UNIVERSITÉ MONTPELLIER II
— Sciences et Techniques du Languedoc —
U.F.R. Sciences de Montpellier

Thèse de Doctorat de l'Université

Discipline	: Informatique
Formation Doctorale	: Informatique
École Doctorale	: Information, Structures et Systèmes

Modèles de Markov cachés et apprentissage par fusions d'états : algorithmes, applications, utilisations pour le test de circuits intégrés

par

Laurent BRÉHÉLIN

présentée et soutenue publiquement le *11 Juin 2001* devant le jury :

M. Michel Habib, Professeur, Université Montpellier II .... Président
M. Olivier Gascuel, Directeur de recherche, CNRS/LIRMM .... Directeur de thèse
M. Dominique Cellier, Maître de conférences, Université de Rouen .... Rapporteur
Mme Sybille Hellebrand, Professeur, Université d'Innsbruck .... Rapporteur
M. Jean-François Mari, Professeur, Université de Nancy II .... Rapporteur
M. Christian Landrault, Directeur de recherche, CNRS/LIRMM .... Examinateur

Remerciements

Cette thèse est dédiée à mes parents, qui ont réussi à me convaincre de faire des études.

Je tiens ensuite à remercier Olivier Gascuel, mon directeur de thèse, pour avoir réussi le périlleux exercice de me guider sans rien m'imposer, et pour avoir su me pousser à toujours aller de l'avant. Sa rigueur scientifique, ses connaissances, mais également son soutien et sa disponibilité, m'ont permis, pendant ces presque quatre années de thèse, de réaliser ce travail en y prenant toujours un réel plaisir.

Gilles Caraux m'a simplifié la vie à plusieurs reprises, notamment à la fin de mon DEA lorsque je cherchais un organisme d'accueil pour être objecteur de conscience, et, après cela, lorsque je lui ai dit que je souhaitais faire une thèse et que j'étais en quête d'un financement (il m'en trouva deux !). Pour cela, mais également bien sûr pour son encadrement pendant les premières années de ma thèse, je le remercie.

Je tiens également à remercier Patrick Girard et Christian Landrault pour avoir accepté de m'inculquer quelques notions de microélectroniques, pour avoir toujours répondu à mes questions (mêmes les plus naïves) avec la plus grande gentillesse, et, bien sûr, pour tout le travail que nous avons fait ensemble.

Merci à Bernard Jacq et Violaine Pillet pour nous avoir présenté leur problème d'extraction d'informations à partir de séquences textuelles.

Je remercie ensuite les membres de mon jury, et plus particulièrement mes rapporteurs, Dominique Cellier, Sybille Hellebrand et Jean-François Mari, qui ont accepté cette tâche presque sans hésiter. Merci pour leur confiance, et leurs remarques pertinentes sur ce travail. Je tiens à remercier également Michel Habib, qui a accepté de présider mon jury, et qui a pris la peine de s'intéresser à mon travail depuis le début.

Je voudrais ensuite remercier deux autres personnes dont j'ai pu apprécier le sérieux (ahhhh ... les feuilles de TD prêtes et corrigées ...), mais également la gentillesse et l'humour : Yolande et Ehoud Ahronovitz. Ils sont les enseignants que j'aurais aimé avoir.

Merci également à Christophe Fiorio et Eric Rivals pour leurs conseils, leur aide, et toutes les discussions que l'on a pu avoir en dehors du travail.

Merci à Josette et Nadine pour avoir toujours accepté avec le sourire un bon de commande à moitié rempli ou un ordre de mission déjà effectuée.

Je tiens ensuite à remercier ma soeurette et mes amis de toujours, Patrice & Élodie, Minus & Cissou, Olive & Sof, Michou, Lole, Laguille, pour le soutien qu'ils m'apportent si souvent, et pour tous les moments que l'on passe ensemble.

Je remercie ensuite tous les thésards que j'ai pu côtoyer, et plus particulièrement ceux qui sont devenus des amis, Patrice, Stéphane, Sèverine, Vincent, David, Christophe, Gab, Gilles, Fabien, Richard, Hervé, Yannic, Sandra et Anne-Lise, et que j'ai pu apprécier en dehors du labo, autour d'un jeu, d'une bière, d'une table et/ou d'une bombe. À ce propos, je ne résiste pas au plaisir de parodier David (voir [Genest, 2001]), et je (22546) remercie donc le jaune (21774), le gris (11026), le Gab (10393), mon Gilles (10014), le barbu (9585), boulet (9297), le marronasse (7988), le Fabien (7928), le pabô (7658), boulette (6829), l'autre bleu (6484), le Pat (4765), et Hervuit (3365). Tac, tac.

Merci enfin à Domi, pour m'avoir accompagné tout au long de cette thèse. Ce travail lui doit beaucoup, bien plus qu'elle ne le pense.

Table des matières

Introduction

La modélisation de séquences couvre un large spectre de disciplines et d'applications. On peut citer parmi les plus classiques le traitement de la parole, le traitement de textes manuscrits, la bioinformatique, la recherche d'informations dans les bases de données, le contrôle de robots, la compression de données ... Toutes ces applications ont en commun la manipulation de séquences d'observations : signal sonore ou séquence de phonèmes dans le cadre du traitement de la parole, séquence de graphèmes pour le traitement de textes manuscrits, séquences de nucléotides ou d'acides aminés en biologie, séquence de mots dans les fouilles de données textuelles ... Ces observations peuvent être de nature discrète (par exemples les caractères d'un alphabet fini) ou continue (la fréquence d'un son, une température, ...). Elles peuvent être pures (c'est-à-dire provenant d'une source unique) ou bruitées (corrompues par des sources annexes). L'objet de la modélisation de séquences est la caractérisation de telles séquences d'observations. La processus de modélisation se décompose en deux phases distinctes qui sont le choix d'une classe de modèles de représentation et le paramétrage de la classe choisie de manière à obtenir un modèle qui représente au mieux les séquences étudiées.

Il existe un grand nombre de classes de modèles adaptées aux séquences. De manière générale, ces classes peuvent être réparties en deux grandes familles qui sont les modèles non probabilistes et les modèles probabilistes. Parmi les classes de modèles non probabilistes on trouvera, entre autres, les automates d'états finis déterministes et non déterministes, et les différents types de grammaires. Les classes de modèles probabilistes comptent parmi elles les automates stochastiques, les n-grams, les modèles de Markov cachés ... Le choix d'une classe de modèles plutôt qu'une autre repose sur plusieurs critères. Le premier critère dépend bien évidemment des séquences à traiter et de leur caractéristiques intrinsèques. Toutes les classes n'expriment pas les mêmes propriétés et certaines sont plus à même de capturer les spécificités voulues. Le second critère qui va guider le choix du modèle est la tâche pour laquelle on désire modéliser les séquences. On comprend en effet aisément que, par exemple, faire de la reconnaissance vocale et isoler les parties codantes et non-codantes d'une chaîne de nucléotides ne relève pas de la même approche. Or toutes les classes de modèles ne permettent pas le traitement de toutes les applications. Le dernier critère important est celui de la facilité avec laquelle on va pouvoir paramétrer le modèle de manière à ce qu'il représente au mieux les séquences. On touche là à la seconde phase du processus de modélisation, celle de l'apprentissage.

Le paramétrage d'un modèle probabiliste à partir d'un corpus de données relève de la *modélisation statistique*. Cette discipline compose une branche importante de ce que l'on appelle *l'apprentissage automatique*. Ce qui fait la spécificité de l'apprentissage par rapport à la modélisation statistique, est qu'il s'intéresse avant tout à l'aspect calculatoire du processus de modélisation, en vue de son automatisation [Baldi et Brunak, 1998]. On retrouve sous cet angle de vue, l'importance du choix du modèle. Certaines classes peuvent être paramétrées de manière optimale à l'aide d'algorithmes d'apprentissage polynômiaux. Pour d'autres, le problème est plus difficile, et il faut alors utiliser des méthodes heuristiques.

La classe à laquelle nous allons nous intéresser dans cette thèse est celle des *modèles de Markov cachés* (*Hidden Markov Model* ou *HMM*). Comme les réseaux de neurones, ces modèles peuvent être définis par une structure, composée d'états et de transitions, et par un ensemble de probabilités qui paramètrent la structure. Généralement, dans les applications classiques comme la reconnaissance de la parole ou la biologie, certaines connaissances *a priori* du domaine permettent d'inférer naturellement une structure, ou tout du moins sa forme générale. Le problème de l'apprentissage se limite alors à estimer les valeurs des paramètres associés à cette structure prédéfinie. Pour d'autres domaines néanmoins, rien ne permet d'inférer facilement la structure. On peut citer parmi ceux-ci, celui de l'extraction d'informations à partir de données textuelles [Freitag et McCallum, 2000]. Le paramétrage d'un HMM de structure connue à partir d'un corpus de séquences d'apprentissage est un problème difficile [Abe et Warmuth, 1992]. Celui de l'apprentissage de la structure en même temps que des paramètres l'est encore plus. Ce problème constitue le sujet principal de la première partie de cette thèse.

Dans cette partie, nous commençons (chapitre 1) par décrire très brièvement le cadre mathématique dans lequel se place le problème de l'apprentissage et les quelques notions de base qui seront utilisées par la suite. Nous présentons ensuite (chapitre 2) une vue d'ensemble des modèles de Markov cachés, des applications qu'ils peuvent traiter, des algorithmes classiques de la littérature et du problème de l'apprentissage. Les chapitres suivants sont dédiés au problème de l'apprentissage sans structure connue, à travers une méthode proposée à la fois par Stolcke et Omohundro [Stolcke et Omohundro, 1994a] dans le cadre des HMMs, et par Carrasco et Oncina [Carrasco et Oncina, 1994] pour les automates probabilistes. Le principe de cette méthode (chapitre 3) est de généraliser itérativement un modèle spécifique des séquences d'apprentissage à l'aide d'une opération de fusions d'états. La procédure de sélection des états à fusionner à chaque itération est la clef de l'algorithme. Elle détermine la qualité et la rapidité de l'apprentissage. Stolcke et Omohundro proposent d'utiliser un critère de sélection construit sur une probabilité *a posteriori*. Nous verrons qu'il est possible de simplifier ce critère, et au chapitre 4, nous proposons une optimisation de la procédure de sélection. Le chapitre 5 présente quant à lui un nouveau critère de sélection, plus élaboré que celui de Stolcke et Omohundro, et qui conduit à un nouvel algorithme d'apprentissage. Le dernier chapitre de cette partie (chapitre 6) présente quelques expérimentations réalisées en utilisant les différents algorithmes d'apprentissage. Une de ces expérimentations est une application réelle d'extraction d'informations à partir de données textuelles. Comme nous l'avons dit, ce domaine constitue l'un des exemples pour lesquels aucune connaissance ne permet d'inférer naturellement la structure du modèle. La seconde partie de ce document en présente un autre : le test de circuits microélectroniques.

Le test d'un circuit microélectronique consiste à mettre en évidence un éventuel dysfonctionnement dû à une défaillance physique. Comme nous le décrivons au chapitre 7, la mise en évidence d'un tel dysfonctionnement peut être effectué de manière logique en appliquant certaines valeurs aux entrées du circuit, et en comparant la réponse du circuit à ces valeurs aux réponses logiquement attendues de la part d'un circuit sain. Dans le cadre qui nous intéresse, c'est-à-dire les circuits qui comportent des cellules mémoires, la mise en évidence des dysfonctionnements nécessite l'application aux entrées du circuit de *séquences* de vecteurs booléens. Ce sont ces séquences que l'on va chercher à modéliser. On utilise généralement l'apprentissage dans le cadre de domaines pour lesquels on dispose de nombreuses données, mais peu de théorie [Rumelhart et al., 1995]. Son application pour le test de circuits microélectroniques est en ce sens déjà une curiosité. Les séquences que l'on modélise sont en effet issues d'un processus déterministe et, qui plus est, complètement déterminé, puisqu'elles sont le résultat d'un algorithme. Pourquoi alors faire de l'apprentissage ? Pour générer de nouvelles séquences. Et c'est là une autre particularité de ce domaine d'application. Contrairement à ce qui se fait classiquement dans les autres domaines, les modèles appris ne servent à faire ni de la classification supervisée, ni de la segmentation, mais de la génération. Nous verrons deux applications de cela. Au chapitre 8, la tâche est la construction d'un générateur de séquences de test destiné à être physiquement implémenté sur silicium. La condition *sine qua non* à la réalisation de cette tâche est que ce générateur soit de taille réduite. Dans ce problème, on dispose d'un ensemble de séquences de test préalablement générées par les outils standards, mais on ne peut, pour des raisons de place, les stocker sur le circuit. Tout le problème est alors la recherche d'un générateur qui génère des séquences ayant les mêmes capacités de détection de fautes que ces séquences. Nous développons pour cela une nouvelle classe de modèles de Markov cachés – que nous appelons les *modèles de Markov cachés avec patterns* –, spécialement adaptée pour être implémentée à moindre coût. On utilise alors les séquences de test dont on dispose comme un ensemble d'apprentissage d'une instance de cette classe. Au chapitre suivant (chapitre 9), le problème est l'amélioration de ces mêmes séquences de test. Là encore, elles sont utilisées comme ensemble d'apprentissage, et le modèle sert à son tour à générer un grand nombre de nouvelles séquences. On sélectionne alors parmi celles-ci un ensemble minimal dont la qualité (estimée en fonction de sa taille et du nombre de fautes potentielles qu'il détecte) est supérieure à celle de l'ensemble original.

Première partie

Modèles de Markov cachés et apprentissage

Chapitre 1

Préliminaires

Nous donnons dans ce chapitre quelques connaissances de base qui seront utilisées dans la suite de ce document. Pour commencer (partie 1.1) nous présentons brièvement les classes de modèles probabilistes, et le cadre mathématique dans lequel nous nous plaçons. Nous décrivons comment utiliser ces modèles et la procédure classique utilisée pour les paramétrer. À la partie 1.2 nous exposons une première classe de modèles probabilistes très simples, la classe des unigrams, qui va nous permettre d'illustrer les quelques notions exposées dans la partie précédente.

1.1 Séquences et modèles probabilistes

1.1.1 Présentation générale

La classe de modèles à laquelle nous allons nous intéresser dans cette thèse est issue du vaste ensemble des modèles probabilistes. Ces modèles sont utilisés lorsque l'on souhaite caractériser les propriétés statistiques des séquences. L'hypothèse sous-jacente à ce genre de modélisation est que les séquences étudiées peuvent être caractérisées par un processus stochastique paramétré, et qu'il existe une méthode efficace pour déterminer (estimer) ces paramètres. Nous nous limiterons ici à l'étude de séquences d'observations de nature discrète. De manière générale, les observations qui composent les séquences seront supposées issues d'un alphabet Σ. On notera $O = o_1 \dots o_T$ une séquence d'observation de longueur T, avec $\forall t = 1$ à T, $o_t \in \Sigma$. La probabilité d'une séquence O, étant donné un modèle probabiliste M, sera notée $P(O|M)$. Elle pourra être interprétée comme la probabilité que M génère O et sera souvent appelée *vraisemblance* de O connaissant M. On notera $\mathcal{O} = \{O^1 \dots O^K\}$ un ensemble de K séquences d'observations. Nous considérerons ici que ces séquences sont indépendantes, et donc que la probabilité de $\mathcal{O}$ étant donné un modèle probabiliste M (ou la vraisemblance de $\mathcal{O}$ connaissant M) est simplement le produit des probabilités de chacune des séquences :

$$P(\mathcal{O}|M) = \prod_{k=1}^{K} P(O^k|M). \tag{1.1}$$

Une théorie générale permettant l'utilisation, la manipulation mais également le paramétrage des modèles probabilistes – et de manière encore plus générale, le raisonnement en présence d'incertitude – est l'approche *Bayesienne* (voir [Baldi et Brunak, 1998] pour une présentation plus détaillée), elle même centrée autour de la formule de Bayes :

$$P(M|X) = \frac{P(X|M)P(M)}{P(X)},$$

avec M un modèle et X un ensemble de données. $P(X)$ est la loi *prédictive* sur les données, $P(M)$ la probabilité *a priori* du modèle M, $P(X|M)$ est la probabilité de X connaissant M – ou la *vraisemblance* de X étant donné M –, et $P(M|X)$ est la probabilité *a posteriori* de M.

1.1.2 Utilisations

On peut faire de la modélisation de séquences pour différentes raisons. Cela peut évidemment être la recherche d'un modèle théorique d'un phénomène observé afin d'aider à sa compréhension, mais cela peut également être pour des raisons d'ordre purement pratique. Parmi celles-ci, le problème le plus classique est la *discrimination*, ou *classification supervisée*. On trouve dans cette catégorie les problèmes de reconnaissance d'un mot parmi un ensemble de mots possibles à partir d'un signal audio (reconnaissance de la parole) [Bahl et al., 1983] ou d'une écriture manuscrite [Kundu et al., 1988], les problèmes de reconnaissance d'une famille de protéines à partir d'une séquence d'acides aminés [Haussler et al., 1992], les problèmes de classification de textes d'une base de donnée textuelle [Mitchell, 1997] ... Pour cette catégorie de problèmes, on dispose d'un ensemble de classe de séquences $\mathcal{C} = \{c_1 \dots c_V\}$ construites sur un même alphabet Σ, et O une séquence d'observations à laquelle on veut assigner sa classe d'appartenance. L'approche classique pour résoudre ce problème consiste à associer à O la classe c_{MAP} dont la probabilité *a posteriori* est maximale (*maximum a posteriori* ou *MAP*) :

$$c_{\text{MAP}} = \operatorname*{argmax}_{c_v \in \mathcal{C}} P(c_v|O),$$

$P(c_v|O)$ étant la probabilité de la classe c_v étant donnée la séquence O. On peut utiliser la formule de Bayes pour réécrire cette expression :

$$c_{\text{MAP}} = \operatorname*{argmax}_{c_v \in \mathcal{C}} \frac{P(O|c_v)P(c_v)}{P(O)},$$

avec $P(O|c_v)$ la probabilité de la séquence O étant donnée la classe c_v, $P(c_v)$ la probabilité *a priori* de la classe c_v, et $P(O)$ la probabilité *a priori* de la séquence O. $P(O)$ étant indépendante de la classe, on peut simplifier cette expression pour obtenir :

$$c_{\text{MAP}} = \operatorname*{argmax}_{c_v \in \mathcal{C}} P(O|c_v)P(c_v). \tag{1.2}$$

Si l'on fait l'hypothèse que chaque classe de séquences c_v obéit à un modèle probabiliste M_v indépendant, c'est-à-dire que l'on a $\forall c_v,\ P(O|c_v) = P(O|M_v)$, alors l'expression (1.2) peut être écrite :

$$c_{\mathrm{MAP}} = \operatorname*{argmax}_{c_v \in \mathcal{C}} P(O|M_v)P(c_v).$$

Une autre catégorie courante de problèmes sont les problèmes de *prédiction*. Pour ce type d'applications, le problème est de prédire, étant donnée une séquence d'observations incomplète $O = o_1 \dots o_{T-1}$, l'observation o_T qui va suivre. Si M est un modèle probabiliste des séquences à traiter, le processus de prédiction consiste à trouver le symbole a_{ML} qui a la probabilité maximale de succéder à O :

$$a_{\mathrm{ML}} = \operatorname*{argmax}_{a \in \Sigma} P(Oa|M),$$

avec Oa le résultat de la concaténation de l'observation a à la suite de la séquence O. a_{ML} est donc l'observation de vraisemblance maximale (*maximum likelihood* ou *ML*).

Enfin, une dernière catégorie de problèmes sont ce que l'on appelle les problèmes de *segmentation*. Pour ce type d'applications, les séquences manipulées sont supposées être composées de sous-séquences de différents types. Le problème est alors de découper une séquence donnée de manière à isoler les sous-séquences qui la composent. On trouve dans cette catégorie les problèmes de découpage d'un signal musical en notes [Raphael, 1998], de localisation de régions codantes/non-codantes dans une chaîne de nucléotides [Krogh et al., 1994], d'extraction de phrases clefs d'un texte parmi un ensemble important de phrases à faible niveau informationnel [Freitag et McCallum, 1999], ou de découpage d'un article en différents champs (titre, date, nom de l'auteur, ...) [Seymore et al., 1999]. Nous verrons au chapitre suivant comment résoudre ce type de problèmes à l'aide de modèles de Markov cachés.

1.1.3 Apprentissage

Une fois une classe de modèles probabilistes choisie, le problème est de trouver les paramètres de cette classe de manière à obtenir un modèle qui représente au mieux les séquences à traiter. Dans le cas de figure le plus favorable, ce paramétrage peut être effectué manuellement, c'est-à-dire à partir de certaines connaissances dont on dispose sur le domaine et qui permettent un choix naturel des paramètres. Mais dans la majorité des cas, on ne dispose pas de telles connaissances, et les paramètres doivent être inférés à l'aide d'un processus d'apprentissage à partir d'un ensemble de séquences supposées représentatives des séquences à traiter. Cet ensemble de séquences est appelé *ensemble d'apprentissage*. Les séquences d'apprentissage devront bien évidemment avoir été préalablement renseignées en fonction de la tâche pour laquelle on souhaite faire de la modélisation : pour les tâches de discrimination, on devra connaître la classe d'appartenance de chacune des séquences d'apprentissage, alors que pour les tâches de segmentation elles devront avoir été découpées en sous-séquences elles-mêmes étiquetées par leur type.

Soit $\mathcal{M}$ une classe de modèle probabiliste et $\mathcal{O}$ un ensemble de séquences d'apprentissage. Suivant une approche Bayesienne, l'apprentissage consiste à trouver, parmi toutes les instances de $\mathcal{M}$, le modèle M_{MAP} qui a la probabilité *a posteriori* maximale :

$$M_{\mathrm{MAP}} = \operatorname*{argmax}_{M \in \mathcal{M}} P(M|\mathcal{O}).$$

On peut là encore utiliser la formule de Bayes pour réécrire cette expression et obtenir :

$$M_{\mathrm{MAP}} = \operatorname*{argmax}_{M \in \mathcal{M}} P(\mathcal{O}|M)P(M), \tag{1.3}$$

$P(M)$ étant la probabilité *a priori* du modèle M. Dans certains cas, on peut considérer que toutes les instances de $\mathcal{M}$ ont la même probabilité *a priori* ($P(M_i) = P(M_j)$ pour tout M_i et M_j de $\mathcal{M}$). Dans ce cas, on peut simplifier l'expression (1.3) et considérer uniquement le terme $P(\mathcal{O}|M)$ pour trouver le modèle le plus probable. Celui-ci est alors le modèle de vraisemblance maximale (noté M_{ML}), et on parle alors d'apprentissage suivant le *principe du maximum de vraisemblance* :

$$M_{\mathrm{ML}} = \operatorname*{argmax}_{M \in \mathcal{M}} P(\mathcal{O}|M). \tag{1.4}$$

1.2 L'exemple des unigrams

1.2.1 Présentation

Les unigrams définissent sans doute la classe de modèles probabilistes la plus simple qui puisse être utilisée pour modéliser des séquences de symboles discrets. La modélisation de séquences à l'aide d'un unigram consiste à considérer que ces séquences ont été générées par un processus stochastique dépourvu de mémoire. Plus précisément, la génération d'un symbole d'une séquence ne dépend que d'une probabilité *a priori* attachée au symbole et en aucun cas des symboles générés précédemment par le modèle. Un unigram peut donc être défini par un doublet $U = \langle \Sigma, P \rangle$ où :

- Σ est un alphabet de M symboles ;
- P est une distribution de probabilité sur Σ ; on note $P(a)$ la probabilité *a priori* de générer le symbole a.

P étant une distribution de probabilité on a :

$$0 \leq P(a) \leq 1, \ \forall a \in \Sigma,$$

et

$$\sum_{a \in \Sigma} P(a) = 1.$$

La probabilité de génération d'une séquence de symbole $O = o_1 \ldots o_T$ par un unigram U est simplement le produit des probabilités de génération de chacun des symboles de la séquence :

$$P(O|U) = \prod_{t=1}^{T} P(o_t|U).$$

Soit $\mathcal{O} = \{O^1 \dots O^K\}$ un ensemble de séquences de symboles indépendantes. La vraisemblance de $\mathcal{O}$ connaissant U exprimée par l'expression (1.1) s'écrit alors :

$$P(\mathcal{O}|U) = \prod_{k=1}^{K} \prod_{t=1}^{T_k} P(o_t^k|U).$$

1.2.2 Utilisation pour les problèmes de discrimination

Les unigrams peuvent être utilisés pour la résolution de problèmes de discrimination. Soit $\mathcal{C} = \{c_1 \dots c_V\}$ un ensemble de classes de séquences construites sur le même alphabet Σ, et $O = o_1 \dots o_T$ une séquence à classer. En suivant l'approche décrite à la section 1.1.2, le processus de classification consiste à associer à O la classe c_{MAP} telle que :

$$c_{\mathrm{MAP}} = \operatorname*{argmax}_{c_v \in \mathcal{C}} P(O|c_v)P(c_v).$$

Si on suppose que chaque classe c_v peut être modélisée par un unigram U_v, alors, cette expression devient :

$$\begin{aligned} c_{\mathrm{MAP}} &= \operatorname*{argmax}_{c_v \in \mathcal{C}} P(O|U_v)P(c_v) \\ &= \operatorname*{argmax}_{c_v \in \mathcal{C}} P(c_v) \prod_{t=1}^{T} P(o_t|U_v) \end{aligned} \tag{1.5}$$

Alors que l'hypothèse d'indépendance totale des probabilités des observations – hypothèse qui constitue la base de la définition de la classe des unigrams – est clairement incorrecte pour la plupart des séquences que l'on traite lorsque l'on fait de la modélisation, cette classe de modèles permet d'obtenir des résultats en discrimination souvent excellents. Ce paradoxe s'explique par le fait que le résultat du processus de classification est seulement fonction du signe (dans le cas où l'on ne considère que deux classes possibles) de la fonction de comparaison ; l'estimation des probabilités conditionnelles peut être fortement biaisée alors que l'estimation de la classe d'appartenance est bonne [Domingos et Pazzani, 1997].

1.2.3 Apprentissage

Si l'on veut utiliser les unigrams pour faire de la discrimination, on doit donc, suivant l'expression (1.5), connaître les probabilités *a priori* $P(c_v)$ de chacune des classes, ainsi que les probabilités conditionnelles $P(a|U_v)$ pour chaque observation $a \in \Sigma$ et chaque classe c_v. Nous allons voir maintenant comment estimer ces termes suivant le principe du maximum de vraisemblance, à partir d'un ensemble de séquences d'apprentissage $\mathcal{O} = \{O^1, \dots, O^k\}$ préalablement étiquetées par leur classe d'appartenance.

On peut aisément estimer chacun des termes $P(c_v)$ en calculant la fréquence de chaque classe c_v dans l'ensemble d'apprentissage. Pour les termes $P(a|U_v)$, la méthode consiste, comme on l'a vu à la section 1.1.3, à chercher, pour chaque classe c_v, les valeurs des probabilités $P(a|U_v)$ qui maximisent la vraisemblance des séquences de c_v donnée par l'expression :

$$P(\mathcal{O}_v|U_v) = \prod_{k=1/\mathcal{C}(O^k)=c_v}^{K} \prod_{t=1}^{T_k} P(o_t^k|U_v),$$

avec $\mathcal{O}_v$ l'ensemble des séquences d'apprentissage de classe c_v, et $\mathcal{C}(O^k)$ désignant la classe de la séquence O^k. En notant $n_{a,v}$ le nombre de fois total où le symbole a est observé dans les séquences d'apprentissage de classe c_v, alors cette expression peut être réécrite :

$$P(\mathcal{O}_v|U_v) = \prod_{a\in\Sigma} P(a|U_v)^{n_{a,v}},$$

et les estimateurs qui maximisent cette expression sont :

$$\widehat{P}(a|U_v) = \frac{n_{a,v}}{\sum_{b\in\Sigma} n_{b,v}}$$

Le problème principal posé par ces estimateurs survient pour les symboles dont la probabilité conditionnelle à la classe est très faible. En effet, si a est un symbole dont la probabilité conditionnelle $P(a|c_v)$ est faible, alors il très possible que ce symbole n'apparaisse pas dans les séquences d'apprentissage de c_v. On aura alors $n_{a,v} = 0$ et $\widehat{P}(a|U_v) = 0$ ce qui pose un problème évident lors de l'utilisation de l'unigram pour la classification de nouvelles séquences où, incidemment, ce symbole apparaît : chacune de ces séquences aura alors une probabilité $P(O|U_v) = 0$ et sera donc invariablement reconnue comme ne faisant pas partie de la classe c_v. Une solution pour résoudre ce problème est de *lisser* les distributions de probabilité de manière à éviter les probabilités nulles. La technique classique, appelée *Laplace smoothing*, consiste, pour chaque symbole, à ajouter une constante l au compte du nombre de fois où il est utilisé. Les estimateurs deviennent alors :

$$\widehat{P}(a|U_v) = \frac{n_{a,v} + l}{|\Sigma| \cdot l + \sum_{b\in\Sigma} n_{b,v}}$$

Un des principaux avantages des unigrams est que, du fait de l'hypothèse d'indépendance, les paramètres du modèle peuvent être estimés séparément. Cela simplifie grandement l'apprentissage et est très appréciable, spécialement lorsque l'alphabet manipulé est de très grande taille. C'est par exemple le cas dans le cadre des applications de discrimination de document textuels. Pour ce type d'applications, les symboles de l'alphabet sont les mots du langage courant, et pour certaine tâches – classification d'articles de *News*, de pages Web, de commentaires issus de bases de données, ... – la taille du vocabulaire peut atteindre plusieurs milliers de mots. Les unigrams ont donc été abondamment utilisés pour ce type de tâches et permettent généralement d'obtenir des résultats extrêmement bons [Lewis et Gale, 1994] [Mitchell, 1997]. Nous en verrons un exemple au chapitre 6, où ils sont utilisés pour discriminer de courts résumés bibliographiques issus de la base de données FlyBase [The FlyBase Consortium, 1999].

Chapitre 2

Modèles de Markov cachés

2.1 Introduction

Les modèles de Markov cachés (Hidden Markov Models ou HMMs) ont été introduits par Baum et ses collaborateurs dans les années 1960-70s [Baum et al., 1970]. Cette classe de modèles est fortement apparentée aux automates probabilistes (PAs) [Casacuberta, 1990]. Un automate probabiliste est défini par une structure composée d'états et de transitions, et par un ensemble de distributions de probabilités sur les transitions. À chaque transition est associée un symbole d'un alphabet fini. Ce symbole est généré à chaque fois que la transition est empruntée. Un HMM se définit également par une structure composée d'états et de transitions et par un ensemble de distributions de probabilités sur les transitions. La génération de symboles est généralement attachée aux états, mais elle peut éventuellement être attachée aux transitions [Bahl et al., 1983]. On associe alors à chaque état (ou transition) non pas un symbole, mais une distribution de probabilités sur les symboles de l'alphabet. Une illustration de la forte parenté entre HMMs et PAs peut être trouvée dans la propriété que n'importe quel HMM peut être simulé par un PA du même nombre d'états [Abe et Warmuth, 1992]. Notons que la propriété inverse n'est pas vraie.

Le premier domaine auquel les HMMs ont été appliqués fut le traitement de la parole au début des années 1970 [Baker, 1975], [Rabiner, 1989]. Dans ce domaine, les HMMs se sont rapidement imposés comme le modèle de référence et une majeure partie des techniques d'utilisation et d'implémentation des HMMs (choix d'une structure ou d'un type de structure, problèmes des probabilités faibles, ...) on été développés dans le cadre de ces applications. Ces techniques furent ensuite appliquées et adaptées avec succès aux problèmes de reconnaissance de textes manuscrits [Kundu et al., 1988], [Schenkel et al., 1994]) et d'analyse de séquences biologiques [Haussler et al., 1992], [Durbin et al., 1998]. Les HMMs ont également été appliqués à d'autres domaines comme la reconnaissance d'images [Povlow et Dunn, 1995], ou la modélisation d'un signal musical [Raphael, 1998]. Depuis très récemment, avec l'augmentation importante de la masse des données électroniques, une nouvelle application prometteuse semble être l'extraction d'informations à partir de données textuelles [Seymore et al., 1999], [Freitag et McCallum, 1999].

L'objectif de ce chapitre est de présenter une vue d'ensemble des HMMs. Les algorithmes et les problèmes classiques de la littérature sont exposés et illustrés autant que possible par des applications réelles. La section 2.2 présente de manière formelle les modèles de Markov cachés. À la section 2.3 nous présentons les deux principaux types de problèmes qui peuvent être résolus par les HMMs, c'est-à-dire les problèmes de discrimination et les problèmes de segmentation, et nous exposons les algorithmes classiques associés à leur résolution : l'algorithme de calcul de la probabilité d'une séquence pour les applications de discrimination et l'algorithme de calcul du chemin le plus probable pour les applications de segmentation. Quelle que soit l'application traitée, un des principaux problèmes que l'on rencontre lorsqu'on utilise des HMMs est la construction d'un modèle adapté aux séquences que l'on souhaite modéliser. La section 2.4 est dédiée à ce problème d'apprentissage. Nous verrons les différents cas de figure qu'il présente, dépendant des séquences à modéliser et des connaissances *a priori* dont on dispose.

2.2 Présentation

Un HMM H peut être défini par un quadruplet $\langle \mathcal{Q}, \Sigma, \mathcal{T}, \mathcal{G} \rangle$ avec :

- $\mathcal{Q}$, un ensemble de N états ; $\mathcal{Q}$ contient deux états spéciaux muets *start* et *end* qui servent respectivement à débuter et conclure une séquence.
- Σ, un alphabet de symboles ;
- $\mathcal{T} = \mathcal{Q} - \{end\} \times \mathcal{Q} - \{start\} \rightarrow [0,1]$, une matrice indiquant les probabilités de transition d'un état à l'autre ; on note $P(s \rightarrow s')$ la probabilité de transition de l'état s vers l'état s' ;
- $\mathcal{G} = \mathcal{Q} - \{start, end\} \times \Sigma \rightarrow [0,1]$, une matrice indiquant les probabilités de génération associées aux états ; on note $P(o|s)$ la probabilité de générer le symbole o à partir de l'état s.

La définition proposée ici n'est pas la définition originale donnée par Baum [Baum et al., 1970] dont elle diffère par l'introduction des états muets *start* et *end*. L'état *start* n'entraîne pas de modification essentielle du modèle et remplace avantageusement l'utilisation de la distribution de probabilités *a priori* associée aux états pour débuter les séquences dans la définition de Baum. L'état *end*, par contre, n'a pas d'équivalent dans la définition originale. Néanmoins, la définition présentée ici est celle habituellement utilisée dans la plupart des applications. Nous nous limiterons ici à la modélisation de séquences de symboles appartenant à un alphabet fini : un ensemble de *phonèmes* dans le cadre d'applications de reconnaissance de la parole, les quatre bases azotées (adénine, guanine, cytosine et thymine) pour la modélisation de séquences d'ADN, l'ensemble des acides aminés pour les séquences protéiques, les mots d'un certain corpus pour l'extraction d'information à partir de données textuelles, ... Néanmoins les HMMs peuvent également être utilisés pour la modélisation de signaux continus [Rabiner, 1989]. On définit la structure d'un HMM par l'ensemble de ses états, ainsi que par les transitions et les symboles de probabilité non nulle. La figure 2.1 représente un exemple de HMM avec sept états et onze transitions.

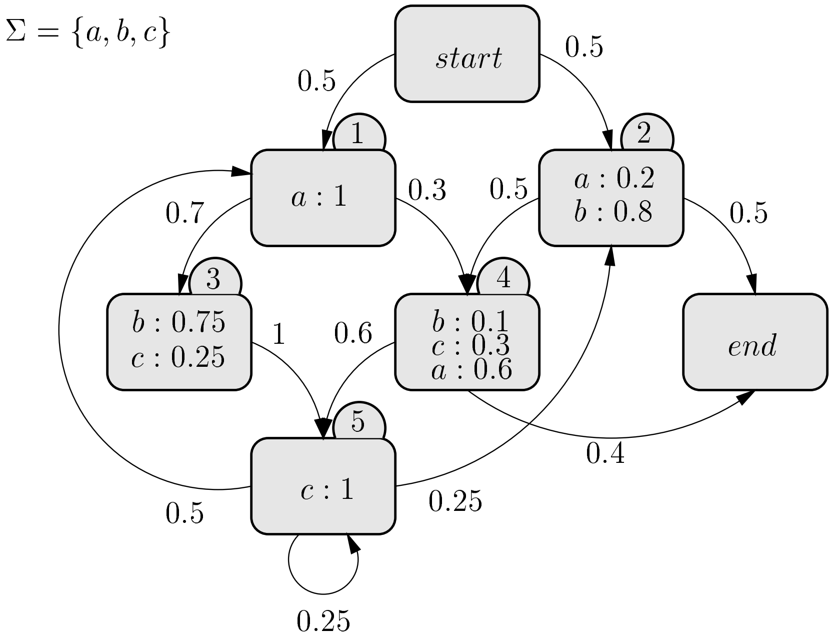


FIG. 2.1: Un exemple de HMM.

La procédure de génération d'une séquence de symboles à l'aide d'un HMM consiste à partir de l'état *start*, à se déplacer d'état en état suivant les probabilités de transition, et à générer un symbole sur chaque état rencontré en utilisant la distribution de probabilités de génération qui lui est associée. Lorsqu'un symbole a été généré, on choisit une transition sortante suivant la distribution de probabilités de transition associée à l'état courant, et la procédure est réitérée jusqu'à atteindre l'état *end*. Par exemple, la séquence *abccb* peut être générée par le HMM de la figure 2.1 en partant de l'état *start*, puis en allant sur l'état 1 puis sur l'état 3, puis 5, 5, 5, 2 et *end*. Notons que cette séquence de symboles peut également être générée en suivant la séquence d'état $start-1-4-5-5-5-2-end$, ou $start-2-4-5-5-5-2-end$. La probabilité de génération suivant la première séquence d'états – ou *chemin* – est la probabilité de passer de l'état *start* à l'état 1 (0.5), multipliée par la probabilité que l'état 1 émette le symbole a (1), multipliée par la probabilité de transition de 1 vers 3 (0.7), multipliée par $\ldots = 6.6 \cdot 10^{-3}$. Suivant les deuxième et troisième chemins, cette probabilité est respectivement $2.3 \cdot 10^{-4}$ et $7.5 \cdot 10^{-5}$. Comme il n'y a pas d'autres chemins qui permettent de générer cette séquence, la probabilité de génération de *abccb* par le HMM est $6.6 \cdot 10^{-3} + 2.3 \cdot 10^{-4} + 7.5 \cdot 10^{-5} = 6.9 \cdot 10^{-3}$. Les HMMs définissent donc un processus stochastique non déterministe, au sens où une même séquence de symboles peut être générée de plusieurs manières différents. On comprend alors mieux le nom donné à ce modèle : le processus de génération est un processus *markovien* – la probabilité de transition vers un état ne dépend que de l'état actuel et non des états rencontrés précédemment – qui est *caché* car il est impossible de connaître le processus suivi pour la génération d'un séquence de symboles donnée.

2.3 Utilisations et algorithmes de base

Les applications classiques des HMMs sont les problèmes de discrimination et les problèmes de segmentation. On trouve dans la première catégorie les applications de reconnaissance d'un mot parmi un ensemble de mots possibles à partir d'un signal audio (reconnaissance de la parole) [Rabiner, 1989] ou d'une écriture manuscrite [Kundu et al., 1988], de reconnaissance d'une famille de protéines à partir d'une séquence d'acides aminés [Haussler et al., 1992] ... Pour ces applications, on construit généralement un HMM différent pour chaque classe à reconnaître – par exemple un HMM pour chaque mot dans le cadre de la reconnaissance de la parole –, et, comme pour les unigrams, la reconnaissance d'une séquence consiste alors à calculer la probabilité de génération de la séquence par chacun des HMM, et à assigner à cette séquence sa classe la plus probable. Ce type d'application nécessite donc un algorithme efficace pour le calcul de la probabilité de génération d'une séquence. Cet algorithme est présenté à la section 2.3.1.

Dans les applications de segmentation qui utilisent les HMMs on trouve les problèmes de découpage d'un signal musical en notes [Raphael, 1998], de localisation de régions codantes/non-codantes dans une chaîne de nucléotides [Krogh et al., 1994], d'extraction de phrases clefs d'un texte parmi un ensemble important de phrases à faible niveau informationnel [Freitag et McCallum, 1999], de découpage d'un article en différents champs (titre, date, nom de l'auteur, ...) [Seymore et al., 1999] ... On utilise pour ces applications un HMM dont les états sont *typés*. La procédure de segmentation d'une séquence de symboles consiste alors à calculer à l'intérieur du HMM le chemin qui a la probabilité maximale de générer cette séquence. On associe ensuite à chaque symbole de la séquence son type, en fonction du type de l'état correspondant dans le chemin calculé. Considérons par exemple le HMM de la figure 2.2. En plus des états *start* et *end* ce HMM est composé de quatre autres états typés ARTICLE, ADJECTIF, NOM et VERBE. Remarquons que deux mots apparaissent dans plusieurs états : "modèle", qui peut être utilisé comme adjectif et comme nom, et "classe" qui peut être utilisé comme nom et comme verbe. Considérons maintenant la phrase "Le modèle classe la séquence.". Notre objectif est de segmenter cette phrase, c'est-à-dire d'assigner à chaque mot son type. Deux chemins du HMM permettent de générer cette phrase : *start*-ARTICLE-ADJECTIF-NOM-ARTICLE-NOM-*end* et *start*-ARTICLE-NOM-VERBE-ARTICLE-NOM-*end*. La probabilité de génération suivant le premier chemin est $1.8 \cdot 10^{-7}$, tandis qu'elle est de $6.22 \cdot 10^{-6}$ suivant le second chemin. La segmentation la plus probable est donc celle induite par le second chemin. Cette segmentation nous permet alors, par exemple, de conclure que "classe" est utilisé dans cette phrase comme verbe et non comme nom, ce qui n'est *a priori* pas évident – toute considération sémantique mise à part – si l'on considère que ce mot est généralement plus souvent utilisé comme nom que comme verbe. Si l'on étudie de plus près les paramètres du HMM, on se rend compte que ce qui nous a permis d'arriver à cette conclusion est que, si ce n'était pas le cas (premier chemin), alors "modèle" serait un adjectif. Or un adjectif a peu de chance de suivre immédiatement un article (probabilité 0.1) et "modèle" n'est en définitive pas un adjectif très usité (probabilité 0.1). Ces paramètres dépendent bien évidemment du corpus de phrases étudié, et le HMM, pour être efficace, doit donc refléter le plus fidèlement possible les caractéristiques de ce corpus. C'est tout le problème

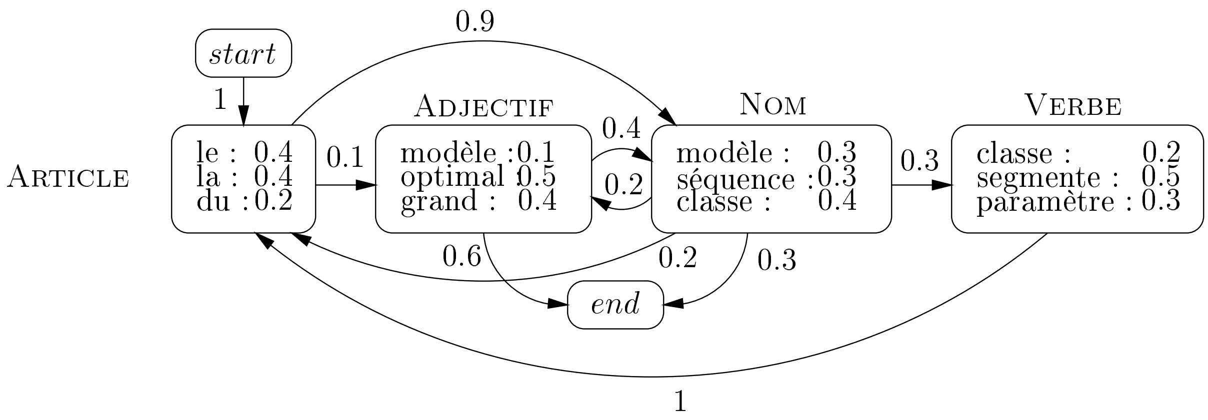


FIG. 2.2: Un HMM dont les états sont typés (ARTICLE, ADJECTIF, NOM, et VERBE) et qui peut être utilisé pour la segmentation de phrases.

du paramétrage abordé à la section 2.4. Les applications de segmentation reposent donc sur une recherche du chemin optimal et nécessitent un algorithme efficace pour résoudre ce problème. Cet algorithme est présenté à la section 2.3.2.

2.3.1 Algorithme *forward-backward*

Si l'on veut calculer la probabilité de générer la séquence de symboles $O = o_1 \dots o_T$ à l'aide du HMM $H = \langle \mathcal{Q}, \Sigma, \mathcal{T}, \mathcal{G} \rangle$, l'approche directe consiste à calculer, comme on l'a fait dans la partie précédente, la probabilité de génération pour chaque chemin possible et faire la somme de ces probabilités. La probabilité de générer O suivant le chemin $S = s_0 s_1 \dots s_T s_{T+1}$ avec $s_0 = start$ et $s_{T+1} = end$ est

$$P(O,S) = P(s_0 \to s_1)P(o_1|s_1)\dots P(s_{T-1} \to s_T)P(o_T|s_T)P(s_T \to s_{T+1}). \tag{2.1}$$

La probabilité de générer O avec H obtenue en sommant sur l'ensemble des séquences d'états possibles est donc

$$P(O|H) = \sum_{\langle s_0 \cdots s_{T+1} \rangle} P(s_0 \to s_1)P(o_1|s_1)\dots P(s_{T-1} \to s_T)P(o_T|s_T)P(s_T \to s_{T+1}), \tag{2.2}$$

avec $\langle s_0 \cdots s_{T+1} \rangle$ l'ensemble des séquences d'états possibles. Si N est le nombre d'états du HMM, alors le nombre de chemins possibles pour générer une séquence de symboles de longueur T est de l'ordre de N^T. Comme, pour chaque chemin, le calcul de la formule (2.1) demande de l'ordre de T opérations, le calcul de la probabilité de génération de O par H suivant l'équation (2.2) est en $O(TN^T)$. Un rapide calcul numérique nous montre que même pour des valeurs de T et N peu élevées, l'approche directe n'est clairement pas acceptable : avec $N = 5$ et $T = 100$ le calcul de $P(O|H)$ demande approximativement 10^{72} opérations. Heureusement une procédure bien plus efficace existe pour réaliser ce calcul : l'algorithme *forward-backward* [Rabiner, 1989].

Considérons la variable *forward* $\alpha_t(s)$ définie par

$$\alpha_t(s) = P(o_1 o_2 \cdots o_t, s_t = s|H),$$

qui exprime la probabilité d'avoir généré la séquence $o_1 \dots o_t$ en partant de l'état *start* et d'être arrivé sur l'état s à l'instant t. Cette variable peut être calculée de manière récursive :

1. Initialisation. À $t = 1$ on a :

$$\alpha_1(s) = P(start \to s) \cdot P(o_1|s).$$

2. Récursion. Pour $t = 2, \dots, T$ on a :

$$\alpha_t(s) = \left(\sum_{s' \in \mathcal{Q}} \alpha_{t-1}(s') P(s' \to s) \right) P(o_t|s). \tag{2.3}$$

Or, connaissant $\alpha_T(s)$ – c'est à dire la probabilité d'avoir généré la séquence O et d'être arrivé sur l'état s – pour tout $s \in \mathcal{Q}$, le calcul de $P(O|H)$ est immédiat :

$$P(O|H) = \sum_{s \in \mathcal{Q}} \alpha_T(s) P(s \to end). \tag{2.4}$$

On peut alors proposer une version complète de l'algorithme de programmation dynamique *forward* (voir algorithme 1).

Algorithme 1: ALGORITHME FORWARD

Données : $O = o_1 \dots o_T$ une séquence ;
H un HMM.

pour chaque $s \in \mathcal{Q}$ **faire**
  $\alpha_1(s) = P(start \to s) \cdot P(o_1|s)$;
pour chaque $t = 2$ à T **faire**
  pour chaque $s \in \mathcal{Q}$ **faire**
    Calculer et stocker la valeur de $\alpha_t(s)$ à l'aide de la formule (2.3);
Calculer $P(O|H)$ à l'aide de la formule (2.4);

Examinons la complexité de cet algorithme. La phase d'initialisation requiert une opération pour chaque état du HMM, donc au total $O(N)$ opérations. Pour la phase de récursion, pour chaque instant et chaque état, on réalise $O(N)$ opérations. Sommé sur l'ensemble des états et la totalité des instants, la phase de récursion requiert donc $O(N^2T)$ opérations. Une fois les $\alpha_T(s)$ calculés, le calcul de $P(O|H)$ suivant la formule (2.4) demande une opération pour chaque état, donc au total $O(N)$ opérations. Le calcul de $P(O|H)$ à l'aide de l'algorithme *forward* ne requiert donc au total que $O(N^2T)$ opérations. En reprenant la même application numérique que précédemment ($N = 5$ et $T = 100$), le calcul de $P(O|H)$ suivant cet algorithme nécessite approximativement 3000 opérations.

Cet algorithme est appelé *forward* car la récursion est réalisée en avant : on calcule tout d'abord la probabilité de générer le premier symbole de la séquence, puis à chaque étape de la récursion on rajoute un symbole et on réitère la procédure jusqu'à ce qu'on ait calculé la probabilité de génération de la séquence entière. Bien que moins naturel, un algorithme similaire, l'algorithme *backward*, peut être utilisé pour réaliser ce calcul à l'envers. On utilise alors la variable *backward* $\beta_t(s)$:

$$\beta_t(s) = P(o_{t+1}o_{t+2}\cdots o_T|s_t = s, H),$$

qui exprime la probabilité de générer la séquence $o_{t+1}o_{t+2}\ldots o_T$ en partant de l'état s et d'arriver sur l'état *end*. La procédure de récursion est alors la suivante :

1. Initialisation. À $t = T$ on a :

$$\beta_T(s) = P(s \to end).$$

2. Récursion. Pour $t = T - 1, \ldots, 1$ on a :

$$\beta_t(s) = \left(\sum_{s' \in \mathcal{Q}} P(s \to s')P(o_{t+1}|s')\beta_{t+1}(s')\right). \tag{2.5}$$

Et, connaissant $\beta_1(s)$ – c'est à dire la probabilité de générer la séquence O en partant de l'état s et d'arriver sur l'état *end* – pour chaque état s, le calcul de $P(O|H)$ peut alors être réalisé suivant la formule

$$P(O|H) = \sum_{s \in \mathcal{Q}} P(start \to s)P(o_1|s)\beta_1(s). \tag{2.6}$$

L'algorithme *backward* est illustré par l'algorithme 2. Sa complexité est la même que celle de l'algorithme *forward*, c'est-à-dire $O(N^2T)$.

Algorithme 2: ALGORITHME BACKWARD

Données : $O = o_1 \ldots o_T$ une séquence ;
H un HMM.

pour chaque $s \in \mathcal{Q}$ **faire**
    $\beta_T(s) = P(s \to end)$;
pour chaque $t = T - 1$ à 1 **faire**
    pour chaque $s \in \mathcal{Q}$ **faire**
        Calculer et stocker la valeur de $\beta_t(s)$ à l'aide de la formule (2.5);
Calculer $P(O|H)$ à l'aide de la formule (2.6);

2.3.2 Algorithme de Viterbi

Le problème que l'on se pose ici est de trouver, étant donnés une séquence de symboles $O = o_1 \ldots o_T$ et un HMM $H = \langle \mathcal{Q}, \Sigma, \mathcal{T}, \mathcal{G} \rangle$, la séquence d'états du HMM qui a la probabilité maximale de générer O. Ce qui nous préoccupe n'est pas la valeur de la probabilité maximale mais le chemin – appelé *chemin de Viterbi* et noté S^* – qui permet de générer la séquence O avec cette probabilité. De manière similaire à l'approche utilisée pour le calcul de $P(O|H)$, l'approche directe pour résoudre ce problème consiste à calculer la probabilité de génération suivant tous les chemins possibles et de choisir parmi ces chemins celui qui a la probabilité la plus élevée. Cette approche a, comme pour le calcul de $P(O|H)$, une complexité en $O(TN^T)$ et est donc également inapplicable. L'algorithme de Viterbi [Rabiner, 1989] est un algorithme de programmation dynamique très similaire à l'algorithme *forward* et qui permet de résoudre efficacement ce problème.

Considérons la variable $\delta_t(s)$ définie par

$$\delta_t(s) = \max_{s_0 \cdots s_{t-1}} P(s_0 \cdots s_t = s, o_1 \cdots o_t | H),$$

et qui exprime la probabilité maximale de générer la séquence $o_1 \cdots o_t$ suivant un unique chemin partant de l'état *start* et arrivant sur l'état s à l'instant t. De la même manière que pour $\alpha_t(s)$, cette variable peut être calculée de manière récursive :

1. Initialisation. À $t = 1$ on a :

$$\delta_1(s) = P(start \rightarrow s) P(o_1 | s).$$

2. Récursion. Pour $t = 2, \ldots, T$ on a :

$$\delta_t(s) = \max_{s' \in \mathcal{Q}} \left(\delta_{t-1}(s') P(s' \rightarrow s)\right) P(o_t | s). \quad (2.7)$$

Alors, connaissant $\delta_T(s)$ pour tous les états $s \in \mathcal{Q}$, on peut calculer la probabilité maximale de générer O avec H suivant un simple chemin (on désignera cette probabilité par $P(O, S^*|H)$) en appliquant la formule :

$$P(O, S^* | H) = \max_{s \in \mathcal{Q}} \left(\delta_T(s) P(s \rightarrow end)\right).$$

Malheureusement, ce n'est pas la valeur de cette probabilité qui nous intéresse mais réellement le chemin qui permet de générer O avec cette probabilité. On doit donc, à chaque étape t de la récursion et pour chaque état s, mémoriser l'état s' qui maximise l'équation (2.7). Désignons par $\psi_t(s)$ cet état. Alors :

1. À $t = 1$ on a :

$$\psi_1(s) = start.$$

2. Pour $t = 2, \ldots, T$ on a :

$$\psi_t(s) = \operatorname*{argmax}_{s' \in \mathcal{Q}} \left(\delta_{t-1}(s') P(s' \rightarrow s)\right). \quad (2.8)$$

Une fois les variables $\delta_t(s)$ et $\psi_t(s)$ calculées pour chaque étape de la récursion et pour chaque état, il ne reste plus qu'à lancer une procédure récursive de rétro-propagation pour "dérouler" le chemin de Viterbi $S^* = start, s_1^*, \ldots, s_T^*, end$ en partant de l'état *end* :

1. Initialisation. À $t = T$:

$$s_T^* = \operatorname*{argmax}_{s \in \mathcal{Q}} \left(\delta_T(s) P(s \rightarrow end)\right). \quad (2.9)$$

2. Rétro-propagation. Pour $t = T-1, T-2, \ldots, 0$:

$$s_t^* = \psi_t(s_{t+1}^*). \tag{2.10}$$

On peut alors proposer une version complète de l'algorithme de Viterbi (voir algorithme 3). On notera que, mise à part la phase de rétro-propagation, l'algorithme de Viterbi est très similaire à l'algorithme *forward*. La principale différence résulte de la maximisation des probabilités attachées aux états précédents (formule (2.7)) au lieu du calcul de la somme de ces probabilités (formule (2.3)) dans le cas de l'algorithme *forward*. La phase de rétro-propagation étant en $O(T)$, la complexité de l'algorithme de Viterbi est donc la même que celle de l'algorithme *forward*, c'est-à-dire $O(N^2T)$.

Algorithme 3: ALGORITHME DE VITERBI

Données : $O = o_1 \ldots o_T$ une séquence ;
H un HMM.

pour chaque $s \in \mathcal{Q}$ **faire**
- $\delta_1(s) = P(start \rightarrow s) \cdot P(o_1|s)$;
- $\psi_1(s) = start$;

pour chaque $t = 2$ à T **faire**
- **pour chaque** $s \in \mathcal{Q}$ **faire**
  - Calculer et stocker la valeur de $\delta_t(s)$ à l'aide de la formule (2.7);
  - Calculer et stocker la valeur de $\psi_t(s)$ à l'aide de la formule (2.8);

Déterminer l'état s_T^* à l'aide de la formule (2.9);
pour chaque $t = T-1$ à 0 **faire**
- Déterminer l'état s_t^* à l'aide de la formule (2.10);

2.4 Apprentissage

On a vu à la partie précédente les deux algorithmes classiques utilisés dans le cadre des applications de reconnaissance et de segmentation utilisant les HMMs. Ces deux algorithmes supposent que l'on dispose d'un HMM construit et paramétré de manière à modéliser de façon satisfaisante les séquences que l'on souhaite traiter. La question qui nous préoccupe dans cette partie est celle de la construction d'un tel HMM. Dans le cas le plus favorable, le HMM recherché peut être construit directement à partir des connaissances *a priori* dont on dispose sur le domaine. Ce cas est, par exemple celui de Krogh *et al.* dans leur article sur la modélisation de séquences d'ADN de *E.coli* [Krogh et al., 1994]. Dans cet article, ils utilisent les HMMs pour segmenter les séquences d'ADN en régions codantes et non-codantes. Les séquences manipulées sont des séquences d'acides nucléiques et un certain nombre de connaissances – telles que la structure générale du HMM et la fréquence des acides nucléiques en fonction de leur rôle dans les séquences – leur permettent de construire facilement leur modèle. Malheureusement ce cas de figure ne se rencontre que rarement, et, dans la plupart des applications, le ou les HMMs désirés doivent être construits à l'aide d'un algorithme d'apprentissage.

On peut distinguer dans le problème de l'apprentissage d'un HMM deux cas de figure distincts, suivant que la structure – le nombre d'états du HMM et les transitions et émissions autorisées – est connue ou ne l'est pas. Lorsque la structure est connue, le problème se réduit à un problème d'*entraînement* consistant à estimer les paramètres numériques – les distributions de probabilités de transition et de génération – de manière à expliquer au mieux les séquences d'apprentissage. Ce problème (considéré comme NP-difficile [Abe et Warmuth, 1992]) est celui traité à la section 2.4.1. Pour certaines applications, on ne dispose pas de connaissances suffisantes pour inférer naturellement la structure du HMM. L'apprentissage devient alors encore plus difficile. Il ne suffit plus de paramétrer une structure mais il faut également déduire cette structure des exemples fournis. Ce problème fait l'objet de la première partie de cette thèse. Il est exposé à la section 2.4.2 et sera plus largement développé dans les chapitres suivants.

2.4.1 Apprentissage quand la structure est connue

Nous nous intéressons dans cette partie à l'apprentissage – ou entraînement – d'un HMM à partir d'une structure connue et d'un ensemble de séquences d'apprentissage. Une approche possible est, suivant le principe du maximum de vraisemblance, de chercher les paramètres $\lambda = \langle \mathcal{T}, \mathcal{G} \rangle$ du HMM qui maximisent la probabilité de génération des séquences d'apprentissage. Soit $\mathcal{O} = \{O^1, \ldots, O^K\}$ l'ensemble des séquences d'apprentissage. On suppose que ces séquences sont indépendantes et donc que la probabilité de générer l'ensemble d'apprentissage est simplement le produit des probabilités de génération de chacune des séquences. Notre but est alors de trouver les paramètres $\lambda = \langle \mathcal{T}, \mathcal{G} \rangle$ de H qui maximisent

$$P(\mathcal{O}|H) = \prod_{k=1}^{K} P(O^k|H). \tag{2.11}$$

Une approche alternative au principe du maximum de vraisemblance est de chercher à maximiser la probabilité de génération des séquences d'apprentissage suivant leur chemin de Viterbi. Le but est alors de trouver les paramètres $\lambda = \langle \mathcal{T}, \mathcal{G} \rangle$ de H qui maximisent

$$P(\mathcal{O}, \mathcal{S}^*|H) = \prod_{k=1}^{K} P(O^k, S^{k*}|H), \tag{2.12}$$

avec S^{k*} le chemin de Viterbi de la séquence O^k dans H, et $\mathcal{S}^* = \{S^{1*}, \ldots, S^{K*}\}$. Les paramètres ne sont donc pas estimés en maximisant la *vraie* probabilité de génération des séquences d'apprentissage mais la probabilité de génération suivant les chemins les plus probables. Cette approche peut sembler moins rigoureuse que l'approche basée sur le principe du maximum de vraisemblance, mais elle est très utilisée en pratique et possède quelques arguments en sa faveur. D'abord on peut se dire que si le HMM recherché est destiné à être appliqué à des problèmes de segmentation *via* la recherche des chemins de Viterbi, il semble judicieux de l'entraîner de cette manière. Ensuite on remarque dans la pratique que la probabilité de génération d'un séquence de symboles suivant son chemin de Viterbi est en général beaucoup plus élevée que suivant n'importe quel autre chemin. Cette observation a conduit à l'hypothèse, appelée *hypothèse de Viterbi*, que tous les chemins, excepté le chemin de Viterbi, ont une probabilité nulle ou négligeable d'engendrer la séquence. Sous cette hypothèse, la probabilité de génération d'une séquence O^k par un HMM H peut être approximée par la probabilité de génération de O^k suivant son chemin de Viterbi dans H. On a alors :

$$P(\mathcal{O}|H) \approx P(\mathcal{O}, \mathcal{S}^*|H),$$

et maximiser l'équation (2.12) revient à maximiser une approximation de l'équation (2.11). Le dernier argument en faveur de cette méthode d'entraînement est que l'on peut, dans le cas de figure particulier où les chemins de Viterbi sont connus, trouver la solution optimale au sens de la maximisation de l'équation (2.12) (ce cas est plus amplement développé à la section 2.4.1.1), alors que l'on ne connaît aucune méthode optimale pour la maximisation de l'équation (2.11). Abe et Warmuth [Abe et Warmuth, 1992] ont étudié la question de savoir s'il existe un algorithme d'entraînement efficace d'un automate probabiliste, dans le cadre d'une adaptation naturelle du paradigme de *PAC-learning* introduit par Valiant [Valiant, 1984]. Leur modèle est inspiré du modèle d'apprentissage non supervisé de Laird [Laird, 1988]. La question est de savoir, pour un ensemble d'exemples de taille *réduite*, s'il existe un algorithme qui converge, avec une *forte probabilité*, vers une solution *optimale*. Dans ce cadre, ils prouvent que la classe des automates probabilistes n'est pas entraînable en temps polynômial à la taille de l'alphabet, à moins que $RP = NP$. Pour les HMMs, la question reste ouverte à notre connaissance, mais on peut raisonnablement penser que le problème est aussi difficile.

