algorithms. *Russian Mathematical Surveys*, *25*(6), 83-124.

Annexe A

Explications additionnelles

A.1 Définitions liées à la complexité algorithmique

Cette annexe correspond aux définitions utilisées dans le chapitre 2. Elle définit de façon rigoureuse ce qu'est une machine de Turing ainsi qu'une machine de Turing Universelle. Elle présente également les codages autodélimitants et leurs propriétés.

A.1.1 Machine de Turing

La notion de « Machine de Turing » a été introduite par A. Turing afin de définir la notion de calculabilité. Une machine de Turing peut être vue comme une généralisation du fonctionnement d'un ordinateur. C'est un automate à états finis composé d'un ruban de longueur infinie sur lequel sont inscrits des caractères issus d'un alphabet fini. Un certain nombre d'actions peuvent être effectuées, comme se déplacer à gauche et à droite sur le ruban ou encore lire et écrire un caractère. À chaque itération, une table d'actions permet de déduire de l'état courant et du caractère lu, quelle sera l'action à effectuer ainsi que l'état suivant (Fig. A.1 page suivante).

De façon formelle, une machine de Turing est définie par un septuplet $(Q, q0, F, \Gamma, \Sigma, B, \delta)$.
- Q est l'ensemble (fini) des états qi que peut prendre la machine.
- $q0$ est le nom donné à l'état initial ($q0 \in Q$).
- F est l'ensemble des états finaux ($F \subseteq Q$).
- Γ est l'alphabet (fini) des symboles du ruban.
- Σ est l'alphabet des symboles d'entrée ($\Sigma \subseteq \Gamma \setminus \{B\}$).
- B est le symbole blanc contenu dans les cases du ruban non encore écrites ($B \in \Gamma \setminus \Sigma$).
- δ est la fonction de transition (table des actions) $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\leftarrow, \rightarrow\}$.

La machine commence dans l'état $q0$ et s'arrête lorsqu'elle se trouve dans l'un des états finaux de F. Le passage de l'état courant à l'état suivant se fait grâce

FIGURE A.1 – Exemple d'action. Si l'état est $q8$ et le caractère lu est 0, alors écrire 1, se décaler à droite et passer dans l'état $q5$. La valeur de la fonction de transition δ correspondant à cette entrée dans la table des actions est : $\delta(q8, 0) = (q5, 1, \rightarrow)$.

à la fonction de transition δ.

Thèse de Church-Turing

La thèse de Church-Turing définit la notion de calculabilité en identifiant les fonctions effectivement calculables aux fonctions calculables par une machine de Turing (Turing, 1936). Toute fonction calculable à l'aide d'une procédure déterministe de type algorithme, peut l'être par une machine de Turing. Cela signifie que s'il est possible de trouver une machine M calculant plus vite ou utilisant moins de mémoire qu'une machine de Turing T, l'ensemble des fonctions calculables par M n'excédera jamais l'ensemble des fonctions calculables par T.

Il existe des fonctions non-calculables. Comme expliqué section 2.3 du chapitre 2, la complexité algorithmique $K(x)$ d'une chaîne binaire x est un nombre parfaitement défini mais qui n'est pas calculable. C'est également le cas du Oméga de Chaitin (Chaitin, 2006) ou encore du problème du castor affairé [1] (Rado, 1962).

Machine de Turing Universelle

Turing a montré (Turing, 1936) qu'il existait une machine Turing U qualifiée d'universelle, capable de reproduire le fonctionnement de n'importe quelle machine de Turing T. La sortie d'une machine de Turing T dépend uniquement de son entrée e et du comportement défini par la table d'actions δ. Il est donc possible de reproduire le comportement de n'importe quelle machine de Turing en passant la table d'action δ en paramètre. Un programme p pour la machine U est une concaténation de la table d'action δ et de l'entrée e ($p = \delta.e$).

$$\forall T, \exists \delta, \forall e, U(\delta.e) = T(e) \tag{A.1}$$

1. Il s'agit du problème consistant à déterminer le nombre maximal de symboles qu'une machine de Turing à n états peut écrire sur un ruban vide avant de s'arrêter. De même que pour $K(x)$) des bornes inférieures peuvent être trouvées empiriquement, mais trouver une borne supérieure reviendrait à résoudre le problème de l'arrêt qui est indécidable.

où δ dépend uniquement de la machine de Turing T considérée. L'universalité de la machine de Turing ainsi construite vient de la généralité de son comportement (table d'actions) qui est lui-même un paramètre d'entrée. Fixer la table d'actions δ revient à spécifier l'indice i identifiant une machine de Turing T_i de façon unique.

