Laboratoire d'Informatique Fondamentale de Lille

Numéro d'ordre : 2305

THÈSE

Nouveau Régime

Présentée à

L'UNIVERSITÉ DES SCIENCES ET TECHNOLOGIES DE LILLE

Pour obtenir le titre de

DOCTEUR EN INFORMATIQUE

par

Cyrille D'HALLUIN

APPRENTISSAGE PAC PAR EXEMPLES SIMPLES ; PLATE-FORME D'APPRENTISSAGE DE LANGAGES RÉGULIERS

Thèse soutenue le 7 juillet 1998, devant la Commission d'Examen :

Président	:	Sophie TISON	Université de Lille I
Rapporteurs	:	Olivier GASCUEL	Université de Montpellier II
		Colin DE LA HIGUERA	Université de Saint-Etienne
Examinateurs	:	Max DAUCHET	Université de Lille I
		François DENIS	Université de Lille III
		Rémi GILLERON	Université de Lille III

UNIVERSITE DES SCIENCES ET TECHNOLOGIES DE LILLE
U.F.R. d'I.E.E.A. Bât. M3. 59655 Villeneuve d'Ascq CEDEX
Tél. 03.20.43.47.24 Fax. 03.20.43.65.66

" Tu te trouves malin ? " dit un oiseau à un ordinateur.
" Tu comptes peut-être plus vite que nous, créatures vivantes...
Mais qu'est-ce que tu peux ressentir ? Est-ce que tu sais pleurer, détester, aimer ? Non, Ordinateur, t'as beau être malin, tu pourras jamais nous remplacer. "
L'oiseau s'envolait et l'ordinateur soupirait : " Je peux encore rêver, n'est-ce pas ? "

À la mémoire de Fabien

Et puis le moment vient...

...d'écrire la dernière page, celle qui clôt presque quatre années de travail. Et puis le moment vient de dire **merci**...

Je tiens d'abord à remercier Sophie Tison d'avoir bien voulu présider ce jury ainsi que pour la gentillesse dont elle fait preuve quotidiennement au sein du LIFL.

Un grand merci a Max Dauchet qui a dirigé « de loin » ce travail mais qui a su être si proche dans les moments difficiles. Sa passion pour la recherche, son goût prononcé pour les exposés et les cours « non standards » ont été pour moi des guides précieux dans mon propre travail.

J'exprime ma gratitude à Colin de la Higuera et Olivier Gascuel pour l'intérêt qu'ils ont porté à ce travail. La version finale de ce manuel doit beaucoup au rapport qu'ils en ont fait.

Il y eut des moments difficiles, des virages dangereux et enfin la compréhension mutuelle. Merci à François Denis et Rémi Gilleron de m'avoir dirigé dans ces travaux. Leur grande rigueur scientifique a été un facteur déterminant pour l'accomplissement de cette thèse. Merci surtout de m'avoir fait confiance jusqu'au bout et d'avoir su trouver les mots qu'il fallait.

Je remercie les membres du LIFL qui savent encore sourire, parler sans arrière pensée et s'arrêter quelques instants quand on les croise dans les couloirs. Ceux-ci se reconnaîtront.

Petits mots d'encouragement à mes « collègues », les thésards du bureau 318 que j'ai souvent perturbés avec mon rire discret. Un merci particulier à Jean-Marc qui a partagé avec moi tant de samedi, de pizzas et de ras-le-bol au moment de rédiger.

Au sein du labo., il est un petit bureau où règnent (souvent) la joie et la bonne humeur. Un petit monde à part où s'échangent nos doutes et nos espoirs et où l'on a tant de plaisir à revenir chaque jour. Un grand merci à mes (vieux) colocataires et amis Francis et Yves pour tous ces bons moments: nos petits cafés, nos discussions tellement décalées, nos actes non manqués et toutes ces choses qui font que... (à bientôt au P.S.G !)
Une pensée également pour Marc, Jean-Pierre et Jean, « temporaires » du bureau 316, sans oublier ma Falco qui doit connaître par cœur le bout de mes doigts.

Un clin d'œil spécial à mes amis avec lesquels j'ai passé tant de bons moments à décompresser autour d'une bonne bière. Je les remercie pour leurs encouragements, leur écoute et leur réconfort quand tant de doutes m'envahissaient. Sbo, devenu écossais. Ludo, François et Fred, mes petits « scarabées » devenus tellement plus. Gillou, mon maître en conception objets: je t'attends (bisous à Virginie et Léa). Virginie, dont le téléphone est toujours disponible, au cœur grand ouvert et à l'amitié si forte. Et enfin, mon poto Pascal, mon petit frère depuis 10 ans qui connaît les gestes et les mots qui prouvent tant de choses: j'arrive (bisous à Aurore).

Un grand et beau merci à ma famille qui a dû essuyer les plâtres, supporter mes humeurs. Merci à Sylvain, Cathou, Eric, Josette et Patrick d'avoir été là, simplement. Un gros bisou à mes sœurs Sophie et Christine qui apportent les réponses si simples à mes questions tellement

complexes (bisous à Celine, Guillaume, Manon, Lorette et Jeanne). Un grand merci à mes parents pour leur amour de toujours. Une tendre pensée à ma maman qui a dû tout supporter, tout comprendre : merci pour tout et bien plus...

Enfin, merci à Sandrine pour le bonheur bleuté d'un jour de février, cette impression éphémère d'être simplement heureux.

Table des matières

Table des figures

Introduction

« Si nous avons de la chance, ils nous garderont peut-être comme animaux familiers ». Ces mots prêtés à Marvin Minsky[1] reflètent à la fois les espoirs et les craintes que l'Homme porte dans les fabuleuses avancées en matière de robotique et d'informatique. Depuis les débuts de l'Intelligence Artificielle (I.A.), certains scientifiques sont persuadés que l'on sera capable, un jour, de construire des machines qui pensent, des machines qui raisonnent, des machines qui dépasseront les capacités humaines et qui auront alors, peut-être, le pouvoir de décision sur notre propre évolution. Dans son livre « Une vie après la vie », Hans Moravec [Mor92] décrit un monde dans lequel les robots ont remplacé l'Homme et ce dans n'importe quelle tâche quotidienne. Il énumère également l'ensemble des caractéristiques attendues de ces machines pour qu'elles atteignent effectivement les capacités humaines. La puissance de ces machines est telle qu'elles deviennent capables de se « reproduire », c'est-à-dire de concevoir et de construire d'elles-même de nouvelles machines encore plus puissantes, perpétuant ainsi l'espèce artificielle et reproduisant, en quelques dizaines d'années, l'évolution de l'humanité.

Aujourd'hui, nous n'en sommes (heureusement ?) pas encore là et les recherches en la matière n'en sont encore qu'à leurs prémices. Après les échecs relatifs de l'I.A., de nouvelles approches en matière de recherches sur une possible « vie artificielle » ont vu le jour, regroupées dans les *sciences cognitives* (pour une présentation détaillée des sciences cognitives, voir l'article de Daniel Andler dans l'Encyclopædia Universalis). Ces sciences ont pour objet de décrire, d'expliquer et éventuellement de simuler les capacités de l'esprit humain. Cette démarche scientifique se fixe comme consigne de penser ensemble le cerveau, l'esprit et la machine. Pour ce faire, de nombreuses disciplines différentes sont impliquées parmi lesquelles la psychologie, la linguistique, la philosophie, les neurosciences et l'Intelligence Artificielle. Plusieurs taxonomies peuvent être introduites pour classer les recherches relatives aux sciences cognitives. L'une d'elles considère les aptitudes cognitives regroupées en quatre grands domaines principaux : le langage, le raisonnement, la perception et l'action. Remarquons que dans toute activité humaine, ces quatre domaines ne sont évidemment pas indépendants les uns des autres ; dès lors, ces études deviennent elles-même étroitement liées.

Le Domaine du Raisonnement

L'objet de cette thèse concerne l'un de ces domaines à savoir le *raisonnement*. Le raisonnement est peut-être l'une des formes les mieux observables de ce que l'on a l'habitude d'appeler *intelligence*. Le terme « raisonnement » doit ici être considéré au sens large, c'est-à-dire comme une activité de l'esprit qui le fait passer d'un certain niveau de connaissances à un autre ; on parle également d'*inférence*.

1. Il niera les avoir jamais prononcés.

L'inférence déductive

La forme la plus classique et la plus étudiée d'inférence est l'inférence dite *déductive*. Il s'agit d'un type de raisonnement direct, progressif et rigoureux parmi lequel on trouve les syllogismes, chers à Aristote. Il consiste à déduire à partir d'un ensemble de connaissances (appelées *prémisses*) une information nouvelle (la *conclusion*) selon un schéma logique. Ce type de raisonnement peut-être formalisé de manière logique comme suit :

$$\frac{x, x \rightarrow y}{y}$$

c'est-à-dire, si x implique y et si x est vrai alors y est vrai. Un tel raisonnement est applicable pour toute valeur de x et de y dès lors que l'on sait qu'il existe une règle du type $x \rightarrow y$ liant ces deux éléments. Le raisonnement fait donc effectivement passer le système (humain ou artificiel) du niveau de connaissances x est vrai et x implique y à un nouveau niveau de connaissances plus riche incluant le fait y est vrai. Cependant, il convient de remarquer que la connaissance sur y existait déjà dans l'ensemble des connaissances initiales de manière latente et que l'inférence n'a eu qu'à la faire émerger de l'ensemble des informations initiales. Peut-on dire alors qu'il y a eu véritablement enrichissement des connaissances?

L'inférence inductive

Un autre type d'inférence commence à être fortement étudié notamment en Intelligence Artificielle : l'inférence *inductive*. Ici, le système est confronté à un ensemble d'observations duquel il essaye de tirer une loi générale. Pour ce faire, il lui faut exhiber de ces observations un certain nombre de caractéristiques communes afin de construire effectivement cette règle. Le processus d'induction comporte essentiellement deux étapes : la première consiste à former une *hypothèse* à partir des connaissances préexistantes et des observations. Ensuite, cette hypothèse est testée, confrontée à de nouvelles observations en vue de la confirmer ou de l'infirmer. Il s'agit donc d'un raisonnement essentiellement expérimental dans lequel hypothèses et observations sont étroitement liées. Remarquons qu'en général, le système n'est confronté qu'à un nombre fini d'observations desquelles il doit tirer une généralité ; ce passage du particulier au général implique *de facto* une notion de probabilité sur l'hypothèse qu'il a inférée et non l'idée de nécessité, comme c'est le cas dans la déduction.

Prenons par exemple le cas du zoologiste s'intéressant aux panthères. Celui-ci, ayant étudié un grand nombre de spécimens de panthères, tire comme loi sur les panthères que : « toute panthère a un pelage de couleur noire, jaune moucheté ou marbré. ». Cette règle est certainement très probable d'autant plus que tous les zoologistes du monde ont tiré la même conclusion de leurs observations. Cette loi décrit par conséquent une caractéristique vraie pour toutes les panthères rencontrées par l'ensemble des zoologistes. Cependant, cette règle devrait être remise en cause si un zoologiste rencontrait un jour une panthère rose.

Remarquons qu'au contraire de l'inférence déductive, le raisonnement par induction ajoute effectivement de nouvelles connaissances au système puisque l'action de généraliser entraîne la faculté d'attribuer des propriétés à des éléments encore inconnus.

L'Inférence Inductive comme Processus d'Apprentissage

Le sujet de cette thèse concerne, nous l'avons déjà dit, le domaine du raisonnement des sciences cognitives, mais plus particulièrement l'acte d'*apprentissage*. L'étude de l'appren-

tissage automatique (*machine learning*) est devenue une spécialité à part entière au sein de l'Intelligence Artificielle. Ces travaux ont pour but de modéliser l'acte d'apprentissage, de manière à pouvoir le reproduire sur les machines : une « machine pensante » se devant de pouvoir apprendre.

Définir ce qu'est l'apprentissage est presque une gageure étant donné le nombre important de formes différentes que peut prendre un tel acte (apprentissage comportemental, apprentissage par analogie, apprentissage par l'exemple, etc...). Quoiqu'il en soit, on peut dire qu'il y a apprentissage lorsqu'un système améliore ses capacités au moyen d'une acquisition de connaissances et non par la description explicite d'un procédé. Lorsque le système est une machine, il s'agit donc d'augmenter son pouvoir de traitement d'informations autrement qu'en programmant effectivement ce processus.

Le but de notre travail est d'étudier l'une de ces formes d'apprentissage : l'*apprentissage par l'exemple*. L'apprentissage par l'exemple est typiquement un raisonnement de type inférence inductive puisqu'il consiste à construire une hypothèse généralisant un ensemble (souvent fini) d'exemples observés. Il est essentiel de comprendre que ce type d'apprentissage n'a pas la prétention de décrire l'ensemble des formes d'apprentissage. La modélisation de l'apprentissage par l'inférence inductive n'est donc qu'un lit de Procuste pour l'apprentissage naturel.

Caractéristiques Attendues de l'Apprentissage

Afin de modéliser l'apprentissage humain, il est naturel de se référer à celui-ci et d'essayer de le caractériser. En effet, si le but est de simuler l'apprentissage humain par une machine, il est essentiel de retrouver les traits majeurs de cet acte. L'un des premiers modèles d'apprentissage a été proposé en 1967 par Gold [Gol67]. Dans celui-ci, l'apprentissage est vu comme un processus infini qui, à la limite, infère exactement la règle décrivant les observations. Ce modèle est intéressant puisqu'il traduit exactement le comportement d'apprentissage des scientifiques (comme par exemple les physiciens) face à des phénomènes naturels qu'ils tentent d'expliquer. Au cours des siècles, ceux-ci construisent des hypothèses de manière à décrire les observations vues jusqu'alors ; ces lois sont constamment remises en cause par de nouvelles découvertes qui les affinent ou bien encore les rejettent complètement. Le système qu'il modélise est donc bien plus l'apprentissage de l'humanité plutôt que l'apprentissage d'un individu en particulier.

Maintenant, si le sujet d'étude devient l'apprentissage individuel, une caractéristique importante à prendre en compte est la rapidité d'un tel acte. En effet, il n'est pas concevable d'imaginer l'apprentissage individuel comme un acte infini. Par exemple, les psychologues admettent que l'apprentissage de la langue maternelle a lieu essentiellement dans les deux premières années de la vie. Notons tout de même qu'ici encore, le modèle de Gold peut s'appliquer puisque les connaissances que chacun a de sa langue maternelle sont constamment raffinées tout au long de la vie. Pour autant, de nombreux cas d'apprentissage naturel requièrent de la part du système que celui-ci infère une hypothèse valable en un temps suffisamment court. Cette exigence sur la rapidité de l'apprentissage implique cependant un relâchement sur la règle générée. En effet, dans le cas du modèle de Gold, l'apprentissage doit être exact à la limite. Dans le cas d'apprentissage en temps acceptable, cette exigence n'est -en général- plus possible. Dès lors, on admet que le système puisse se tromper. Cette erreur possible se situe en général à deux niveaux : d'abord l'hypothèse inférée peut ne pas être une règle décrivant exactement le monde observé, c'est-à-dire que pour la plupart des cas elle est

valable mais il existe certains exemples pour lesquels elle peut ne pas s'appliquer : on parle alors d'*apprentissage approximatif*. Le deuxième type d'erreur possible est dû au fait que dans certains cas, les données observées ne permettent pas d'inférer une hypothèse valide, même approximative ; nous nous trouvons alors dans un cas d'échec d'apprentissage. Le modèle PAC (pour **P**robablement **A**pproximativement **C**orrect), proposé par Valiant [Val84], formalise ce type d'apprentissage : l'apprentissage est dans ce cas souvent atteint c'est-à-dire que dans la plupart des cas, le système infère une hypothèse assez proche de la réalité. Ces deux degrés de liberté sont intégrés au modèle au moyen de deux paramètres : le paramètre de précision qui définit l'erreur tolérée sur l'hypothèse inférée et le paramètre de confiance qui décrit la probabilité que l'hypothèse inférée soit effectivement une règle acceptable. Le modèle PAC est considéré comme un modèle très exigeant. En effet, il impose que l'apprentissage réussisse dans la plupart des cas et ce quelque soit la façon dont les exemples ont été présentés à l'apprenant.

Lorsqu'on étudie l'apprentissage, le cas de l'enseignement semble un bon référant. Nous nous trouvons alors en présence de deux entités : l'enseignant et l'élève. Dans l'ensemble de cette thèse nous utiliserons cette analogie présentant l'avantage d'être bien connue. Remarquons que le terme « enseignant » doit être pris de manière générique comme étant le système ayant la connaissance et présentant des clichés de cette connaissance au moyen d'exemples. C'est typiquement ce que fait le professeur face à ses élèves mais c'est également ce que fait la Nature face aux scientifiques. Lorsqu'on observe un système enseignant-élève, un principe naturel semble bien souvent guider le processus d'apprentissage : l'enseignant qui veut apprendre à son élève une nouvelle notion, lui présentera, en général, des exemples (*i.e.* des observations) simples de manière à ne pas introduire de difficulté supplémentaire. Le modèle de Valiant classique ne reflète pas cette caractéristique de l'apprentissage humain puisque ce modèle impose d'apprendre (approximativement) quelque soit l'ensemble des observations (c'est d'ailleurs la raison pour laquelle l'apprentissage échoue complètement dans certains cas).

Li et Vitányi [LV91] proposent un nouveau modèle dans lequel les exemples simples occupent une place plus importante en vue de faciliter l'apprentissage et de manière à rendre compte de ce principe naturel. Intuitivement, un objet est simple s'il est facilement descriptible, soit encore s'il contient une (des) structure(s) régulière(s). On oppose souvent la notion d'objet simple à celle d'objet aléatoire ; en effet, de part sa nature, un objet aléatoire ne peut être décrit qu'en énumérant effectivement l'ensemble de ses composantes. Un objet simple, au contraire, peut être décrit au moyen d'un « algorithme » permettant de le construire. Par exemple, la suite composée de 1 million de 0 peut être décrite autrement qu'en énumérant successivement chacun des 0 mais simplement en disant « une suite de 1 million de 0 ». Au contraire, la suite composée de 1 million de valeurs 0 ou 1 disposées de manière aléatoire ne peut être décrite qu'en énumérant effectivement chacune de ces valeurs. La *complexité de Kolmogorov* (ou bien encore *complexité algorithmique*) définit la complexité d'un objet (d'une chaîne le représentant) comme étant la longueur du plus petit programme permettant de générer cet objet. La complexité de Kolmogorov d'un objet est souvent considérée comme la valeur optimale de compression de cet objet. Le modèle de Li et Vitányi favorise l'utilisation des exemples ayant une complexité de Kolmogorov faible ; pour ce faire, le modèle de Li et Vitányi s'appuie sur la mesure de Solomonoff-Levin qui affecte une probabilité forte aux exemples de faible complexité.

La notion de simplicité considérée par Li et Vitányi est cependant par trop absolue et ne reflète pas encore complètement la méthode traditionnelle de l'enseignant qui adapte le

mieux possible les exemples à la notion à apprendre. Aussi, de nombreux autres modèles ont vu le jour, dans lesquels les notions d'exemples caractéristiques ou bien encore d'exemples simples par rapport à la cible tiennent un rôle déterminant ; citons par exemples les travaux de Goldman et Kearns [GK91], ceux de Jackson et Tomkins [JT92] ou bien encore de Goldman et Mathias qui propose un modèle d'enseignabilité [GM93]. L'un des résultats principaux de cette thèse est la proposition d'un tel modèle, appelé modèle PACS, dans lequel les exemples les plus simples pour une cible donnée sont privilégiés au moment de l'apprentissage [DdG96]. Ici encore, le principe de simplicité est utilisé mais la notion de simplicité que nous proposons de considérer est relative à la cible. De la même manière que l'enseignant propose des exemples bien adaptés et simples par rapport à la notion qu'il désire enseigner, nous favorisons la présentation des exemples les plus simples par rapport au concept cible. Notre modèle s'appuie sur la *complexité de Kolmogorov d'un objet x relativement à un objet y*. Il s'agit, dans ce cas, de la longueur du plus petit programme construisant x en connaissant l'objet y ; un tel programme a alors la possibilité d'utiliser l'information qu'il a sur y pour générer x. Si x dépend de y (par exemple si x est une observation du concept y), il est évident qu'un programme connaissant y décrit l'objet x de manière beaucoup plus simple qu'en l'absence d'une telle information. De même que la distribution de Solomonoff-Levin affecte une probabilité forte aux objets ayant une complexité de Kolmogorov faible, la distribution de Solomonoff-Levin relative à un objet y privilégie les objets ayant une faible complexité de Kolmogorov relative à y. Dans le modèle que nous proposons, c'est cette mesure de probabilité qui est utilisée de manière à modéliser le principe d'adaptation des exemples par rapport au concept cible.

L'Inférence Inductive : Classer et Généraliser pour Apprendre

Avant tout, précisons de quelles manières l'interaction entre l'enseignant et l'élève peut être établie. Jusqu'à présent, nous avons parlé d'un ensemble d'observations (on utilisera par la suite le terme d'*échantillons*) à partir duquel l'élève doit tirer une loi générale. Une observation d'un phénomène peut être de deux types : soit cette observation est un cas particulier du phénomène qui répond alors positivement à la description de celui-ci. On parle dans ce cas d'*exemple positif*. Soit l'observation considérée est rejetée par la règle décrivant le phénomène ; il s'agit alors d'un *exemple négatif*. Deux choix s'offrent alors à l'apprenant afin d'aborder le problème. Soit il ne considère que les exemples positifs du phénomène étudié : on parle alors d'*apprentissage par exemples positifs*, soit il traite tous les exemples possibles (qu'ils soient positifs ou négatifs), ce qui correspond alors à un *apprentissage par exemples positifs et négatifs*. L'utilisation d'exemples négatifs permet, dans bien des cas, de guider l'élève dans son apprentissage. Ainsi, en fournissant à l'élève des contre-exemples présentant des traits partagés avec les exemples positifs, l'enseignant indique simplement que ces propriétés ne suffisent pas à caractériser les observations positives. Prenons par exemple le cas du physicien étudiant les métaux. Lors de ses expériences, il observe que tout métal est un bon conducteur pour l'électricité ; il est alors tenté de dire « les métaux sont les bons conducteurs d'électricité ». Cependant, si notre physicien découvre un jour que le carbone, qu'il sait ne pas être un métal, est également un bon conducteur d'électricité, il modifiera alors sa loi en essayant de trouver d'autres caractéristiques des métaux non présentes dans le carbone.

Pour le moment, nous avons présenté l'inférence inductive comme un acte consistant à construire une loi générale permettant de décrire un ensemble d'observations. Lorsqu'il est confronté à un ensemble d'observations diverses, le système apprenant va essayer de trouver

des caractéristiques communes à tous les exemples. Il est alors évident que l'ensemble des objets présentant les mêmes caractéristiques forme une classe. La règle inférée peut donc bien être vue comme une règle de classification, décidant de l'appartenance à la classe (appelée *concept* par la suite) de tout élément de l'univers étudié. Remarquons cependant que restreindre l'apprentissage à la construction d'une règle de classification n'est pas suffisant pour modéliser cet acte. Soit par exemple un ensemble d'observations $O_1, O_2, \ldots O_p$ qui sont des exemples positifs pour le concept C à apprendre. Une première règle très simple peut être inférée : « Un objet o est un élément du concept C si c'est soit O_1, soit O_2, ..., soit O_p.». La loi inférée est évidemment une règle de classification puisque l'on est en mesure de décider pour tout objet de l'univers s'il fait ou non partie du concept, simplement en le comparant aux diverses observations connues. La génération d'une telle loi ne peut cependant être désignée sous le nom d'apprentissage ou bien alors il s'agit tout au plus d'un *apprentissage par cœur.* Une qualité essentielle demandée à l'apprentissage est la généralisation qui permet, par la suite, de classer des exemples non encore vus. C'est ici que la puissance de l'inférence inductive est effectivement mise en jeu. Supposons maintenant que je découvre que mes observations O_1, O_2, ..., O_p partagent les caractéristiques communes $C_1, C_2, \ldots, C_n$. Alors, la loi : « Un objet o est un élément du concept C s'il présente les caractéristiques $C_1, C_2, \ldots C_n$. » est effectivement une règle généralisant mon ensemble d'observations, en tout cas bien plus que la simple énumération de ces observations. Nous pouvons affirmer qu'il y a eu, lors de la construction de cette règle, un véritable acte d'apprentissage.