Suivant la méthode d'estimation des paramètres choisie, il existe deux heuristiques permettant l'entraînement des HMMs. Ces deux algorithmes, très similaires, sont tous deux issus d'une méthode générale – les algorithmes d'*Expectation-Maximization* (EM)[1] [Dempster et al., 1977] – servant à l'estimation des paramètres d'une grande famille de modèles probabilistes. Pour des raisons de simplicité, nous commencerons par exposer l'algorithme adapté à la maximisation de l'équation (2.12) (section 2.4.1.1), et nous verrons à la section 2.4.1.2 l'algorithme servant à maximiser l'équation (2.11).

2.4.1.1 Entraînement de Viterbi

Comme évoqué dans la partie précédente, il existe un cas où il est possible de paramétrer la structure du HMM de manière à maximiser l'équation (2.12) de façon optimale. Ce cas est celui rencontré lorsque l'on connait les chemins de Viterbi des séquences d'apprentissage dans la structure. En effet, on peut alors associer à chaque état, chaque transition

[1]En fait, l'algorithme d'entraînement adapté à la maximisation de l'équation (2.12) n'est pas à proprement parlé un algorithme de type *EM*, mais une adaptation de cet algorithme qui remplace la phase *E* par la recherche des chemins de Viterbi.

et chaque symbole attaché aux états du HMM le nombre de fois où ils sont utilisés pour générer ces séquences. Soit n_s, $n_{s\to s'}$ et n_s^o respectivement le nombre de fois où l'état s est utilisé, le nombre de fois où la transition $s \to s'$ est utilisée et le nombre de fois où le symbole o est généré par l'état s dans les chemins de Viterbi. Alors la formule (2.12) peut être réécrite

$$P(\mathcal{O}, \mathcal{S}^*|H) = \prod_{s\in\mathcal{Q}} \left(\prod_{o\in\Sigma} P(o|s)^{n_s^o} \prod_{s'\in\mathcal{Q}} P(s \to s')^{n_{s\to s'}} \right). \tag{2.13}$$

Maximiser cette formule revient à maximiser indépendamment chacun de ses sous-produits. On peut donc estimer les paramètres $\mathcal{T}$ en maximisant l'expression $\prod_{s'\in\mathcal{Q}} P(s \to s')^{n_{s\to s'}}$ et les paramètres $\mathcal{G}$ en maximisant $\prod_{o\in\Sigma} P(o|s)^{n_s^o}$. Les estimateurs de $\mathcal{T}$ et $\mathcal{G}$ sont donc respectivement

$$\widehat{P}(s \to s') = \frac{n_{s\to s'}}{n_s} \tag{2.14}$$

et

$$\widehat{P}(o|s) = \frac{n_s^o}{n_s}. \tag{2.15}$$

Cette méthode d'estimation n'est possible que dans le cas favorable où les chemins de Viterbi sont connus. Lorsque ce n'est pas le cas, le problème est évidemment plus difficile et l'algorithme d'entraînement de Viterbi peut alors être une solution.

L'algorithme d'entraînement de Viterbi (voir algorithme 4) est un algorithme de réestimation itératif. Il consiste, à partir d'un paramétrage initial du HMM, à calculer les chemins de Viterbi des séquences d'apprentissage à l'aide de l'algorithme de Viterbi décrit à la section 2.3.2. Les chemins de Viterbi son utilisés pour calculer, comme on l'a fait ci-dessus, le nombre de fois où chaque transition, chaque état et chaque symbole attaché aux états est utilisé. On réestime alors à l'aide des formules (2.14) et (2.15) les paramètres du HMM, on reparamètre la structure à l'aide de ces estimations et on réitère la procédure jusqu'à obtention de la stabilité.

On peut montrer que la valeur de $P(\mathcal{O}, \mathcal{S}^*|H)$ augmente à chaque itération et que l'algorithme converge vers un optimum local [Juang et Rabiner, 1990]. Malheureusement, il existe généralement un grand nombre d'optima locaux et le paramétrage obtenu par ce processus dépend fortement du paramétrage initial choisi.

Examinons la complexité de cet algorithme. À chaque tour, il faut calculer le chemin de Viterbi de chacune des K séquences d'apprentissage. Si on note T la taille totale des K séquences, ce calcul se fait en $O(N^2T)$. La réestimation des paramères du HMM à l'aide des formules (2.14) et (2.15) se fait en $O(N^2)$. En bornant le nombre de tours de l'algorithme par une constante L (en pratique, une dizaine de tours suffit généralement à assurer la convergence de l'algorithme), la complexité totale de l'algorithme d'entraînement de Viterbi est donc $O(LN^2T)$.

Algorithme 4: Algorithme d'entraînement de Viterbi

Données : $\mathcal{O} = \{O^1, \ldots, O^K\}$;
H une structure de HMM.

Choisir un paramétrage initial $\lambda = \langle \mathcal{T}, \mathcal{G} \rangle$ de H;
répéter
  Initialiser toutes les variables n_s, $n_{s \to s'}$ et n_s^o à 0;
  pour chaque $O^k \in \mathcal{O}$ **faire**
    Calculer le chemin de Viterbi de O^k dans H;
    Ajouter aux variables n_s, $n_{s \to s'}$ et n_s^o la contribution de O^k;
  Réestimer les paramètres λ de H en utilisant les formules (2.14) et (2.15);
jusqu'à stabilité du paramétrage;

2.4.1.2 Entraînement de Baum-Welch

L'algorithme d'entraînement de Baum-Welch estime les paramètres $\lambda = \langle \mathcal{T}, \mathcal{G} \rangle$ du HMM en maximisant l'équation (2.11). C'est un algorithme de réestimation itératif qui fonctionne sur le même principe que l'algorithme d'entraînement de Viterbi. Dans ce dernier, on compte le nombre de fois où chaque état, chaque transition et chaque symbole attaché aux état est utilisé dans les chemins de Viterbi. Ici on veut maximiser les probabilités de génération réelles, et non celles des chemins les plus probables. On va donc associer au états, aux transitions et aux symboles le nombre de fois où ils sont utilisés pour toutes les séquences et *tous* les chemins susceptibles de générer les séquences, pondéré par la probabilité de chacun de ces chemins. Ces comptes pondérés sont alors utilisés pour réestimer les paramètres du modèle de la même manière que pour l'algorithme d'entraînement de Viterbi.

De manière plus formelle, considérons pour commencer une unique séquence d'observations $O = o_1 \ldots o_T$. Notre but est de trouver les paramètres $\lambda = \langle \mathcal{T}, \mathcal{G} \rangle$ qui maximisent la probabilité $P(O|H)$ de générer O avec H. Nous verrons ensuite comment étendre l'algorithme à un ensemble de séquences d'observations. Soit

$$\xi_t(s, s') = P(s_t = s, s_{t+1} = s'|O, H)$$

la probabilité qu'en générant O avec H on passe par l'état s à l'instant t et par l'état s' à l'instant $t+1$. Cette équation peut se réécrire :

$$\xi_t(s, s') = \frac{P(s_t = s, s_{t+1} = s', O|H)}{P(O|H)},$$

et en utilisant les variables *forward* et *backward* introduites à la section 2.3.1 :

$$\xi_t(s, s') = \frac{\alpha_t(s) P(s \to s') P(o_{t+1}|s') \beta_{t+1}(s')}{P(O|H)}. \tag{2.16}$$

Considérons également la variable $\gamma_t(s)$:

$$\gamma_t(s) = P(s_t = s|O, H),$$

qui exprime la probabilité qu'en générant O avec H on on se trouve sur l'état s à l'instant t. Exprimée en fonction de $\xi_t(s,s')$, $\gamma_t(s)$ s'écrit :

$$\gamma_t(s) = \sum_{s' \in \mathcal{Q}} \xi_t(s,s'). \tag{2.17}$$

Si on somme $\gamma_t(s)$ sur l'ensemble des instants t, on obtient une quantité que l'on peut interpréter comme l'espérance du nombre de fois où l'état s est utilisé pour générer la séquence O. De même, si on somme $\xi_t(s,s')$ sur l'ensemble des instants t, on obtient une quantité que l'on peut interpréter comme l'espérance du nombre de fois où la transition $s \to s'$ est utilisée pour générer la séquence O. On a donc :

$$\sum_{t=0}^{T} \gamma_t(s) = \text{espérance du nombre de passages dans } s, \tag{2.18}$$

$$\sum_{t=0}^{T} \xi_t(s,s') = \text{espérance du nombre de transitions de } s \text{ vers } s', \tag{2.19}$$

et de manière similaire à (2.18) on a également :

$$\sum_{t=1 \,/\, o_t=o}^{T} \gamma_t(s) = \text{espérance du nombre de générations de } o \text{ dans } s. \tag{2.20}$$

En combinant les formules (2.18), (2.19) et (2.20) on peut alors proposer un ensemble d'estimateurs des paramètres $\lambda = \langle \mathcal{T}, \mathcal{G} \rangle$ du HMM :

$$\begin{aligned} \widehat{P}(s \to s') &= \frac{\text{espérance du nombre de transitions de } s \text{ vers } s'}{\text{espérance du nombre de passages dans } s} \\ &= \frac{\sum_{t=0}^{T} \xi_t(s,s')}{\sum_{t=0}^{T} \gamma_t(s)} \end{aligned}$$

$$\begin{aligned} \widehat{P}(o|s) &= \frac{\text{espérance du nombre de générations de } o \text{ dans } s}{\text{espérance du nombre de passages dans } s} \\ &= \frac{\sum_{t=1 \,/\, o_t=o}^{T} \gamma_t(s)}{\sum_{t=0}^{T} \gamma_t(s)} \end{aligned}$$

Dans le cas où l'on dispose d'un ensemble de séquences d'observations $\mathcal{O} = \{O^1, \ldots, O^K\}$, ces estimateurs se généralisent simplement de la manière suivante :

$$\widehat{P}(s \to s') = \frac{\sum_{k=1}^{K} \sum_{t=0}^{T} \xi_t^k(s,s')}{\sum_{k=1}^{K} \sum_{t=0}^{T} \gamma_t^k(s)} \tag{2.21}$$

$$\widehat{P}(o|s) = \frac{\sum_{k=1}^{K} \sum_{t=1 \,/\, o_t=o}^{T} \gamma_t^k(s)}{\sum_{k=1}^{K} \sum_{t=0}^{T} \gamma_t^k(s)} \tag{2.22}$$

On peut alors proposer une version complète de l'algorithme de Baum-Welch (voir algorithme 5). Comme pour l'algorithme d'entraînement de Viterbi, on peut prouver [Baum et al., 1970] que la probabilité de génération des séquences $P(\mathcal{O}|H)$ augmente à chaque itération et que l'algorithme converge vers un optimum local. Néanmoins, dans ce cas là, l'espace de recherche est un espace continu (on n'utilise pas des comptes discrets comme pour l'algorithme d'entraînement de Viterbi, mais l'espérance de ces comptes), et l'optimum local n'est donc en réalité jamais atteint [Durbin et al., 1998]. Il est alors nécessaire de définir un critère d'arrêt, par exemple en stoppant la procédure lorsque l'augmentation de $P(\mathcal{O}|H)$ est inférieure à un certain seuil, ou lorsque le nombre d'itérations effectuées est jugé suffisamment important.

Éxaminons la complexité de cet algorithme. Le calcul des variables $\alpha_t^k(s)$ à l'aide de l'algorithme *forward* est en $O(N^2T)$. Celui des $\beta_t^k(s)$ à l'aide de l'algorithme *backward* est également en $O(N^2T)$. Le calcul des $\xi_t^k(s,s')$ à l'aide de la formule (2.16) se fait en $O(N^2)$. Le calcul des $\gamma_t^k(s)$ à l'aide de la formule (2.17) se fait également en $O(N^2)$. La complexité totale du calcul des expressions (2.18), (2.19) et (2.20) est $O(N^2T)$. Chacune de ces opérations est à faire à chaque tour de l'algorithme et pour chaque séquence de $\mathcal{O}$. En bornant le nombre de tours par une constante L, et en notant T la taille totale des K séquences d'apprentissage, on obtient une complexité totale de l'algorithme d'entraînement de $O(LN^2T)$.

Algorithme 5: ALGORITHME D'ENTRAÎNEMENT DE BAUM-WELCH

Données : $\mathcal{O} = \{O^1, \ldots, O^n\}$;
H une structure de HMM.

Choisir un paramétrage initial $\lambda = \langle \mathcal{T}, \mathcal{G} \rangle$ de H;
répéter
  pour chaque $O^k \in \mathcal{O}$ **faire**
    Calculer les valeurs des variables $\alpha_t^k(s)$ à l'aide de l'algorithme *forward*;
    Calculer les valeurs des variables $\beta_t^k(s)$ à l'aide de l'algorithme *backward*;
    Calculer les valeurs des variables $\xi_t^k(s,s')$ à l'aide de la formule (2.16);
    Calculer les valeurs des variables $\gamma_t^k(s)$ à l'aide de la formule (2.17);
    Calculer et stocker les valeurs des expressions (2.18), (2.19) et (2.20);
  Réestimer les paramètres λ de H en utilisant les formules (2.21) et (2.22);
jusqu'à critère d'arrêt;

2.4.2 Apprentissage quand la structure est inconnue

On a vu à la partie précédente les algorithmes d'entraînement classiques à partir d'une structure connue. Pour certaines applications, néanmoins, aucune connaissance ne permet d'inférer naturellement cette structure (on en verra par exemple un cas dans la partie II de ce document). D'autre part, comme le soulignent Freitag et McCallum [Freitag et McCallum, 2000], la construction "à la main" de HMMs n'est pas aisée. Enfin, il est à noter que l'intuition humaine ne correspond pas toujours à la structure qui tire le mieux parti du potentiel des HMMs. Dans le même article [Freitag et McCallum, 2000], Freitag et McCallum montrent expérimentalement sur un ensemble de problèmes d'extraction d'informations à partir de données textuelles, que les modèles construits à partir des données donnent généralement de meilleurs résultats que les modèles construits à la main et entraînés à l'aide de l'algorithme de Baum-Welch. Le problème est alors, non seulement de paramétrer une structure, mais encore d'inférer cette structure à partir des exemples d'apprentissages. Plusieurs méthodes ont été proposées pour résoudre ce problème. Remarquons tout d'abord qu'une approche du type maximum de vraisemblance n'est clairement pas suffisante. En effet, il n'est pas difficile de construire un HMM qui a la probabilité maximale de générer les séquences d'apprentissage. Il suffit de construire un HMM équivalent à une disjonction de ces séquences : chaque chemin du *start* vers le *end* représente exactement une séquence d'apprentissage ; chaque état ne génère qu'un des symboles de l'alphabet et n'a – mis à part l'état *start* – qu'une transition sortante (voir l'exemple de la figure 2.3). Les éventuelles séquences qui apparaissent plusieurs fois dans l'ensemble d'apprentissage sont représentées par un unique chemin, et on assigne aux transitions sortantes de l'état *start* une probabilité égale à la fréquence de la séquence dans l'ensemble d'apprentissage. On obtient alors un HMM qui assigne à chaque séquence sa fréquence d'apparition et maximise donc la vraisemblance de l'ensemble d'apprentissage. Le problème de ce HMM est qu'il est spécifique aux séquences d'apprentissage – il ne peut générer que ces séquences – et est donc inutilisable en pratique puisque incapable de reconnaître d'autres séquences. On se trouve alors face à un problème classique en apprentissage, celui de la recherche d'un *biais inductif*, c'est-à-dire d'une règle de généralisation *a priori* permettant d'inférer des HMMs qui puissent être utilisés sur de nouvelles séquences.

La solution la plus simple pour résoudre ce problème consiste à entraîner un HMM avec un nombre d'états fixé et un nombre important de transitions – voire un modèle entièrement connecté – à l'aide de l'algorithme de Baum-Welch ou une de ses variantes, et de laisser le soin à cet algorithme de déterminer la structure du modèle, c'est-à-dire les transitions et émissions de probabilité non nulle. Il apparaît cependant, comme nous le verrons au chapitre 6, que cette solution ne permet pas toujours l'apprentissage de HMMs de bonne qualité. Diverses solutions mieux adaptées à ce problème ont donc été proposées dans la littérature. Nous en décrivons brièvement quelques-unes dans la suite de cette section.

2.4.2.1 Apprentissage par généralisation

Nous n'abordons ici que très brièvement cette technique d'apprentissage qui sera plus amplement détaillée dans les chapitres suivants. L'apprentissage d'un HMM par généralisation, introduit par Stolcke et Omohundro [Stolcke et Omohundro, 1994a], procède de manière gloutonne et ascendante. Elle consiste à partir d'un HMM H_0 spécifique – construit comme exposé ci-dessus – et, par une opération élémentaire de *fusion d'états* (voir figure 2.4), de transformer progressivement ce HMM en un modèle plus simple et plus général. Ce gain en généralité s'accompagne généralement d'une baisse de la vraisemblance des séquences d'apprentissage. Cette méthode construit ainsi une suite de HMMs

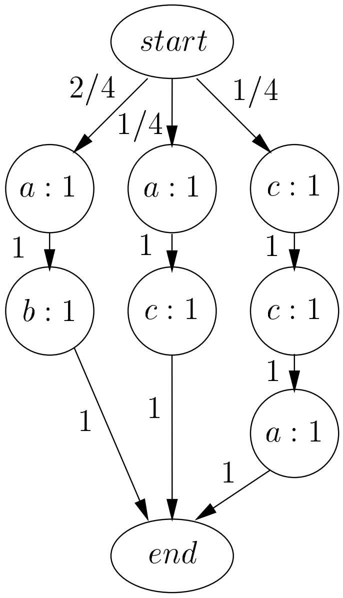


FIG. 2.3: HMM de vraisemblance maximale construit à partir de l'ensemble de séquences d'apprentissage $\{ab, ac, ab, cca\}$.

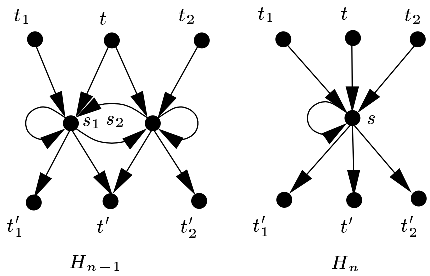


FIG. 2.4: Structure avant et après la fusion des états s_1 et s_2.

$H_0, H_1, \cdots, H_n$ où H_n est le HMM obtenu en fusionnant deux états de H_{n-1}. Après chaque fusion, les paramètres de la structure sont réestimés de manière à maximiser la vraisemblance des séquences d'apprentissage.

Différentes techniques peuvent être utilisées pour guider la fusion au cours de l'apprentissage. Il faut : 1) choisir un couple d'états à fusionner à chaque itération, 2) définir un critère d'arrêt pour l'algorithme. La solution la plus simple consiste à choisir, à chaque itération, le couple d'états dont la fusion va engendrer la plus petite perte de vraisemblance des séquences d'apprentissage. Stolcke et Omohundro [Stolcke et Omohundro, 1994a] proposent une solution alternative en utilisant un biais s'exprimant comme une probabilité *a priori* sur l'ensemble des HMMs possibles – cette probabilité est construite de manière à favoriser les structures de tailles réduites – et leur critère de fusion est alors la maximisation d'une probabilité *a posteriori*. La procédure est stoppée lorsque plus aucun couple ne permet d'augmenter cette probabilité. Leur critère de fusion et d'arrêt définit donc un compromis entre généralisation et vraisemblance des séquences d'apprentissage.

Seymore *et al.* [Seymore et al., 1999] utilisent également une approche de ce type pour l'apprentissage de HMMs destinés à la segmentation des entêtes d'articles scientifiques. Leur critère de fusion n'est pas basé sur des probabilités mais sur certaines relations topologiques entre les états. Ils proposent deux règles de fusion simples : la première, *neighbor-merging*, fusionne les états du même type reliés par une transition ; la seconde, *V-merging*, fusionne deux états du même type ayant un voisin commun.

2.4.2.2 Apprentissage par spécialisation

À l'inverse de l'apprentissage par généralisation, l'apprentissage par spécialisation consiste à partir d'un HMM très général – par exemple composé d'un unique état qui boucle sur lui-même et des états *start* et *end* –, et par une opération de spécialisation – la création d'un nouvel état, ou l'ajout d'une transition entre deux états existants – de construire progressivement un HMM plus spécifique de manière à augmenter la vraisemblance des séquences d'apprentissage. Freitag et McCallum [Freitag et McCallum, 2000] utilisent une approche de ce type pour construire des HMMs permettant d'isoler certains champs à partir de données textuelles. Ils définissent à la fois un type de structure approprié à leur domaine d'application et un ensemble de règles de spécialisation qui permettent, à partir d'un HMM général, la construction incrémentale de HMMs plus spécifiques et conformes à leur type de structure. À chaque itération, leur algorithme construit et paramètre à l'aide de l'algorithme de Baum-Welch, tous les HMMs qui peuvent être composés à l'aide d'une des règle de spécialisation appliquée sur le HMM courant. Le meilleur HMM obtenu – au sens d'un critère basé sur le taux d'erreur de prédiction calculé sur un échantillon indépendant des séquences d'apprentissage – est alors stocké et devient le HMM courant. À l'issue d'un nombre d'itérations prédéfini, l'algorithme renvoie le meilleur HMM parmi tous les HMMs stockés.

D'autres approches peuvent être utilisées pour l'apprentissage de la structure des HMMs. Brand [Brand, 1999] propose par exemple une variante de l'algorithme de Baum-Welch dont l'objet est la maximisation, non pas de la vraisemblance des séquences d'apprentissage comme c'est le cas dans l'algorithme classique, mais d'une probabilité *a posteriori* construite à l'aide d'une probabilité *a priori* de forme entropique. Cette probabilité *a priori* est fonction des paramètres du HMM – probabilités de transition et d'émission – et sa forme entropique favorise les probabilités proches des extrêmes (0 ou 1). Alliée à une opération d'extinction – suppression des transitions et des symboles qui ont une probabilité trop faible –, elle permet de construire, à partir d'un HMM initial fortement connexe ayant un nombre d'états important, des HMMs de tailles réduites et avec peu de transitions.

2.4.3 Problème des probabilités faibles

Comme pour beaucoup de problèmes d'apprentissage – et comme on l'a vu pour les unigrams –, un large corpus d'apprentissage est nécessaire pour apprendre un modèle qui soit suffisamment général. Ce problème est d'autant plus important que l'alphabet manipulé est étendu. Avec des alphabets de taille importante, les méthodes d'estimation des probabilités d'émission par maximum de vraisemblance conduisent à des estimateurs biaisés par les probabilités nulles, ce qui pose un problème crucial lors de l'utilisation des HMMs sur de nouvelles séquences. L'utilisation de probabilités *a priori*, en conjonction avec les techniques classiques d'estimation, permet de résoudre ce problème.

Comme pour les unigrams, la technique de *Laplace smoothing*, peut être utilisée [Freitag et McCallum, 1999]. Elle consiste, pour chaque état du HMM, à ajouter une constante l au compte du nombre de fois – ou de l'espérance du nombre de fois – où chaque symbole de l'alphabet est généré par l'état. Les estimateurs des expressions (2.15) et (2.22) deviennent alors respectivement :

$$\widehat{P}(o|s) = \frac{n_s^o + l}{n_s + l \cdot |\Sigma|},$$

et

$$\widehat{P}(o|s) = \frac{\sum_{k=1}^{K} \sum_{t=1\ /\ o_t=o}^{T} \gamma_t^k(s) + l}{\sum_{k=1}^{K} \sum_{t=0}^{T} \gamma_t^k(s) + l \cdot |\Sigma|}.$$

Une alternative à cette méthode qui peut être utilisée lorsque le nombre de symboles d'occurence nulle varie fortement suivant les états, consiste à soustraire une constante d (avec $0 < d < 1$) aux comptes – supérieurs à 0 – de chaque symbole [Freitag et McCallum, 1999]. La masse de probabilités ainsi libérée est alors redistribuée aux symboles d'occurence nulle. Les estimateurs de l'expression (2.15) se réécrivent alors :

$$\widehat{P}(o|s) = \begin{cases} \frac{n_s^o - d}{n_s}, \text{si } n_s^o > 0 \\ \frac{d \cdot (|\Sigma| - |Z_s|)}{|Z_s|}, \text{si } n_s^o = 0, \end{cases}$$

avec $|Z_s|$ le nombre de symboles d'occurence nulle dans l'état s.

Ces deux techniques estiment les distributions de probabilités de génération des états en utilisant seulement les données d'apprentissage attachées aux états. Lorsque les états du HMM sont typés (généralement pour faire de la segmentation) des techniques plus fines peuvent être utilisées. Ces techniques, appelées techniques de *shrinkage*, consistent à définir une hiérarchie sur les types des états [Freitag et McCallum, 1999]. La distribution de probabilités de génération associée à un état est alors estimée par un mélange pondéré de la distribution définie par les symboles générés par l'état, de la distribution définie par les symboles générés par les états du même type, et des distributions définies par les symboles générés par les états de type supérieur dans la hiérarchie.

2.5 Conclusion

Il est intéressant de remarquer que les HMMs se sont imposés comme le modèle de référence pour la résolution de certains types de problèmes dans plusieurs domaines d'application, que ce soit en reconnaissance de la parole, en modélisation de séquences biologiques ou pour l'extraction d'information à partir de données textuelles. La lisibilité du modèle et surtout la simplicité et l'efficacité de ses principaux algorithmes sont très certainement responsables de ce succès.

Nous avons présenté ici une vue d'ensemble des HMMs, de leurs applications et des algorithmes classiques utilisés dans la littérature : les algorithmes de calcul de la probabilité de génération d'une séquence par un HMM et l'algorithme de recherche du chemin optimal. Bien d'autres sujets auraient pu être abordés, comme les différents types de structure qui peuvent être utilisées [Rabiner, 1989], les problèmes liés à l'implémentation des algorithmes (notamment les problèmes d'arrondis posés par les calculs de probabilité) [Rabiner, 1989], ou les méthodes de comparaison des HMMs [Juang et Rabiner, 1985]. Concernant le problème de l'apprentissage des HMMs, nous avons présenté les deux algorithmes d'entraînement généralement utilisés lorsque la structure du modèle est connue (l'algorithme de Baum-Welch et l'algorithme d'entraînement de Viterbi), et nous avons également évoqué les différentes approches qui peuvent être utilisées pour l'apprentissage dans le cadre particulier où aucune connaissance ne permet d'inférer naturellement la structure. Ce dernier problème constitue le sujet principal de la première partie de cette thèse. Les chapitres suivants lui sont dédiés. Le chapitre 3 commence par détailler les algorithmes d'apprentissage par généralisation ou fusions d'états brièvement abordés dans le présent chapitre. Nous présentons l'algorithme original proposé par Stolcke et Omohundro [Stolcke et Omohundro, 1994a], ainsi qu'une version simplifiée de cet algorithme que ces auteurs ont simplement évoquée sans la détailler complètement. Cette version diffère de la version originale uniquement par la procédure de sélection des états à fusionner utilisée. Nous présentons également dans ce chapitre l'algorithme ALERGIA [Carrasco et Oncina, 1994], qui, comme celui de Stolcke et Omohundro, procède par fusions d'états, mais qui est utilisé dans le cadre des automates probabilistes. Au chapitre suivant (chapitre 4), nous proposons une implémentation efficace de la procédure de sélection des états à fusionner dans sa version simplifiée. Nous présentons ensuite au chapitre 5 un nouveau critère de sélection plus élaboré que les critères utilisés jusqu'à présent (le critère de Stolcke et Omohundro et le critère simplifié) et qui s'inspire du critère de sélection de l'algorithme ALERGIA. Nous proposons alors un nouvel algorithme d'apprentissage par fusions d'états utilisant ce critère. Dans le dernier chapitre de cette partie, nous comparons les performances des différents algorithmes d'apprentissage de HMMs sur trois problèmes différents.

Chapitre 3

Algorithmes d'apprentissage par fusions d'états

Nous avons abordé au chapitre précédent quelques techniques d'apprentissage des HMMs. Nous allons détailler ici l'une des techniques qu'il est possible d'utiliser lorsque la structure est inconnue : l'apprentissage par généralisation, ou fusions d'états. Comme nous l'avons déjà mentionné, la fusion d'états est un principe qui a été utilisé aussi bien dans le cadre de l'apprentissage de HMMs que dans celui des automates d'états finis déterministes probabilistes [Carrasco et Oncina, 1994] et non probabilistes [Oncina et Garcia, 1992], [Lang, 1992], [de la Higuera et al., 1996]. Pour cette dernière classe, cette technique s'est d'ailleurs imposée comme la technique de référence lors du concours d'inférence d'automates *Abbadingo One DFA Learning Competition* [Lang et al., 1998]. La fusion d'états a donc donné naissance à différents algorithmes, suivant les classes de modèles auxquelles elle est appliquée. Du fait des liens étroits qui existent entre HMMs et automates probabilistes, il serait incomplet de présenter les algorithmes d'apprentissage de HMMs sans évoquer au moins brièvement ceux dédiés à l'apprentissage des automates. La première section présente donc l'algorithme de base développé par Stolcke et Omohundro [Stolcke et Omohundro, 1994a] dans le cadre des HMMs, et à la section suivante, nous nous intéressons à l'algorithme ALERGIA introduit par Carrasco et Oncina [Carrasco et Oncina, 1994] pour l'apprentissage d'automates d'états finis déterministes et probabilistes.

3.1 Cadre des HMMs

L'apprentissage de HMMs par fusions d'états a été introduite par Stolcke et Omohundro dans [Stolcke et Omohundro, 1994a] et [Stolcke et Omohundro, 1994b]. Ce travail fait suite à un article antérieur de Omohundro [Omohundro, 1992] où il est question d'apprentissage par fusions d'états dans le cadre des réseaux de neurones. Stolcke et Omohundro proposent un algorithme d'apprentissage général de HMMs (voir algorithme 6) dont le principe est de partir d'un modèle spécifique aux séquences d'apprentissage, et de généraliser itérativement ce modèle grâce à une opération de fusion d'états, jusqu'à obtenir un modèle suffisamment simple ou général. L'idée de base de cet algorithme est de construire progressivement le modèle souhaité en se guidant, à chaque itération, sur le

Algorithme 6: ALGORITHME GÉNÉRAL D'APPRENTISSAGE PAR FUSIONS D'ÉTATS.

Données : $\mathcal{O}$ un ensemble de séquences d'apprentissage

Créer un HMM initial qui maximise la vraisemblance de $\mathcal{O}$;
répéter
  Sélectionner un couple d'états (u, u');
  Fusionner u et u' et réestimer les paramètres du HMM;
jusqu'à critère d'arrêt;

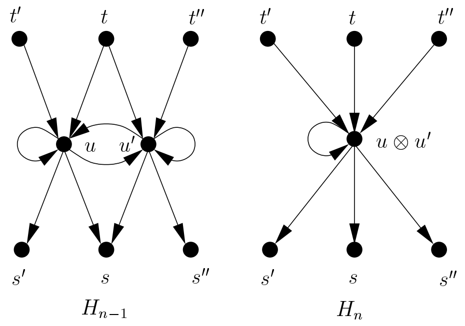


FIG. 3.1: Structure du HMM avant et après la fusion des états u et u'. Le nouvel état est noté $u \otimes u'$.

rapport entre la perte de vraisemblance occasionnée par une fusion et le gain en généralité obtenu. Cette approche très simple exige la spécification de plusieurs opérations. Il faut :

1. une méthode de construction du modèle initial ;
2. une fonction de fusion de deux états ;
3. une fonction de sélection des états à fusionner à chaque itération ;
4. un critère d'arrêt.

Construction du HMM initial Le HMM initial est un HMM pour lequel la vraisemblance des données d'apprentissage est maximale. On a vu au chapitre précédent qu'il est facile d'obtenir un tel modèle, simplement en construisant une disjonction des séquences d'apprentissage (voir figure 2.3 page 37). Ce HMM assigne à chaque séquence une probabilité égale à sa fréquence dans l'ensemble d'apprentissage et maximise donc la vraisemblance de cet ensemble.

Fusion de deux états La fonction de fusion de deux états est l'opération élémentaire. Elle doit préserver la capacité de générer toutes les séquences d'apprentissage, mais peut également entraîner la possibilité de générer de nouvelles séquences. Elle définit donc une opération de généralisation. Cette opération se décompose en deux phases. La première phase est la construction de la structure résultant de la fusion. Pour cela, on supprime les deux états sélectionnés et on les remplace par un nouvel état (voir figure 3.1). Les transitions entrantes et sortantes des anciens états sont connectées de la même manière au nouvel état, et les éventuelles transitions doubles consécutives à la fusion – par exemple les transitions $t \to u$ et $t \to u'$ de la figure 3.1 – sont également fusionnées. La seconde phase de cette opération est la réestimation des paramètres de la structure obtenue. Ces paramètres sont estimés, suivant la technique classique, en maximisant la vraisemblance des données d'apprentissage. Toute la question est alors de choisir entre maximiser la vraisemblance totale – calculée grâce à l'expression (2.11) – ou la vraisemblance suivant les chemins de Viterbi des séquences – expression (2.13). Ce point fait l'objet de la section 3.1.2.

Sélection des états à fusionner La fonction de sélection des états à fusionner est la fonction la plus sensible et la plus délicate à définir. Plusieurs solutions sont envisageables. La solution la plus naturelle, qui est proposée par Omohundro dans son article antérieur [Omohundro, 1992] sur l'apprentissage de réseaux de neurones mais qui est simplement évoquée par Stolcke et Omohundro dans [Stolcke et Omohundro, 1994a], consiste à choisir le couple d'états dont la fusion engendre la plus petite perte de vraisemblance des données. Stolcke et Omohundro lui préfèrent une solution alternative qui utilise un biais s'exprimant comme une probabilité *a priori* sur l'ensemble des HMMs possibles. Leur critère de fusion est alors la maximisation d'une probabilité *a posteriori* construite suivant le théorème de Bayes :

$$P(H|\mathcal{O}) = \frac{P(H)P(\mathcal{O}|H)}{P(\mathcal{O})}, \tag{3.1}$$

où $P(H)$ est la probabilité *a priori* du HMM H, $P(\mathcal{O}|H)$ est la probabilité de générer les séquences d'apprentissage $\mathcal{O}$ avec H, et $P(\mathcal{O})$ est la probabilité d'observer les séquences $\mathcal{O}$. Sachant qu'un HMM H se définit par sa structure (ou topologie) S_H (les états, transitions et symboles attachés aux états) et les paramètres λ_H qui étiquettent cette structure (les différentes probabilités associées), ils proposent de décomposer la probabilité *a priori* de H de la manière suivante :

$$P(H) = P(S_H)P(\lambda_H|S_H),$$

avec $P(S_H)$ la probabilité *a priori* de la structure, et $P(\lambda_H|S_H)$ celle du paramétrage. De manière à favoriser les structures de tailles réduites, ils proposent d'exprimer $P(S_H)$ à l'aide d'une fonction du type

$$P(S_H) = \exp(-T(H)),$$

où $T(H)$ est la taille du HMM calculée en fonction du nombre d'états, de transitions et de symboles associés aux états. D'un autre côté, les probabilités $P(\lambda_H|S_H)$ sont définies à l'aide de distributions de Dirichlet pour chacune des distributions de transition et d'émission associées aux états du modèle. À chaque itération, ils choisissent alors le couple d'états dont la fusion engendre l'augmentation la plus importante de la probabilité *a posteriori* $P(H|\mathcal{O})$ – ou plus simplement du produit $P(H)P(\mathcal{O}|H)$, puisque $P(\mathcal{O})$ est une constante indépendante du HMM. Dans certains cas, des contraintes dépendantes du domaine d'application peuvent également guider le choix des fusions, ou tout du moins diminuer l'ensemble des fusions potentielles à considérer. En reconnaissance de la parole, par exemple, les HMMs utilisés sont souvent des HMMs gauche-droite [Rabiner, 1989]. Lors de l'apprentissage de tels HMMs, seules les fusions non créatrices de circuits sont alors autorisées.

Critère d'arrêt de l'algorithme Le critère d'arrêt est également un point sensible de l'algorithme et est fortement lié à la fonction de sélection des états adoptée. Si cette fonction est celle qui minimise la perte de vraisemblance des données, un critère possible est de stopper la procédure lorsque la vraisemblance des séquences est jugée trop faible, ou lorsque la baisse de vraisemblance occasionnée par la meilleure fusion possible est trop importante. Un meilleur critère peut être d'utiliser un ensemble de séquences indépendantes de l'ensemble d'apprentissage, de calculer la vraisemblance de ces séquences à chaque étape de l'algorithme, et de stopper la procédure lorsque la vraisemblance diminue. La fonction de sélection des états à fusionner proposée par Stolcke et Omohundro [Stolcke et Omohundro, 1994a] impose pour sa part un critère d'arrêt naturel. Puisque le but est alors la maximisation d'une probabilité *a posteriori*, la procédure est stoppée lorsque plus aucun couple ne permet d'augmenter cette probabilité. Leur critère de fusion et d'arrêt définit donc un compromis entre généralisation et vraisemblance des séquences d'apprentissage. Comme pour la procédure de sélection des états, les spécificités du domaine d'application peuvent également guider, voire imposer, le critère d'arrêt de l'algorithme. Dans la deuxième partie de ce document, par exemple, nous utilisons les HMMs pour construire un générateur de séquences servant au test de circuits microélectroniques. Ce générateur est destiné à être physiquement implémenté sur le circuit. Il est donc nécessaire que le HMM inféré ne dépasse pas une certaine taille, ce qui nous impose un critère d'arrêt naturel de l'algorithme.

Comme nous avons pu nous en apercevoir, la clef de l'algorithme réside dans le critère de sélection des états à fusionner. Stolcke et Omohundro évoquent ce que nous appellerons dans la suite de ce document le critère de sélection de vraisemblance maximale, mais lui préfèrent un critère plus élaboré construit à l'aide d'une probabilité *a posteriori* et qu'ils nomment critère Bayesien. Nous prétendons que, bien qu'induisant un critère d'arrêt pratique pour l'algorithme, il n'est pas évident que cette procédure ait de meilleures propriétés que le critère de vraisemblance maximale quand à la sélection des états à fusionner. En outre, nous pensons que la meilleure manière de déterminer la taille idéale du HMM en l'absence d'informations suplémentaires (et quand cela est possible) est de générer plusieurs modèles de tailles décroissantes (comme proposés par l'algorithme), d'évaluer ces modèles sur un échantillon indépendant des séquences d'apprentissage, et de choisir le meilleur modèle en conséquence. Pour ces raisons, nous allons nous intéresser exclusivement dans la suite de ce chapitre au critère de vraisemblance maximale. Néamoins, bien que nous ne nous inscrivions donc pas complètement dans le cadre de l'algorithme de Stolcke et Omohundro, la méthode d'estimation des paramètres et les techniques d'approximations développées par ces auteurs s'appliquent également au critère (plus simple) que nous avons choisi. Avant d'examiner plus complètement ces techniques (section 3.1.2), nous allons tout d'abord détailler sur un exemple simple une trace de l'algorithme lorsqu'il utilise le critère de sélection classique.

3.1.1 Un premier exemple

Considérons le langage régulier $(ab)^+$, c'est-à-dire le langage formé des chaînes de caractères ab, $abab$, $ababab$, $\dots$, sans la chaîne vide. On veut construire le HMM représentant ce langage à partir de l'ensemble de séquences d'apprentissage $\mathcal{O} = \{ab, abab\}$. La procédure d'apprentissage par fusion d'état appliquée à cet ensemble est illustrée à la figure 3.2. Le HMM initial H_0 est construit comme une disjonction des deux séquences d'apprentissage. Chacune de ces séquences a la probabilité 0.5 d'être générée et la vraisemblance de $\mathcal{O}$ est donc $0.5 \cdot 0.5 = 0.25$. La fusion des deux états grisés de H_0 pour produire le HMM H_1 n'entraîne pas de baisse de la vraisemblance. Il en est de même pour la fusion suivante, et on obtient ainsi un HMM H_2 dont la vraisemblance est toujours égale à 0.25. À partir de ce HMM par contre, toutes les fusions possibles entraînent une diminution de la vraisemblance de $\mathcal{O}$. Les états dont la fusion engendre la plus petite baisse sont les deux états qui génèrent le symbole b. Ce sont donc ces états qui sont choisis et fusionnés pour produire le HMM H_4. Deux états de H_4 peuvent ensuite être fusionnés sans diminution de la vraisemblance – les états qui génèrent le symbole a – et on obtient alors le HMM H_5, qui est le HMM minimal produisant le langage $(ab)^+$.

3.1.2 Estimation des paramètres

On a vu que le HMM initial peut aisément être construit de manière à maximiser la vraisemblance des séquences d'apprentissage. Les estimateurs des paramètres de ce HMM – probabilités d'émission et probabilités de transition – sont donc les estimateurs de maximum de vraisemblance. On peut d'ailleurs noter que ces estimateurs sont valides quel que soit le type de vraisemblance choisi : ils maximisent aussi bien la vraisemblance réelle que la vraisemblance suivant les chemins de Viterbi. Lors des étapes suivantes, c'est-à-dire après chacune des fusions, le problème se révèle plus difficile. Il faut en effet reparamétrer la structure résultante de la fusion. Ceci peut bien sûr être effectué, suivant le type de vraisemblance choisi, grâce aux algorithmes d'entraînement de Baum-Welch ou de Viterbi exposés au chapitre précédent. Néanmoins, une rapide étude de complexité nous montre que cette solution n'est pas raisonnable lorsque les séquences d'apprentissage sont longues et/ou en nombre important. Notons n le nombre d'états du HMM à un moment donné de l'algorithme d'apprentissage, et N le nombre d'états du HMM initial. On a vu que la complexité des algorithmes d'entraînement de Baum-Welch et de Viterbi est $O(Ln^2T)$ avec T la taille totale des séquences d'apprentissage, et L le nombre de tours de l'algorithme. À chaque itération, il y a $O(n^2)$ fusions possibles, et on doit donc réaliser $O(n^2)$ exécutions de l'une ou l'autre des procédures d'entraînement, c'est-à-dire $O(Ln^4T)$ opérations. Le nombre d'itérations de l'algorithme est de l'ordre de la taille du HMM initial (N) et on fait donc au total $O(LN^5T)$ opérations. On remarquera de plus que N est de l'ordre de la somme des longueurs des séquences d'apprentissage – c'est-à-dire T –, et la complexité générale de l'algorithme d'apprentissage est donc $O(LT^6)$.

Moyennant quelques approximations, une solution plus efficace est possible. Cette solution nécessite tout d'abord l'utilisation des vraisemblances de Viterbi plutôt que des

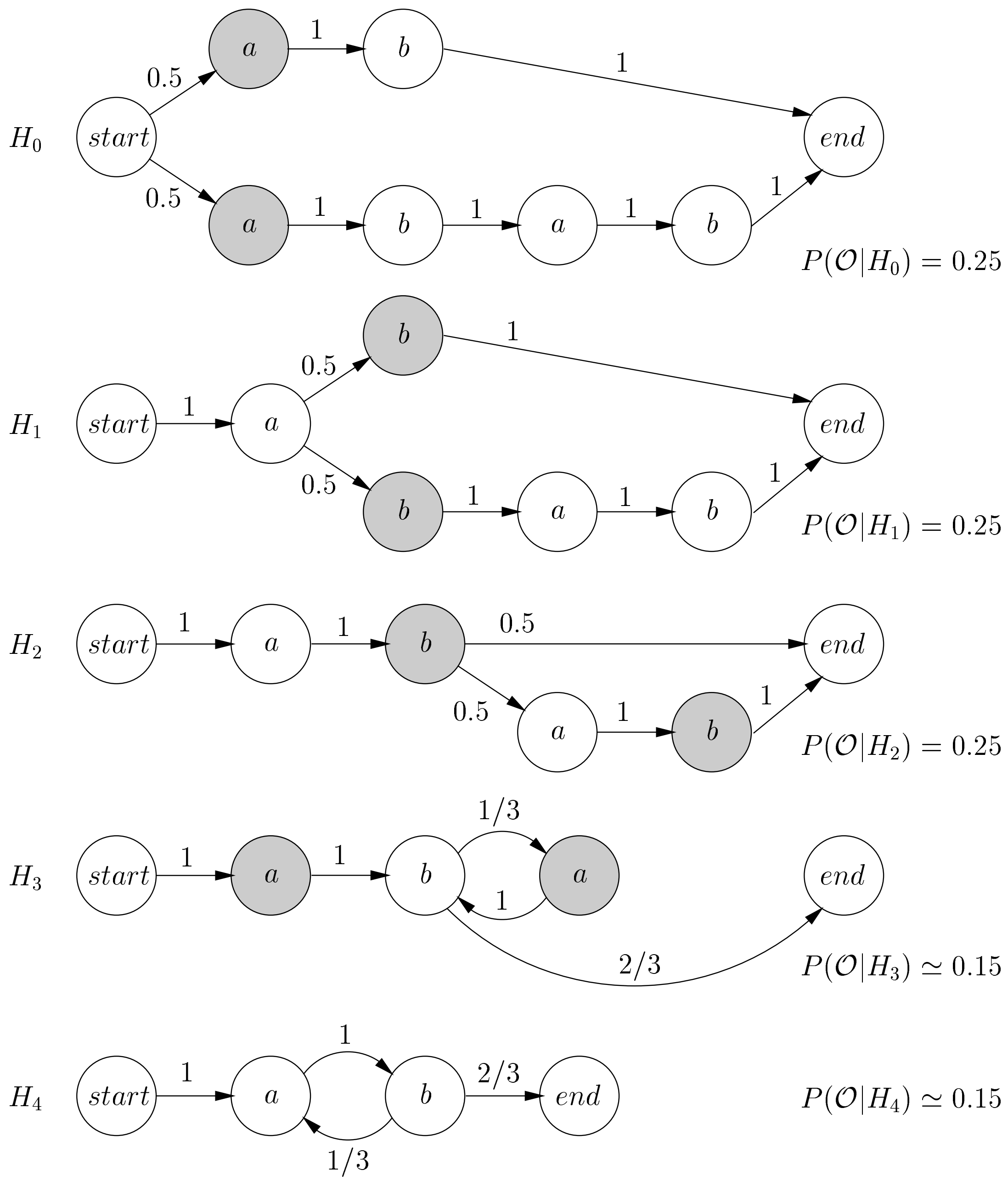


FIG. 3.2: HMMs obtenus lors d'une suite de quatre fusions d'états à partir de l'ensemble de séquences $\mathcal{O} = \{ab, abab\}$. On passe d'un HMM à l'autre en fusionnant les états grisés. À coté de chaque HMM est notée la vraisemblance des données.

vraisemblances totales. Elle implique ensuite l'utilisation d'une hypothèse forte – mais qui se vérifie souvent en pratique – qui est que les chemins de Viterbi des séquences ne sont pas modifiés par les fusions. En d'autres termes, les chemins de Viterbi après une fusion peuvent être déduits des chemins de Viterbi d'avant la fusion en remplaçant simplement les occurrences des états fusionnés par l'état résultant de la fusion. Par exemple, si le chemin de Viterbi d'une séquence O avant la fusion des états u et u' en un état $u \otimes u'$ est $start{-}s{-}u{-}t{-}u{-}u'{-}end$, il sera après la fusion $start{-}s{-}u{\otimes}u'{-}t{-}u{\otimes}u'{-}u{\otimes}u'{-}end$.

L'intérêt de cette hypothèse pour la complexité de l'algorithme est qu'elle permet une réestimation rapide des paramètres du HMM. Sous cette hypothèse, seules les probabilités de transition attachées aux états adjacents au nouvel état ont à être réestimées. Les autres paramètres ne sont pas affectés par la fusion. En effet, les chemins de Viterbi étant invariants, les nombres de fois n_s et $n_{s \to s'}$ où ces états et transitions sont utilisés dans les chemins de Viterbi sont également invariants. Il en est de même pour le nombre de fois n_s^o où chacun des symboles o de l'alphabet est émis sur ces états. Les estimateurs $\widehat{P}(s \to s')$ et $\widehat{P}(o|s)$ des formules (2.14) et (2.15) sont donc eux aussi invariants à la fusion. Ensuite, on peut facilement calculer et remettre à jour le nombre de fois où le nouvel état est utilisé, le nombre de fois où ses transitions adjacentes sont utilisées, ainsi que le nombre de fois où chaque symbole de l'alphabet est généré par le nouvel état. Si u et u' sont fusionnés en un état $u \otimes u'$, on a :

$$n_{u \otimes u'} = n_u + n_{u'},$$

$$n_{t \to u \otimes u'} = n_{t \to u} + n_{t \to u'},$$

$$n_{u \otimes u' \to t} = n_{u \to t} + n_{u' \to t},$$

$$n_{u \otimes u' \to u \otimes u'} = n_{u \to u} + n_{u \to u'} + n_{u' \to u} + n_{u' \to u'},$$

$$n_{u \otimes u'}^o = n_u^o + n_{u'}^o,$$

et on peut alors réestimer les paramètres des états et transitions affectés par la fusion grâce aux formules (2.14) et (2.15). La version détaillée de la procédure de fusion d'états incluant la modification de la structure et la réestimation des paramètres est illustrée par l'algorithme 7. Dans cet algorithme, on note

$$In(u) = \{t \in \mathcal{Q},\ n_{t \to u} \neq 0\},$$

$$Out(u) = \{t \in \mathcal{Q},\ n_{u \to t} \neq 0\},$$

et

$$\Sigma_u = \{o \in \Sigma, n_u^o \neq 0\}.$$

Cet algorithme commence par créer un nouvel état, puis construit l'ensemble des symboles émis par cet état et calcule le nombre de fois où il est utilisé dans les chemins de Viterbi. La première boucle de cet algorithme (ligne 1) met à jour les comptes du nombre de fois où chaque symbole est émis par le nouvel état et estime les probabilités d'émission associées. Les deuxième et troisième boucles (lignes 2 et 3) comptent respectivement le nombre de fois où chaque transition entrante et sortante du nouvel état est utilisée dans les chemins de Viterbi et estime les probabilités de ces transitions. La ligne 4 traite le cas particulier d'une éventuelle boucle présente sur le nouvel état. Une fois que toutes les transitions

Algorithme 7: PROCÉDURE DE FUSION D'ÉTATS

Données : H, un HMM
u et u' deux états à fusionner

ajouter un nouvel état $u \otimes u'$ à H;
$\Sigma_{u\otimes u'} \leftarrow \Sigma_u \cup \Sigma_{u'}$;
$n_{u\otimes u'} \leftarrow n_u + n_{u'}$;
1 **pour chaque** $o \in \Sigma_{u\otimes u'}$ **faire**
  $n^o_{u\otimes u'} \leftarrow n^o_u + n^o_{u'}$;
  estimer $P(o|u \otimes u')$ à l'aide de l'expression (2.15);
2 **pour chaque** $t \in In(u) \cup In(u') - \{u, u'\}$ **faire**
  ajouter la transition $t \to u \otimes u'$;
  $n_{t\to u\otimes u'} \leftarrow n_{t\to u} + n_{t\to u'}$;
  estimer $P(t \to u \otimes u')$ à l'aide de l'expression (2.14);
3 **pour chaque** $t \in Out(u) \cup Out(u') - \{u, u'\}$ **faire**
  ajouter la transition $u \otimes u' \to t$;
  $n_{u\otimes u'\to t} \leftarrow n_{u\to t} + n_{u'\to t}$;
  estimer $P(u \otimes u' \to t)$ à l'aide de l'expression (2.14);
4 **si** $(Out(u) \cup Out(u')) \cap \{u, u'\} \neq \emptyset$ **alors**
  ajouter la transition $u \otimes u' \to u \otimes u'$;
  $n_{u\otimes u'\to u\otimes u'} \leftarrow n_{u\to u'} + n_{u\to u'} + n_{u\to u'} + n_{u\to u'}$;
  estimer $P(u \otimes u' \to u \otimes u')$ à l'aide de l'expression (2.14);
supprimer u et u' (et leurs transitions adjacentes);

adjacentes à cet état ont été créées et leurs probabilités estimées, les anciens états u et u' sont supprimés.

Examinons la complexité de l'algorithme d'apprentissage ainsi obtenu. Si on borne le nombre maximal de transitions adjacentes aux états du HMM par une constante b que l'on espère très inférieure à N – nous verrons au chapitre 4 que cette hypothèse est largement vérifiée – et le nombre de symboles générés par chaque état par une constante b', alors la complexité de l'algorithme 7 est $O(b + b')$. Cette procédure étant à faire $O(n^2)$ fois pour chacun des $O(T)$ tours de l'algorithme, la complexité générale de l'algorithme d'apprentissage n'est alors plus que de $O(T^3(b + b'))$ (contre $O(T^6L)$ pour la première version). À titre indicatif, cela nous permet alors de traiter des problèmes pour lesquels T est de l'ordre de quelques milliers, tandis que pour la première version, on est limité à quelques dizaines.

3.1.3 Discussion et limites de l'algorithme

Revenons sur les conventions et hypothèses de la méthode d'estimation que nous venons de détailler. Elle suppose 1) que l'on maximise la vraisemblance des séquences suivant leur chemin de Viterbi ; 2) que ces chemins ne soient pas affectés par les fusions. Ces deux points peuvent évidemment prêter à discussion. Ils sont néanmoins cohérents et possèdent, outre leur avantage pour la complexité de la procédure, une propriété intéressante. On peut en effet montrer que, pour une taille de HMM N donnée – c'est-à-dire un nombre d'états N fixé –, et à la condition de ne commettre aucune erreur dans la sélection des états à fusionner, une procédure d'apprentissage par fusion d'états utilisant cette méthode d'estimation construit le HMM de taille N pour lequel la vraisemblance des séquences d'apprentissage est maximale au sens de Viterbi. On note $H^*_{(N)}$ et on appelle *HMM optimal* le HMM de taille N qui maximise la vraisemblance de Viterbi des séquences d'apprentissage. Chacune des séquences d'apprentissage possède, dans $H^*_{(N)}$, un chemin de Viterbi. Puisque le HMM initial H_0 est une disjonction des séquences d'apprentissage, chaque état de H_0 est utilisé une et une seule fois pour la génération des séquences d'apprentissage (on supposera pour simplifier que toutes les séquences d'apprentissage sont différentes ; néanmoins le raisonnement ci-dessous s'étend également au cas où elles ne le sont pas). Il existe donc une application des états de H_0 dans les états de $H^*_{(N)}$, qui définit une partition $P^*_{(N)}$ des états de H_0 : il suffit d'associer à chaque état s de H_0 l'état s' qui lui correspond dans $H^*_{(N)}$ en cherchant le chemin de Viterbi dans $H^*_{(N)}$ de la séquence d'où s est tiré. L'ensemble des états de H_0 qui sont de cette manière associés à un même état de $H^*_{(N)}$ définissent alors un sous-ensemble de la partition $P^*_{(N)}$. De plus, $H^*_{(N)}$ étant optimal, ses paramètres sont les estimateurs de maximum de vraisemblance calculés grâce aux formules (2.14) et (2.15) à partir du nombre de fois où chaque état, chaque transition et chaque symbole émis par les états est utilisé dans les chemins de Viterbi. Or on a vu que la procédure de fusion de l'algorithme 7 préserve et met à jour ces comptes après chaque fusion. Si l'on fusionne les états du HMM initial en respectant la partition $P^*_{(N)}$, le HMM que l'on obtient est donc le HMM optimum recherché $H^*_{(N)}$. On vérifie cela aisément en remarquant que la structure du HMM que l'on obtient en fusionnant les états de H_0 suivant $P^*_{(N)}$ est la même que celle de $H^*_{(N)}$ (ils ont le même nombre d'états et les mêmes transitions entre états), et que le fait qu'ils aient la même structure implique également qu'ils aient les mêmes paramètres.

Pour un nombre d'états fixé, le problème de l'apprentissage d'un HMM peut donc être réduit à la recherche de la partition optimale $P^*_{(N)}$. Cette manière de voir ne simplifie bien évidemment pas le problème de l'apprentissage mais amène une remarque intéressante. En effet, à la condition du respect total de $P^*_{(N)}$, l'ordre avec lequel on va fusionner les états de H_0 n'a aucune importance pour le résultat de l'apprentissage. Cela signifie qu'en partant de H_0, l'algorithme d'apprentissage peut décrire plusieurs séquences de HMMs différentes pour arriver au HMM optimal.

Une autre remarque concerne l'hypothèse de non modification des chemins de Viterbi. On constate, comme Stolcke et Omohundro le soulignent, que cette hypothèse se vérifie largement en pratique. Mais lorsque ce n'est pas le cas, l'impact sur le résultat de l'apprentissage n'est pas pour autant catastrophique. Dans tous les cas, le chemin dont on maximise la probabilité, même s'il ne s'agit pas du chemin de Viterbi, est quand même un chemin possible ayant une probabilité non nulle. La probabilité que l'on maximise est donc une sous estimation de la probabilité totale et sa maximisation tend donc à maximiser cette probabilité totale.

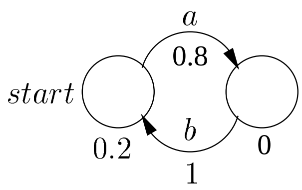


FIG. 3.3: Exemple de PDFA. Les transitions sont étiquetées par leur symbole (a ou b) et par leur probabilité. Les états sont étiquetés par leur probabilité de terminer une séquence (0.2 ou 0).

3.2 Cadre des automates d'états finis probabilistes

Un automate d'états fini déterministe et probabiliste (PDFA) peut être défini par un quadruplet $\langle \mathcal{Q}, \Sigma, \delta, \mathcal{P} \rangle$ avec :

- $\mathcal{Q}$, un ensemble fini d'états ; $\mathcal{Q}$ contient un état spécial noté *start* servant à débuter une séquence.
- Σ, un alphabet fini ;
- $\delta : \mathcal{Q} \times \Sigma \to \mathcal{Q}$, une fonction de transition associant à chaque état de $\mathcal{Q}$ et chaque symbole de Σ un état de $\mathcal{Q}$;
- $\mathcal{P} = \mathcal{Q} \times \Sigma \cup \{\#\} \to [0, 1]$, une matrice qui associe à chaque état de $\mathcal{Q}$ une distribution de probabilités sur $\Sigma \cup \{\#\}$; # est un symbole spécial servant à conclure une séquence. On note $P(o|s)$ la probabilité de générer le symbole o au départ de l'état s, et $P(s|\#)$ la probabilité de terminer le processus de génération sur l'état s.

La figure 3.3 représente un exemple de PDFA qui génère le langage $(ab)^*$, c'est-à-dire le langage formé des chaînes de caractères ab, $abab$, $ababab$, ... , et de la chaîne vide.