A.1.2 Codage autodélimitant

La notion de codage autodélimitant est indispensable pour pouvoir définir une distribution de probabilité sur l'espace des programmes.

Décodage non-ambigu

Un codage est autodélimitant si la simple concaténation des mots du code peut être décodée de façon unique. L'intérêt d'un tel codage est qu'il ne nécessite pas l'utilisation de séparateurs. Le codage $\{0, 10, 11\}$ est autodélimitant. Il y a unicité du décodage de la chaîne 10100110 sous forme 10.10.0.11.0. Ce n'est pas le cas avec le codage $\{0, 1, 10\}$ qui n'est pas autodélimitant : les deux premiers bits de la chaîne précédente pouvant être décodés 10 ou 1.0.

Construction d'un codage autodélimitant

Il existe de nombreuses méthodes pour construire un codage autodélimitant. La plus célèbre est sans doute le codage de Huffman (Huffman, 1952). Le but des algorithmes de recherche d'un codage optimal est d'associer un code court aux mots fréquents [2]. Pour être autodélimitant, aucun mot du code ne doit être préfixe d'un autre. La construction d'un tel codage peut se faire à l'aide d'un arbre (Fig. A.2 page suivante).

Mesure de probabilité

Dans la section 2.2.5 page 26, nous avons utilisé sans le détailler un résultat important : une mesure de probabilité peut être définie sur l'ensemble des chaînes binaires autodélimitantes, ce qui n'est pas le cas sur l'ensemble des chaînes binaires. Pour tout codage autodélimitant PA on a

$$\sum_{p \in PA} 2^{-|p|} = 1 \tag{A.2}$$

où $|p|$ est la longueur du code associé à p.

2. L'optimum est atteint si l'on est capable d'associer un code de longueur $log_2\left(\frac{1}{P(p)}\right)$ à tout mot p de fréquence $P(p)$. Cet optimum est rarement atteint en pratique. Le but de ces algorithmes est de s'en approcher le plus possible.

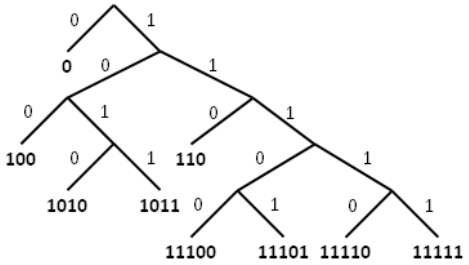


FIGURE A.2 – Le codage $\{0, 100, 110, 1010, 1011, 11100, 11101, 11110, 11111\}$ est autodélimitant car chaque mot du code est une feuille de l'arbre : aucun mot du code ne peut donc être préfixe d'un autre. Ajouter par exemple 111 qui n'est pas une feuille, supprime cette propriété car 111 est alors préfixe de tous les noeuds du sous-arbre dont il est le père.

Cette propriété est aisément vérifiable pour le codage autodélimitant trivial $\{0, 10, 110, 1110, 11110, 111110, \ldots\}$ car $\sum_{k\geq 1} 2^{-k} = 1$. La figure A.3 montre pourquoi cette propriété reste vraie pour tous les codages autodélimitants.

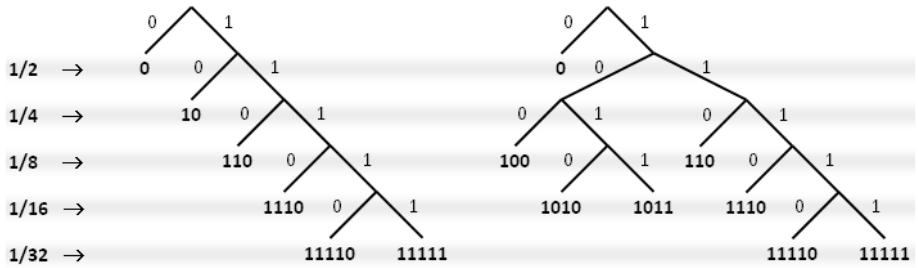


FIGURE A.3 – Pour le codage autodélimitant trivial (gauche) on a bien $\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \frac{1}{32} + \frac{1}{32} = 1$. Tout autre codage autodélimitant peut être vu comme une modification de ce codage trivial. Chaque feuille peut être remplacée par un sous-arbre dont la somme des probabilités des feuilles est nécessairement égale à la probabilité de la feuille développée. Dans le codage de droite $P(10) = \frac{1}{4}$ est remplacé par un sous-arbre de même probabilité : $P(100) + P(1010) + P(1011) = \frac{1}{8} + \frac{1}{16} + \frac{1}{16} = \frac{1}{4}$.