Reprenons l'exemple de notre zoologiste étudiant les panthères. Au cours de son étude, celui-ci a constaté que les panthères étaient des félins, sauvages, dont la robe est de couleur noire, jaune tachetée ou marbrée. Il est alors en mesure de classer comme panthère ou non tout animal qu'il rencontrera par la suite. Ainsi, en face d'un canari, celui-ci classera cet animal en tant que non panthère puisque ne présentant aucune des caractéristiques communes aux panthères.

Dans le cas de l'apprentissage par exemples positifs seuls, un autre problème peut se poser, connu sous le nom de *sur-généralisation.* Les exemples positifs ne permettent pas, dans ce cas, d'affiner la règle de classification ; celle-ci devient alors trop générale et classera comme positives les observations futures présentant des traits communs à tous les exemples de l'échantillon d'apprentissage, ceux-ci n'étant pourtant pas caractéristiques uniquement du concept étudié. L'exemple précédemment évoqué du physicien étudiant les métaux est un cas typique de sur-généralisation. Le problème de la sur-généralisation fait l'objet de nombreux travaux ; on trouvera notamment dans [Ang80] une étude de ce problème dans le cas de l'apprentissage des langages réguliers.

Connaissances *a priori*

Lorsqu'un système apprend ou étudie un phénomène, celui-ci a conscience de sa tâche, il possède un minimum de connaissances présentes au niveau interne sous diverses formes. Ces connaissances *a priori* permettent au système de comprendre ce qu'on attend de lui et ainsi d'orienter son raisonnement dans le domaine convenable.

La première de ces connaissances *a priori* est évidemment le mécanisme d'apprentissage lui-même. Le jeune enfant apprenant sa langue maternelle dispose d'un mécanisme interne implantant l'ensemble des traitements indispensables au processus d'apprentissage, c'est-à-dire une prédisposition de ses neurones lui permettant de traiter l'ensemble des informations

relatives au langage. En ce qui concerne les systèmes artificiels, le principe est le même, c'est-à-dire que nous considérons que ce système dispose d'un algorithme d'inférence préexistant à la phase d'apprentissage. Précisons de plus que cet algorithme peut être connu de l'enseignant, ce qui peut éventuellement impliquer un biais lors de l'apprentissage comme c'est le cas dans le phénomène de *collusion*. Ce phénomène peut être vu comme un « accord » entre l'enseignant et l'élève, le premier s'engageant à fournir l'intégralité du concept au second par l'intermédiaire des exemples. Le travail de l'apprenant consiste alors en un simple décodage de ces exemples, de manière à reconstruire le concept attendu. Cette « tricherie » ne peut évidemment pas être qualifiée d'apprentissage puisque l'élève se contente simplement de retrouver l'objet attendu parmi tous les objets dont il dispose, éclipsant complètement le travail de **compréhension** sur les observations disponibles.

Un deuxième type d'informations supposées connues du système avant le début de l'apprentissage est le type d'« objets » étudiés. Intuitivement, on peut imaginer que l'enseignant qui aborde un nouveau sujet d'étude avec ses élèves commence par expliquer quel sera le sujet général du cours. Il en est de même dans le cas de l'apprentissage automatique, c'est-à-dire qu'un concept n'est pas appris en dehors de tout domaine. Il s'agit de définir dès le début un univers d'étude (appelé *classe de concepts*) auquel appartient le concept étudié. Nous verrons dans la suite de cette thèse que la définition d'apprenabilité ne s'applique non pas à un concept en particulier mais à une classe de concepts. En effet, définir des algorithmes d'apprentissage dont le seul but est de générer une règle de classification pour un seul concept ne présente aucune difficulté et aucun intérêt. Un tel algorithme ne serait en fait qu'un système de décodage des caractéristiques différenciant les éléments appartenant au concept des autres. Les algorithmes d'apprentissage doivent au contraire être capables, pour une classe de concepts donnée, de générer toute règle de classification pour chaque concept particulier de la classe. En ce qui concerne le zoologiste, celui-ci est capable de générer tout aussi bien[2] une loi caractérisant les panthères, les canaris ou les cochons d'inde. Précisons enfin qu'un système naturel apprenant dispose également d'un ensemble de connaissances annexes relatives ou non au domaine étudié. Un élève en cours de mathématiques par exemple connait *a priori* un ensemble de théorèmes et de concepts mathématiques qu'il pourra utiliser lors d'un nouveau cours. C'est une dimension que les systèmes artificiels se doivent d'intégrer également.

Importance de la Représentation des Connaissances

Tout système vivant, en phase d'apprentissage, dispose de multiples dispositifs sensoriels afin de percevoir son environnement. C'est grâce à ses sens, que le système est capable de découvrir certaines caractéristiques de ses observations. Les systèmes artificiels n'ont, actuellement, pas cette faculté. Concevoir des machines « intelligentes » demande donc d'intégrer cette capacité de sentir ou de ressentir l'environnement extérieur. Ces recherches sont, pour le moment, du ressort de la robotique et n'est pas notre sujet de préoccupation ici. En ce qui nous concerne, nous nous situons en aval de la perception de l'univers, c'est-à-dire que nous supposons acquis l'ensemble des perceptions décrivant les observations. Cet ensemble d'informations est mémorisé au niveau interne dans une représentation spécifique. Ainsi, l'ensemble des données (observations ou règles inférées) est décrit au moyen de cette représentation interne.

Quoiqu'il en soit il convient de comprendre l'importance de ce système de représentations

2. En utilisant la même méthode d'inférence.

interne dans le processus d'apprentissage. Intuitivement, le système de représentations se doit d'être suffisamment riche pour que les caractéristiques intéressantes et devant être découvertes par l'apprenant puissent être effectivement décrites dans cette représentation.

Prenons l'exemple de deux enfants à qui l'on désire apprendre la notion de danger[3]. Le premier de ces enfants habite la ville. Le danger est par conséquent synonyme de voitures, de trafic routier etc.... Il convient par conséquent que son système de représentations permette de décrire des caractéristiques telles que la vitesse, la puissance des moteurs, etc.... Le deuxième enfant habite la jungle. Pour lui, le danger est constitué des animaux sauvages qu'il est susceptible de rencontrer. Son système de représentations doit lui permettre de reconnaître de tels animaux caractérisés par une mâchoire puissante ou bien encore par le fait qu'ils rampent. Maintenant, supposons que l'on intervertisse le milieu de chacun de ces enfants. Ils ne sont alors plus capables de percevoir la notion de danger respectif à ces deux univers puisque leurs systèmes de représentations ne permet pas d'intégrer les caractéristiques des dangers de l'univers dans lequel ils se trouvent.

Nous verrons par la suite que la propriété d'apprenabilité d'une classe de concepts est établie étant donné un ensemble de représentations. Un système de représentations particulièrement étudié en théorie de l'apprentissage est la représentation sous forme de formules logiques. Chaque caractéristique est alors représentée au moyen d'une variable booléenne prenant la valeur vrai ou faux suivant la présence ou non de cette caractéristique. Les observations sont, quant à elles, représentées aux moyen de vecteurs booléens. Ce système de représentations est évidemment bien adapté à l'étude de l'apprenabilité des classes booléennes. Nous verrons un autre type de représentations, au moyen des automates finis, lorsque nous étudierons l'apprentissage des langages formels.

Le système de représentations joue également un rôle déterminant dans l'apprentissage au niveau de la complexité du traitement. Ainsi, une même classe peut être prouvée apprenable polynomialement (c'est-à-dire en un temps acceptable) dans un système de représentations donné, alors qu'elle peut être apprenable avec une complexité beaucoup plus importante (et donc non praticable) dans un autre système de représentations. Plus précisément, remarquons que l'ensemble de représentations est essentiellement utilisé par l'apprenant pour construire ses hypothèses. Un tel système de représentations doit donc permettre de décrire au moins le concept cible. Cependant, il se peut que ce système permette de décrire une classe de concepts beaucoup plus générale que la classe des concepts cibles. Ainsi, l'hypothèse construite par l'élève peut effectivement lui permettre de prédire de futures observations relatives au concept cible sans pour autant être la description *stricto sensu* de celui-ci, telle que celle dont dispose l'enseignant.

Compresser pour Apprendre

Tout le problème de l'apprentissage automatique consiste à trouver une règle généralisant au mieux un ensemble d'observations. Nous avons déjà évoqué le fait que se contenter de mémoriser l'ensemble des exemples vus ne peut être considéré comme une méthode satisfaisante d'apprentissage. Il s'agit plutôt de déceler, parmi l'ensemble des exemples, les informations communes à ces exemples et caractérisant ceux-ci. L'apprentissage peut donc être vu comme une méthode de compression ; l'échantillon disponible au départ n'étant pas mémorisé en extension mais plutôt en intension au moyen de ses propriétés intrinsèques.

3. Cet exemple est emprunté à la littérature.

Il semble que cette idée de comprimer pour apprendre se retrouve effectivement dans l'acte d'apprentissage humain. Par exemple, nous avons tous plus ou moins une idée de ce qu'est une maison. Lorsqu'un enfant dessine une maison, il s'agit en général de la « maison type » c'est-à-dire un bâtiment massif avec deux fenêtres, une porte au milieu, un toit rouge et une cheminée. Cette maison type est, pour la plupart des humains, la représentation interne qu'ils se font du concept de maison. Pourtant, il est bien rare de rencontrer une telle maison idéale ; bien plus, les exemples de maisons auxquels nous sommes confrontés présentent un niveau de détails bien supérieur que l'on s'empresse d'oublier pour ne retenir que l'essentiel.

Remarquons que ce principe de « compression » s'applique tant sur les exemples positifs que sur les exemples négatifs d'un phénomène. Ainsi, dans certains cas, il est possible de se contenter de généraliser un ensemble d'observations en inférant une règle décrivant les caractéristiques communes aux exemples positifs et éventuellement en listant de manière exhaustive les quelques exemples négatifs présentant les mêmes propriétés. Cependant, si le nombre de tels contre-exemples devient trop important, une telle énumération devient alors impossible et le principe de compression s'applique alors à nouveau, de manière à simplifier la règle inférée. Revenons par exemple sur le cas du physicien. Il infère comme loi que « un élément est un métal si c'est un bon conducteur d'électricité ». Il a cependant observé que le carbone, qui n'est pas un métal, est lui même bon conducteur d'électricité. Il peut donc inférer comme règle que : « un élément est un métal si c'est un bon conducteur d'électricité et si ce n'est pas du carbone ». Cette règle simple est satisfaisante si le seul contre-exemple à la partie générale se limite au carbone. Maintenant, si ce physicien découvre un grand nombre de nouveaux éléments non métalliques $e_1, e_2, \ldots e_n$ qui sont également bons conducteurs d'électricité, il ne peut se contenter d'inférer la règle trop complexe : « un élément est un métal si c'est un bon conducteur d'électricité et si ce n'est ni du carbone, ni e_1, ni e_2, ..., ni e_n ». Il doit alors soit trouver une caractéristique commune uniquement aux conducteurs non métalliques, soit encore raffiner la description des métaux au moyen d'une propriété ne se retrouvant pas sur les conducteurs d'électricité non métalliques.

Cette manière de voir l'apprentissage comme un acte de compression sera l'un des fils conducteurs de cette thèse lors de la présentation du rasoir d'Occam par exemple. Ce principe énonce que pour expliquer un phénomène, bien souvent l'explication la plus simple est la meilleure. Appliqué à l'apprentissage automatique, notamment dans le modèle d'apprentissage PAC, ce principe permet d'établir un théorème qui est l'un des plus beaux résultats de l'apprentissage PAC [BEHW87]. Sommairement, ce théorème établit qu'un algorithme qui retourne une hypothèse courte et consistante avec un échantillon d'entrée peut être utilisé comme algorithme de PAC apprentissage.

Plan de la Thèse

Dans cette thèse, nous nous fixons deux objectifs principaux. D'une part, nous essayerons de montrer l'importance des échantillons caractéristiques dans l'apprentissage, c'est-à-dire de certains ensembles d'observations contenant « toute l'information » du concept à apprendre. Le deuxième objectif consistera à montrer que l'apprentissage peut effectivement être vu comme un acte de compression. De plus, l'ensemble des idées énoncées dans cette introduction sera développé en détail par la suite.

Cette thèse se décompose en deux parties relativement indépendantes. La première partie (chapitres 1 à 3) présente la théorie de l'apprentissage automatique. L'ensemble des définitions

relatives à ce domaine sera présenté ainsi que trois modèles d'apprentissage particuliers. Un état de l'art de l'inférence des langages réguliers sera l'occasion d'étudier ces quelques modèles de manière plus fine. Enfin, nous présenterons un nouveau modèle d'apprentissage. La seconde partie (chapitres 4 à 7) concerne notre travail de développement d'une plate-forme générique pour l'expérimentation sur l'inférence. Cette plate-forme, appelée <PEpIn> (**P**late-forme d'**E**xpérimentations **p**our l'**In**férence) est une librairie implantant l'ensemble des objets définis dans la théorie de l'apprentissage automatique (concepts, mesures de probabilité, exemples, échantillons etc...) ainsi que les dépendances existantes entre ceux-ci. Cette librairie permet de construire des applications utilisant de tels objets. Une telle application, PEpIn<LR>, est ensuite présentée; il s'agit d'un atelier d'expérimentations d'algorithmes d'apprentissage des langages réguliers.

Chapitre 1

Le premier chapitre présente en détail l'ensemble des définitions relatives à la théorie de l'apprentissage. Trois modèles d'apprentissage sont ensuite présentés: le modèle de Gold, le modèle de Valiant et le modèle de Li et Vitányi. Le modèle de Gold voit l'apprentissage comme un acte infini. Dans ce modèle, l'apprentissage doit être exact à la limite, c'est-à-dire qu'il doit exister une étape à partir de laquelle l'apprenant retourne exactement le concept cible comme hypothèse. Le modèle de Valiant intègre l'idée d'apprentissage approximatif, c'est-à-dire qu'il peut exister des points de divergence entre l'hypothèse inférée et la cible. Enfin, le modèle de Li et Vitányi se base sur le modèle de Valiant en intégrant le principe consistant à privilégier les exemples les plus simples lors de l'apprentissage. Enfin, un exemple d'apprentissage d'une classe très simple dans chacun de ces modèles est proposé de manière à pouvoir comparer les différences entre ces trois modèles.

Chapitre 2

Dans ce chapitre, nous étudions l'apprentissage dans le domaine particulier des langages réguliers. L'apprentissage de grammaires (ou *inférence grammaticale*) occupe à lui seul un pan complet des recherches en matière d'apprentissage. L'ensemble de ce chapitre présente un certain nombre d'algorithmes connus d'inférence des langages réguliers (*e.g.* `ZR`, `RPNI` ...). Parallélement, nous étudions dans ce chapitre l'existence d'échantillon caractéristique adapté à chacun de ces algorithmes. Nous rappelons dans un premier temps les définitions relatives à la théorie des langages formels ainsi qu'un certain nombre de résultats concernant les langages réguliers. L'inférence des langages réguliers à la limite est ensuite étudiée en détail. Le cas de l'apprentissage par exemples positifs et négatifs est considéré dans un premier temps. Ensuite, nous étudions le cas de l'apprentissage par exemples positifs seuls, ce qui donne l'occasion de détailler les problèmes de sur-généralisation. La classe des langages réguliers n'étant pas Gold-apprenable par exemples positifs seuls, nous étudions l'apprenabilité d'une sous-classe de celle-ci : la classe des langages réversibles. L'apprentissage des langages réguliers dans le modèle PAC fait l'objet de la dernière partie de ce chapitre. Cette étude montre les résultats très forts de non PAC-apprenabilité de la classe des langages réguliers. Une variante du modèle de Valiant, proposée par Angluin, est alors présentée de manière à outrepasser cette difficulté. Dans ce nouveau modèle, l'apprenant dispose d'un oracle supplémentaire l'aidant dans son apprentissage.

Chapitre 3

Ce chapitre présente l'apport de notre travail théorique en matière d'apprentissage. Nous proposons, dans celui-ci, un nouveau modèle d'apprentissage, appelé modèle PACS. Ce modèle, inspiré du modèle de Li et Vitányi, intègre l'idée d'exemples simples par rapport au concept à apprendre. La notion de simplicité d'exemple n'est plus « absolue », comme c'est le cas dans le modèle de Li et Vitányi, mais « relative » au concept cible. Ce modèle s'appuie sur la complexité de Kolmogorov relative à un objet et, plus précisément, sur la distribution de Solomonoff-Levin relative à un objet. Cette mesure de probabilité favorise les exemples pouvant être facilement décrit lorsque l'on connaît le concept cible. Il s'agit, la plupart du temps, d'exemples bien choisis, c'est-à-dire permettant d'aider au mieux l'apprenant dans son apprentissage. Quelques échantillons caractéristiques présentés au chapitre 2 présentent la propriété de simplicité, c'est-à-dire que chacun des exemples les constituant a une grande probabilité d'être tiré. Cette propriété est utilisée de manière à prouver la PACS-apprenabilité de la classe des langages réversibles. Nous montrons également que la classe des formules DNF est apprenable dans ce modèle. Enfin, la comparaison de ce modèle avec d'autres modèles proches (c'est-à-dire incluant l'idée d'exemples caractéristiques) est évoquée ainsi qu'une étude succincte du problème de collusion dans ce modèle. Nous montrons, en effet, que certaines classes, telles que la classe des langages réguliers, sont apprenables dans notre modèle au moyen d'un algorithme collusif.

Chapitres 4 et 5

Les deux premiers chapitres de la deuxième partie présentent en détail les objectifs de la plate-forme <PEpIn> (**P**late-forme d'**E**xpérimentations **p**our l'**In**férence). Il s'agit d'une plate-forme générique permettant de concevoir et d'implanter des algorithmes d'apprentissage selon divers modèles et dans n'importe quel contexte. Le chapitre 4 propose une étude détaillée du cahier des charges de ce projet ainsi que le modèle conceptuel établi à partir de celui-ci. Le chapitre 5, plus technique, présente en détail l'implantation qui a été faite de <PEpIn>. Quelques-uns des problèmes techniques inhérents à l'implantation d'un tel projet sont évoqués. Une section particulière présente nos choix pour implanter l'« aléatoirité » dans notre plate-forme, c'est-à-dire la définition et l'implantation des distributions de probabilité. Les différentes techniques de calcul d'erreur sont également étudiées, tant au niveau théorique qu'au niveau implantation que nous en avons faite. Enfin, la dernière partie s'attache à présenter, de manière détaillée, le moteur du système (*run-time*) en incluant la description des possibilités offertes par celui-ci.

Chapitre 6

Le chapitre 6 est la suite logique des deux précédents chapitres. Dans celui-ci, nous étudions de quelle manière la plate-forme <PEpIn> a été utilisée afin de créer une application d'expérimentations sur l'inférence des langages réguliers, nommée PEpIn<LR>. Dans un premier temps, la conception et l'implantation du contexte des langages réguliers sont présentées. Il s'agit de modéliser et d'implanter l'ensemble des objets relatifs à la théorie des langages réguliers, ce qui donne lieu à la création d'une bibliothèque appelée CLR(**C**ontexte des **L**angages **R**éguliers). Ce travail est fait indépendamment de l'objectif final d'intégration de ce contexte à la plate-forme <PEpIn>. La partie suivante présente la conception proprement dite de l'application PEpIn<LR>. Il s'agit alors de mettre en correspondance les deux bibliothèques

<PEpIn> et CLR, c'est-à-dire d'intégrer le contexte particulier des langages réguliers à la plate-forme d'apprentissage. Ce chapitre peut être vu non seulement comme la présentation de la conception d'une application particulière au dessus de <PEpIn>, effectivement réalisée par notre équipe, mais également comme une « recette » générale, applicable par chacun en vu de créer de telles applications pour d'autres contextes (nos propres travaux validant cette recette).

Chapitre 7

Le dernier chapitre présente de manière concrète l'application PEpIn<LR>. Cette application permet de lancer des expériences d'apprentissage sur les langages réguliers. Dans ce chapitre, nous présentons de quelle manière cette application peut être paramétrée, ce qu'elle permet de faire et quels types de résultats elle retourne. Enfin, nous proposons un exemple de problème théorique d'apprentissage pour lequel une utilisation de l'application PEpIn<LR> pourrait aider le chercheur dans son travail.

Première partie

Théorie de l'Apprentissage Automatique

Chapitre 1

Présentation de l'Apprentissage

Ce premier chapitre présente les bases de la théorie de l'apprentissage. L'ensemble des termes qui seront employés dans la suite de cette thèse sont définis dans la section 1.1 . Une brève description d'un cas d'apprentissage « réel » vient illustrer ces définitions. Les trois sections qui suivent présentent les trois modèles majeurs de la théorie de l'apprentissage ; il s'agit des modèles de Gold, de Valiant et de Li et Vitányi. Les termes classiques de l'apprentissage automatique seront replacés dans le contexte propre à chaque paradigme. Nous donnerons ensuite notre propre point de vue sur chacun de ces modèles ainsi que quelques pistes et critiques qui ont contribué à les faire évoluer. Enfin, la dernière section sera l'occasion d'étudier, dans chacun des modèles, la résolution d'un même problème simple d'apprentissage. Cette étude original, imaginée par notre équipe permettra d'une part, de comprendre les principaux mécanismes qui participent au fonctionnement de chaque modèle et d'autre part de mettre en évidence les différences majeures qui existent entre ces trois modèles.

1.1 Généralités

1.1.1 Idées intuitives

Le type d'apprentissage auquel nous nous intéressons est l'apprentissage à partir d'exemples. Deux entités sont généralement en présence. Le maître (ou *enseignant*) qui a la connaissance (ou *concept cible*) et qui désire la partager avec son élève (ou *apprenant*). La méthode pédagogique du maître ne consiste pas à donner directement les différents constituants de sa connaissance mais à les faire découvrir par son élève. Ainsi, le maître va proposer à son élève un certain nombre d'*exemples* en qualifiant chaque exemple de *positif ou négatif* suivant son appartenance ou non au concept cible[1]. L'ensemble de ces exemples forme un *échantillon*. À partir de cet échantillon, l'élève forge sa propre connaissance et propose alors un *concept hypothèse*. Il lui sera dès lors possible de décider lui-même de l'appartenance ou non d'un exemple au concept, en utilisant la représentation qu'il s'en est faite.