Comme l'algorithme de Stolcke et Omohundro, l'algorithme ALERGIA introduit par Carrasco et Oncina [Carrasco et Oncina, 1994] pour l'apprentissage des PDFAs utilise le principe de généralisation par fusions d'états. Le déroulement général de cet algorithme est assez similaire à celui utilisé pour les HMMs. Un automate initial est construit à partir des séquences d'apprentissage de manière à maximiser leur probabilité de génération. Du fait du déterminisme des PDFAs (à chaque état n'est attaché qu'au plus une transition étiquetée par un symbole donné), cet automate n'est pas, comme c'est le cas dans le cadre des HMMs, une simple disjonction des séquences, mais est construit comme l'*arbre préfixe* de ces séquences. Un arbre préfixe est un arbre pour lequel les préfixes communs ne sont pas répétés mais sont représentés par un même unique chemin. L'automate du haut de la figure 3.4, par exemple, est l'automate initial construit à partir des séquences d'apprentissage $\{aba, abb, bbb\}$. Ensuite, à chaque tour de l'algorithme, deux états sont sélectionnés et fusionnés pour produire un nouvel automate. Mais là encore, du fait du déterminisme du modèle, la manière de procéder est légèrement différente que pour les HMMs. Ici, chaque fusion est susceptible d'entraîner en cascade de nombreuses autres fusions. En effet, si les deux états fusionnés possèdent tous les deux une transition sortante étiquetée par un même symbole, alors, afin de préserver le déterminisme de l'automate résultant de la fusion, ces deux transitions et les états sur lesquelles elles pointent doivent eux aussi être fusionnés ; et ce processus doit être répété tant qu'il existe des transitions redondantes. Par exemple, sur la figure 3.4, la fusion des états 1 et 2 crée un indéterminisme (il y a deux transitions étiquetées b à la sortie de l'état résultant de la fusion), et entraîne alors la fusion des états 3 et 4, puis 6 et 7 pour arriver à un automate à nouveau déterministe. Après chaque fusion, les paramètres de la structure sont réestimés de la même manière que pour les HMMs, en comptant le nombre de fois où chaque état et chaque transition est utilisé. En outre, comme il existe pour ce modèle au plus un chemin capable de générer une séquence donnée, l'hypothèse de Viterbi est complètement vérifiée. En reprenant les notation n_s et n_s^o introduites pour les HMMs et qui désignent respectivement pour ce modèle le nombre de fois où l'état s est utilisé et le nombre de fois où le symbole o est généré à partir de l'état s – c'est-à-dire en fait le nombre de fois où on utilise la transition étiquetée o au départ de s –, on peut alors proposer l'estimateur de maximum de vraisemblance suivant :

$$\widehat{P}(o|s) = \frac{n_s^o}{n_s}.$$

En fait, la différence essentielle de cet algorithme avec celui de Stolcke et Omohundro réside dans la procédure de sélection des états à fusionner. Plutôt que de sélectionner à chaque étape le couple d'états dont la fusion engendre la plus petite baisse de vraisemblance (ou la plus grande augmentation d'une probabilité *a posteriori*), ALERGIA utilise un critère afin de juger de la compatibilté de deux états et les fusionner le cas échéant. Ce critère de compatibilité prend en compte l'ensemble des états accessibles à partir des deux états en question et est défini récursivement. De manière informelle, deux états s et s' sont considérés comme *compatibles* si les distributions de probabilités sur Σ qui leur sont associées sont proches, et si leurs états sortants sont également compatibles. Pour juger de la ressemblance des probabilités des transitions associées aux états considérés, Carrasco et Oncina proposent d'utiliser une mesure tirée de la borne de Hoeffding [Hoeffding, 1963]. Plus formellement, deux états s et s' sont alors dit α-compatibles (avec $0 < \alpha < 1$) si les deux conditions suivantes sont satisfaites :

$$(1) \quad \left|\frac{n_s^o}{n_s} - \frac{n_{s'}^o}{n_{s'}}\right| < \sqrt{\frac{1}{2}\ln\left(\frac{2}{\alpha}\right)}\left(\frac{1}{\sqrt{n_s}} + \frac{1}{\sqrt{n_{s'}}}\right), \ \forall o \in \Sigma \cup \{\#\}$$

$$(2) \quad \delta(s,o) \text{ et } \delta(s',o) \text{ sont } \alpha\text{-compatibles}, \ \forall o \in \Sigma.$$

Du fait du caractère récursif de sa définition (exprimé par la deuxième condition), la compatibilité de deux états ne peut être calculée en temps fini que si l'un au moins de ces états est racine d'un arbre. Pour garantir cela, les auteurs introduisent un ordre sur les fusions défini de la manière suivante. Les états de l'automate initial sont tout d'abord ordonnés lexicographiquement. Cet ordre est calculé à partir des séquences préfixes associées à chacun des états de l'automate. Par exemple, à l'état 4 de l'automate initial de la figure 3.4 est associé la séquence *bb*. En ordonnant lexicographiquement l'ensemble de ces séquences préfixes (c'est-à-dire en les classant par taille et suivant un ordre alphabétique pour les séquences de même taille) on obtient donc un ordre sur les états de l'automate initial. Cet ordre est utilisé pour désigner les états de l'automate : 0 désigne l'état *start*, et le nombre le plus grand désigne une des feuilles de l'arbre préfixe (voir l'automate initial

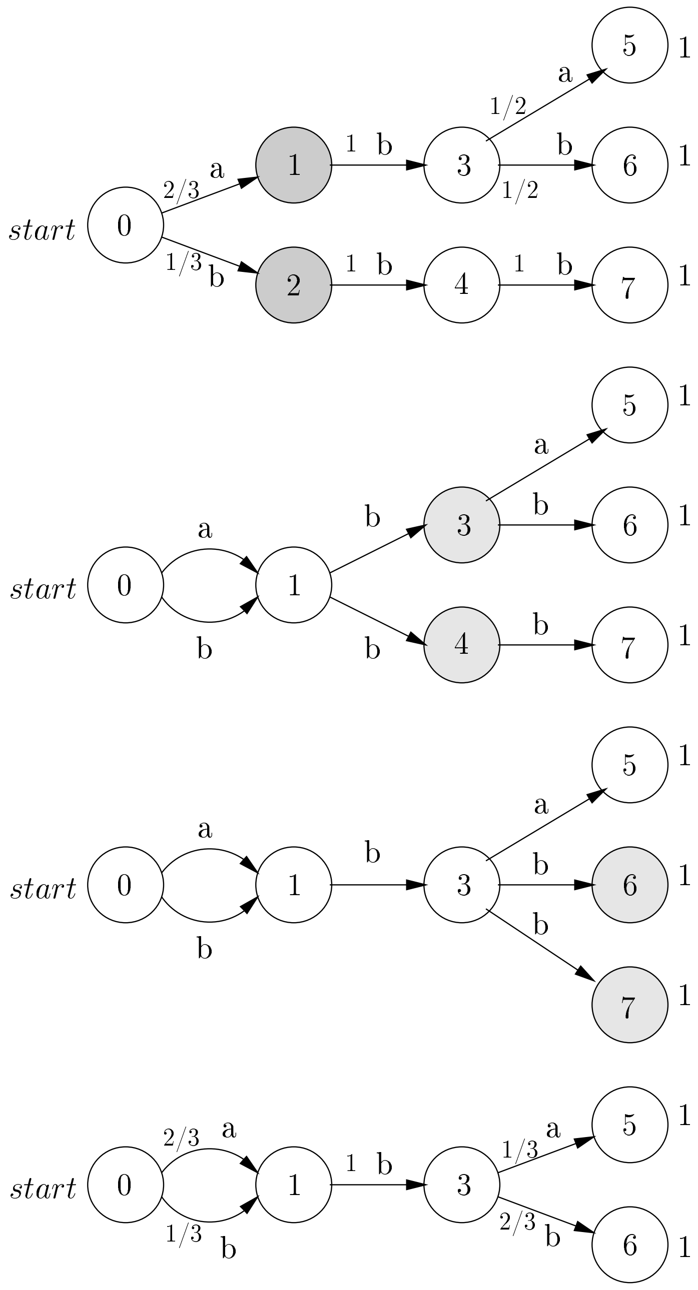


FIG. 3.4: Automate initial construit à partir de l'ensemble d'apprentissage $\{aba, abb, bbb\}$ (en haut), et les différentes étapes de la fusion des états 1 et 2. Pour des raisons de lisibilité, les probabilités de conclure une séquence ne sont reportées que sur les états pour lesquels cette probabilité est non nulle (les états feuilles).

Algorithme 8: ALGORITHME ALERGIA.

Données : $\mathcal{O}$ un ensemble de séquences d'apprentissage
$0 < \alpha < 1$

Créer le PDFA initial construit comme l'arbre préfixe des séquences d'apprentissage;
pour chaque état j de 1 à $|\mathcal{Q}|$ **faire**
  pour chaque état i de 0 à $j-1$ **faire**
    si i et j sont α-compatibles **alors**
      fusionner i et j;
      sortir de la boucle;

---

de la figure 3.4). Les fusions potentielles sont alors considérées de manière suivante : pour chaque état j de 1 à $|\mathcal{Q}|$, on calcule la compatibilité de j avec i pris entre 0 et $j-1$. Cet ordre sur les fusions garantit que l'état j est toujours la racine d'un arbre. Si j et i sont compatibles, ils sont fusionnés et l'état résultant est alors désigné par le nombre le plus petit, c'est à dire i. Une version de l'algorithme ALERGIA est illustré par l'algorithme 8. Dans cet algorithme, la procédure de fusion n'inclut pas seulement la fusion des états i et j identifiés comme compatibles mais également toutes les fusions consécutives nécessaires pour rendre l'automate déterministe.

La figure 3.5 représente une trace de l'algorithme tirée de [Carrasco et Oncina, 1994]. Le PDFA du haut est l'arbre préfixe construit à partir de l'ensemble d'apprentissage

$$\mathcal{O} = \{bba, \lambda, \lambda, \lambda, a, \lambda, aa, aa, \lambda, \lambda, \lambda, babba, \lambda, \lambda, baa\},$$

avec λ la chaîne vide. On commence par tester si les états 1 et 2 sont compatibles. Avec, par exemple, $\alpha = 0.8$, on a

$$\sqrt{\frac{1}{2}\ln\left(\frac{2}{\alpha}\right)} \simeq 0.67$$

On doit comparer les probabilités attachées à 1 et à 2. Pour les probabilités de terminaison, on a :

$$|1/3 - 9/15| \simeq 0.26 < 0.67(1/\sqrt{15} + 1/\sqrt{3}) \simeq 0.55 \;;$$

pour la probabilité de a, on a :

$$|2/3 - 3/15| \simeq 0.46 < 0.55 \;;$$

et pour b :

$$|0 - 0| = 0 < 0.55.$$

De plus, l'équivalence de 1 et 2 nécessite de tester la compatibilité de 4 et 2 (du fait de la définition récursive). De la même manière, on trouve que ces états sont également compatibles, et on peut donc fusionner les états 1 et 2. On obtient alors le deuxième PDFA de la figure 3.5. À cette étape, ALERGIA va tester l'équivalence de 1 et 3 – ils ne sont pas compatibles du fait de leurs probabilités de terminaison (respectivement 0.6 et 0, ce qui fait une différence supérieure à 0.53) –, puis de 1 et 5 (ce qui nécessite également la comparaison de 7 et 1, 8 et 3, 10 et 6, et 11 et 9) qui eux sont α-compatibles et sont donc fusionnés. Finalement, ALERGIA fusionne les états 3 et 6, et on obtient le dernier PDFA de la figure.

3.3 Conclusion

Nous avons présenté dans ce chapitre deux algorithmes classiques procédant par fusions d'états : l'algorithme d'apprentissage de Stolcke et Omohundro pour les HMMs, et l'algorithme ALERGIA de Carrasco et Oncina pour l'apprentissage des PDFAs. Nous avons vu que ces deux algorithmes, très proches dans leur fonctionnement général, diffèrent en fait par le critère de fusion utilisé. L'algorithme de Stolcke et Omohundro utilise un critère que l'on peut qualifier de global car il se base pour juger une fusion potentielle sur la probabilité *a posteriori* du modèle qu'elle induit. L'algorithme ALERGIA, au contraire, juge une fusion entre deux états à l'aide d'un critère local qui mesure la similarité des différents états accessibles à partir des deux états en question. Concernant l'algorithme de Stolcke et Omohundro, nous avons également présenté un autre critère de sélection des états à fusionner plus simple que le critère qu'ils proposent et que nous appellerons dans la suite de ce document *critère de vraisemblance maximale*. Il consiste à sélectionner le couple d'états dont la fusion engendre, non pas la plus importante augmentation de la probabilité *a posteriori*, mais la plus faible diminution de la vraisemblance des séquences d'apprentissage. Ce critère, mentionné par ces auteurs mais non étudié, est moins élaboré que celui qu'ils proposent mais nous semble plus naturel et plus souple à utiliser car il n'impose pas de critère d'arrêt à l'algorithme. Il est de plus intrinsèquement lié à leur critère, puisqu'il correspond au cas particulier où la distribution de probabilités *a priori* sur l'ensemble des HMMs est uniforme.

Concernant l'estimation des paramètres des modèles inférés à chaque tour de l'algorithme, la nature déterministe des PDFAs permet une méthode d'estimation de type maximum de vraisemblance exacte. Pour les HMMs, Stolcke et Omohundro suggèrent d'utiliser à la place des algorithmes d'entraînement classiques, une méthode d'estimation qui, moyennant quelques approximations, permet de réduire considérablement la complexité de l'algorithme d'apprentissage. Cette méthode peut être appliquée aussi bien pour le critère de sélection de vraisemblance maximale que pour leur critère.

Dans les prochains chapitres, nous allons nous intéresser plus particulièrement à ce qui constitue le coeur de l'algorithme d'apprentissage par fusions d'états, c'est-à-dire la procédure de sélection des états à fusionner. Au chapitre 4, nous proposons une implémentation efficace de la procédure de sélection utilisant le critère de vraisemblance maximale et nous montrons que cette implémentation permet de réduire encore la complexité de l'algorithme d'apprentissage. Au chapitre 5, nous proposons un tout autre critère de sélection qui s'ap-

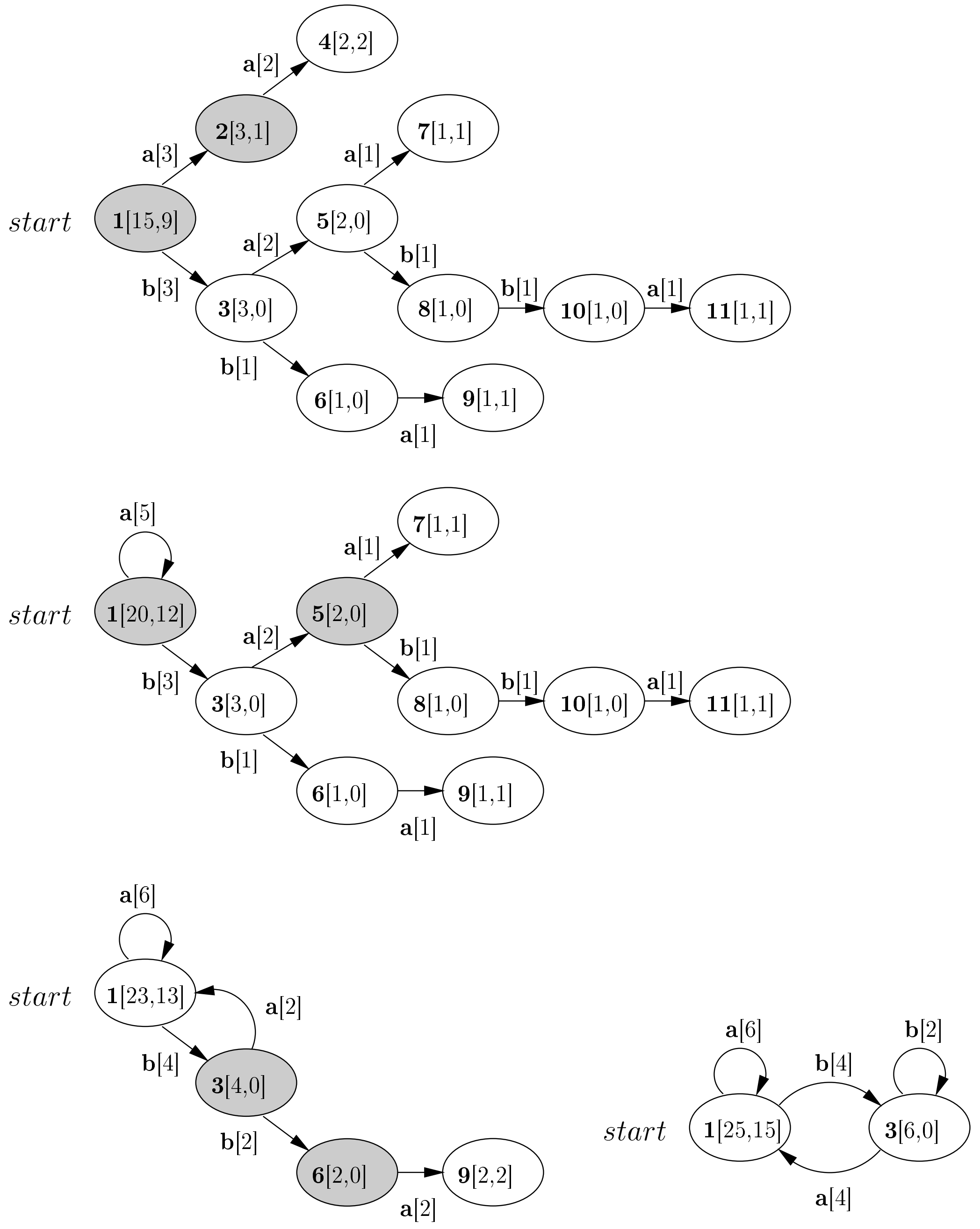


FIG. 3.5: Une trace d'exécution de l'algorithme ALERGIA à partir de l'ensemble d'apprentissage $\mathcal{O} = \{bba, \lambda, \lambda, \lambda, a, \lambda, aa, aa, \lambda, \lambda, \lambda, babba, \lambda, \lambda, baa\}$. Chaque état s est désigné par son numéro (l'état *start* a le numéro 1) et, entre crochets, le nombre de fois n_s où il est utilisé, et le nombre de séquences $n_s^{\#}$ qui se terminent sur cet état. Chaque transition est labellée par le symbole (a ou b) qui l'étiquette et, entre crochets, le nombre de fois où elle est utilisée.

parente au critère employé par ALERGIA, et nous présentons alors un nouvel algorithme d'apprentissage qui utilise ce critère.

Chapitre 4

Optimisation de l'algorithme par fusions d'états

Au chapitre précédent, nous avons présenté l'algorithme d'apprentissage par fusions d'états proposé par Stolcke et Omohundro [Stolcke et Omohundro, 1994a]. Nous avons exposé deux critères qu'il est possible d'utiliser pour sélectionner à chaque étape de l'algorithme le couple d'états à fusionner : le critère proposé par Stolcke et Omohundro, basé sur une probabilité *a posteriori* et qu'ils nomment *critère Bayesien*, et un second critère, plus simple, que nous appelons *critère de vraisemblance maximale*, et qui consiste simplement à sélectionner le couple d'états dont la fusion engendre une baisse minimale de la vraisemblance. Nous nous intéressons dans ce chapitre à l'optimisation de la procédure d'apprentissage lorsqu'elle utilise le critère de vraisemblance maximale. La première section présente une implémentation efficace de la procédure de sélection qui diminue la complexité de l'algorithme. La seconde section détaille une procédure de création du HMM initial qui permet de diminuer de manière importante le nombre d'états de celui-ci, sans changer le résultat de l'algorithme.

4.1 Sélection des états à fusionner

Nous avons montré au chapitre précédent que, moyennant quelques approximations, la complexité de la procédure d'apprentissage par fusions d'états, quel que soit le critère de sélection utilisé, pouvait être réduite à $O(N^3(b + b'))$. Nous allons voir maintenant que dans le cadre du critère de vraisemblance maximale, une implémentation judicieuse de la procédure de sélection peut encore diminuer cette complexité. L'idée de base de l'implémentation que nous proposons est de ne pas recalculer les performances de toutes les paires d'états que l'on peut potentiellement fusionner à chaque étape de l'algorithme. On utilise pour cela une des spécificités de l'algorithme d'apprentissage qui est que les HMMs générés à chaque tour ont généralement un nombre de transitions réduit comparativement au nombre d'états. Cette caractéristique de l'algorithme a d'ailleurs déjà été exploitée par la procédure de fusion de l'algorithme 7, page 48. Nous allons voir maintenant qu'elle permet également d'implémenter une procédure de sélection efficace.

Considérons la quantité $\theta(u,u')$ définie par :

$$\theta(u,u') = \frac{P(\mathcal{O},\mathcal{S}^*|H_{u\otimes u'})}{P(\mathcal{O},\mathcal{S}^*|H)},$$

avec $H_{u\otimes u'}$ le HMM résultant de la fusion des états u et u' dans le HMM H. Cette quantité représente le rapport de la vraisemblance des séquences d'apprentissage dans le HMM résultant de la fusion de u et u', sur la vraisemblance de ces séquences avant que la fusion n'ait lieu. Elle peut avantageusement remplacer la vraisemblance des données dans la recherche du meilleur couple d'états. On a en effet :

$$P(\mathcal{O},\mathcal{S}^*|H_{u\otimes u'}) > P(\mathcal{O},\mathcal{S}^*|H_{t\otimes t'}) \iff \theta(u,u') > \theta(t,t'). \tag{4.1}$$

$P(\mathcal{O},\mathcal{S}^*|H)$ et $P(\mathcal{O},\mathcal{S}^*|H_{u\otimes u'})$ sont définis par l'expression (2.13), qui consiste en un produit de facteurs indépendants sur l'ensemble des états du HMM. Entre $P(\mathcal{O},\mathcal{S}^*|H)$ et $P(\mathcal{O},\mathcal{S}^*|H_{u\otimes u'})$, plusieurs de ces facteurs sont invariants. C'est le cas du produit $\prod_{o\in\Sigma_s} P(o|s)^{n_s^o}$ des états s différents de u et u', et c'est également le cas des probabilités des transitions non adjacentes à u et u'. À la figure 3.1, par exemple, seules les probabilités des transitions sortantes de u, u' et t sont affectées par la fusion. Les quantités invariantes sont donc éliminées lors du calcul de $\theta(u,u')$. Soit $In(u,u') = In(u) \cap In(u') - \{u,u'\}$ l'ensemble des états entrants communs à u et u', et $Out(u,u') = Out(u) \cup Out(u')$ l'union des états sortants de u et u'. Le rapport des facteurs relatifs aux probabilités de transitions affectées par la fusion est alors :

$$T(u,u') = IT(u,u') \cdot OT(u,u'), \tag{4.2}$$

avec

$$IT(u,u') = \frac{\prod_{t\in In(u,u')} P(t\to u\otimes u')^{n_{t\to u\otimes u'}}}{\left(\prod_{t\in In(u,u')} P(t\to u)^{n_{t\to u}} P(t\to u')^{n_{t\to u'}}\right)},$$

et

$$OT(u,u') = \frac{\prod_{t\in Out(u,u')} P(u\otimes u'\to t)^{n_{u\otimes u'\to t}}}{\left(\prod_{t\in Out(u)} P(u\to t)^{n_{u\to t}}\right)\left(\prod_{t\in Out(u')} P(u'\to t)^{n_{u'\to t}}\right)}.$$

Soit, pour finir, E_s défini par :

$$E_s = \prod_{o\in\Sigma_s} P(o|s)^{n_s^o}. \tag{4.3}$$

On a alors :

$$\theta(u,u') = T(u,u') \cdot \frac{E_{u\otimes u'}}{E_u E_{u'}}. \tag{4.4}$$

Le premier terme de l'expression (4.4) dépend des probabilités des transitions adjacentes à u, u' et $u\otimes u'$. Le second terme dépend uniquement des probabilités d'émission associées aux états u, u' et $u\otimes u'$. La quantité $\theta(u,u')$ n'est donc pas affectée par les fusions

Algorithme 9: ALGORITHME D'APPRENTISSAGE PAR FUSIONS D'ÉTATS UTILISANT LE CRITÈRE DE SÉLECTION CLASSIQUE.

Données : $\mathcal{O}$ un ensemble de séquences d'apprentissage

Créer un HMM initial H_0 qui maximise la vraisemblance de $\mathcal{O}$;
Créer la liste $\mathcal{L}$ (algorithme 10);
répéter
  $(u, u') \leftarrow$ couple minimal de $\mathcal{L}$;
  Fusionner u et u' (algorithme 7);
  Mettre à jour $\mathcal{L}$ (algorithme 11);
jusqu'à critère d'arrêt;

Algorithme 10: CRÉATION DE LA LISTE DES COUPLES ORDONNÉE

Données : H, un HMM

$\mathcal{L} \leftarrow$ liste vide;
pour chaque $s \in \mathcal{Q}$ **faire**
  Stocker $E_s = 1$;

pour chaque couple d'états (s, s') **faire**
  Calculer $T(s, s')$ à l'aide de la formule (4.2);
  Calculer et stocker $E_{s \otimes s'}$ à l'aide de la formule (4.3);
  Calculer $\theta(s, s')$ à l'aide de la formule (4.4);
  Insérer (s, s') à sa place dans $\mathcal{L}$;

d'états qui ne mettent pas en jeu l'un des états adjacents à u ou u'. Regardons maintenant la complexité du calcul de l'expression (4.4). Le terme $T(u, u')$ se calcule grâce à la formule (4.2) qui consiste en une exploration des transitions adjacentes à u et u' et a donc une complexité de $O(b)$. Le deuxième terme de l'équation (4.4) se calcule en $O(b')$. La complexité du calcul de $\theta(u, u')$ est donc $O(b + b')$.

Considérons la liste $\mathcal{L}$ des couples d'états du HMM, ordonnée suivant θ et mise à jour tout au long de l'algorithme. En utilisant cette liste, et en vertu de l'équivalence (4.1), le processus de sélection des états à fusionner se résume alors à choisir, à chaque itération, le premier couple de $\mathcal{L}$. On peut donc proposer une version améliorée de l'algorithme d'apprentissage (voir algorithme 9), qui crée la liste $\mathcal{L}$ en début de procédure, et remet à jour cette liste après chaque fusion. Détaillons maintenant les procédures de création et de mise à jour de $\mathcal{L}$.

4.1.1 Création de la liste des couples

La création de la liste $\mathcal{L}$ au début de la procédure d'apprentissage (voir algorithme 10) consiste à calculer et insérer dans $\mathcal{L}$ les valeurs $\theta(s, s')$ des $O(N^2)$ couples d'états (s, s') possibles du HMM initial H_0. Insérer un couple d'états se fait en $O(\log(N))$ et la complexité de cette procédure de création est donc $O(N^2(b + b' + \log(N)))$.

Algorithme 11: Mise à jour de la liste des couples ordonnée

Données : H, $\mathcal{L}$, u, u', $u \otimes u'$

Calculer et stocker la valeur de $E_{u \otimes u'}$ à l'aide de la formule (4.3);
1 Supprimer tous les couples d'états de $\mathcal{L}$ contenant u, u', ou tout état s appartenant à $In(u, u') \cup Out(u, u')$;
pour chaque $s \in \{u \otimes u'\} \cup In(u, u') \cup Out(u, u')$ **faire**
  pour chaque $s' \in \mathcal{Q} - \{s\}$ **faire**
    Calculer $T(s, s')$ à l'aide de la formule (4.2);
    Calculer et stocker $E_{s \otimes s'}$ à l'aide de la formule (4.3);
2   Calculer $\theta(s, s')$ à l'aide de la formule (4.4);
3   Insérer (s, s') à sa place dans $\mathcal{L}$;

4.1.2 Mise à jour de la liste des couples

La procédure de mise à jour (voir algorithme 11) consiste, après chaque fusion de deux états u et u', à supprimer les $O(Nb)$ couples d'états dans lesquels apparaissent u, u' ou l'un de leurs états adjacents (ligne 1), à calculer la valeur θ des $O(Nb)$ couples d'états que l'on peut former avec le nouvel état $u \otimes u'$ ou ses états adjacents et les autres états du HMM (ligne 2), et à insérer ces couples à leur place dans la liste (ligne 3). La complexité de cette procédure est donc $O(Nb(b + b' + \log(N)))$.

4.1.3 Complexité de l'algorithme

Étudions la complexité de l'algorithme d'apprentissage ainsi obtenu (algorithme 9). La création du HMM H_0 se fait en $O(N)$. La création de $\mathcal{L}$ est en $O(N^2(b + b' + \log(N)))$. La fusion a une complexité de $O(b + b')$ et la mise à jour se fait en $O(Nb(b + b' + \log(N)))$. Ces deux dernières procédures sont à exécuter à chacun des $O(N)$ tours de l'algorithme, et celui-ci a donc une complexité totale de $O(N^2 b(b + b' + \log(N)))$. Cette complexité est fortement dépendante des facteurs b et b'. Or, comme on l'a déjà évoqué, les valeurs de ces facteurs dans les HMMs successifs produits par la procédure d'apprentissage sont relativement faibles. Cette caractéristique est essentiellement due au critère de sélection utilisé – la maximisation de la vraisemblance des séquences d'apprentissage – et s'explique aisément en observant l'expression (2.13). On s'aperçoit alors que plus les états ont de transitions sortantes – plus $|Out(s)|$ est grand –, moins la vraisemblance est élevée. Par exemple, avec $|Out(s)| = 1$, alors $P(s \to s') = 1$ et donc $\prod_{s' \in Out(s)} P(s \to s')^{n_{s \to s'}} = 1$, tandis qu'avec $|Out(s)| = 2$ et $P(s \to s') = 1/2 \ \forall s' \in Out(s)$ on a $\prod_{s' \in Out(s)} P(s \to s')^{n_{s \to s'}} = (1/2)^{n_s}$. En étudiant sur un exemple les courbes d'évolution du facteur de branchement des états au cours de l'algorithme, on s'aperçoit que le facteur de branchement maximal (figure 4.1) baisse à mesure des fusions, tandis que le facteur de branchement moyen (figure 4.2) reste relativement bas. Les mêmes raisons expliquent également que la valeur du facteur b' reste elle aussi relativement faible comparativement à la taille de l'alphabet. En effet, là aussi, plus $|\Sigma_s|$ est grand, moins la vraisemblance des séquences est élevée. Les figures 4.3 et

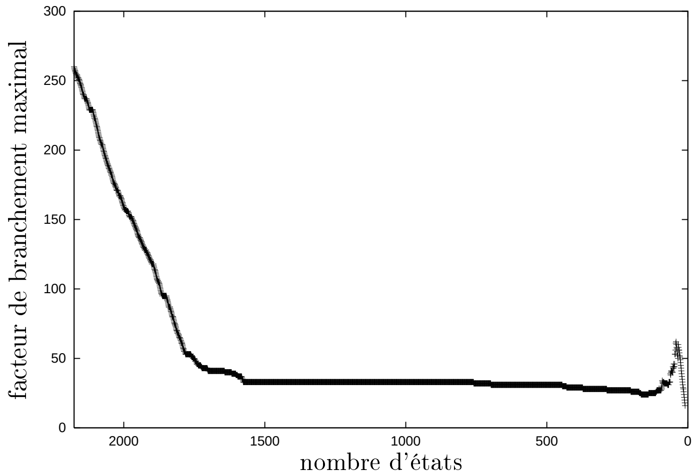


FIG. 4.1: Évolution du facteur de branchement maximal des états du HMM au cours de l'algorithme. Cette courbe a été obtenue lors de l'apprentissage d'un HMM dans la série d'expériences détaillées au chapitre 8 pour modéliser des séquences servant au test de circuits microélectroniques.

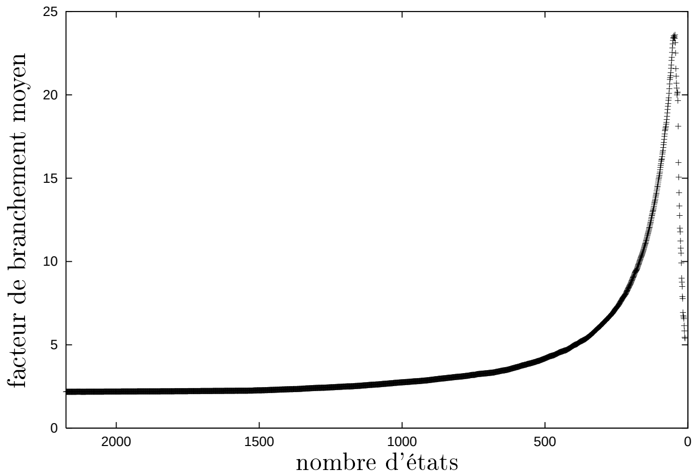


FIG. 4.2: Évolution du facteur de branchement moyen des états du HMM au cours de l'algorithme. Cette courbe a été obtenue lors de l'apprentissage d'un HMM dans la série d'expériences détaillées au chapitre 8 pour modéliser des séquences servant au test de circuits microélectroniques.

4.4, par exemple, représentent l'évolution du nombre de symboles, respectivement maximal et moyen, portés par les états d'un HMM au cours de la procédure d'apprentissage. Cet exemple est issu des expérimentations effectuées au chapitre 6 sur la modélisation de séquences textuelles. La taille totale de l'alphabet est de l'ordre de 3000 symboles. On observe que le nombre moyen de symboles (aussi bien que le nombre maximal) reste très bas pendant une majeure partie de l'algorithme, puis la courbe prend l'allure d'un pic, en restant toutefois dans des valeurs raisonnables comparées à la taille de l'alphabet.

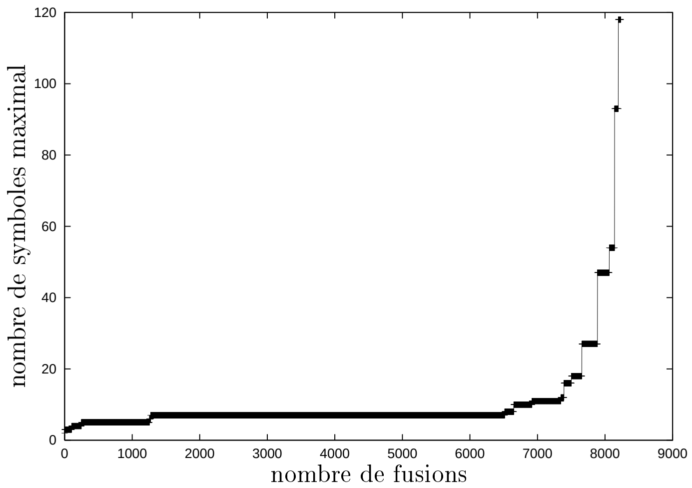


Fig. 4.3: Évolution du nombre maximal de symboles portés par les états du HMM au cours de l'algorithme. Cette courbe a été obtenue lors de l'apprentissage d'un HMM dans la série d'expériences réalisées au chapitre 6 pour modéliser des séquences textuelles. La taille totale de l'alphabet est de l'ordre de 3000 symboles.

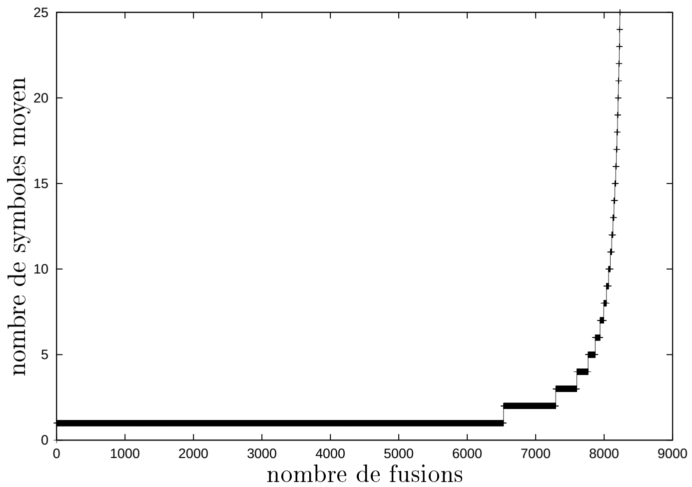


Fig. 4.4: Évolution du nombre moyen de symboles portés par les états du HMM au cours de l'algorithme. Cette courbe a été obtenue lors de l'apprentissage d'un HMM dans la série d'expériences réalisées au chapitre 6 pour modéliser des séquences textuelles. La taille totale de l'alphabet est de l'ordre de 3000 symboles.

4.2 Réduction du nombre d'états du HMM initial

Le facteur le plus important pour la complexité de l'algorithme est bien évidemment N, c'est à dire le nombre d'états du HMM initial H_0. Ce HMM étant construit comme une disjonction des séquences d'apprentissage, N est égal à la taille totale de ces séquences. Nous allons voir maintenant que, du fait du critère de sélection choisi, il est possible de réduire le nombre d'états de H_0. En effet, les premières fusions qui sont effectuées par l'algorithme sont les fusions qui n'entraînent pas de baisse de la vraisemblance des séquences d'apprentissage. En construisant le HMM initial de telle manière que ces fusions soient en quelque sorte déjà effectuées, on peut réduire le nombre d'états de ce HMM sans que cela n'ait d'incidence sur la suite du processus. Observons de nouveau l'exemple de la figure 3.2 à la page 46. Entre H_0 et H_2, aucune des fusions effectuées n'engendre de baisse de la vraisemblance des données. En fait, H_0 et H_2 sont strictement équivalents au sens du langage qu'ils représentent : ils génèrent exactement les mêmes séquences – celles de l'ensemble d'apprentissage – et avec les mêmes probabilités. Abstraction faite de l'état *end*, H_2 est un arbre préfixe – similaire à celui utilisé par l'algorithme ALERGIA lors de la construction de l'automate initial –, c'est à dire un arbre pour lequel les préfixes communs ne sont pas répétés mais représentés par un unique chemin. Il est aisé de vérifier que la fusion de deux états fils d'un même état qui émettent le même symbole n'entraîne pas de modification de la vraisemblance des séquences d'apprentissage, et que l'arbre préfixe peut être obtenu par itération de cette procédure sur le HMM disjonction de ces séquences. Une première solution pour réduire la taille du HMM initial est donc de construire celui-ci non pas comme une simple disjonction des séquences d'apprentissage, mais comme l'arbre préfixe de ces séquences. La construction d'un tel HMM à partir d'un ensemble de séquences $\mathcal{O}$ est détaillée dans l'algorithme 12. Dans cet algorithme, on prend les séquences une à une (ligne 1), et on lit chaque séquence symbole après symbole (ligne 3). L'état q représente l'état courant ; pour chaque nouvelle séquence, q est l'état *start* (ligne 2). Ensuite, pour chaque nouveau symbole, on vérifie qu'il n'y a pas un état sortant de l'état courant qui génère déjà ce symbole. Si c'est le cas (ligne 4), cet état est réutilisé pour générer le symbole et il devient l'état courant. Si ce n'est pas le cas (ligne 5), on crée alors un nouvel état que l'on relie à l'état courant et qui devient l'état courant. On réitère la procédure avec le symbole suivant jusqu'à ce que l'on ait lu la séquence entière et on recommence alors avec une nouvelle séquence. Lorsque toutes les séquences ont été traitées, on crée l'état *end*, auquel on relie toutes les feuilles de l'arbre (ligne 6). On vérifie aisément que cette procédure est linéaire dans la taille totale des séquences d'apprentissage.

De la même manière que l'on peut fusionner les préfixes communs, il est également possible de fusionner ensuite les suffixes communs de manière à réduire encore le nombre d'états du HMM. La procédure de fusion des suffixes est illustrée par l'algorithme 13. Cette opération est faite de manière récursive, par un parcours en largeur partant de l'état *end* (ligne 1) et en fusionnant un à un les états, parents d'un même état, qui émettent le même symbole (ligne 3) – on utilise dans l'algorithme 13 les ensembles $\mathcal{Z}$ et $\mathcal{Z}'$ pour stocker les états à traiter à chaque étape du parcours. Elle doit par contre être effectuée précautionneusement en contrôlant que chaque fusion s'applique sur des états pointant sur les mêmes états, et avec les mêmes probabilités (ligne 2). En effet, du fait de la fusion préalable des préfixes, on peut rencontrer au cours de cette procédure, deux états qui génèrent le même symbole et qui pointent tous les deux sur un même état mais qui ont en fait des suffixes différents. Sur le HMM H_2 de la figure 3.2, par exemple, les états grisés pointent sur *end* et génèrent tous les deux le symbole b ; ils n'ont pourtant pas les mêmes suffixes, et leur fusion entraîne une baisse de la vraisemblance des données. Il est aisé de vérifier que les fusions effectuées par la procédure FusionnerSuffixes n'a aucun impact sur la vraisemblance des séquences d'apprentissage, puisque les états émettent toujours les

Algorithme 12: Création du HMM initial

Données : $\mathcal{O} = \{O^1, \ldots, O^K\}$, un ensemble de séquences d'apprentissage

Créer l'état *start*;
$n_{start} \leftarrow 0$;
pour chaque $O = o_1 \ldots o_T \in \mathcal{O}$ **faire**
  $q \leftarrow start$;
  $n_{start} \leftarrow n_{start} + 1$;
  pour $t = 1$ à T **faire**
    si $\exists s \in Out(q)$ tel que $n_s^{o_t} \neq 0$ **alors**
      $n_{q \to s} \leftarrow n_{q \to s} + 1$;
      $n_s \leftarrow n_s + 1$;
      $n_s^{o_t} \leftarrow n_s^{o_t} + 1$;
      $q \leftarrow s$;
    sinon
      Créer un nouvel état s;
      Créer la transition $q \to s$;
      $n_s \leftarrow 1$;
      $n_{q \to s} \leftarrow 1$;
      $n_s^{o_t} \leftarrow 1$;
      $q \leftarrow s$;
Créer l'état *end*;
pour chaque état s tel que $Out(s) = \varnothing$ **faire**
  Créer la transition $s \to end$;
  $n_{s \to end} = n_s$;
FusionnerSuffixes;

mêmes symboles avec une probabilité 1, et qu'ils ont toujours les mêmes probabilités de transitions sortantes. À la figure 4.5 sont représentés trois exemples de HMMs construits à partir du même ensemble d'apprentissage. Le HMM de gauche est le HMM construit simplement comme une disjonction des séquences d'apprentissage, celui du milieu est le HMM obtenu en fusionnant les préfixes communs, tandis que celui de droite est obtenu en fusionnant en plus les suffixes communs.

Notons que le HMM initial construit à l'aide de l'algorithme 12 n'est pas obligatoirement celui que l'algorithme d'apprentissage obtient en effectuant les premières fusions (celles qui n'engendrent pas de baisse de la vraisemblance) à partir du HMM construit comme une disjonction des séquences d'apprentissage. En effet, l'ordre avec lequel celui-ci effectue ces fusions n'est pas nécessairement celui utilisé par l'algorithme 12 (il peut, par exemple, commencer par fusionner les suffixes communs et seulement ensuite les préfixes communs). En revanche, ces HMMs ont, du point de vue du critère de fusion utilisé, exactement la même qualité et peuvent donc être utilisés indifféremment.

Les taux de réductions du HMM initial obtenus à l'aide de cette procédure dépendent bien évidemment des séquences manipulées. Si elle permet généralement des taux de

---

Algorithme 13: FUSIONNERSUFFIXES

Données : H, un HMM construit comme un arbre préfixe

$\mathcal{Z} \leftarrow \{end\}$;
tant que $\mathcal{Z} \neq \varnothing$ **faire**
  $\mathcal{Z}' \leftarrow \varnothing$;
  pour chaque $s \in \mathcal{Z}$ **faire**
    pour chaque couple d'états $(t, t') \in In(s) \times In(s)$ **faire**
      si t et t' émettent le même symbole et $\forall u \in Out(t) P(t \rightarrow u) = P(t' \rightarrow u)$ **alors**
        Fusionner t et t' en un état $t \otimes t'$ (algorithme 7);
        $\mathcal{Z}' \leftarrow \mathcal{Z}' \cup \{t \otimes t'\}$;
  $\mathcal{Z} \leftarrow \mathcal{Z}'$;

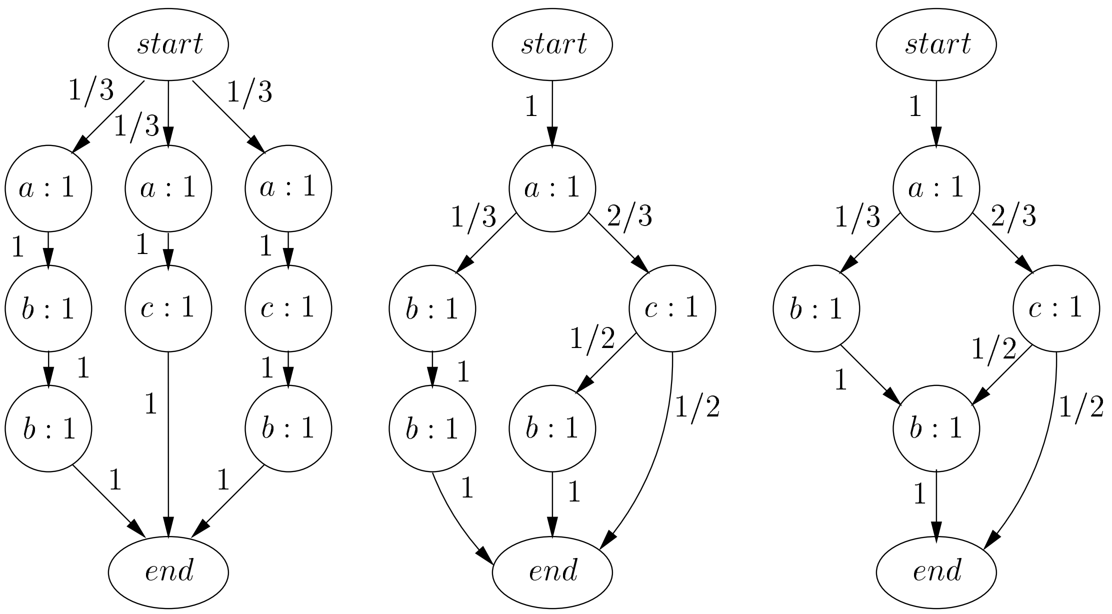


FIG. 4.5: Trois HMMs H_0 construits à partir de l'ensemble de séquences d'apprentissage $\mathcal{O} = \{abb, ac, acb\}$. À gauche, le HMM construit comme une disjonction des séquences de $\mathcal{O}$; au milieu, le même HMM après fusion des préfixes communs ; à droite, le même HMM après fusion des préfixes et des suffixes communs.

réduction de l'ordre de 5 ou 10% seulement, elle peut, pour certains domaines, obtenir des résultats beaucoup plus importants. Comme nous le verrons au chapitre 8 pour les séquences de test de circuits microélectroniques, par exemple, la seule construction de l'arbre préfixe permet dans certains cas de réduire la taille de H_0 de près de 80%.

4.3 Conclusion

Nous avons proposé ici deux solutions pour optimiser la procédure d'apprentissage par fusions d'états utilisant le critère de vraisemblance maximale. La première solution est une implémentation de la procédure de sélection qui diminue la complexité générale de l'algorithme. Celle-ci passe alors de $O(N^3(b + b'))$ à $O(N^2b(b + b' + \log(N)))$. On a pu vérifier en outre que le critère de vraisemblance maximale tends à minimiser le facteur de branchement des états (b). Cette réduction de la complexité a donc un effet particulièrement bénéfique sur le temps de calcul de l'algorithme. La seconde solution adresse un autre côté du problème, et permet de diminuer encore le temps de calcul en réduisant le nombre d'états du HMM initial, sans que cela n'ait d'incidence sur la qualité du HMM inféré.

Chapitre 5

Un nouveau critère de sélection

La qualité d'un HMM inféré à l'aide d'un algorithme par fusions d'états dépend essentiellement du critère de sélection des états employé. Nous avons utilisé jusqu'à présent le critère de vraisemblance maximale qui consiste à sélectionner, à chaque itération, le couple d'états dont la fusion entraîne la plus petite diminution de vraisemblance des données. L'inconvénient majeur de ce critère est qu'il se base, pour juger d'une fusion potentielle entre deux états, uniquement sur les probabilités de transition et d'émission associées aux états considérés. Le peu d'information utilisé par ce critère peut très facilement conduire à effectuer des fusions non judicieuses, et engager le processus de généralisation sur une voie aboutissant à la construction de HMMs mal adaptés aux séquences étudiées. Considérons par exemple le HMM H_0 de la figure 5.1. Ce HMM est construit à partir de l'ensemble d'apprentissage $\mathcal{O} = \{addb, bdda, ccc\}$. Toutes les fusions potentielles des états de ce HMM entraînent une baisse de la vraisemblance des données. Par contre, plusieurs couples d'états engendrent une baisse minimale. Parmi ces couples, il y a par exemple (s, s') et (s, s'') ; la fusion de l'un ou l'autre de ces couples a exactement le même impact sur la vraisemblance des données. Or, pour la suite du processus, la fusion de s et s' semble nettement plus judicieuse. Le HMM obtenu permet en effet par la suite d'effectuer une série de fusions de coût nul – c'est-à-dire n'entraînant pas de diminution de la vraisemblance –, tandis que la fusion de s et s'' ne permet aucune fusion supplémentaire sans perte de vraisemblance. Le critère de vraisemblance maximale n'a aucun moyen de s'apercevoir de cette différence ; pour lui, ces deux fusions ont la même valeur, et il a donc toutes les chances d'engager le processus de généralisation sur la mauvaise voie.

L'idée du critère de sélection que nous proposons est de s'appuyer, pour juger d'une fusion entre deux états u et u', non plus uniquement sur les états considérés, mais également sur l'ensemble des états qui sont accessibles à partir de u et de u'. Ce critère est inspiré de celui utilisé par l'algorithme ALERGIA [Carrasco et Oncina, 1994] dans le cadre des automates probabilistes (voir chapitre 3). Comme dans cet algorithme, nous utilisons un ordre sur les fusions qui garantit qu'au moins un des deux états considérés lors d'une fusion potentielle soit racine d'un arbre. De manière schématique, l'utilisation d'un tel ordre sur les fusions partage les états du modèle en deux groupes : le groupe des états dont l'ordre garantit qu'ils sont racines d'un arbre, et le groupe des états restants – qui peuvent éventuellement être racine d'un arbre mais sans que cela soit implicitement garanti par l'ordre utilisé. Les états appartenant au deuxième groupe ne peuvent pas être

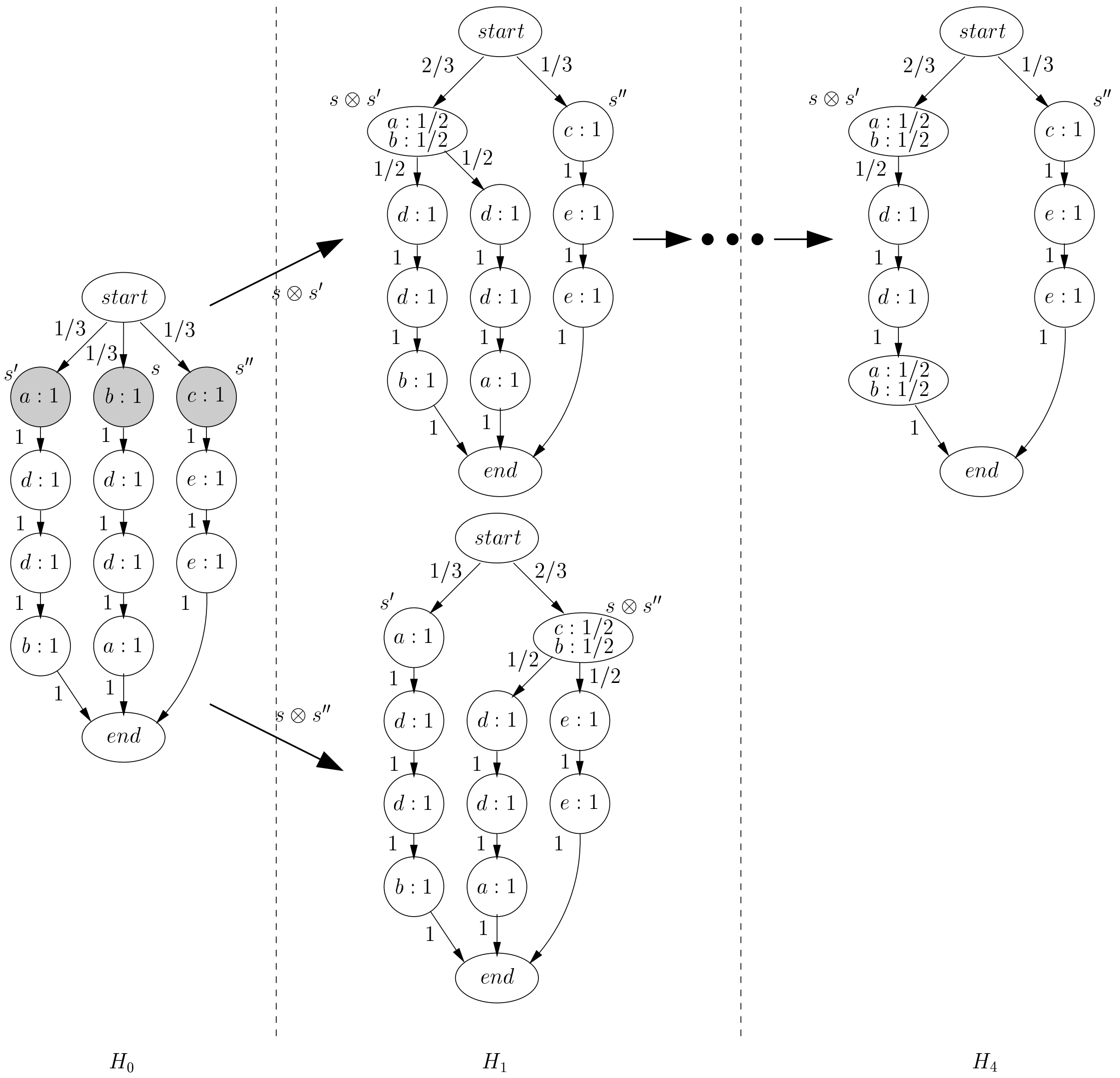


FIG. 5.1: Évolution du processus d'apprentissage par fusions d'états suivant que l'on choisisse de fusionner s et s' (en haut), ou s et s'' (en bas).

fusionnés entre eux, et ne peuvent participer à une fusion qu'avec les états du premier groupe. Une des conséquences est que si l'on compte le nombre d'états du deuxième groupe à un instant donné de l'algorithme, on est sûr que le nombre d'états du modèle obtenu à l'issue de l'apprentissage sera au moins aussi grand. Au début du processus, l'ensemble des états du deuxième groupe se limite à l'état *start*, et il est peu à peu augmenté au fil des fusions effectuées. L'algorithme d'apprentissage consiste alors, à chaque itération, à sélectionner un couple d'états parmi l'ensemble des couples formés d'un état appartenant au premier groupe et d'un état appartenant au premier ou au deuxième groupe et à les fusionner. Dans l'algorithme ALERGIA, le critère de sélection d'un tel couple consiste à tester la compatibilité locale des deux états (en se basant sur les symboles qu'il est possible de générer à partir de ces états), et à étendre ce test récursivement sur leurs états

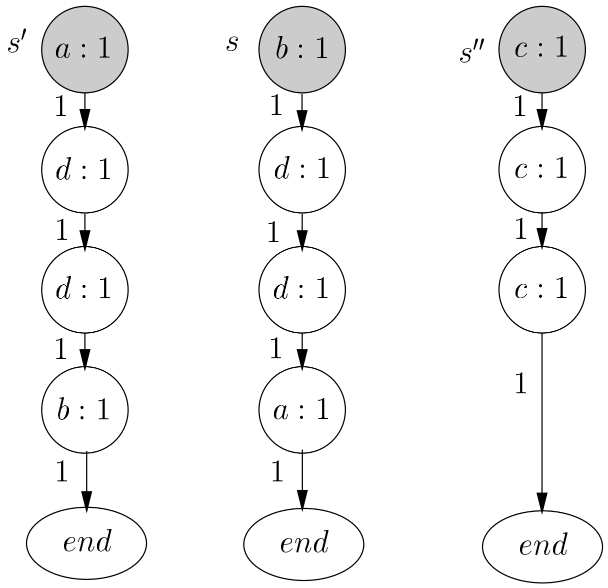


FIG. 5.2: Sous-HMMs associés aux états s, s' et s'' du HMM H_0 de la figure 5.1

fils. ALERGIA utilise pour cela la nature déterministe des PDFAs qui permet d'associer à chaque fils du premier état au plus un fils de l'autre état. Dans le cadre des HMMs, un tel test est impossible du fait du caractère non déterministe de cette classe. Nous proposons d'utiliser à sa place une mesure de similarité – ou plutôt de divergence – entre états, définie à partir des séquences qu'il est possible de générer au départ des deux états.

Une première définition de cette mesure de divergence basée sur la vraisemblance des séquences d'apprentissage est présentée dans la section suivante. Nous proposons ensuite à la section 5.2, une extension de la définition de la divergence qui généralise la définition donnée à la section 5.1, et qui conduit à une nouvelle mesure plus adaptée au cadre d'une heuristique. À la section 5.3, nous présentons un algorithme de programmation dynamique pour le calcul de cette divergence. Nous proposons alors à la section 5.4 un nouvel algorithme d'apprentissage qui utilise la mesure de divergence définie. La section 5.5 est dédiée au paramétrage de cet algorithme et aux différentes solutions envisageables pour réduire son temps de calcul.

5.1 Une nouvelle mesure de divergence

5.1.1 Idée générale

Considérons un ensemble de séquences d'apprentissage $\mathcal{O}$, et un algorithme d'apprentissage de HMMs par fusions d'états. Cet algorithme estime les paramètres du modèle à chaque itération en utilisant la méthode décrite au chapitre 3, c'est-à-dire en maximisant la probabilité de génération des séquences d'apprentissage suivant leurs chemins de Viterbi, et en faisant l'hypothèse de non modification de ces chemins. À un instant donné de cet algorithme, on se trouve en possession d'un HMM H. Soient u et u' deux états de ce HMM dont on désire évaluer la fusion. Dans la suite, nous considérerons que l'ordre sur les fusions utilisé garantit :

1. que u est racine d'un arbre (si l'on fait abstraction de l'état *end*) ;
2. que u' est un état quelconque mais qui ne peut être atteint à partir de u ; notons par contre que u peut potentiellement être atteint à partir de u'.

Remarquons que lorsqu'on dit *u est racine d'un arbre*, il ne s'agit pas obligatoirement d'un arbre préfixe ; certains états de cet arbre (u y compris) sont susceptibles de générer plusieurs symboles de l'alphabet. L'idée générale de notre critère est que pour mesurer la qualité d'une fusion, on doit pouvoir préjuger de la facilité avec laquelle on pourra, par la suite, fusionner les états descendants des deux états fusionnés. Lors de la fusion de deux états, si les distributions d'émission attachées à ces états sont trop dissemblables, la distribution attachée à l'état résultant de la fusion va entraîner beaucoup d'indécision, et donc une forte baisse de la vraisemblance des séquences d'apprentissage. Il en est de même pour les distributions de probabilités de transitions attachées à ces états : si leurs états sortants ne peuvent être fusionnés à leur tour, l'état résultant de la fusion va posséder beaucoup de transitions sortantes, ce qui entraîne une augmentation de l'indécision et une baisse importante de la vraisemblance.