D'une façon générale, on a toujours

$$\sum_{p \epsilon PA} 2^{-|p|} \leq 1 \tag{A.3}$$

C'est l'inégalité de Kraft, avec égalité dans le cas où toutes les feuilles de l'arbre sont utilisées. Le codage est alors dit « autodélimitant complet » . Dans le cas d'un codage qui ne vérifie pas cette propriété et sous la condition qu'aucun élément ne se voit associer de probabilité nulle, la somme dans l'équation A.2 page précédente est infinie.

A.2 Comparaison détaillée entre CC et le MDL-Chunker

Familiarité

Dans CC, la familiarité d'un stimulus est un indicateur qui dépend uniquement du nombre de chunks dans la représentation factorisée du stimulus. C'est un nombre réel compris entre 0 (non-familier) et 1 (familier), défini comme étant $e^{1-nbChunks}$. L'importance donnée à tous les chunks est identique.

Dans le MDLChunker, la quantité équivalente est la taille de codage du stimulus, que l'on pourrait définir comme étant l'absence de familiarité du stimulus pour laquelle l'importance donnée à chaque chunks dépend de sa fréquence dans les stimuli passés.

Utilité d'un chunk

Pour calculer l'utilité d'un chunk (*strength*), CC intègre explicitement un facteur d'oubli. L'utilité d'un chunk est définie comme le nombre d'apparitions du chunk, chacune étant pondérée par un facteur de décroissance dépendant du temps écoulé depuis l'apparition :

$$utilite = \sum_i T_i^{-d} \tag{A.4}$$

T_i représente le temps écoulé depuis la $i$ème apparition du chunk. Le taux d'oubli est fixé par le paramètre réel d compris entre 0 et 1.

Par opposition, le MDLChunker ne modélise pas explicitement l'oubli. L'utilité d'un chunk est fixée par sa taille de codage et dépend donc uniquement de sa fréquence empirique. Un chunk qui n'apparaît plus va voir son utilité diminuer car sa fréquence relative diminue, mais il ne s'agit pas une modélisation de l'oubli à proprement parler. Pour information, une modélisation possible de l'oubli est proposée au chapitre 8.

Factorisation

Pour CC, la factorisation se fait de façon probabiliste. Chaque chunk se voit associer une probabilité dépendant de l'utilité de ses fils directs :

$$P(chunk) = \frac{1 - e^{-c \times support}}{1 + e^{-c \times support}} \tag{A.5}$$

où $c > 0$ est un paramètre fixant la concavité de la courbe (Fig. A.4 page suivante). Le support représente la moyenne des utilités des fils directs du chunk. Les chunks-canoniques n'ayant pas de fils, la valeur de leur support est fixée à une valeur arbitrairement élevée (10). Un chunk se trouve donc d'autant plus

souvent sélectionné pour factoriser un stimulus que l'utilité de ses fils directs est élevée. Dans le cas problématique où l'intersection entre deux chunks n'est pas vide (par exemple ABC et BCD), le chunk retenu est celui de plus forte utilité. Pour résumer, un chunk ne peut être utilisé pour factoriser un stimulus que si son utilité ainsi que celle de toute sa descendance est élevée [3].

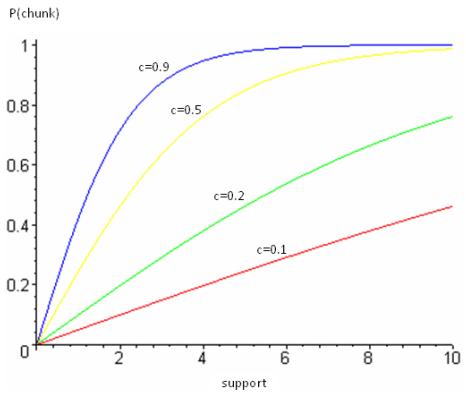


FIGURE A.4 – Influence du paramètre c sur la fonction reliant la probabilité d'un chunk à la valeur de son support.

Dans le MDLChunker, la factorisation est déterministe : c'est celle qui minimise la taille de codage du stimulus courant. Le résultat de la factorisation dépend uniquement de la taille de codage (donc de la fréquence) des chunks impliqués et non pas de leurs descendants. C'est notamment ce qui permet de reproduire le *vanishing sub-chunk effect* (Robinet & Lemaire, 2009) décrit au chapitre 8. L'algorithme A* utilisé pour la factorisation permet de résoudre naturellement le cas problématique où l'intersection entre deux chunks n'est pas vide.