Peut-on dire que l'élève a appris? Cela dépend de ce que l'on attend de lui, c'est-à-dire de ce que l'on considère comme un apprentissage réussi. Ainsi, on peut s'attendre à ce que l'élève propose exactement le même concept que celui du maître : on parle alors d'*apprentissage exact*. On peut aussi être plus tolérant et considérer que l'apprentissage a abouti lorsque le concept proposé est suffisamment « proche » du concept cible. L'ensemble des règles fixant

1. Remarquons qu'il s'agit donc essentiellement de problèmes de classification.

la manière dont l'enseignant doit proposer les exemples à son étudiant (peut-il les choisir au hasard, doit-il présenter des exemples particuliers ...) ainsi que celles déterminant la réussite ou l'échec de l'apprentissage est établi indépendamment du maître et de l'élève et ce avant que le cours ne débute. Ces « directives ministérielles » constituent le *modèle d'apprentissage*.

Remarquons que l'apprenant peut connaître *a priori* un certain nombre d'informations. Par exemple, il sait toujours quel est le sujet du cours auquel il participe ; ainsi, il ne proposera pas un concept *de nihilo* mais faisant bien partie du domaine concerné (*la classe des concepts* possibles).

1.1.2 Définitions formelles

La présentation qui précède nous a permis d'introduire le vocabulaire de base de la théorie de l'apprentissage. Nous replaçons ici ce vocabulaire dans un cadre formel afin de le définir plus précisément.

Par la suite, on notera $\mathbb{B}$ l'ensemble des booléens, c'est-à-dire $\mathbb{B} = \{0, 1\}$. De plus, on notera Σ un alphabet, tel que défini dans la théorie des langages (voir section 2.1.1, page 52). Sauf indication contraire, Σ sera supposé être un alphabet à deux lettres, disons $\Sigma = \{0, 1\}$.

Soit X un ensemble dénombrable que l'on appellera *domaine* (*sample space*). Un *concept* f est un sous-ensemble de X. Une *classe de concepts* F est un ensemble de concepts. En général, une classe de concepts regroupe des concepts ayant des propriétés communes. Un *exemple* ou *item* est un élément de X. Un *exemple labellé d'un concept* f est une paire $e = (x, b)$ de $X \times \mathbb{B}$ avec $b = 1$ si $x \in f$ (on dira alors que x est un *exemple positif* de f) et $b = 0$ sinon (x est alors appelé *exemple négatif* ou *contre exemple* de f). Un *échantillon labellé* est une suite[2] d'exemples labellés.

Il peut être pratique, dans certains cas, de passer facilement d'un échantillon labellé d'exemples à l'ensemble des exemples débarrassés de leur label. La définition suivante permet ce passage.

Définition 1.1.1 *Soit $S \subset X \times \mathbb{B}$ un échantillon d'exemples labellés. L'ensemble S_+ (resp. S_-) des exemples positifs (resp. négatifs) de S est l'ensemble défini par $S_+ = \{u \in X \mid \exists\, e \in S$ avec $e = (u, 1)\}$ (resp. $S_- = \{u \in X \mid \exists\, e \in S$ avec $e = (u, 0)\}$).*

Remarque 1.1.1 Par abus d'écriture et par soucis de simplification, on pourra écrire $S = S_+ + S_-$ pour signifier que S est l'échantillon dont tous les exemples labellés positifs sont construits à partir des éléments de S_+ et tous les exemples labellés négatifs sont construits à partir des éléments de S_-. De plus, lorsque cela ne posera pas d'ambiguïté, on utilisera le terme *exemple* à la place d'*exemple labellé*.

Définition 1.1.2 *Un concept f est* consistant *(ou* compatible*) avec un échantillon S si tous les exemples labellés positifs dans S sont positifs pour f et si tous les exemples labellés négatifs dans S sont des contre-exemples de f.*

Jusqu'à présent, on parle de concept en tant que sous-ensemble fini ou non d'un ensemble X. Dans une situation d'apprentissage cependant, on manipule rarement directement un concept mais on y fait plutôt référence au moyen d'un ou plusieurs *noms*. Un *nom* est une

2. Remarquons que dans une telle suite, un même exemple peut être rencontré plusieurs fois. De plus, l'ordre des éléments de la suite peut, dans certains cas, avoir son importance.

chaîne de Σ^*. Un *ensemble de représentations* R pour une classe F de concepts est un ensemble d'affectations, non obligatoirement unique, de noms à chaque concept de F. Pour tout concept $f \in F$, $R(f)$ est l'ensemble des noms identifiant le concept f avec $R(f) \neq \emptyset$. Si f et f' sont deux concepts distincts, $R(f) \cap R(f') = \emptyset$. On notera la longueur minimale d'un nom pour un concept f par $l_{min}(f)$, c'est-à-dire $l_{min}(f) = min\{|r| \mid r \in R(f)\}$. Soit n un entier, il sera utile de considérer l'ensemble F^n des concepts ayant une représentation de longueur au plus n, c'est-à-dire l'ensemble $F^n = \{f \in F \mid l_{min}(f) \leq n\}$. De plus, on supposera que la représentation R est calculable en temps polynomial, c'est-à-dire qu'il existe un algorithme déterministe en temps polynomial qui, étant donnée en entrée une paire (x, r) de $X \times R$, décide si x est un exemple de f avec $r \in R(f)$. Cette hypothèse est primordiale dans la mesure où chaque concept sera essentiellement manipulé *via* une de ses représentations. En général, il existe un ensemble de représentations R naturellement associé à une classe de concepts F ; cet ensemble R sera le plus souvent construit au moyen d'une méthode de codage des concepts de la classe F.

L'ensemble de représentations est d'une importance capitale car c'est dans celui-ci que l'apprenant va rechercher ses hypothèses. En général, on supposera que l'ensemble des représentations R est strictement associé à la classe des concepts F c'est-à-dire que tout nom de R représente un concept de F. Cependant, dans certains cas, on peut envisager de travailler dans un espace de représentations permettant de représenter plus de concepts. Dans ce cas, la classe de concepts H implicitement associées à R, appelée *classe d'hypothèses*, est telle que $H \supseteq F$. Il est important de comprendre qu'un apprenant travaillant dans une classe d'hypothèses plus grande que la classe de concepts dispose d'un atout supplémentaire, puisqu'il a à sa disposition un choix plus large de concepts. Dans la suite, et sauf cas particuliers, nous supposerons que $H = F$.

1.1.3 Algorithmes d'apprentissage *versus* Algorithmes de consistance

Il convient de présenter un peu plus en détail ce qu'est un *algorithme d'apprentissage*. Donner une définition précise d'un tel objet est, à ce stade, impossible. En effet, le but d'un tel algorithme est de retourner, à partir d'un certain nombre d'exemples relatifs à un concept cible, un « concept admissible ». Il faut, par conséquent, définir avant tout ce que signifie « concept admissible » ; or, c'est le modèle d'apprentissage qui fixe les règles d'apprentissage et qui détermine à quelles exigences doivent répondre les concepts retournés pour être qualifiés « d'admissibles ». On parlera donc d'algorithme d'apprentissage selon un modèle donné.

Nous pouvons cependant dresser une liste de propriétés que l'on est en droit d'attendre d'un tel algorithme. Certaines de ces propriétés se retrouveront dans la définition d'algorithme d'apprentissage des modèles qui seront présentés par la suite.

- **Rapidité** : étant donné un échantillon, l'algorithme doit retourner un résultat convenable (pour le modèle) en un temps raisonnable par rapport à et échantillon.

- **Généralisation** : le résultat retourné doit avoir un pouvoir de généralisation et non pas simplement retourner les exemples de l'échantillon[3]. Il doit ainsi être en mesure de bien classer des éléments non encore vus.

- **Caractérisation** : l'algorithme d'apprentissage doit être capable de dégager les caractéristiques communes des exemples de l'échantillon afin de construire son résultat.

3. Ce qui correspondrait à un apprentissage *par cœur*.

- **Robustesse** : Une petite modification de l'échantillon passé à l'algorithme ne devrait pas dégrader l'apprentissage.

- **Compréhension** : un algorithme d'apprentissage devrait permettre de comprendre les principes qui mènent à opter pour tel concept plutôt que pour tel autre.

Ces caractéristiques se basent principalement sur des idées intuitives que l'on peut se faire de l'apprentissage naturel. Certaines d'entre elles (caractérisation, compréhension) sont, à l'heure actuelle, difficilement mesurables et ne sont pas toujours prises en compte dans l'établissement d'algorithmes d'apprentissage. Néanmoins, nous pensons qu'écrire de « vrais » algorithmes d'apprentissage dans lesquels l'aspect cognitif prendrait la place qui lui revient, devrait inclure l'ensemble de ces exigences. Remarquons cependant que certaines techniques d'apprentissage (au moyen d'arbres de décision par exemple) tendent à intégrer ces dimensions.

S'il est impossible, à ce stade, de définir clairement ce que peut être un algorithme d'apprentissage, il est par contre possible de définir précisément la notion d'*algorithme de consistance*. Ce type d'algorithmes s'est imposé chez de nombreux auteurs [BEHW87, KLPV87] comme étant une première étape d'algorithme d'apprentissage. De plus, ce type d'algorithmes trouvera un intérêt dans l'établissement de quelques résultats.

Définition 1.1.3 *Soit F une classe de concepts. Un algorithme* `B` *prenant en entrée un échantillon d'exemples et retournant un concept de F compatible avec S est appelé* algorithme de consistance *pour F.*

Un algorithme de consistance ne peut donc pas, en général, être qualifié d'algorithme d'apprentissage puisque un algorithme se contentant (quand c'est possible) de retourner l'échantillon d'entrée codé sous la forme d'un concept est un algorithme de consistance.

Prouver qu'il existe un algorithme d'apprentissage selon un modèle pour une classe de concepts F consistera souvent à montrer qu'un algorithme de consistance connu est un algorithme d'apprentissage selon le modèle.

1.1.4 Un exemple d'apprentissage naturel

Nous proposons ici un exemple d'apprentissage « naturel » dans lequel chacun des termes que l'on vient de présenter trouvera facilement sa place. Supposons qu'un professeur de mathématiques veuille enseigner le concept de carré à ses élèves. Plus précisément, son but est de leur faire découvrir les différentes propriétés des carrés et ainsi leur permettre d'identifier, par la suite, toute forme géométrique carrée comme étant effectivement un carré. Le domaine est, par exemple, l'ensemble infini de toutes les figures géométriques d'un repère orthonormé donné. On peut facilement dégager plusieurs concepts dans ce domaine : la classe de tous les rectangles, celle des cercles, celle des carrés etc.... Supposons que notre enseignant limite son champ d'étude à la classe des parallélogrammes lors de l'apprentissage. Ainsi, les élèves disposent d'une connaissance *a priori* qui est : « les objets du concept à identifier sont des polygones à quatre cotés égaux et parallèles deux à deux ». Une représentation possible pour chaque concept de cette classe est, par exemple, la liste de toutes les propriétés remarquables des éléments du concept :
`parallélogrammes_à_quatre_côtés_égaux`,
`parallélogrammes_à_quatre_angles_droits`,

parallélogrammes_à_quatre_côtés_égaux_et_à_quatre_angles_droits ou plus simplement le nom des figures géométriques qualifiant chaque concept[4] : losange, rectangles, carrés, etc... Notons qu'il existe un algorithme en temps polynomial permettant de décider de l'appartenance d'un exemple à un concept (il suffit de vérifier que les propriétés du concept considéré sont vérifiées). L'enseignant va proposer aux élèves un ensemble de figures géométriques en qualifiant d'exemples positifs toutes celles présentant les propriétés du carré, les autres étant les exemples négatifs. Cet ensemble de figures forme l'échantillon d'apprentissage. Vraisemblablement, les élèves vont facilement découvrir qu'un carré est « un parallélogramme (connaissance *a priori*) dont les quatre côtés sont égaux et dont les quatre angles sont droits ». Dans ce cas, l'apprentissage est dit exact. Si nos élèves ne sont pas très doués (ou bien encore si notre enseignant n'est pas très pédagogue!), peut-être que seule l'une des deux propriétés sera découverte par les étudiants ; par exemple, un carré est « un parallélogramme dont les quatre angles sont droits ». Dans ce cas, ils identifieront effectivement les carrés comme tels, mais ils classeront aussi les rectangles comme étant des carrés. Nous sommes en présence d'un apprentissage approximatif. Selon que le modèle d'apprentissage autorise ou non un apprentissage approximatif et suivant la limite de l'approximation autorisée, le deuxième cas pourra être ou non qualifié de succès.

Dans la suite, nous présentons trois modèles particuliers de la théorie de l'apprentissage. Ceux-ci fixent l'ensemble des modalités permettant de définir l'apprentissage. Parmi celles-ci, on trouve les règles déterminant le succès d'un apprentissage.

1.2 Modèle de Gold : Identification à la Limite

En 1967, E.M Gold propose l'un des premiers modèles théoriques d'apprentissage automatique [Gol67]. Bien que très éloigné, par certains côtés, d'une quelconque modélisation de l'apprentissage naturel[5], ce modèle présente entre autre l'avantage d'être très simple. C'est certainement l'une des raisons pour laquelle ce modèle a été et est encore tellement étudié et décliné de nombreuses manières.

1.2.1 Présentation

Le modèle de Gold (ou *identification à la limite*) voit l'inférence inductive comme un processus infini[6]. Le critère de succès d'un apprentissage est son comportement à la limite. Ce type d'apprentissage est dit exact, ce qui signifie que lorsqu'un concept est identifiable à la limite, il existe une étape du processus d'apprentissage à partir de laquelle l'apprenant proposera exactement le concept attendu.

Dans ce modèle, il n'existe aucune contrainte sur la façon dont les exemples seront présentés si ce n'est que tout exemple du domaine doit être proposé au moins une fois à l'apprenant. Cette contrainte est prise en compte dans la définition de présentation complète d'exemples :

Définition 1.2.1 *Soit X un domaine. Une présentation infinie d'exemples $\sigma = \{e_1, e_2, e_3, \ldots\}$ est dite* complète *si $\forall\, x \in X, \exists\, t \geq 0$ tel que $e_t = (x, b)$ avec $b \in \mathbb{B}$.*

4. Ces noms ne sont finalement que des raccourcis de langage pour signifier la liste des propriétés attachées à chaque type de figure.

5. Certains considèrent ce paradigme comme une formalisation de l'« apprentissage de toute une vie ».

6. Bien que l'apprentissage doit avoir lieu en un temps fini.

Un des protocoles d'apprentissage naturellement associé à ce modèle est celui dit *incrémental*. Il s'agit d'un processus infini d'étapes successives. À chaque étape i, l'enseignant fournit à l'apprenant un exemple labellé pour un concept cible. L'apprenant propose alors une hypothèse qui prendra en compte l'ensemble des exemples déjà présentés. Formellement,

Définition 1.2.2 *Soit F une classe de concepts et $f \in F$ le concept cible. Soient $\sigma = \{e_1, e_2, e_3, \ldots\}$ une présentation complète d'exemples et* A *un algorithme prenant à chaque étape du processus un exemple labellé e_i et sortant un concept hypothèse f_i. Si $\exists\ t_0 \geq 0$ tel que $\forall\ t \geq t_0, f_t = f_{t_0}$ et $f_{t_0} = f$ alors* A identifie f à la limite.

L'identification se fait effectivement en un nombre fini d'étapes puisqu'à partir de l'étape t_0, l'algorithme se stabilise sur un concept qui est **exactement** la cible. Cependant, il lui est impossible de savoir qu'il est définitivement stabilisé sur le concept cible. En effet, imaginons la situation dans laquelle un concept f' est égal au concept cible f sauf pour un seul exemple e. Dans ce cas, il se peut que l'algorithme se stabilise un très grand nombre d'étapes sur le concept f' ; c'est seulement lorsqu'il se verra présenter l'exemple e qu'il modifiera sa réponse pour se stabiliser définitivement sur f.

Définition 1.2.3 *Soit F une classe de concepts. S'il existe un algorithme* A *qui identifie à la limite tous les concepts de F, alors la classe F est dite* identifiable à la limite.

1.2.2 Identification par énumération

L'*identification par énumération* est une procédure d'apprentissage qui peut être appliquée à certaines classes identifiables à la limite. Elle assure que tout concept de la classe sera appris exactement.

Cette procédure consiste à rechercher systématiquement une hypothèse dans l'espace de tous les concepts possibles. Une hypothèse est sélectionnée lorsqu'elle est en adéquation avec l'échantillon courant (c'est-à-dire l'ensemble des exemples déjà présentés). Lorsqu'un nouvel exemple est fourni à l'apprenant, celui-ci vérifie que l'hypothèse courante est compatible avec le nouvel exemple. Dans le cas négatif, une nouvelle recherche est lancée afin de trouver une hypothèse plus adaptée.

Le théorème de Gold [Gol67] établit que toute classe énumérable de concepts est identifiable à la limite (il suffit d'utiliser la méthode d'identification par énumération).

1.2.3 Quelques résultats

Nous proposons ici quelques résultats connus sur l'apprenabilité de classes de concepts dans le modèle de Gold. Ces résultats sont essentiellement des conséquences du théorème de Gold.

La classe PR des fonctions primitives récursives est identifiable à la limite (puisqu'elle est énumérable). Il en est de même pour la classe des langages réguliers $\mathcal{R}eg$ (voir chapitre 2, page 51), celle des langages hors-contexte ou bien encore celle des langages contextuels.

La classe R des fonctions récursives totales n'est pas énumérable, on ne peut donc pas appliquer le théorème de Gold sur celle-ci ; il se pourrait cependant que cette classe soit tout de même identifiable à la limite. En fait, Gold montre que cette classe ne l'est pas.

1.2.4 Cas de l'apprentissage par exemples positifs seuls

Le modèle de Gold tel que nous venons de le présenter considère que tout exemple (qu'il soit positif ou non) est présenté à l'apprenant au moins une fois pendant le processus. Aucune autre hypothèse n'est faite quant à la présentation d'exemples.

Il existe un autre cadre d'apprentissage dans lequel seuls des exemples positifs sont utilisés. Ce type d'apprentissage trouve son intérêt pour modéliser des cas réels (*e.g.* apprentissage du langage naturel) dans lesquels l'être humain n'est confronté qu'à des instances positives du concept à inférer. Dans ce cas, l'apprentissage est, en général, beaucoup plus difficile puisqu'alors nous ne disposons plus des exemples négatifs, susceptibles d'éliminer des hypothèses incorrectes. On peut être alors confronté au problème de la *sur-généralisation* ; le concept inféré est compatible avec l'échantillon d'exemples positifs présenté, il généralise effectivement cet échantillon mais il généralise trop. Dans un cas extrême, on peut imaginer que l'algorithme d'apprentissage retourne comme hypothèse le domaine lui-même.

La notion de présentation complète d'exemples positifs est le pendant de celle de présentation complète d'exemples restreinte aux exemples positifs :

Définition 1.2.4 *Soit f un concept de 2^X. Une présentation d'exemples positifs $\sigma = \{e_1, e_2, e_3, \ldots\}$ de f est* complète *si $\forall\ x \in f, \exists\ i \geq 0$ tel que $e_i = (x, 1)$.*

Dans ce type d'apprentissage, le nombre de classes prouvées identifiables à la limite est, par là même, notablement affaibli. Ainsi, Gold a montré [Gol67] que toute classe *non superfinie* c'est-à-dire contenant tous les concepts finis du domaine et au moins un concept infini n'est pas identifiable à la limite par exemples positifs seuls.

1.2.5 Modèles dérivés du modèle de Gold

Le modèle de Gold a donné lieu à un grand nombre d'études théoriques et de nombreux modèles dérivent de celui-ci. Ceux-ci diffèrent du modèle d'origine sur quelques points en affaiblissant ou en contraignant certaines caractéristiques du modèle de Gold. Il est, par exemple, possible de choisir d'autres critères de succès pour l'inférence. Dans l'*identification comportementale*, par exemple, on demande simplement que l'hypothèse converge sémantiquement (et non plus syntaxiquement) à la limite. Ainsi, à partir d'une certaine étape, l'hypothèse peut changer indéfiniment à condition que toutes les hypothèses choisies soient des descriptions correctes du concept sous-jacent. Un autre critère de succès possible consiste à autoriser l'hypothèse finale à être égale au concept cible sauf pour un nombre fini d'exemples ; ce type d'inférence est appelé *identification à la limite presque partout.* Cette variante laisse percevoir l'idée qu'il peut être plus « naturel » de ne pas exiger de l'apprenant un apprentissage exact mais plutôt approximatif. Cette idée sera reprise dans le modèle de Valiant. D'autres déclinaisons du modèle de Gold sont présentées en détail dans [AS83] ou encore dans le chapitre 5 de [Del93].

1.2.6 Point de vue sur ce modèle

Le modèle de Gold, en tant que premier modèle théorique de l'apprentissage, a le mérite de poser les premiers jalons formalisant l'apprentissage. De part sa simplicité (tant au niveau de son énonciation, que de sa mise en œuvre), ce modèle présente l'avantage d'être facilement utilisable. De plus, le théorème de Gold donne une condition suffisante pour qu'une classe soit

apprenable. Cependant, plusieurs défauts l'empêchent d'être un bon candidat pour formaliser l'apprentissage naturel en général. Le problème principal réside dans l'indécidabilité de l'arrêt ; à aucun moment, un apprenant peut dire qu'il a atteint un niveau suffisant d'apprentissage. De plus, l'exigence d'un apprentissage exact semble, elle aussi, très éloignée de l'apprentissage naturel[7]. Enfin, le fait que toute présentation d'exemples doit donner lieu à un apprentissage entraîne une certaine indépendance entre exemples et concept cible, indépendance qui ne semble pas être le propre de l'apprentissage naturel.

Le modèle de Valiant que l'on présente dans la section suivante pallie la plupart de ces inconvénients, au détriment de la simplicité du modèle.

1.3 Modèle de Valiant

1.3.1 Présentation informelle

En 1984, Valiant a proposé dans [Val84] un modèle dans lequel les défauts du modèle de Gold n'apparaissent plus. Ce modèle, aussi connu sous le nom d'*apprentissage PAC* (pour Probablement Approximativement Correct), considère que l'apprentissage d'un concept doit se faire avec une grande probabilité (bien que l'on considère que l'échec est quelquefois possible), avec une certaine erreur (le concept peut être mal appris), en temps polynomial et avec un nombre polynomial d'exemples.

Au moins deux de ces caractéristiques correspondent à l'idée intuitive que l'on peut se faire de l'apprentissage naturel : l'apprentissage doit être rapide et il est suffisant d'apprendre approximativement, c'est-à-dire de donner une hypothèse qui soit acceptable dans la plupart des cas.

De plus, la caractéristique selon laquelle l'apprentissage peut, dans certains cas, ne pas aboutir trouve, elle aussi, une interprétation dans l'apprentissage naturel. En effet, si l'apprenant est confronté à un mauvais pédagogue, c'est-à-dire un enseignant ne donnant que des exemples sans grand rapport avec sa connaissance du concept, l'apprentissage n'a que peu de chance d'aboutir.

Une présentation détaillée du modèle PAC peut être trouvée dans [KLPV87], [Nat91] ou [KV92]. D'autres modèles dérivés du modèle de Valiant sont proposés dans [BI91], [KLPV87] et [Nat87].

1.3.2 Présentation formelle du modèle

Nous définissons ici, de manière formelle, l'apprentissage PAC dans des domaines dénombrables tel qu'il a été présenté dans [Val84].

Soit X un domaine, $F \in 2^X$ une classe de concepts et $f \in F$ un concept de cette classe. R est un ensemble de représentations pour la classe F. Soit n un entier, F^n est l'ensemble des concepts de F ayant une représentation de longueur au plus n.