Pour simplifier, nous désignons l'ensemble des états et transitions accessibles à partir d'un état s par le terme *sous-HMM* de s, et nous le notons $H^{(s)}$. Par exemple, la figure 5.2 représente les sous-HMMs des états s, s' et s'' du HMM H_0 de la figure 5.1. Remarquons que le sous-HMM de u est l'arbre dont u est racine. La méthode que nous proposons pour évaluer la difficulté avec laquelle il va être possible de fusionner les états descendants de u et u' est d'utiliser une *projection* de $H^{(u)}$ dans $H^{(u')}$, c'est-à-dire une application M des états de $H^{(u)}$ dans les états de $H^{(u')}$ qui vérifie les trois propriétés suivantes :

$$\forall\ s \to t \in H^{(u)}, M(s) \to M(t) \in H^{(u')}, \tag{5.1}$$

$$M(u) = u', \tag{5.2}$$

et

$$M(end) = end. \tag{5.3}$$

Chacune de ces projections définit un ensemble de fusions potentielles entre les états descendants de u et les états descendants de u' (il suffit de fusionner les états s de $H^{(u)}$ avec leur image $M(s)$ dans $H^{(u')}$). La propriété (5.1) garantit que si deux états t et $M(t)$ sont fusionnés, alors au moins deux de leurs états pères s et $M(s)$ ont également été fusionnés en un état $s \otimes M(s)$, et donc que la fusion de t et $M(t)$ s'accompagne d'une diminution de l'indécision à la sortie de $s \otimes M(s)$. On associe ensuite un coût à chaque projection M, qui soit une mesure de la difficulté (estimée) avec laquelle on va pouvoir réaliser l'ensemble des fusions définies par M. On explore alors l'ensemble des projections de $H^{(u)}$ dans $H^{(u')}$, et le coût de la projection de moindre coût est utilisé pour évaluer la pertinence de la fusion de u et u'. La mise en place de cette méthode nécessite donc :

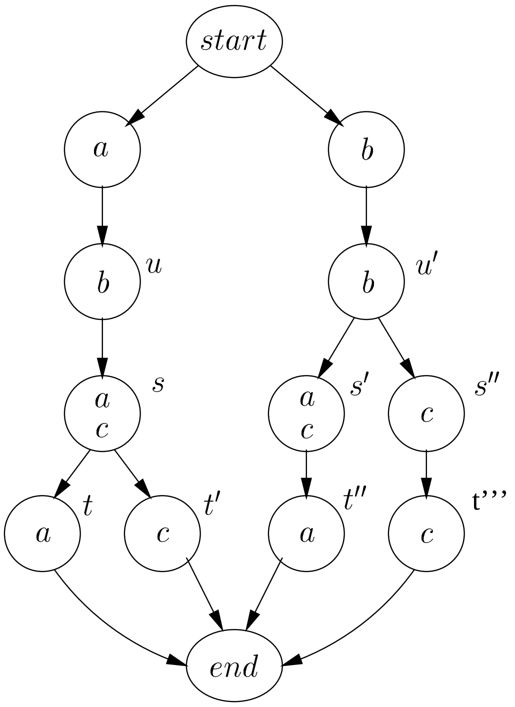


FIG. 5.3: Un HMM obtenu au cour d'un processus d'apprentissage par fusions d'états, à partir de l'ensemble de séquences d'apprentissage $\mathcal{O} = \{abaa, abcc, bbac, bbca, bbcc\}$.

1. une méthode pour estimer la pertinence de l'ensemble de fusions potentielles définie par une projection – c'est-à-dire pour définir le coût de cette projection ;
2. un algorithme efficace pour la recherche de la projection optimale (la projection de moindre coût).

Ce deuxième point est délicat. La recherche d'une projection, telle qu'on l'a définie, s'apparente à celle d'un homomorphisme de $H^{(u)}$ dans $H^{(u')}$ [Levin, 1973]. Dans le cas général, la recherche d'une projection, peut donc être considérée comme un problème difficile. Si $H^{(u)}$ est un arbre (cas dans lequel nous nous trouvons), par contre, il existe un algorithme polynômial qui permet la recherche de la projection optimale. Cet algorithme est décrit dans la partie 5.3. Quant à la définition du coût d'une projection, nous allons voir maintenant que les séquences d'apprentissage nous permettent de définir une fonction de coût très naturelle.

5.1.2 Définition de la divergence

Reprenons les états u et u' dont on désire évaluer la fusion. u étant racine d'un arbre, on peut lui associer un ensemble de séquences, suffixes de l'ensemble d'apprentissage. Le HMM de la figure 5.3, par exemple, a été obtenu à partir de l'ensemble d'apprentissage $\{abaa, abcc, bbac, bbca, bbcc\}$, et à l'état s de ce HMM est associé l'ensemble de suffixes $\{baa, bcc\}$. Notons $\mathcal{O}_s$ l'ensemble des suffixes des séquences de $\mathcal{O}$ associé à un état s. À chacun de ces suffixes est de plus associé un chemin dans le HMM, lui-même suffixe du chemin de Viterbi de la séquence complète, et donc (par définition des chemins de Viterbi) chemin de Viterbi du suffixe au départ de u. On note $\mathcal{S}_u^*$ ces chemins de Viterbi.

Considérons maintenant une projection M de $H^{(u)}$ dans $H^{(u')}$. L'application de M sur les chemins de $\mathcal{S}_u^*$ conduit à un nouvel ensemble de séquences d'états $M(\mathcal{S}_u^*)$. On vérifie aisément que ces séquences définissent des chemins valides au départ de u'. Les propriétés (5.2) et (5.3) garantissent respectivement que chacune de ces séquences commence par u' et se termine par *end*. De plus, la propriété (5.1) implique que pour toute portion de séquence $M(s) - M(s')$, la transition $M(s) \rightarrow M(s')$ est présente dans $H^{(u')}$, et que la séquence d'états produite définit donc bien un chemin valide.

Nous proposons d'évaluer l'ensemble des fusions potentielles défini par la projection M en utilisant la probabilité $P(\mathcal{O}_u, M(\mathcal{S}_u^*))$, qui désigne la probabilité de générer $\mathcal{O}_u$ au départ de u' en suivant les chemins $M(\mathcal{S}_u^*)$. Par extension, nous proposons d'évaluer la pertinence de la fusion de u et u' en utilisant la probabilité maximale de générer $\mathcal{O}_u$ au départ de u' suivant des chemins obtenus par application sur $\mathcal{S}_u^*$ d'une projection de $H^{(u)}$ dans $H^{(u')}$. En d'autres termes, si $\mathcal{M}_{u,u'}$ est l'ensemble des projections de $H^{(u)}$ dans $H^{(u')}$, on définit la projection optimale M^* de $H^{(u)}$ dans $H^{(u')}$ comme la projection qui maximise la probabilité de générer $\mathcal{O}_u$ au départ de u' :

$$M^* = \arg \max_{M \in \mathcal{M}_{u,u'}} P(\mathcal{O}_u, M(\mathcal{S}_u^*)). \tag{5.4}$$

On définit alors la quantité $\mathcal{D}(u||u')$, appelée *divergence de u' par rapport à u* :

$$\mathcal{D}(u||u') = \log \left(\frac{P(\mathcal{O}_u, \mathcal{S}_u^*)}{P(\mathcal{O}_u, M^*(\mathcal{S}_u^*))} \right). \tag{5.5}$$

Plus $\mathcal{D}(u||u')$ est élevée, moins la fusion de u et u' sera considérée comme pertinente. En général, on a $P(\mathcal{O}_u, \mathcal{S}_u^*) \geq P(\mathcal{O}_u, M^*(\mathcal{S}_u^*))$, et donc $\mathcal{D}(u||u') \geq 0$

Le problème de notre mesure est qu'elle peut prendre des valeurs infinies. Il y a deux cas de figure pour cela. Le premier intervient lorsqu'il n'y a tout simplement aucune projection de $H^{(u)}$ dans $H^{(u')}$. On a alors par vacuité $P(\mathcal{O}_u, M(\mathcal{S}_u^*)) = 0$, et donc $\mathcal{D}(u||u') = \infty$. Le second cas de figure se pose lorsqu'il existe bien une projection M de $H^{(u)}$ dans $H^{(u')}$, mais que son coût est infini, c'est-à-dire que la probabilité de génération de $\mathcal{O}_u$ suivant les chemins $M(\mathcal{S}_u^*)$ est nulle. Nous allons voir maintenant que ces deux problèmes peuvent être résolus par lissage des distributions de probabilités du sous-HMM $H^{(u')}$.

5.1.3 Lissage du HMM

Pour des raisons de simplicité, nous allons tout d'abord aborder le problème des projections de coût infini. Nous verrons ensuite comment résoudre celui posé par l'absence de projection.

5.1.3.1 Projections de coût infini

C'est le problème qui se pose, par exemple, lorsque l'on veut calculer la divergence entre les états s et s' du HMM H_0 de la figure 5.1. Il existe bien une projection M de $H^{(s)}$ dans $H^{(s')}$ (il suffit d'utiliser l'application $s \mapsto s'$, $t \mapsto t'$, $u \mapsto u'$, $v \mapsto v'$, $end \mapsto end$) mais la probabilité de générer l'unique séquence *bdda* de $\mathcal{O}_s$ en suivant le chemins $s' - t' - u' - v' - end$ (que l'on obtient en appliquant M sur le chemin de Viterbi de *bdda* au départ de u) est nulle. Ce problème est lié aux distributions de probabilités d'émission attachées aux états du sous-HMM dans lequel est effectuée la projection. Il peut être efficacement résolu par lissage de ces distributions – par exemple par la technique de *Laplace smoothing*. Rappelons que cette technique consiste à calculer les estimateurs de probabilités de manière classique, mais en intégrant une constante l afin d'éviter les probabilités nulles. L'expression de ces estimateurs est alors :

$$\widehat{P}(o|s) = \frac{n_s^o + l}{n_s + l \cdot |\Sigma|}. \tag{5.6}$$

L'utilisation de ces estimateurs garantit que pour toute séquence de symboles O, et tout chemin S de même longueur que O, la probabilité de générer O suivant S est non nulle.

5.1.3.2 Absence de projection

Ce problème peut lui-même avoir deux types de causes. La première est liée à l'absence de transition vers l'état *end*. C'est le cas par exemple entre les sous-HMMs $H^{(s'')}$ et $H^{(s)}$ de la figure 5.2. On peut proposer une application de $H^{(s'')}$ dans $H^{(s)}$ en associant s'' à l'état s, t'' à t, et u'' à u, mais *end* ne peut être associé à *end* sans violer la propriété 5.1. Pour résoudre ce problème, il suffit de garantir que chaque état du sous-HMM dans lequel est effectuée la projection ait une probabilité de transition non nulle vers l'état *end*.

Le second problème responsable de l'absence de projection d'un sous-HMM $H^{(u)}$ dans un sous-HMM $H^{(u')}$ se pose lorsque la profondeur de $H^{(u)}$ est plus importante que celle de $H^{(u')}$. En d'autres termes, il existe dans $\mathcal{O}_u$ une séquence dont la longueur est supérieure à tout chemin qu'il est possible de tracer à partir de u'. C'est le cas par exemple entre les sous-HMMs $H^{(s)}$ et $H^{(s'')}$ de la figure 5.2. On peut associer l'état s à l'état s'', l'état t à l'état t'' et l'état u à l'état u'', mais on ne peut associer l'état v à aucun état de $H^{(s'')}$ sans violer la propriété (5.1). Un moyen simple de remédier à ce problème est de rajouter au sous-HMM un nouvel état spécial – noté *smooth* et appelé *état de lissage* –, à partir duquel il soit possible de générer toutes les séquences de Σ^* avec une probabilité non nulle. L'état *smooth* génère tous les symboles de Σ et possède deux transitions sortantes : une sur lui-même et une sur *end* (voir l'exemple de la figure 5.4).

On peut lisser les distributions de probabilité de transition, de manière à ce que chaque état ait une transition de probabilité non nulle vers l'état *end* et l'état *smooth*, à l'aide de la technique de Laplace *smoothing*. L'expression des estimateurs des probabilités de

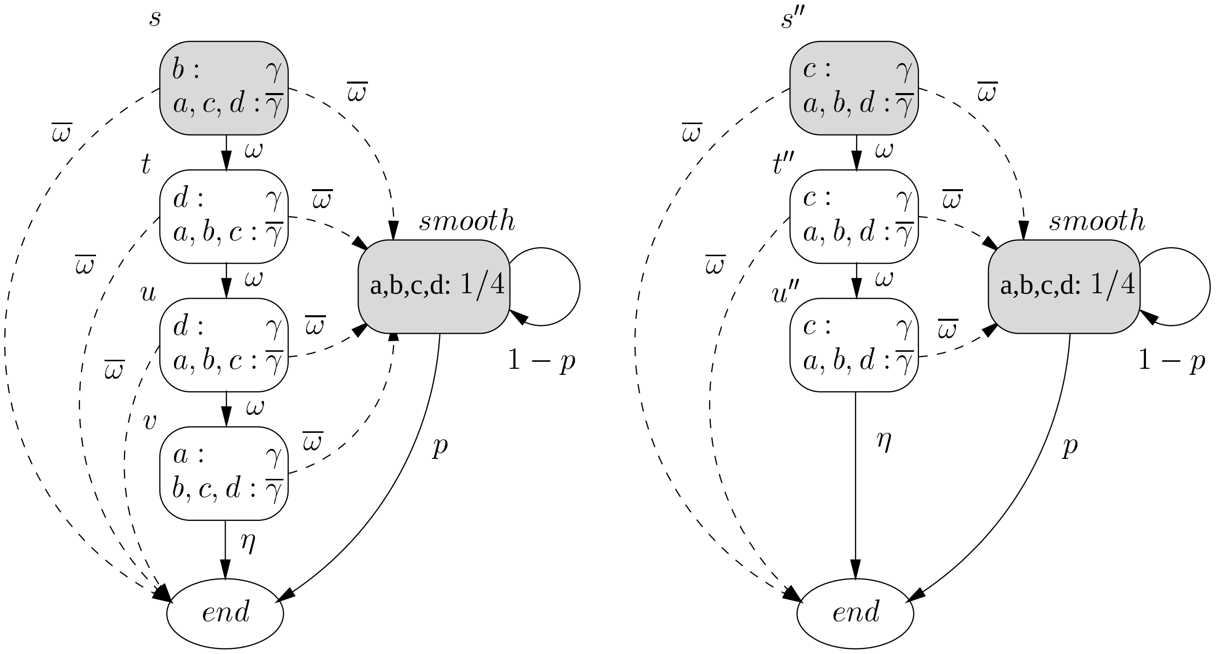


FIG. 5.4: Sous-HMMs de la figure 5.2 après lissage des distributions de probabilités de génération et de transition par la technique de *Laplace smoothing* et ajout de l'état *smooth* – voir expressions (5.6) et (5.7). On a $\gamma = (1+l)/(1+4l)$, $\overline{\gamma} = l/(1+4l)$, $\omega = 1/(1+2l')$, $\overline{\omega} = l'/(1+2l')$, et $\eta = (1+l')/(1+2l')$.

transitions devient alors :

$$\widehat{P}(s \to s') = \frac{n_{s\to s'} + \delta(s')l'}{n_s + 2l'}, \tag{5.7}$$

avec $\delta(s')$ une fonction binaire qui renvoie 1 si s' appartient à l'ensemble $\{smooth, end\}$, et 0 sinon.

Plusieurs solutions sont possibles pour définir les distributions de probabilités attachées à l'état *smooth*. On peut par exemple, comme c'est le cas sur la figure 5.4, utiliser une distribution de probabilités d'émission équiprobable. Une autre solution est d'utiliser les fréquences d'apparitions relatives de chaque symbole dans les séquences d'apprentissage. Pour les probabilités de transitions attachées à cet état, la solution la plus simple est, de la même manière, d'utiliser les fréquences de l'ensemble d'apprentissage. On remarque que le nombre de fois X où on va rester dans l'état *smooth* avant d'aller sur *end* suit une loie géométrique : $P(X = k) = (1-p)^k p$ (p désigne, comme sur la figure, la probabilité de transition sur l'état *end*). Une solution est alors de chercher la valeur de p telle que l'espérance de X soit égale à la taille moyenne des séquences d'apprentissage, c'est-à-dire

$$E(X) = (1-p)/p = \frac{\text{taille totale}}{\#\text{ séquences}},$$

et on obtient

$$p = \frac{\#\text{séquences}}{\#\text{séquences} + \text{taille totale}}.$$

5.1.4 Expression de la divergence

Intéressons nous à l'expression que prend la divergence après ajout de l'état *smooth*, et lissage des probabilités d'émission et de transition. Dans la suite, nous noterons $\widetilde{P}(.)$ la valeur d'une probabilité calculée à partir des distributions lissées, et nous laisserons $P(.)$ pour la valeur de la probabilité originale (non lissée). Par exemple, $\widetilde{P}(o|s)$ désigne la probabilité de générer le symbole o à partir de l'état s en utilisant la distribution de probabilité d'émission lissée. Avec cette convention typographique, la définition de la divergence devient :

$$\mathcal{D}(u||u') = \log\left(\frac{P(\mathcal{O}_u, \mathcal{S}_u^*)}{\widetilde{P}(\mathcal{O}_u, M^*(\mathcal{S}_u^*))}\right).$$

De la même manière que, dans l'expression (2.13), la vraisemblance des séquences d'apprentissage suivant leurs chemins de Viterbi peut être calculée par un produit des probabilités d'émission et de transition associées aux états du HMM, la probabilité de génération de $\mathcal{O}_u$ suivant les chemins de Viterbi au départ de u peut être calculée par un produit des probabilités d'émission et de transition associées aux états de $H^{(u)}$:

$$P(\mathcal{O}_u, \mathcal{S}_u^*) = \prod_{s\in H^{(u)}}\left(\prod_{o\in\Sigma_s} P(o|s)^{n_s^o} \prod_{s'\in In(s)} P(s'\to s)^{n_{s'\to s}}\right),$$

et on remarquera que, puisque $H^{(u)}$ est un arbre, pour chaque état $s \neq end$ de $H^{(u)}$, $In(s)$ ne contient qu'un seul état, et $In(u)$ est l'ensemble vide. Puisque M^* est une application des états de $H^{(u)}$ dans les états de $H^{(u')}$, l'expression de $\widetilde{P}(\mathcal{O}_u, M^*(\mathcal{S}_u^*))$ se dérive tout simplement de l'expression précédente en utilisant

$$\widetilde{P}(\mathcal{O}_u, \mathcal{S}_u^*) = \prod_{s\in H^{(u)}}\left(\prod_{o\in\Sigma_s} \widetilde{P}(o|M^*(s))^{n_s^o} \prod_{s'\in In(s)} \widetilde{P}(M^*(s')\to M^*(s))^{n_{s'\to s}}\right).$$

En intégrant la formule (5.4) à cette dernière expression on obtient :

$$\widetilde{P}(\mathcal{O}_u, M^*(\mathcal{S}_u^*)) = \max_{M\in\mathcal{M}_{u,u'}} \prod_{s\in H^{(u)}}\left(\prod_{o\in\Sigma_s} \widetilde{P}(o|M(s))^{n_s^o} \prod_{s'\in In(s)} \widetilde{P}(M(s')\to M(s))^{n_{s'\to s}}\right).$$

La divergence prend alors la forme :

$$\begin{aligned}\mathcal{D}(u||u') \;=\; \min_{M\in\mathcal{M}_{u,u'}} \sum_{s\in H^{(u)}}\Bigg(\sum_{o\in\Sigma_s} n_s^o \log\left(\frac{P(o|s)}{\widetilde{P}(o|M(s))}\right) +\\ \sum_{s'\in In(s)} n_{s'\to s}\log\left(\frac{P(s'\to s)}{\widetilde{P}(M(s')\to M(s))}\right)\Bigg). \qquad (5.8)\end{aligned}$$

5.2 Extension de la définition de la divergence

De l'expression (5.8), il ressort que la divergence $\mathcal{D}(u||u')$ s'exprime comme le coût minimal de la projection de moindre coût. Le coût d'une projection se calcule comme la somme des coûts locaux $D(s||M(s))$ attachés à chaque projection $s \mapsto M(s)$, et les coûts locaux sont calculés grâce à l'expression

$$D(s||M(s)) = \sum_{o \in \Sigma_s} n_s^o \log\left(\frac{P(o|s)}{\widetilde{P}(o|M(s))}\right) + \sum_{s' \in In(s)} n_{s' \to s} \log\left(\frac{P(s' \to s)}{\widetilde{P}(M(s') \to M(s))}\right).$$

Le problème de cette mesure est qu'elle donne le même poids à tous les états descendants de u, indépendamment de leur distance à cet état. Or, ce que l'on recherche pour juger de l'opportunité d'une fusion dans le cadre d'une heuristique est qu'elle permette d'effectuer un *nombre raisonnable* de fusions sans baisse importante de la vraisemblance des séquences d'apprentissage. Dans cette optique, une solution possible est d'accorder un poids différent aux quantités $D(s||M(s))$ en fonction de la distance de l'état s à l'état u. Il est raisonnable en effet que l'état u ait plus de poids dans le calcul de sa divergence à l'état u' que tout autre état de $H^{(u)}$; de la même manière, il est logique que les états proches de u aient plus de poids que les états qui en sont éloignés. Nous proposons alors d'utiliser la définition suivante :

$$\mathcal{D}(u||u') = \min_{M \in \mathcal{M}_{u,u'}} \sum_{s \in H^{(u)}} \alpha^{u \rightsquigarrow s} D(s||M(s)) \tag{5.9}$$

avec $0 \leq \alpha \leq 1$ une constante, et $u \rightsquigarrow s$ la distance du chemin entre l'état u et l'état s avec la convention $u \rightsquigarrow u = 0$ (rappelons que puisque $H^{(u)}$ est un arbre, il existe un unique chemin entre u et un état de cet arbre). La valeur de α détermine le poids des états de $H^{(u)}$ en fonction de leur distance à l'état u : pour $\alpha = 0$, seul l'état u a un poids dans le calcul de la divergence, tandis que pour $\alpha = 1$ tous les états ont le même poids indépendamment de leur distance à u, et on se retrouve alors dans le cadre de la première définition de la divergence (expression (5.5)).

5.3 Procédure de calcul de la divergence

Considérons u, un état du HMM qui soit racine d'un arbre. Nous proposons ici un algorithme efficace pour le calcul de la divergence $\mathcal{D}(u||u')$ d'un état u' par rapport à u. C'est un algorithme de programmation dynamique similaire à l'algorithme *backward* exposé au chapitre 2. Il a, comme nous allons le voir, l'avantage de pouvoir calculer, en une seule fois, l'ensemble des divergences $\mathcal{D}(u||u')$ de tous les états u' du HMM par rapport à l'état u, ce qui sera d'un grand avantage pour la complexité de notre procédure d'apprentissage.

Abstraction faite de l'état *end*, le sous-HMM $H^{(u)}$ est un arbre. Les états entrants dans l'état *end* sont désignés comme les feuilles de cet arbre. On effectue pour commencer une numérotation en largeur des états de $H^{(u)}$ en partant des feuilles : les feuilles ont les numéros les plus bas, chaque état qui n'est pas une feuille possède un numéro supérieur aux numéros de ses états fils, et l'état u possède alors le numéro maximum de l'arbre. Cette numérotation est utilisée pour désigner les états du sous-HMM : si $H^{(u)}$ est un sous-HMM à K états, ceux-ci sont désignés $1, \ldots, K$, et on a $K = u$. $H^{(u)}$ étant un arbre, chacun de ses états est également la racine d'un arbre. On désigne par $H^{(k)}$ le sous-HMM associé à l'état k du sous-HMM $H^{(u)}$. Considérons maintenant la variable $\mu_k(s)$ qui désigne le coût minimal d'une projection de $H^{(k)}$ dans un sous-HMM quelconque $H^{(s)}$:

$$\mu_k(s) = \min_{M \in \mathcal{M}_{k,s}} \sum_{k' \in H^{(k)}} \alpha^{u \rightsquigarrow k'} D(k'||M(k')).$$

Cette variable peut être calculée récursivement de la manière suivante :

1. Initialisation. Pour chaque feuille k de $H^{(u)}$ on a :

$$\mu_k(s) = \alpha^{u \rightsquigarrow k} \sum_{o \in \Sigma_k} n_k^o \log\left(\frac{P(o|k)}{\widetilde{P}(o|s)}\right) + \alpha^{u \rightsquigarrow k+1} n_{k \to end} \log\left(\frac{P(k \to end)}{\widetilde{P}(s \to end)}\right). \quad (5.10)$$

2. Récursion. Pour chaque état k non feuille :

$$\begin{aligned}\mu_k(s) \quad = \quad & \alpha^{u \rightsquigarrow k} \sum_{o \in \Sigma_k} n_k^o \log\left(\frac{P(o|k)}{\widetilde{P}(o|s)}\right) + & (5.11)\\ & \sum_{k' \in Out(k)} \min_{s' \in Out(s)} \left(\alpha^{u \rightsquigarrow k+1} n_{k \to k'} \log\left(\frac{P(k \to k')}{\widetilde{P}(s \to s')}\right) + \mu_{k'}(s')\right).\end{aligned}$$

Pour la phase d'initialisation, l'expression (5.10) est due aux propriétés (5.2) et (5.3) des projections de $\mathcal{M}_{k,s}$: $H^{(k)}$ n'est, dans ce cas, composé que de deux états (k et *end*), et puisque k se projette sur s (propriété (5.2)) et *end* sur *end* (propriété (5.3)), il n'existe qu'une projection possible des états $H^{(k)}$ sur les états de $H^{(s)}$. En ce qui concerne la phase de récursion, l'expression (5.11) est vraie du fait que $H^{(k)}$ soit un arbre, et en vertu des propriétés (5.2) et (5.1) des projections de $\mathcal{M}_{k,s}$: k se projette sur s (propriété (5.2)), et on peut donc directement calculer la partie relative aux distributions d'émission dans le coût local de cette projection (partie haute de l'expression 5.11). Ensuite, du fait que $\mu_{k'}(s')$ est le coût minimal de la projection du sous-HMM de racine k', fils de k, sur le sous-HMM de racine s', fils de s, il suffit, en vertu de la propriété (5.1), d'assigner aux fils k' de k les fils s' de s tels que la valeur de l'expression

$$n_{k \to k'} \log\left(\frac{P(k \to k')}{\widetilde{P}(s \to s')}\right) + \mu_{k'}(s')$$

soit minimale. On obtient alors, à la fin de la procédure de récursion, $\mu_K(s) = \mathcal{D}(K||s) = \mathcal{D}(u||s)$.

Si l'on calcule, à chaque étape de l'algorithme, la valeur $\mu_k(s)$ pour *tous* les états s du HMM, on connaît donc, à la fin de la procédure, la divergence par rapport à l'état u de

Algorithme 14: PROCÉDURE DE CALCUL DE LA DIVERGENCE

Données : H un HMM
H^u un sous-HMM de H

pour chaque état feuille k de H^u **faire**
  pour chaque état s de H **faire** Calculer $\mu_k(s)$ à l'aide de la formule (5.10) ;

pour chaque état non-feuille k de H^u pris dans l'ordre de la numérotation **faire**
  pour chaque état s de H **faire** Calculer $\mu_k(s)$ à l'aide de la formule (5.11) ;

tous les états u' du HMM. C'est la procédure illustrée par l'algorithme 14. Étudions la complexité de cette procédure en reprenant les constantes b et b' qui désignent respectivement le nombre maximal de transitions adjacentes à chaque état, et le nombre maximal de symboles générés par chaque état. La phase d'initialisation requiert $O(b')$ opérations pour chaque feuilles du sous-HMM et chaque état s. Lors de la phase de récursion, pour chaque état du sous-HMM et chaque état du HMM, on réalise $O(b^2 + b')$ opérations. Si on borne le nombre d'états du sous-HMM par une constante K, la complexité totale de la procédure est $O(KN(b^2 + b'))$.

5.4 Algorithme et ordre des fusions

Nous allons maintenant voir comment utiliser notre mesure de divergence dans un nouvel algorithme d'apprentissage par fusions d'états. Le principe général de l'algorithme consiste, à chaque tour de la procédure, à calculer la divergence d'un ensemble de couples d'états, à sélectionner le couple qui possède la divergence minimale et à le fusionner si cette divergence est inférieure à un certain seuil τ fixé. La clef de l'algorithme réside dans l'ordre utilisé pour fusionner les états. Comme pour l'algorithme ALERGIA, cet ordre doit garantir que pour chaque fusion potentielle envisagée, au moins un des deux états soit racine d'un arbre. Ce genre de restriction a été utilisée dans de nombreux autres travaux sur l'inférence d'automates, y compris non probabilistes – voir, par exemple, [Lang, 1992], [Oncina et Garcia, 1992], [de la Higuera et al., 1996] et [Lang et al., 1998]. Nous reprenons ici la terminologie de [Lang et al., 1998] – basée sur l'utilisation de couleurs pour marquer les états du HMM – pour détailler cette procédure.

Le HMM initial H_0 est, comme on l'a déjà vu, le HMM construit comme une disjonction des séquences d'apprentissage. L'état *start* est coloré en rouge, ses fils sont bleus, et tous les autres états sont blancs. Tout au long de l'algorithme, nous maintenons les invariants suivants :

- les états fils d'un état rouge peuvent être rouges ou bleus ;
- les états bleus et blancs sont racines d'un arbre ;
- les états fils d'un état bleu sont blancs ;
- les états fils d'un état blanc sont blancs.

Les opérations que l'on s'autorise sont les suivantes :

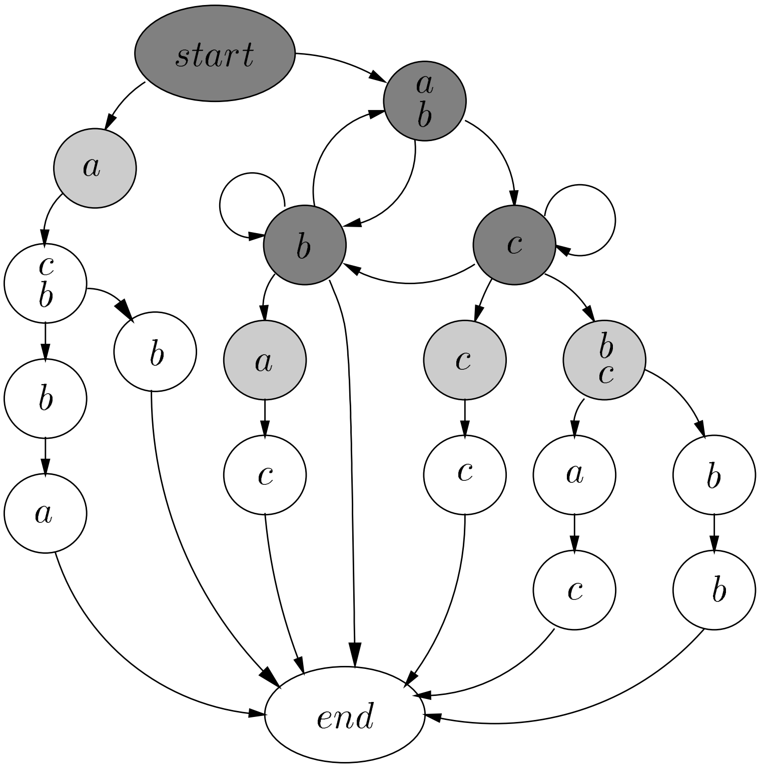


FIG. 5.5: Un HMM au cours de l'algorithme d'apprentissage. Les états rouges, bleus et blancs sont respectivement représentés ici en gris foncé, gris clair et blanc.

– calculer la divergence d'un état rouge par rapport à un état bleu, d'un état bleu par rapport à un autre état bleu, ou d'un état blanc par rapport à un autre état blanc ;
– fusionner un état bleu avec un état rouge (l'état résultant est rouge), un état bleu et un autre état bleu (l'état résultant est bleu), ou un état blanc avec un autre état blanc (l'état résultant est blanc) ;
– colorer un état bleu en rouge ou un état blanc en bleu.

Cette restriction sur les opérations garantit que lors de chaque fusion potentielle considérée – chaque fois que l'on calcule la divergence d'un état par rapport à un autre – au moins un des deux états est bleu ou blanc, et donc racine d'un arbre. La promotion des états se fait dans l'ordre blanc → bleu → rouge, et quand un état devient rouge il ne peut plus être fusionné avec un autre état rouge. La figure 5.5 représente un HMM au cours de l'algorithme d'apprentissage.

Différentes stratégies respectant ces restrictions et invariants peuvent être utilisées lors de la procédure d'apprentissage. L'efficacité de l'algorithme en terme de temps de calcul et de qualité du HMM inféré dépend fortement de la stratégie utilisée. De manière générale, plus le nombre de fusions que l'on considère à chaque étape de l'algorithme pour choisir une fusion effective est important, plus le temps de calcul est long mais plus le processus a de chances d'aboutir à un HMM de vraisemblance élevée. Nous proposons ici d'utiliser la stratégie suivante :

1. fusionner en priorité les états frères, c'est-à-dire les états qui ont un état père com-

mun ;

2. si aucun couple d'états frères ne peut être fusionné, calculer la divergence de tous les états rouges et bleus par rapport à tous les états bleus et fusionner le couple de divergence minimale si celle-ci est inférieure au seuil fixé ;
3. si aucun des couples étudiés dans les points 1 et 2 ne peuvent être fusionnés, colorer les états bleus en rouge, et leurs fils en bleu.

Au début de la procédure, les seuls états frères existants sont les états fils de *start*. Ensuite, chaque nouvelle fusion est susceptible de créer de nouveaux états frères. On va donc utiliser une procédure récursive spéciale appelée FUSIONNERÉTATSFRÈRES(s) qui compare la divergence des états fils bleus ou blancs d'un état s avec tous les autres états fils de s, fusionne le couple de divergence minimale s'il est inférieur au seuil fixé et recommence tant qu'il existe des fusions possibles. La raison pour laquelle on fusionne les états frères en priorité – c'est-à-dire avant de les comparer aux autres états – est que leur fusion n'a pas le même impact sur la vraisemblance des données qu'une fusion classique. La fusion de deux états frères entraîne en effet la suppression d'une transition sortante, et on observe alors une augmentation de la valeur du terme $\prod_{s' \in Out(s)} P(s \to s')^{n_{s \to s'}}$ dans l'expression de la vraisemblance – voir expression (2.13). La stratégie de fusion choisie est illustrée en détail par l'algorithme 15. L'algorithme 16 présente quant à lui la procédure FUSIONNERÉTATSFRÈRES. Dans ces algorithmes les ensembles d'états rouges, bleus et blancs sont notés respectivement $\mathcal{R}$, $\mathcal{B}$ et $\mathcal{W}$.

L'algorithme 15 commence par fusionner les états frères (c'est-à-dire les états fils de l'état *start*) qui peuvent l'être (ligne 1). Ensuite, à chaque tour de l'algorithme (ligne 2), on calcule la divergence de tous les états rouges et de tous les états bleus par rapport à tous les états bleus (boucle de la ligne 3). Une seule exécution de la procédure de calcul de la divergence (algorithme 14) suffit à cela. Une fois que cette opération a été effectuée pour tous les états bleus, on fusionne la paire d'états de divergence minimale si elle est inférieure à la borne τ (ligne 4). On colore alors l'état résultant de la fusion en fonction de la couleur des états fusionnés (ligne 5), et on colore ensuite ses états fils en conséquence (ligne 6). Si un des deux états fusionnés est rouge alors l'état résultant de la fusion est également rouge et ses fils sont bleus ou rouges ; sinon l'état est bleu et ses fils sont blancs. On lance alors une procédure de fusion des états fils du nouvel état (ligne 7) et on recommence la procédure tant qu'une nouvelle fusion est possible. Si ce n'est pas le cas, on colore les états bleus en rouge et leurs fils en bleu (ligne 8). L'apprentissage s'arrête lorsqu'il n'y a plus que des états rouges.

L'algorithme 16 calcule la divergence de tous les état fils de s par rapport à tous les états fils de couleur bleue ou blanche (boucle de la ligne 1) et fusionne le couple de divergence minimale si celle-ci est inférieure à la borne τ (ligne 2). Il colore ensuite le nouvel état et ses états fils en fonction de la couleur des deux états fusionnés (lignes 3 et 4) et relance la procédure sur les états fils du nouvel état (ligne 5). La procédure est répétée tant qu'il existe des états frères qui peuvent être fusionnés.

Étudions la complexité de la procédure d'apprentissage obtenue. À chaque tour de l'algorithme, on calcule la divergence de tous les états rouges et bleus par rapport à tous les états bleus à l'aide de l'algorithme 14. Cet algorithme est donc exécuté un nombre

Algorithme 15: ALGORITHME D'APPRENTISSAGE PAR FUSIONS D'ÉTATS UTILISANT LA MESURE DE DIVERGENCE.

Données : $\mathcal{O}$ un ensemble de séquences d'apprentissage
τ une borne maximale

Créer un HMM initial H_0 qui maximise la vraisemblance de $\mathcal{O}$;
`couleur`$(start) \leftarrow$ `rouge`;
pour chaque état $s \in Out(start)$ **faire**
  `couleur`$(s) \leftarrow$ `bleu`;
FUSIONNERÉTATSFRÈRES$(start)$;
tant que $\mathcal{B} \neq \varnothing$ **faire**
  pour chaque $u \in \mathcal{B}$ **faire**
    Exécuter l'algorithme 14;
  Sélectionner le couple d'états $(u, u') \in \mathcal{B} \times \mathcal{B} \cup \mathcal{R}$ tel que $\mathcal{D}(u||u')$ est minimal;
  si $\mathcal{D}(u||u') < \tau$ **alors**
    Fusionner u et u' ;
    si `couleur`(u') =`rouge` **alors**
      `couleur`$(u \otimes u') \leftarrow$ `rouge` ;
      pour chaque état $s \in Out(u \otimes u')$ **faire**
        si `couleur`(s) =`blanc` **alors** `couleur`$(s) \leftarrow$ `bleu`;
    sinon `couleur`$(u \otimes u') \leftarrow$ `bleu` ;
    FUSIONNERÉTATSFRÈRES$(u \otimes u')$;
  sinon Colorer les états bleus en rouge et leurs fils en bleu;

de fois égal au nombre d'états bleus du HMM. Si on borne ce nombre par une constante B, la phase de calcul des divergences à chaque tour de l'algorithme requiert donc $O(BKN(b^2 + b'))$ opérations. De la même manière, la phase de calcul des divergences dans l'algorithme 16 est en $O(bKN(b^2 + b'))$. L'une ou l'autre de ces phases de calcul sont à faire au total $O(N)$ fois, et la complexité de l'algorithme d'apprentissage est donc $O((b + B)KN^2(b^2 + b'))$.

5.5 Temps de calcul et paramétrage de l'algorithme

5.5.1 Réduction du temps de calcul

On a vu que la complexité de l'algorithme 14 est $O(KN(b^2 + b'))$. Plusieurs stratégies sont possibles pour réduire le temps de calcul de cette procédure. Une première solution est de réduire le nombre d'états K du sous-HMM. Il peut en effet être avantageux de ne pas construire ce sous-HMM complètement mais de se limiter à une certaine profondeur d préalablement fixée. Cette profondeur peut par exemple être liée à la valeur de α choisie : plus importante lorsque α est proche de 1, plus faible lorsque α s'en éloigne. La valeur

Algorithme 16: FUSIONNERÉTATSFRÈRES

Données : s un état du HMM
τ une borne maximale

tant que fusions possibles **faire**
  pour chaque état $s' \in (Out(s) \cap (\mathcal{B} \cup \mathcal{W}))$ **faire**
    Exécuter l'algorithme 14;
  Sélectionner le couple d'état $(u, u') \in (Out(s) \cap (\mathcal{B} \cup \mathcal{W})) \times Out(s)$ tel que $\mathcal{D}(u||u')$ est minimal;
  si $\mathcal{D}(u||u') < \tau$ **alors**
    Fusionner u et u' ;
    si `couleur`(u') =`rouge` **alors**
      `couleur`$(u \otimes u') \leftarrow$ `rouge`;
      pour chaque état $t \in Out(u \otimes u')$ **faire**
        si `couleur`(t) =`blanc` **alors** `couleur`$(t) \leftarrow$ `bleu`;
    sinon
      si `couleur`(u) =`bleu` ou `couleur`(u') =`bleu` **alors**
        `couleur`$(u \otimes u') \leftarrow$ `bleu`;
      sinon `couleur`$(u \otimes u') \leftarrow$ `blanc`;
    FUSIONNERÉTATSFRÈRES$(u \otimes u')$;

de la divergence calculée est alors une approximation de la divergence réelle. Notons que cela peut engendrer une légère modification de la procédure de calcul de la divergence. En effet, si $H^{(u)}$ a été élagué à une certaine profondeur fixée, cela signifie que certains des états de ce sous-HMM que l'on appelle *les feuilles* ne sont plus reliées à l'état *end* (elles deviennent des feuilles au sens propre du terme). On a alors, dans l'expression (5.10), $n_{k \to end} = 0$, et le terme $\mu_k(s)$ de ces états s'initialise simplement avec

$$\mu_k(s) = \alpha^{u \rightsquigarrow k} \sum_{o \in \Sigma_k} n_k^o \log \left(\frac{P(o|k)}{\widetilde{P}(o|s)} \right).$$

D'autre part, du fait de l'ordre des fusions adopté, il apparaît évident que, dans l'algorithme 14, calculer les valeurs $\mu_k(s)$ de tous les états s du HMM est loin d'être une procédure optimale. Seules les valeurs $\mu_k(s)$ associées aux états descendants des états pour lesquels on désire connaître la divergence par rapport à u (c'est-à-dire les états bleus et rouges pour la procédure décrite par l'algorithme 15, et les états frères de u pour celle de l'algorithme 16) sont en effet concernés par cette procédure. Une solution est donc d'effectuer, préalablement à l'algorithme 14, une exploration du HMM à partir de ces états, et de considérer uniquement l'ensemble des états s atteints par cette exploration dans l'algorithme 14.

Enfin, comme pour la procédure de sélection classique, il est possible de construire le HMM initial, non pas comme une simple disjonction de l'ensemble d'apprentissage, mais comme un arbre préfixe de ces séquences. Même si, dans ce cas, ce choix a un impact certain sur le résultat de l'apprentissage, l'arbre préfixe peut néanmoins être un point de départ intéressant pour la procédure.

Utilisées conjointement, ces stratégies contribuent à une nette réduction du temps de calcul de la procédure d'apprentissage qui se révèle alors en pratique – comme nous le voyons au chapitre suivant – plus rapide, non seulement que la procédure utilisant le critère de sélection classique, mais également que les procédures d'entraînement classiques comme la procédure d'entraînement de Viterbi.

5.5.2 Paramétrage de l'algorithme

Plusieurs paramètres contrôlent le déroulement de l'algorithme d'apprentissage :

- l et l', les constantes de Laplace utilisées pour lisser respectivement les distributions de probabilités d'émission et de transitions du HMM ;
- α, qui sert à pondérer les états du HMM en fonction de leur distance à l'état u dans le calcul de la divergence ;
- d, la profondeur maximale du sous-HMM $H^{(u)}$ utilisée pour réduire le temps de calcul de la divergence ;
- τ, la valeur de divergence maximale acceptée pour procéder à une fusion.

Un nombre aussi important de paramètres peut constituer un problème lors de l'utilisation de l'algorithme. Plusieurs de ces paramètres sont néanmoins étroitement liés, et on peut proposer quelques pistes et valeurs types pour simplifier le paramétrage.

- La valeur de l doit être liée à la taille de l'alphabet Σ si l'on veut que les estimateurs $P(o|s)$ de la formule 5.6 ne soient pas trop éloignés des données. Typiquement, une valeur possible est de choisir

$$l = \frac{1}{|\Sigma|} \text{ ou } l = \frac{2}{|\Sigma|}.$$

- De la même manière, une valeur raisonnable pour l' est 1 ou 2.
- La valeur de d est liée à celle de α et définit la finesse de l'approximation de la valeur réelle de la divergence. On peut alors choisir d de manière à ce que le poids des états de cette profondeur dans le calcul de la divergence soit par exemple inférieur à $\alpha/10$ et on obtient

$$d = \lceil \frac{\log(\alpha/10)}{\log \alpha} \rceil$$

- Une manière possible de définir la valeur de τ est de fixer en fait une valeur maximale τ_p pour le coût local $D(s||M(s))$ d'une projection $s \mapsto M(s)$, et d'utiliser alors

$$\tau = \sum_{i=0}^{d} \alpha^i \tau_p.$$

On peut ensuite se faire une idée de la valeur à donner à τ_p en remarquant que dans le cas simple où les sous-HMMs $H^{(u)}$ et $H^{(u')}$ se composent tous deux d'une simple chaîne d'états (leurs états ne possèdent qu'une transition sortante) qui ne sont utilisés qu'une seule fois dans les chemins de Viterbi ($n_s = 1, \forall\, s \in H^{(u)}$), alors, si $s \in H^{(u)}$ et $M(s) \in H^{(u')}$ n'émettent pas le même symbole, on a

$$D(s||M(s)) = \log\left(\frac{1+l}{l}\right) + \log(1+l').$$

Appelons τ_0 cette quantité. Une solution possible est alors de chercher la valeur à donner à τ_p parmi $\tau_0/2$, $\tau_0/3$, $\tau_0/4$, ...

5.6 Conclusion

Nous avons proposé un nouvelle mesure pour évaluer la fusion de deux états u et u'. L'idée de notre mesure est que pour juger la pertinence de cette fusion, on doit être capable d'évaluer la difficulté avec laquelle on pourra, par la suite, fusionner les états descendants de u et u'. On utilise pour cela un ordre sur les fusions qui nous garantisse que u est racine d'un arbre, ainsi qu'une fonction de projection de $H^{(u)}$ dans $H^{(u')}$. Cette projection définit à la fois un ensemble de fusions potentielles entre les états des deux sous-HMMs, et un ensemble de chemins au départ de u' qui permettent de générer les séquences d'apprentissage attachées à u. Le coût de la projection est alors dérivée de la probabilité de génération des séquences de u suivant ces chemins, et utilisée pour estimer la pertinence de l'ensemble de fusions potentielles. On définit alors la divergence de u' par rapport à u comme le coût de la projection de moindre coût, et on l'utilise pour évaluer la fusion de u et u'. De plus, le fait que u soit racine d'un arbre nous permet de proposer un algorithme polynômial pour la recherche de la projection optimale, et d'intégrer celui-ci dans un nouvel algorithme d'apprentissage. Le chapitre suivant présente (entre autres) les premières expérimentations réalisées avec cet algorithme, et compare ses performances à celles du critère de vraisemblance maximale et de l'algorithme d'entraînement de Viterbi.

Chapitre 6

Expériences

6.1 Introduction

Ce chapitre reporte quelques expérimentations réalisées afin de comparer les performances des algorithmes d'apprentissage par fusions d'états et de la procédure d'entraînement de Viterbi en l'absence de structure connue. L'objet est d'observer leurs capacités à modéliser les séquences manipulées sans connaissances *a priori* – ou très peu – des domaines d'où elles sont tirées. Nous avons pour cela sélectionné trois applications différentes. Dans toutes ces applications, les séquences étudiées sont des séquences construites sur un alphabet discret. La taille de cet alphabet varie entre douze caractères et plusieurs milliers. Les tâches à résoudre appartiennent aux trois catégories présentées au chapitre 1 : discrimination, segmentation et prédiction. Chacune des applications fait l'objet d'une partie débutant par une courte présentation des séquences manipulées, suivie d'un résumé des résultats obtenus. En outre, chaque partie renvoie à une page en annexe où est proposé un exemple des séquences étudiées. Les applications sont présentées par ordre croissant de taille des alphabets : 12 pour les chorales de Bach (section 6.2), une quarantaine pour les séquences de lettres issues de ce document (section 6.3) et plusieurs milliers pour les séquences textuelles tirées de FlyBase (section 6.4). Des trois applications proposées, les deux premières sont des applications académiques assez classiques en apprentissage. La dernière, au contraire, traite un problème réel d'extraction d'informations à partir de données textuelles.

6.2 Chorales de Bach [Blake et Merz, 1998]

Ce jeu de test est tiré de l'*UCI machine learning repository*[1] [Blake et Merz, 1998]. Il s'agit d'un ensemble de lignes mélodiques de 100 des 371 chorales connues composées par J.S. Bach. Les séquences sont des suites de notes décrites par leur pitch et leur durée, ainsi que par la clef de la mélodie (voir [Blake et Merz, 1998]). Nous avons supprimé les informations relatives au temps et à la durée des notes, et combiné le pitch et la clef pour obtenir un ensemble de 100 séquences construites sur un alphabet de 12 symboles

[1] `http://www.ics.uci.edu/~mlearn/MLRepository.html`

représentant le pitch par rapport à la tonique. Pour simplifier, nous pouvons considérer ces séquences comme des lignes mélodiques en gamme de do majeur sur un octave. Les 12 symboles qui les composent sont alors les notes `do`, `do#`, `ré`, `ré#`, `mi`, `fa`, `fa#`, `sol`, `sol#`, `la`, `la#`, et `si`. Un exemple des séquences ainsi obtenues est proposé en annexe A, page 147. La longueur moyenne de ces séquences est de 47 notes.

6.2.1 Problèmes

Deux types de tests portant sur des tâches différentes ont été effectués. Les premiers concernent une tâche de prédiction. L'objet est de prédire, étant donnée une séquence de notes $O = o_1 \dots o_{T-1}$, la note o_T qui va immédiatement suivre. Si H est un HMM qui modélise les séquences, la méthode consiste à trouver le symbole a_{ML} qui a la probabilité maximale de succéder à O :

$$a_{\mathrm{ML}} = \operatorname*{argmax}_{a \in \Sigma} P(Oa|H),$$

avec Oa le résultat de la concaténation de la note a à la suite de la séquence O. $P(Oa|H)$ désigne ici simplement la probabilité de générer Oa au départ de l'état *start*, mais sans arriver sur l'état *end*. Dans nos tests, la probabilité $P(Oa|H)$ sera approximée par la probabilité de Viterbi de générer Oa au départ de *start*.

La deuxième série de test a pour but d'évaluer les modèles sur des tâches de discrimination. Pour cela, un nouvel ensemble de 100 séquences a été construit artificiellement en renversant les séquences originales. Le but est alors de discriminer les séquences renversées des séquences originales. Cette tâche est considérée comme peu aisée du fait que Bach utilisait fréquemment le renversement mélodique comme technique de composition [Brand, 1999]. Deux HMMs, appris respectivement à partir d'un ensemble de séquences originales et d'un ensemble de séquences renversées, sont utilisés pour cela. Le processus de classification d'une séquence de test consiste à calculer la probabilité de génération de cette séquence par chacun des HMMs, et d'assigner alors à la séquence sa classe la plus probable (les deux classes ont la même probabilité *a priori*). Comme pour le problème de discrimination, la probabilité de génération d'une séquence sera approximée par sa probabilité de Viterbi.

6.2.2 Déroulement des tests

6.2.2.1 Paramétrage des algorithmes

L'ensemble des 100 chorales a été partagé une première fois en 90 séquences d'apprentissage et 10 séquences de test, afin de fixer les paramètres des trois algorithmes d'apprentissage. Ce paramétrage a été réalisé de la manière suivante :

Algorithme d'entraînement de Viterbi Pour cet algorithme, les paramètres à fixer sont la taille des HMMs (on utilise des modèles entièrement connectés) et les densités des distributions de probabilités d'émission attachées aux états du HMM initial. Trois tailles (100, 150 et 200 états), et trois types de densités ont été essayés pour chacune de ces tailles : densité faible (chaque état émet 10% des symboles de l'alphabet), densité moyenne (chaque état émet 50% des symboles de l'alphabet) et forte densité (chaque état émet 80% des symboles de l'alphabet). Le paramétrage optimal a été déterminé en calculant la vraisemblance des 10 séquences de test sur les 3 tailles $\times$ 3 densités $=$ 9 HMMs entraînés. Les meilleurs résultats ont été obtenus avec le HMM de 150 états et des densités de 10%.

Algorithme utilisant le critère de vraisemblance maximale Pour cet algorithme, seule la taille du HMM doit être fixée, et les trois même tailles (100, 150 et 200 états) ont été essayées. Le HMM initial a été construit comme l'arbre préfixe des séquences d'apprentissage. Sa taille avoisine les 4000 états. À l'issu de l'apprentissage, les HMMs obtenus ont été réentraînés à l'aide de l'algorithme d'entraînement de Viterbi, et la taille optimale a également été déterminée en calculant la vraisemblance des 10 séquences de test. Les meilleurs résultats ont été obtenus grâce au HMM de 100 états.

Algorithme utilisant le nouveau critère Comme pour le critère de vraisemblance maximale, le HMM initial utilisé est l'arbre préfixe des séquences d'apprentissage. Les constantes de Laplace l et l' utilisées pour lisser respectivement les distributions de probabilités d'émission et de transition ont été fixées à

$$l = \frac{2}{|\Sigma|} = \frac{2}{12} = \frac{1}{6},$$

et $l' = 2$. Trois valeurs de α (la constante utilisée pour pondérer les états des sous-HMMs en fonction de leur distance à l'état racine) ont été essayées : 0.3, 0.5 et 0.7. La valeur de k (la profondeur maximale des sous-HMMs utilisée pour accélérer le processus) a été fixée à partir de celle de α comme décrit au chapitre 5. Pour τ (la valeur de la divergence maximale pour laquelle on accepte une fusion), nous avons également procédé comme décrit dans ce chapitre, en fixant la valeur de τ_p. Trois valeurs de τ_p ont été essayées : $1/4 \cdot \tau_0$, $1/3 \cdot \tau_0$, et $1/2 \cdot \tau_0$, avec

$$\tau_0 = \log\left(\frac{1+l}{l}\right) + \log(1+l').$$

Les 3 (α) $\times$ 3 (τ_p) $=$ 9 HMMs appris ont également été réentraînés à l'aide de l'algorithme d'entraînement de Viterbi, et là encore, le paramétrage optimal a été fixé grâce à la vraisemblance des 10 séquences de test. Les meilleurs résultats ont été obtenus avec $\alpha = 0.3$ et $\tau_p = 1/3 \cdot \tau_0$.

6.2.2.2 Prédiction

Une fois le paramétrage optimal fixé pour les trois algorithmes, la procédure suivante a été répétée 10 fois :

1. Les 100 séquences ont été partagées une nouvelle fois en 90 séquences d'apprentissage et 10 séquences de test.
2. Les procédures d'apprentissage ont été exécutées en utilisant le paramétrage déterminé. La procédure d'entraînement de Viterbi a été exécutée 10 fois en modifiant les paramètres initiaux (mais en gardant le même nombre d'états, et la même densité pour les distributions de probabilités d'émission du HMM initial), et le HMM maximisant la vraisemblance des séquences d'apprentissage a été retenu. Pour les HMMs inférés par fusions d'états, les modèles obtenus après apprentissage ont été réentraînés à l'aide de l'algorithme de Viterbi pour parfaire leur paramétrage.
3. Chaque séquence de test a été tronquée 10 fois à une longueur aléatoire (en choisissant à chaque fois un point de coupure différent) et présentée aux modèles appris afin de prédire la note suivante.

Cette procédure a donc consisté en $10 \times 10 \times 10 = 1000$ prédictions. Les résultats sont reportés dans le tableau 6.1.

6.2.2.3 Discrimination

Le même paramétrage que celui fixé pour la tâche de prédiction a été utilisé pour chacun des algorithmes. Les tests ont ensuite été réalisés de la manière suivante. 90 séquences originales et 90 séquences renversées ont servi à l'apprentissage des modèles qui ont été utilisés à leur tour pour classer les 20 séquences restantes. Comme pour les expériences de prédiction, ces test ont été répétés 10 fois en changeant les séquences d'apprentissage et de test, et constituent donc un total de 200 discriminations.

6.2.3 Résultats

Les résultats sont résumés dans le tableau 6.1. La ligne Viterbi présente les résultats obtenus en utilisant l'algorithme d'entraînement de Viterbi. Les lignes Fusion 1 et Fusion 2 présentent les résultats obtenus par les algorithmes d'apprentissage par fusions d'états utilisant respectivement le critère de vraisemblance maximale et le nouveau critère. Remarquons que dans les colonnes Vrais$\rightarrow$ et Vrais$\leftarrow$ est reporté l'*opposé* du logarithme de la vraisemblance des séquences de test originales et renversées. Plus ces valeurs sont proches de 0, plus la vraisemblance est élevée.

Au vu des résultats, il apparaît que, bien que produisant des HMMs légèrement meilleurs en terme de vraisemblance des séquences de test, l'algorithme d'apprentissage par fusions d'états utilisant le nouveau critère ne se démarque pas de l'algorithme d'entraînement de Viterbi pour ce qui concerne les taux de réussite en prédiction et discrimination où il obtient des résultats très comparables. L'algorithme d'apprentissage par fusions d'états utilisant le critère de vraisemblance maximale obtient des résultats très similaires à ceux de l'algorithme d'entraînement de Viterbi en ce qui concerne la vraisemblance des séquences de test, mais ses résultats en prédiction et discrimination sont nettement moins bons (31.3% de réussite en prédiction contre 35.4% pour Viterbi, et 67.6% de réussite en

Algo	Vrais →		Vrais ←		% predic.		% discrim.		#E	#T	Temps	
	moy	σ	moy	σ	moy	σ	moy	σ			App.	+Vit.
Viterbi	1010	105	1010	114	35.4	7.7	72.7	9.3	152	802	25'	-
Fusion 1	998	102	997	106	31.3	5.9	67.6	1.0	100	634	75'	80"
Fusion 2	975	100	962	104	34.6	2.0	73.7	10.5	70	476	75"	19"

TAB. 6.1: Résultats obtenus sur les chorales de Bach suivant trois algorithmes d'apprentissage de HMMs : l'algorithme d'entraînement de Viterbi, et les algorithmes d'apprentissage par fusions d'états utilisant le critère de vraisemblance maximale (Fusion 1) et le nouveau critère (Fusion 2). Les colonnes Vrais → et Vrais ← reportent respectivement l'opposé du logarithme de la vraisemblance des séquences de test originales et renversées. Les colonnes suivantes présentent les taux de réussite obtenus en prédiction et en discrimination. Dans les colonnes #E et #T sont reportées les moyennes du nombre d'états et de transitions des différents modèles obtenus. Les colonnes App. et +Vit. reportent respectivement le temps de calcul des procédures d'apprentissage et le temps de calcul de la procédure d'entraînement utilisée pour parfaire le paramétrage des HMMs inférés par fusions d'états.

discrimination contre 72.7% pour Viterbi). Si l'on compare les courbes d'évolution de la vraisemblance des séquences d'apprentissage au cours des deux algorithmes par fusions d'états (voir figure 6.1), on s'aperçoit que la vraisemblance est plus élevée pour le critère de vraisemblance maximale pendant une grande partie de la procédure, puis la courbe d'évolution atteint un point d'inflexion et c'est alors le nouveau critère qui permet d'obtenir une vraisemblance maximale. À la fin de l'apprentissage, l'écart entre les logarithmes des vraisemblances est supérieur à 3000. Notons cependant qu'il s'agit là des vraisemblances suivant les chemins de Viterbi calculées – comme pendant l'apprentissage – en faisant l'hypothèse de non modification de ces chemins. Les vraisemblances réelles peuvent donc différer sensiblement de ces résultats.

En fait, les différences essentielles entre ces trois algorithmes apparaissent lorsque l'on compare la taille des HMMs obtenus et les temps d'apprentissage. Concernant la taille des HMMs, ceux inférés avec le nouveau critère sont en effet beaucoup plus petits (70 états en moyenne contre 150 pour l'entraînement de Viterbi). Quant au temps de calcul, il est pour sa part nettement en faveur du nouveau critère, puisque l'apprentissage est d'en moyenne 75 secondes, alors qu'il est de 25 minutes pour une seule exécution de la procédure d'entraînement de Viterbi (et cette procédure a été exécutée 10 fois pour chaque test), et de 75 minutes pour le critère de vraisemblance maximale. Notons que les procédures de réentraînement lancées sur les modèles obtenus par fusions d'états ont un temps de calcul nettement plus courts que celles qui s'exécutent sur des HMM initiaux complètement connectés (19 secondes en moyenne pour le nouveau critère et 80 secondes pour le critère de vraisemblance maximale). La raison est tout simplement que les modèles appris par fusions d'états ont un nombre de transitions réduit par rapport au nombre d'états.

Afin de comparer les algorithmes sur des HMMs de tailles similaires, nous avons relancé la procédure d'entraînement de Viterbi sur des modèles de 70 états en utilisant les mêmes

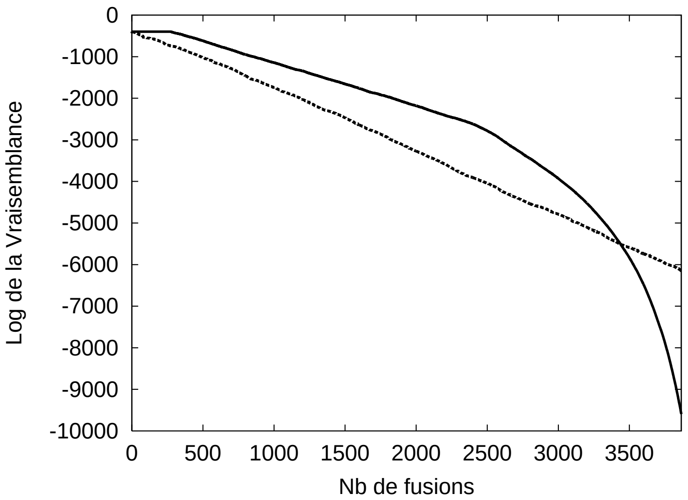


FIG. 6.1: Apprentissage sur les chorales de Bach. Évolution du logarithme de la vraisemblance des séquences d'apprentissage en fonction du nombre d'états fusionnés pour les deux critères de sélection : le critère de vraisemblance maximale (ligne courbe) et le nouveau critère (ligne droite).