Optimisation

Pour CC, l'étape de création des chunks utilise le même mécanisme que la factorisation et se limite aux chunks binaires. Si le stimulus perçu se factorise en (AB) C (DEF) (GH), CC envisage la création des trois chunks binaires ((AB)C), (C(DEF)) et ((DEF)(GH)), regroupant ainsi les chunks adjacents. Le nombre de chunks créés à chaque itération est arbitrairement fixé à 1. Le chunk effectivement créé est celui qui a la plus forte probabilité (équation A.5 page précédente), c'est-à-dire celui qui a le plus grand support. Dans le cas où il existe déjà, son utilité est renforcée.

Les chunks créés par le MDLChunker sont également binaires (pour les mêmes raisons de dimension de l'espace de recherche). Selon que le stimulus est ordonné ou non, tous les chunks binaires sont envisagés, ou seulement les chunks composés de symboles adjacents. Autant de chunks que nécessaire sont créés à chaque

3. En réalité, il suffit que son utilité et celle de ses fils directs soit élevée. Mais comme l'utilité de ses fils ne peut être élevée que s'ils sont fréquemment utilisés, et qu'ils ne seront utilisés que si leurs fils directs le sont également, il faut par récurrence que toute la descendance du chunk considéré possède une utilité élevée.

itération, c'est-à-dire tant que la taille de codage de la définition du chunk est inférieure au gain apporté par son utilisation.

A.3 Comparaison détaillée entre PARSER et le MDLChunker

Focus attentionnel

Le rôle de PARSER étant la segmentation de mots, l'entrée est un flux de syllabes continu. PARSER intègre donc une modélisation du focus attentionnel permettant de segmenter ce flux en unités appelées « percepts » . La longueur de chaque percept est tirée aléatoirement de 1 à 3 chunks. Le nombre de syllabes d'un percept dépend donc à la fois de la taille du focus attentionnel (1, 2 ou 3 chunks) et des chunks possédés par PARSER à cette phase de l'apprentissage. C'est le phénomène de *perception shaping* (Perruchet et al., 2002) : les différentes perceptions permettent la création de représentations (chunks) qui en retour influencent les perceptions futures.

Le format des données d'entrée du MDLChunker n'est pas un flux continu mais en ensemble de stimuli. Dans cette situation, le focus attentionnel est supposé porter sur l'intégralité de chaque stimulus. Le notion de *perception shaping* correspond dans ce cas à la minimisation de la taille de codage du focus attentionnel.

Familiarité

La notion de familiarité d'un stimulus est binaire dans le cas de PARSER. Si le stimulus correspond à un chunk connu dont l'utilité est supérieure au seuil de familiarité (*shaping threshold*), alors le stimulus est considéré comme familier. Dans le cas contraire, le stimulus est considéré comme non-familier.

Pour le MDLChunker, l'indicateur de familiarité est un nombre réel qui dépend de la taille de codage du stimulus. Un stimulus est d'autant plus familier que sa taille de codage est faible, donc que sa fréquence est élevée.

Utilité d'un chunk

Dans PARSER, chaque fois qu'un chunk est perçu (contenu dans le stimulus courant), son utilité se voit augmentée d'une quantité donnée (paramètre fixé à 0.5). PARSER intègre également une modélisation de l'oubli (oubli linéaire). À chaque nouvelle itération tous les chunks voient leur utilité décroître d'une quantité donnée (paramètre d'oubli fixé à 0.05). Lorsque l'utilité d'un chunk devient nulle, le chunk disparaît.

Dans le MDLChunker, la perception d'un nouveau chunk accroît naturellement

sa fréquence donc son utilité (sa taille de codage diminue). L'oubli n'est pas modélisé dans la version basique du MDLChunker (voir chapitre 8 pour une modélisation de l'oubli), cependant un chunk qui n'apparaît plus va voir sa fréquence, donc son utilité, diminuée.

Factorisation

Lorsqu'il y a chevauchement entre plusieurs chunks, le découpage d'un stimulus donné est susceptible de ne pas être unique. PARSER lève cette ambiguïté en choisissant systématiquement le chunk le plus long. L'éventualité consistant à choisir le chunk de plus forte utilité a été rejetée car elle conduisait à de moins bons résultats. Les chunks qui n'ont pas été choisis et qui possèdent une partie commune avec le chunk choisi voient leur utilité diminuée d'une quantité donnée (paramètre d'interférence fixé à 0.005). Cet effet d'interférence a pour but de pénaliser les chunks de longueur faible qui sont contenus dans un chunk plus grand apparaissant fréquemment. Par exemple, si les chunks *patubi*, *pa* et *tubi* existent, le stimulus *tupatubibu* sera factorisé en *tu patubi bu* et les chunks *pa* et *tubi* subiront l'effet d'interférence du chunk *patubi*.