Dans le modèle PAC, le domaine X est muni d'une distribution de probabilité P. Un *oracle d'exemples* est une « boîte noire » fournissant des exemples labellés (relativement à un concept f), ces exemples étant tirés aléatoirement par l'oracle selon la distribution de probabilité P. On notera un tel oracle $Exemple_{X,P}(f)$. On suppose de plus que chaque appel à l'oracle coûte une unité de temps.

7. Qui peut se targuer, en effet, de connaître exactement sa langue maternelle?

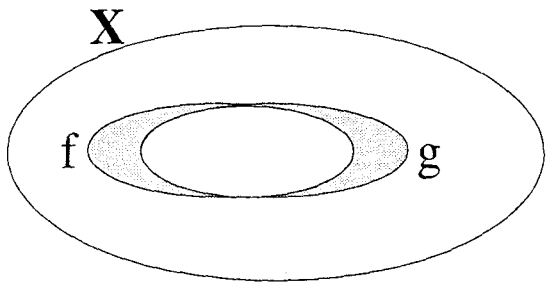


FIG. 1.1 - Erreur entre les deux concepts f et g.

Notation :
Soient E_1 et E_2 deux ensembles quelconques. La différence symétrique entre E_1 et E_2 est notée $E_1 \Delta E_2$. On rappelle qu'il s'agit de l'ensemble contenant tous les éléments n'appartenant qu'à un seul des ensembles E_1, E_2, c'est-à-dire : $E_1 \Delta E_2 = (E_1 \cup E_2) \backslash (E_1 \cap E_2)$.

La définition suivante permet d'introduire la notion d'erreur d'un concept par rapport à un autre. Intuitivement, l'erreur du concept g par rapport au concept f représente le poids de l'ensemble des éléments pour lesquels les deux concepts sont en désaccord.

Définition 1.3.1 *Soit X un domaine muni d'une distribution de probabilité P. Soient f et g deux concepts de 2^X. L'*erreur de g par rapport à f selon P, *notée $Erreur_{P,f}(g)$, est définie par :*

$$Erreur_{P,f}(g) = \sum_{x \in f \Delta g} P(x)$$

Si l'on représente graphiquement les deux concepts f et g par deux ensembles (voir figure 1.1), alors l'erreur entre f et g selon P est la probabilité qu'un élément de X tiré selon P, appartienne à l'ensemble grisée.

Définition 1.3.2 *Soit* A *un algorithme prenant en entrée un* paramètre de précision $\varepsilon \in]0,1]$ *et un* paramètre de confiance $\delta \in]0,1]$. A *est un algorithme de PAC-apprentissage pour la classe de concepts F associée à l'ensemble de représentations R si pour tout concept f de F et pour toute distribution de probabilité P (fixée et inconnue de* A*),* A *a recours à un oracle d'exemples $Exemple_{X,P}(f)$ et retourne la représentation d'un concept h de F telle que avec la probabilité au moins $1-\delta$, $Erreur_{P,f}(h) \leq \varepsilon$.*

Définition 1.3.3 *Une classe de concepts F est* PAC apprenable *dans l'ensemble des représentations R s'il existe un algorithme de PAC-apprentissage pour F associée à R.*

Le paramètre de précision ε borne l'erreur commise par l'hypothèse h. C'est ce paramètre qui donne au modèle une certaine liberté sur l'hypothèse à inférer et qui permet de passer de l'apprentissage exact du modèle de Gold à un apprentissage **approximatif.** Le paramètre de confiance δ fixe dans quelle mesure l'hypothèse produite peut être prise au sérieux. Ce paramètre indique que l'apprentissage, même approximatif, peut échouer, bien qu'il aboutisse **probablement.** Il est à noter que la mesure de la qualité de l'apprentissage (c'est-à-dire l'évaluation de l'erreur commise par l'hypothèse) est faite au moyen de la même distribution de probabilité que celle ayant servi à l'apprentissage. Ainsi, une « mauvaise » distribution de probabilité[8] entraînera un apprentissage de piètre qualité dans l'absolu mais de qualité pro-

8. C'est-à-dire une distribution de probabilité ne rendant pas bien compte des caractéristiques du concept.

bablement acceptable relativement à la distribution utilisée. Dans un apprentissage naturel, cette caractéristique imposerait à l'enseignant lui-même d'évaluer son élève, sous le couvert de ses propres connaissances. Le succès ou l'échec d'un apprentissage est donc toujours relatif à l'enseignement lui-même.

Remarquons que ce modèle d'apprentissage demande à l'algorithme d'apprendre sous toutes les distributions, même les plus invraisemblables ; cette exigence permet d'assurer une totale indépendance de l'algorithme d'apprentissage par rapport à la distribution des exemples. Notons encore que dans de nombreux cas, la classe F et l'ensemble des représentations R sont naturellement associés ; nous écrirons alors que A est un algorithme de PAC-apprentissage pour la classe F.

Jusqu'à présent, le modèle PAC n'impose aucune contrainte quant à l'efficacité de l'algorithme d'apprentissage. La définition qui suit permet d'assurer que l'apprentissage se fera en temps raisonnable, c'est-à-dire polynomial.

Définition 1.3.4 *Une classe de concepts F est* polynomialement PAC-apprenable *(poly-PAC-apprenable) dans R si elle est PAC apprenable dans R et si* A, *l'algorithme d'apprentissage, tourne en temps polynomial par rapport à $1/\varepsilon$, $1/\delta$, la longueur d'une plus petite représentation du concept cible et la longueur du plus long exemple vu par* A.

Remarquons d'abord que la définition 1.3.4 implique de fait que la taille de l'échantillon d'apprentissage (*i.e.* le nombre d'exemple de celui-ci) soit elle-même polynomiale. De plus, la longueur d'une plus petite représentation du concept cible doit être prise en compte dans le calcul de la borne du temps d'exécution de l'algorithme puisque celui-ci doit au moins disposer du temps nécessaire à la construction d'une telle représentation. De même, la longueur du plus long exemple présenté doit, elle aussi, être prise en compte pour des raisons similaires au niveau de la lecture des exemples. Hormis les longueurs d'exemple et de représentation, la présence des deux autres paramètres (précision et confiance) dans le calcul du temps d'exécution de l'algorithme s'explique par des raisons purement probabilistes. Intuitivement, si l'on désire un apprentissage de bonne qualité (c'est-à-dire avec une valeur de ε faible), il semble évident que l'algorithme demandera un échantillon de taille importante (et donc un temps de calcul plus long) afin d'atteindre la précision demandée. Il en est de même si l'on exige que l'apprentissage ait lieu souvent (valeur de δ proche de 0).

La figure 1.2 (page suivante) illustre graphiquement le modèle d'apprentissage poly-PAC.

Prouver la poly-PAC-apprenabilité (ou la non apprenabilité) d'une classe de concepts est soit un problème trivial (et l'algorithme d'apprentissage est alors lui-même trivial) soit un problème très difficile[9]. Il existe cependant un théorème donnant une condition suffisante pour qu'une telle classe soit poly-PAC-apprenable. Ce théorème, présenté dans la section suivante, doit être vu comme un outil permettant de simplifier, dans certains cas, les preuves de poly-PAC-apprenabilité d'une classe F.

1.3.3 Rasoir d'Occam

Le *rasoir d'Occam* est un principe « naturel » énoncé la première fois par le moine anglais William of Occam sous la forme « *Non sunt multiplicanda entia praeter necessitatem* »[10].

9. Il existe de nombreuses classes de concepts pour lesquelles les résultats de PAC-apprenabilité ne sont à ce jour, toujours pas prouvés.

10. « *Ne multipliez pas les objets si ce n'est pas nécessaire.* »

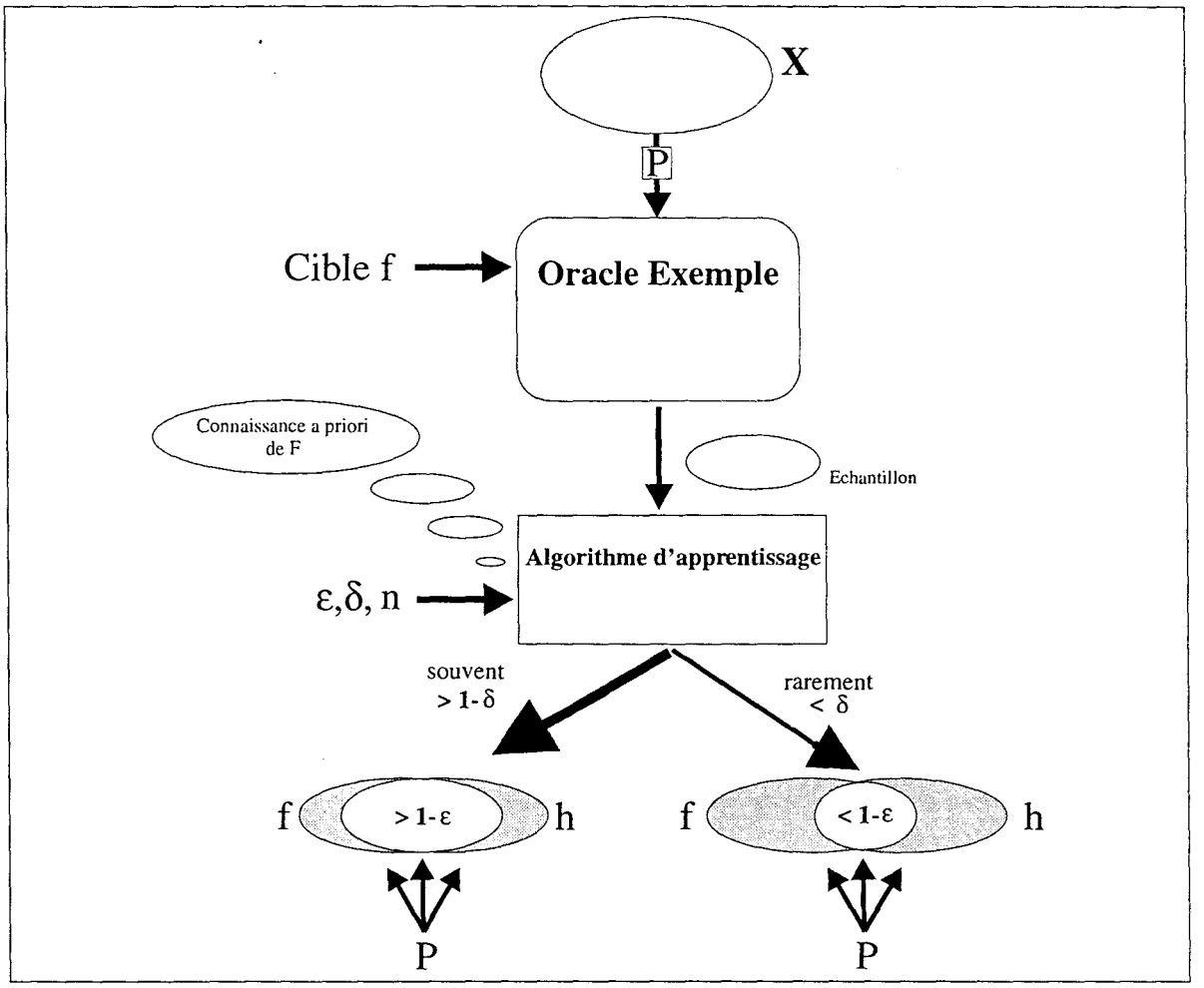


FIG. 1.2 - L'apprentissage poly-PAC.

Ce principe affirme que si plusieurs causes peuvent expliquer un même phénomène, il est préférable de ne retenir que la plus simple d'entre elles. Appliqué au contexte de l'inférence inductive [BEHW87], ce principe pose donc qu'il suffit de chercher le concept le plus simple[11] qui soit consistant avec l'échantillon. Trouver le plus petit concept consistant avec un échantillon donné peut cependant être un problème NP-dur ; par exemple trouver la formule DNF de longueur minimale consistante avec un échantillon ou bien le plus petit automate compatible avec un échantillon de mots labellés sont deux problèmes NP-durs [PV88, Gol78]. Aussi, pourra-t-on se contenter de ne rechercher qu'un concept suffisamment simple (et non le plus simple).

Algorithmes de poly-PAC-apprentissage et algorithmes de consistance

Avant de considérer le cas particulier d'algorithmes de consistance retournant un concept de description suffisamment courte, nous allons étudier le rapport qui existe entre les algorithmes de consistance généraux et l'apprentissage poly-PAC. Il s'agit d'une première étape

11. On prendra en général comme mesure de simplicité la longueur de la représentation.

avant d'aborder les algorithmes d'Occam qui assurent une propriété supplémentaire sur le résultat inféré.

Théorème 1.3.1 ([BEHW87]) *Soit F une classe dont tous les concepts ont une représentation de longueur n (n un entier). S'il existe un algorithme de consistance pour F alors la classe F est PAC-apprenable.*

Preuve :
Ce théorème se base sur l'idée qu'une mauvaise hypothèse, c'est-à-dire dont l'erreur est grande, peut être éliminée en tirant un nombre suffisant d'exemples. Dès lors, si le nombre de mauvaises hypothèses est faible, toutes peuvent être éliminées en tirant un nombre réduit d'exemples.

Supposons que B est un algorithme de consistance sur la classe F. Nous allons montrer que l'algorithme A qui suit est un algorithme de PAC-apprentissage pour la classe F.

Algorithme A
Entrée: ε, δ
Constante connue: n longueur maximal des représentations des concepts de F
Début
Calculer $N(n, \varepsilon, \delta)$ la taille de l'échantillon
Tirer un échantillon S de taille $N(n, \varepsilon, \delta)$ au moyen de l'oracle $Exemple_{X,P}(f)$
Sortir B(S)
Fin

Soit $f \in F$ le concept cible. On définit $\hat{H}_f$ comme étant l'ensemble des mauvaises hypothèses de F pour f , c'est-à-dire,

$$\hat{H}_f = \{g \in F \mid Erreur_{P,f}(g) > \varepsilon\}$$

Évidemment, la taille de $\hat{H}_f$ est inférieure ou égale à 2^n, le nombre de concepts de F (on suppose ici que l'alphabet utilisé pour représenter les concepts est de taille 2).

Nous allons montrer qu'en tirant un échantillon S de taille N suffisante, avec une grande probabilité, le concept retourné par B n'appartient pas à $\hat{H}_f$ et donc approxime f.

Soit g un concept de $\hat{H}_f$. La probabilité qu'un exemple x labellé pour f soit bien classé par g est inférieure ou égale à $1 - \varepsilon$. La probabilité que N exemples labellés pour f soient bien classés par g (c'est-à-dire que le mauvais concept g soit consistant avec un échantillon de taille N) est inférieure ou égale à $(1-\varepsilon)^N$. La probabilité qu'un mauvais concept (un élément de $\hat{H}_f$) soit consistant avec un échantillon de taille N est donc inférieure ou égale à $2^n(1-\varepsilon)^N$ puisque $|\hat{H}_f| \leq 2^n$. Nous voulons que la taille N de S soit suffisante pour que la probabilité de retourner un concept de $\hat{H}_f$ consistant avec S soit inférieure à δ. Il faut donc que,

$$2^n(1-\varepsilon)^N \leq \delta$$

En passant au logarithme naturel de chaque côté de l'inégalité,

$$n\ln(2) + N\ln(1-\varepsilon) \leq \ln(\delta)$$

Soit,

$$N \ln(1-\varepsilon) \leq \ln(\delta) - n\ln(2)$$

En utilisant l'approximation $\ln(1-\varepsilon) \leq -\varepsilon$, il suffit que,

$$-N\varepsilon \leq \ln(\delta) - n\ln(2)$$

En divisant de chaque côté par $-\varepsilon < 0$,

$$N \leq \frac{\ln(\delta) - n\ln(2)}{-\varepsilon}$$

Soit encore,

$$N \leq \frac{1}{\varepsilon}\left(\ln\left(\frac{1}{\delta}\right) + n\ln(2)\right)$$

Ainsi, en prenant un échantillon de taille suffisante (en $O(n, 1/\delta, 1/\varepsilon)$), le concept retourné par B et donc par A est, avec une probabilité supérieure à $1-\delta$, un concept approximant f. Ceci est valable quel que soit f un concept de F, ce qui prouve que l'algorithme A est un algorithme de PAC-apprentissage pour F et termine la preuve. □

Un algorithme de consistance tel que défini à la section 1.1.3 (page 27) est polynomial s'il retourne un résultat en un temps polynomial par rapport à la taille de l'échantillon.

Si l'algorithme de consistance du théorème 1.3.1 (page précédente) est polynomial alors la classe F devient évidemment poly-PAC-apprenable soit :

Corollaire 1.3.2 *Soit F une classe dont tous les concepts ont une représentation de longueur n. S'il existe un algorithme de consistance polynomial pour F alors la classe F est poly-PAC-apprenable.*

Les classes que nous venons d'étudier sont finies puisque chacun des concepts les constituant peuvent être codés par une représentation de longueur bornée. Remarquons cependant que bien que finie, une telle classe peut contenir un nombre non polynomial de concepts par rapport aux paramètres à prendre en compte. Cela interdit donc à l'algorithme de consistance de faire une recherche exhaustive dans l'ensemble des représentation.

Dans le cas de telles classes (finies) de concepts, l'existence d'un algorithme de consistance polynomial prouve donc la poly-PAC-apprenabilité de la classe. Cependant, les classes de concepts ne sont pas, en général, finies. La définition d'algorithme d'Occam contraint celle d'algorithme de consistance et permet ainsi d'étendre le théorème 1.3.1 (page précédente) au cas des classes infinies.

Algorithmes d'Occam

Les algorithmes de consistance polynomiaux permettent de construire la représentation d'un concept compatible avec un échantillon d'entrée. Cependant, le résultat qu'ils retournent peut n'être qu'une simple reformulation de l'échantillon d'entrée. Les algorithmes d'Occam assurent une propriété supplémentaire sur la longueur de description du concept retourné, à savoir que celui-ci est bien plus court que la taille de l'échantillon.

Définition 1.3.5 *Soit F une classe de concepts associée à l'ensemble de représentations R. Soit $n \geq 0$. Un algorithme* B, *prenant en entrée un échantillon S, avec $|S| = m$, dont tous les*

exemples sont de longueur au plus l, est un algorithme d'Occam polynomial pour (F, R) s'il existe deux constantes $c \geq 1$ et $\alpha \in [0, 1[$ telles que, en un nombre d'étapes polynomial par rapport à $m.l$, B *retourne la représentation r d'un concept de F s'il existe, compatible avec S et tel que : $|r| \leq n^c (m.l)^\alpha$.*

Un algorithme d'Occam polynomial n'est donc rien de plus qu'un algorithme de consistance retournant un concept court par rapport à la longueur de l'échantillon (somme des longueurs des exemples) et la taille minimale de représentation des concepts. Ce calcul se fait en temps polynomial par rapport à la longueur de l'échantillon ; en effet, comme dans le cas des algorithmes de PAC-apprentissage, B doit au moins lire chacun des exemples de l'échantillon.

Remarque 1.3.1 Dans ce cas, la longueur du concept retourné $|r|$ n'intervient pas puisque celle-ci est inférieure à la taille de l'échantillon.

La variable α, quant à elle, empêche de générer un concept qui ne serait qu'une énumération de l'échantillon lui-même.

Théorème d'Occam

Le théorème d'Occam tel qu'il est présenté ici diffère quelque peu de l'énoncé qui en est fait dans [BEHW87]. Ce théorème (qui peut être vu comme l'extension annoncée du théorème 1.3.1 (page 36)) est l'un des plus beaux résultats de la PAC-apprenabilité puisqu'il donne une condition suffisante pour qu'une classe F de concepts soit poly-PAC-apprenable.

Théorème 1.3.3 *Soit F une classe de concepts associée à l'ensemble de représentations R. S'il existe un algorithme d'Occam pour (F, R) alors la classe F est poly-PAC-apprenable.*

Idée de preuve :
La preuve de ce théorème s'inspire de celle donnée pour le théorème 1.3.1 (page 36) en utilisant le fait que l'ensemble des concepts ayant une représentation suffisamment courte est fini (ce qui permet d'appliquer un raisonnement similaire à celui précédemment utilisé). L'algorithme d'apprentissage est du même ordre que l'algorithme A introduit dans la preuve du théorème 1.3.1 (page 36), l'algorithme B étant ici un algorithme d'Occam. Précisons que pour certains domaines d'apprentissage[12], il se peut que la taille des exemples ne soit pas fixée *a priori* ; dans ce cas, il peut être utile d'introduire la taille du plus long exemple tiré en tant que paramètre de l'algorithme. Ce nouveau paramètre intervient alors dans le calcul de la taille de l'échantillon à tirer. Intuitivement, cela signifie que l'apprentissage requiert plus d'exemples si ceux-ci sont courts. □

Pour une étude plus approfondie du théorème d'Occam (et différente de celle qui en est faite dans [BEHW87]) voir aussi [KLPV87], chapitre 3 de [Nat91], chapitre 2 de [KV92] ou chapitre 5 de [LV93].

12. Tel que Σ^* par exemple, domaine d'apprentissage des langages.

1.3.4 Quelques résultats sur la PAC apprenabilité

Démontrer la poly-PAC apprenabilité d'une classe de concepts F est un problème difficile. La plupart des résultats de PAC-apprenabilité porte sur des classes de formules booléennes. Nous définissons brièvement quelques une de ces classes sur lesquelles il existe de tels résultats. D'autres classes de formules booléennes viendront compléter cette liste dans la suite de cette thèse.

Quelques classes de formules booléennes

Soit un ensemble de variables booléennes $\{x_1, x_2, \ldots, x_n\}$. Un *littéral* est soit le symbole x_i soit sa négation $\overline{x_i}$, avec $1 \leq i \leq n$. Un *monôme* est une conjonction de littéraux. Une *clause* est une disjonction de littéraux. Une *formule DNF* est une formule booléenne sous forme normale disjonctive, c'est-à-dire une disjonction de monômes (exemple : $x_1x_3x_5 \vee \overline{x_2}x_4\overline{x_6} \vee x_1\overline{x_3}$) ; dans une telle formule, les monômes sont appelés *termes*. Une *formule CNF* est une formule booléenne sous forme normale conjonctive, c'est-à-dire une conjonction de clauses (exemple : $(x_1 \vee x_3 \vee \overline{x_5}) \wedge (x_2 \vee x_6) \wedge \overline{x_1}$). Pour un k donné, une *formule k-DNF* est une formule DNF dans laquelle chaque terme contient au plus k littéraux (exemple : $(x_1 \vee x_3 \vee \overline{x_5}) \wedge (x_2 \vee x_6) \wedge \overline{x_1}$ est une 3-DNF). Une *formule k-term-DNF* est une formule DNF contenant au plus k termes ; il n'y a aucune restriction sur le nombre de littéraux de chaque terme.

L'ensemble des formules de chacun de ces types forme une classe ; par exemple la classe des DNF est l'ensemble des formules DNF.

Résultats de PAC-apprenabilité sur des classes de formules booléennes

La première classe qui a été montrée poly-PAC-apprenable est la classe k-CNF (k fixé), par exemples positifs seuls [Val84]. La preuve se base essentiellement sur des considérations probabilistes. L'utilisation d'un algorithme de consistance permet également de prouver ce résultat. La poly-PAC-apprenabilité de la classe k-DNF (k fixé) par exemples négatifs seuls a été prouvée de manière similaire.