Algo	Vrais $\rightarrow$		Vrais $\leftarrow$		% predic.		% discrim.		Temps	
	moy	σ	moy	σ	moy	σ	moy	σ		
Viterbi	1033	106	1063	118	32.2	2.9	67.6	11.8	5'	-
Fusion 1	984	102	998	104	32.3	6.0	70.2	9.1	75'	56"
Fusion 2	975	100	962	104	34.6	2.0	73.7	10.5	75"	19"

TAB. 6.2: Résultats obtenus sur les chorales de Bach avec des HMMs de 70 états en utilisant l'algorithme d'entraînement de Viterbi (ligne Viterbi) et l'algorithme d'apprentissage par fusions d'états utilisant le critère de vraisemblance maximale (Fusion 1). La dernière ligne (Fusion 2) reporte les résultats du tableau 6.1 obtenus avec le nouveau critère.

ensembles d'apprentissage et de test. Les résultats sont reportés dans le tableau 6.2, en même temps que ceux obtenus avec des HMM de 70 états inférés en utilisant le critère de vraisemblance maximale. Il apparaît qu'à tailles égales, les HMMs inférés à l'aide du nouveau critère sont meilleurs que ceux obtenus par les deux autres algorithmes, aussi bien en ce qui concerne la vraisemblance des séquences de test qu'en prédiction et discrimination. On remarquera également que la procédure de réentraînement appliquée après l'apprentissage est plus rapide pour le nouveau critère que pour le critère de vraisemblance maximale (19 secondes contre 56 secondes). La raison est que les HMMs inférés à l'aide du nouveau critère ont un nombre de transitions réduit par rapport à ceux obtenus avec le critère de vraisemblance maximale ($\sim$ 500 contre $\sim$ 900). La procédure de réentraînement a pour effet de réduire le nombre de transitions qui passe alors à une valeur proche de 450 pour les deux critères.

6.3 Séquences textuelles

Nous avons ensuite testé l'aptitude des HMMs à modéliser des séquences textuelles issues de ce document. Dans ce test, les séquences sont des phrases constituées de caractères, et l'objet est de faire de la prédiction de caractère. Pour cela, on propose aux modèles appris une phrase tronquée, et ils doivent prédire le caractère qui va suivre. Trois ensembles de séquences composées respectivement de 2500, 700, et 2000 caractères au total ont été constitués de la manière suivante. Les phrases ont été extraites du chapitre 9, et les accents ont été supprimés pour réduire la taille de l'alphabet manipulé, qui se compose alors de 40 caractères (34 lettres dont 10 majuscules et 6 symboles tels que l'espace, la virgule, l'apostrophe, le tiret et les parenthèses).

6.3.1 Déroulement des tests

L'ensemble de 2500 caractères a servi à l'apprentissage des HMMs suivant les trois algorithmes. Ces algorithmes ont été exécutés avec différents paramétrages, suivant une procédure strictement similaire à celle utilisée pour les chorales de Bach. Le paramétrage optimal de chacun des algorithmes a été déterminé en calculant la vraisemblance des séquences du deuxième ensemble (celui de 700 caractères) :

- Pour l'algorithme de Viterbi, les meilleurs résultats ont été obtenus à l'aide d'un HMM initial de 100 états, et avec une densité de probabilité de 10% sur les états. L'algorithme de Viterbi a alors été relancé 10 fois en utilisant ce paramétrage, et le modèle maximisant la vraisemblance des séquences d'apprentissage a été retenu.

- Pour l'algorithme utilisant le critère de vraisemblance maximale, le HMM initial a été construit comme l'arbre préfixe des séquences d'apprentissage. Sa taille est de 2440 états. Après apprentissage, les HMMs de taille 100, 150 et 200 états inférés ont été réentraînés à l'aide de la procédure de Viterbi pour parfaire leur paramétrage. Les meilleurs résultats ont été obtenus avec le HMM de 100 états.

- Pour l'algorithme utilisant le critère de vraisemblance maximale, le HMM initial a également été construit comme l'arbre préfixe des séquences d'apprentissage. L'algorithme a été lancé avec différents paramètres (les mêmes que ceux utilisés pour les chorales de Bach), et les HMM appris ont été réentraînés à l'aide de la procédure de Viterbi. Les meilleurs résultats ont été obtenus avec les paramètres $\alpha = 0.3$ et $\tau_p = 1/2 \cdot \tau_0$.

Après apprentissage, les HMMs ont été utilisés pour prédire les caractères des séquences du troisième ensemble (celui de 2000 caractères). 40 séquences composent cet ensemble. Chacune de ces séquences a été tronquée à une longueur variant entre 3 et 60 caractères. Les HMMs ont alors été utilisés pour prédire le caractère suivant. Ces tests ont été répétés 10 fois pour chaque séquence de test, en choisissant à chaque fois un point de coupure différent. Les résultats sont regroupés dans le tableau 6.3.

Algo	Vrais	% Prédiction	#E	#T	Temps
Viterbi	5363	24.7	89	423	3'
Fusion 1	5107	24.0	100	607	35'
Fusion 2	4864	27.2	89	470	25"

Tab. 6.3: Résultats obtenus sur des séquences textuelles issues de ce document suivant trois algorithmes d'apprentissage de HMMs : l'algorithme d'entraînement de Viterbi, et les algorithmes d'apprentissage par fusions d'états utilisant le critère de vraisemblance maximale (Fusion 1) et le nouveau critère (Fusion 2).

6.3.2 Résultats

Au vu des résultats, il apparaît que pour cette application, l'algorithme d'apprentissage par fusions d'états utilisant le nouveau critère de sélection se démarque des deux autres algorithmes, que ce soit en terme de vraisemblance des séquences de test que pour les résultats obtenus en prédiction (27.2% de réussite contre 24.7% et 24.0% respectivement pour la procédure d'entraînement de Viterbi et le critère de vraisemblance maximale). À titre indicatif, en prédisant à chaque essai le symbole le plus fréquent (le `e`) on obtient sur le même jeu de test un taux de réussite de 17.2%. Comme pour les chorales de Bach, on peut comparer les courbes d'évolution de la vraisemblance des séquences d'apprentissage au cours des deux algorithmes par fusions d'états (voir figure 6.2). Si l'on s'intéresse à la structure des modèles inférés, on observe que le rapport du nombre de transitions sur le nombre d'états est plus important pour le modèle construit par fusions d'états en utilisant le critère de vraisemblance maximale que ceux obtenus avec les deux autres approches. On peut observer également que le HMM entraîné à l'aide de l'algorithme de Viterbi a été élagué, puisqu'il se compose de 89 états, alors que le HMM initial en comptait 100.

6.4 FlyBase [The FlyBase Consortium, 1999]

FlyBase[2] [The FlyBase Consortium, 1999] est une base de données relatives au génome de *Drosophila melanogaster*. Les données de FlyBase constituent un vaste ensemble d'informations telles que :

- les informations génétiques, moléculaires, et phénotypiques de tous les gènes de *D. melanogaster* recensés, ainsi que quelques informations annexes portant sur d'autres espèces de drosophiles ;
- les aberrations chromosomiques ;
- une liste des clones moléculaires isolés ;
- les séquences ADN et protéiques des gènes ;
- une bibliographie des publications sur la drosophile ;

[2] `http://flybase.bio.indiana.edu/`

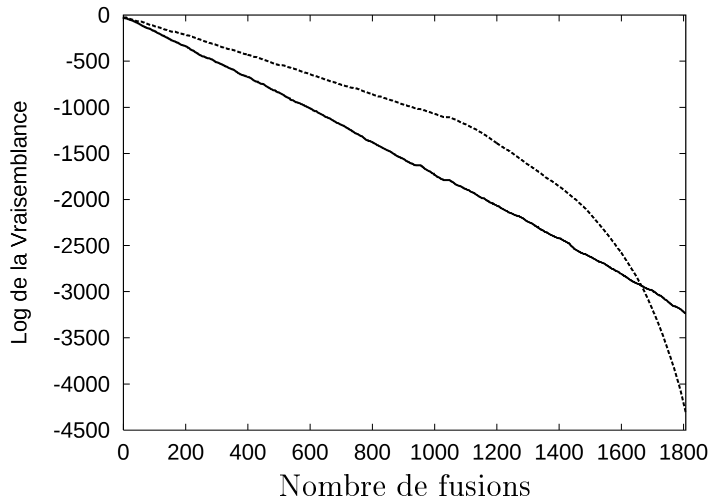


FIG. 6.2: Apprentissage sur des séquences textuelles issues de ce document. Évolution du logarithme de la vraisemblance des séquences d'apprentissage en fonction du nombre d'états fusionnés pour les deux critères de sélection : le critère de vraisemblance maximale (ligne courbe) et le nouveau critère (ligne droite).

Les informations relatives aux gènes sont structurées en différents champs parmi lesquels on trouve : le nom du gène, son symbole, sa position sur la carte génétique, un ensemble de références bibliographiques, ainsi qu'un champs libre constitué d'un court résumé concernant certaines des références bibliographiques.

6.4.1 Problème

Le problème que l'on se propose de traiter nous a été présenté par Bernard Jacq et Violaine Pillet de l'université Aix-Marseille III et a fait l'objet d'une partie de la thèse de Violaine Pillet [Pillet, 2000]. Globalement, il s'agit d'extraire de la base les informations relatives aux interactions entre gènes. Ces informations sont présentes dans FlyBase mais de manière non indexée. Elles apparaissent en fait exclusivement dans les résumés bibliographiques qui constituent le champs textuel libre de chaque enregistrement. Ce problème se compose de deux tâches distinctes :

1. isoler les résumés traitant effectivement d'une interaction ;
2. extraire de ces résumés le nom des gènes impliqués dans l'interaction.

Nous considérerons les résumés comme des séquences textuelles. Un résumé est donc une séquence, et les symboles qui constituent cette séquence sont les mots du résumé.

La première tâche est un problème de discrimination. On peut le traiter comme on l'a fait pour les chorales de Bach en apprenant deux modèles, un qui modélise les résumés positifs (traitant effectivement d'une interaction), et un qui modélise les résumés négatifs (les autres). Le processus de classification d'une nouvelle séquence consiste alors à calculer la probabilité de génération de cette séquence par chacun des modèles, et à lui assigner sa classe la plus probable, en tenant compte de la probabilité *a priori* de chacune des classes (voir chapitre 1). Cette tâche va nous permettre de comparer les résultats que l'on obtient en utilisant, d'un côté l'approche classique pour ce type de tâches (les unigrams), et de l'autre les HMMs.

La deuxième tâche est un problème de segmentation dont l'objet est d'extraire, à partir des résumés traitant effectivement d'une interaction, le nom des gènes impliqués. Cette tâche est relativement complexe pour plusieurs raisons. Tout d'abord, il existe plusieurs manières de désigner un gène : par son nom complet, par une abréviation ou par son symbole. Ensuite, les erreurs de typographie ne sont pas exclues. Enfin, de nombreux gènes tirent leur nom d'un mot du langage courant : `by`, `cell` et `for`, par exemple, sont des noms de gènes existants. Pour toutes ces raisons, ce problème ne peut pas être simplement résolu par utilisation d'un dictionnaire des gènes. La résolution de cette tâche à l'aide de HMMs passe par l'utilisation de modèles typés. En substance, comme on l'a décrit au chapitre 2, partie 2.3, la méthode consiste à apprendre un HMM dont certains des états ont le type GÈNE (c'est-à-dire qu'ils ne peuvent générer que des noms de gènes). On extrait ensuite le nom des gènes d'une nouvelle séquence en calculant son chemin de Viterbi dans le HMM et en isolant les mots qui sont générés sur les états de type GÈNE.

6.4.2 Préparation des données

Le corpus original des résumés n'a pas été utilisé tel quel mais a subi plusieurs traitements destinés à extraire les phrases présentant un intérêt réel pour le problème. Toutes les phrases dans lesquelles n'apparaît pas le nom du gène de l'enregistrement d'où est tiré le résumé, par exemple, ont été supprimées. À l'issue de ces prétraitements, un corpus de 1143 phrases était formé, chacune d'elles constituant un résumé. Parmi ces phrases, 653 traitent effectivement d'une interaction, tandis que les 490 restantes concernent tout autre chose. La taille moyenne d'une séquence est de 19 mots. La tâche de classification ne pose pas de problème spécifique, et peut être traitée sans autre préparation des données. Il n'en va pas de même pour la tâche de segmentation, pour laquelle il est nécessaire d'annoter manuellement le corpus de manière à marquer le nom des gènes apparaissant dans les phrases. L'ensemble de ces traitements est décrit précisément dans [Pillet, 2000]. À ces prétraitements, qui étaient déjà réalisés lorsque nous avons pris possession du corpus, nous avons ajouté un marquage automatique des signes de ponctuation présents (virgules, point virgule et deux points), et nous avons supprimé le *s* de la troisième personne et du pluriel de manière à réduire le vocabulaire manipulé. La taille de l'alphabet à l'issue de cette procédure avoisine les 3000 mots. Un exemple des séquences manipulées est proposé en annexe B, page 149.

6.4.3 Déroulement des tests

6.4.3.1 Apprentissage

Afin de pouvoir traiter le problème de segmentation, nous avons utilisé des HMMs typés. Le type des états a été réduit aux trois types GÈNE, PONCTUATION, et AUTRE, qui correspondent aux marquages effectués sur le corpus. Les états de type Gène ne peuvent émettre que des noms de gènes, les états de type Ponctuation n'émettent que les caractères de ponctuation, tandis que les autres mots qui composent la majeure partie du corpus sont générés par les états typés Autre. Les HMMs ont une taille de 100 états et sont inférés en utilisant d'un côté l'algorithme d'apprentissage par fusions d'états utilisant le critère de vraisemblance maximale, et de l'autre côté l'algorithme d'entraînement de Viterbi :

- L'apprentissage par fusions d'états se déroule de manière classique. Le HMM initial est le HMM construit comme l'arbre préfixe des séquences d'apprentissage. Ses états sont typés en fonction du marquage des mots qu'ils génèrent. Ensuite, lors de la phase d'apprentissage, seules les fusions d'états de même type sont autorisées, et l'état résultant de la fusion conserve le même type. La procédure est stoppée lorsque le nombre d'états du HMM est égal à 100.
- Pour la procédure d'entraînement de Viterbi, un HMM de 100 états entièrement connecté, et avec une densité de probabilité d'émission faible (des test préalables ont servi à déterminer le paramétrage optimal) est utilisé comme HMM initial. Pour cet algorithme, le problème du typage des états s'est posé. Il faut en effet fixer dans le HMM initial (avant entraînement) le type de chacun des états. Ce problème ne se pose pas dans le cadre des algorithmes par fusions d'états où les types sont naturellement déduits du processus d'apprentissage. Nous avons donc décidé de typer les états du modèle initial de manière à ce que le nombre d'états de chaque type soit égal au nombre moyen observé après apprentissage dans les HMM inférés par fusions d'états, à savoir 8 états de type Gène, 2 états de type Ponctuation et 90 états de type Autre. L'algorithme d'entraînement de Viterbi est ensuite lancé de manière classique, à ceci près que, dans la phase d'*expectation*, les chemins de Viterbi des séquences sont calculés de manière à respecter le typage des mots qui les composent.

Comme mentionné, nous avons également utilisé les unigrams afin de discriminer les résumés traitant d'une interaction ; l'objectif étant de comparer les HMMs à la méthode couramment utilisée pour ce genre de tâche. Les unigrams ont été entraînés suivant la méthode classique décrite au chapitre 1.

6.4.3.2 Discrimination

Les expérimentations ont été effectuées de la manière suivante. Les 1143 phrases du corpus ont été réparties en 10 ensembles de même taille. Ces ensembles ont servi tour à tour d'ensemble de test, tandis que les 9 ensembles restant étaient, après union, redivisés en deux ensembles suivant les classes des séquences pour constituer 2 ensembles d'apprentissage. Chacun de ces 2 ensembles a servi à l'apprentissage des HMMs et des unigrams suivant les procédures décrites ci-dessus. Notons que pour l'algorithme d'apprentissage par fusions d'états, la taille du HMM initial avoisine les 9000 états. Quant à l'algorithme d'entraînement de Viterbi, il a été exécuté 10 fois pour chacun des tests, et le HMM maximisant la vraisemblance des séquences d'apprentissage a été retenu.

Modèle/Algo	Vrais		%Discrim		#E	#T	Temps
	moy	σ	moy	σ			
Unigram	6930	562	79.3	3.0	-	-	$<$ 1”
HMM-Viterbi	6632	400	77.2	3.0	94	382	30’
HMM-Fusion	6083	404	80.2	1.8	100	2000	280’

Tab. 6.4: Résultats obtenus en discrimination sur les séquences textuelles issues de FlyBase en utilisant trois types d'approche : des unigrams, des HMMs entraînés à l'aide de l'algorithme d'entraînement de Viterbi, et des HMMs inférés à l'aide de l'algorithme d'apprentissage par fusions d'états utilisant le critère de vraisemblance maximale.

Les résultats obtenus sont résumés dans le tableau 6.4. Dans les premières colonnes sont reportés la moyenne et l'écart type observés de l'opposé du logarithme de la vraisemblance des séquences de test positives (qui traitent réellement d'une interaction). Les colonnes suivantes rapportent les taux de réussite obtenus en discrimination pour chacune des approches, tandis que dans les dernières colonnes sont reportés les nombres d'états et de transitions des HMMs, ainsi que le temps de calcul des différentes procédures d'apprentissage.

Au vu de ces résultats, il apparaît que l'utilisation des HMMs pour la discrimination de données textuelles n'est pas d'un réel intérêt au regard des performances des unigrams : les taux de réussite en discrimination lorsque l'on utilise l'algorithme d'entraînement de Viterbi sont en effet inférieurs à ceux que l'on obtient avec les unigrams, tandis que ceux obtenus grâce à l'algorithme d'apprentissage par fusions d'états sont à peine meilleurs. En outre, les temps de calcul sont nettement en faveur des unigrams, puisque le temps d'apprentissage et de classification à l'aide de ce modèle nécessite moins de 1 seconde par test, alors qu'une seule exécution de l'algorithme d'entraînement de Viterbi requiert 30 minutes, et que le temps de calcul de l'algorithme d'apprentissage par fusions d'états est de 280 minutes. Remarquons que les HMMs inférés par fusions d'états ont un nombre de transitions extrèmement élevé, comparé aux HMMs que l'on obtient par la procédure d'entraînement. En fait, ce fait s'observe également pour les deux applications précédentes (chorales de Bach et séquences textuelles), mais le nombre de transitions a été réduit par la procédure de réentraînement appliquée après l'apprentissage. La même procédure devrait avoir un effet similaire pour cette application.

6.4.3.3 Segmentation

En ce qui concerne la tâche de segmentation, deux types d'expériences ont été menées. Les mêmes ensembles de test et d'apprentissage que pour la tâche de discrimination ont été utilisés, mais en les réduisant aux seules séquences positives. Dans un premier temps, les séquences de test ont été utilisées sans aucune modification, et les HMMs appris ont servi à identifier dans ces séquences les mots désignant un nom de gène. La procédure d'extraction des noms de gène d'une séquence consiste, comme on l'a vu, à calculer le chemin de Viterbi de la séquence et à isoler les mots générés sur les états typés GÈNE. Les résultats sont évalués en terme de *précision* et *rappel*. La précision représente le nombre de gènes correctement identifiés par le modèle, divisé par le nombre total de gènes (corrects ou non) qu'il a proposé :

$$\text{précision} = \frac{\#\text{ de mots extraits qui sont effectivement des gènes}}{\#\text{ total de mots extraits}}.$$

Le rappel est le nombre de gènes correctement identifiés par le modèle, divisé par le nombre total de gènes présents dans les séquences :

$$\text{rappel} = \frac{\#\text{ de mots extraits qui sont effectivement des gènes}}{\#\text{ total de gènes présents dans les séquences}}.$$

Une manière pratique d'évaluer les résultats est d'utiliser une seule quantité, notée $F1$ [Freitag et McCallum, 1999], définie comme la moyenne harmonique de la précision et du rappel :

$$F1 = 2/(1/\text{précision} + 1/\text{rappel}).$$

Afin de se faire une meilleure idée des capacités d'extraction d'information des HMMs lorsqu'ils sont en présence de noms de gènes inconnus (c'est-à-dire des gènes qui n'apparaissent pas dans les séquences d'apprentissage), une deuxième série de tests a été effectuée. Tous les noms de gènes présents dans les séquences de test ont été remplacés par un mot inconnu des HMMs (`XXX` en l'occurence) et les expériences d'extraction ont été relancées. Les résultats obtenus lors de ces deux séries de tests sont reportés dans le tableau 6.5. Les premières colonnes rapportent les scores obtenus en précision, rappel et $F1$ sur les séquences de test non modifiées, tandis que dans les colonnes suivantes sont reportés les scores obtenus sur les séquences de test dans lesquelles les noms de gènes ont été remplacés par un terme inconnu. La première ligne rapporte les résultats obtenus grâce aux HMMs entraînés à l'aide de l'algorithme d'entraînement de Viterbi, tandis que sur les deux dernières lignes sont reportés les résultats des HMMs inférés à l'aide de l'algorithme d'apprentissage par fusions d'états. La ligne Fusion-100 rapporte ceux des HMMs à 100 états déjà utilisés en discrimination et Fusion-130 rapporte les résultats d'une autre série de modèles à 130 états.

Les résultats obtenus montrent une nette supériorité des HMMs inférés à l'aide de l'algorithme d'apprentissage par fusions d'états. Exprimée en terme de $F1$, pour les séquences originales et avec des HMMs à 100 états, la qualité des HMMs inférés par fusions d'états est de 0.83, contre 0.78 pour les HMMs entraînés par l'algorithme d'entraînement de Viterbi. Pour les séquences dans lesquelles les noms de gènes ont été remplacés par un mot inconnu, l'écart est encore plus important (0.64 contre 0.45). Remarquons qu'en utilisant des HMMs de taille légèrement plus grande (130 états), les performances diminuent pour les séquences originales mais augmentent pour les séquences modifiées.

Algo	Séq. originales			Séq. modifiées		
	précision	rappel	$F1$	précision	rappel	$F1$
Viterbi	0.79	0.76	0.78	0.60	0.36	0.45
Fusion-100	0.83	0.82	0.83	0.74	0.57	0.64
Fusion-130	0.80	0.81	0.80	0.74	0.60	0.67

TAB. 6.5: Extraction des noms de gènes à partir des séquences originales et des séquences modifiées (en remplaçant le nom des gènes par `XXX`) en utilisant 3 types de HMMs : des HMMs de 100 états entraînés à l'aide de l'algorithme d'entraînement de Viterbi (ligne Viterbi) et des HMMs de 100 et 130 états (lignes Fusion-100 et Fusion-130) inférés à l'aide de l'algorithme d'apprentissage par fusions d'états utilisant le critère de vraisemblance maximale.

6.4.4 Le nouveau critère

Nous avons également testé le nouveau critère de sélection sur ce jeu de données. Quels que soient les paramètres utilisés dans la procédure d'apprentissage (valeur de α et de τ), les résultats (non reportés) se sont révélés extrêmement décevants. Les HMMs inférés étaient en effet très gros (plus de 300 états), et très mauvais en terme de vraisemblance (aussi bien pour les séquences de test que pour les séquences d'apprentissage) comparés aux résultats obtenus par les deux autres approches. La raison est sans doute l'absence ou le faible nombre de motifs (portions de séquences) récurrents dans les séquences d'apprentissage – absence due, entre autres, à la grande taille de l'alphabet. Or ce sont justement ces motifs que notre critère essaye d'isoler pour sélectionner les états à fusionner. En l'absence de tels motifs, seules des informations très locales, comme celles utilisées par le critère de vraisemblance maximale, peuvent guider les fusions : les distributions d'émission et surtout de transition associées aux états. C'est donc, pour ce jeu de test, la quasi absence d'informations en dehors des informations d'ordre topologique (états ayant des états adjacents communs) qui explique les bonnes performances du critère de vraisemblance maximale comparées à celles du nouveau critère. Ces résultats sont à rapprocher de ceux obtenus dans le cadre de l'inférence d'automates d'états finis probabilistes (PDFAs) par Thollard et Dupont [Thollard et Dupont, 2000]. Dans cet article, les auteurs proposent un algorithme pour l'inférence de PDFAs qui peut être vu comme une adaptation de l'algorithme par fusions d'états utilisant le critère de vraisemblance maximale. Ils appliquent alors leur algorithme à la modélisation de séquences issues du système d'informations de voyage aérien ATIS [Hirschman et al., 1992]. La taille de l'alphabet de ce corpus est de l'ordre de 1300 mots. Parallèlement à cela, ils testent sur les mêmes données l'algorithme d'inférence ALERGIA qui utilise, comme nous l'avons vu, un principe proche de celui de l'algorithme par fusions d'états utilisant le nouveau critère, et leurs résultats font apparaître pour ce problème une supériorité assez nette de leur algorithme.

6.5 Conclusion

Nous avons testé les trois algorithmes d'apprentissage de HMMs exposés aux chapitres précédents sur trois domaines d'applications. Nous avons pu observer que l'algorithme d'apprentissage par fusions d'états utilisant le nouveau critère de sélection permet souvent l'apprentissage de modèles de bonne qualité et en un temps de calcul nettement inférieur aux deux autres approches. Au vu de l'ensemble des résultats obtenus, il apparaît néanmoins que les performances des trois algorithmes sont étroitement liées aux applications, et qu'aucun d'eux ne peut prétendre à une supériorité globale : l'algorithme d'entraînement de Viterbi et le nouveau critère obtiennent des résultats comparables sur les chorales de Bach, le nouveau critère est légèrement meilleur que les autres approches sur les séquences textuelles issues de ce document, et l'algorithme par fusions d'états utilisant le critère de vraisemblance maximale se démarque nettement des deux autres en ce qui concerne les séquences textuelles issues de FlyBase. Il reste cependant difficile de tirer de ces expériences une règle d'utilisation *a priori* de tel ou tel algorithme. Même si la taille de l'alphabet semble être un indicateur intéressant, elle ne peut suffire à caractériser les séquences à modéliser et donc à présumer des performances des différents algorithmes. Par ailleurs, pour un domaine d'application fixé, les performances des modèles inférés sont susceptibles de varier suivant le type de tâche à résoudre (reconnaissance, segmentation ou prédiction). Il n'en reste pas moins vrai que, comme le prouve l'ensemble de ces expériences, les algorithmes d'apprentissage par fusions d'états (quel que soit le critère de sélection utilisé) constituent, dans le cadre de l'apprentissage de HMMs sans structure connue, une alternative intéressante aux techniques d'entraînement classiques.

Deuxième partie

Utilisations pour le test de circuits intégrés

Chapitre 7

Introduction au test de circuits intégrés

Ce chapitre présente quelques notions relatives au (vaste) domaine du test de circuit microélectronique. Il s'adresse avant tout aux non spécialistes du domaine (les spécialistes du test le trouveront sans doute un peu naïf), et son objectif est d'introduire aux problèmes qui seront abordés dans les deux chapitres suivants. Il débute (partie 7.1) par une courte introduction générale au test, et plus particulièrement aux différentes solutions utilisées pour modéliser les circuits microélectroniques, ainsi que les dysfonctionnements qui peuvent les affecter. Nous présentons ensuite (partie 7.2) une des méthodes couramment utilisée pour tester un circuit, c'est-à-dire détecter un dysfonctionnement potentiel, et à la partie 7.3, nous verrons que cette méthode peut être mise en place suivant deux processus qui posent l'un et l'autre un problème particulier. Ce sont ces deux problèmes auxquels nous nous intéresserons dans les chapitres suivants.

7.1 Présentation du domaine

Grâce aux progrès constants réalisés en physique des composants, les circuits intégrés deviennent de plus en plus complexes. De nos jours, la plupart des circuits sont composés de plusieurs millions de composants. Cette augmentation de la complexité s'est tout naturellement accompagnée d'un accroissement des possibilités de pannes et d'imperfections qui peuvent affecter un circuit et nuire à son bon fonctionnement. Ces imperfections peuvent apparaître à différents stades de sa vie. Elles peuvent résulter de la phase de production (fabrication) du circuit mais elles peuvent également se déclarer bien plus tard, lors de l'utilisation du circuit.

Le test d'un circuit microélectronique consiste à mettre en évidence un éventuel dysfonctionnement dû à une défaillance physique. Lorsque l'on teste un circuit, on suppose que sa conception – c'est-à-dire son modèle théorique – est bonne, mais qu'un événement extérieur a physiquement altéré l'une de ses composantes. La mise en évidence d'un dysfonctionnement (ou *défaut*) peut être réalisée de deux manières distinctes. La première consiste à effectuer des mesures paramétriques (courant, tension, ...) sur le circuit ; on parle alors de *test paramétrique*. La deuxième consiste à inspecter les données logiques produites par le circuit et à les comparer avec les valeurs logiquement attendues. Cette technique de test, appelée *test logique*, est la technique à laquelle nous allons nous inter-

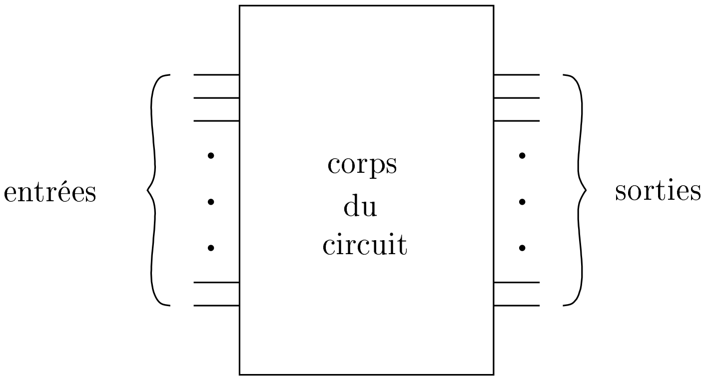


FIG. 7.1: Structure générale d'un circuit microélectronique.

resser ici. Elle nécessite, pour être mise en place, de pouvoir modéliser le comportement du circuit (qu'il soit sain ou défectueux), c'est-à-dire de trouver une modélisation : 1) du circuit lui-même ; 2) des défauts qui peuvent affecter ce circuit. Les deux parties suivantes traitent respectivement de l'un et l'autre de ces problèmes.

7.1.1 Modélisation d'un circuit

La modélisation d'un circuit peut être effectuée à différents niveaux d'abstraction ; plus le niveau est élevé, plus la manipulation du modèle est aisée mais moins celui-ci est précis. Cette modélisation peut être réalisée au niveau physique (niveau le plus proche de l'implémentation), au niveau transistor, au niveau porte (une porte se compose de plusieurs transistors) ou au niveau comportemental (niveau le plus haut). Nous nous intéresserons ici à une modélisation au niveau porte qui a l'avantage de caractériser assez précisément le circuit, tout en restant relativement souple à manipuler.

Un circuit microélectronique décrit au niveau porte se compose de trois parties distinctes comme représentées sur la figure 7.1 : les *entrées*, les *sorties*, et le *corps* du circuit composé de *portes logiques* et de *cellules mémoires* interconnectées. Les portes logiques sont les opérateurs de logique booléenne AND, OR, NOT, NAND, NOR et XOR (voir figure 7.2). Les cellules mémoires sont des composants ayant la capacité de mémoriser une valeur booléenne (`0` ou `1`). Les circuits microélectroniques dépourvus de cellules mémoires sont appelés *circuits combinatoires*. Les circuits comportant des cellules mémoires sont les *circuits séquentiels*. C'est à ce dernier type de circuit que nous allons nous intéresser.

L'utilisation d'un circuit microélectronique consiste à appliquer certaines valeurs booléennes sur les entrées du circuit, et à lire sur les sorties la réponse du circuit à ces valeurs. Pour les circuits combinatoires, les valeurs de sortie ne dépendent que des valeurs d'entrée. Pour les circuits séquentiels, ces valeurs dépendent également des valeurs booléennes contenues dans les cellules mémoires au moment où sont appliquées les valeurs d'entrée. Cela signifie que la réponse d'un circuit à certaines valeurs d'entrée ne sera pas forcément la même suivant le moment où l'on applique ces valeurs. Considérons par exemple le circuit représenté à la figure 7.3. En plus d'un ensemble de portes logiques, ce circuit comporte

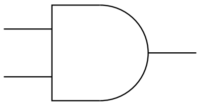
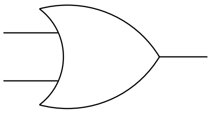
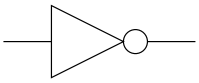
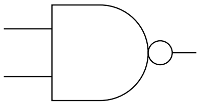
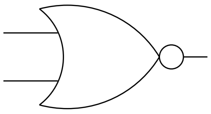
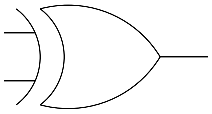
Opérateur	Symbole	Table de vérité
AND		$\begin{array}{c\|cc} & 0 & 1 \\ \hline 0 & 0 & 0 \\ 1 & 0 & 1 \end{array}$
OR		$\begin{array}{c\|cc} & 0 & 1 \\ \hline 0 & 0 & 1 \\ 1 & 1 & 1 \end{array}$
NOT	 OU 	$\begin{array}{c\|c} 0 & 1 \\ 1 & 0 \end{array}$
NAND		$\begin{array}{c\|cc} & 0 & 1 \\ \hline 0 & 1 & 1 \\ 1 & 1 & 0 \end{array}$
NOR		$\begin{array}{c\|cc} & 0 & 1 \\ \hline 0 & 1 & 0 \\ 1 & 0 & 0 \end{array}$
XOR		$\begin{array}{c\|cc} & 0 & 1 \\ \hline 0 & 0 & 1 \\ 1 & 1 & 0 \end{array}$

FIG. 7.2: Représentation des portes

une cellule mémoire notée m. Si on applique simultanément 1 et 0, respectivement sur les entrées i_1 et i_2, la valeur de sortie sur la ligne o_1 sera 1. Il n'y a pas de cellule mémoire entre i_1, i_2 et o_1 et donc le résultat ne dépend que des valeurs appliquées en i_1 et i_2. Appliquons maintenant 1 et 0 sur les entrées i_4 et i_5 et essayons de trouver les valeurs de sortie de o_3 et o_4. La valeur de sortie d'une cellule est la valeur mémorisée par cette cellule. La valeur de o_3 dépend donc de la valeur mémorisée par m : si la cellule mémoire contient 1, o_3 aura la valeur 1, et si la cellule mémoire contient 0, o_3 aura la valeur 0. Pour o_4, par contre, du fait de la porte AND, on peut d'ores et déjà conclure que, quelle que soit la valeur de la cellule mémoire, cette sortie aura la valeur 0.

Intéressons nous maintenant à la valeur mémorisée par une cellule mémoire. Comme on peut le voir sur la figure 7.3, une cellule mémoire comporte deux entrées et une sortie. Les entrées sont de deux types : une entrée classique – du même type que les entrées d'une porte logique – et une entrée pour l'*horloge* (notée clock). L'horloge est un composant essentiel des circuits microélectroniques dont nous n'avons pas encore parlé. Elle sert à cadencer le circuit et permet (entre autres) de changer les valeurs mémorisées par les cellules mémoires. En fait, la valeur contenue par une cellule mémoire est la valeur qui a été appliquée en entrée de cette cellule au coup d'horloge précédent. Dans notre exemple, puisque l'on vient

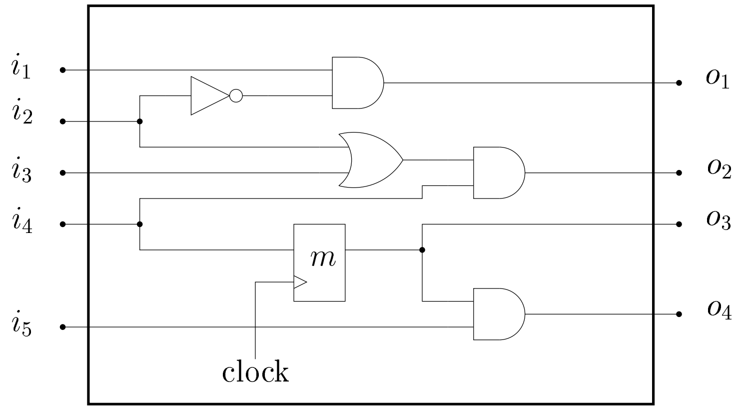


FIG. 7.3:

d'appliquer 1 sur l'entrée i_4, au coup d'horloge suivant la valeur de m sera donc 1. Mais quelle valeur contient une cellule mémoire au premier coup d'horloge, lorsque l'on vient de mettre le circuit sous tension ? Impossible de le savoir. C'est pourquoi l'utilisation d'un circuit séquentiel est toujours précédée d'une phase d'initialisation des cellules mémoires. Cette phase d'initialisation peut être effectuée de deux manières différentes, suivant les spécificités des cellules. La première consiste à appliquer des valeurs appropriées aux entrées d'un circuit. La deuxième technique, qui ne peut être utilisée que pour certains types de cellules, est d'utiliser une ligne d'entrée spéciale sur les cellules mémoires qui permet, indépendamment de la valeur appliquée sur l'entrée classique, de les initialiser aux valeurs 0 (si la cellule dispose d'une ligne *reset*) ou 1 (si elle dispose d'une ligne *set*). Néanmoins, toutes les cellules mémoires ne disposent pas de ces lignes spéciales, et on est alors contraint de les initialiser de manière classique, en utilisant exclusivement les lignes d'entrées du circuit. La phase d'initialisation peut alors être longue et nécessiter plusieurs coups d'horloge. En effet, les cellules pouvant être interconnectées, un coup d'horloge n'est pas toujours suffisant.

7.1.2 Modélisation des dysfonctionnements

Nous avons décrit dans la section précédente la modélisation d'un circuit au niveau porte. Cette modélisation nous a permis de représenter le fonctionnement du circuit et de déduire, à partir des valeurs appliquées aux entrées, les valeurs logiquement attendues en sortie. On va maintenant s'intéresser à la modélisation d'un dysfonctionnement du circuit. Là encore, plusieurs modèles sont possibles. On pourra distinguer entre autres les trois modèles suivants.

Le modèle de collage simple (*stuck-at fault*) : C'est le premier modèle ayant été utilisé. Il consiste à considérer que les défauts qui peuvent affecter un circuit ont pour effet de maintenir (ou *coller*) une ligne du circuit à une valeur fixe. En d'autres termes, on considère, qu'une ligne collée est une ligne qui a toujours la même valeur booléenne, quelle que soit la valeur qui lui est appliquée en entrée. On dira alors qu'une ligne est *collée à* `0`, ou collée à `1`. L'autre hypothèse importante du modèle de collage simple est que l'on considère qu'une seule faute à la fois peut se produire. Un circuit défectueux est donc un circuit dont une des lignes (et non plusieurs) est collée à une certaine valeur. Sous ce modèle, chaque ligne peut être affectée par deux fautes de collage, et donc le nombre total de défauts potentiels d'un circuit est égal à deux fois le nombre de lignes du circuit.

Le modèle de collage multiple : C'est une extension du modèle de collage simple pour lequel plusieurs lignes (et non une seule) peuvent être affectées par un collage.

Le modèle de délai : On peut également le voir comme une extension du modèle de collage simple. Dans le modèle de délai, on considère en effet qu'une ligne est collée à une certaine valeur mais pendant un temps fini – et non plus indéfiniment comme dans le modèle de collage simple. Plus précisément, une faute de délai est une ligne pour laquelle la valeur logiquement attendue ne se positionne pas en temps voulu mais avec un léger retard.

Le modèle que nous allons utiliser est le modèle de collage simple. Il dispose en effet de plusieurs avantages, parmi lesquels :

- il se manipule aisément au niveau logique ;
- il couvre un grand nombre de défaillances susceptibles d'affecter un circuit (et entre autres la majorité des défaillances affectant les connexions entre le circuit et le boîtier) ;
- il est utilisé par une grande majorité d'outils dédiés au test ;
- il permet, par effet de bord, de détecter de nombreux défauts physiques pourtant non modélisable par un simple collage ;
- il est couramment utilisé entre clients et fournisseurs pour juger de la qualité d'un test.

7.2 Le test logique

Nous avons présenté à la section précédente les modèles de circuits (modèle niveau porte) et de défauts (modèle de collage simple) que nous allons utiliser pour tester de manière logique un circuit. Tester une faute potentielle, c'est s'assurer que cette faute n'est pas présente. Tester un circuit, c'est tester le plus grand nombre de ses fautes potentielles. Le *taux de couverture de fautes* d'un test est le rapport du nombre de fautes testées sur le nombre total de fautes potentielles du circuit. Étant donnés les modèles choisis, le test d'un circuit va donc consister à s'assurer qu'aucune des fautes de collage potentielles du circuit – c'est-à-dire aucune des $2 \times$ (*nombre de lignes*) fautes de collage possibles – ne se produit. Nous allons voir maintenant comment tester logiquement une faute de collage donnée.

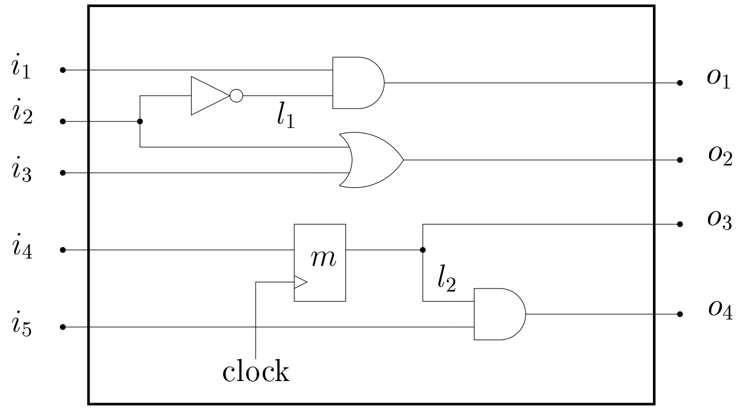


FIG. 7.4:

7.2.1 Tester une faute de collage

Lorsque l'on fait du test logique, on peut appliquer des valeurs aux entrées du circuit, lire les réponses du circuit à ces valeurs, mais on ne peut accéder, ni en lecture, ni en écriture, aux éléments internes du circuit. Le principe du test logique est alors d'appliquer certaines valeurs sur les entrées du circuit et de vérifier que les valeurs de sorties sont bien celles logiquement attendues. Cela n'est évidemment possible que parce que l'on connaît la structure logique du circuit et que l'on peut, comme on l'a fait sur l'exemple de la partie précédente, en déduire les valeurs de sortie logiquement attendues.

Considérons le circuit représenté à la figure 7.4 et étudions comment s'assurer que la ligne de sortie o_1 n'est pas collée à `0`. Il suffit pour cela d'appliquer aux entrées du circuit des valeurs telles que cette ligne soit logiquement positionnée à `1` et d'observer si c'est effectivement le cas. Pour positionner cette ligne à `1`, il suffit d'appliquer sur les entrées i_1 et i_2 respectivement les valeurs `1` et `0`. Si la valeur de o_1 est alors `1` on pourra conclure que cette ligne n'est pas collée à `0`. Remarquons que si ce n'est pas le cas, on ne peut pas pour autant en déduire qu'elle est collée à `0` car d'autres collages peuvent avoir le même effet : cela peut également être, par exemple, le résultat du collage à `0` de la ligne d'entrée i_1. Cette remarque est importante, elle signifie :

1. que l'objectif du test logique est simplement de tester si, oui ou non, le circuit est défectueux ; il n'est pas d'identifier précisément la faute responsable ;
2. qu'un même test peut permettre de tester différentes fautes potentielles.

L'exemple de test que nous venons de donner était relativement simple. La raison est que la ligne testée est une ligne de sortie et qu'elle est donc directement observable. Il nous a en fait suffi pour tester la faute de positionner cette ligne à la valeur inverse du collage. Pour les lignes internes au circuit, la procédure à suivre est un peu plus compliquée. Il faut : 1) positionner la ligne à la valeur inverse du collage testé ; 2) propager les effets du collage sur une ligne de sortie pour le rendre observable. Testons par exemple le collage à `1` de la ligne l_1. Pour positionner cette ligne à la valeur inverse (valeur `0`), il suffit d'appliquer `1` sur l'entrée i_2. Maintenant si on veut que la faute de collage soit observable, il faut que ses effets soient propagés sur la ligne de sortie o_1, c'est-à-dire que la valeur de o_1 ne soit pas celle logiquement attendue. On peut faire cela en appliquant dans le même temps la

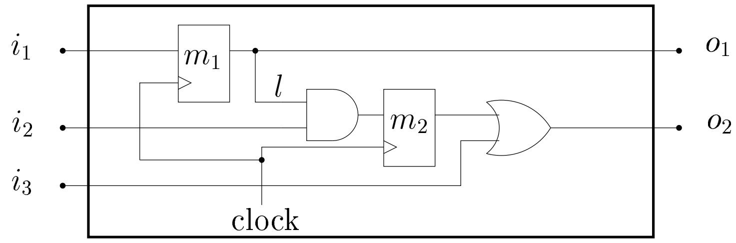


FIG. 7.5:

valeur `1` sur l'entrée i_1. En effet la valeur logiquement attendue sur o_1 est alors `0` (puisque la valeur logique de l_1 est `0`), alors que si l_1 est collée à `1`, on observera `1`. Remarquons par contre que si on avait appliqué `0` sur i_1 cela ne nous aurait pas permis de détecter le collage. L'ensemble des valeurs que l'on applique ainsi simultanément sur les entrées d'un circuit définissent ce que l'on appelle un *vecteur de test*. Par exemple, suivant ce que l'on vient de dire, `01000` est un vecteur de test permettant de détecter le collage à `1` de la ligne l_1 – le premier bit représente la valeur appliquée sur la ligne i_1, le deuxième la ligne i_2, et ainsi de suite. Ce n'est bien sûr pas le seul vecteur de test pour cette faute, puisqu'il suffit, comme on l'a vu, que le vecteur commence par `01`. Cela nous amène à une autre propriété importante du test qui est qu'une même faute peut être testée par plusieurs vecteurs de test différents. Par exemple, pour tester le collage à `0` de la ligne o_2, tous les vecteurs du type `*01**`, `*10**` ou `*11**` peuvent être utilisés. Le caractère `*` représente un bit de valeur non fixée (elle peut être indifféremment `0` ou `1`) et `*01**` est ce que l'on appellera dans la suite de ce document un *pattern de test* ou plus simplement un *pattern*, c'est-à-dire un ensemble de vecteurs booléens ayant certains bits communs.

Les fautes que nous venons de tester sont des fautes que l'on peut appeler *fautes de type combinatoire*, au sens où elle ne mettent pas en jeu de cellules mémoires. Elles peuvent être testées de manière relativement aisée, en appliquant simplement un vecteur de test judicieusement choisi sur les entrées du circuit. Testons maintenant le collage à `0` de la ligne i_4. Pour positionner cette ligne à la valeur inverse du collage, il suffit d'appliquer `1` sur l'entrée i_4. La phase de propagation de la faute est par contre plus compliquée que ce que l'on a vu précédemment. En effet, il y a entre cette ligne et la sortie o_3 la cellule mémoire m. La propagation de la faute va donc nécessiter un coup d'horloge, ce qui signifie qu'il va falloir appliquer deux vecteurs de test pour la tester. La séquence de patterns qui permet de tester cette faute est alors `***1*`, `*****`. De la même manière, pour tester le collage à `0` de la ligne l_2, il faut appliquer la séquence `***1*`, `****1` : le premier pattern positionne à `1` l'entrée de la cellule mémoire – ce qui positionnera à `1` la sortie de cette cellule au coup d'horloge suivant, c'est-à-dire lors de l'application du second pattern –, et le deuxième pattern propage la faute en appliquant `1` sur la deuxième entrée de la porte AND. La longueur d'une séquence de test n'est bien évidemment pas limitée à deux, mais dépend des interconnexions entre les cellules mémoires qui composent le circuit. Étudions par exemple comment tester le collage à `0` de la ligne notée l sur le circuit de la figure 7.5. Il faut déjà deux patterns pour positionner cette ligne à la valeur inverse du collage étudié (`1**`, `*1*`), et il faut ensuite un troisième pattern pour propager la faute jusqu'à la sortie o_2 (`**0`). La séquence de test complète est alors : `1**`, `*1*`, `**0`.

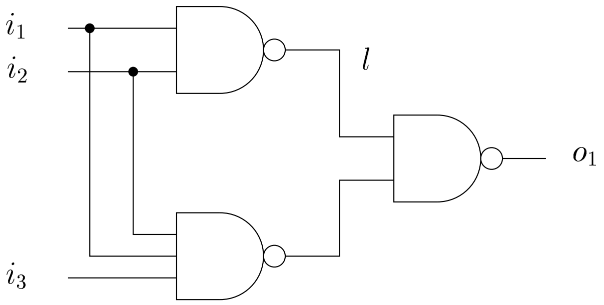


FIG. 7.6: Exemple de faute indétectable.

7.2.2 Quelques notions de testabilité

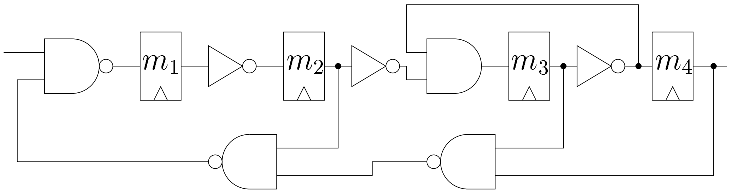
La testabilité a pour objet de quantifier la facilité avec laquelle un circuit, ou plus simplement une faute donnée, peuvent être testés. De nombreuses méthodes d'analyse de testabilité existent, aussi bien pour les cicuits combinatoires que pour les circuits séquentiels. Nous n'aborderons pas ici ces méthodes, mais regrouperons simplement sous le terme *testabilité* un ensemble de notions qui permettent d'appréhender la difficulté de la tâche du test, en particulier dans le cadre des circuits séquentiels.

7.2.2.1 Indétectabilité

Une faute affectant un circuit peut être telle qu'il n'existe pas de vecteur de test permettant de la tester. Une telle faute est déclarée indétectable. Si on observe le circuit de la figure 7.6, par exemple, le collage à 1 de la ligne i_3 est indétectable car il nécessite simultanément le positionnement à 1 des lignes i_1, i_2 et l. Si l'on s'intéresse simplement à la fiabilité d'un test, une faute indétectable ne pouvant pas affecter le bon fonctionnement du circuit (par définition), il apparaît sans importance de ne pas pouvoir la détecter. D'un autre coté – celui du concepteur d'un test –, les fautes indétectables rendent difficile, entre autres inconvénients, l'appréciation de l'efficacité du test : Ibara et Sahni [Ibarra et Sahni, 1975] ont en effet montré que, pour les circuits combinatoires, la question de l'existence d'un vecteur de test pour une faute donnée est un problème NP-complet. Il est, dans ces conditions, difficile de savoir si, au cours d'un test, une faute non testée a un impact sur la fiabilité du test ou s'il s'agit tout simplement d'une faute non testable.

7.2.2.2 Problèmes structurels

La testabilité des circuits séquentiels décroît en fonction de plusieurs paramètres structurels. Pour pouvoir les appréhender de façon simple, il faut introduire une représentation spéciale des circuits séquentiels appelée *graphe de stockage* [Cheng et Agrawal, 1989]. Un graphe de stockage est un graphe orienté qui représente les interconnexions entre cellules mémoires du circuit : chaque noeud est associé à une cellule, tandis que les arcs représentent les parties combinatoires (les lignes et les portes) qui relient ces cellules. La figure 7.7 donne un exemple de circuit et son graphe de stockage.



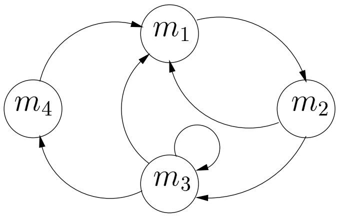


FIG. 7.7: Exemple de circuit et graphe de stockage associé.

Deux composantes du graphe de stockage sont susceptibles de faire décroître la testabilité du circuit : la profondeur séquentielle et les composantes connexes.

La profondeur séquentielle C'est le nombre de cellules mémoires du plus long chemin (ne repassant pas deux fois par la même cellule mémoire) entre une entrée et une sortie du circuit. C'est donc le chemin de longueur maximale dans le graphe de stockage. La profondeur séquentielle est connue pour augmenter la longueur d'une séquence de test de manière linéaire. En effet, dans le cas d'un circuit sans cycle, si la profondeur séquentielle du circuit est D, alors il faudra dans le pire des cas D vecteurs pour tester une faute. Cette augmentation linéaire a également été vérifiée expérimentalement par [Cheng et Agrawal, 1989].

Les composantes connexes Les composantes connexes d'un graphe de stockage ont pour effet d'augmenter de manière exponentielle la longueur nécessaire des séquences de test. Considérons une composante connexe à N sommets. Chaque cellule mémoire pouvant prendre la valeur `0` ou `1`, il existe 2^N configurations des cellules mémoires de cette composante. Considérons une faute qui, pour être testée, nécessite de placer les cellules mémoires dans une configuration bien précise. Si l'on considère que toutes les configurations sont accessibles, il faudra dans le pire des cas pour arriver à cette configuration, passer par toutes les autres configurations, c'est-à-dire appliquer 2^N vecteurs de test. Comme pour la profondeur séquentielle, ce fait a également été observé de manière expérimentale par [Cheng et Agrawal, 1989].

D'autres facteurs peuvent être responsables d'une baisse de la testabilité des circuits. On pourra noter par exemple la *sortance* (*fanout*) des cellules mémoires, c'est-à-dire le nombre de lignes qui sont contrôlées par la sortie d'une cellule, ou les problèmes de *retiming* [Marchock et al., 1995] liés au positionnement des cellules mémoires par rapport à certains composants combinatoires du circuit.

7.2.3 Deux outils

La mise en place d'une procédure de test nécessite l'utilisation de deux outils fondamentaux qui sont les programmes de recherche de séquences de test, et les programmes de simulation de séquences. Les premiers ont pour objet la construction d'une séquence de test étant donnée une faute, tandis que les seconds déterminent les fautes potentielles détectées par une séquence donnée.

7.2.3.1 Les ATPGs

Comme mentionné à la section précédente, [Ibarra et Sahni, 1975] ont prouvé dans le cadre des circuits combinatoires que la question de l'existence d'un vecteur de test pour une faute donnée est un problème NP-complet. La recherche d'un vecteur de test est donc un problème NP-difficile. Pour les circuits qui nous intéressent, c'est-à-dire les circuits séquentiels, le problème est évidemment au moins aussi dur. Les outils de recherche de séquences de tests (appelés *Automatic Test Pattern Generator* ou *ATPG*) tentent de pallier cette difficulté en utilisant diverses heuristiques. Ils se divisent en deux grandes catégories : les ATPGs utilisant des algorithmes déterministes et ceux utilisant des algorithmes non-déterministes. Dans les algorithmes déterministes on trouve, entre autres, le *D-algorithme* [Roth, 1965], ainsi que ses diverses améliorations telles que le *9V-algorithme* [Muth, 1975] ou l'algorithme *PODEM* [Goel, 1981]. Ces diverses heuristiques, d'abord introduites pour le test de circuits combinatoires ont été ensuite adaptées aux circuits séquentiels (voir par exemple [Niermann et Patel, 1991]). Parmi les algorithmes non déterministes, on trouve les algorithmes basés sur la simulation de fautes [Agrawal et al., 1989] et les algorithmes génétiques [Corno et al., 1996]. Les deux types d'approche donnent des résultats assez similaires du point de vue du taux de couverture de fautes des séquences générées. Du point de vue des séquences elles-mêmes, la différence essentielle est que les séquences générées de manière déterministe sont des séquences de patterns, c'est-à-dire que seuls les bits réellement nécessaires au test sont fixés ; les autres bits prennent la valeur `*` et peuvent être indifféremment remplacés par `0` ou `1` sans que cela n'affecte la détection de la faute pour laquelle a été construit la séquence. Les séquences générées de manière non déterministe, au contraire, sont des séquences de vecteurs booléens dont tous les bits sont fixés, y compris ceux dont la valeur n'a aucune influence sur la capacité de détection de la séquence. La raison est que le processus qui a généré ces séquences est un processus de génération probabiliste qui s'est basé sur la simulation de fautes (voir paragraphe suivant) pour évaluer les séquences construites ; il n'a donc aucun moyen de distinguer les bits réellement importants pour le test de ceux qui ne le sont pas.

7.2.3.2 Les simulateurs de fautes

La simulation de faute consiste à trouver, étant donnés un circuit et une séquence de patterns (ou de vecteurs), l'ensemble des fautes potentielles du circuit que cette séquence permet de tester. Contrairement au problème de la recherche d'une séquence de test, la simulation de faute peut se faire en temps polynômial au nombre de lignes du circuit.

Néanmoins, étant donné le nombre de plus en plus important de lignes qui composent les circuits, même polynômial le temps de calcul reste un problème crucial pour ces outils. Différentes techniques et algorithmes ont donc été introduits pour améliorer le temps de calcul en essayant de tirer profit au maximum des spécificités des circuits étudiés, mais également des machines sur lesquelles sont effectuées les simulations. Parmi ces techniques on pourra distinguer (de manière non objective) deux algorithmes assez astucieux : la simulation basée sur le tracé de chemins critiques [Abramovici et al., 1984], et la simulation parallèle [Seshu, 1965].

7.3 Procédures de test et problèmes posés

Une fois les séquences de test générées, on peut procéder au test proprement dit. Il existe pour cela deux techniques fondamentales que nous présentons dans les parties suivantes.

7.3.1 La procédure classique

Cette approche consiste à utiliser un testeur externe, c'est-à-dire un appareillage adéquat dont la tâche est d'appliquer les séquences de test générées aux entrées du circuit, et d'analyser si les valeurs produites en sortie sont bien celles logiquement attendues. Ces appareillages peuvent être utilisés pour tester un grand nombre de circuits identiques à différents stades de leur vie (à la fin de la chaîne de production des circuits aussi bien que, par la suite, afin de s'assurer de leur bon fonctionnement). Un des problèmes principaux de cette méthode est le temps nécessaire pour tester un circuit, problème d'autant plus crucial que le nombre de circuits à tester est important. Le temps de test d'un circuit est proportionnel à la taille totale des séquences de test. L'objectif principal lorsqu'on utilise cette méthode est donc la construction de séquences de test les plus petites possibles. Bon nombres de travaux ont été conduits dans ce sens. Ce problème est celui que nous abordons au chapitre 9.

7.3.2 Le test intégré

La technique de test intégré (*Built-in Self-Test* ou *BIST*) consiste à intégrer directement au circuit une partie supplémentaire dédiée à son test. Les intérêts de cette approche sont divers :

- les temps de test sont globalement plus courts car il est possible de tester en parallèle différentes parties du circuit ;
- elle permet de tester le circuit à sa vitesse nominale ;
- elle est plus souple d'utilisation car elle permet de tester le circuit lors des temps de dormance du système ;
- elle permet de tester des circuits auxquels on ne peut pas physiquement accéder ;

– elle est moins coûteuse que les testeurs externes. De plus, alors que le prix des testeurs externes croît en même temps que la complexité des nouvelles générations de circuits, le test intégré, utilisant les mêmes technologies que les circuits et bénéficiant donc des mêmes améliorations, voit son prix décroître.

La partie ajoutée dédiée au test doit effectuer les mêmes tâches que celles qui sont assurées par les testeurs externes utilisés dans l'approche classique, à savoir : 1) générer les séquences de test ; 2) analyser les réponses du circuit à ces séquences. Curieusement, alors que l'on connaît une méthode simple et efficace pour assurer l'analyse des réponses [Rajski et Tyszer, 1998], la génération des séquences de test reste un problème crucial. La raison est que, pour que le test intégré soit réellement une technique intéressante, la partie ajoutée au circuit pour assurer son test doit être la plus petite possible. On ne peut donc raisonnablement pas stocker sur le circuit toutes les séquences produites par les ATPGs. Le problème est alors de construire un générateur de séquence de taille réduite et permettant néanmoins de tester le circuit avec un taux de couverture de fautes acceptable. Il existe plusieurs techniques pour traiter ce problème dans le cadre des circuits combinatoires (voir par exemple [Touba et McCluskey, 1996] ou [Fagot et al., 1998]), mais les circuits séquentiels exigent des solutions spécifiques, et peu de travaux ont été conduits dans ce sens. C'est le problème que nous abordons au chapitre 8. Les travaux présentés dans ce chapitre ont fait l'objet de plusieurs articles, dont on peut trouver une synthèse dans [Bréhélin et al., 2001] et [Bréhélin et al., 2000].