Le MDLChunker utilise la factorisation qui minimise la taille de codage du stimulus. Un chunk plus long comme *patubi* est choisi uniquement si sa taille de codage est inférieure à la somme des tailles de codage de *pa* et *tubi*. Le MDLChunker est en plus capable de factoriser un stimulus dont les chunks se chevauchent sans que l'un soit inclus dans l'autre.

Optimisation

Avec PARSER, la création de nouveaux chunks se fait de la façon suivante. Si le stimulus courant n'est pas familier (il ne correspond à aucun chunk dont l'utilité est supérieure au seuil de familiarité), alors un nouveau chunk représentant le stimulus est créé. La création systématique de nouveaux chunks est compensée par le phénomène d'oubli décrit ci-avant. Seuls subsistent les chunks dont le taux d'apparition est supérieur au taux d'oubli.

Dans le cas du MDLChunker, un chunk n'est créé que s'il contribue à diminuer la taille de codage totale du système. Ce mécanisme donne en pratique des résultats comparables à ceux obtenus par le mécanisme de création-oubli de PARSER.

Annexe B

Ensembles d'apprentissage

B.1 Exemple 5.2.4

Stimuli	Contenu
Stimulus 1	N6 N3 N4 N5
Stimulus 2	N2 N1 N0
Stimulus 3	N0 N2 N1
Stimulus 4	N5 N6
Stimulus 5	N5 N4 N6 N3
Stimulus 6	N4 N3
Stimulus 7	N2 N1 N0
Stimulus 8	N3 N4
Stimulus 9	N2 N1 N0
Stimulus 10	N3 N6 N5 N4
Stimulus 11	N3 N4
Stimulus 12	N4 N5 N3 N6
Stimulus 13	N1 N0 N2
Stimulus 14	N5 N4 N6 N3
Stimulus 15	N5 N6
Stimulus 16	N5 N3 N4 N6
Stimulus 17	N4 N6 N3 N5
Stimulus 18	N3 N4 N6 N5
Stimulus 19	N5 N6
Stimulus 20	N4 N6 N4 N5
Stimulus 21	N3 N5 N6 N4
Stimulus 22	N2 N0 N1
Stimulus 23	N5 N3 N4 N6
Stimulus 24	N6 N5 N3 N4
Stimulus 25	N4 N3 N6 N5

TABLE B.1 – Liste des stimuli utilisés comme ensemble d'apprentissage pour l'exemple 5.2.4 page 113

.