De nombreux travaux ont été réalisés afin de prouver la poly-PAC apprenabilité de classes de formules booléennes ayant différentes caractéristiques. Dans de nombreux cas, le théorème d'Occam a joué un rôle primordial afin d'obtenir ces résultats.

Démontrer la non poly-PAC-apprenabilité d'une classe de concepts revient, dans bien des cas, à un problème de réduction. On peut citer comme exemple d'une telle preuve, le cas de la classe des k-terms-DNF (k fixé) qui n'est pas poly-PAC apprenable[13] sous la conjecture $RP \neq NP$ [PV88].

Remarque 1.3.2 Il est possible de montrer qu'il n'existe pas d'algorithme d'Occam polynomial pour la classe k-terms-DNF. Cependant, cela ne suffit pas à prouver que cette classe n'est pas poly-PAC-apprenable, le théorème d'Occam ne donnant qu'une condition suffisante et non une condition nécessaire pour la poly-PAC-apprenabilité. Il existe une réciproque à ce théorème, cependant celle-ci n'est utilisable qu'avec des classes de concepts vérifiant des contraintes supplémentaires[14] (à ce propos, on pourra se reporter à [BP90]).

13. Cette classe a cependant été montrée poly-PAC-apprenable dès lors que l'on autorise d'apprendre dans une classe d'hypothèses plus grande et en utilisant un oracle d'appartenance [BFJ+94].

14. Peu sévères en général.

1.3.5 Critiques et points de vue sur ce modèle

Comparé au modèle de Gold, le modèle de Valiant est un modèle beaucoup plus complexe tant au niveau de sa présentation que de son utilisation. Il permet de modéliser quelques caractéristiques propres à l'apprentissage naturel : l'apprentissage doit être relativement rapide, il peut échouer, il est approximatif.

De plus, contrairement au modèle de Gold, la qualité de l'apprentissage dépend de la manière dont les exemples ont été présentés : les échantillons doivent donc être mieux « choisis » par rapport au concept à apprendre et par rapport au poids que leur attribue la distribution de probabilité. Ainsi, on n'exige pas qu'une présentation d'exemples issue d'un mauvais tirage conduise nécessairement au bon résultat.

Cependant, il reste encore de nombreuses classes sur lesquels il n'existe aucun résultat de PAC-apprenabilité (la classe DNF en est un exemple). De plus, les classes qui ont été prouvées poly-PAC-apprenables le sont en général par des algorithmes par trop simplistes[15].

Le modèle PAC de base considère que l'apprenant connaît *a priori* la classe à laquelle appartient le concept à inférer. Certaines versions du modèle PAC proposent un cadre plus général dans lequel l'algorithme d'apprentissage ne fait aucune hypothèse sur l'appartenance de la cible à une certaine classe d'hypothèses. De fait, l'apprenabilité dans ce cadre s'avère encore plus complexe que dans le modèle PAC [KSS92, HS92].

Une première critique sur le modèle PAC concerne sa très (trop?) grande exigeance. En effet, les conditions d'apprenabilité sont établies dans le « pire des cas » tant au niveau de la complexité en temps des algorithmes d'apprentissage qu'au niveau des distributions de probabilité utilisées. Ceci peut interdire à certaines classes d'être apprenables alors qu'elles pourraient y prétendre la plupart du temps.

La trop grande liberté dans le choix de la distribution de probabilité est l'un des points les plus critiqués de ce modèle. En effet, aucune restriction n'est faite sur les mesures servant à tirer les exemples, ce qui « durcit » (inutilement?) le modèle. Est-il vraiment nécessaire de considérer toutes les distributions de probabilité possibles, même celles qui ne sont pas calculables, ni même énumérables? Ne serait-il pas plus naturel de ne considérer que des distributions adaptées au concept à apprendre, comme il semble plus sérieux de faire enseigner les mathématiques par un professeur de mathématiques et non par un professeur de latin? Malgré tout, il faut considérer le modèle de Valiant comme un des modèles d'apprentissage les plus exigeants ; si une classe est apprenable dans ce modèle, elle a de fortes chances de l'être aussi dans d'autres modèles plus souples.

Cette dernière critique fut à l'origine de nombreux travaux qui ont fait évoluer le modèle de Valiant vers d'autres modèles moins contraignants [BI91, KLPV87, Nat87] en le restreignant à des distributions particulières. Cependant, le choix d'une distribution fixée est souvent trop restrictif.

Le modèle de Li et Vitányi tente lui aussi de pallier cette exigence du modèle de Valiant en ne considérant que la mesure de probabilité de Solomonoff-Levin (voir section 1.4.1, page 44). Ce modèle est présenté dans la section suivante.

15. Ceux-ci méritent-ils vraiment le qualificatif d'*algorithmes d'apprentissage*?

1.4 Présentation du Modèle de Li et Vitányi

Li et Vitányi ont proposé un nouveau modèle [LV91], appelé *modèle d'apprentissage simple PAC*. L'argument informel avancé pour justifier cette déclinaison du modèle de Valiant est que « fournir les exemples les plus simples en priorité doit aider à améliorer la vitesse de l'apprentissage ». Celui-ci s'appuie donc fortement sur le modèle de Valiant, mais la distribution de probabilité utilisée doit favoriser les exemples les *plus simples*. La distribution sur les exemples n'est plus quelconque comme dans le modèle PAC. La *complexité de Kolmogorov* permet de définir une mesure de simplicité sur les objets. La *mesure de probabilité de Solomonoff-Levin* s'appuie sur cette mesure de complexité et donne un poids plus fort aux exemples dits *simples*. La présentation du modèle de Li et Vitányi est précédée des définitions de base de la *complexité de Kolmogorov* et de la *mesure de probabilité de Solomonoff-Levin* afin de mieux appréhender le modèle lui-même.

1.4.1 Complexité de Kolmogorov

Nous présentons ici sommairement la théorie de la complexité de Kolmogorov dans sa version préfixe, sans donner l'intégralité des définitions, ni des preuves. Cette mesure de complexité, aussi connue sous le nom de *complexité algorithmique*, définit la complexité d'un objet comme étant la longueur de la plus courte représentation de cet objet ou encore comme la longueur du plus court programme permettant de générer celui-ci.

Pour une présentation plus détaillée de la complexité de Kolmogorov, voir [LV92], [LV93] ou [Del93].

Complexité de Kolmogorov préfixe

Soit $x \in \{0,1\}^*$, on identifie la chaîne x avec son numéro dans l'énumération selon l'*ordre standard*[17].

Dans la suite, on considère les machines de Turing à trois rubans :

- un ruban d'entrée unidirectionnel pouvant contenir uniquement des 0 et des 1 (pas de caractère « blanc »),
- un ruban de sortie unidirectionnel,
- un ruban de travail bidirectionnel.

La complexité de Kolmogorov de la chaîne x est définie pour une machine de Turing à trois rubans T particulière comme suit : si T s'arrête avec la chaîne x sur le ruban de sortie, alors T a lu un morceau initial fini p sur le ruban d'entrée et donc $T(p) = x$. L'ensemble de toutes les chaînes p pour lesquelles T s'arrête forme un code préfixe, c'est-à-dire qu'aucune des chaînes de cet ensemble n'est préfixe d'une autre chaîne de celui-ci.

Définition 1.4.1 *La* complexité de Kolmogorov préfixe *de la chaîne x pour la machine de Turing T, notée $K_T(x)$, est égale à :*

$$\begin{cases} min\{|p| \mid T(p) = x\} & s'il\ existe\ un\ tel\ p, \\ \infty & sinon. \end{cases}$$

17. Sur l'alphabet $\{0,1\}$, l'énumération des chaînes selon l'ordre standard est : $\varepsilon, 0, 1, 00, 01, 10, 11, 000, \ldots$

Intuitivement, la complexité de Kolmogorov de la chaîne x sur la machine de Turing T est la longueur d'un plus petit programme générant x sur la machine T.

De plus, il peut être pratique de permettre à la machine T d'utiliser de l'information supplémentaire, notée y. Alors, la machine T avec l'entrée p et l'information supplémentaire y s'arrête avec comme résultat la chaîne x. On écrit dans ce cas : $T(p, y) = x$.

Définition 1.4.2 *La* complexité de Kolmogorov préfixe *de la chaîne x pour la machine de Turing T connaissant y, notée $K_T(x \mid y)$ est égale à :*

$$\begin{cases} min\{|p| \mid T(p, y) = x\} & s'il\ existe\ un\ tel\ p, \\ \infty & sinon. \end{cases}$$

Cette notion de complexité paraît séduisante puisqu'elle mesure la taille maximale de compression d'une chaîne pour une machine donnée. Cependant, sa dépendance vis à vis de la machine sur laquelle sont faits les calculs semble être un frein à la puissance de cette mesure de complexité. Le théorème d'invariance permet d'outrepasser cette limite.

Théorème d'invariance

Ce théorème permet d'asseoir la robustesse de la complexité de Kolmogorov en éliminant la dépendance vis à vis de la machine de Turing.

Une machine de Turing est dite *universelle* si elle est capable de « simuler » n'importe quel exécution de tout autre machine de Turing.

Théorème 1.4.1 *Soit U une machine de Turing universelle. Quelle que soit T une machine de Turing, $\exists\ C_{U,T} \in I\!N$ tel que $\forall\ x, y \in \{0,1\}^*, K_U(x \mid y) \leq K_T(x \mid y) + C_{U,T}$.*

Un corollaire simple à ce théorème libère complètement la complexité de Kolmogorov vis à vis de la machine de Turing universelle utilisée.

Corollaire 1.4.2 *Soient U et U' deux machines de Turing universelles. Alors, $\exists\ C_{U,U'} \in I\!N$ tel que $\forall\ x, y \in \{0,1\}^*, |K_U(x \mid y) - K_{U'}(x \mid y)| \leq C_{U,U'}$.*

Idée de preuve :
La preuve est immédiate en utilisant le théorème d'invariance. □

Ainsi donc, la complexité de Kolmogorov d'une chaîne x est unique à une constante près quelle que soit la machine de Turing sur laquelle elle est calculée. Il est donc complètement indifférent de choisir une machine de Turing préfixe universelle plutôt qu'une autre. Pour la suite, on fixe une machine de Turing préfixe universelle U et on note $K(x) = K_U(x)$ et $K(x \mid y) = K_U(x \mid y)$.

Borne supérieure sur la complexité de Kolmogorov

Nous l'avons vu, la complexité de Kolmogorov définit une notion de compression optimale pour une chaîne x. Cependant, cette valeur idéale présente l'inconvénient d'être non calculable. Seule une borne supérieure sur celle-ci peut être donnée.

Théorème 1.4.3 *Soit $x \in \{0,1\}^*$, alors $K(x) \leq |x| + 2\log(|x|) + O(1)$, avec $|x| = \log(x)$.*

Preuve :
Supposons le programme p suivant : `print(x)`. Il est évident que ce programme affiche sur le ruban de sortie la chaîne x. La longueur de p est donc une borne supérieure pour $K(x)$. Étant donné que nous nous plaçons dans le cas des machines de Turing préfixes, x doit s'écrire de manière préfixe. Un moyen pour ce faire consiste à faire précéder la chaîne proprement dite par sa propre longueur. Le codage de la longueur doit aussi être fait de manière préfixe, par exemple en doublant chacun des bits le constituant et en terminant par un 0 suivi d'un 1 [18]. Le codage de la longueur de x aura donc comme longueur $2\log(|x|)+2$. Finalement, sur le ruban d'entrée de la machine U, notre programme complet s'écrit comme suit :

$$\overbrace{10111010010}^{\texttt{print}}\overbrace{1111001101}^{\text{Longueur de } x}\overbrace{0111001011110}^{x}$$

et a donc une longueur totale égale à $|x|+2\ln_2(|x|)+2+|\text{instruction } \texttt{print}|$, ce qui prouve la borne proposée. Remarquons que la constante capte et la longueur occupée par l'instruction `print`, et les quelques codes permettant de délimiter les différentes parties du programme. □

Inégalités générales

Nous donnons ici deux inégalités « célèbres » de la complexité de Kolmogorov. Ces inégalités trouveront leur intérêt par la suite afin d'évaluer la complexité d'un objet x relativement à un objet y. De plus, indépendamment de toute utilisation, ces inégalités expliquent quelque peu la notion de complexité de Kolmogorov relative.

Théorème 1.4.4 *Soient $x, y, z \in \Sigma^*$.*

- $K(x \mid y) \leq K(x) + O(1)$.
- $K(x \mid y) \leq K(x \mid z) + K(z \mid y) + O(1)$.

La première inégalité montre qu'il est moins complexe de décrire une chaîne x si l'on dispose déjà d'information y qu'en partant de rien. En effet, si x et y ont un rapport entre elles, la connaissance préalable sur y permet de décrire plus simplement x. Intuitivement, on peut imaginer que si x et y sont deux versions d'un même fichier et que l'on dispose déjà de l'un d'entre eux, il est suffisant de ne donner que les différences existantes entre ces deux versions afin de décrire l'autre.

De manière intuitive, la deuxième inégalité se comprend également facilement. Si l'on dispose d'un programme générant x au moyen de z et d'un autre programme générant z à partir de y, alors on dispose d'un programme simple générant x à partir de y ; il suffit de faire tourner les deux programmes précédents à la suite l'un de l'autre. Par conséquent, la longueur du plus petit programme générant x en connaissant y sera au plus égale à la longueur de ce programme trivial.

18. Remarquons que cette manière d'écrire x sous forme préfixe est généralement plus courte qu'en doublant simplement chaque bit de x. Ceci nous permet d'obtenir une borne plus précise.

Distribution universelle de Solomonoff-Levin

Avant de présenter la mesure de probabilité de Solomonoff-Levin, il convient d'énoncer le théorème suivant:

Théorème 1.4.5 *Soit U une machine de Turing préfixe universelle.*

$$\sum_{x \in \Sigma^*} 2^{-K(x)} \leq 1$$

La preuve ce théorème sera présentée par la suite dans sa variante « conditionnelle » (voir théorème 3.1.1, page 86).

Remarque 1.4.1 Ce théorème est une application directe d'un théorème plus général du à L.G. Kraft [Kra49] concernant les ensembles préfixes. Le chapitre 1 de [LV93] présente en détail ce théorème.

Le théorème 1.4.5 établit que la fonction $\sigma : \Sigma^* \longmapsto I\!R, \sigma(x) = 2^{-K(x)}$ est une semi-mesure sur Σ^*. Il est possible dès lors, de s'appuyer sur cette fonction afin de définir une mesure de probabilité sur Σ^* :

Définition 1.4.3 *La distribution de Solomonoff-Levin, notée* $\mathbf{m}(x)$, *est telle que :* $\mathbf{m}(x) = \lambda 2^{-K(x)}$, *où* λ *satisfait*

$$\lambda \sum_{x \in \Sigma^*} 2^{-K(x)} = 1$$

Une distribution de probabilité est *énumérable* si elle peut être approximée par le bas; par exemple, les distributions de probabilité récursives (c'est-à-dire pour lesquelles il existe un programme calculant la valeur de probabilité pour tout entier n) sont énumérables.

On peut montrer que la semi-mesure σ est énumérable et que $\mathbf{m}$ est une mesure non énumérable. Cependant, $\mathbf{m}$ est une distribution de probabilité *universelle*, c'est-à-dire que pour toute distribution de probabilité énumérable P, il existe une constante c telle que pour toute chaîne x, $c\mathbf{m}(x) \geq P(x)$.

De plus, pour toutes paires de machines de Turing universelles U et U' qui satisfont le théorème d'invariance, il existe une constante $C_{U,U'}$ telle que
$\forall x \in \Sigma^*, 2^{-C_{U,U'}} \leq \mathbf{m}(x)/\mathbf{m}'(x) \leq 2^{C_{U,U'}}$ où $\mathbf{m}$ et $\mathbf{m}'$ représentent les distributions de Solomonoff-Levin associées aux machines U et U'.

Intuitivement, la distribution de probabilité de Solomonoff-Levin favorise le tirage des chaînes qui ont une complexité de Kolmogorov faible, c'est-à-dire les objets pour lesquels il est possible de trouver une description courte. On peut qualifier ce type de chaînes de *chaînes simples*[19].

1.4.2 Définition du modèle de Li et Vitányi

L'énoncé de l'apprenabilité d'une classe dans ce modèle est très proche de celui donné par le modèle de Valiant. Il suffit simplement de remplacer la mesure de probabilité P (quelconque et inconnue) par la distribution universelle de Solomonoff-Levin $\mathbf{m}$. On obtient ainsi les définitions suivantes :

Définition 1.4.4 *Soit* A *un algorithme prenant en entrée un* paramètre de précision $\varepsilon \in]0,1]$. *un* paramètre de confiance $\delta \in]0,1]$. A *a recours à un oracle d'exemples* $Exemple_{X,\mathbf{m}}(f)$. *Si*

19. Nous verrons par la suite que ce terme trouve son utilité lorsqu'il est relatif à un concept.

pour tout concept f de F, A *retourne la représentation d'un concept h de F tel que avec la probabilité au moins $1-\delta$, $Erreur_{\mathbf{m},f}(h) \leq \varepsilon$ alors* A *est un* algorithme de simple-PAC-apprentissage *pour la classe F associée à R.*

Définition 1.4.5 *Une classe de concepts F est* simple-PAC-apprenable *dans l'ensemble des représentations R si il existe un algorithme de simple-PAC-apprentissage pour F dans R.*

Ici encore, il est possible d'imposer une contrainte supplémentaire sur la complexité en temps de l'algorithme d'apprentissage. La définition qui suit prend ce paramètre en compte.

Définition 1.4.6 *Une classe de concepts F est* polynomialement simple-PAC-apprenable *(poly-simple-PAC-apprenable) dans R si elle est simple-PAC-apprenable dans R et si* A *l'algorithme d'apprentissage tourne en temps polynomial par rapport à $1/\varepsilon$, $1/\delta$ et la longueur d'une plus petite représentation du concept cible.*

Remarquons que dans le cas du poly-simple-PAC-apprentissage, à la différence du poly-PAC-apprentissage, la complexité en temps de l'algorithme d'apprentissage ne dépend plus de la taille du plus long exemple de l'échantillon. En effet, les exemples de complexité de Kolmogorov faible peuvent être arbitrairement longs[20]. Aussi, inclure ce paramètre dans le calcul de la complexité en temps de l'algorithme conduirait à autoriser un temps de calcul non raisonnable dans le cas où de tels mots sont tirés. Par conséquent, un algorithme polynomial de simple-PAC apprentissage doit traiter seulement des préfixes de ces exemples dont la longueur est bornée par un polynôme en $1/\varepsilon$, $1/\delta$ et la longueur d'une plus courte description du concept cible. Remarquons que ce problème ne se pose pas dans le cas de l'apprentissage PAC ; en effet, l'apprentissage doit alors se faire sous toutes les distributions de probabilité possibles, ce qui inclut celles n'ayant pas la propriété de privilégier des chaînes arbitrairement longues.

1.4.3 Quelques résultats

Plusieurs classes ont été montrées poly-simple-PAC apprenables dans [LV91]. On peut citer par exemples, la classe des $\log n$-DNF. Une $\log n$-DNF *formule* est une formule DNF sur n variables dont la longueur est bornée par un polynôme en n et dont chaque monôme a une complexité de Kolmogorov en $O(\log n)$. Li et Vitányi ont également prouvé l'apprenabilité de la classe des automates déterministes simple-0-réversibles[21]. Un automate déterministe à n états est simple si chaque état se trouve sur un chemin menant de l'état initial à un état final dont la complexité de Kolmogorov est au plus en $\log n$.

Plus récemment, Castro et Balcàzar ont montré que la classe des $\log n$-listes de décision est poly-simple-PAC apprenable [Cas95, CB95].

1.4.4 Points de vue et critiques du modèle

Ce modèle étant moins contraignant sur le tirage des exemples que le modèle PAC, toutes les classes prouvées poly-PAC-apprenables sont poly-simple-PAC-apprenable. D'autres classes, qui ne sont jusqu'à présent pas connues poly-PAC-apprenables, ont pu être prouvées

20. Intuitivement, une chaîne composée de n fois la lettre 1 a une complexité de Kolmogorov faible et ce quelque soit la valeur de n.

21. Les langages 0-réversibles seront présentés à la section 2.1.3 (page 57).

poly-simple-PAC-apprenables. En fait, nous n'avons aucune idée s'il existe des classes poly-simple-PAC-apprenables qui ne sont pas poly-PAC-apprenables.

Un résultat du à Li et Vitányi montre que toute distribution dominée par $\mathbf{m}$[22] peut être utilisée pour évaluer l'apprentissage dès lors que les exemples ont été tirés au moyen de $\mathbf{m}$. Ce modèle joue donc un rôle particulier dans le domaine des distributions énumérables.

On peut cependant reprocher à ce modèle les défauts suivants :

1. Étant donnée leur trop grande dépendance vis à vis de la complexité de Kolmogorov, les nouvelles classes prouvées apprenables ne sont pas calculables, c'est-à-dire qu'il n'existe pas d'algorithme permettant de construire l'ensemble des concepts appartenant à celles-ci. À notre connaissance, aucune nouvelle classe « naturelle »[23] de concepts n'a été montrée apprenable dans ce modèle.

2. La mesure de simplicité d'exemple est faite indépendamment du concept cible. Il s'agit d'une mesure de simplicité « absolue ». De ce fait, quel que soit le concept à apprendre la distribution des exemples sera toujours du même ordre. Intuitivement, il semble pourtant qu'afin de réellement aider l'apprentissage, une distribution d'exemples adaptée au concept cible serait préférable. Cette idée est à la base du modèle PACS présenté au chapitre 3 (page 83).

1.5 Exemple d'Apprentissage dans les Trois Modèles

Afin d'illustrer un apprentissage dans chacun des trois modèles présentés précédemment, nous nous proposons d'étudier le cas de l'apprentissage de la classe des singletons sur $I\!N$. Ce premier exemple « concret » est suffisamment simple pour permettre de comparer le comportement de ces trois modèles.

1.5.1 Énoncé du problème

Le domaine n'est autre que l'ensemble des entiers naturels $X = I\!N$. La classe de concepts F est l'ensemble des singletons dans $I\!N$, c'est-à-dire $F = \{\{i\} \mid i \in I\!N\}$. L'ensemble de représentations R associé à F est l'ensemble des chaînes binaires, c'est-à-dire les entiers écrits en base 2. Un concept f est un singleton $f = \{i\}$ avec $i \in I\!N$. Un exemple est un élément de $I\!N \times I\!B$. Remarquons qu'étant donné tout concept $f = \{i\}$, avec $i \in I\!N$, il existe un unique exemple positif pour f : $(i, 1)$.

La classe F est apprenable dans chacun des trois modèles. Nous présentons, pour chacun d'eux, une étude succincte de l'algorithme d'apprentissage et du comportement de celui-ci. Précisons aussi que la connaissance *a priori* qu'a chacun de ces algorithmes est la nature de la classe F dans laquelle le concept hypothèse doit être inféré.

1.5.2 Apprentissage à la Gold

Un algorithme d'apprentissage dans le paradigme de Gold apprend à la limite, il s'agit donc d'un processus infini. Cependant, on est assuré qu'en un nombre fini d'étapes, l'algorithme est stabilisé sur le concept cible.

22. Ce qui inclut toutes les distributions récursives et énumérables.
23. C'est-à-dire non construite artificiellement pour vérifier les conditions d'apprentissage.