Chapitre 8

Construction d'un générateur de séquences pour le BIST

8.1 Introduction

Le problème que nous abordons dans ce chapitre est celui de la construction d'un générateur de séquences de test pour le BIST de circuits séquentiels. Comme nous l'avons vu au chapitre précédent, un tel générateur doit allier taux de couverture de fautes élevé et place réduite. Différents types d'approches peuvent être utilisés pour résoudre ce problème. D'un coté, on peut construire un générateur de séquences reproduisant exactement les séquences de test préalablement calculées par un ATPG. Cette approche (*e.g.* [Iyengar et al., 1998]) consiste à stocker les séquences de l'ATPG sous une forme compressée (dans des ROMs par exemple) et à ajouter au circuit une structure de décodage qui va décompresser les séquences et les appliquer. Bien que cette approche permette d'obtenir des taux de couverture de fautes élevés (ceux de l'ATPG) et des temps de test minimaux, la place requise par la logique de décompression et les composants de mémorisation limite grandement son application. D'un autre coté, on peut construire des générateurs de séquences de test pseudo-aléatoires basés sur des structures de registres à décalage (*Linear Feedback Shift Register* ou *LFSR*) ou des automates cellulaires[1] (*Cellular Automata* ou *CA*). Ces générateurs sont paramétrés à l'aide de processus aléatoires [Pomeranz et Reddy, 1997] ou d'algorithmes génétiques [Corno et al., 1998] qui utilisent intensément la simulation de fautes. Les générateurs construits de cette manière sont généralement de faible taille mais ils ne permettent pas toujours d'atteindre le taux de couverture de fautes des ATPGs. De plus, la simulation de fautes étant un processus extrêmement coûteux en temps de calcul, son utilisation intensive entraîne bien souvent un temps de paramétrage excessif.

Comme on l'a évoqué au chapitre précédent, différentes séquences peuvent tester la même faute. Les séquences fournies par l'ATPG ne sont donc pas les seules séquences possibles pour tester le circuit. L'approche que nous proposons d'utiliser ici est de construire un générateur de séquences de test à l'aide des techniques de modélisation statistiques classiques appliquées aux HMMs et aux unigrams. Plus précisément, on va, à partir des séquences de l'ATPG, apprendre une instance de l'une ou l'autre de ces classes que l'on

[1] Les LFSRs sont en fait des automates cellulaires d'un type particulier.

implémentera ensuite sur silicium pour en faire un générateur. Bien qu'utilisant les mêmes techniques, cette approche diffère des applications classiques, telles que les applications de reconnaissance, pour plusieurs raisons. La première a trait à l'objet même de l'application qui est de faire de la génération. Une des conséquences est que l'on est pas sujet aux problèmes d'*overfitting* tel que c'est généralement le cas pour les applications de reconnaissance. En effet, notre objectif ici est simplement de générer les séquences de l'ATPG, ou des séquences similaires, mais non d'être capable de reconnaître des séquences inconnues. Le deuxième point qui fait que cette application diffère des applications classiques tient à la forme des données d'apprentissage. Une séquence de patterns (comme proposée par un ATPG déterministe) définit en effet un *ensemble* de séquences de vecteurs booléens (toutes les séquences que l'on peut former en remplaçant les `*` des patterns par `0` ou `1`). Notre objectif est alors de construire un générateur capable de produire au moins une des séquence de vecteurs de chaque séquence de patterns. Enfin, le dernier point qui rend cette application un peu particulière est la contrainte de place imposée par l'implémentation : les modèles que l'on va implémenter devront être de petite taille et leurs paramètres spécialement adaptés à une implémentation peu coûteuse.

Ce chapitre est organisé de la manière suivante. Dans la partie 8.2, nous présentons un nouveau modèle, appelé modèle de Markov caché avec patterns (*Hidden Markov model with patterns* ou *HMMP*) inspiré des modèles de Markov cachés classiques et spécialement adapté aux contraintes de place de notre application. Nous expliquons comment estimer les paramètres de ce modèle et proposons un algorithme d'apprentissage par fusions d'états approprié. Dans la partie 8.3, nous présentons un autre modèle (*Independence model with patterns* ou *IMP*), inspiré cette fois de la classe des unigrams et proposons également un algorithme d'apprentissage adapté. La partie 8.4 est consacrée à l'implémentation microélectronique de l'un et l'autre de ces modèles. Dans la partie 8.5, nous testons ces modèles et les algorithmes exposés sur les *benchmarks* classiques de la communauté microélectronique. Nous comparons leurs performances et vérifions que, malgré leur taille réduite, ils permettent généralement d'obtenir des taux de couverture de fautes comparables à ceux de l'ATPG.

8.2 Les HMMPs

Les HMMs que nous voulons inférer sont destinés à produire des séquences de vecteurs booléens. Différentes solutions existent pour modéliser de telles séquences. La plus naturelle est sans doute de considérer que les bits des vecteurs booléens sont indépendants et d'utiliser des produits de distributions de Bernoulli pour définir les probabilités d'émission associées aux états du HMM. Malheureusement, du fait du grand nombre de distribution de Bernoulli requises – une pour chaque bit de chaque état –, cette approche conduit à des modèles dont l'implémentation nécessite une place trop importante. Ce point est d'autant plus sensible que la taille des vecteurs (le nombre d'entrées du circuit) est importante (il existe des circuits avec plusieurs dizaines, voire centaines, d'entrées). Pour contourner cette difficulté, nous proposons d'utiliser des distributions de formes très simples, basées sur des probabilités discrètes (0, 0.5 et 1) ayant des coûts d'implémentation très faibles.

L'utilisation de ces distributions dans le cadre des HMMs conduit aux modèles de markov cachés avec patterns. La partie 8.2.1 définit notre classe de distributions et les HMMPs. La partie 8.2.2 définit la vraisemblance d'un ensemble de séquences de patterns et la manière de la calculer. Dans la partie 8.2.3 nous verrons les estimateurs de maximum de vraisemblance des paramètres des HMMPs, tandis que la partie 8.2.4 présente un algorithme d'apprentissage par fusions d'états utilisant le critère de sélection classique.

8.2.1 Définitions

Pour définir les probabilités d'émission associées aux états d'un HMMP, nous utilisons des produits de distributions de Bernoulli pour lesquelles les paramètres ne peuvent pas prendre toutes les valeurs comprises entre 0 et 1 mais sont contraints aux trois valeurs 0, 1 et 1/2. L'avantage de telles distributions pour le BIST est que ces probabilités sont aisément implémentables : la probabilité 1/2 a un coût nul, tandis que les probabilités 0 et 1 ont des coûts extrêmement faibles (nous détaillerons ce point dans la partie 8.4). Une distribution définie de cette manière peut être facilement représentée par un pattern booléen similaire à ceux des séquences de l'ATPG : les bits de paramètre 0 sont représentés par un `0`, ceux de paramètre 1 sont représentés par un `1` et ceux de probabilité 1/2 sont représentés par le caractère `*`. De tels patterns seront appelés *pattern d'émission* pour éviter les confusions avec les patterns des séquences de l'ATPG. Dans les distributions définies par un pattern d'émission, les vecteurs consistants avec le pattern sont équiprobables, tandis que les autres vecteurs ont une probabilité nulle. Par exemple, dans la distribution `*1*`, les vecteurs `010`, `011`, `110` et `111` ont la probabilité 1/4, tandis que les quatre autres vecteurs de longueur 3 ont une probabilité nulle. Contrairement aux applications de reconnaissance, l'utilisation de telles probabilités extrêmes (nulles) dans le cadre du BIST n'est pas un inconvénient. La raison est que l'on ne cherche pas à reconnaître des séquences inconnues, mais à générer des séquences les plus proches possible de celles de l'ATPG.

Un HMMP H est défini par un triplet $\langle \mathcal{Q}, \mathcal{P}, \mathcal{T} \rangle$ où

- $\mathcal{Q}$ est un ensemble d'états fini ; $\mathcal{Q}$ contient deux états spéciaux *start* et *end* qui servent respectivement à débuter et conclure une séquence. Chaque état, excepté *start* et *end*, est étiqueté par un pattern d'émission de $\mathcal{P}$.

- $\mathcal{P} = \{p_s, s \in \mathcal{Q} - \{start, end\}\}$ est l'ensemble des patterns d'émission associés aux états ; on note p_s le pattern associé à l'état s.

- $\mathcal{T} : \mathcal{Q} \times \mathcal{Q} \to [0, 1]$ est une matrice indiquant les probabilités de transition d'un état à l'autre ; on note $P(s \to s')$ la probabilité de transition de l'état s sur l'état s'.

On définit la *structure* d'un HMMP par l'ensemble de ses états et par les transitions autorisées (les transitions de probabilité non nulle). La Figure 8.1 représente un exemple de HMMP avec sept états et neuf transitions.

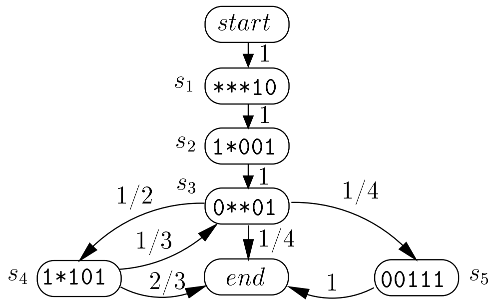


FIG. 8.1: Une exemple de HMMP. Chaque état est étiqueté par son nom et son pattern d'émission associé ; les transitions sont étiquetées par leur probabilité de transition.

8.2.2 Vraisemblance d'un ensemble de séquences de test

On dit que deux patterns p et p' sont *compatibles* si les vecteurs qu'ils recouvrent admettent une intersection non vide. Cela se traduit par le fait que leurs bits fixés (ceux qui ont la valeur 0 ou 1) ne diffèrent pas. Par exemple, $p = \texttt{11*}$ et $p' = \texttt{1*0}$ sont compatibles, tandis que $p = \texttt{11*}$ et $p' = \texttt{100}$ ne le sont pas. Soit p un pattern de test (un ensemble de vecteurs) et s un état d'un HMMP ; la *probabilité d'émission* de p par s est la probabilité qu'en générant un vecteur avec la distribution p_s on génère un vecteur de p. Si p et p_s sont incompatibles la probabilité d'émission de p par s est nulle. Pour un pattern compatible, la probabilité d'émission est directement liée au nombre de bits fixés dans p (dont la valeur est 0 ou 1), mais qui ont la valeur * dans p_s. Notons $*^p_{p_s}$ ce nombre. Par exemple, si $p_s = \texttt{1***}$ et $p = \texttt{10**}$ alors $*^p_{p_s} = 1$. Puisque un * a une équiprobabilité d'émettre 0 ou 1, la probabilité d'émettre le pattern p avec s est égale à

$$P(p|s) = \left(\frac{1}{2}\right)^{*^p_{p_s}} \tag{8.1}$$

si p et p_s sont compatibles et à 0 sinon. Par exemple, la probabilité d'émettre 1*01* avec $p_s = \texttt{1*010}$ est 1. Par contre, la probabilité d'émettre 1*010* avec $p_s = \texttt{***10*}$ est $\frac{1}{4}$.

La *probabilité de génération* $P(O|H)$ d'une séquence de patterns O par un HMMP H est la probabilité de générer avec H une séquence de vecteurs booléens qui appartienne à l'ensemble de séquences défini par O. Si on a $O = p_1 p_2 \ldots p_l$, avec $\forall i, p_i \in \{0, 1, *\}^k$, alors :

$$P(O|H) = \sum_{\langle s_0, \cdots, s_{l+1} \rangle} \left(\prod_{i=0}^{l-1} P(s_i \to s_{i+1}) P(p_{i+1}|s_{i+1}) \right) P(s_l \to s_{l+1}),$$

avec $\langle s_0, \cdots, s_{l+1} \rangle$ l'ensemble des séquences à $l+2$ éléments des éléments de $\mathcal{Q}$ telles que $s_0 = \textit{start}$ et $s_{l+1} = \textit{end}$. La *vraisemblance* d'un ensemble de séquences $\mathcal{O}$ pour H, est alors, comme dans le cadre des HMMs classiques :

$$P(\mathcal{O}|H) = \prod_{O \in \mathcal{O}} P(O|H). \tag{8.2}$$

De la même manière que pour les HMMs classiques (voir chapitre 2), on peut calculer le chemin de Viterbi d'une séquence de pattern O par un HMMP H. Considérons par exemple les trois séquences suivantes :

Séquence 1	Séquence 2	Séquence 3
***10	***10	***10
1*001	1*001	1**01
0*101	0*101	0*001
10*01	11101	00111
0**01		

Le chemin de Viterbi de la séquence 1 pour le HMMP de la Figure 8.1 est $start - s_1 - s_2 - s_3 - s_4 - s_3 - end$.

Soit $\mathcal{O}$ un ensemble de séquences de test, H un HMMP et $\mathcal{S}^*$ les chemins de Viterbi associés aux séquences de $\mathcal{O}$ dans H. En reprenant les notations introduites pour les HMMs au chapitre 2, auxquelles on adjoint $P_{\mathcal{O},s}$, l'ensemble des patterns émis par l'état s dans $\mathcal{S}^*$, et n_s^p, le nombre de fois où le pattern p est émis par l'état s dans $\mathcal{S}^*$, alors $P(\mathcal{O}, \mathcal{S}^*|H)$, la vraisemblance de $\mathcal{O}$ pour H suivant $\mathcal{S}^*$ (voir formule (2.13)), se réécrit dans le cadre des HMMPs :

$$P(\mathcal{O}, \mathcal{S}^*|H) = \prod_{s \in \mathcal{Q}} \left(\prod_{p \in P_{\mathcal{O},s}} P(p|s)^{n_s^p} \prod_{s' \in Out(s)} P(s \to s')^{n_{s \to s'}} \right). \tag{8.3}$$

En posant, comme on l'a fait pour les HMMs,

$$E_s = \prod_{p \in P_{\mathcal{O},s}} P(p|s)^{n_s^p}, \tag{8.4}$$

on obtient :

$$P(\mathcal{O}, \mathcal{S}^*|H) = \prod_{s \in \mathcal{Q}} \left(E_s \prod_{s' \in Out(s)} P(s \to s')^{n_{s \to s'}} \right). \tag{8.5}$$

8.2.3 Estimation des paramètres

De la même manière que pour les HMMs, l'estimation des paramètres d'un HMMP peut être réalisée en maximisant la vraisemblance des séquences d'apprentissage suivant leur chemin de Viterbi, c'est-à-dire ici en maximisant l'expression 8.5.

Les estimateurs des probabilités de transition sont les mêmes que ceux des HMMs classiques :

$$\widehat{P}(s \to s') = \frac{n_{s \to s'}}{n_s}. \tag{8.6}$$

Pour les probabilités d'émission, par contre, la méthode est un peu différente. Elle va consister à trouver, pour chaque état s, le pattern p_s qui maximise l'expression (8.4). On va pour cela introduire quelques notations supplémentaires. On note $\delta_*(p,r)$ la fonction binaire qui renvoie 1 si le bit de rang r dans p est égal à $*$, et 0 sinon. Soit $\delta_{\neg *}(p,r)$ la négation de cette fonction, et soit

$$n_{\neg *}(P_{\mathcal{O},s}, r) = \sum_{p \in P_{\mathcal{O},s}} n_s^p \cdot \delta_{\neg *}(p,r) \tag{8.7}$$

le nombre de patterns émis par l'état s et dont le bit de rang r est fixé. Des expressions (8.1) et (8.4), il suit que, pour que la valeur de E_s ne soit pas nulle, p_s doit être compatible avec tous les patterns de $P_{\mathcal{O},s}$. Donc s'il existe dans $P_{\mathcal{O},s}$ des patterns dont certains bits sont fixés de manière différente, p_s doit avoir sur ces positions la valeur $*$. Cette contrainte étant satisfaite, en intégrant l'expression (8.1) à la formule (8.4) on obtient :

$$E_s = \left(\frac{1}{2}\right)^{\sum_{p \in P_{\mathcal{O},s}} *^p_{p_s} n_s^p}, \tag{8.8}$$

qui peut être réécrite

$$E_s = \left(\frac{1}{2}\right)^{\sum_{r=1}^{k} \delta_*(p_s,r) \cdot n_{\neg *}(P_{\mathcal{O},s},r)}. \tag{8.9}$$

Maximiser cette expression est équivalent à minimiser, pour chaque r, l'expression

$$\delta_*(p_s, r) \cdot n_{\neg *}(P_{\mathcal{O},s}, r). \tag{8.10}$$

On a vu que, pour une position donnée, s'il existe dans $P_{\mathcal{O},s}$ des bits incompatibles alors p_s a sur cette position la valeur $*$. Maintenant, si tous les bits à cette position sont compatibles il y a deux cas de figure possibles. Soit certains de ces bits sont fixés à une valeur donnée (0 ou 1) et pour minimiser (8.10) il faut que le bit de p_s correspondant soit également fixé à cette valeur car on a alors $\delta_*(p_s, r) = 0$. Soit tous les bits à cette position ont la valeur $*$. On a alors $n_{\neg *}(P_{\mathcal{O},s}, r) = 0$ et toutes les solutions possibles (0, 1 ou $*$) minimisent (8.10). Néanmoins, nous verrons par la suite qu'il est intéressant de garder p_s le plus général possible, et nous choisissons donc de garder le bit correspondant à la valeur $*$.

Nous venons de voir que le pattern p_s peut être estimé en calculant indépendemment chaque bit de p_s à partir des bits correspondants dans les patterns de $P_{\mathcal{O},s}$. Nous allons maintenant voir que ce calcul peut être effectué itérativement, ce qui sera très utile dans l'algorithme d'apprentissage. On utilise pour cela un opérateur commutatif et associatif que nous nommons γ. En suivant ce que l'on a dit précédemment, on a, pour les bits incompatibles $\gamma(1,0) = *$, tandis qu'on a pour les bits compatibles : $\gamma(0,0) = \gamma(*,0) = 0$, $\gamma(1,1) = \gamma(*,1) = 1$, et $\gamma(*,*) = *$. Cependant, défini ainsi, γ n'est pas associatif puisque $\gamma(1,\gamma(1,0)) = \gamma(1,*) = 1$, alors que $\gamma(\gamma(1,1),0) = \gamma(1,0) = *$ qui est la bonne solution puisqu'il y a des bits incompatibles. Le problème vient du fait que le caractère $*$ résultant de l'opération $\gamma(1,0) = *$ doit rester à cette valeur lors des opérations futures. On va donc pour cela *marquer* les $*$ obtenus de cette manière. Les $*$ ainsi marquées seront notées $\bar{*}$. L'opérateur γ prend cela en compte, et on a : $\gamma(\bar{*},0) = \bar{*}$, $\gamma(\bar{*},1) = \bar{*}$ et $\gamma(\bar{*},*) = \bar{*}$. Le tableau 8.1 résume la table des réponses de l'opérateur γ. Si on a, par exemple, $P_{\mathcal{O},s} = \{10*, 110, *1*\}$ alors $p_s = \gamma(10*, \gamma(110, *1*)) = \gamma(10*, 110) = 1\bar{*}0$.

γ	1	0	$*$	$\bar{*}$
1	1	$\bar{*}$	1	$\bar{*}$
0	$\bar{*}$	0	0	$\bar{*}$
$*$	1	0	$*$	$\bar{*}$
$\bar{*}$	$\bar{*}$	$\bar{*}$	$\bar{*}$	$\bar{*}$

TAB. 8.1: Table des réponses de l'opérateur γ.

Algorithme 17: ALGORITHME D'APPRENTISSAGE D'UN HMMP

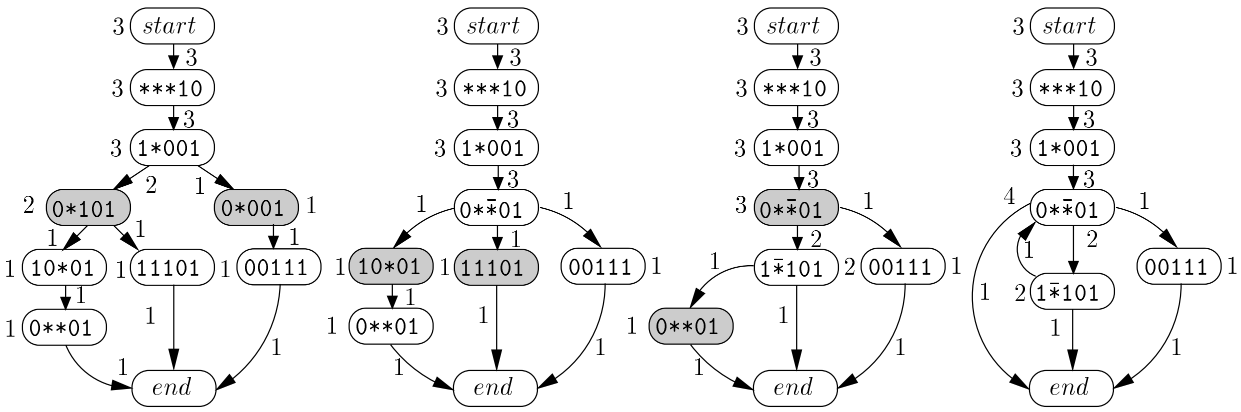
Données : $\mathcal{O}$ un ensemble de séquences d'apprentissage
N la taille du HMMP final

Créer un HMMP initial H_0;
tant que $|\mathcal{Q}| > N$ **faire**
  Sélectionner le couple (u, u') qui minimise la perte de vraisemblance;
  Fusionner u et u';

8.2.4 Apprentissage

Soit $\mathcal{O}$ un ensemble de séquences de patterns (par exemple générées par un ATPG). Notre but est de construire un HMMP de petite taille – c'est-à-dire avec un nombre d'états et de transitions peu important – qui génère $\mathcal{O}$ avec une probabilité aussi élevée que possible. Comme nous n'avons aucune connaissance *a priori* sur la structure de ce HMMP, une solution pourrait être d'entraîner plusieurs HMMPs entièrement connectés – le graphe de leur structure est une clique – et de différentes tailles avec l'algorithme de Baum-Welch ou une de ses variantes, et de garder le meilleur modèle obtenu à l'issu de cette phase. Malheureusement, comme nous le verrons à la section 8.6, la nature non continue des probabilités définies par les patterns d'émission a un fort impact sur l'efficacité de ces algorithmes. Une meilleure solution pour appréhender ces distributions particulières est d'utiliser l'algorithme d'apprentissage classique par fusions d'états que l'on a présenté aux chapitres 3 et 4.

La procédure d'apprentissage générale (algorithme 17) est la même que celle utilisée dans le cadre des HMMs classiques. La figure 8.2 représente une trace des HMMPs obtenus lors de l'exécution de cet algorithme sur les trois séquences proposées à la section 8.2.2. L'algorithme s'arrête lorsque le nombre d'états désirés (N) est atteint. N est simplement utilisé pour stopper l'algorithme, et le HMMP final n'est pas nécessairement celui qui sera utilisé comme générateur et implémenté sur le circuit. Contrairement à ce qui se passe pour les tâches de reconnaissance, choisir la taille du HMMP désiré dans le cadre du BIST n'est pas problématique. La méthode consiste à simuler les HMMPs dans l'ordre décroissant de leur taille (comme générés par l'algorithme), de calculer le taux de couverture de fautes de chacun d'eux, et de choisir le meilleur suivant les contraintes de place et de taux de couverture fixées. La procédure de création du HMMP initial est une simple adaptation de la procédure utilisée pour les HMMs et consiste à construire l'arbre préfixe des séquences d'apprentissage. Les procédures de fusion d'états et surtout de sélection



$H_0 : P(\mathcal{O}|H_0) = 6/27$ $H_1 : P(\mathcal{O}|H_1) = 1/36$ $H_2 : P(\mathcal{O}|H_2) = 1/144$ $H_3 : P(\mathcal{O}|H_3) = 3/4096$

FIG. 8.2: HMMPs obtenus lors d'une suite de trois fusions d'états. On passe d'un HMMP à l'autre en fusionnant les états grisés. En dessous de chaque HMMP est notée la vraisemblance des données. Les états et transitions sont étiquetés par le nombre de fois où il sont utilisés dans les chemins de Viterbi.

des états à fusionner, par contre, font l'objet d'adaptations plus conséquentes et méritent d'être détaillées.

8.2.4.1 Fusion de deux états

La procédure de fusion de deux états d'un HMMP est décrite par l'algorithme 18. La différence essentielle avec l'algorithme de fusion utilisé pour les HMMs classiques (algorithme 7) concerne l'estimation des probabilités d'émission du nouvel état qui est maintenant réalisée *via* l'opérateur γ. L'associativité de cet opérateur prend alors tout son sens : plutot que de calculer $p_{u\otimes u'}$ à partir des patterns de $P_{\mathcal{O},u} \cup P_{\mathcal{O},u'}$, il est directement calculé à partir des patterns d'émission p_u et $p_{u'}$.

8.2.4.2 Sélection des états à fusionner

La critère de sélection des états à fusionner est sensiblement le même que le critère classique utilisé pour les HMMs : on va fusionner à chaque étape de l'algorithme les états qui entraînent une baisse minimale de la vraisemblance. Néanmoins, quelquefois (spécialement au début du processus), plusieurs paires engendrent une baisse minimale de vraisemblance. On va alors choisir parmi ces paires celle dont la fusion requiert de fixer un nombre minimal de bits – c'est-à-dire entraîne la disparition d'un nombre minimal de $*$. Le nombre de bits à fixer lors de la fusion de deux patterns p et p' est calculé de la manière suivante :

$$\varphi(p, p') = \min(*^{p}_{\gamma(p,p')}, *^{p'}_{\gamma(p,p')}). \tag{8.11}$$

Par exemple, si $p = \mathtt{1{*}11{*}}$ et $p' = \mathtt{{*}0\bar{*}{*}{*}}$, alors $\gamma(p, p') = \mathtt{10\bar{*}1{*}}$. On a $*^{p}_{\gamma(p,p')} = 1$, $*^{p'}_{\gamma(p,p')} = 2$ et donc $\varphi(p, p') = 1$. Notons que si p est plus général que p' (ou inversement),

---

Algorithme 18: PROCÉDURE DE FUSION D'ÉTATS D'UN HMMP

Données : H, un HMMP
u et u' deux états à fusionner

ajouter un nouvel état $u \otimes u'$ à H;
$n_{u\otimes u'} \leftarrow n_u + n_{u'}$;
$p_{u\otimes u'} \leftarrow \gamma(p_u, p_{u'})$;
pour chaque $t \in In(u) \cup In(u') - \{u, u'\}$ **faire**
- ajouter la transition $t \to u \otimes u'$;
- $n_{t\to u\otimes u'} \leftarrow n_{t\to u} + n_{t\to u'}$;
- estimer $P(t \to u \otimes u')$ à l'aide de l'expression (8.6);

pour chaque $t \in Out(u) \cup Out(u') - \{u, u'\}$ **faire**
- ajouter la transition $u \otimes u' \to t$;
- $n_{u\otimes u'\to t} \leftarrow n_{u\to t} + n_{u'\to t}$;
- estimer $P(u \otimes u' \to t)$ à l'aide de l'expression (8.6);

si $(Out(u) \cup Out(u')) \cap \{u, u'\} \neq \emptyset$ **alors**
- ajouter la transition $u \otimes u' \to u \otimes u'$;
- $n_{u\otimes u'\to u\otimes u'} \leftarrow n_{u\to u'} + n_{u\to u'} + n_{u\to u'} + n_{u\to u'}$;
- estimer $P(u \otimes u' \to u \otimes u')$ à l'aide de l'expression (8.6);

supprimer u et u' (et leurs transitions adjacentes);

---

alors $\varphi(p, p') = 0$. L'utilisation de la formule (8.11) a pour effet de préserver un nombre maximal de $*$ et de faciliter ainsi les fusions à venir en terme de conservation de la vraisemblance. À chaque étape de l'algorithme, on choisit donc la paire d'états maximale suivant l'ordre lexicographique $\succ$ définit comme suit :

$(u, u') \succ (t, t')$ *ssi* une des deux conditions suivantes est remplie :

$$P(\mathcal{O}, \mathcal{S}^*|H_{u\otimes u'}) > P(\mathcal{O}, \mathcal{S}^*|H_{t\otimes t'}),$$
$$P(\mathcal{O}, \mathcal{S}^*|H_{u\otimes u'}) = P(\mathcal{O}, \mathcal{S}^*|H_{t\otimes t'}) \text{ et } \varphi(p_u, p_{u'}) < \varphi(p_t, p_{t'}),$$

avec $H_{u\otimes u'}$ le HMMP résultant de la fusion de u et u'.

L'utilisation de cet ordre lexicographique à la place du critère de sélection classique ne remet pas en cause la méthode de sélection rapide exposée au chapitre 4. La valeur de $\varphi(p_u, p_{u'})$ ne change pas tant que u et u' ne sont pas fusionnés et n'a donc pas à être recalculée après chaque fusion. On peut alors utiliser, comme on l'a fait pour les HMMs, la liste $\mathcal{L}$ des paires d'états ordonnée suivant $\succ$ et la quantité $\theta(u, u')$ définie par l'expression (4.4). La seule différence est la procédure de calcul des termes E_s de cette expression. Ces termes peuvent être calculés en utilisant la formule (8.8) mais cette méthode peut être gourmande en temps de calcul lorsque $P_{\mathcal{O},s}$ contient de nombreux patterns. Une meilleure approche est d'utiliser les $n_{\neg *}(P_{\mathcal{O},s}, r)$ définis par l'expression (8.7). En effet, si u et u' sont fusionnés en un état $u \otimes u'$, on a :

$$n_{\neg *}(P_{\mathcal{O},u\otimes u'}, r) = n_{\neg *}(P_{\mathcal{O},u}, r) + n_{\neg *}(P_{\mathcal{O},u'}, r), \tag{8.12}$$

et on peut donc calculer $E_{u\otimes u'}$ en $O(k)$ grâce à la formule (8.9). Au début du processus,

Algorithme 19: Création de la liste des couples d'états d'un HMMP

Données : H, un HMMP

$\mathcal{L} \leftarrow$ liste vide;
pour chaque $s \in \mathcal{Q}$ **faire**
  Calculer et stocker $n_{\neg *}(P_{\mathcal{O},s}, r)\ \forall r \in [1, k]$ en utilisant la formule (8.13);
  Stocker $E_s = 1$;
pour chaque couple d'états (s, s') **faire**
  Calculer $p_{s \otimes s'} = \gamma(p_s, p_{s'})$;
  Calculer et stocker $n_{\neg *}(P_{\mathcal{O},s\otimes s'}, r)\ \forall r \in [1, k]$ en utilisant la formule (8.12);
  Calculer et stocker $E_{s \otimes s'}$ en utilisant la formule (8.9);
  Calculer $T(s, s')$ à l'aide de la formule (4.2);
  Calculer $\theta(s, s')$ à l'aide de la formule (4.4);
  Calculer $\varphi(p_s, p_{s'})$ à l'aide de la formule (8.11);
  Insérer (s, s') à sa place dans $\mathcal{L}$;

Algorithme 20: Mise à jour de la liste des couples d'états d'un HMMP

Données : H, $\mathcal{L}$, u, u', $u \otimes u'$

Supprimer tous les couples d'états de $\mathcal{L}$ contenant u, u', ou tout état s appartenant à $In(u, u') \cup Out(u, u')$;
pour chaque $s \in \{u \otimes u'\} \cup In(u, u') \cup Out(u \otimes u')$ **faire**
  pour chaque $s' \neq s$ **faire**
    si $s = u \otimes u'$ **alors**
      Calculer $p_{s \otimes s'} = \gamma(p_s, p_{s'})$;
      Calculer et stocker $n_{\neg *}(P_{\mathcal{O},s\otimes s'}, r)\ \forall r \in [1, k]$ en utilisant la formule (8.12);
      Calculer et stocker $E_{s \otimes s'}$ en utilisant la formule (8.9);
    Calculer $T(s, s')$ à l'aide de la formule (4.2);
    Calculer $\theta(s, s')$ à l'aide de la formule (4.4);
    Calculer $\varphi(p_s, p_{s'})$ à l'aide de la formule (8.11);
    Insérer (s, s') à sa place dans $\mathcal{L}$;

$P_{\mathcal{O},s}$ est réduit au singleton $\{p_s\}$, et les termes $n_{\neg *}(P_{\mathcal{O},s}, r)$ sont initialisés à

$$n_{\neg *}(P_{\mathcal{O},s}, r) = n_s \cdot \delta_{\neg *}(p_s, r). \tag{8.13}$$

On peut maintenant proposer une version détaillée des procédure de création et de mise à jour de $\mathcal{L}$ dans le cadre particulier des HMMPs (algorithmes 19 et 20).

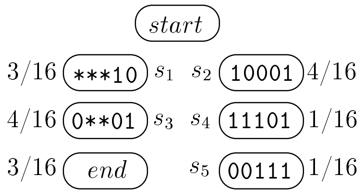


FIG. 8.3: Un exemple de modèle d'indépendance avec patterns.

8.3 Les IMPs

Le *modèle d'indépendance avec patterns* est un modèle inspiré à la fois des HMMPs et des unigrams. Comme les HMMPs il est composé d'un ensemble d'états étiquetés par des patterns d'émission. Comme les unigrams il définit un processus stochastique dépourvu de mémoire : la probabilité de transition sur un état ne dépend pas de l'état actuel mais d'une probabilité *a priori* sur l'ensemble des états. Un IMP I est défini par un triplet $\langle \mathcal{Q}, \mathcal{P}, \pi \rangle$ où :

- $\mathcal{Q}$ est un ensemble d'états fini ; comme pour les HMMPs, $\mathcal{Q}$ contient deux états spéciaux *start* et *end* qui servent respectivement à débuter et conclure une séquence. Chaque état, excepté *start* et *end*, est étiqueté par un pattern d'émission de $\mathcal{P}$.
- $\mathcal{P} = \{p_s, s \in \mathcal{Q} - \{start, end\}\}$ est l'ensemble des patterns d'émission associés aux états ; on note p_s le pattern associé à l'état s.
- $\pi : \mathcal{Q} \to [0,1]$ est la distribution de probabilité *a priori* sur les états ; on note $P(s)$ la probabilité *a priori* de l'état s.

La figure 8.3 représente un IMP à sept états.

8.3.1 Vraisemblance d'un ensemble de séquences de test

La probabilité de génération d'un pattern p par un état s est la probabilité $P(p|s)$ de générer un vecteur de p à l'aide de la distribution p_s. Elle se calcule de la même manière que pour les HMMPs et est définie par l'expression (8.1).

De même, la probabilité de génération $P(O|I)$ d'une séquence de patterns O par un IMP I est la probabilité de générer avec I une séquence de vecteurs booléens qui appartienne à l'ensemble de séquences défini par O. Si on a $O = p_1 p_2 \dots p_l$, avec $\forall i, p_i \in \{\mathtt{0}, \mathtt{1}, *\}^k$, alors :

$$P(O|I) = \sum_{\langle s_0, \cdots, s_{l+1} \rangle} \left(\prod_{i=1}^{l} P(s_i) P(p_i|s_i) \right) P(s_{l+1}),$$

avec $\langle s_0, \cdots, s_{l+1} \rangle$ l'ensemble des séquences à $l+2$ éléments des éléments de $\mathcal{Q}$ telles que $s_0 = start$ et $s_{l+1} = end$. La vraisemblance d'un ensemble de séquences $\mathcal{O}$ pour I est :

$$P(\mathcal{O}|I) = \prod_{O \in \mathcal{O}} P(O|I).$$

Algorithme 21: ALGORITHME D'APPRENTISSAGE D'UN IMP

Données : $\mathcal{O}$ un ensemble de séquences d'apprentissage
N la taille de l'IMP final

Créer un IMP initial I_0;
tant que $|\mathcal{Q}| > N$ **faire**
  Sélectionner le couple (u, u') maximal au sens de $\succ$;
  Fusionner u et u';

Comme dans le cadre des HMMs classiques et des HMMPs, on peut associer à une séquence de patterns O son chemin de Viterbi dans un IMP I. Par exemple, le chemin de Viterbi de la première séquence proposée à la section 8.2.2 dans l'IMP de la figure 8.3 est $start - s_1 - s_2 - s_3 - s_2 - s_3 - end$. Soit $\mathcal{O}$ un ensemble de séquences de test, I un IMP et $\mathcal{S}^*$ les chemins de Viterbi associés aux séquences de $\mathcal{O}$ dans I. En reprenant les notations introduites pour les HMMs et les HMMPs, $P(\mathcal{O}, \mathcal{S}^*|I)$, la vraisemblance de $\mathcal{O}$ pour I suivant $\mathcal{S}^*$, s'écrit :

$$P(\mathcal{O}, \mathcal{S}^*|I) = \prod_{s \in \mathcal{Q}} \left(P(s)^{n_s} \prod_{p \in P_{\mathcal{O},s}} P(p|s)^{n_s^p} \right). \tag{8.14}$$

8.3.2 Estimation des paramètres

De la même manière que pour les HMMPs, l'estimation des paramètres d'un IMP peut être réalisée en maximisant la vraisemblance des séquences d'apprentissage suivant leur chemin de Viterbi (expression (8.14)). Maximiser (8.14) revient à maximiser indépendemment les sous-produits $P(s)^{n_s}$ et $\prod_{p \in P_{\mathcal{O},s}} P(p|s)^{n_s^p}$. On obtient alors comme estimateur des probabilités *a priori* :

$$\widehat{P}(s) = \frac{n_s}{n_{total}}, \tag{8.15}$$

avec n_{total} la somme des patterns des séquences d'apprentissage. Quant aux probabilités d'émission associées aux états, les estimateurs sont calculés de la même manière que pour les HMMPs, *via* l'opérateur γ.

8.3.3 Apprentissage

Soit $\mathcal{O}$ un ensemble de séquences de patterns (par exemple proposées par un ATPG). Comme pour les HMMPs, notre but est de construire un IMP de petite taille – c'est-à-dire avec un nombre d'états peu important – qui génère $\mathcal{O}$ avec une probabilité aussi élevée que possible. Nous proposons d'utiliser pour cela un algorithme d'apprentissage par fusions d'états similaire à celui utilisé dans le cadre des HMMPs. Cet algorithme est illustré par l'algorithme 21.

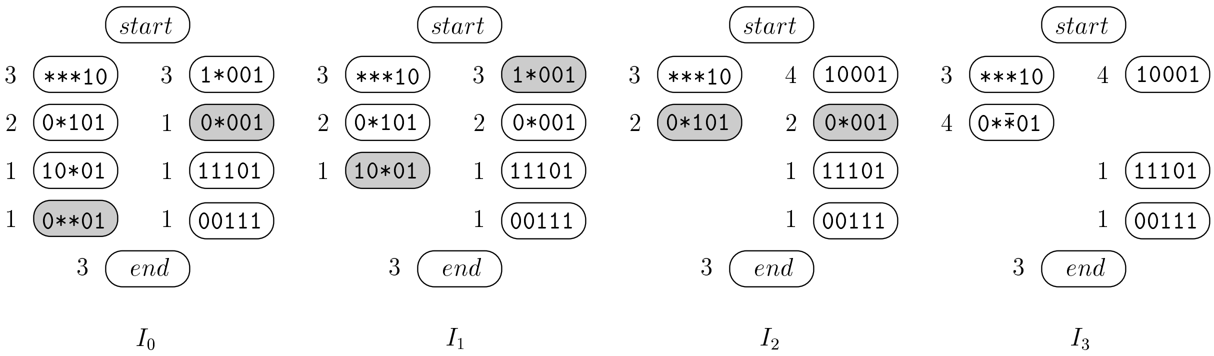


FIG. 8.4: IMPs obtenus lors d'une suite de trois fusions d'états. On passe d'un IMP à l'autre en fusionnant les états grisés. Les états sont étiquetés par le nombre de fois où il sont utilisés dans les chemins de Viterbi, et on a $P(s) = \frac{n_s}{16}$ (voir expression (8.15)).

Création de l'IMP initial La procédure consiste à créer un état pour les différents patterns des séquences d'apprentissage. On associe alors à chaque état s son pattern d'émission p_s et le nombre de fois n_s où il est utilisé dans les séquences d'apprentissage. On crée le *start* et le *end* et on associe à ce dernier le nombre de fois où il est utilisé, c'est-à-dire le nombre de séquences de $\mathcal{O}$. Enfin, on stocke dans n_{total} le nombre de patterns total de $\mathcal{O}$. L'IMP I_0 de la figure 8.4 est l'IMP initial obtenu à partir des trois séquences d'apprentissage proposées à la section 8.2.2.

Fusion de deux états La procédure est similaire à celle utilisée pour les HMMPs. Les deux états sélectionnés u et u' sont supprimés et remplacés par un unique état $u \otimes u'$. $p_{u \otimes u'}$ est calculé grâce à l'opérateur γ à partir des anciens patterns d'émission p_u et $p_{u'}$, et on a $n_{u \otimes u'} = n_u + n_{u'}$.

Sélection des états à fusionner Le même critère de sélection est utilisé : à chaque tour de l'algorithme, on sélectionne le couple d'états maximal au sens de l'ordre lexicographique $\succ$. Comme pour les HMMPs, une liste ordonnée de l'ensemble des couples d'états possibles peut être utilisée pour accélérer la sélection. Les procédures de création et mise à jour de cette liste fonctionnent exactement sur le même principe, à ceci près qu'une fusion n'affecte que les couples d'états formés avec l'un des états fusionnés, et donc que seuls les couples formés avec l'état résultant de la fusion doivent être pris en compte pour remettre la liste à jour.

8.4 Implémentation microélectronique

Après apprentissage, si le HMMP ou l'IMP obtenu génère des séquences de test ayant un taux de couverture de faute acceptable, il peut être physiquement implémenté sur le circuit. Comme pour toute implémentation BIST, l'objectif principal est d'utiliser le minimum de place.

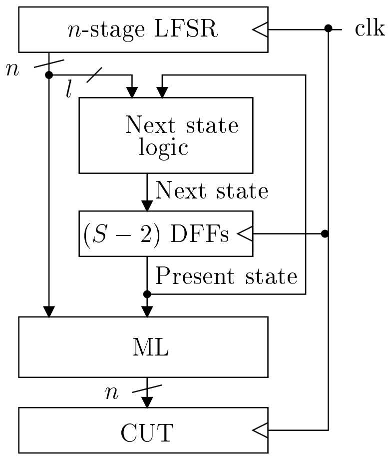


FIG. 8.5: Diagramme d'implémentation d'un générateur BIST.

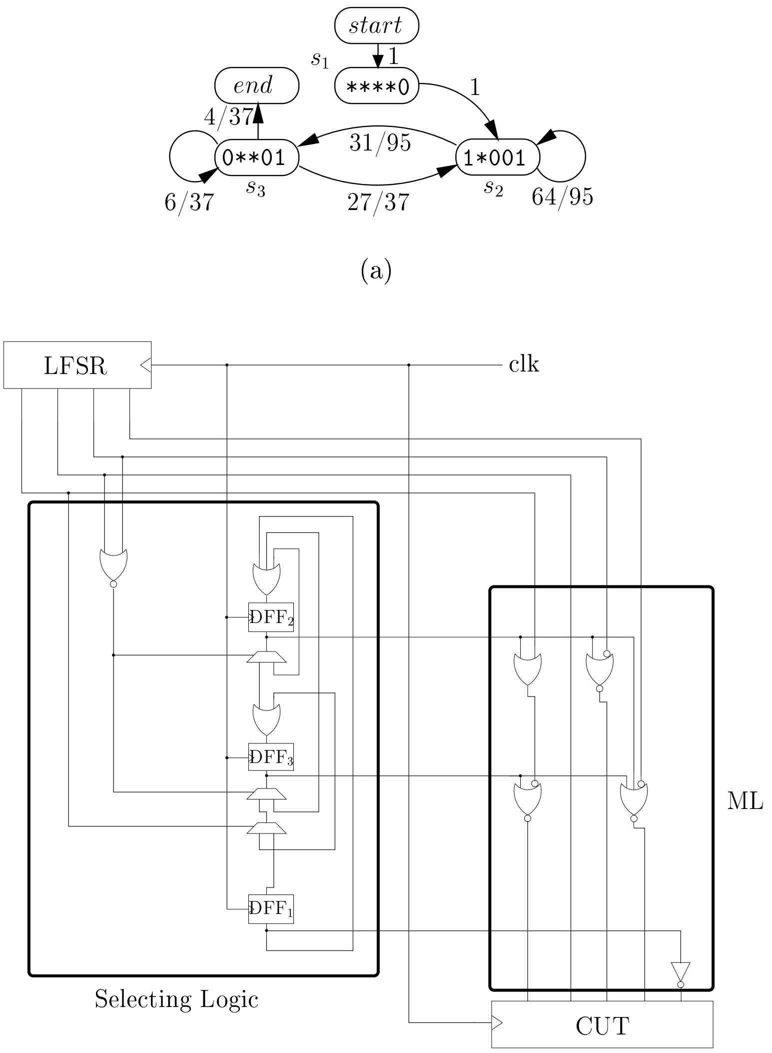
8.4.1 Implémentation d'un HMMP

Le HMMP est implémenté par un machine d'état fini de Mealy branchée sur les sorties d'un LFSR de taille n (voir figure 8.5). Les états du HMMP sont codés par des cellules mémoire (bascules) de type D (DFF). Seulement $(|\mathcal{Q}|-2)$ DFFs sont nécessaires à l'implémentation car les états *start* et *end* ne sont pas réellement codés. Une seule DFF est positionnée à `1` à une période donnée. Les sorties du LFSR sont considérées comme aléatoires et sont utilisées :

1. pour implémenter les probabilités de transition grâce à la fonction *Next state logic* de la machine de Mealy ;
2. pour produire les valeurs booléennes aléatoires à la place des $*$ (et $\bar{*}$) dans les patterns d'émission. Ceci est réalisé par la fonction de sortie de la machine de Mealy (dénotée ML sur la figure).

Le LFSR, la machine de Mealy et le circuit (CUT) sont supposés synchronisés et utilisent la même horloge (clk).

Décrivons maintenant la manière dont chaque bloc de l'architecture est implémenté. Considérons par exemple le HMMP de la figure 8.6.(a). L'architecture de BIST correspondante est illustrée à la figure 8.6.(b). Comme nous l'avons mentionné, les états *start* et *end* ne sont pas physiquement implémentés. Puisque nous désirons utiliser notre générateur de manière continue – pour qu'il génère plusieurs séquences de test les unes après les autres –, les états entrant dans l'état *end* sont directement liés aux états sortants de l'état *start* avec une probabilité égale au produit des deux probabilités de transition. Dans notre exemple, le seul état entrant dans *end* (état s_3) est relié au seul état sortant de l'état *start*, et cette transition a la probabilité $4/37 \times 1 = 4/37$. Les interconnexions entre DFFs sont déterminées à partir des transitions du HMMP. Par exemple, trois transitions mènent à l'état s_2 dans la figure 8.6.(a), et trois lignes sont donc connectées à l'entrée de la DFF correspondante (DFF_2) *via* une porte OR dans la figure 8.6.(b). Les probabilités associées à ces transitions sont discrétisées de manière à être aussi proche que possible des valeurs calculées lors de l'apprentissage, et sont implémentées en utilisant un ensemble de



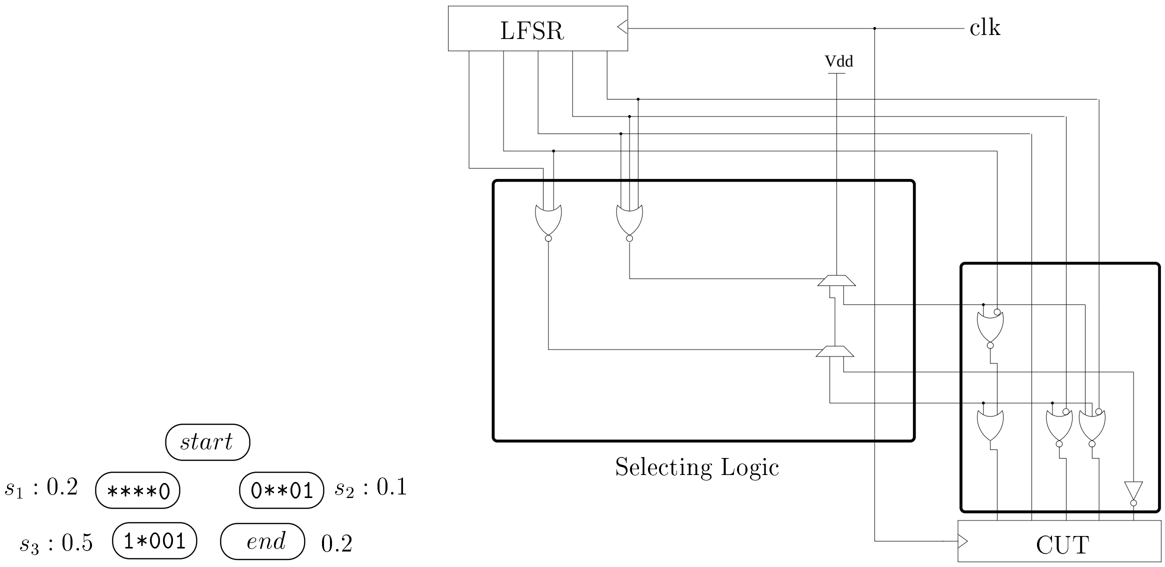
(a)

(b)

FIG. 8.6: Exemple de HMMP (a) et implémentation microélectronique associée (b).

démultiplexeurs, de portes NOR, ainsi que les lignes de sortie du LFSR. Par exemple, la transition $s_3 \rightarrow s_1$ a la probabilité 4/37 et est approximée par la probabilité 1/4 (sortie du NOR) $\times$ 1/2 (sortie du LFSR) =1/8 dans l'implémentation physique.

Le bloc *Mapping Logic* (ML) est utilisé pour implémenter les patterns d'émission. Dans le HMMP de la figure 8.6.(a), par exemple, trois patterns d'émission doivent être implémentés (`****0`, `1*001` et `0**01`). Comme on l'a mentionné, les bits `*` et $\bar{*}$ sont produits par les sorties du LFSR. Les bits fixés (`0` ou `1`) sont quand à eux implémentés par des portes logiques. Par exemple, lorsque le pattern p_{s_1} est activé, seul le dernier bit doit être positionné à une valeur fixée (`0`). Les autres bits peuvent n'importe quelle valeur. Puisque ce bit est positionné à `1` pour les deux autres patterns d'émission, une simple porte NOT sur l'entrée correspondante du circuit est suffisante pour implémenter ce pattern



(a) (b)

FIG. 8.7: Exemple de IMP (a) et implémentation microélectronique associée (b).

d'émission. Maintenant, si le pattern p_{s_3} est sélectionné à un tour d'horloge ultérieur (ce qui dépend du chemin suivi à l'intérieur du HMMP), le premier et le quatrième bit doivent être positionnés à la valeur 0. On réalise cela en utilisant deux portes NOR avec une entrée inversée sur la première et quatrième ligne du bloc ML. Comme on peut le voir sur cet exemple, peu de portes logiques sont nécessaires à l'implémentation de la structure interne du bloc ML. De plus, pour les circuits nécessitant de plus grosses logiques de sélection et des blocs ML plus importants, les outils de synthèse logique classiques peuvent être utilisés pour réduire davantage la place du générateur.

8.4.2 Implémentation d'un IMP

Comme on peut s'y attendre, l'implémentation microélectronique d'un IMP se révèle moins coûteuse en place que celle des HMMPs. La manière de concevoir chaque bloc de l'architecture BIST à partir d'un IMP donné est en fait très similaire à celle que l'on vient de décrire pour les HMMPs, et nous ne la détaillerons donc pas. La principale différence entre les deux implémentations est l'absence de DFFs dans l'architecture BIST des IMPs qui résulte de l'absence de transitions de ce modèle. La figure 8.7.(b) illustre par exemple l'architecture de BIST obtenue à partir de l'IMP de la figure 8.7.(a).

8.5 Résultats expérimentaux

Les performances de notre méthode ont été évaluées sur les circuits *benchmark* ISCAS'89[2]. Les test ont été effectués pour chaque circuit de la manière suivante. Pour commencer, dix séquences pseudo-aléatoires générées par un LFSR avec différentes configuration initiales[3] ont été simulées et la séquence obtenant le meilleur taux de couverture de faute a été retenue. De telles séquences générées par un processus stochastique détectent de nombreuses fautes (faciles) en début de processus, puis la fréquence de détection de nouvelles fautes décroît, et après un certain temps, aucune nouvelle faute ne semble pouvoir être encore détectée. La longueur des séquences du LFSR a donc été limitée à cette portion efficace. Nous avons ensuite utilisé un ATPG déterministe séquentiel pour générer un ensemble de séquences de test. Les séquences ne détectant que des fautes déjà détectées par la séquence de LFSR choisie ont été éliminées, et nous avons appliqué nos deux algorithmes d'apprentissage sur les séquences restantes. Les HMMPs et IMPs inférés ont alors été implémentés comme exposé à la section 8.4 et utilisés pour générer chacun 10 ensembles de nouvelles séquences de test. La longueur totale de ces séquences a été choisie de la même manière que pour les séquences du LFSR, et nous avons retenu parmi ces dix ensembles de test celui détectant le plus grand nombre de fautes non encore détectées par la séquence de LFSR choisie. Notons que notre méthode peut être adaptée à toutes les configurations de *set* et *reset* des cellules mémoires du circuit. Dans nos expériences, nous avons considéré un *reset* global sur ces cellules mémoires, et toutes les séquences générées par l'ATPG débutent donc par un tel *reset*.

Les résultats sont reportés dans le tableau 8.2. Après le nom du circuit, le nombre de ses entrées primaires (#PI), de ses cellules mémoires (#FF) et de ses fautes potentielles (#F), sont reportés la taille totale et le taux de couverture de fautes des séquences de l'ATPG. Dans les colonnes suivantes sont indiquées la taille totale et le taux de couverture de la séquence de LFSR choisie pour chaque circuit. Les colonnes suivantes indiquent les résultats obtenus en utilisant les HMMPs et les IMPs. La colonne *Lg* indique la taille totale des séquences de test générées par les HMMPs et les IMPs – pour des raisons de comparaison, nous avons utilisé la même taille pour tester les HMMPs et les IMPs. Les colonnes *%F* indiquent le meilleur taux de couverture obtenu lors des dix simulations pour chaque modèle et chaque circuit – c'est-à-dire les fautes détectées par le HMMP ou l'IMP *plus* les fautes déjà détectées par la séquence de LFSR seule. La colonne *#S* indique le nombre d'états des HMMPs et IMPs. Ce nombre a été ajusté expérimentalement pour tenir compte des contraintes de place et de taux de couverture et a, lui aussi, été fixé à la même valeur pour les deux modèles pour des raisons de comparaison. Dans la colonne suivante (*#E*) est reporté le nombre de transitions de probabilité non nulle des HMMPs. Les colonne *%S* indique le rapport de la taille du générateur obtenu sur la taille du circuit. Ces tailles ont été calculées en nombre de *portes équivalentes* (*Gate Equivalent* ou *GE*)

[2]Ces *benchmarks* ont été créés à l'occasion du *International Symposium on Circuits & Systems* (*ISCAS*) de 1989. Ils sont disponibles à l'adresse `http://www.cbl.ncsu.edu/benchmarks/`

[3]Un LFSRs de taille k (lorsqu'il est convenablement paramétré) a la propriété de générer une séquence contenant $2^k - 1$ vecteurs booléens de taille k (tous les vecteurs de cette taille à l'exclusion du vecteur 0^k). La configuration initiale (le *seed* en anglais) détermine le premier vecteur de la séquence. En changeant de configuration initiale, on change donc la séquence de vecteurs générée.

Circuit				ATPG		LFSR		HMMP						IMP		
Nom	#I	#FF	#F	Lg	%F	Lg	%F	Lg	%F	S	E	%S	Tmp	%F	%S	Tmp
s298	3	14	596	2472	89.9	1K	63.8	1K	89.9	3	6	13.2	5.5	89.3	5.7	0.01
s344	9	15	670	414	97.6	1K	93.9	1K	97.6	3	7	18.3	0.3	97.6	9.0	0.02
s382	3	21	764	13930	94.0	1K	14.4	5K	94.5	4	9	15.7	327.2	89.8	6.9	0.02
s386	7	6	772	736	90.2	1K	50.3	2K	89.6	3	7	11.7	0.6	87.0	3.5	0.03
s420	18	16	916	8434	64.8	5K	69.4	1K	74.3	3	5	8.2	17.1	73.1	2.9	0.01
s444	3	21	888	15510	91.8	1K	13.4	5K	93.4	4	11	15.9	322.8	88.6	6.4	0.01
s526	3	21	1052	3032	56.8	1K	10.8	5K	85.5	5	14	14.9	62.5	83.8	5.6	0.01
s526n	3	21	1052	3706	59.9	1K	10.8	5K	84.5	4	9	10.3	65.4	84.8	4.3	0.01
s641	35	19	1278	1194	88.1	2K	87.1	2K	88.1	3	6	10.7	3.7	88.0	7.4	0.07
s713	35	19	1426	1250	83.9	2K	82.9	1K	83.9	4	9	14.1	4.7	83.9	9.3	0.09
s820	18	5	1640	4540	96.3	1K	39.8	28K	84.0	6	17	18.0	25.0	72.1	9.8	1.12
s832	18	5	1664	4626	95.3	1K	40.4	28K	82.3	6	17	17.7	24.5	72.0	10.0	1.10
s1196	14	18	2392	926	99.9	10K	96.8	1K	98.2	6	20	15.7	1.0	98.0	8.0	0.14
s1238	14	18	2476	988	96.4	5K	90.6	1K	93.7	6	22	15.5	0.3	93.5	8.6	0.06
s1423	17	74	2846	1538	64.3	2K	38.4	5K	89.2	6	24	10.0	22.2	77.3	5.8	0.30
s1488	8	6	2976	5680	98.6	1K	37.3	10K	97.2	7	17	8.2	9.3	96.8	4.8	0.70
s1494	8	6	2988	5258	98.1	1K	36.8	10K	96.8	7	16	8.2	20.6	93.6	4.7	0.68
s3330	40	132	6660	5958	79.5	2K	68.3	5K	78.2	8	43	6.9	628.8	77.2	4.1	15.73
s5378	35	179	10590	1602	65.0	1K	65.0	5K	69.6	4	10	2.4	1.4	72.2	1.4	0.39
s13207	62	638	26358	3018	31.6	1K	31.9	1K	37.6	4	7	0.7	5.8	37.3	0.8	0.06
Moy :					82.1		52.1		85.4			11.8		82.8	5.9	

TAB. 8.2: Taux de couverture de faute obtenus par un ATPG, un LFSR, un LFSR plus un HMMP et un LFSR plus un IMP.

de la manière suivante : une porte NAND ou NOR à n entrées est comptée comme $0.5n$ GEs ; une porte AND ou OR à n entrées vaut $0.5(n+1)$ GEs ; on compte 1.5 GEs pour un démultiplexeur et 4 GEs pour une cellule mémoire. Notons que la taille du générateur inclut tous les composants logiques des blocs *Selecting Logic* et *ML* des figures 8.6.(b) et 8.7.(b), et correspond à l'implémentation proposée à la section 8.4 sans utilisation d'outils d'optimisation logique. Finalement, la colonne *Tmp* indique le temps de calcul (en secondes) des procédures d'apprentissage sur un Pentium II à 350 Mhz.

On peut tout d'abord noter que l'utilisation des HMMPs et des IMPs améliore le taux de couverture de fautes obtenu par le LFSR seul. On observe ensuite que ce taux de couverture est souvent égal (s344, s713), parfois moins bon (s820, s832) et parfois meilleur (s526, s1423 et moyenne générale) que celui des séquences de l'ATPG. Ce résultat confirme la pertinence de notre approche. Il démontre qu'il est possible d'inférer, à partir des séquences de l'ATPG, des règles de construction de séquences qui, si elles ne permettent pas de reproduire exactement les séquences originales, permettent néanmoins d'obtenir des taux de couverture de fautes élevés lorsque le nombre de séquences générées est suffisamment grand. Ces bons résultats s'expliquent également par le fait que, pour certains circuits, l'ATPG échoue à générer des séquences ayant un bon taux de couverture de fautes dans un temps raisonnable – par exemple, le temps de calcul de l'ATPG pour le circuit s13207 a été de plusieurs jours –, par la difficulté de la tâche, ainsi que par la relative facilité avec laquelle on peut atteindre des taux de couverture de faute élevés pour certains circuits (voir les résultats obtenus par la séquence de LFSR seule).

Comparés aux HMMPs, il apparaît que, pour plusieurs circuits, les IMPs atteignent des taux de couverture aussi élevés, alors que la taille additionnelle du générateur obtenu est réduite de moitié. Néanmoins, pour certains circuits ayant une grande profondeur séquentielle (par exemple les circuits s820 et s832), les IMPs n'arrivent pas à générer des séquences permettant d'obtenir un taux de couverture acceptable, et il est alors préférable d'utiliser les HMMPs. Notons de plus que ni les HMMPs, ni les IMPs n'ont réussi à améliorer le taux de couverture de la séquence de LFSR pour le circuit s991 (résultat non reporté dans le tableau). La raison est que les fautes non détectées par le LFSR pour ce circuit (mais détectées par l'ATPG) sont des fautes de type combinatoire détectées par un grand nombre de patterns de test (un pour chaque faute) qui ne peuvent pas être facilement fusionnés. Ce circuit est cependant le seul pour lequel nous ayons rencontré une telle difficulté.