B.2 Stimuli de l'expérience du chapitre 7

1	.	.	✓	.	.	.	.	.	✓	.	.	.	.	.	.	.	.	.	.	.
2	.	.	.	✓	.	✓	.	✓	.	✓	✓	.	.	✓	.	.	.	.	.	✓
3	.	✓	.	.	.	.	✓	.	✓	.	.	.	.	.	✓	✓	✓	✓	.	.
4	.	✓	.	.	.	.	.	.	✓	.	.	.	.	.	.	✓	.	✓	✓	.
5	✓	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	✓	.
6	.	✓	.	✓	.	✓	✓	✓	✓	✓	✓	.	.	✓	✓	✓	.	✓	.	✓
7	.	✓	✓	.	✓	✓	✓	✓	.	✓	.	✓	✓	.	✓	.	✓	.	.	.
8	.	✓	.	.	.	.	✓	.	✓	✓	.	.	.	.	✓	✓	.	✓	.	.
9	✓	.	✓	.	✓	.	.	.	.	.	.	✓	✓	.	.	.	✓	.	✓	.
10	✓	.	✓	.	✓	.	.	.	.	.	.	✓	✓	.	.	.	✓	.	✓	.
11	✓	✓	.	.	.	.	✓	.	✓	.	.	.	.	.	✓	✓	.	✓	✓	.
12	.	.	.	✓	.	✓	.	✓	.	✓	✓	.	.	✓	.	.	.	.	.	✓
13	.	.	.	.	.	.	.	.	.	.	.	✓	.	.	.	.	.	.	.	.
14	.	✓	.	.	.	✓	✓	✓	✓	✓	.	.	.	.	✓	✓	.	✓	.	.
15	.	✓	.	.	.	.	✓	.	✓	.	.	.	.	.	✓	✓	.	✓	.	.
16	.	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	.	✓
17	.	.	.	.	.	.	.	.	.	.	✓	.	.	.	✓	.	.	.	.	.
18	.	.	✓	✓	✓	.	.	.	.	.	✓	✓	✓	✓	.	.	✓	.	.	✓
19	✓	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	✓	.
20	✓	✓	✓	.	✓	.	✓	.	✓	.	.	✓	✓	.	✓	✓	✓	✓	✓	.
21	✓	.	✓	.	✓	.	✓	.	.	.	✓	✓	✓	.	✓	.	✓	.	✓	.
22	.	✓	.	✓	.	.	✓	.	✓	.	✓	.	.	✓	✓	✓	.	✓	.	✓
23	✓	✓	✓	✓	✓	.	✓	.	.	.	✓	✓	✓	✓	✓	.	✓	.	✓	✓
24	.	.	.	✓	.	✓	.	✓	✓	✓	✓	.	.	✓	.	✓	.	✓	.	✓
25	✓	.	✓	.	✓	✓	.	✓	.	✓	.	✓	✓	.	.	.	✓	.	✓	.
26	✓	.	✓	.	✓	.	.	.	.	.	.	✓	✓	.	.	.	✓	.	✓	.
27	✓	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	✓	.
28	.	✓	✓	✓	✓	✓	✓	✓	.	✓	✓	✓	✓	✓	✓	.	✓	.	.	✓
29	.	.	.	✓	.	.	.	.	✓	.	✓	.	.	✓	.	✓	.	✓	.	✓
30	.	.	.	.	.	✓	.	✓	.	✓	.	.	.	.	.	.	.	.	.	.
31	.	✓	.	.	.	.	✓	.	.	.	✓	.	.	.	✓	.	.	.	.	.
32	.	.	.	✓	.	.	.	.	.	.	✓	.	.	✓	.	.	.	.	.	✓
33	.	.	.	✓	.	✓	✓	✓	.	✓	✓	.	.	✓	.	.	.	.	.	✓
34	.	.	.	.	.	✓	.	✓	.	✓	.	.	.	.	.	.	.	.	.	.
35	.	.	.	✓	.	.	.	.	✓	.	✓	.	.	✓	.	✓	.	✓	.	✓
36	✓	.	.	✓	.	.	.	.	✓	.	✓	.	.	✓	.	✓	.	✓	.	✓
37	.	.	.	✓	.	✓	.	✓	.	✓	✓	.	.	✓	.	.	.	.	.	✓
38	.	.	✓	.	.	.	.	.	.	.	✓	.	.	.	.	.	.	✓	.	.
39	.	.	.	.	.	✓	.	✓	.	✓	.	.	.	.	.	.	.	.	.	.

TABLE B.2 – Les 75 lignes du tableau représentent les 75 stimuli de l'expérience du chapitre 7. Les 20 colonnes représentent les 20 symboles utilisés. La présence du symbole dans le stimulus est noté par « ✓ » et son absence par « . »