L'algorithme de la figure 1.4 (page suivante) est un tel algorithme d'apprentissage de F dans le modèle de Gold. La seule contrainte est que l'exemple positif correspondant au concept cible f à apprendre doit être présenté à l'algorithme en un nombre fini d'étapes.

1.5.3 Poly-PAC Apprentissage

Dans le cas du modèle de Valiant, on suppose que le domaine est muni d'une distribution de probabilité quelconque P inconnue de l'algorithme d'apprentissage. L'algorithme d'apprentissage de la classe F est ici encore très simple (voir figure 1.3page suivante).

Intuitivement, c'est le calcul d'une taille appropriée d'échantillon qui fait fonctionner l'algorithme. Cette taille m est d'autant plus grande que ε et δ sont proches de 0. L'échantillon contient alors un grand nombre d'exemples ayant, en principe, une forte probabilité d'être tirés selon P. Dans le cas où la probabilité sur l'exemple positif de f est grande, celui-ci appartient certainement à l'échantillon et l'algorithme sort alors le concept attendu ; dans le cas contraire, il sort un concept dont l'exemple positif a certainement une faible probabilité (puisqu'il ne fait pas partie de l'échantillon) et donc $Erreur_{P,f}(h)$ est petite.

Formellement, pour montrer que cet algorithme est bien un algorithme de PAC-apprentissage polynomial, il suffit de montrer qu'il existe effectivement un polynôme p tel que si $m >= p(1/\varepsilon, 1/\delta)$ alors les conditions de PAC-apprenabilité seront atteintes. Dans ce cas, il est évident que l'algorithme est bien polynomial en $1/\varepsilon, 1/\delta$, longueur du concept généré et longueur du plus long exemple lu dans S.

Évaluation du nombre d'exemples nécessaires

Soit le concept cible $f = \{i\}$, avec $i \in I\!N$. Considérons les deux cas possibles qui peuvent arriver lors de l'apprentissage :

1. **l'exemple positif de f a été tiré** : dans ce cas, l'algorithme sort comme hypothèse $h = f$, et alors $Erreur_{P,f}(h) = 0$ (car $f \Delta h = \emptyset$). L'apprentissage est atteint, on est même dans un cas d'apprentissage **exact**.

2. **l'exemple positif de f n'a pas été tiré** : l'algorithme retourne alors comme hypothèse un concept h consistant avec l'échantillon présenté, c'est-à-dire tel qu'aucun des exemples de l'échantillon ne soit l'exemple positif de h. Posons $h = \{j\}$, alors $f \Delta h = \{i, j\}$.

   Si $Erreur_{P,f}(h) \leq \varepsilon$, c'est-à-dire si $P(i) + P(j) \leq \varepsilon$ alors l'apprentissage PAC a, ici encore, réussi ; l'hypothèse inférée est « proche » de la cible, nous nous trouvons dans un cas d'apprentissage **approximatif**.

   Il reste alors le cas où $Erreur_{P,f}(h) > \varepsilon$, c'est à dire où l'apprentissage même approximatif n'a pas abouti. Nous allons montrer qu'en choisissant une taille d'échantillon suffisante (mais bornée par un polynôme) ce cas arrive rarement, c'est-à-dire avec une probabilité inférieure à δ. Si $Erreur_{P,f}(h) > \varepsilon$, cela signifie donc que soit $P(i) > \varepsilon/2$, soit $P(j) > \varepsilon/2$. Cela signifie donc que cet élément de probabilité supérieure à $\varepsilon/2$ (c'est-à-dire soit i, soit j) n'a pas été tiré lors des m tirages. Ceci a lieu avec une probabilité inférieure à $(1 - \varepsilon/2)^m$. Si $(1 - \varepsilon/2)^m \leq \delta$, on assure que ce mauvais cas arrive rarement. Il suffit pour cela que $m > \frac{2}{\varepsilon} \log(\frac{1}{\delta})$.

```
Algorithme Apprend_Singletons_1
Entrée:
Début
    Soit hypothèse courante h ← {0}
    Tant que pas Fin_du_monde
        Demander un nouvel exemple e ← (i, b)
        Si e est un exemple positif
        Alors h ← {i}
        Fsi
    Ftq
Fin
```

FIG. 1.4 - Algorithme d'apprentissage à la limite de la classe des singletons sur $\mathbb{N}$.

```
Algorithme Apprend_Singletons_2
Entrée:ε, δ
Début
    Tirer un échantillon S de taille m = p(1/ε, 1/δ) en utilisant la distribution P
    Si S contient un exemple positif e = (i, 1)
    Alors h ← {i}
    Sinon h ← {j} tel que (j, 0) ∉ S
    Fsi
    Sortir h
Fin
```

FIG. 1.3 - Algorithme de poly-PAC-apprentissage des singletons sur $\mathbb{N}$.

1.5.4 Poly Simple-PAC Apprentissage

L'algorithme d'apprentissage dans le modèle de Li et Vitányi (voir figure 1.5, page suivante) est le même que celui utilisé dans l'apprentissage PAC, à la différence que la distribution de probabilité utilisée est **m**.

On peut appliquer ici les mêmes remarques que dans le cas du poly-PAC-apprentissage. La probabilité que l'exemple positif du concept cible soit tiré dépend simplement de la simplicité ou non de cet exemple. Ainsi, si $f = \{i\}$ avec i ayant une représentation binaire complexe. l'exemple positif n'a que peu de chance d'être présenté. Dans ce cas, il est donc très peu probable que l'algorithme d'apprentissage retourne le concept cible. Le modèle assure seulement que $Erreur_{\mathbf{m},f}(h)$ sera faible.

Algorithme Apprend_Singletons_3
Entrée : ε, δ
Début
Tirer un échantillon S de taille $m = p(\frac{1}{\varepsilon}, \frac{1}{\delta})$ en utilisant la distribution $\mathbf{m}$
Si S contient un exemple positif $e = (i, 1)$
Alors $h \leftarrow \{i\}$
Sinon $h \leftarrow \{j\}$ tel que $(j, 0) \notin S$
Fsi
Sortir h
Fin

FIG. 1.5 - Algorithme de poly-simple-PAC-apprentissage des singletons sur $I\!N$

1.5.5 Conclusion

Cet exemple simple a donc permis de présenter les principes généraux d'utilisation des trois modèles ainsi que les inconvénients majeurs de chacun d'eux. Dans le cas de Gold, l'algorithme apprend exactement le concept cible. Remarquons que dans ce cas précis, l'algorithme est en mesure de savoir qu'il a effectivement appris le concept cible, étant donné la connaissance qu'il a sur la classe de concepts (il sait que chaque concept est un singleton); c'est un fait rare dans le cas de l'apprentissage à la Gold pour le souligner. Dans le cas du modèle de Valiant, si la distribution de probabilité P ne donne pas à l'exemple positif de f un poids suffisant, l'apprentissage « réussit » seulement par le fait que les poids sur les éléments de $f \Delta g$ sont faibles. Enfin, dans le cas du modèle de Li et Vitányi, on se retrouve dans une situation analogue si le concept cible est un entier peu compressible.

Dans une situation naturelle d'un tel apprentissage, le professeur aurait certainement passé l'exemple positif dans l'échantillon d'apprentissage. L'élève aurait alors, avec une grande probabilité, sorti le concept cible correspondant à cet exemple positif. Cette idée sera développée dans le chapitre 3 (page 83) afin de proposer un nouveau modèle d'apprentissage.

Chapitre 2

Application de l'Apprentissage aux Langages Réguliers

Le chapitre précédent présentait les définitions concernant l'apprentissage automatique ainsi qu'une vue d'ensemble de trois modèles d'apprentissage particuliers. Nous nous proposons dans ce chapitre d'étudier, dans chacun de ces trois modèles, l'apprentissage de classes de concepts particulières : celles des langages formels. Ce type d'inférence fait l'objet d'un sujet d'étude à part entière. Il occupe à lui seul un pan complet du domaine de l'apprentissage automatique. Il s'agit essentiellement de générer, à partir d'un ensemble fini de mots, un langage qui capte les propriétés communes de ces mots. Historiquement parlant, l'étude de ces algorithmes commence bien avant les recherches en apprentissage automatique puisqu'on la trouve déjà dans les années 70 dans les travaux concernant la reconnaissance de formes. C'est au cours des années 80 que ces travaux ont été transposés et renouvelés dans le cadre de l'apprentissage automatique. On parle alors d'*inférence grammaticale* ou encore d'*apprentissage de langages*. En ce qui nous concerne, nous étudierons uniquement l'apprentissage de la plus simple des classes des langages formels (selon la hiérarchie définie par Chomsky [Cho56]), celle des langages réguliers. Nous présenterons d'abord les définitions de base concernant les langages réguliers. Ensuite, nous présenterons la classe des langages réversibles qui est une classe particulière de langages réguliers. Enfin, nous étudierons dans quelle mesure, la classe des langages réguliers est apprenable dans chacun des trois modèles d'apprentissage définis dans le chapitre 1 (page 25). Cette étude sera l'occasion de présenter quelques algorithmes et techniques qui font l'état de l'art dans ce domaine.

2.1 Définitions Préliminaires

L'ensemble des notations et définitions utiles par la suite est présenté dans cette section. Celles-ci concernent le domaine de la théorie des langages et plus particulièrement celui des langages réguliers. En effet, dans la suite de cette thèse, nous porterons notre attention essentiellement sur l'apprentissage de ce type de langages. Nous supposons que le lecteur est familier avec la théorie des langages réguliers, aussi l'ensemble des définitions et résultats présenté ici n'est pas développé. On pourra trouver une présentation détaillée des langages réguliers dans [HU79].

2.1.1 Généralités

Alphabets, mots et langages

Un *alphabet* Σ est un ensemble fini non vide de symboles appelés *lettres*. Un *mot* est une suite finie de lettres. Le mot vide ε est la chaîne de longueur nulle. La longueur d'un mot u est notée $|u|$. Si $a \in \Sigma$, $|u|_a$ représente le nombre de a contenu dans la chaîne u. On note u_i le $i^{\text{ème}}$ caractère de u avec $1 \leq i \leq |u|$. Si E est un ensemble fini de mots, on note $\|E\|$ la somme des longueurs des mots de E, c'est-à-dire $\|E\| = \sum_{u \in E} |u|$.

Σ^* est l'ensemble des mots finis sur Σ. Comme nous l'avons déjà indiqué, sans autre précision, on suppose que $\Sigma = \{0, 1\}$.

On considère l'*ordre standard* sur les chaînes de Σ^*, c'est-à-dire, si $\Sigma = \{0, 1\}$, l'énumération des chaînes de Σ^* selon cet ordre est: $\varepsilon, 0, 1, 00, 01, 10, 11, 000, \ldots$ On notera $u < v$ pour signifier que le numéro d'ordre de u est strictement inférieur à celui de v dans cette énumération.

Soit $w \in \Sigma^*$; si $w = uv$, alors u et v sont des mots de Σ^* appelés respectivement *préfixe* et *suffixe* de w. Soit $i \leq |w|$; on note $Pre_i(w)$ le préfixe de w de longueur i et $Suf_i(w)$ le suffixe de w de longueur i.

Un *langage* L est un sous-ensemble de Σ^*.

Automates finis

Un *automate fini* est un quintuplet $A = (Q, I, F, \Sigma, \delta)$ avec Q un ensemble fini d'états, I un sous-ensemble de Q appelé *ensemble d'états initiaux*, F un sous-ensemble de Q appelé *ensemble d'états terminaux* et δ une *fonction de transitions*: $\delta : Q \times \Sigma \to 2^Q$. Si $q' \in \delta(q, a)$ avec $q, q' \in Q$ et $a \in \Sigma$, alors q' est appelé *a-successeur* de q et q est appelé *a-prédécesseur* de q'. La fonction δ peut être naturellement étendue aux mots sur Σ^*. Un état q est un *état puits* s'il n'existe aucun mot labellant un chemin depuis q vers un état final, c'est-à-dire $\forall\, v \in \Sigma^*, \delta(q, v) \cap F = \emptyset$. Le langage accepté par l'automate A, noté $L(A)$ est constitué de l'ensemble des mots u de Σ^* tels que $\delta(q_i, u) \in F$ avec $q_i \in I$.

L'*ensemble des langages réguliers* est l'ensemble des langages reconnus par de tels automates ; on note $\mathcal{R}eg$ la classe de ces langages.

Un automate fini A est *déterministe*[1], si I contient un seul état (noté q_0) et si pour tout état $q \in Q$, pour toute lettre $a \in \Sigma$, $\delta(q, a)$ a au plus un élément $q' \in Q$. Si de plus, pour tout état $q \in Q$ et pour toute lettre $a \in \Sigma$, $\delta(q, a)$ a exactement un élément $q' \in Q$, l'automate déterministe est dit *complet*.

Il est bien connu que tout langage régulier $L \in \mathcal{R}eg$ est reconnu par un DFA. Étant donné un langage régulier $L \in \mathcal{R}eg$, il existe un DFA minimal au niveau du nombre d'états reconnaissant L ; cet automate, appelé *automate canonique de* L, est unique[2] ; on le notera $A(L)$. De plus, étant donné un DFA A, il existe une procédure efficace permettant de calculer l'automate canonique de $L(A)$; une telle procédure est appelée *procédure de minimalisation*. On appelle *automate universel* pour un alphabets Σ, l'automate canonique reconnaissant le langage Σ^*.

Soit $L_{ex1} = \{u \in \Sigma^* \mid u_1 = 1 \text{ et } u_{|u|} = 1\}$, c'est-à-dire le langage composé des mots commençant et se terminant par la lettre 1. L'expression rationnelle correspondante à ce langage est $1(0 + 1)^*1$. La figure 2.1 (page suivante) présente l'automate canonique reconnaissant ce

1. Encore appelé DFA pour *Deterministic Finite Acceptor*.
2. Au renommage des états près.

langage. Les états sont représentés par des cercles, les transitions par des flèches labellées par la lettre de transition. L'état initial (ici q_0) est symbolisé par une flèche plus épaisse. Les états terminaux (ici q_2) sont caractérisés par un double cercle.

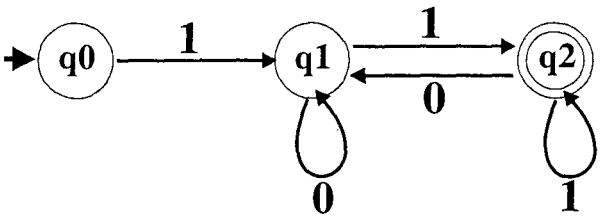


FIG. 2.1 - Automate canonique du langage $L_{ex1} = 1(0+1)^*1$.

Partitions et automates quotients

Soit E un ensemble quelconque. Une *partition* π de E est un ensemble de sous-ensembles non vides de E, deux à deux disjoints et dont l'union est E. La partition *canonique* de E est l'ensemble des singletons construits à partir des éléments de E. Soit $e \in E$, on note $B(e,\pi)$ l'élément unique de π contenant e ; ce sous-ensemble est appelé *bloc*. Soient $\pi_1 = \{B_{1,1}, B_{1,2}, \ldots, B_{1,p}\}$ et $\pi_2 = \{B_{2,1}, B_{2,2}, \ldots, B_{2,q}\}$ deux partitions d'un ensemble E ; on dit que π_2 *dérive directement de* π_1 si et seulement si $\exists\ 1 \leq i \leq p, 1 \leq j \leq p$ tels que $\pi_2 = (\pi_1 \setminus \{B_{1,i}, B_{1,j}\}) \cup \{B_{1,i} \cup B_{1,j}\}$. On a alors évidemment $q = p - 1$. Cette opération définit une relation d'ordre partiel que l'on note $\pi_1 \preceq \pi_2$. Soit $\ll$ la clôture transitive de cette relation. On dira qu'une partition π' est *plus fine que*[3] la partition π si tout bloc de π' est une union de un ou plusieurs blocs de π ; on notera $\pi \ll \pi'$.

Soit $A = (Q, \{q_0\}, F, \Sigma, \delta)$ un automate et soit π une partition de l'ensemble Q. On appelle *automate quotient* A/π (ou encore *automate dérivé de A par la partition* π) l'automate $A/\pi = (Q', B(q_0,\pi), F', \Sigma, \delta')$ tel que :

- $Q' = \{B(q,\pi) \mid q \in Q\}$,
- $F' = \{B \in Q' \mid B \cap F \neq \emptyset\}$,
- $\delta' : Q' \times \Sigma \to 2^{Q'}, \forall\ a \in \Sigma, \forall\ B_1 \in Q', \forall\ B_2 \in Q', B_2 \in \delta'(B_1, a)$ si et seulement si $\exists\ q_1 \in B_1, \exists\ q_2 \in B_2$ tel que $q_2 \in \delta(q_1, a)$.

Exemple :
La figure 2.2 (a) (page suivante) présente un automate (non canonique) A_1 reconnaissant le langage $L_{ex1} = 1(0+1)^*1$. Soit $\pi_1 = \{\{q_a, q_c\}, \{q_b\}, \{q_d\}, \{q_e\}\}$. La figure 2.2 (b) (page suivante) représente l'automate (non déterministe) A_1/π_1. Soit $\pi_2 = \{\{q_a\}, \{q_b, q_c\}, \{q_d\}, \{q_e\}\}$. La figure 2.2 (c) (page suivante) représente l'automate A_1/π_2. ◇

Il existe une procédure efficace qui construit à partir d'un automate A et d'une partition π de son ensemble d'états l'automate quotient A/π. En effet, il suffit de fusionner ensemble tous les états faisant partie d'un même bloc. Notons que l'automate canonique d'un langage L peut être trouvé à partir d'un DFA reconnaissant L et d'une partition particulière.

3. Ou bien encore *raffine*.

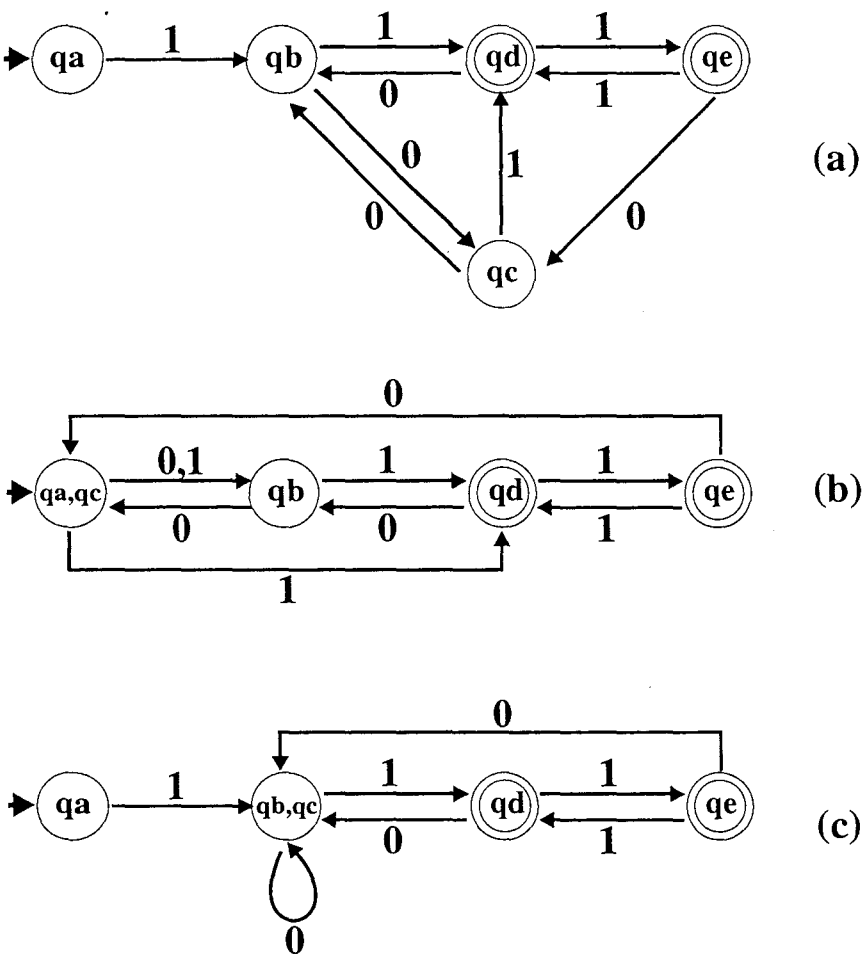


FIG. 2.2 - (a) Automate A_1 non canonique reconnaissant le langage $L_{ex1} = 1(0+1)^*1$.
(b) Automate A_1/π_1. (c) Automate A_1/π_2.

Exemple :
Considérons encore l'automate A_1 non canonique reconnaissant le langage L_{ex1} (voir figure 2.2 (a)). L'automate A_1/π avec $\pi = \{\{q_a\}, \{q_b, q_c\}, \{q_d, q_e\}\}$ est l'automate canonique du langage L_{ex1} représenté à la figure 2.1 (page précédente) avec $q_0 = \{q_a\}$, $q_1 = \{q_b, q_c\}$ et $q_2 = \{q_d, q_e\}$. ◇

Étant donné un automate $A = (Q, \{q_0\}, F, \Sigma, \delta)$, l'ensemble de tous les automates possibles obtenus par dérivation de toutes les partitions possibles de Q forme un treillis d'automates finis [PC78] avec $A/\pi \ll A/\pi'$ si et seulement si $\pi \ll \pi'$. L'élément nul (resp. universel) ce treillis est l'automate A (resp. l'automate universel). On note ce treillis d'automates obtenu à partir de A, $Lat(A)$. La forme générale d'un tel treillis est représentée à la figure 2.3 (page suivante).

Une propriété importante due à la construction de A/π établit que $L(A) \subseteq L(A/\pi)$ [FB75].

Remarque 2.1.1 Un grand nombre d'algorithmes d'apprentissage pour les langages réguliers utilise des dérivations d'automates à partir d'un automate initial et d'une succession de

partitions calculées. Nous verrons par la suite que dans de nombreux cas, l'essentiel du problème d'inférence régulière consiste à trouver la meilleure partition qui soit afin d'atteindre les objectifs de l'apprentissage.

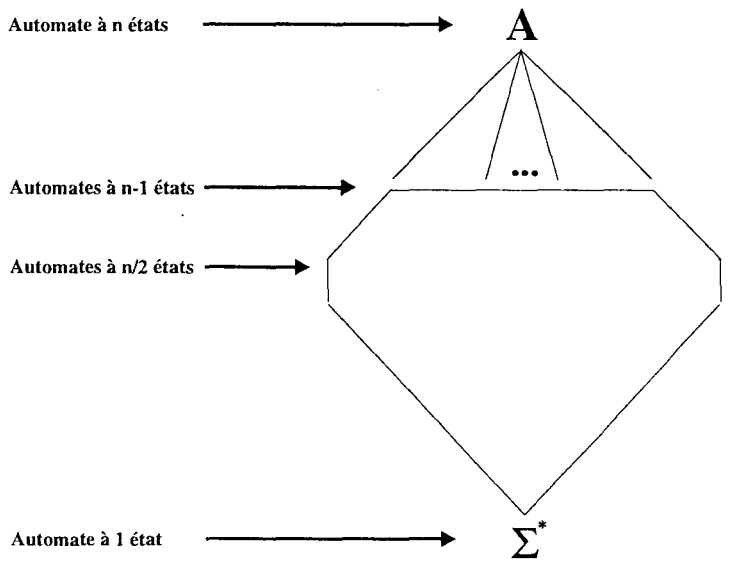


FIG. 2.3 - Forme générale du treillis d'un automate A à n états.