8.6 Comparaison avec d'autres approches

Comme nous l'avons mentionné à la section 8.2, les patterns d'émission définissent des distributions de probabilité très simples sur $\{\mathtt{0},\mathtt{1}\}^k$. Cette simplicité est requise par le problème du BIST du fait de la contrainte de place. On peut pourtant s'interroger sur son impact sur les performances du modèle, comparée à des distributions plus classiques comme des produits de distributions de Bernoulli. Une autre question concerne la pertinence de l'utilisation d'un algorithme d'apprentissage par fusions d'états plutôt que l'algorithme d'entraînement de Baum-Welch ou une de ses variantes. Afin de répondre à ces deux questions, un nouveau jeu de test présenté dans le tableau 8.3 a été appliqué à certains circuits *benchmark*. Pour commencer, nous avons relancé la procédure d'apprentissage des HMMPs que nous avons décrite. Pour simplifier, nous avons supprimé la phase de simulation des séquences du LFSR, et toutes les séquences de l'ATPG ont alors servi pour l'apprentissage. La taille des HMMPs et la longueur des séquences ont été choisies à la main, de la même manière que pour les expériences précédentes. Nous avons ensuite entraîné un ensemble de HMMPs à l'aide de l'algorithme d'entraînement de Viterbi – voir chapitre 2 – (colonne HMMP Vit.). La taille de ces HMMPs a été fixée à la même valeur que celle choisie à l'issue de l'algorithme par fusions d'états. Enfin, nous avons également entraîné des HMMs classiques avec des produits de distribution de Bernoulli associées aux états. Ces HMMs ont été entraînés de deux manières différentes : en utilisant un algorithme de fusion d'état avec le critère de sélection classique (colonne Bernoulli fus.), et avec l'algorithme d'entraînement de Viterbi (colonne Bernoulli Vit.). Là encore, la taille des HMMs a été fixée à la même valeur que celle choisie pour les HMMPs. Les modèles initiaux entraînés par l'algorithme d'entraînement de Viterbi sont des modèles entièrement connectés, et les patterns d'émission des HMMPs initiaux ont été fixés à $*^k$ pour éviter les vraisemblances nulles (cause d'arrêt prématuré de l'algorithme). Pour chaque circuit,

Circuit	Bernoulli Vit.		Bernoulli fus.		HMMP Vit.		HMMP fus.	
Nom	F.C.	log lik	F.C.	log lik	F.C.	log lik	F.C.	log lik
s344	97.6%	270	97.6%	287	97.0%	367	97.6%	321
s526	84.3%	905	83.7%	985	12.6%	2117	82.6%	1363
s820	93.4%	3061	95.5%	2281	57.6%	6858	94.2%	2671
s3330	80.3%	6337	80.0%	6931	78.4%	8639	78.5%	7990
Moy. :	88.9%	2643	89.2%	2621	61.4%	4495	88.2%	3086

Tab. 8.3: Taux de couverture de fautes et négation du log de la vraisemblance des séquences de l'ATPG obtenus pour des HMMPs et HMMs avec distribution de Bernoulli, entraînés par l'algorithme d'entraînement de Viterbi et un algorithme d'apprentissage par fusions d'états.

dix modèles avec différents paramétrages ont été entraînés, et à l'issu de cette phase le modèle maximisant la vraisemblance des séquences de l'ATPG a été retenu. Le tableau 8.3 inclut le taux de couverture de fautes et la négation du logarithme de la vraisemblance des séquences de l'ATPG.

On peut tout d'abord observer que, comme soupçonné, la nature non continue des probabilités d'émission des HMMPs a un impact négatif sur l'efficacité des algorithmes d'entraînement classiques, tant au niveau de la vraisemblance des séquences d'apprentissage que pour le taux de couverture de fautes des HMMPs appris. Le taux de couverture de ces modèles est généralement proche de celui obtenu par une séquence complètement aléatoire (comparer avec les résultats du LFSR du tableau 8.2). Au contraire, les trois autres approches engendrent des modèles ayant des taux de couverture de fautes élevés et proches, bien que les vraisemblances des séquences d'apprentissage diffèrent sensiblement. Ces résultats indiquent que l'apprentissage par fusions d'états est une approche pertinente dans le cadre des HMMPs (et des HMMs), et que la performance des HMMPs pour le BIST est peu affectée par la grande simplicité des distributions de probabilité d'émission, puisque le gain obtenu avec des distributions continues est négligeable (en terme de couverture de fautes).

Chapitre 9

Amélioration des séquences de l'ATPG

9.1 Introduction

Nous nous attachons dans ce chapitre à un autre problème du test de circuits, qui est celui de l'amélioration des séquences proposées par les ATPGs. La qualité d'un ensemble de séquences de test se mesure au taux de couverture de fautes de cet ensemble, mais également à la taille totale des séquences qui le composent. Comme nous l'avons mentionné au chapitre 7, le temps de test est proportionnel au nombre de vecteurs qui composent les séquences. Réduire la taille de ces séquences c'est donc réduire le temps de test, et par voie de conséquence son coût. Ce problème, commun à toutes les techniques de test, est crucial dans le cadre du test externe (qui fait appel à un testeur externe), où, contrairement au BIST, le test ne peut s'effectuer à la vitesse nominale du circuit, mais est contraint par les performances du testeur utilisé.

Il existe classiquement deux catégories de techniques pour réduire la taille des séquences de test. La première, appelée technique de *compaction dynamique* (voir par exemple [Raghunathan et Chakradhar, 1995]) s'applique directement pendant la phase de génération des séquences de test et peut être considérée comme faisant partie intégrante de l'ATPG. La deuxième, nommée *compaction statique* (voir par exemple [Niermann et al., 1992]) est utilisée en post-traitement, après la phase de génération, et consiste à construire, à partir d'un ensemble de séquences original, un nouvel ensemble de taille plus petite mais ayant néanmoins un taux de couverture de faute comparable. Expérimentalement, il apparaît que la compaction statique permet généralement de meilleurs taux de compression que la compaction dynamique. Elle est par ailleurs plus souple à utiliser car ne nécessite aucune modification des ATPGs. On distingue généralement deux types d'algorithmes de compaction statique, suivant que l'on veuille compresser une unique séquence de test ou plusieurs. Cela dépend en fait de l'ATPG utilisé. Certains ATPGs ne génèrent, pour tester un circuit, qu'une seule séquence de test. Les fautes du circuit ne sont bien évidemment pas toutes détectées en même temps à la fin de la séquence mais réparties tout au long. Au contraire, d'autres ATPGs fournissent un ensemble de séquences de test qui détectent chacune un ensemble de fautes potentielles du circuit. Les capacités de détection de ces séquences ne dépendent pas de l'état du circuit au moment où elles sont appliquées et sont donc indépendantes les unes des autres. On peut donc appliquer ces séquences dans n'importe quel ordre sans modification du taux de couverture global. Chacune de ces séquences débute par une phase d'initialisation des cellules mémoires, qui peut être effectuée à l'aide de *sets* ou *resets* si les cellules mémoires le permettent, ou à l'aide d'une séquence d'initialisation. Les algorithmes de compaction statique d'une unique séquence de test (voir par exemple [Lin et al., 2000]) consistent à sélectionner, omettre ou insérer des vecteurs de test à l'intérieur de la séquence de l'ATPG de manière à créer une nouvelle séquence de taille plus petite. Les algorithmes de compaction d'un ensemble de séquences (voir par exemple [Niermann et al., 1992]) tentent de réduire la taille totale des séquences de l'ATPG en réordonnant ces séquences et en utilisant des techniques de chevauchement.

Nous avons vu au chapitre précédent que les HMMs pouvaient être utilisés pour la construction de générateurs de séquences de test adaptés au BIST de circuits séquentiels. Nous avons observé que les HMMs inférés à partir des séquences d'un ATPG déterministe permettent d'obtenir des taux de couverture de faute souvent comparables à ces séquences et parfois même meilleurs. Nous allons voir maintenant comment utiliser les HMMs pour améliorer l'ensemble de séquences de test calculé par l'ATPG. L'objectif ici n'est plus de créer un générateur BIST – avec les contraintes de place que cela implique – mais de modifier un ensemble de séquences de manière à le rendre plus performant, c'est-à-dire en augmentant (si possible) son taux de couverture et en diminuant sa taille. Notre méthode s'inscrit donc en partie dans le cadre des méthodes de compaction statique d'un ensemble de séquences de test mais diffère fondamentalement des techniques classiques. Comme pour la construction d'un générateur BIST, la première étape consiste à apprendre, à partir des séquences fournies par un ATPG déterministe, une instance de la classe des HMMs. Le HMM appris est alors utilisé pour produire un ensemble de nouvelles séquences qui sont ensuite simulées indépendamment afin d'identifier les fautes potentielles qu'elles permettent de tester. À cet ensemble sont rajoutées les séquences originales de l'ATPG elles aussi étiquetées par les fautes qu'elles détectent. On se trouve alors en possession d'un ensemble de test dont le taux de couverture de faute est au moins aussi élevé que l'ensemble original. La dernière étape consiste à sélectionner parmi cet ensemble un ensemble de séquences minimal permettant de détecter les mêmes fautes. Ce problème s'apparente à un problème de *set cover* classique mais nécessite une heuristique adaptée à ses spécificités. La section suivante décrit le modèle, l'algorithme d'apprentissage et l'algorithme de sélection utilisés. La section 9.3 présente quelques résultats expérimentaux obtenus en utilisant notre méthode sur les circuits *benchmark* ISCAS'89.

9.2 Méthode

Nous décrivons dans cette section les différentes étapes de la méthode proposée. La section 9.2.1 présente le modèle choisi pour modéliser les séquences de l'ATPG et l'algorithme d'apprentissage utilisé. La section suivante présente une heuristique efficace pour sélectionner un ensemble de séquences minimal sans modification du taux de couverture.

9.2.1 Modèle et apprentissage

Contrairement au problème du BIST, l'objectif ici n'est pas de construire un modèle de taille aussi réduite que possible et qui puisse être facilement implémenté sur silicium. Nous proposons donc de modéliser les séquences de l'ATPG à l'aide de modèles de Markov cachés classiques en définissant les distributions de probabilités d'émission attachées aux états à l'aide de distributions de Bernoulli : on considère que les bits des vecteurs booléens sont indépendants et la probabilité d'émission d'un vecteur booléen par un état est alors le produit des probabilités d'émission de chacun de ses bits. Si on note $P_{r,s}(1)$ la probabilité de générer un `1` à l'aide de la distribution de Bernoulli de rang r associée à l'état s, et $P_{r,s}(0) = 1 - P_{r,s}(1)$ la probabilité de générer un `0` avec la même distribution, alors la probabilité de générer un vecteur booléen v à l'aide de l'état s est :

$$P(v|s) = \prod_r P_{r,s}(v_r),$$

avec v_r la valeur du bit de rang r dans v. De la même manière que pour les HMMPs, la probabilité de génération d'un pattern booléen p par un état s est la probabilité qu'en générant un vecteur booléen avec s on génère un vecteur de p. Cette probabilité est donc le produit des probabilités d'émission des bits différents de `*` dans p, et se calcule à l'aide de la formule suivante :

$$P(p|s) = \prod_{r|v_r \neq *} P_{r,s}(v_r). \tag{9.1}$$

La probabilité de génération d'un ensemble de séquences de patterns $\mathcal{O}$ suivant leurs chemins de Viterbi respectifs par un HMM H prend la même expression que dans le cadre des HMMPs et s'écrit donc à l'aide de la formule (8.3).

L'estimation des paramètres du modèle se fait en maximisant cette expression. Pour les probabilités de transition, les estimateurs sont les estimateurs classiques présentés au chapitre 2. Pour les probabilités d'émission, il s'agit d'estimer les paramètres $P_{r,s}(.)$ qui maximisent le produit $\prod_{p \in P_{\mathcal{O},s}} P(p|s)^{n_s^p}$. En intégrant l'expression (9.1) à ce produit on obtient la quantité à maximiser suivante :

$$E_s = \prod_{p \in P_{\mathcal{O},s}} \prod_{r|v_r \neq *} \left(P_{r,s}(v_r)\right)^{n_s^p},$$

qui peut se réécrire en utilisant la fonction $\delta_{\neg *}(p,r)$ introduite au chapitre 8 (rappelons qu'il s'agit de la fonction binaire qui renvoie 1 si le bit de rang r dans p est différent de `*`, et 0 sinon) :

$$E_s = \prod_{p \in P_{\mathcal{O},s}} \prod_r \left(P_{r,s}(v_r)\right)^{\delta_{\neg *}(p,r) n_s^p}.$$

Maximiser cette quantité revient à maximiser indépendamment les produits :

$$\prod_{p \in P_{\mathcal{O},s}} \left(P_{r,s}(v_r)\right)^{\delta_{\neg *}(p,r) n_s^p}. \tag{9.2}$$

On obtient alors le système d'estimateurs suivant :

$$\left.\begin{array}{l}\widehat{P}_{r,s}(1) = \frac{n_{r,s}(1)}{n_{r,s}(1)+n_{r,s}(0)} \\ \widehat{P}_{r,s}(0) = \frac{n_{r,s}(0)}{n_{r,s}(1)+n_{r,s}(0)}\end{array}\right\} \text{ si } \sum_{p\in P_{\mathcal{O},s}} \delta_{\neg *}(p,r) \neq 0,$$

avec $n_{r,s}(1)$ le nombre de patterns de $P_{\mathcal{O},s}$ dont le bit de rang r est `1`, et $n_{r,s}(0)$ le nombre de patterns de $P_{\mathcal{O},s}$ dont le bit de rang r est `0`. Si $\sum_{p\in\mathcal{O},s} \delta_{\neg *}(p,r) = 0$ (c'est-à-dire si tous les bits de rang r ont la valeur `*`), toutes les solutions maximisent le produit (9.2) et on choisira par défaut :

$$\widehat{P}_{r,s}(1) = \widehat{P}_{r,s}(0) = 1/2.$$

Ces estimateurs peuvent alors être intégrés dans un algorithme d'apprentissage. Les algorithmes d'entraînement et les algorithmes d'apprentissage par fusions d'états peuvent être employés indifféremment, et de la même manière que pour les HMMs classiques. Nous avons pour notre part utilisé l'algorithme d'apprentissage par fusions d'états utilisant le critère de vraisemblance maximale, qui a l'avantage, par rapport aux procédures d'entraînement, de proposer, en une seule exécution de l'algorithme, un éventail de HMMs de différentes tailles. On peut alors choisir expérimentalement, comme on l'a fait dans le chapitre précédent, la taille la mieux adaptée au problème.

9.2.2 Sélection d'un ensemble de séquences minimal

Une fois un HMM appris à partir des séquences de l'ATPG, il est utilisé pour générer de nouvelles séquences. Ces séquences, différentes des séquences originales, peuvent être intéressantes pour plusieurs raisons. Elles peuvent évidemment (cas le plus intéressant) détecter des fautes non détectées par les séquences de l'ATPG. Mais elles peuvent également détecter les mêmes fautes tout en étant plus courtes ; ou détecter plusieurs fautes à l'aide d'une seule séquence alors qu'il fallait plusieurs séquences de l'ATPG pour arriver au même résultat. Toutes ces séquences – et celles de l'ATPG – sont donc simulées afin d'identifier pour chacune d'elles les fautes qu'elle permet de détecter, ainsi que le rang du vecteur où la détection a lieu. Par exemple, dans une séquence de 15 vecteurs générées pas un HMM, le premier vecteur peut permettre de détecter 4 fautes, le second 3, puis aucune faute nouvelle n'est détectée jusqu'au 11$^{\text{ème}}$ vecteur qui en détecte à son tour 5 de plus et les 4 derniers vecteurs n'apportent aucune nouvelle faute et peuvent être considérés comme inutiles. La portion efficace de cette séquence est donc limitée aux 11 premiers vecteurs. Une fois toutes les séquences de l'ATPG et du HMM simulées et étiquetées par leurs fautes, il reste à sélectionner parmi cet ensemble un ensemble de séquences minimal (de taille totale minimale) permettant de détecter les mêmes fautes. Ce problème est NP-difficile – la réduction du *set cover* classique est immédiate –, et nous proposons de le résoudre à l'aide de l'heuristique gloutonne illustrée par l'algorithme 22.

Appelons $\mathcal{Z}$ l'ensemble formé au début de l'algorithme par l'union des séquences de l'ATPG et des séquences générées par le HMM. $\mathcal{Z}'$ est l'ensemble des séquences sélectionnées par l'algorithme à un instant donné ; il est réduit à l'ensemble vide au début du

Algorithme 22: ALGORITHME DE SÉLECTION D'UN ENSEMBLE DE SÉQUENCES MINIMAL

Données : $\mathcal{Z}$, un ensemble de séquences étiquetées par les fautes qu'elles détectent et la position des vecteurs où ont lieu ces détections ;

$\mathcal{Z}' \leftarrow \varnothing$;
construire $\mathcal{F}$, l'ensemble des fautes détectées par $\mathcal{Z}$;
pour chaque faute $F \in \mathcal{F}$ **faire**
    construire $\mathcal{Z}_F$, l'ensemble des préfixes de $\mathcal{Z}$ qui détectent F;
    calculer D_F;
tant que $\mathcal{F} \neq \varnothing$ **faire**
    sélectionner la faute $F \in \mathcal{F}$ pour laquelle $|\mathcal{Z}_F|$ est minimal;
    pour chaque préfixe $S' \in \mathcal{Z}_F$ **faire**
        construire $\mathcal{F}_{S'}$, l'ensemble des fautes de $\mathcal{F}$ détectées par S';
        calculer $Q(S')$;
    sélectionner le préfixe $S' \in \mathcal{Z}_F$ tel que $Q(S')$ est maximal;
    supprimer S' de $\mathcal{Z}$ (le reste de la séquence d'où S' est tiré reste dans $\mathcal{Z}$);
    si S' est une portion de séquence dont un préfixe a déjà été inséré dans $\mathcal{Z}'$ **alors**
        concaténer S' à la suite de ce préfixe dans $\mathcal{Z}'$;
    sinon insérer S' dans $\mathcal{Z}'$;
    pour chaque faute $F' \in \mathcal{F}_{S'}$ **faire**
        supprimer F' de $\mathcal{F}$;

processus. Soit $\mathcal{F}$ l'ensemble des fautes détectées par $\mathcal{Z}$ mais non encore détectées par $\mathcal{Z}'$. À chaque tour de l'algorithme, la procédure consiste à choisir une faute de $\mathcal{F}$, à sélectionner dans $\mathcal{Z}$ un préfixe de séquence permettant de détecter cette faute, à supprimer cette portion de séquence de $\mathcal{Z}$ (le reste de la séquence n'est pas supprimée et reste dans $\mathcal{Z}$) et à la rajouter à $\mathcal{Z}'$. On parle de *préfixe de séquence* car il n'est pas indispensable de sélectionner une séquence entière pour détecter une faute particulière. Si on reprend notre exemple de séquence de 15 vecteurs, on peut, à un moment donné de l'algorithme, sélectionner les 2 premiers vecteurs de cette séquence, supprimer cette portion de $\mathcal{Z}$ et l'insérer dans $\mathcal{Z}'$ afin de pouvoir tester les $4+3=7$ fautes qu'elle permet de détecter. Si, à un tour ultérieur, on sélectionne une des fautes détectées au $9^{\text{ème}}$ vecteur de la portion restante (au $11^{\text{ème}}$ vecteur de la séquence d'origine), il ne restera plus, pour que $\mathcal{Z}'$ détecte également cette faute, qu'à sélectionner ce nouveau préfixe et à le concaténer à la suite du préfixe déjà présent dans $\mathcal{Z}'$. Notons S_F le préfixe de S qui permet de détecter F, et $\overline{S_F}$ la longueur de ce préfixe. Finalement, soient $\mathcal{Z}_F$ l'ensemble des préfixes de séquences de $\mathcal{Z}$ permettant de détecter la faute F et $\mathcal{F}_S$ l'ensemble des fautes de $\mathcal{F}$ détectés par la séquence (ou le préfixe de séquence) S. À chaque tour de l'algorithme, la faute de $\mathcal{F}$ sélectionnée est la faute qui peut être détectée par le moins de séquences possibles – c'est-à-dire la faute F dont le cardinal de $\mathcal{Z}_F$ est minimal. On sélectionne ensuite un des préfixes de $\mathcal{Z}$ permettant de détecter cette faute. Ce préfixe est choisi à l'aide d'un critère basé sur une quantité, appelée *qualité* du préfixe, calculée à partir de la taille du préfixe, ainsi que du nombre et de la *difficulté* des fautes de $\mathcal{F}$ qu'il permet de détecter.

La difficulté D_F d'une faute F est inversement proportionnelle au nombre de séquences de $\mathcal{Z}$ qui permettent de la détecter et est définie par :

$$D_F = \frac{1}{|\mathcal{Z}_F|}.$$

On définit alors la qualité $Q_{S'}$ d'un préfixe S' par :

$$Q(S') = \frac{\sum_{F' \in \mathcal{F}_{S'}} D_{F'}}{\overline{S'}}.$$

La qualité d'un préfixe est donc proportionnelle à la difficulté des fautes que ce préfixe permet de détecter, et inversement proportionnelle à sa taille.

9.3 Résultats expérimentaux

Nous avons testé notre méthode sur quelques-uns des circuits benchmarks ISCAS'89. Les résultats obtenus sont reportés dans le tableau 9.1. Après le nom du circuit sont indiquées les performances des séquences de l'ATPG. La colonne %FC indique le taux de couverture de fautes de cet ensemble, et la colonne L_A la taille totale de ces séquences. La colonne $\tilde{L}_A$ indique la taille totale de l'ensemble de séquences obtenu par application de notre heuristique de sélection sur les séquences de l'ATPG seules – sans y ajouter d'autres séquences. Dans les colonnes suivantes sont reportés les résultats obtenus en ajoutant aux séquences de l'ATPG les séquences générées à l'aide du HMM appris. La colonne #S indique le nombre d'état du HMM utilisé pour générer ces séquences, et la colonne L_H la taille totale de ces séquences. Ces valeurs ont été choisies manuellement de manière à maximiser le taux de couverture obtenu par ces séquences. La colonne %FC indique le taux de couverture de fautes obtenu à l'aide de toutes les séquences – celles de l'ATPG plus celles du HMM. Dans la colonne $\tilde{L}_{AH}$ sont reportés la taille totale de l'ensemble de séquences obtenu par l'heuristique de sélection appliquée sur les séquences de l'ATPG augmentées des séquences du HMM. Les trois dernières colonnes comparent les différents taux de compression obtenus : $\tilde{L}_A$ *versus* L_A, $\tilde{L}_{AH}$ *versus* L_A, et $\tilde{L}_{AH}$ *versus* $\tilde{L}_A$. Ces taux de compression sont calculés de manière classique à l'aide de la formule (taille originale − taille compressée)/(taille originale).

Remarquons tout d'abord les résultats obtenus par notre heuristique de sélection appliquée aux séquences de l'ATPG seules. Cette heuristique suffit la plupart du temps à réduire de manière très significative (près de 40% en moyenne) la taille totale des séquences de l'ATPG. Ce résultat surprenant s'explique par le processus utilisé par les ATPGs pour construire un ensemble de séquences de test. Ces outils utilisent généralement une procédure gloutonne qui consiste, à chaque tour de l'algorithme, à choisir une faute non encore détectée, à construire – ou essayer de construire – une séquence de test pour cette faute, et à réitérer la procédure en choisissant une nouvelle faute. Mais à aucun moment ils ne remettent en cause les séquences qui ont déjà été créées. Il peut donc arriver qu'une séquence, construite pour tester une faute particulière, détecte également une faute précédemment adressée et pour laquelle il existe déjà une séquence spécifique. Cette

Circuit	ATPG			HMM				%Compression		
	%FC	L_A	$\tilde{L}_A$	#S	L_H	%FC	$\tilde{L}_{AH}$	$\tilde{L}_A/L_A$	$\tilde{L}_{AH}/L_A$	$\tilde{L}_{AH}/\tilde{L}_A$
s344	97.6%	179	116	20	20000	97.6%	40	35.2%	77.7%	65.5%
s386	90.2%	363	218	10	20000	90.2%	118	39.9%	67.5%	45.9%
s444	91.8%	7216	2747	30	20000	94.8%	1259	61.9%	82.6%	54.2%
s526	56.8%	1362	1045	20	20000	84.1%	1324	23.3%	2.8%	-26.7%
s526n	59.9%	1717	1054	20	20000	86.1%	1189	38.6%	30.8%	-12.8%
s641	88.1%	571	454	30	20000	88.1%	117	20.5%	79.5%	74.2%
s713	83.9%	594	471	20	10000	83.9%	112	20.7%	81.1%	76.2%
s820	96.3%	2256	1172	30	20000	96.3%	624	48.0%	72.3%	46.8%
s832	95.3%	2298	1225	30	20000	95.3%	593	46.7%	74.2%	51.6%
s991	99.2%	749	417	20	10000	99.2%	57	44.3%	92.4%	86.3%
s1196	99.9%	453	333	30	20000	99.9%	219	26.5%	51.7%	34.2%
s1423	64.3%	738	429	30	20000	87.7%	623	41.9%	15.6%	-45.2%
s1488	98.6%	2824	1741	30	20000	98.6%	614	38.3%	78.3%	64.7%
s5378	65.0%	797	507	20	1000	68.4%	222	36.4%	72.1%	56.2%
Moyenne	84.8%					90.7%		37.3%	62.7%	40.8%

TAB. 9.1: Taux de couverture de fautes et taux de compression obtenus pour les séquences de l'ATPG, les séquences de l'ATPG compressées en utilisant simplement l'heuristique de sélection, et les séquences de l'ATPG plus les séquences générées à l'aide du HMM.

dernière, si elle ne détecte pas d'autres fautes intéressantes (c'est-à-dire non détectées par d'autres séquences), est alors inutile et peut être éliminée.

On remarque ensuite que les ensembles de séquences construits à l'aide de notre méthode permettent généralement d'augmenter le taux de couverture de fautes des séquences originales (+6% en moyenne). Du point de vue du taux de compression, on obtient également des résultats intéressants. Le taux de compression moyen des séquences de l'ATPG originales est d'à peu près 63%, et si on compare nos résultats aux séquences de l'ATPG compressées (obtenus en appliquant la procédure de sélection sur les séquences originales), on obtient un taux de compression d'à peu près 41%. Remarquons de plus que ces taux de compression moyens sous estiment les résultats réels. Parmi les *benchmarks* testés, trois circuits obtiennent en effet des taux de compression négatifs (le s526, le s526n, et le s1423) qui s'expliquent aisément lorsque l'on observe les taux de couverture obtenus : les séquences de test construites améliorent en effet le taux de couverture de fautes de plus de 20% pour chacun de ces circuits. Si on exclut ces trois circuits du calcul de la moyenne des taux de compression, on obtient alors respectivement 75.4% de compression par rapport aux séquences originales, et 59.6% de compression par rapport aux séquences originales compressées.

9.4 Conclusion

Nous avons proposé une méthode pour améliorer le taux de couverture et réduire la taille d'un ensemble de séquences produit par un ATPG déterministe. Notre méthode utilise ces séquences pour apprendre une instance des HMMs qui est utilisée à son tour pour générer un nouvel ensemble de séquences de test. Une heuristique de sélection est alors utilisée pour choisir parmi cet ensemble (augmenté des séquences de l'ATPG originales) un ensemble de séquences de taille minimale permettant de détecter les mêmes fautes. Les résultats préliminaires montrent que cette méthode est intéressante, aussi bien pour les taux de couverture de fautes que pour les taux de compression obtenus. Néanmoins plusieurs points mériteraient d'être étudiés plus profondément. Par exemple, l'algorithme de sélection des séquences utilisé est une heuristique et est donc certainement sujet à amélioration. Mais le point d'interrogation le plus intéressant reste l'impact de la taille du HMM sur la qualité et l'intérêt des séquences produites. Lorsqu'on génère des séquences de test à l'aide d'un processus stochastique, la grande majorité des séquences produites sont inintéressantes du point de vue des fautes qu'elles détectent – elles ne détectent que des fautes faciles, c'est-à-dire des fautes détectées par beaucoup de séquences. Les séquences de l'ATPG, au contraire, détectent en majorité des fautes difficiles (et incidemment les fautes faciles). Or plus la taille du HMM est importante, plus les séquences qu'il produit sont proches des séquences de l'ATPG, et donc plus ces séquences sont individuellement intéressantes du point de vue des fautes détectées mais inintéressantes comparées à celles de l'ATPG. On se retrouve alors face à un problème d'*overfitting* similaire à celui que l'on rencontre en reconnaissance. Le problème ici est de s'éloigner des séquences d'apprentissage, non pour être capable d'en reconnaître de nouvelles, mais pour pouvoir en générer certaines qui apportent réellement quelque chose.

Conclusion

Nous nous sommes intéressés dans cette thèse à la modélisation stochastique de séquences d'observations discrètes à l'aide de modèles de Markov cachés. Dans la première partie de ce document, nous avons présenté ces modèles en nous appuyant sur les applications classiques qu'ils sont susceptibles de traiter : les problèmes de discrimination et de segmentation. Nous nous sommes ensuite plus particulièrement intéressés au problème de l'apprentissage des HMMs dans le cadre d'applications pour lesquelles aucune connaissance ne permet la construction *a priori* d'une structure. En reprenant l'algorithme d'apprentissage par fusions d'états introduit par Stolcke et Omohundro [Stolcke et Omohundro, 1994a], nous avons proposé une optimisation de la procédure de sélection de vraisemblance maximale, c'est-à-dire la procédure qui consiste à sélectionner, à chaque tour de l'algorithme, le couple d'états dont la fusion entraîne une baisse minimale de la vraisemblance des séquences d'apprentissage. Nous avons ensuite expliqué pourquoi cette procédure est loin d'être optimale, et dans quels cas elle peut facilement être prise en défaut. Nous avons alors proposé, en nous inspirant des algorithmes d'inférence de type ALERGIA [Carrasco et Oncina, 1994] utilisés pour les automates d'états finis probabilistes, un nouveau critère de sélection des états à fusionner. Ce critère ne se base plus, comme c'est le cas pour la procédure classique, uniquement sur les distributions de probabilités directement attachées aux états, mais sur l'ensemble des états qui peuvent être atteints. Nous avons intégré ce critère dans un nouvel algorithme d'apprentissage de HMMs efficace. Pour finir, nous avons comparé notre nouveau critère, le critère de vraisemblance maximale et l'algorithme d'entraînement de Viterbi sur trois domaines d'application variés. Nous avons pu observer que notre critère permet d'obtenir des résultats souvent aussi bons que les deux autres approches, et en un temps de calcul nettement moins élevé. D'un point de vue plus général, il apparaît que, dans le cadre de l'apprentissage de HMMs sans structure connue, les algorithmes de généralisation par fusions d'états constituent une alternative intéressante aux algorithmes d'entraînement classiques de type Baum-Welch.

Dans la deuxième partie de ce document, nous avons appliqué les HMMs à la modélisation de séquences servant au test de circuits microélectroniques séquentiels. Les applications ont porté sur deux types de tâches originales. Comparées aux applications classiques, ces tâches ont plusieurs particularités. Premièrement, chaque séquence d'apprentissage définit en fait un ensemble de séquences. Ensuite, l'objet est de faire de la génération de séquences, et non de la reconnaissance ou de la segmentation comme c'est habituellement le cas. La première tâche à laquelle nous nous sommes intéressés est la création d'un générateur de séquences pour le BIST. Pour cela, nous nous sommes inspirés des unigrams et des HMMs pour créer deux nouvelles classes, les IMPs et les HMMPs, dont la principale caractéristique est d'utiliser des distributions de probabilités extrêmement simples pour définir les probabilités d'émission associées aux états. Nous avons alors adapté l'algorithme d'apprentissage par fusions d'états utilisant le critère de sélection classique aux HMMPs, et avons proposé un algorithme similaire pour l'apprentissage des IMPs. HMMPs et IMPs sont inférés à partir d'un ensemble de séquences de test proposées par un ATPG déterministe. Grâce à la grande simplicité des distributions de probabilités d'émission, nous avons alors pu proposer une implémentation efficace (c'est-à-dire de taille réduite) des modèles obtenus en générateurs de séquences de test. Les résultats expérimentaux obtenus sur les *benchmarks* classiques de la communauté microélectronique montrent que les séquences générées, sans être exactement similaires aux séquences d'apprentissage, ont les mêmes pouvoirs de détection de fautes que celles-ci. Nous avons ensuite appliqué les HMMs classiques au problème de l'amélioration des séquences de l'ATPG. Là encore, les résultats se sont révélés très concluants, aussi bien en ce qui concerne les taux de compression obtenus que les taux de couverture de fautes atteints. L'ensemble de ces résultats a montré les potentialités et l'intérêt des techniques de modélisation de séquences dans le cadre du test de circuits microélectroniques. Nul doute qu'il en existe encore bien d'autres.

Les perspectives de ce travail sont de plusieurs ordres. Concernant le nouveau critère de sélection et l'algorithme d'apprentissage par fusions d'états qui lui est adapté, diverses améliorations sont à étudier. Nous avons utilisé la technique de Laplace *smoothing* pour lisser les distributions de probabilités d'émission et de transition, mais d'autres méthodes de lissage pourraient se révéler plus intéressantes. On pourrait par exemple utiliser une fonction de lissage construite à l'aide d'un unigram comme dans [Thollard et Dupont, 2000] ou [Dupont et Amengual, 2000]. Une autre solution serait d'utiliser des techniques plus perfectionnées comme la méthode de *shrinkage* décrite dans [Freitag et McCallum, 1999]. Concernant les mauvais résultats obtenus par notre algorithme sur les séquences construites sur un très large alphabet (données textuelles de FlyBase), il serait intéressant d'étudier l'impact d'un *clustering* préalable des symboles de l'alphabet sur le processus d'apprentissage comme cela a été fait dans le cadre des automates d'états finis probabilistes dans [Dupont et Chase, 1998].

D'autres améliorations concernent les méthodes d'apprentissage par fusions d'états dans leur ensemble. Il serait par exemple très intéressant d'étudier comment intégrer à ce processus diverses connaissances *a priori* sur la forme de la structure souhaitée. En effet, même si cette forme n'est pas parfaitement connue, certaines connaissances peuvent en revanche aider à sa construction. Ces connaissances peuvent être d'ordre général – l'exemple classique est la structure de type gauche-droite –, mais peuvent également prendre une forme très locale. Dans les séquences ADN, par exemple, certaines portions de séquences ont, de part leur fonction spéciale, une forme particulière bien connue (la *TATA box* des séquences promotrices, par exemple). Intégrer cette connaissance au processus d'apprentissage pourrait très certainement améliorer la qualité du modèle obtenu. Les procédures d'apprentissage par fusions d'états nous semblent particulièrement bien adaptées à l'intégration de ce genre de connaissances très localisées. Exprimées sous forme de contraintes (par exemple sur les transitions autorisées ou sur le typage des états), celles-ci pourraient alors guider l'algorithme, lui laissant inférer la structure en se conformant simplement aux contraintes fixées.

Annexes

Annexe A : Chorales de Bach [Blake et Merz, 1998]

5 chorales de Bach (transposées en gamme de do majeur) après réduction à un alphabet de 12 symboles (un octave). La premières séquence, par exemple, débute par : `do-do-sol-mi-ré-do-do-` ...

Annexe B : Séquences issues de FlyBase [The FlyBase Consortium, 1999]

8 séquences positives (traitant d'une interaction) et 7 séquences négatives. Les noms de gènes apparaissent entourés d'un cadre, tandis que les signes de ponctuation sont soulignés. On remarquera, par exemple, le gène `for` et le "mot" `[s]`, présents dans la troisième séquence positive.

1. Mutations in zygotic homeotic gene Ubx do not interact with RpII140
2. Wild type l(1)2Bc function can compensate for defective br function
3. One allele of Csr interacts with for [s] homozygous larvae to produce rover behaving larvae
4. Activation depends on cooperative interactions between hb and bcd
5. Results suggest that tra2 is a positive regulator of dsx pre-mRNA processing
6. In some cases the negative autoregulation stabilizes Ubx levels while in others it modulates the spatial and temporal patterns of Ubx expression
7. The anterior boundaries of h stripes 5 and 6 are set by Kr
8. wg acts through inactivation of the sgg protein kinase to specify ventral cell fate in the leg

1. 2 distinct DNA ligases are present in embryos , DNA-ligI and DNA-ligII
2. 128up is transcribed in the same direction as RpIII128 and they are separated by a short intergenic region
3. Early expression of Hnf4 at the terminal regions of the syncytial blastoderm may indicate a role of Hnf4 in midgut determination
4. The proximal element alone directs twi expression in the ventralmost 12 to 14 nuclei while the complete promoter with both proximal and distal elements directs twi expression in the ventralmost 18 to 20 cells
5. A segment of ple from -1kb to +7kb shows limited rescue of ple mutants , FBrf0058913
6. Giant fibre growth studies in cells where l(1)sc has been ablated suggest l(1)sc is normally required to repel the growth cone of the fibre
7. Transgenic flies expressing both Rh1 (ninaE) and Rh2 in photoreceptors R1-6 demonstrate a response that indicates that photoreceptors trigger receptor potentials tuned to combined spectral response of both rhodopsins

Annexe C : Séquences de test microélectroniques

12 séquences tirées des 284 séquences de test proposées par l'ATPG *Sunrise* pour le circuit s820.

Séquence 1

```
*******************0
*010************0001
****0*****1****11001
***************01001
****1**********01001
*1**1**********01001
*010************0001
*0*1**********010001
****0*1001*****11001
****0*********011001
****0*****00***11001
****0***********0001
*****0***********001
*0*0*1***********001
```

Séquence 2

```
*******************0
*010************0001
****0*****1****11001
***************01001
****1**********01001
*1**1**********01001
*010************0001
*0*1**********010001
****0*1001*****11001
****0*********011001
01**0*****010**11001
****0********0*11001
0***0***********0001
```

Séquence 3

```
*******************0
*010************0001
****0*****1****11001
***************01001
****1**********01001
*1**1**********01001
*010************0001
*0*1**********010001
****0*1001*****11001
****0*********011001
****0*****00***11001
****0***********0001
*****0***********001
*000*************001
```

Séquence 4

```
*******************0
*010************0001
****0*****1****11001
***************01001
****1**********01001
*1**1**********01001
*010************0001
*0*1**********010001
****0*1001*****11001
****0*********011001
****0*****00***11001
****0***********0001
*****0***********001
*001*************001
```

Séquence 5

```
*******************0
*010************0001
****0*****1****11001
***************01001
****1**********01001
*1**1**********01001
*010************0001
*0*1**********010001
****0*1001*****11001
****0*********011001
****0*****00***11001
****0***********0001
*****0***********001
*010*************001
```

Séquence 6

```
*******************0
*010************0001
****0*****1****11001
***************01001
****1**********01001
*1**1**********01001
*010************0001
*0*1**********010001
****0*1001*****11001
****0*********011001
****0*****00***11001
****0***********0001
*****0***********001
*100*************001
0***************0001
```

Séquence 7

```
*******************0
*010************0001
****0*****1****11001
***************01001
****1**********01001
*1**1**********01001
*010************0001
*0*1**********010001
****0*1001*****11001
****0*********011001
****0*****00***01001
```

Séquence 8

```
*******************0
*010************0001
****0**********11001
***************01001
****1**********01001
*1**1**********01001
*010************0001
*0*1**********010001
****0*1001*****11001
****0*********011001
****0***1*00***11001
```

Séquence 9

```
*******************0
*010************0001
****0*****1**0*11001
***************01001
****1**********01001
*1**1**********01001
*010************0001
*0*1**********010001
****0*1001*****11001
****0*********011001
****0***1*00*1*11001
```

Séquence 10

```
*******************0
*010************0001
****0*****1****11001
***************01001
****1**********01001
*1**1**********01001
*010************0001
*0*1**********010001
****0*1001*****11001
****0*********011001
****0*****1*1**11001
1***1****0******1001
```

Séquence 11

```
*******************0
*010************0001
****0*****1****11001
***************01001
****1**********01001
*1**1**********01001
*010************0001
*0*1**********010001
****0*1001*****11001
****0*********011001
01**0*****010**11001
****0********1*01001
```

Séquence 12

```
*******************0
*010************0001
****0*****1****11001
*0*************01001
****1**********01001
*1**1**********01001
*010************0001
*0*1**********010001
****0*1001*****11001
****0*********011001
****0*****1*1**11001
11**************1001
```

Annexe D : [Bréhélin et al., 2001]

Nous reproduisons ici la référence [Bréhélin et al., 2001], qui constitue une synthèse du problème de l'apprentissage des HMMPs. Cet article a l'avantage de présenter, en quelques pages, plusieurs concepts et apports qui sont, dans ce manuscrit, répartis sur plusieurs chapitres. Nous pensons que sa lecture peut apporter un autre angle de vue et faciliter la compréhension.

Hidden Markov Models with Patterns to Learn Boolean Vector Sequences and Application to the Built-in Self-Test for Integrated Circuits

Laurent Bréhélin Olivier Gascuel Gilles Caraux

Département Informatique Fondamentale et Applications
Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier
Université Montpellier II, 161 rue Ada, 34392 Montpellier Cedex 5, France

`http://www.lirmm.fr/~gascuel/MAAS/US-MAAS.html`

Email: `{brehelin,gascuel,caraux}@lirmm.fr`

Abstract

We present a new model, derived from the Hidden Markov Model (HMM), to learn Boolean vector sequences. Our *Hidden Markov Model with Patterns* (*HMMP*) is a simple, hybrid and interpretable model that uses Boolean patterns to define emission probability distributions attached to states. Vectors consistent with a given pattern are equiprobable, while inconsistent ones have probability zero to be emitted. We define an efficient learning algorithm for this model, which relies on the maximum likelihood principle, and proceeds by iteratively simplifying the structure and updating the parameters of an initial specific HMMP that represents the learning sequences. Each simplification involves merging two states of the current HMMP, while keeping the likelihood as high as possible, and the algorithm stops when the HMMP has a sufficiently small structure. HMMPs and our learning algorithm are applied to the *Built-in Self-Test* (*BIST*) for integrated circuits, which is one of the key microelectronic problems. An HMMP is learned from a test sequence set (computed using a specific tool) that covers most of the potential faults of the circuit at hand. Then this HMMP is used as test sequence generator. Our experiments, carried out with classical microelectronic benchmark circuits, show that learned HMMPs have a very high fault coverage. Furthermore, their small sizes combined with their simplicity allow these models to be easily implemented on the circuits for self-testing purposes.

Index terms: Boolean Vector Sequence Modeling; Hidden Markov Models; Hybrid Approach; Structure (and parameters) Learning; Built-in Self-Test for Integrated Circuits.

1 Introduction

The Hidden Markov Model (or HMM) was introduced by Baum and colleagues in the late 1960s [1]. This model is closely related to probabilistic automata (PAs) [2]. A probabilistic automaton is defined by its *structure*, made up of *states* and *transitions*, and by probability distributions over the transitions. Moreover, each transition is associated with a symbol from a finite alphabet that is generated each time the transition is run over. An HMM is also defined by its structure, composed of states and transitions, and by probability distributions over the transitions. For most authors, symbol generation is attached to the states, but it can be also attached to the transitions [3]. Each state (or transition) is then associated with a probability distribution over the alphabet that expresses the probability for each symbol to be generated when the state (or transition) is encountered. These models are said to be *hidden* because the series of states (or transitions) are not apparent in the generated sequences.

When the structure is known, the HMM learning (or training) problem is reduced to estimating the value of its parameters (transition and symbol generation probabilities) from a sample of sequences. The *Baum-Welch* algorithm [4] is a well-known approach, which complies with the maximum likelihood principle and is a special case of the Expectation-Maximization (EM) algorithm [5]. This is an iterative re-estimation algorithm that ensures convergence to a local optimum. Abe and Warmuth [6] studied the training problem from a Computational Learning Theory perspective. They proved that the PA class is not polynomially trainable unless *RP*=*NP* while, to the best of our knowledge, the question remains open for HMMs. However, we can reasonably assume that the problem is not easier and that heuristics have to be used.

When it is not possible to infer the HMM structure from the *a priori* knowledge we have about the problem under investigation, the HMM learning problem becomes even more difficult. We have to estimate the parameters of the structure, and also infer this structure from the training sample. A solution is to train a fully connected HMM with the Baum-Welch algorithm (or one of its variants), leaving this one to determine the probability zero transitions. Various authors have proposed an alternative approach, derived from automata theory [7], which involves generalizing an initial specific automaton that represents the learning sample, by iteratively merging "similar" states until a "convenient" (*e.g.*, sufficiently general or small) structure is obtained. This principle has been applied to Deterministic Finite Automata (DFA) [8], Stochastic Finite Automata (SFA) [9], and HMMs [10]. In the recent Abbadingo One DFA Learning Competition [11], this principle appeared to be the best strategy for learning DFA.

HMMs are intensively used in speech recognition (*e.g.*, [3], [12]) and handwriting recognition (*e.g.*, [13], [14]). In these applications, the usual approach involves using an HMM for each word or character to be recognized. Typically, HMMs have a pre-determined left-to-right structure of fixed size, and they are trained using the Baum-Welch method. A different approach is proposed for speech recognition in [10]: a state merging algorithm learns both the HMM structure and parameters; the structure is not *a priori* determined, but the usual constraints of the application are used, *e.g.* the left-to-right topology. HMMs are successfully used in many other recognition applications, such as computational biology [15] or texture classification [16], and also for various segmentation tasks (*e.g.* [17]). In most of these applications, the learning scheme is similar to that employed for speech and handwriting recognition, *i.e.* a special structure is pre-determined and then trained with the Baum-Welch algorithm or one of its variants.

In this article, we present a new application of HMMs: the *Built-in Self-Test* (*BIST*) for integrated circuits. Due to the increasing complexity of integrated circuits, in recent years this test technique has become one of the most important research areas in the microelectronic community [18]. Our approach involves learning an HMM from a set of *test sequences*. These Boolean vector sequences (obtained using a specific tool) are able to detect most potential faults in the circuit at hand. Then the learned HMM is used to generate sequences similar to the original test (learning) sequences, and having good fault detection capabilities. Moreover, since this generator is intended to be electronically implemented, the aim is to obtain an HMM of low size, which may be seen as a compression of the learning sequences. BIST differs from classical HMM applications for several reasons. First we use HMMs for generation purposes, which involves specificities regarding recognition tasks. Secondly, size constraints require appropriate solutions for the model to be efficiently implemented on circuits. This point is particularly important concerning the choice of emission probability distributions attached to states, and led us to develop a new class of HMM, which we called Hidden Markov Model with Patterns (HMMP). Finally, there is no *a priori* knowledge to infer the HMM structure, which is learned from the original test sequences using a state merging algorithm.

The organization of this paper is as follows. In Section 2, we rapidly present the problem of testing integrated circuits (see also [18]); we indicate the main features of the manipulated data and explain how this problem can naturally be dealt with using HMMs. In Section 3, we define HMMPs and present the method for estimating its parameters using the Maximum Likelihood principle. In Section 4, we detail our HMMP learning algorithm. Section 5 is devoted to experimental results. We first present the performance of our method with the classical benchmarks of the test community, then compare the results with those of alternative models and learning algorithms. In Section 5.2, a brief discussion wraps up the paper.

2 Testing integrated circuits

An integrated circuit is composed of three parts:

- inputs;
- outputs;
- the body of the circuit, which is made up of interconnected *logic gates* and *memory cells*.

The manipulated values are Boolean (0 or 1). The logic gates are operators (or functions) such as AND, OR, NOT, NAND and NOR. Memory cells have one input, one output and store a Boolean value. An integrated circuit is *sequential* if it contains memory cells, and *combinational* if it is memoryless. Note that we can apply values to circuit inputs and read its output values, but it is not possible to access its internal elements.

Integrated circuits may be affected by different failures. We usually consider that the logic conception of the circuit is correct, but that an external factor has physically altered one of its components. Such failures are named *faults*. Since the logic structure of the circuit is known, we can infer its set of potential faults. Testing a fault of a circuit involves checking that this fault is not present. Testing a circuit involves testing all (or most) of its potential faults. The *fault coverage* of a test is the ratio of the number of tested faults over the total number of potential faults.

A *test vector* is a set of Boolean values simultaneously applied to the inputs. For a circuit with k inputs, a test vector has length k. Testing a potential fault of a combinational circuit can be achieved using a single test vector. In this article, we deal with testing sequential circuits (general and common case), which requires sequences of test vectors in order to initialize the memory cells. Then the test of a potential circuit fault is achieved by applying such a sequence and comparing the obtained output values with those logically expected. If these values are equal, we conclude that the fault is absent. All inputs do not have to be specified to test a given fault, and only some inputs are determinant. So *Boolean patterns* are used rather than test vectors. A Boolean pattern is a vector defined over $\{1, 0, *\}^k$. A bit is set at * if its value does not influence the result, *i.e.* if the fault occurs, it is detected regardless of the value of this bit. A Boolean pattern defines a set of vectors; a pattern with n * bits represents a set of 2^n vectors. In the same way, a sequence of patterns defines a set of vector sequences. We say that a pattern sequence *tests* a fault if every vector sequence it defines detects this fault. Such sequences are named *test sequences*. In a test sequence, the order of patterns is as important as the patterns themselves. However, one sequence usually tests several faults, and one fault may be tested by several sequences. Moreover, the sequence length varies depending on the tested fault. Note too that we can easily deduce, by simulation, all potential faults detected by a given test sequence and hence compute the fault coverage of any sequence set. This is used in our experiments (Section 5) to evaluate the performance (fault-coverage) of learned HMMPs and to calibrate their size.

The search of a test sequence for a given circuit and a given fault is a hard task. For combinational circuits, the problem of existence of a test pattern is *NP-Complete* [19]. For sequential circuits, the task is even more difficult. Nevertheless, *Automatic Test Pattern Generators* (*ATPGs*) are available for such purposes. They try to circumvent the difficulty by using stochastic algorithms (*e.g.*, genetic [20]), or heuristics based on topological analysis [21] or fault simulation [22].

A set of test sequences is shown in Annex. It was produced by the ATPG of *Sunrise*[1], for the classical benchmark[2] circuit *s820*. We note two essential features of test sequences: their patterns are hollow (they have few fixed bits), and many patterns appear several times (*e.g.*, `*010************0001`).

When test sequences with sufficient fault coverage have been produced (which is solved by ATPGs with various degrees of success depending on the circuits), we have to position the test procedure. The classical approach involves using an external tester. Due to the price of these testers and sometimes the fact that there is no physical access to the tested circuit, the *Built-in Self-Test* or *BIST* method is often preferred. The BIST principle is to incorporate a supplementary test structure into the circuit. This structure should be able to generate test sequences and analyse circuit responses. Response analysis is a task that has efficient solutions [18]. This is not the case for test sequence generation. The problem is to find a generator of sequences that combines high fault coverage and small size (in terms of silicon area). We cannot physically stock all ATPG sequences because of the silicon cost. Otherwise, we can build small generators of pseudo-random sequences, but they have insufficient fault coverage.

Given the fact that different sequences can test the same fault and hence that ATPG sequences are not the only possible sequences, our approach relies on statistical modeling techniques. This involves learning from the ATPG sequences an instance of a new class of HMM (called HMMP) that generates ATPG sequences or similar sequences with sufficiently high probability. We shall see (Section 5) that, at the

[1] ©Sunrise test system. A Viewlogic Company.

[2] These benchmarks were created for the *International Symposium on Circuits & Systems* (*ISCAS*) of 1989. They are available at http://www.cbl.ncsu.edu/benchmarks/

price of larger but reasonable test sequence sets, relatively small HMMPs effectively generate sequences with fault coverage as high as that of the ATPG.

3 A new class of HMMs: HMMPs

The HMMs we want to infer are intended to generate Boolean vector sequences. There are numerous solutions for modelizing such sequences. The most natural is to consider that bits of Boolean vectors are independent and hence to use products of Bernoulli distributions to define the emission probabilities. Unfortunately, due to the great number of Bernoulli distributions involved – one for each bit of each state –, this approach raises size problems when implementing the obtained models as test sequence generators. This becomes particularly important when the vector length (number of inputs of the circuit) is high.To deal with this difficulty, we introduce very simple distributions based on discrete probabilities with low implementation cost. Using these distributions in the HMM framework leads to the *Hidden Markov Model with patterns* or *HMMP*. Section 3.1 defines both of these distributions and the HMMP. Sections 3.2 and 3.3 provide the likelihood of a set of ATPG sequences for a given HMMP and the way to compute this likelihood, while Section 3.4 presents the maximum likelihood estimators of HMMP parameters.

3.1 Definitions

To define the HMMP emission probabilities, we use products of Bernoulli distributions whose parameters cannot handle all possible values between 0 and 1 but are constrained to 0, 1 or 1/2. The advantage of such distributions for the BIST is that these probabilities are easily implemented: probability 1/2 is free, 1 and 0 have very low cost in terms of silicon area [23], while implementing continuous probabilities is much more expensive. A distribution defined in this way can be represented by a Boolean pattern similar to those introduced in the previous section: bits with parameter 0 are set at `0`, those with parameter 1 at `1`, and those with 1/2 at `*`. Such patterns will be called *emission patterns* in the following to avoid confusion with the test patterns of ATPG sequences. In distributions defined by emission patterns, the vectors consistent with the pattern are equiprobable, while the others have probability zero. For example, in the distribution `*1*`, vectors `010`, `011`, `110` and `111` have probability 1/4, while the four other vectors have probability zero. Such extreme (zero) probabilities are usually avoided in recognition tasks, so as to be able to recognize new unseen sequences. Our aim is different and involves generating sequences as close as possible to ATPG sequences, not new sequences with unseen vectors.

An HMMP H is defined by a triplet $\langle S, P, M\rangle$, where

- S is a finite set of states; S contains two special states *start* and *end* which are used to initiate and conclude a sequence, respectively. Each state of S, except *start* and *end*, is labeled by an emission pattern from P.
- $P = \{p_s, s \in S - \{start, end\}\}$ is the set of emission patterns associated with the states; p_s is the emission pattern associated with state s.
- $M : S - \{end\} \times S - \{start\} \rightarrow [0, 1]$ is the matrix that contains transition probabilities between states. We have: $\forall s, t,\ M(s \rightarrow t) \geq 0$, and $\forall s \neq end,\ \sum_{t \in S} M(s \rightarrow t) = 1$.

The *structure* of an HMMP is the set of its states and of its non-zero transitions. Figure 1 gives an example of HMMP with seven states and nine transitions.

3.2 Likelihood of a set of test sequences

We say that two patterns p and p' are *compatible* if the sets of vectors they represent have a non-empty intersection. This is expressed by the fact that their fixed bits (those with value `0` or `1`) do not differ. For example, $p =$ `11*` and $p' =$ `1*0` are compatible, while $p =$ `11*` and $p' =$ `100` are not compatible. Let p be a test pattern (a set of vectors) and s a state of an HMMP; the *emission probability of* p *by* s is the probability that in generating a vector with distribution p_s we generate a vector of p. If p and p_s are not compatible, the emission probability of p by s is zero. For a compatible pattern, the emission probability

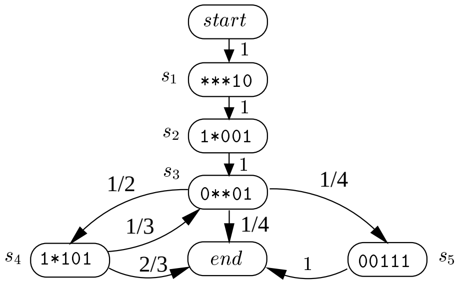


Figure 1: Example of HMMP: each state is labelled with its name and associated emission pattern; transitions are labeled with their transition probability.

depends on the number of bits which are fixed in p but have the value `*` in p_s. Let $*^p_{p_s}$ denote this number. For example, if $p_s = \texttt{1***}$ and $p = \texttt{10**}$ then $*^p_{p_s} = 1$. Since `*` has equiprobability of generating a `0` or a `1`, the probability of emitting the pattern p on the state s is given by

$$L(s,p) = \left(\frac{1}{2}\right)^{*^p_{p_s}} \tag{1}$$

when p is compatible with p_s, and 0 otherwise. For example, the probability of emitting `1*01*` with $p_s = \texttt{1*010}$ is 1, while the probability of emitting `1*010` with $p_s = \texttt{***10}$ is $\frac{1}{4}$.

The *generation probability* $P(x|H)$ of a test pattern sequence x by an HMMP H is the probability of generating with H a Boolean vector sequence that belongs to the test sequence set defined by x. Let x be equal to $p_1 p_2 \ldots p_l$, with $p_i \in \{\texttt{0}, \texttt{1}, *\}^k$, we have:

$$P(x|H) = \sum_{\langle s_0, \cdots, s_{l+1}\rangle \in S^{l+2}} \left[\left(\prod_{i=0}^{l-1} M(s_i \to s_{i+1}) L(s_{i+1}, p_{i+1}) \right) M(s_l \to s_{l+1}) \right],$$

with S^{l+2} being the set of sequences with $l+2$ states from S, such that $s_0 = \textit{start}$ and $s_{l+1} = \textit{end}$. The *likelihood* of a set of test pattern sequences X for H, represents the probability of generating, with $|X|$ trials, the set X with H:

$$P(X|H) = |X|! \prod_{x \in X} P(x|H). \tag{2}$$

3.3 The Viterbi approximation

A common simplification used to compute the generation probability of a sequence by an HMM is the *Viterbi assumption* that the sequence can only be generated by a unique path (or sequence of states) through HMM. In other words, all paths, except the most likely are assumed to have a negligible probability. Applied to HMMPs, the most likely path in H, named *Viterbi path* [24], of a pattern sequence $x = p_1 \cdots p_l$, is defined as

$$V_x = \underset{\langle s_0, \cdots, s_{l+1}\rangle \in S^{l+2}}{\arg\max} \left[\left(\prod_{i=0}^{l-1} M(s_i \to s_{i+1}) L(s_{i+1}, p_{i+1}) \right) M(s_l \to s_{l+1}) \right]. \tag{3}$$

For example, consider the set of three sequences below:

Sequence 1	Sequence 2	Sequence 3
`***10`	`***10`	`***10`
`1*001`	`1*001`	`1**01`
`0*101`	`0*101`	`0*001`
`10*01`	`11101`	`00111`
`0**01`		

The Viterbi path of Sequence 1 for the HMMP depicted in Figure 1 is $start - s_1 - s_2 - s_3 - s_4 - s_3 - end$. Moreover, this is the only path that can generate this sequence, which often occurs in practice, and the Viterbi assumption holds in this case.

Let us now introduce some notations that we shall use in the following. If X is a set of pattern sequences and H an HMMP, we note:

- P_X: the set of test patterns included in sequences from X;
- V_x: the Viterbi path of sequence x, and $V = \{V_x, x \in X\}$ the set of Viterbi paths of sequences from X;
- n_s, $n_{s\to t}$ and n_s^p: the number of times state s is used in V, the number of times transition $s \to t$ is used (in V), and the number of times state s emits the pattern p, respectively;
- $P_{X,s} = \{p \in P_X, n_s^p \neq 0\}$: the set of patterns from X emitted by state s in V ($P_X = \cup_{s\in S} P_{X,s}$);
- $Out(s)$ and $In(s)$: the sets of out and in states from and to the state s with non-zero transition probability, respectively; by convention, we consider that loops are outgoing transitions, *i.e.* if s loops to itself, then we have $s \in Out(s)$.

Using these notations, the Viterbi approximation of Expression (2) can be written as:

$$P(X, V|H) = |X|! \left(\prod_{s\in S}\prod_{t\in S} M(s \to t)^{n_{s\to t}}\right)\left(\prod_{s\in S}\prod_{p\in P_{X,s}} L(s,p)^{n_s^p}\right). \tag{4}$$

Let E_s be defined by:

$$E_s = \prod_{p\in P_{X,s}} L(s,p)^{n_s^p}, \tag{5}$$

and assume $E_{start} = E_{end} = 1$. Expression (4) can then be rewritten as

$$P(X, V|H) = |X|! \prod_{s\in S}\left(E_s \prod_{t\in Out(s)} M(s \to t)^{n_{s\to t}}\right). \tag{6}$$

This Formula underestimates the exact likelihood defined by Formula (2). As we shall see, our approach (as others, *e.g.* [10], [26]) involves maximizing Formula (6), instead of Formula (2). Because (6) is equal to or smaller than (2), but never greater, maximizing (6) actually tends to maximize (2), which explains the efficiency of this (sometimes rough) approximation. Furthermore, we shall see below that using the Viterbi approximation provides an efficient way to simplify and greatly speed up the learning procedure.