40	.	✓	.	.	.	.	✓	.	✓	.	.	.	.	.	✓	✓	.	✓	.	.
41	.	✓	✓	.	✓	✓	✓	✓	.	✓	.	✓	✓	.	✓	.	✓	.	.	.
42	.	✓	.	.	.	.	✓	.	✓	.	.	.	.	.	✓	✓	.	✓	.	.
43	.	.	.	✓	✓	✓	.	✓	✓	✓	✓	.	.	✓	.	✓	.	✓	.	✓
44	.	.	.	✓	.	✓	.	✓	.	✓	✓	.	.	✓	.	.	.	.	.	✓
45	✓	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	✓	.
46	.	✓	✓	.	✓	.	✓	.	.	.	.	✓	✓	.	✓	.	✓	.	✓	.
47	✓	✓	.	✓	.	✓	✓	✓	✓	✓	✓	.	.	.	✓	✓	.	✓	✓	✓
48	.	.	.	✓	.	✓	.	✓	.	✓	✓	.	.	✓	.	.	.	.	.	✓
49	✓	✓	✓	.	✓	.	✓	.	✓	.	.	✓	✓	.	✓	✓	✓	✓	✓	.
50	.	✓	.	.	.	.	✓	.	✓	.	.	.	✓	.	✓	✓	.	✓	.	.
51	✓	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	✓	.
52	.	✓	✓	.	✓	.	✓	.	✓	.	.	✓	✓	.	✓	✓	✓	✓	.	.
53	.	.	.	✓	.	.	.	.	✓	.	✓	✓	.	✓	.	✓	.	✓	.	✓
54	.	.	✓	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
55	.	✓	✓	.	✓	✓	✓	✓	✓	.	.	✓	✓	.	✓	✓	.	✓	.	.
56	.	.	.	✓	.	✓	.	✓	.	✓	✓	.	.	✓	.	.	.	.	✓	✓
57	.	.	.	.	.	✓	.	✓	.	✓	.	.	.	.	.	.	.	.	✓	.
58	.	.	.	.	.	.	✓	.	.	.	.	.	.	.	.	.	.	.	.	.
59	.	.	.	✓	.	✓	.	✓	.	✓	✓	.	.	✓	.	.	.	.	.	✓
60	.	✓	✓	.	✓	.	✓	.	.	.	.	✓	✓	.	✓	.	✓	.	.	.
61	.	✓	.	.	.	.	✓	.	✓	.	.	.	.	.	✓	✓	.	✓	.	.
62	✓	.	.	✓	.	.	.	.	✓	.	✓	.	.	✓	.	✓	.	✓	✓	✓
63	✓	✓	✓	.	✓	.	✓	.	✓	.	.	✓	✓	.	✓	✓	✓	✓	✓	✓
64	.	✓	✓	.	✓	.	✓	.	.	.	.	✓	✓	.	✓	.	✓	.	.	.
65	.	.	.	.	.	.	.	✓	.	.	.	.	.	.	.	.	.	.	.	.
66	✓	.	.	✓	.	✓	.	✓	.	✓	✓	.	.	✓	.	.	.	.	✓	.
67	✓	✓	✓	.	✓	.	✓	.	.	.	.	✓	✓	.	✓	.	✓	.	.	.
68	.	✓	.	.	.	.	✓	.	.	.	.	.	.	.	✓	.	.	.	.	.
69	.	.	.	.	.	✓	.	✓	.	✓	.	.	.	.	.	.	.	✓	.	.
70	✓	.	✓	.	✓	.	.	.	.	.	.	✓	✓	.	.	.	✓	.	✓	.
71	✓	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	✓	.
72	✓	✓	✓	.	✓	.	✓	.	.	.	.	✓	✓	.	✓	.	✓	.	✓	.
73	.	.	.	✓	.	.	.	✓	✓	✓	✓	.	.	✓	.	✓	.	✓	.	✓
74	.	✓	.	.	.	.	✓	✓	✓	.	.	.	.	.	✓	✓	.	✓	.	.
75	.	✓	✓	.	✓	✓	.	✓	.	✓	.	✓	✓	.	.	.	✓	.	.	.

TABLE B.3 – Les 75 lignes du tableau représentent les 75 stimuli de l'expérience du chapitre 7. Les 20 colonnes représentent les 20 symboles utilisés. La présence du symbole dans le stimulus est noté par « ✓ » et son absence par « . »

B.3 Interface de validation du MDLChunker

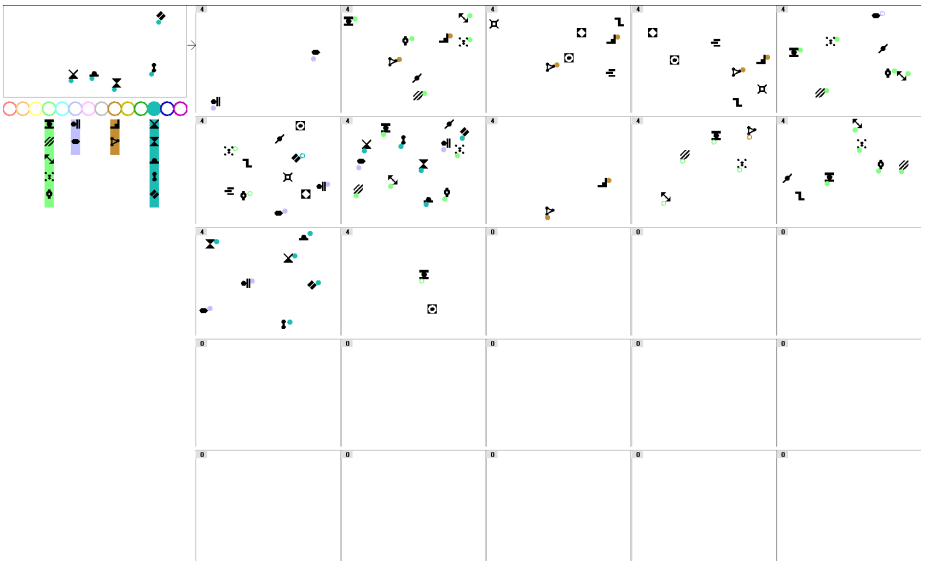


FIGURE B.1 – Interface complète utilisée pour l'expérience du chapitre 7. L'historique comporte les 25 dernières phrases vues.