2.1.2 Outils pratiques

Nous présentons ici un ensemble d'« objets » de la théorie des langages. Ceux-ci ont pour point commun d'être des éléments clés dans nombre d'algorithmes d'apprentissage sur la classe $\mathcal{R}eg$. Nous en verrons l'utilisation par la suite, lors de la présentation d'algorithmes d'apprentissage.

Langage préfixe, langage des facteurs droits

Soit L un langage sur l'alphabet Σ. Le *langage préfixe de* L, noté $Pr(L)$, est l'ensemble composé de tous les mots préfixes des mots appartenant à L. Formellement,
$Pr(L) = \{u \in \Sigma^* \mid \exists\, v \in \Sigma^*,\ uv \in L\}$. Il est évident que $L \subseteq Pr(L)$. Notons aussi que pour tout langage $L \neq \emptyset$, $\varepsilon \in Pr(L)$ puisque le mot vide est préfixe de tout mot.

Exemple :
Le langage préfixe du langage $L_{ex1} = 1(0+1)^*1$ (voir figure 2.1, page 53) est le langage $Pr(L_{ex1}) = \varepsilon + 1(0+1)^*$. ◇

Soit $u \in \Sigma^*$. Le langage des facteurs droits de L par u, noté[4] $T_L(u)$, est l'ensemble des mots v tels que $uv \in L$, c'est-à-dire $T_L(u) = \{v \in \Sigma^* \mid uv \in L\}$.

Remarque 2.1.2 $T_L(\varepsilon) = L$.

4. Ce type d'ensembles est appelé *tail* en anglais.

Exemple :
On considère encore le langage L_{ex1} de la figure 2.1 (page 53). Soit le mot $u = 1$, alors $T_{L_{ex1}}(u) = (0+1)^*1$ (ensemble des mots se terminant par 1). ◇

Automate arbre préfixe

Soit E un ensemble **fini** de mots de Σ^*. L'*automate arbre préfixe* de E, noté $PT(E)$, est un DFA acceptant tous les mots de E et uniquement eux. Pour construire l'automate arbre préfixe, il suffit de faire correspondre un état à chaque mot de $Pr(E)$; ainsi, $PT(E) = (Pr(E), \{\varepsilon\}, E, \Sigma, \delta)$ avec $\delta(u, a) = ua$ $(u, ua \in Pr(E)$ et $a \in \Sigma)$.

Exemple :
Soit $E = \{0, 10, 100, 110\}$, $Pr(E) = \{\varepsilon, 0, 1, 10, 11, 100, 110\}$. L'automate arbre préfixe correspondant à cet ensemble est présenté à la figure 2.4 . ◇

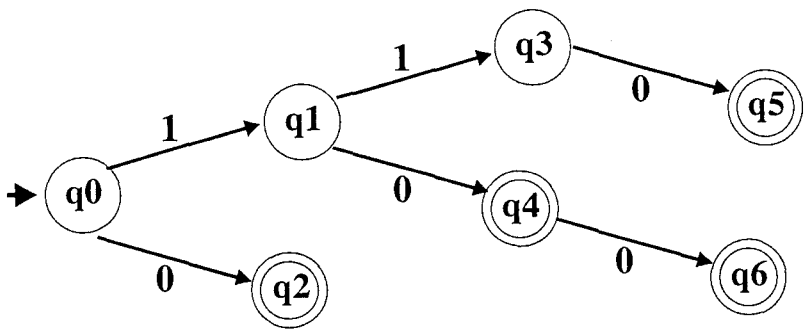


FIG. 2.4 - Automate arbre préfixe pour l'ensemble $S = \{0, 10, 100, 110\}$.

Ensemble des préfixes courts et noyau

Soit L un langage sur l'alphabet Σ^*. L'*ensemble des préfixes courts de* L, noté $SP(L)$ est défini par: $SP(L) = \{u \in Pr(L) \mid \nexists\ v \in \Sigma^*$ tq $T_L(u) = T_L(v)$ et $v < u\}$. Par rapport à l'automate canonique de L, cet ensemble contient tous les mots labellant les plus courts chemins menant de l'état initial à chacun des états; il y a donc autant d'éléments dans $SP(L)$ que d'états dans $A(L)$.

Exemple :
Pour le langage $L_{ex1} = 1(0+1)^*1$, $SP(L_{ex1}) = \{\varepsilon, 1, 11\}$. ◇

Le *noyau de* L, noté $N(L)$, est défini par $N(L) = \{\varepsilon\} \cup \{ua \mid u \in SP(L), a \in \Sigma, ua \in Pr(L)\}$ Remarquons que $SP(L) \subseteq N(L)$. Par rapport à l'automate canonique de L, cet ensemble contient tous les mots labellant un chemin menant de l'état initial à chaque état de l'automate et passant au maximum deux fois dans un seul état; le cardinal de cet ensemble est donc égal au nombre de transitions de $A(L)$ plus 1.

Exemple :
Pour le langage $L_{ex1} = 1(0+1)^*1$, $N(L_{ex1}) = \{\varepsilon, 1, 10, 11, 110, 111\}$. ◇

2.1.3 Les langages réversibles

La classe des langages réversibles est une sous-classe des langages réguliers. Celle-ci se compose de l'ensemble des classes des langages *k-réversibles* avec $k = 0, 1, 2, \ldots$ La classe des 0-réversibles a été étudiée sans être nommée comme telle dans [McN67]. Une étude approfondie de l'inférence des langages réversibles est due à Dana Angluin [Ang82a]. La plupart des définitions et résultats qui suivent sont tirées de [Ang82a].

Les langages réversibles tiendront un rôle important tout au long de cette thèse puisque ils apparaîtront soit en tant qu'exemples illustrant des algorithmes d'apprentissage, soit en tant que classes pour lesquelles existent des résultats d'apprenabilité.

Langages 0-réversibles

L'automate miroir A^r d'un automate quelconque A est obtenu en inversant chacune des transitions, et en échangeant le rôle des états terminaux et initiaux.

Définition 2.1.1 *Un automate déterministe A est* 0-réversible *si et seulement si son automate miroir A^r est déterministe.*

Un langage régulier est 0-réversible s'il existe un automate 0-réversible qui le reconnaît. De plus, le lemme 6 de [Ang82a], donne une condition nécessaire et suffisante pour qu'un langage régulier soit 0-réversible.

Lemme 2.1.1 *Un langage régulier est 0-réversible si et seulement si son automate canonique est 0-réversible.*

Exemple :
Le langage $L_{ex2} = 0^*10^*$ (voir figure 2.5 (a)) est un langage 0-réversible, puisque son automate miroir (voir figure 2.5 (b)) est aussi déterministe. ◇

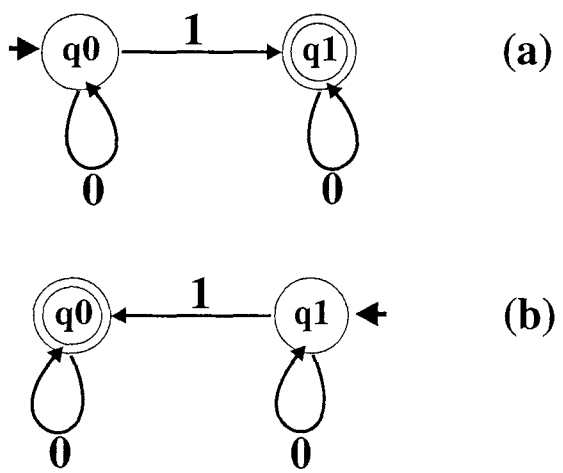


FIG. 2.5 - (a) Automate canonique du langage $L_{ex2} = 0^*10^*$ et (b) son automate miroir.

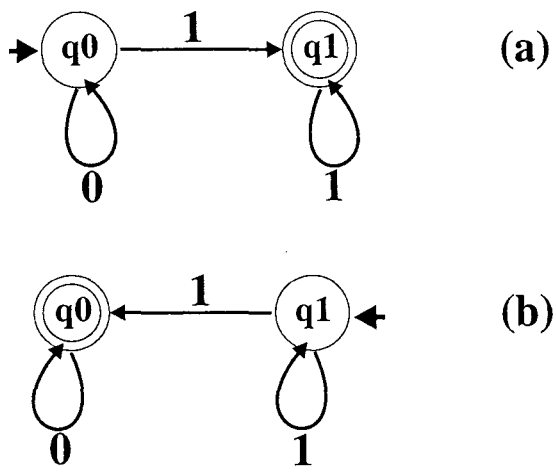


FIG. 2.6 - (a) Automate canonique du langage $L_{ex3} = 0^*1^+$ et (b) son automate miroir.

Exemple :
Le langage $L_{ex3} = 0^*1^+$ (voir figure 2.6 (a)) n'est pas 0-réversible puisque son automate miroir (voir figure 2.6 (b)) n'est pas déterministe. ◇

La classe des langages 0-réversibles a été étendue naturellement de manière à définir d'autres familles de langages « réversibles ». Cette extension est présentée ci-après.

Langages k-réversibles

Définition 2.1.2 *Soit A un automate, $A = (Q, I, F, \Sigma, \delta)$. Soit $u \in \Sigma^*$. Le mot u est un* k-suiveur *de $q \in Q$ si et seulement si $|u| = k$ et $\exists\ q' \in Q$, tel que $q' \in \delta(q, u)$. Le mot u est un* k-meneur *de q'.*

Remarque 2.1.3 Chaque état d'un automate a exactement un 0-suiveur et un 0-meneur qui est le mot vide ε.

Définition 2.1.3 *Soit un automate $A = (Q, I, F, \Sigma, \delta)$. Soit q_1, q_2 un couple d'états appartenant à Q. Le couple d'états distincts q_1, q_2 est dit* critique *si soit $q_1, q_2 \in I$ soit $\exists\ q_3 \in Q$ et $a \in \Sigma$ tels que $q_1, q_2 \in \delta(q_3, a)$.*

Définition 2.1.4 *Soit un automate $A = (Q, I, F, \Sigma, \delta)$. L'automate A est* déterministe à l'horizon k *si et seulement si pour tout couple critique d'états q_1, q_2, il n'existe aucune chaîne $u \in \Sigma^*$ qui soit à la fois k-suiveur de q_1 et de q_2.*

Intuitivement, cela signifie que l'indéterminisme peut être levé avec des mots de longueur supérieure à k.

Étant donnés un automate A et une valeur de k, on peut décider si A est ou non déterministe à l'horizon k. Pour ce faire, il suffit d'étudier pour toutes les paires d'états tous deux initiaux ou tous deux a-successeurs d'un même état ($a \in \Sigma$) s'il existe un mot qui est k-suiveur pour les deux états.

Définition 2.1.5 *Un automate A est* k-réversible *s'il est déterministe et si son automate miroir A^r est déterministe à l'horizon k.*

Le déterminisme à l'horizon k d'un automate A étant décidable, il en est de même pour la k-réversibilité.

Théorème 2.1.2 (Angluin, [Ang82a]) *Un langage régulier L est k-réversible si et seulement si son automate canonique est k-réversible.*

Exemple :
Le langage $L_{ex3} = 0^*1^+$ est 1-réversible puisque son automate canonique (voir figure 2.6 (a), page précédente) est 1-réversible. ◇

Théorème 2.1.3 (Angluin, [Ang82a]) *Soit $k \geq 0$, soit L un langage régulier. Si L est un langage k-réversible, alors L est $k+1$-réversible.*

La preuve du théorème 2.1.3 est immédiate en utilisant les définitions de k-réversibilité.

Remarque 2.1.4 Pour tout $k \geq 0$, il existe des langages $k+1$-réversibles qui ne sont pas k-réversibles. Le langage $1^{k+1}1^*$ en est un exemple.

Définition 2.1.6 *Un automate A est* réversible *si et seulement si il existe un $k \geq 0$ tel que A est k-réversible.*

Exemple :
Le langage $L_{ex4} = 0^*(1+\varepsilon)0^*$ dont l'automate canonique est représenté à la figure 2.7 (a) n'est pas réversible. En effet, pour toute valeur de k, les mots 0^k sont k-suiveurs des deux états initiaux q_0 et q_1 pour l'automate miroir (voir figure 2.7 (b)) ◇

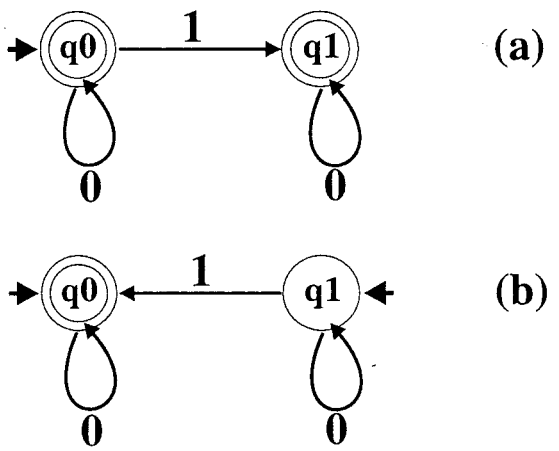


FIG. 2.7 - (a) Automate canonique du langage $L_{ex4} = 0^*(1+\varepsilon)0^*$ et (b) son automate miroir.

Définition 2.1.7 *On note $\mathcal{R}ev_k$ la classe des langages k-réversibles, avec $k \geq 0$ fixé. La classe des langages réversibles, notée $\mathcal{R}ev$, est définie par :*

$$\mathcal{R}ev = \bigcup_{k=0}^{\infty} \mathcal{R}ev_k$$

2.1.4 Décidabilité de la réversibilité

Nous avons vu dans ce qui précède que pour tout automate A et pour toute valeur de k, on peut décider de la k-réversibilité de A. Nous allons voir ici que le caractère de réversibilité d'un automate est également décidable ; il s'agit donc de donner un algorithme qui, pour tout automate A retourne si elle existe, la valeur du plus petit k tel que A est k-réversible et qui retourne faux sinon. Précisons qu'à notre connaissance, il n'existe pas dans la littérature de tels résultats de décidabilité sur la classe $\mathcal{R}ev$.

Un automate est réversible s'il est déterministe et s'il existe une valeur de k telle que son automate miroir est déterministe à l'horizon k. La définition suivante permet de parler de *déterminisme étendu* de la même manière que l'on parle de réversibilité.

Définition 2.1.8 *Soit A un automate. On dit que A est* déterministe étendu *s'il existe un $k \geq 1$ tel que A est déterministe à l'horizon k.*

Dans un premier temps, nous établissons que le déterminisme étendu est décidable. Pour montrer ce résultat, il convient cependant d'introduire quelques définitions et lemmes préliminaires.

Définition 2.1.9 *Soit un automate $A = (Q, I, F, \Sigma, \delta)$. Soit $q \in Q$. On appelle* langage de l'état q, *noté $L_q(A)$ le langage reconnu par l'automate $A_q = (Q, \{q\}, Q, \Sigma, \delta)$.*

Remarque 2.1.5 La construction de l'automate A_q peut se faire en temps polynomial par rapport au nombre d'états de A puisqu'il suffit de positionner tous les états de A comme états terminaux et de positionner q comme seul état initial. Remarquons également que le langage $L_q(A)$ n'est pas vide puisqu'il contient au moins le mot vide (dans l'automate A_q, l'état q étant à la fois état initial et terminal).

Lemme 2.1.4 *Soit un automate $A = (Q, I, F, \Sigma, \delta)$. Soit $q \in Q$. Le mot $u \in \Sigma^*$ appartient à $L_q(A)$ si et seulement si il existe $q' \in Q$ tel que $q' \in \delta(q, u)$.*

Preuve :
Évident par construction de A_q l'automate reconnaissant $L_q(A)$. □

Définition 2.1.10 *Soit L un langage fini non vide. On appelle* plus grande longueur de L, *notée $PGL(L)$, la longueur du plus long mot de L, c'est-à-dire $PGL(L) = \max\{|u| \mid u \in L\}$.*

Le lemme suivant permet de caractériser les automates déterministes étendus. Cette caractéristique sera ensuite utilisée afin de décider du déterminisme étendu d'un automate.

Lemme 2.1.5 *Soit un automate $A = (Q, I, F, \Sigma, \delta)$. L'automate A est déterministe étendu si et seulement si pour tout couple critique d'états q_1, q_2 alors $L_{q_1}(A) \cap L_{q_2}(A)$ est un langage fini.*

Preuve :
Soit un automate $A = (Q, I, F, \Sigma, \delta)$.

- **Supposons que A est déterministe étendu. Montrons alors que pour tout couple critique d'états q_1,q_2, le langage $L_{q_1}(A) \cap L_{q_2}(A)$ est fini.** La preuve

se fait par l'absurde. D'abord, par hypothèse, l'automate A est déterministe étendu, il existe donc un $k \geq 0$ tel que A est déterministe à l'horizon k. Maintenant, supposons qu'il existe un couple critique d'états q_1 et q_2 tel que le langage $L_{q_1}(A) \cap L_{q_2}(A)$ est infini. Pour tout $l \geq 0$, il existe donc un mot de longueur l appartenant au langage $L_{q_1}(A) \cap L_{q_2}(A)$. Ceci est donc vrai pour $l = k$. Par conséquent, il existe un mot u de longueur k tel que $u \in L_{q_1}(A) \cap L_{q_2}(A)$, soit encore $u \in L_{q_1}(A)$ et $u \in L_{q_2}(A)$. D'après le lemme 2.1.4 (page précédente), il existe $q_1' \in Q$ tel que $q_1' \in \delta(q_1, u)$ et il existe $q_2' \in Q$ tel que $q_2' \in \delta(q_2, u)$ ce qui est impossible puisque l'automate A est déterministe à l'horizon k.

- **Supposons que pour tout couple critique d'états q_1,q_2, le langage $L_{q_1(A)} \cap L_{q_2}(A)$ est fini. Montrons qu'alors l'automate A est déterministe étendu.** Soit l'ensemble des langages $\mathcal{L} = \{L_{q_1}(A) \cap L_{q_2}(A) \mid$ le couple d'états q_1, q_2 est critique$\}$. Par hypothèse, pour tout couple critique d'états q_1, q_2, le langage $L_{q_1}(A) \cap L_{q_2}(A)$ est fini non vide. Il existe donc une plus grande longueur pour chacun de ces langages. Soit $k = \max\{PGL(L) \mid L \in \mathcal{L}\}$. Pour tout couple critique d'états q_1, q_2, il n'existe donc aucun mot de longueur supérieure strictement à k qui soit k-suiveur à la fois de q_1 et de q_2. L'automate A est donc déterministe à l'horizon $k+1$. Ce qui termine la preuve.

□

En utilisant ce dernier lemme, il est dès lors très facile de pouvoir décider si un automate est déterministe étendu.

Théorème 2.1.6 *Soit un automate A à n états. Il existe un algorithme décidant en temps $O(n^4)$ si l'automate A est déterministe étendu.*

Preuve :
Nous proposons un algorithme prenant en entrée un automate A et retournant si elle existe la valeur du plus petit k tel que A est déterministe à l'horizon k et retournant -1 sinon.

Algorithme `Decide_Determinisme_Etendu`
Entrée: Automate A
Début
  Soit $k = 0$
  Pour tout couple critique d'états q_1, q_2
    Soit $L_{q_1,q_2} = L_{q_1}(A) \cap L_{q_2}(A)$
    Si le langage L_{q_1,q_2} est fini
    Alors $k = \max(PGL(L_{q_1,q_2}), k)$
    Sinon retourner -1
  Fpour
  retourner k
Fin

La preuve de correction de cet algorithme est évidente d'après le lemme 2.1.5 (page précédente). Remarquons que cet algorithme retourne la valeur 0 si l'automate est déterministe. puisque dans ce cas il n'existe pas de couple critique d'états.

Complexité en temps de l'algorithme
Soit n le nombre d'états de A, l'automate d'entrée. Le nombre maximal de couples critiques d'états est n^2. Pour chacun des couples critiques d'états q_1, q_2, on construit le langage $L_{q_1,q_2} = L_{q_1}(A) \cap L_{q_2}(A)$; plus précisément, on construit un automate A_{q_1,q_2} reconnaissant le langage L_{q_1,q_2}. Ceci peut se faire en temps $O(n^2)$. En effet, on construit d'abord en temps $O(n)$ les deux automates A_{q_1} et A_{q_2}. Chacun de ces automates a n états. La construction de l'automate intersection A_{q_1,q_2} prend donc un temps en $O(n^2)$. Pour vérifier que le langage reconnu par cet automate est fini, il suffit de vérifier que l'automate A_{q_1,q_2} ne contient pas de boucles, ce qui peut être fait en temps $O(n^2)$. Dans le cas où le langage L_{q_1,q_2} est fini, la longueur du plus long mot appartenant à ce langage est au maximum $n^2 - 1$ (puisque l'automate A_{q_1,q_2} a au plus n^2 états). Aussi, trouver la valeur du plus long mot reconnu par A_{q_1,q_2} peut se faire en temps $O(n^2)$. Ceci montre finalement que l'algorithme a une complexité en temps en $O(n^4)$.
□

Remarque 2.1.6 Si l'automate A a n états, alors les automates reconnaissant les langages $L_{q_1} \cap L_{q_2}$ ont au maximum n^2 états puisque chacun des automates du type A_q a n états. Ainsi, si l'automate A est déterministe étendu, tous les langages intersection considérés sont finis et reconnus par un automate à au plus n^2 états. La longueur du plus long mot de ces langages est donc au maximum $n^2 - 1$. Ceci signifie donc qu'un automate A à n états non déterministe à l'horizon $n^2 - 1$ n'est par déterministe étendu. Ce résultat permet donc d'imaginer un autre algorithme décidant du déterminisme étendu et testant de manière incrémentale pour k variant de 1 à $n^2 - 1$ le déterminisme à l'horizon k.

Nous pouvons maintenant donner le résultat de décidabilité sur la propriété de réversibilité d'un automate.

Théorème 2.1.7 *Soit A un automate quelconque à n états. Il existe un algorithme qui décide en temps $O(n^4)$ s'il existe une valeur de k telle que A est k-réversible.*

Preuve :
La preuve est évidente au vu des résultats précédents. L'algorithme prend en entrée un automate A et retourne si elle existe la valeur du plus petit k tel que A est k-réversible et retourne -1 sinon.

Algorithme `Decide_Reversibilité`
Entrée: Automate A
Début
  Si A est déterministe
  Alors retourner `Decide_Determinisme_Etendu`(A^r)
  Sinon retourner -1
  Fsi
Fin

La preuve de correction est ici encore évidente puisqu'un automate est réversible si et seulement si il est déterministe (ce qui est vérifié par le test) et si son automate miroir est déterministe étendu. Remarquons que si A est 0-réversible, son automate miroir est déterministe et l'algorithme `Decide_Determinisme_Etendu` retourne bien la valeur 0.

Complexité de l'algorithme

Soit n le nombre d'états de l'automate A passé en entrée à l'algorithme. Pour vérifier que l'automate A est déterministe, il suffit de vérifier que tout état a au plus une transition par chacune des lettres de l'alphabet, ce qui prend un temps $O(n)$. La complexité de l'algorithme est donc dominée par celle de l'algorithme `Decide_Determinisme_Etendu` soit $O(n^4)$ puisque l'automate miroir de A a également n états. □

2.1.5 Quelques résultats sur les langages réversibles

Proposition 2.1.1 *Soit L un langage fini non vide. Il existe $k \geq 0$ tel que L est un langage k-réversible.*

Preuve :

La preuve est assez évidente. D'après le théorème 2.1.2 (page 59), il suffit de montrer qu'il existe un entier k tel que l'automate canonique du langage L est k-réversible. L étant un langage fini, $A(L) = PT(L)$. L'automate $PT(L)^r$ reconnaît un langage fini et est au plus déterministe à l'horizon n, avec n la longueur du plus long mot de L. □

Le théorème suivant, du à Angluin [Ang82a], trouvera son intérêt dans la suite.