3.4 Estimation of transition and emission probabilities

We saw (Section 1) that the HMM training problem, *i.e.*,, the problem of estimating parameters of a known structure from a given sample of sequences, is a hard task. Nevertheless, when the Viterbi paths of the sample are known, there is a simple efficient approximation [15]. The same method applies to HMMPs. It uses the maximum likelihood principle and searches the parameter values which maximize Expression (6). This is equivalent to individually maximizing each sub-product E_s and $\left(\prod_{t\in Out(s)} M(s \to t)^{n_{s\to t}}\right)$ of this expression.

We recognize in $\left(\prod_{t\in Out(s)} M(s \to t)^{n_{s\to t}}\right)$ an expression identical to that of a multinomial law distribution. Then, the values that maximize it are

$$M(s \to t) = \frac{n_{s\to t}}{n_s}, \tag{7}$$

which gives us a simple estimation of the M parameters.

γ	1	0	$*$	$\bar{*}$
1	1	$\bar{*}$	1	$\bar{*}$
0	$\bar{*}$	0	0	$\bar{*}$
$*$	1	0	$*$	$\bar{*}$
$\bar{*}$	$\bar{*}$	$\bar{*}$	$\bar{*}$	$\bar{*}$

Table 1: Response table for the γ operator.

For emission probabilities, the problem is slightly more difficult and requires additional notations. Let $\delta_*(p, r)$ be the binary function that returns 1 if the bit of rank r in p is equal to $*$, and 0 otherwise. Let $\delta_{\neg *}(p, r)$ be the negation of this function, and

$$n_{\neg *}(P_{X,s}, r) = \sum_{p \in P_{X,s}} n_s^p \cdot \delta_{\neg *}(p, r) \tag{8}$$

the number of bits fixed on position r in the patterns emitted by s.

For every s, we have to find the emission pattern p_s that maximizes E_s. From Expressions (1) and (5), it follows that p_s must be compatible with all patterns of $P_{X,s}$, otherwise the value of E_s is zero. Therefore, if there are patterns in $P_{X,s}$ for which some bits are differently fixed, then p_s must have the value $*$ on these bits. Once this constraint is satisfied, when incorporating Expression (1) in Formula (5), we obtain:

$$E_s = \left(\frac{1}{2}\right)^{\sum_{p \in P_{X,s}} *^p_{p_s} n_s^p}, \tag{9}$$

that can be rewritten as

$$E_s = \left(\frac{1}{2}\right)^{\sum_{r=1}^{k} \delta_*(p_s, r) \cdot n_{\neg *}(P_{X,s}, r)}. \tag{10}$$

Maximizing Expression (10) is equivalent to minimizing, for every r, the expression:

$$\delta_*(p_s, r) \cdot n_{\neg *}(P_{X,s}, r). \tag{11}$$

We saw that, for a given position r, if there are incompatible bits in $P_{X,s}$ then p_s has the value $*$ on this position. Now if all bits on a given position r are compatible, two cases are possible. Either some bits have a given fixed value (0 or 1); then, to minimize Expression (11), the corresponding bit of p_s has to be fixed at the same value because we then have $\delta_*(p_s, r) = 0$. Or every bit on this position has the value $*$. Then, we have $n_{\neg *}(P_{X,s}, r) = 0$ and all possible solutions (0, 1 or $*$) minimize Expression (11). However, we shall see in the following that it is more relevant to keep p_s as general as possible, and thus we use the value $*$.

It follows that the emission pattern p_s can be estimated by independently computing each of its bits from the corresponding bits in the patterns from $P_{X,s}$. Now, we shall see that p_s can be computed iteratively, which accelerates our learning algorithm by state merging (Section 4.1). For this purpose, we use an associative and commutative operator denoted as γ. According to the above, for incompatible bits we have $\gamma(1, 0) = *$, while for compatible bits we have: $\gamma(0, 0) = \gamma(*, 0) = 0$, $\gamma(1, 1) = \gamma(*, 1) = 1$, and $\gamma(*, *) = *$. However, γ is not associative as defined. For example, $\gamma(1, \gamma(1, 0)) = \gamma(1, *) = 1$ whereas $\gamma(\gamma(1, 1), 0) = \gamma(1, 0) = *$ which is the right solution because there are incompatible bits. Therefore, when $*$ is obtained from $\gamma(1, 0) = *$, it has to conserve this value during subsequent steps. To solve this difficulty, we mark the $*$ obtained in this way; the marked $*$ are denoted as $\bar{*}$. The γ operator takes this into account and we have: $\gamma(1, 0) = \gamma(\bar{*}, 0) = \gamma(\bar{*}, 1) = \gamma(\bar{*}, *) = \bar{*}$. Table 1 summarizes the response table of the γ operator. For example, when $P_{X,s} = \{10*, 110, *1*\}$ then $p_s = \gamma(10*, \gamma(110, *1*)) = \gamma(10*, 110) = 1\bar{*}0$.

4 HMMP Learning

Let X be a set of pattern sequences (*e.g.* computed by an ATPG). Our aim is to build an HMMP of low size (*i.e.* with a low number of states and transitions) that generates X with as high as possible probability. Since we have no *a priori* knowledge about the structure of this HMMP, a solution could be to train several fully connected HMMPs of different sizes with the Baum-Welch algorithm or one of its variants, and to keep the best obtained model. Unfortunately, as we shall see in Section 5.2, the non-continuous nature of probabilities involved in emission patterns lowers the efficiency of these algorithms. So we chose a state merging approach [10]. The main difficulty of this latter is the computing time, usually higher than that of the classical Baum-Welch algorithm, and emphasis will be given in this section to algorithmic refinement in order to reach acceptable runtimes. This point is critical for practical applications, owing to the size of test sequences. For example, in our experiments (Section 5), one of the largest test sequence sets has a total length of 5616 vectors with size 40. Moreover, circuits are now often bigger than those of the ISCAS'89 benchmarks, and it is common to have test sequence sets of about $100,000$ vectors with size 100.

4.1 Outline of the algorithm

The learning algorithm (HmmpLearning, summarized below) proceeds in a greedy ascending way. From a specific HMMP that represents sequences from X, and only these sequences, it progressively builds a simpler HMMP using an elementary operation (merging of two states). This simplification usually induces a generalization (more sequences have non-zero probability of being generated) but also a loss of likelihood of the learning sequences. This method builds a series of HMMP: $H_0, H_1, \cdots, H_i, \cdots$ where H_i is the HMMP obtained by merging two states of H_{i-1}. Figure 2 shows an example of a run with three state mergings for the set of three sequences proposed in Section 3.3. The algorithm stops when the desired number of states (N) is reached. N is just used to stop the algorithm, and the final HMMP is not necessarily the one that will be used and implemented on the circuit. As opposed to recognition tasks, choosing the right size in the BIST framework is relatively easy. A simple method (used in Section 5) involves simulating HMMPs with decreasing sizes (as obtained from the learning algorithm), computing the fault coverage for each one, and selecting the best one according to some size/fault coverage compromise.

In the following, we first (Section 4.2) present the initial HMMP building procedure. Sections 4.3 and 4.4 respectively describe the state merging procedure and the criterion used to select the state pair to be merged at each step. In Section 4.5, we describe a method to efficiently compute the best state pair according to this criterion. The time complexity of the whole learning procedure is discussed in Section 4.6.

Algorithm 1: HmmpLearning

```
Data : X, N
H ← CreateInitialHmmp(X);
while |S| > N do
    (s_i,s_j) ← SelectBestStatePair(S);
    H ← Merge(H,s_i,s_j);
```

4.2 Building the initial HMMP

The initial HMMP is obtained by building the *prefix tree* of X. In such a tree, each path from the root to a leaf corresponds to a sequence of X, and the common prefixes are not repeated but represented by a unique path starting from the tree root. In our case, the root represents the *start* state. Next, to each state s we attach its emission pattern p_s (except for the *start* with which no emission pattern is associated) and the number of times (n_s) it is used. This number is equal to the number of leaves of the sub-tree with root s. In the same way, to each transition $s \to t$ we attach the number of times $n_{s \to t}$ it is run over, *i.e.* the number of leaves of the sub-tree with root t. Finally, we create the *end* state to which every leaf is linked with transition probability 1. The transition probabilities of the initial HMMP are estimated from n_s and $n_{s \to t}$ parameters using Formula (7). The HMMP obtained is the most specific, as it describes all sequences of X, but only these sequences. Moreover, note that the Viterbi assumption holds, since each sequence of X is generated by a unique path from the root to a leaf.

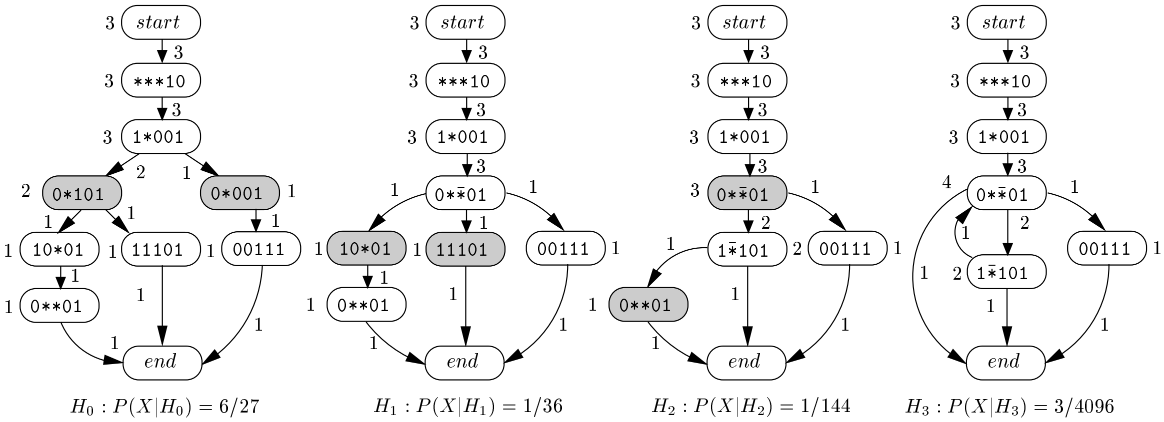


Figure 2: Structure compression achieved by three state mergings. The successive HMMPs are obtained by merging the grey nodes. For each HMMP, we indicate the likelihood of the sample (given in Section 3.3). States and transitions are labeled by the number of times they are used in Viterbi paths, *i.e.* n_s and $n_{s\to t}$, respectively. The transition probability associated with $s \to t$ is equal to the ratio $n_{s\to t}/n_s$ (Formula (7)).

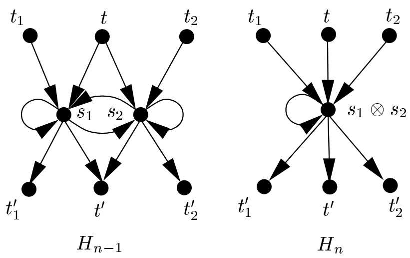


Figure 3: States and transitions before and after merging of s_1 and s_2.

The HMMP H_0 depicted in Figure 2 is the initial HMMP obtained from the sequence set of Section 3.3. Each sequence has probability $1/3$ of being generated by H_0. Therefore, $P(X|H_0) = 3! \times 1/3 \times 1/3 \times 1/3 = 6/27$.

4.3 State merging

When a state pair (s_i, s_j) has been selected (the criterion is described in Section 4.4), s_i and s_j are merged. The merging procedure is described by the MERGE algorithm (summarized below), and an example of state merging is provided in Figure 3. This procedure updates the HMMP structure and the parameters of the model. A new state $s_i \otimes s_j$ is created and the *in* and *out* transitions to and from s_i and s_j are connected to $s_i \otimes s_j$. Since this operation has to be made for each merging – and as we will see in Section 4.4 for every state pair considered in the SELECTBESTSTATEPAIR procedure – we need an efficient method to update the model parameters. Fortunately, the Viterbi approximation as used in [10] allows such a procedure. We use the optimistic assumption, which often holds in practice, that state merging does not alter Viterbi paths. Then Viterbi paths are not recomputed using Formula (3), but instead are obtained by uniting the previous paths. This yields the following simple formulae: $n_{s_i \otimes s_j} = n_{s_i} + n_{s_j}$, $n_{t \to s_i \otimes s_j} = n_{t \to s_i} + n_{t \to s_j}$, $n_{s_i \otimes s_j \to t} = n_{s_i \to t} + n_{s_j \to t}$, and $n_{s_i \otimes s_j \to s_i \otimes s_j} = n_{s_i \to s_i} + n_{s_i \to s_j} + n_{s_j \to s_i} + n_{s_j \to s_j}$. Note that even when this assumption does not hold, we still maximize an underestimate of the exact likelihood, which explains the good practical results (see also Section 3.3).

The emission pattern associated with the new state is computed using the γ operator (Table 1). The associativity of γ is relevant: rather than computing $p_{s_i \otimes s_j}$ from the set $P_{X,s_i \otimes s_j} = P_{X,s_i} \cup P_{X,s_j}$, which may yield prohibitive computing time when $P_{X,s_i \otimes s_j}$ contains numerous elements, we compute it more efficiently from the p_{s_i} and p_{s_j} emission patterns. The time complexity of the Merge procedure is $O(b+k)$, with k being the size of the patterns and b the maximal branching factor of the current HMMP.

Algorithm 2: Merge

Data : H, s_i, s_j

add a new state $s_i \otimes s_j$ to H;
$n_{s_i \otimes s_j} \leftarrow n_{s_i} + n_{s_j}$;
$p_{s_i \otimes s_j} \leftarrow \gamma(p_{s_i}, p_{s_j})$;
foreach $t \in In(s_i) \cup In(s_j) - \{s_i, s_j\}$ **do**
    add the transition $t \to s_i \otimes s_j$;
    $n_{t \to s_i \otimes s_j} \leftarrow n_{t \to s_i} + n_{t \to s_j}$;
foreach $t \in Out(s_i) \cup Out(s_j) - \{s_i, s_j\}$ **do**
    add the transition $s_i \otimes s_j \to t$;
    $n_{s_i \otimes s_j \to t} \leftarrow n_{s_i \to t} + n_{s_j \to t}$;
if $(Out(s_i) \cup Out(s_j)) \cap \{s_i, s_j\} \neq \emptyset$ **then**
    add the transition $s_i \otimes s_j \to s_i \otimes s_j$;
    $n_{s_i \otimes s_j \to s_i \otimes s_j} = n_{s_i \to s_i} + n_{s_i \to s_j} + n_{s_j \to s_i} + n_{s_j \to s_j}$;
delete s_i and s_j (and their adjacent transitions);
return H;

4.4 Merging criterion

The aim of the learning algorithm is to reduce the structure of the initial HMMP while keeping the highest possible likelihood. At each step, the algorithm chooses the state pair which, when merged, involves the lowest loss of likelihood. Nevertheless, many pairs sometimes (especially at beginning of the process) agree with this criterion. Then, from among these pairs we choose that for which emission pattern merging involves fixing the lowest number of bits. The number of bits fixed by merging the two emission patterns p and p' is obtained by

$$\varphi(p, p') = \min(*^{p}_{\gamma(p,p')}, *^{p'}_{\gamma(p,p')}). \tag{12}$$

For example, if $p =$ `1*11*` and $p' =$ `*0`$\bar{*}$`**`, then $\gamma(p,p') =$ `10`$\bar{*}$`1*`; we have $*^{p}_{\gamma(p,p')} = 1$, $*^{p'}_{\gamma(p,p')} = 2$ and then $\varphi(p,p') = 1$. Note that if p is more general than p', then $\varphi(p,p') = 0$. At the beginning of the learning procedure, using Formula (12) avoids fixing bits in the emission patterns too soon, which would make further mergings more difficult in terms of likelihood. At each step of the algorithm, the selected pair is maximal according to the lexicographic order $\succ$, defined as:

$(s_i, s_j) \succ (s_k, s_l)$ *iff* one of the two following clauses is fulfilled:
$P(X, V|H(s_i \otimes s_j)) > P(X, V|H(s_k \otimes s_l))$,
$P(X, V|H(s_i \otimes s_j)) = P(X, V|H(s_k \otimes s_l))$ and $\varphi(p_{s_i}, p_{s_j}) < \varphi(p_{s_k}, p_{s_l})$,

where $H(s_i \otimes s_j)$ denotes the HMMP resulting from the merging of s_i and s_j in H.

4.5 Fast selection of the best state pair

When selecting the best pair relative to $\succ$, we have to know $P(X, V|H(s_i \otimes s_j))$ and $\varphi(p_{s_i}, p_{s_j})$ for every pair (s_i, s_j). The value of $\varphi(p_{s_i}, p_{s_j})$ does not change while s_i or s_j are not merged. However, for $P(X, V|H(s_i \otimes s_j))$ this is not the case because its value changes after each merging. Then, we have to compute $P(X, V|H(s_i \otimes s_j)V)$ $O(|S|^2)$ times at each step, which may yield prohibitive computing time. This difficulty can be overcome by conjointly using the list $\mathcal{L}$ of the state pairs ordered according to $\succ$

(then the best state pair is the first pair of $\mathcal{L}$) and the criterion $\theta(s_i, s_j)$ defined as

$$\theta(s_i, s_j) = \frac{P(X, V|H(s_i \otimes s_j))}{P(X, V|H)}.$$

This criterion expresses the conservation of likelihood when s_i and s_j are merged. This is a good substitute for $P(X, V|H(s_i \otimes s_j))$ in searching for the best pair, because

$$P(X, V|H(s_i \otimes s_j)) > P(X, V|H(s_k \otimes s_l)) \iff \theta(s_i, s_j) > \theta(s_k, s_l).$$

Moreover, as we shall see, θ is only altered in some special cases by successive mergings, and it allows the list $\mathcal{L}$ to be rapidly updated.

$P(X, V|H(s_i \otimes s_j))$ and $P(X, V|H)$ are defined by Formula (6), which involves a product of independent factors over the HMMP state set. Between H and $H(s_i \otimes s_j)$, many factors remain identical. This is the case for E_s factors associated with states that differ from s_i, s_j and $s_i \otimes s_j$, and this is also the case for transition probabilities associated with edges non-adjacent to s_i, s_j and $s_i \otimes s_j$. For example, in Figure 3, transition probabilities associated with *out*-transitions from s_1, s_2 and t are the only ones that change (among all transitions from the current HMMP). Moreover, factors which are not altered by the merging are removed in the ratio $P(X, V|H(s_i \otimes s_j))/P(X, V|H)$. Let $In(s_i, s_j) = In(s_i) \cap In(s_j)$ denotes the *in*-states common to s_i and s_j, and $Out(s_i, s_j) = Out(s_i) \cup Out(s_j)$ be the *out*-states from s_i or s_j (recall that loops are *out*-transitions). Then, the ratio of factors related to transition probabilities that change when merging is

$$T(s_i, s_j) = IT(s_i, s_j) \cdot OT(s_i, s_j), \tag{13}$$

with

$$IT(s_i, s_j) = \frac{\prod_{t \in In(s_i, s_j)} M(t \to s_i \otimes s_j)^{n_{t \to s_i \otimes s_j}}}{\left(\prod_{t \in In(s_i, s_j)} M(t \to s_i)^{n_{t \to s_i}} M(t \to s_j)^{n_{t \to s_j}}\right)},$$

and

$$OT(s_i, s_j) = \frac{\prod_{t \in Out(s_i, s_j)} M(s_i \otimes s_j \to t)^{n_{s_i \otimes s_j \to t}}}{\left(\prod_{t \in Out(s_i)} M(s_i \to t)^{n_{s_i \to t}}\right) \left(\prod_{t \in Out(s_j)} M(s_j \to t)^{n_{s_j \to t}}\right)}.$$

And we have:

$$\theta(s_i, s_j) = T(s_i, s_j) \cdot \frac{E_{s_i \otimes s_j}}{E_{s_i} E_{s_j}}. \tag{14}$$

The first term of Expression (14) depends on the transition probabilities associated with transitions adjacent to s_i, s_j and $s_i \otimes s_j$. The second term depends only on emission patterns associated with s_i, s_j and $s_i \otimes s_j$. Therefore, $\theta(s_i, s_j)$ keeps the same value while no child or parent state of s_i or s_j is merged with another state. So, by using the θ criterion, the list $\mathcal{L}$ can be updated by only computing (and sorting) the performances of pairs which have a state adjacent to the merged pair. Then we only have to deal with $O(|S|b)$ state pairs at each step, with b being the maximal branching factor of the current HMMP.

Let us now describe the time complexity required to compute Formula (14). The $T(s_i, s_j)$ term is computed using Formula (13), that ran over transitions adjacent to s_i and s_j, and its time complexity is $O(b)$. For E_s terms, at the beginning of the learning procedure, we have $E_s = 1\ \forall s \in S$ since emission patterns associated with states have not yet been merged. Next, the $E_{s_i \otimes s_j}$ terms can be computed using Formula (9), but this computation may be too time-consuming when $P_{X, s_i \otimes s_j}$ contains numerous patterns. A better approach involves using the $n_{\neg *}(P_{X,s}, r)$ defined in Equation (8). Indeed, if s_i and s_j are merged to produce the state $s_i \otimes s_j$, we obviously have:

$$n_{\neg *}(P_{X, s_i \otimes s_j}, r) = n_{\neg *}(P_{X, s_i}, r) + n_{\neg *}(P_{X, s_j}, r), \tag{15}$$

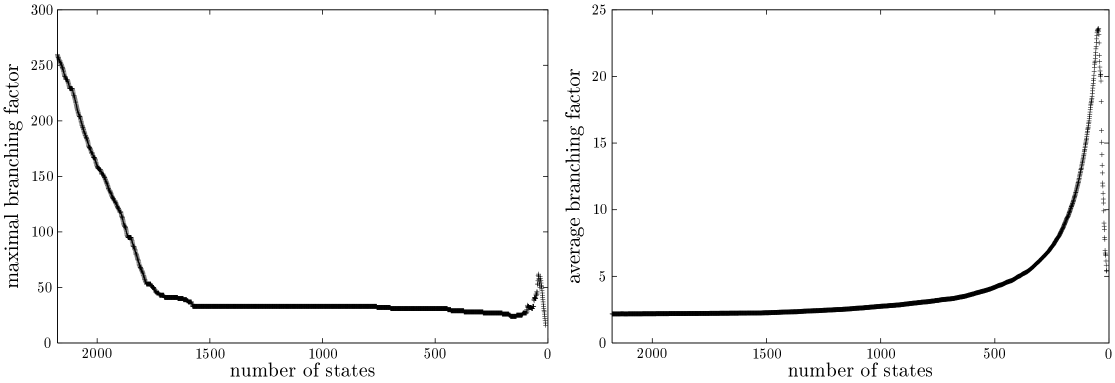


Figure 4: Evolution of the maximal (left) and average (right) branching factor.

and this equality allows $E_{s_i \otimes s_j}$ to be computed in $O(k)$ using Formula (10). At the beginning of the procedure, $P_{X,s}$ is reduced to the singleton $\{p_s\}$, and the $n_{\neg *}(P_{X,s}, r)$ terms are initialized as:

$$n_{\neg *}(P_{X,s}, r) = n_s \cdot \delta_{\neg *}(p_s, r). \tag{16}$$

To create the list $\mathcal{L}$, for each state s of H_0, we have to compute and store every $n_{\neg *}(P_{X,s}, r)$ using Formula (16) and initialize E_s to 1. Then we consider all possible $(O(|S|^2))$ state pairs; for every pair (s_i, s_j) we compute $p_{s_i \otimes s_j}$, $n_{\neg *}(P_{X,s_i \otimes s_j}, r)$, $E_{s_i \otimes s_j}$ and $T(s_i, s_j)$ and finally $\theta(s_i, s_j)$ and $\varphi(p_{s_i}, p_{s_j})$; this is done in $O(b + k)$ (with b being the maximal branching factor of H_0). Inserting a pair in $\mathcal{L}$ is done in $O(\log(|S|))$. Then the total time complexity of the $\mathcal{L}$ creating procedure is $O(|S|^2(b + k + \log(|S|)))$. The procedure for updating $\mathcal{L}$ is very similar, but involves $b|S|$ pairs instead of $|S|^2$, so its time complexity is $O(|S|b(b + k + \log(|S|)))$.

4.6 Time complexity of the whole learning procedure

The time complexity of the list creating procedure is $O(|S|^2(b + k + \log(|S|)))$, with k the size of the patterns and b the maximal branching factor of the initial HMMP. The time complexity of the updating procedure is $O(|S|b(b + k + \log(|S|)))$, and merging two states requires $O(b + k)$, with b now being the branching factor of the current HMMP. Both latter procedures have to be executed at each step of the learning procedure, *i.e.* $O(|S|)$ times. Therefore, the total time complexity strongly depends on the evolution of the maximal branching factor b. Fortunately, b remains relatively low in practice. This is essentialy due to the pair selection criterion that tends to keep the likelihood of the training sequences high. Indeed, the more the states have *out*-transitions (the higher b), the lower the likelihood. Consider Formula (6). The larger $|Out(s)|$, the lower $\prod_{t \in Out(s)} M(s \to t)^{n_{s \to t}}$. For example, with $|Out(s)| = 1$ we have $M(s \to t) = 1$ and $\prod_{t \in Out(s)} M(s \to t)^{n_{s \to t}} = 1$, while with $|Out(s)| = 2$ and $n_{s \to t_1} = n_{s \to t_2}$, we have $M(s \to t_1) = M(s \to t_2) = 1/2$ and $\prod_{t \in Out(s)} M(s \to t)^{n_{s \to t}} = (1/2)^{n_s}$.

Experiments confirm this analysis. Figure 4 shows the evolution of the maximal (mb) and average (ab) branching factor during the learning procedure for the benchmark circuit $s3330$ (c.f. Section 5). The number of states in the initial HMMP is 2176, while mb is equal to 260 and ab to 2.2; mb rapidly decreases until about 50, while ab slowly increases but remains lower than 25; throughout the procedure, we clearly have $b \ll |S|$. Therefore, the worst case analysis (mb) as well as the average analysis (ab) confirm the advantage of our approach, which replaces the $|S|^3$ factor of the naive algorithm (computing the performance of every pair at each step) by a factor $|S|^2 b$. The CPU learning time with this circuit is about 40 minutes (*PC Pentium III 350 MHz*), while the naive algorithm requires about 40 hours.

5 Experimental results

5.1 Experiments with microelectronic benchmarks

After the learning phase, the HMMP obtained can be physically implemented as a test sequence generator using a natural microelectronic translation [23]. The size of the implementation, a crucial factor for real applications, is closely connected to the size of the HMMP in terms of number of states and transitions.

We tested our method with classical benchmarks of the test community [25]. The results are reported in Table 1:

- The first three columns *Name, #I* and *#F* provide the name, the number of inputs and the number of potential faults of the tested circuit, respectively.
- The following two columns *%Cov* and *Leng* indicate the fault coverage and total length of ATPG sequences.
- For each circuit, we simulated 10 sets of sequences generated with the learned HMMP. For comparison purposes, we also simulated 10 long random sequences for every circuit. Only simulating one long sequence is justified here, due to the randomness of the approach, and no improvement is obtained by decomposing this long sequence into many shorter ones. The *T.Leng* column provides the total length of HMMP sequences; the same length was used for random sequences. When applying sequences generated by a stochastic model such as an HMMP or a fully random process, numerous (easy) faults are detected at the beginning. The frequency of new detected faults then decreases and no new faults seem to be detectable after a certain time. So the total length was manually tuned and limited to the efficient beginning part of HMMP sequences. The following column (*ratio*) provides the ratio of this length over the total length of ATPG sequences.
- The *%Best* columns indicate the best fault coverage achieved in the 10 simulations; the *%Av* columns indicate the average of these 10 fault coverages.
- Columns #S. and #E. provide the number of states and the number of transitions of the learned HMMP, respectively. As stated in Section 4, the number of states of the HMMPs was manually tuned according to the size and fault coverage constraints.
- On the bottom line, we report the means of all of these quantities.

The fault coverages achieved by our method are much larger than those of the random sequences. They are often equal (s298, s1494), sometimes slighty smaller (s820, s832) and sometimes larger (s444, s526) than the fault coverages of ATPG sequences. On average, our method provides better fault coverage than ATPG sequences, at the expense of larger (about 5 times on average) test sequence sets. This result is surprising and confirms the validity of our approach. It demonstrates that it is possible to infer very efficient construction rules from ATPG sequences. These rules do not ensure accurate generation of the original sequences, but they generate similar sequences that achieve high fault coverage when the sequence set is large enough. The good results obtained with our method could also be explained by the weakness of some ATPG sequence sets (and by the NP-hardness of the task), and by the easiness of achieving relatively high fault coverages for some circuits (see results obtained by the random method). Note also that these fault coverages are obtained with small HMMPs: the maximum number of states is 13, and the number of transitions is low relative to the number of states. This allows HMMPs to be electronically implemented using a reasonable silicon area, which makes HMMPs a suitable solution for the BIST problem [23].

5.2 Comparison with other approaches

As stated in Section 3, emission patterns define very simple probability distributions over $\{0,1\}^k$. Due to the size constraint, this simplicity is required for microelectronic purposes. However, an issue concerns the effect of this choice on the model accuracy as compared to most conventional distributions such as products of Bernoulli distributions. Another question is the relevance of using a learning algorithm by state merging rather than the classical Baum-Welch algorithm. To answer these questions, another set

Circuit Name	#I	#F	ATPG %Cov	ATPG Leng	T.Leng	ratio	Random %Best	Random %Av	HMMP %Best	HMMP %Av	HMMP #S	HMMP #E
s298	3	596	89.9	2408	2.5K	1.03	75.2	68.1	89.9	89.3	5	19
s344	9	670	97.0	427	1K	2.34	96.1	94.2	97.6	97.6	3	7
s382	3	764	85.7	9178	15K	1.63	15.4	15.4	96.7	96.3	5	13
s386	7	772	90.2	754	5K	6.63	74.00	68.8	89.6	89.3	7	20
s444	3	888	75.5	2074	10K	4.82	13.4	13.3	97.1	92.5	4	9
s526	3	1052	52.9	966	10K	10.35	10.8	10.7	82.6	76.6	5	10
s820	18	1640	96.3	4993	30K	6.00	49.4	48.4	94.2	91.8	13	40
s832	18	1664	95.3	5024	30K	5.97	47.3	46.3	93.6	90.6	13	43
s991	65	1948	99.2	1139	6K	5.27	93.8	93.6	96.9	96.6	8	25
s1488	8	2976	95.6	6776	30K	4.42	78.0	75.1	98.6	98.5	8	30
s1494	8	2988	98.1	6723	30K	4.46	78.3	75.1	98.1	98.0	8	29
s3330	40	6660	79.2	5616	30K	5.34	76.5	74.4	78.5	78.1	13	46
Avg:			87.9			4.85	59.0	56.9	92.8	91.3		

Table 2: Fault coverage achieved by ATPG, random sequences and HMMPs.

Circuit Name	Bernoulli K-M. F.C.	Bernoulli K-M. log lik	Bernoulli merg. F.C.	Bernoulli merg. log lik	HMMP K-M. F.C.	HMMP K-M. log lik	HMMP merg. F.C.	HMMP merg. log lik
s344	97.6%	270	97.6%	287	97.0%	367	97.6%	321
s526	84.3%	905	83.7%	985	12.6%	2117	82.6%	1363
s820	93.4%	3061	95.5%	2281	57.6%	6858	94.2%	2671
s3330	80.3%	6337	80.0%	6931	78.4%	8639	78.5%	7990
Avg. :	88.9%	2643	89.2%	2621	61.4%	4495	88.2%	3086

Table 3: Negative log likelihood of the ATPG sequences and fault coverages achieve by HMMPs and HMMs with Bernoulli distributions trained by the segmental K-Means algorithm and a state merging algorithm.

of experiments summarized in Table 3 was carried out for some benchmark circuits. First we trained HMMPs with a variant of the Baum-Welch algorithm known as the segmental K-Means method [26] (column HMMP K-M.). Secondly, we trained HMMs with products of Bernoulli distributions associated with the states. These HMMs were trained in two different ways: using a learning algorithm by state merging as for HMMPs (column Bernoulli merg.) and using the segmental K-Means algorithm (column Bernoulli K-M.). For each of these experiments, the numbers of states of HMMs and HMMPs and the total length of the test sequences were set at the same value as those in Table 2. The initial models trained with the segmental K-Means algorithm were fully connected, and the emission patterns of the initial HMMPs were set at $*^k$ to avoid null likelihood. For each circuit, ten models with different initial parameters were trained, and we kept the model maximizing the likelihood of ATPG sequences. Table 3 includes both the negative log likelihood of the ATPG sequences and the fault coverage achieved by the models.

First we note that, as suspected, the non-continuous nature of HMMP emission probabilities often lowers the accuracy of the classical training algorithm, both for the likelihood of the ATPG sequences and the fault coverage of the learned HMMPs. The obtained fault coverages are then close to those of random sequences (see Table 2). In contrast, the three other approaches provide models that achieve high fault coverages, which are very close although the likelihoods differ significantly. These results indicate that the merging approach is well suited for HMMP (and HMM), and that HMMP performance for BIST is not very affected by the high simplicity of emission distributions.

6 Conclusion

We presented a new model for learning Boolean vector sequences. This model is close to the classical HMM, but differs in that it defines emission probability distributions with Boolean patterns. We also presented an efficient algorithm to learn HMMPs, which uses the state merging principle. We used HMMP and our algorithm to build test sequence generators for integrated circuits, and observed that, despite their reduced size and simplicity, inferred HMMPs achieve very high coverage of potential faults of circuits at hand. The simplicity of HMMPs could also be relevant in other frameworks. Boolean patterns are very similar to the condition parts of the rules used in symbolic and hybrid classification methods [27], and, just as in these methods, this simplicity is associated with explanatory virtues which should be of interest from a modeling and learning perspective. Moreover, the algorithmic refinements we proposed for the merging algorithm (Section 4.5), which yield a great reduction of the computing time, could be fruitfully used for other models. However, improvements could be done concerning the learning procedure. First, all the ATPG sequences do not have the same testing capabilities, and it could be of interest in the learning phase to weight the sequences by the number of faults they detect in order to improve the fault coverages achieved by the learned HMMPs. Adapted techniques, such as a preliminary clustering step of the patterns of the ATPG sequences or of the bit positions within these patterns, could also enhance the efficiency and accuracy of the learning procedure.

Acknowledgements

We thank Patrick Girard and Christian Landrault of the Microelectronics Department of the LIRMM for presenting and explaining to us the problem of Built-in Self-Test for sequential integrated circuits, and for their help at the start of this work and throughout the experimental evaluation process.

Annex : Subset (14 among 284) of the test sequence set obtained using the ATPG *Sunrise* for the circuit *s820*.

```
Sequence 1
*******************0
*010************0001
*****1**********0001

Sequence 2
*******************0
*010************0001
****0*****1****11001
***************01001
****1**********01001
*1**1**********01001
*010************0001
*0*1**********010001
****0*1001*****11001
****0*********011001
****0*****00***11001
****0***********0001
*****0***********001
*0*0*1***********001

Sequence 3
*******************0
*010************0001
****0*****1****11001
***************01001
****1**********01001
*1**1**********01001
*010************0001
*0*1**********010001
****0*1001*****11001
****0*********011001
01**0*****010**11001
****0********0*11001
0***0***********0001

Sequence 4
*******************0
*010************0001
****0*****1****11001
***************01001
****1**********01001
*1**1**********01001
*010************0001
*0*1**********010001
****0*1001*****11001
****0*********011001
****0*****00***11001
****0***********0001
*****0***********001
*000*************001

Sequence 5
*******************0
*010************0001
****0*****1****11001
***************01001
****1**********01001
*1**1**********01001
*010************0001
*0*1**********010001
****0*1001*****11001
****0*********011001
****0*****00***11001
****0***********0001
*****0***********001
*001*************001

Sequence 6
*******************0
*010************0001
****0*****1****11001
***************01001
****1**********01001
*1**1**********01001
*010************0001
*0*1**********010001
****0*1001*****11001
****0*********011001
****0*****00***11001
****0***********0001
*****0***********001
*010*************001

Sequence 7
*******************0
*010************0001
****0*****1****11001
***************01001
****1**********01001
*1**1**********01001
*010************0001
*0*1**********010001
****0*1001*****11001
****0*********011001
****0*****00***11001
****0***********0001
*****0***********001
*100*************001
0***************0001

Sequence 8
*******************0
*010************0001
****0*****1****11001
***************01001
****1**********01001
*1**1**********01001
*010************0001
*0*1**********010001
****0*1001*****11001
****0*********011001
****0*****00***01001

Sequence 9
*******************0
*010************0001
****0**********11001
***************01001
****1**********01001
*1**1**********01001
*010************0001
*0*1**********010001
****0*1001*****11001
****0*********011001
****0***1*00***11001

Sequence 10
*******************0
*010************0001
****0*****1**0*11001
***************01001
****1**********01001
*1**1**********01001
*010************0001
*0*1**********010001
****0*1001*****11001
****0*********011001
****0***1*00*1*11001

Sequence 11
*******************0
*010************0001
****0*****1****11001
***************01001
****1**********01001
*1**1**********01001
*010************0001
*0*1**********010001
****0*1001*****11001
****0*********011001
****0*****1*1**11001
1***1****0******1001

Sequence 12
*******************0
*010************0001
****0*****1****11001
***************01001
****1**********01001
*1**1**********01001
*010************0001
*0*1**********010001
****0*1001*****11001
****0*********011001
01**0*****010**11001
****0********1*01001

Sequence 13
*******************0
*010************0001
****0*****1****11001
***************01001
****1**********01001
*1**1**********01001
*010************0001
*0*1**********010001
****0*1001*****11001
****0*********011001
****0*****1*1**11001

Sequence 14
*******************0
*010************0001
****0*****1****11001
*0*************01001
****1**********01001
*1**1**********01001
*010************0001
*0*1**********010001
****0*1001*****11001
****0*********011001
****0*****1*1**11001
11**************1001
```

References

[1] L. E. Baum, T. Petrie, G. Soules, and N. Weiss, "A maximization technique occurring in statistical analysis of probabilistic functions in Markov chains," *The Annals of Mathematical Statistics*, vol. 41, no. 1, pp. 164–171, 1970.

[2] F. Casacuberta, "Some relations among stochastic finite state networks used in automatic speech recognition," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 12, pp. 691–695, July 1990.

[3] L. R. Bahl, F. Jelinek, and R. L. Mercer, "A maximum likelihood approach to continuous speech recognition," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 5, pp. 179–190, Mar. 1983.

[4] L. E. Baum, "An inequality and associated maximization technique in statistical estimation for probabilistic functions of a Markov process," *Inequalities*, vol. 3, pp. 1–8, 1972.

[5] A. P. Dempster, N. M. Laird, and D. B. Rubin, "Maximum likelihood from incomplete data via the EM algorithm.," *J. Royal Stat. Soc. B*, vol. 39, pp. 1–38, 1977.

[6] N. Abe and M. Warmuth, "On the computational complexity of approximating distributions by probabilistic automata," *Machine Learning*, vol. 9(2-3), pp. 205–260, 1992.

[7] J. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Language, and Computation.* 1979.

[8] J. Oncina and P. Garcia, "Inferring regular languages in polynomial updated time.," *Pattern Recognition and Image Analysis*, pp. 49–61, 1992.

[9] R. C. Carrasco and J. Oncina, "Learning stochastic regular grammars by means of a state merging method," in *Proceedings of the Second International ICGI Colloqium on Grammatical Inference and Applications*, vol. 862 of *LNAI*, pp. 139–152, 1994.

[10] A. Stolcke and S. Omohundro, "Inducing probabilistic grammars by bayesian model merging," in *Proceedings of the ICGI*, vol. 862 of *LNAI*, (Berlin), pp. 106–118, 1994.

[11] K. J. Lang, B. A. Pearlmutter, and R. A. Price, "Results of the Abbadingo One DFA learning competition and new evidence-driven state merging algorithm," in *Proceedings of the 4th International Colloquium on Grammatical Inference (ICGI-98)*, pp. 1–12, July 1998.

[12] L. R. Rabiner, "A tutorial on hidden Markov models and selected applications in speech recognition," *Proceedings of the IEEE*, vol. 77, no. 2, pp. 257–285, 1989.

[13] A. Kundu, Y. He, and P. Bahl, "Recognition of handwritten word: First and second order Hidden Markov Model based approach," in *CVPR'88 (IEEE Computer Society Conference on Computer Vision and Pattern Recognition, Ann Arbor, MI, June 5–9, 1988)*, pp. 457–462, 1988.

[14] M. Chen, A. Kundu, and J. Zhou, "Off-line handwritten word recognition using a hidden Markov model type stochastic network," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 16, pp. 485–496, May 1994.

[15] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison, *Biological sequence analysis probabilistic models of proteins and nucleic acids.* Cambridge University Press, 1998.

[16] B. Povlow and S. Dunn, "Texture classification using noncausal hidden Markov models," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 17, pp. 1010–1014, Oct. 1995.

[17] C. Raphael, "Automatic segmentation of acoustic musical signals using hidden Markov models," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 21, no. 4, pp. 360–370, 1998.

[18] J. Rajski and J. Tyszer, *Arithmetic Built-In Self-Test.* Prentice Hall PTR, 1998.

[19] Ibarra and Sahni, "Polynomially complete fault detection problems," *IEEE Transactions on Computers*, vol. 24, 1975.

[20] F. Corno, P. Prinetto, M. Rebaudengo, and M. Sonza Reorda, "A parallel genetic algorithm for automatic generation of test sequences for digital circuits," *Lecture Notes in Computer Science*, vol. 1067, pp. 454–??, 1996.

[21] A. Ghosh, S. Devadas, and A. Newton, *Sequential Logic Testing and Verification.* 1992.

[22] V. D. Agrawal, K. T. Cheng, and P. Agrawal, "A directed search method for test generation using a concurrent simulator," *IEEE Trans. on Computer-Aided Design*, vol. 8, pp. 131–138, Feb. 1989.

[23] L. Bréhélin, O. Gascuel, G. Caraux, P. Girard, and C. Landrault, “Hidden Markov and Independence Models with Patterns for sequential BIST,” in *18th IEEE VLSI Test Symposium*, 2000.

[24] G. D. Forney, “The Viterbi algorithm,” *Proc. IEEE*, vol. 61, pp. 268–278, Mar. 1973.

[25] F. Brglez, D. Bryan, and K. Kozminski, “Combinational profiles of sequential benchmark circuits,” in *International Symposium on Circuits and Systems*, May 1989.

[26] B. H. Juang and L. R. Rabiner, “The segmental K-means algorithm for estimating parameters of hidden markov models,” *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 38, no. 9, p. 1639, 1990.

[27] O. Gascuel and the SYMENU group, “Twelve numerical, symbolic and hybrid supervised classification methods,” *International Journal of Pattern Recognition and Artificial Intelligence*, pp. 517–572, 1998.

Liste des Algorithmes

Liste des principaux symboles utilisés

Généralités et HMMs

- $\Sigma = \{a, b, c, \dots\}$: un alphabet de symboles ;
- o : un symbole de Σ ;
- $O = o_1 \dots o_T$: une séquence de symboles de longueur T ;
- $\mathcal{O} = \{O^1, \dots, O^K\}$: un ensemble de K séquences de symboles ;
- Oa : concaténation du symbole a à la suite de la séquence O ;
- H : un HMM ;
- $\mathcal{Q}$: ensemble des états ;
- $\mathcal{T}$: matrice des probabilités de transition ;
- $\mathcal{G}$: matrice des probabilités d'émission ;
- s, s', ... , t, t', ... : états de $\mathcal{Q}$;
- $s \to t$: une transition ;
- $S = s_0 - \dots - s_{T+1}$: une séquence d'états, avec $s_0 = start$ et $s_{T+1} = end$;
- S^* : chemin de Viterbi d'une séquence O ;
- S^{k*} : chemin de Viterbi de la séquence O^k ;
- $\mathcal{S}^* = \{S^{1*}, \dots, S^{K*}\}$: ensemble des chemins de Viterbi associés aux séquences de $\mathcal{O}$;
- n_s, $n_{s \to t}$, n_s^o : nombre de fois où l'états s est utilisé, nombre de fois où la transition $s \to t$ est utilisée, et nombre de fois où l'état s génère le symbole o dans les chemins de Viterbi des séquences d'apprentissage ;
- Σ_s : ensemble des symboles de Σ qui ont une probabilité non nulle d'être générés sur l'état s ;
- $In(s)$ et $Out(s)$: nombres de transitions entrantes et sortantes de l'état s ;
- $\alpha_s(t)$, $\beta_s(t)$, $\delta_s(t)$, $\psi_s(t)$, $\gamma_s(t)$, $\xi_t(s, s')$: quantité utilisées dans les algorithmes *forward*, *backward*, Viterbi, et Baum-Welch ;
- l : constante de Laplace utilisée pour lisser les distributions d'émission ;

Apprentissage par fusions d'états

- u et u' : deux états quelconques, candidats à une fusion ;
- $u \otimes u'$: état résultant de la fusion de u et u' ;

- $H_{u \otimes u'}$: HMM résultant de la fusion des états u et u' dans le HMM H ;
- H_0 : HMM initial utilisé dans la procédure d'apprentissage ;
- $\theta(u, u')$, E_s et $T(u, u') = IT(u, u') \cdot OT(u, u')$: quantités utilisées pour optimiser la procédure de sélection suivant le critère de vraisemblance maximale ;
- $\mathcal{L}$: liste des couples d'états ordonnée suivant θ ;
- $H^*_{(N)}$: HMM de taille N qui maximise la vraisemblance de Viterbi des séquences d'apprentissage ;

Nouveau critère de sélection

- $H^{(s)}$: sous-HMM de l'état s ;
- $\mathcal{O}_s$: ensemble des suffixes de $\mathcal{O}$ associés à un état s racine d'un arbre ;
- M : une projection d'un sous-HMM dans un autre ;
- $\mathcal{M}_{u,u'}$: ensemble des projections de $H^{(u)}$ dans $H^{(u')}$;
- M^* : la projection optimale ;
- $\mathcal{D}(u||u')$: divergence de u' par rapport à u ;
- l et l' : constantes de Laplace utilisées pour lisser les distributions de probabilités d'émission et de transition ;
- $\widetilde{P}(.)$: une probabilité issue d'une distribution lissée ;
- α : constante utilisée pour pondérer les états de $H^{(u)}$;
- $u \rightsquigarrow s$: distance de u à s ;
- $\mu_k(s)$: variable utilisée dans l'algorithme de calcul de la divergence ;
- τ : valeur de divergence maximale pour laquelle une fusion peut être acceptée ;
- $\mathcal{R}, \mathcal{B}, \mathcal{W}$: ensembles des états rouges, bleus et blancs ;
- d : profondeur maximale de $H^{(u)}$;

HMMPs et IMPs

- H : un HMMP ;
- $\mathcal{P}$: ensemble de patterns d'émission ;
- p : un pattern ;
- p_s : pattern d'émission associé à l'état s ;
- $*^p_{p_s}$: nombre de bits fixés dans p, mais qui ont la valeur $*$ dans p_s ;
- $O = p_1 \dots p_l$: une séquence de patterns ;
- $P_{\mathcal{O},s}$: ensemble des patterns émis par l'état s dans les chemins de Viterbi ;
- $\delta_*(p, r)$: fonction binaire qui renvoie 1 si le bit de rang r dans p est égal à $*$, et 0 sinon ; $\delta_{\neg *}(p, r)$ est la négation de cette fonction ;

- $n_{\neg *}(P_{\mathcal{O},s}, r)$: nombre de patterns émis par l'état s et dont le bit de rang r est fixé ;
- γ : opérateur de fusion ;
- $\varphi(p, p')$: nombre de bits à fixer lors de la fusion de p et p' ;
- $\succ$: ordre lexicographique ;
- I : un IMP ;
- π : distribution de probabilités *a priori* ;

Amélioration des séquences de l'ATPG

- $P_{r,s}(1)$ et $P_{r,s}(0)$: probabilité de générer un `1` (resp. `0`) à l'aide de la distribution de Bernoulli de rang r associée à l'état s ;
- $n_{r,s}(1)$ et $n_{r,s}(0)$: nombre de patterns de $P_{\mathcal{O},s}$ dont le bit de rang r est `1` (resp. `0`) ;
- $\mathcal{Z}$ et $\mathcal{Z}'$: ensembles utilisés pour sélectionner un ensemble de séquences minimal ;
- $\mathcal{F}$: un ensemble de fautes ;
- F : une faute ;
- S_F préfixe de la séquence S qui permet de détecter F ;
- $\mathcal{Z}_F$: ensemble des préfixes de séquences de $\mathcal{Z}$ permettant de détecter F ;
- $\mathcal{F}_S$: ensemble des fautes de $\mathcal{F}$ détectées par la séquence (ou le préfixe de séquence) S ;
- D_F : difficultée de F ;
- $Q_{S'}$: qualité du préfixe S' ;

Complexité des algorithmes

- N : nombre d'états d'un HMM ;
- T : taille d'une séquence, ou taille totale d'un ensemble de séquences ;
- L : nombre de tours des algorithmes d'entraînement (Baum-Welch et Viterbi) ;
- b et b' : facteur de branchement maximum, et nombre maximum de symboles portés par les états ;
- K : nombre d'états maximum du sous-HMM $H^{(u)}$;
- B : taille maximale de l'ensemble $\mathcal{B}$;

Bibliographie

Abe, N. et Warmuth, M. (1992). On the computational complexity of approximating distributions by probabilistic automata. *Machine Learning*, 9(2-3) :205–260.

Abramovici, M., Menon, P., et Miller, D. (1984). Critical path tracing : An alternative to fault simulation. *IEEE Design and Test of Computer*, 1 :83–93.

Agrawal, V. D., Cheng, K. T., et Agrawal, P. (1989). A directed search method for test generation using a concurrent simulator. *IEEE Trans. on Computer-Aided Design*, 8 :131–138.

Bahl, L. R., Jelinek, F., et Mercer, R. L. (1983). A maximum likelihood approach to continuous speech recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 5(2) :179–190.

Baker, J. K. (1975). The DRAGON system – an overview. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 23 :24–29.

Baldi, P. et Brunak, S. (1998). *Bioinformatics : The machine learning approach.* MIT Press.

Baum, L. E., Petrie, T., Soules, G., et Weiss, N. (1970). A maximization technique occurring in statistical analysis of probabilistic functions in Markov chains. *The Annals of Mathematical Statistics*, 41(1) :164–171.

Blake, C. et Merz, C. (1998). UCI repository of machine learning databases.

Brand, M. (1999). Structure learning in conditional probability models via an entropic prior and parameter extinction. *Neural Computation*, 11(5) :1155–1182.

Bréhélin, L., Gascuel, O., et Caraux, G. (2001). Hidden Markov models with patterns to learn boolean vector sequences and application to the built-in self-test for integrated circuits. *IEEE Transactions on Pattern Analysis and Machine Intelligence, à paraître.*

Bréhélin, L., Gascuel, O., Caraux, G., Girard, P., et Landrault, C. (2000). Hidden Markov and Independence Models with Patterns for sequential BIST. In *18th IEEE VLSI Test Symposium*, pages 359–367.

Carrasco, R. C. et Oncina, J. (1994). Learning stochastic regular grammars by means of a state merging method. In *Proceedings of the Second International ICGI Colloqium on Grammatical Inference and Applications*, volume 862 of *LNAI*, pages 139–152.

Casacuberta, F. (1990). Some relations among stochastic finite state networks used in automatic speech recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(7) :691–695.

Cheng, K.-T. et Agrawal, V. D. (1989). An economical scan design for sequential logic test generation. In *International Symposium on Fault-Tolerant Computing (FTCS '89*, pages 28–35.

Corno, F., Gaudenzi, N., Prinetto, P., et Sonza-Reorda, M. (1998). On the identification of optimal cellular automata for built-in self-test of sequential circuits. In *Proceedings of the VTS*, pages 424–429.

Corno, F., Prinetto, P., et Sonza Reorda, M. (1996). Advanced techniques for GA-based sequential atpgs. In *ED&TC'96*, pages 375–379.

de la Higuera, Oncina, et Vidal (1996). Identification of DFA : Data-dependent vs data-independent algorithms. In *ICGI : International Colloquium on Grammatical Inference and Applications*.

Dempster, A. P., Laird, N. M., et Rubin, D. B. (1977). Maximum likelihood from incomplete data via the EM algorithm. *J. Royal Stat. Soc. B*, 39 :1–38.

Domingos, P. et Pazzani, M. (1997). On the optimality of the simple Bayesian classifier under zero-one loss. *Machine Learning*, 29 :103–130.

Dupont, P. et Amengual, J. C. (2000). Smoothing probabilistic automata : an error-correcting approach. In *Proceedings of the 5th International Colloquium on Grammatical Inference (ICGI-00)*, LNAI. Springer.

Dupont, P. et Chase, L. (1998). Using symbol clustering to improve probabilistic automaton inference. In Honavar, V. et Slutzki, G., editors, *Proceedings of the 4th International Colloquium on Grammatical Inference (ICGI-98)*, volume 1433 of *LNAI*, pages 232–243. Springer.

Durbin, R., Eddy, S., Krogh, A., et Mitchison, G. (1998). *Biological sequence analysis probabilistic models of proteins and nucleic acids*. Cambridge University Press.

Fagot, C., Gascuel, O., Girard, P., et Landrault, C. (1998). A ring architecture strategy for BIST test pattern generation. In *7th Asian Test Symposium*, pages 418–423.

Freitag, D. et McCallum, A. (1999). Information Extraction with HMM and Shrinkage. In *AAAI'99 Workshop on ML for Information Extraction*, pages 31–36.

Freitag, D. et McCallum, A. (2000). Information Extraction with HMM structures learned by stochastic optimization. In *AAAI*.

Genest, D. (2001). *Extension du modèle des graphes conceptuels pour la recherche d'informations*. PhD thesis, Université Montpellier II, LIRMM.

Goel, P. (1981). An implicit enumeration algorithm to generate tests for combinational circuits. *IEEE Transactions on Computers*, C-30(3).

Haussler, D., Krogh, A., Mian, S., et Sjolander, K. (1992). Protein modeling using hidden Markov models : analysis of globins. Technical Report UCSC-CRL-92-23, University of California at Santa Cruz, Department of Computer and Information Sciences.

Hirschman, L., Bates, M., Dahl, D., Fisher, W., Garofolo, J., Hunicke-Smith, K., Pallett, D., Pao, C., Price, P., et Rudnicky, A. (1992). Multi-site data collection for a spoken language corpus. In *Proc. DARPA Speech and Natural Language Workshop '92*, pages 7–14.

Hoeffding, W. (1963). Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301) :13–30.

Ibarra et Sahni (1975). Polynomially complete fault detection problems. *IEEE Transactions on Computers*, 24.

Iyengar, V., Chakrabarty, K., et Murray, B. (1998). Built-in self-testing of sequential circuits using precomputed test sets. In *Proceedings of the VTS*, pages 418–423.

Juang, B. et Rabiner, L. (1985). A probabilistic distance measure for hidden markov models. *AT&T Technical Journal*, 64 :391–408.

Juang, B. H. et Rabiner, L. R. (1990). The segmental K-means algorithm for estimating parameters of hidden markov models. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 38(9) :1639.

Krogh, A., Mian, I. S., et Haussler, D. (1994). A hidden Markov model that finds genes in e.coli DNA. Technical Report UCSC-CRL-93-33, University of California, Santa Cruz, Jack Baskin School of Engineering.

Kundu, A., He, Y., et Bahl, P. (1988). Recognition of handwritten word : First and second order Hidden Markov Model based approach. In *CVPR'88 (IEEE Computer Society Conference on Computer Vision and Pattern Recognition, Ann Arbor, MI, June 5–9, 1988)*, pages 457–462.

Laird, P. D. (1988). Efficient unsupervised learning. In *Proceedings of the 1988 Workshop on Computational Learning Theory*, pages 297–311, MIT.

Lang, K. J. (1992). Random DFA's can be approximately learned fron sparse uniform examples. In Haussler, D., editor, *Proceedings of the 5th Annual ACM Workshop on Computational Learning Theory*, pages 45–52. ACM Press.

Lang, K. J., Pearlmutter, B. A., et Price, R. A. (1998). Results of the Abbadingo One DFA learning competition and new evidence-driven state merging algorithm. In *Proceedings of the 4th International Colloquium on Grammatical Inference (ICGI-98)*, pages 1–12.

Levin, L. A. (1973). Universal sorting problems. *Problemy Peredaci Informacii*, 9 :115–116. In Russian. English translation in *Problems of Information Transmission* 9 :265–266.

Lewis, D. D. et Gale, W. A. (1994). A sequential algorithm for training text classifiers. In Croft, W. B. et van Rijsbergen, C. J., editors, *Proceedings of the 17th Annual International Conference on Research and Development in Information Retrieval*, pages 3–12. Springer Verlag.

Lin, X., Cheng, W.-T., I.Pomeranz, et Reddy, S. (2000). SIFAR : Static test compaction for synchronous sequential circuits based on single fault restoration. In *IEEE VLSI Test Symposium*.

Marchock, T., El-Maleh, A., Maly, W., et Rajski, J. (1995). Complexity of sequential ATPG. In *European Design and Test Conference*, pages 252–261.

Mitchell, T. M. (1997). *Machine Learning*. McGraw-Hill.

Muth, P. (1975). A nine-valued circuit model to generate tests for sequential circuits. In *International Symposium on Fault-Tolerant Computing*.

Niermann, T. et Patel, J. (1991). HITEC : A test generation package for sequential circuits. In *European Conference on Design Automation*, pages 214–218.

Niermann, T., Roy, R., Patel, J., et Abraham, J. (1992). Test compaction for sequential circuits. *IEEE Transactions on Computer-Aided Design*, 11(2) :260–267.

Omohundro, S. M. (1992). Best-first model merging for dynamic learning and recognition. In Moody, J. E., Hanson, S. J., et Lippmann, R. P., editors, *Advances in Neural*

Information Processing Systems, volume 4, pages 958–965. Morgan Kaufmann Publishers, Inc.

Oncina, J. et Garcia, P. (1992). Inferring regular languages in polynomial updated time. *Pattern Recognition and Image Analysis*, pages 49–61.

Pillet, V. (2000). *Méthodologie d'extraction automatique d'information à partir de la littérature en science en vue d'alimenter un nouveau système d'information. Application à la génétique moléculaire pour l'extraction de données sur les interactions.* PhD thesis, Université Aix-Marseille III.

Pomeranz, I. et Reddy, S. M. (1997). Built-in test generation for synchronous sequential circuits. In *Proceedings of the ACM ICCAD*, pages 421–427.

Povlow, B. et Dunn, S. (1995). Texture classification using noncausal hidden Markov models. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 17(10) :1010–1014.

Rabiner, L. R. (1989). A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2) :257–285.

Raghunathan, A. et Chakradhar, S. T. (1995). Acceleration techniques for dynamic vector compaction. In *International Conference on Computer Aided Design*, pages 310–317.

Rajski, J. et Tyszer, J. (1998). *Arithmetic Built-In Self-Test.* Prentice Hall PTR.

Raphael, C. (1998). Automatic segmentation of acoustic musical signals using hidden Markov models. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 21(4) :360–370.

Roth, J. (1965). Algorithm for diagnosis for automata failures. *IBM Journal research and development.*

Rumelhart, D. E., Durbin, R., Golden, R., et Chauvin, Y. (1995). Backpropagation : The basic theory. In Chauvin, Y. et Rumelhart, D. E., editors, *Back-propagation : Theory, Architectures, and Applications.* Lawrence Erlbaum, Hillsdale, NJ.

Schenkel, M., Guyon, I., et Henderson, D. (1994). On-line cursive script recognition using time delay neural networks and hidden Markov models. In *Proc. ICASSP '94*, pages II–637–II–640, Adelaide, Austrailia.

Seshu, S. (1965). On a improved diagnosis program. *IEEE Transactions on Electronic Computers*, 14 :76–79.

Seymore, K., McCallum, A., et Rosenfeld, R. (1999). Learning hidden Markov model structure for Information Extraction. In *AAAI'99 Workshop on ML for Information Extraction*, pages 37–42.

Stolcke, A. et Omohundro, S. (1994a). Inducing probabilistic grammars by bayesian model merging. In *Proceedings of the ICGI*, volume 862 of *LNAI*, pages 106–118, Berlin.

Stolcke, A. et Omohundro, S. M. (1994b). Best-first model merging for hidden markov model induction. Technical Report TR-94-003, International Computer Science Institute, Berkeley, CA.

The FlyBase Consortium (1999). The FlyBase database of the drosophila genome projects and community literature. *Nucleic Acids Research*, 27(1) :85–88.

Thollard, F. et Dupont, P. (2000). Inférence grammaticale probabiliste utilisant la divergence de Kullback-Leibler et un principe de minimalité. In *CAp 2000*, pages 259–275.

Touba, N. et McCluskey, E. (1996). Altering a pseudo-random bit sequence for scan-based BIST. In *Proceedings of the ITC*, pages 167–175.

Valiant, L. G. (1984). A theory of the learnable. *Communications of the ACM*, 27(11) :1134–1142.