B.4 Stimuli de l'expérience du chapitre 7

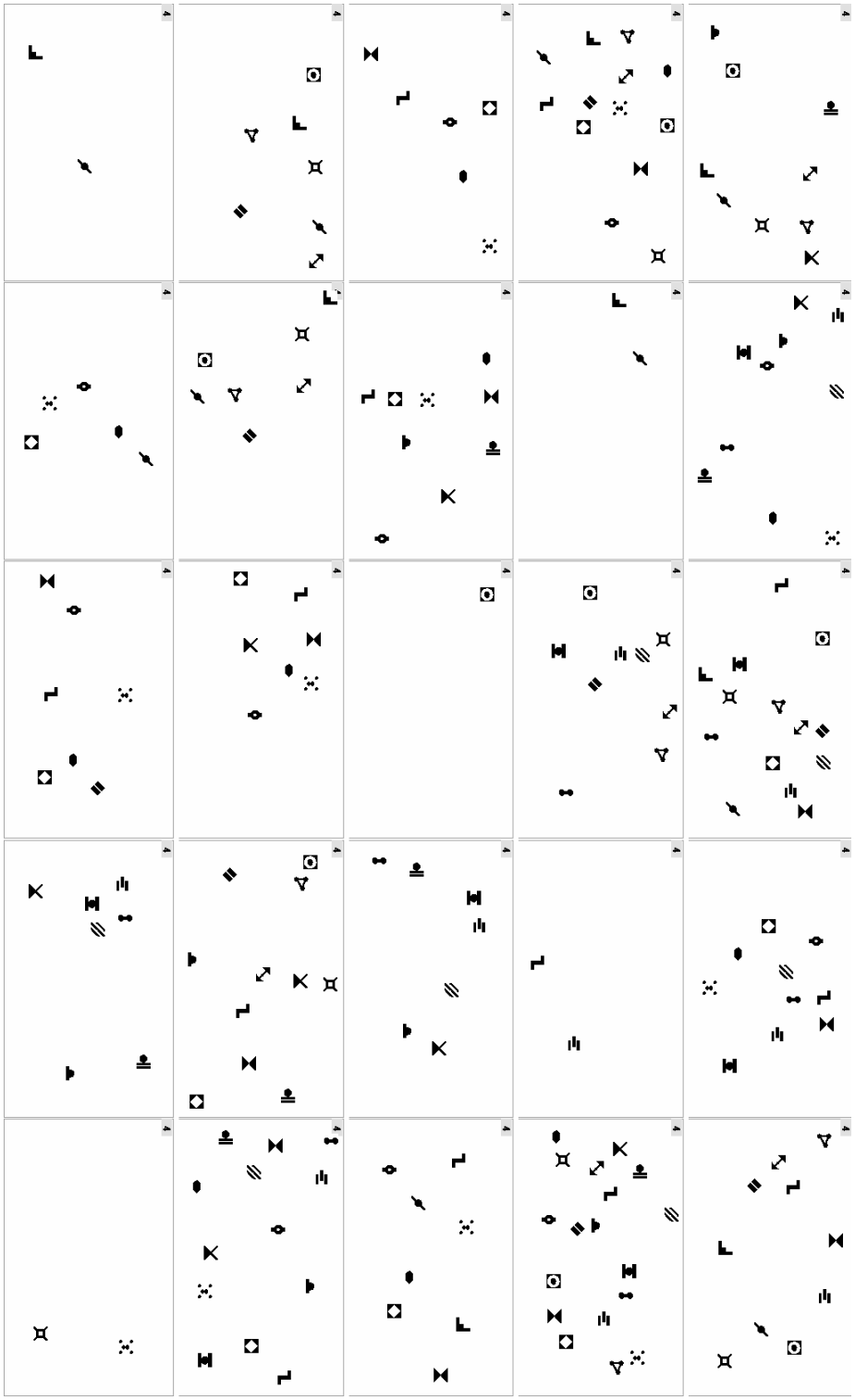


FIGURE B.2 – 25 premiers stimuli utilisés pour l'expérience du chapitre 7.