Théorème 2.1.8 (Angluin, [Ang82a]) *Soit L un langage régulier. L est k-réversible si et seulement si quand $u_1vw \in L$ et $u_2vw \in L$ et $|v| = k$, $T_L(u_1v) = T_L(u_2v)$.*

Exemple :

Le langage $L_{ex3} = 0^*1^+$ est 1-réversible (voir figure 2.6, page 58). Les mots 111 et 000111 appartiennent tous deux au langage L_{ex3}. Posons $u_1 = \varepsilon$, $u_2 = 00$, $v = 1$ et $w = 111$. Alors $T_{L_{ex3}}(u_1v) = T_{L_{ex3}}(1) = 1^*$ et $T_{L_{ex3}}(u_2v) = T_{L_{ex3}}(001) = 1^*$. ◇

Le théorème suivant donne quelques propriétés de clôture sur les classes $\mathcal{R}ev_k$ et $\mathcal{R}ev$.

Théorème 2.1.9 (Angluin, [Ang82a])

- *$\mathcal{R}ev_k$ est proprement contenue dans $\mathcal{R}ev_{k+1}$,*
- *$\mathcal{R}ev_k$ est close sous l'intersection,*
- *$\mathcal{R}ev_k$ est close pour l'opération miroir,*
- *$\mathcal{R}ev$ n'est pas close sous le complémentaire,*
- *$\mathcal{R}ev$ n'est pas close sous l'union,*
- *$\mathcal{R}ev$ n'est pas close sous la concaténation,*

- *$\mathcal{R}ev$ n'est pas close sous la clôture de Kleene.*

Dans son article, Angluin compare la classe $\mathcal{R}ev$ avec d'autres classes de langages réguliers connues. Par exemple, elle montre que la classe $\mathcal{R}ev$ contient proprement la classe des *langages bien déterminés*[5] définie par Perles, Rabin et Shamir [PRS63]. Un langage régulier L est *k-bien déterminé* si et seulement si quand deux mots u_1 et u_2 ont un suffixe commun de longueur k alors $T_L(u_1) = T_L(u_2)$.

Les deux résultats suivant montrent qu'il est possible de construire des langages réversibles à partir de langages finis.

Lemme 2.1.10 (Angluin, [Ang82a]) *Soient F un langage fini et L un langage réversible. Alors le langage $F \cup L$ est réversible.*

Théorème 2.1.11 *Soient E, F et G trois langages finis sur l'alphabet Σ. Alors, le langage $E \cup F\Sigma^*G$ est un langage réversible.*

Ce dernier théorème est une première étape dans la comparaison entre les langages réversibles et les *langages bien déterminés généralisés* définis par Ginzburg [Gin66]. Un langage est bien déterminé généralisé si et seulement si il peut être exprimé comme l'union d'un langage fini et l'union finie de langages de la forme $F\Sigma^*G$ où F et G sont des langages finis sur l'alphabet Σ. Le théorème 2.1.11 établit donc que la classe $\mathcal{R}ev$ contient quelques langages bien déterminés généralisés. Cependant, Angluin montre sur un exemple simple que la classe des langages réversibles et celle des langages bien déterminés généralisés sont incomparables puisque chacune de ces classes contient des langages n'appartenant pas à l'autre.

Cette comparaison succincte entre la classe des langages réguliers et d'autres classes de langages montre donc que $\mathcal{R}ev$ contient un grand nombre de langages réguliers connus. Ceci assied donc le fait qu'étudier l'apprenabilité de la classe $\mathcal{R}ev$ n'est pas sans intérêt ne serait-ce que parce que cette classe contient un grand nombre de langages « classiques ».

2.2 Inférence des Langages Réguliers dans le Paradigme de Gold

2.2.1 Présentation du problème

Replaçons le problème de l'apprentissage automatique dans le cadre des langages réguliers. Il s'agit, ici encore, d'inférer à partir d'un ensemble d'exemples, un concept généralisant ces exemples. Dans ce contexte, le domaine $X = \Sigma^*$, la classe de concepts $F = \mathcal{R}eg$, les exemples étant, quant à eux, des éléments de $\Sigma^* \times I\!B$. En ce qui concerne l'ensemble des représentations associé à $\mathcal{R}eg$, plusieurs choix s'offrent à nous ; en effet, on peut décider de coder les langages réguliers au moyen des **expressions régulières**, des **grammaires régulières** ou bien encore des **automates finis déterministes**. C'est cette dernière option qui sera, par la suite, utilisée. Ainsi, R l'ensemble des représentations associé à la classe de concepts $\mathcal{R}eg$ est l'ensemble des DFAs, noté $\mathcal{DFA}$. Remarquons que ces deux ensembles sont étroitement liés ; par conséquent, la distinction entre classe de concepts et ensemble de représentations sera, en général, superflue. Constatons de plus que la classe de représentations $\mathcal{DFA}$ est calculable en temps polynomial. En effet, étant donnée une paire $(u, A) \in \Sigma^* \times$

5. En anglais *definite languages*.

$\mathcal{DFA}$, on peut décider en temps polynomial si u est un élément d'un langage L avec A une représentation de L.

Nouvelle formulation du modèle de Gold

Le protocole d'apprentissage incrémental associé au modèle de Gold et présenté à la section 1.2.1 (page 29) ne permet pas toujours d'étudier facilement le comportement de certains algorithmes. En effet, le caractère infini du processus est directement intégré à l'algorithme d'apprentissage qui a la charge de demander un nouvel exemple à chaque étape.

Gold [Gol78] propose un nouveau protocole d'apprentissage, appelé *données fixées*[6], dans lequel l'ensemble des exemples est fourni à l'algorithme à chaque étape de l'apprentissage. Le processus reste infini mais la boucle n'est plus intégrée directement à l'algorithme d'inférence. Ainsi, étant donné un langage cible L, un échantillon d'apprentissage pour L est donné en entrée à l'algorithme d'apprentissage. Celui-ci propose une hypothèse $h(S)$, compatible avec cet échantillon. L'algorithme identifie à la limite le concept L, s'il existe un échantillon S_L pour L tel que $\forall\ S \supseteq S_L, h(S) = h(S_L) = L$.

L'intérêt de cette variante dans l'énoncé du modèle de Gold est triple. D'abord, la définition d'algorithmes selon ce protocole semble plus facile à établir et donc à implanter. Ensuite, de par leur utilisation, les algorithmes d'apprentissage semblent très proches des algorithmes de consistance (voir section 1.1.3, page 27) ; ainsi, on pourra éventuellement envisager d'étudier le comportement d'un même algorithme à la fois dans le modèle de Gold et dans le modèle PAC. Enfin, cette variante permet de définir la notion d'*échantillon représentatif d'un concept* L. Intuitivement, l'échantillon S_L est représentatif pour le concept L si c'est le (un) plus petit échantillon permettant de générer L. Remarquons que la propriété de représentativité d'un échantillon dépend non seulement du concept cible, mais aussi de l'algorithme d'apprentissage[7]. De nombreux algorithmes se basent sur cette idée d'échantillon représentatif afin d'atteindre leur but. Nous verrons qu'en général, il est indispensable de disposer d'un tel échantillon.

Étant donné un algorithme de consistance $\mathtt{A}$, il sera parfois utile de parler de la version incrémentale de $\mathtt{A}$, notée $\mathtt{A}_\infty$. Cette version de l'algorithme peut être vue comme une boucle infinie qui tire un nouvel exemple à chaque étape, l'ajoute à un échantillon S puis propose l'hypothèse retournée par $\mathtt{A}(S)$.

Par la suite, nous nous proposons d'étudier l'apprentissage des langages réguliers dans le modèle de Gold selon ce protocole. Quelques algorithmes seront présentés ainsi que la définition d'échantillon représentatif pour ceux-ci. Deux situations d'apprentissage seront envisagées: d'abord celle où l'apprentissage se fait par présentation *complète* des exemples, c'est-à-dire que la totalité des exemples possibles (positifs et négatifs) est susceptible d'être fournie à l'algorithme. Le cas où seuls des exemples positifs sont manipulés par l'algorithme sera ensuite présenté.

2.2.2 Apprentissage par présentation complète des exemples

La question qui se pose ici est de savoir si la classe des langages réguliers est ou non inférable à la limite. Dès 1967, Gold [Gol67] a montré que toute classe de grammaires *ad-*

6. En anglais, *given data*.

7. Pour l'algorithme d'apprentissage, s'attendre à recevoir un échantillon représentatif peut être un élément important des connaissances *a priori*.

missible est identifiable à la limite par présentation complète d'exemples, répondant ainsi par l'affirmative à cette question puisque la classe $\mathcal{R}eg$ est admissible. Rappelons qu'une classe de grammaires C est admissible si elle est dénombrable et si pour toute grammaire $G \in C$ et pour tout mot x, on peut décider si $x \in L(G)$ (c'est à dire si le mot x appartient ou non au langage engendré par la grammaire).

Cependant, bien que la classe $\mathcal{R}eg$ soit théoriquement inférable à la limite par échantillon complet, il subsiste, en pratique, quelques difficultés pour trouver la meilleure hypothèse. Le théorème de Gold suppose implicitement qu'il est toujours possible d'écrire un algorithme recherchant, à chaque étape, le premier langage compatible avec l'échantillon donné. La classe des langages réguliers étant récursivement énumérable, ceci est toujours possible et l'on est assuré de la convergence à la limite du processus vers le concept cible. Cependant, cette technique n'est pas satisfaisante car non utilisable dans la pratique. De plus, son côté « recherche exhaustive » semble lui interdire la qualité d'algorithme d'apprentissage.

Il convient donc, avant tout, de réduire l'espace de recherche du meilleur langage. Une première idée consiste à limiter la recherche de l'hypothèse à l'ensemble des langages compatibles avec S_+ et à éliminer ceux qui ne sont pas consistants avec S_-. Parmi les hypothèses qui peuvent être proposées par une telle méthode, certaines ont cependant l'inconvénient de ne pas généraliser l'échantillon. En effet, étant donné un échantillon $S = S_+ + S_-$, l'automate arbre préfixe $PT(S_+)$ est consistant avec S mais ne possède pas le pouvoir de généralisation attendu.

Il s'agit donc de donner un critère de choix, permettant de sélectionner, parmi l'ensemble des langages consistants avec l'échantillon, celui qui généralisera au mieux cet échantillon. Ici encore, le principe du rasoir d'Occam s'applique puisque le choix portera sur le plus petit DFA consistant avec l'échantillon. En fait, tout algorithme de consistance capable de construire le plus petit DFA consistant avec l'échantillon d'entrée peut être dérivé dans une version incrémentale afin d'identifier à la limite tout langage régulier par présentation complète d'exemples :

Théorème 2.2.1 *Soit* A *un algorithme de consistance retournant le plus petit DFA consistant avec l'échantillon d'entrée. Alors* $\mathtt{A}_\infty$*, la version incrémentale de* A*, identifie à la limite la classe* $\mathcal{R}eg$ *par présentation complète d'exemples.*

La preuve de ce théorème est assez évidente. En effet, par définition, une présentation complète d'exemples code exactement le langage cible. Le plus petit DFA reconnaissant un langage étant l'automate canonique de ce langage, on est assuré qu'à la limite, l'algorithme retournera cet automate.

Cependant, Gold a montré que :

Théorème 2.2.2 (Gold, [Gol78]) *Le problème de trouver, pour un échantillon fini S donné et un entier positif t, un automate fini déterministe d'au plus t états consistant avec S est un problème NP-complet.*

En particulier, trouver le plus petit DFA consistant avec un échantillon S donné est un problème NP-dur. Cela implique donc une impraticabilité éventuelle du processus, le temps de calcul de l'algorithme de consistance à chaque étape pouvant être, dans certains cas, trop long.

Une idée pour contourner ce problème consiste à imposer à l'échantillon de contenir certains exemples qui guideront l'algorithme afin de générer (rapidement) le résultat escompté.

Plus généralement, on peut imposer à l'échantillon certaines contraintes, connues de l'algorithme, de sorte que celui-ci retourne en un temps raisonnable (*ie.* polynomial par rapport à la taille de l'échantillon) le meilleur concept.

C'est le cas de l'algorithme de Trakhtenbrot et Barzdin [TB73] qui construit le plus petit DFA complet consistant avec un échantillon *uniforme-complet.*

Algorithme de Trakhtenbrot-Barzdin

Définition 2.2.1 *Soit L un langage régulier. Soit S un échantillon relatif à L. L'échantillon S est* l-uniforme-complet *si* $\forall\ u \in \Sigma^{\leq l}, \exists\ e \in S$ *tel que* $e = (u, b)$ *et* $\forall\ u \in \Sigma^{>l}\ \nexists\ e \in S$ *tel que* $e = (u, b)$.

Définition 2.2.2 *Un échantillon S est* uniforme-complet *s'il existe $l \geq 0$ tel que S est l-uniforme-complet.*

La construction d'un automate arbre préfixe relatif à un ensemble de mots (voir section 2.1.2, page 56) peut être étendue à celle d'un automate arbre préfixe relatif à un échantillon labellé $S = S_+ + S_-$. Il s'agit de l'automate $PT(S) = (Pr(S_+ + S_-), \{\varepsilon\}, S_+, \Sigma, \delta)$ avec $\delta(u, a) = ua$ $(u, ua \in Pr(S_+ + S_-)$ et $a \in \Sigma)$.

Pour un tel automate, chaque mot de l'ensemble S (et uniquement eux) labelle un chemin minimal depuis l'état initial vers un état de l'automate. De plus, seul les états relatifs aux mots de S_+ sont terminaux. Il est évident qu'en général, cet automate arbre préfixe n'est pas minimal. De plus, le langage qu'il reconnaît est égal à S_+.

Remarque 2.2.1 Pour un échantillon l-uniforme-complet, le nombre d'états de l'automate $PT(S)$ est égal à $\frac{|\Sigma|^l - 1}{|\Sigma| - 1}$.

L'algorithme de Trakthenbrot-Barzdin (voir figure 2.8page suivante) prend en entrée l'automate arbre préfixe d'un échantillon S supposé uniforme-complet et retourne en un temps $O(mn^2)$ le plus petit DFA complet consistant avec S, où m est le nombre d'états de $PT(S)$ et n est le nombre d'états de l'automate résultat. Le principe de celui-ci consiste à fusionner ensemble toutes les paires d'états q_a et q_b si les sous-arbres de racines q_a et q_b sont isomorphes. Il effectue donc implicitement la recherche d'un automate dans le treillis $Lat(PT(S))$. La procédure `sous_arbre(`i`)` retourne le sous-arbre dont la racine est le nœud i. Si les deux sous-arbres n'ont pas la même profondeur, la comparaison ne prend pas en compte les états situés à une profondeur supérieure à celle du plus petit sous-arbre. La procédure `fusionne(`i, j`)` remplace la transition entre le père de i et i (une seule transition possible puisque à ce niveau l'automate a encore une structure d'arbre) par une transition de même label entre le père de i et j. L'état i et tous ses fils deviennent donc inaccessibles.

Étant donné un langage régulier L, les deux définitions qui suivent permettent de caractériser un échantillon uniforme-complet particulier relatif au langage L. Cet échantillon jouera le rôle d'échantillon représentatif du langage L pour l'algorithme de Trakhtenbrot et Barzdin.

Définition 2.2.3 *Soit $A = (Q, \{q_0\}, F, \Sigma, \delta\}$ un automate fini. La* profondeur *de A, notée $d(A)$, est la longueur de la plus longue chaîne canonique (*ie. *sans boucle) menant de l'état initial à un état quelconque : $d(A) = Max_{q \in Q}\{Min_{s \in \Sigma^*}\{|s|\ |\ \delta(q_0, s) = q\}\}$*

```
Algorithme Trakhtenbrot-Barzdin
Entrée: A = PT(S) avec S un échantillon uniforme-complet
Début
    Pour i parcourant en largeur d'abord tous les nœuds de A
        Pour j de racine à i − 1
            Si sous_arbre(j) = sous_arbre(i)
            Alors fusionne(i, j)
            Fsi
        Fpour
    Fpour
    retourner A
Fin
```

FIG. 2.8 - L'Algorithme de Trakhtenbrot et Barzdin (T_B).

Définition 2.2.4 *Soit $A = (Q, \{q_0\}, F, \Sigma, \delta\}$ un automate fini. Le* degré de distingabilité *de A, noté $\rho(A)$ est le plus petit entier i tel que pour chaque paire d'états différents, il existe un suffixe de longueur inférieure ou égale à i qui envoie un et un seul des états sur un état terminal.*

Soit L un langage régulier. Soit $A(L)$ l'automate canonique associé à L. Si S_L est un échantillon l-uniforme-complet pour L avec $l = d(A(L)) + \rho(A(L)) + 1$ alors l'algorithme de Trakhtenbrot-Barzdin retourne exactement $A(L)$. S_L est donc un échantillon représentatif de L pour l'algorithme de Trakhtenbrot-Barzdin que l'on peut noter S_L^{TB}. En utilisant le théorème 2.2.1 (page 66), l'algorithme de Trakhtenbrot-Barzdin peut donc être utilisé pour identifier à la limite et par présentation d'exemples complète tout langage régulier.

Notons que dans le pire des cas, $l = 2|A(L)| - 1$. De plus, la taille de l'échantillon croît exponentiellement avec n la taille de $A(L)$; il n'est cependant pas envisageable de réduire la taille de l'échantillon, Angluin ayant montré que l'apprentissage exact devenait impossible dès lors que l'on supprime une toute petite fraction d'exemples de l'échantillon uniforme-complet [Ang78]. Ce résultat d'Angluin est dû au fait qu'il est toujours possible d'utiliser un automate particulièrement bien choisi ainsi qu'un échantillon dont on aurait retirer les exemples les plus « utiles » en vue de faire échouer l'algorithme. Il s'agit par conséquent d'un résultat basé sur le pire des cas. En fait, Lang [Lan92] montre de manière empirique, qu'il est possible d'apprendre approximativement un DFA en utilisant un échantillon peu dense, si l'automate et l'échantillon sont tirés de manière aléatoire (il s'agit donc d'un résultat en moyenne). L'algorithme présenté par Lang est un algorithme de type glouton fortement inspiré de l'algorithme de Trakhtenbrot-Barzdin : la totalité des possibilités de fusion est évaluée et l'algorithme « backtrack » si le choix est incorrect. L'automate retourné n'est pas nécessairement minimal mais Lang montre de manière empirique que l'hypothèse retournée approxime l'automate cible.

Dans l'algorithme de Trakhtenbrot-Barzdin, nous avons vu que la taille de l'échantillon peut être importante (car il doit être uniforme-complet) ce qui implique une taille importante de l'automate arbre préfixe. L'espace de recherche (c'est-à-dire le treillis engendré par cet automate) est donc lui-aussi de taille importante. La taille de l'automate arbre préfixe peut

être réduite de deux manières complémentaires. D'abord, on peut éviter de considérer les exemples négatifs dans la construction de l'arbre. Ceux-ci peuvent être pris en considération de manière indépendante. Ensuite, on peut définir d'autres critères de représentativité de l'échantillon, en ajoutant, par exemple, de l'information relative à l'automate canonique du langage cible.

Des algorithmes tels que `RPNI` [OG92], `BRIG` [MdG94], `GIG` [Dup94] s'appuient entre autre sur ces deux idées afin de réduire l'espace de recherche. Ils se basent notamment sur la notion d'échantillon structurellement complet afin d'atteindre cet objectif. Nous présentons la définition de ce type d'échantillon ainsi qu'un résultat important sur lequel est bâti la plupart des algorithmes d'inférence de langages réguliers. Nous étudierons ensuite l'un de ces algorithmes : `RPNI`.

Échantillon structurellement complet

La définition d'*échantillon structurellement complet* pour un automate A permet d'assurer que les éléments clés de la structure de A seront intégrés à l'échantillon. Plusieurs auteurs [FB75, Mic79, Mic80, Ang82a] proposent une telle définition d'échantillon structurellement complet :

Définition 2.2.5 *Un échantillon S_+ est* structurellement complet *relativement à un automate A si toutes les transitions de A sont utilisées dans l'acceptation d'au moins un mot de S_+.*

Par ce biais, toutes les transitions de l'automate A sont « codées » dans un tel échantillon. Cependant, pour certains algorithmes cette information n'est pas encore suffisante. Dans ce cas, il peut être nécessaire d'obliger que l'ensemble des états terminaux de A soit utilisé comme état d'acceptation pour au moins un mot de S_+ [DMV94]. Par la suite, nous utiliserons cette seconde définition.

De nombreux algorithmes d'inférence de langages réguliers sont basés sur les mêmes principes. Ils reposent essentiellement sur un théorème du à Dupont, Miclet et Vidal [DMV94].

Théorème 2.2.3 (Dupont, Miclet et Vidal [DMV94]) *Soit S_+ un échantillon positif relativement à un langage régulier L. Soit $A(L)$ l'automate canonique reconnaissant L. Si S_+ est structurellement complet relativement à $A(L)$ alors $A(L)$ appartient à $Lat(PT(S_+))$.*

Ce théorème permet donc de réduire l'espace de recherche dans lequel l'algorithme doit trouver la meilleure hypothèse. Ainsi, dès lors que l'on est assuré que l'ensemble S_+ de l'échantillon $S = S_+ + S_-$ est structurellement complet pour l'automate canonique reconnaissant la cible, l'algorithme peut se contenter d'évaluer la consistance de S_- uniquement avec les automates appartenant à $Lat(PT(S_+))$. De plus, il est suffisant de faire la recherche dans $Lat(PT(S_+))$ si l'on désire inférer l'automate canonique reconnaissant le langage cible.

Remarque 2.2.2 Si l'on n'impose pas cette contrainte, et que l'on désire, par exemple, inférer exactement la cible (non obligatoirement canonique) la recherche doit se faire dans un espace plus large : $Lat(MCA(S_+))$, où $MCA(S_+)$ est l'automate canonique maximal relativement à S_+. Il s'agit de l'automate (non obligatoirement déterministe) ayant le plus grand nombre d'états utiles et pour lequel S_+ est structurellement complet (voir [DMV94] ou [Dup94] pour plus de précisions à ce sujet).

Les algorithmes d'inférence pourront donc rechercher dans le treillis $PT(S_+)$ le DFA optimal consistant avec S_-. Les divers algorithmes diffèrent entre eux sur la manière de parcourir le treillis.

Nous présentons dans la suite l'algorithme RPNI (Regular Positive and Negative Inference) [OG92]. Celui-ci, outre le fait d'utiliser la démarche précédemment évoquée, présente l'avantage d'apprendre exactement lorsque l'échantillon contient un ensemble représentatif d'exemples. Cette propriété permet donc à RPNI d'être utilisable dans le modèle de Gold.

L'algorithme RPNI

L'algorithme RPNI proposé par Oncina et Garcia [OG92][8] s'inspire fortement de l'algorithme de Trakhtenbrot et Barzdin (voir section 2.2.2, page 67). La recherche d'un automate consistant avec $S = S_+ + S_-$ va se faire en profondeur d'abord dans le treillis engendré par $PT(S_+)$, guidée par l'ensemble des exemples négatifs. Un certain nombre d'automates ne sont ainsi pas évalués car ils dérivent d'automates non compatibles avec S_-. L'algorithme retourne un DFA consistant avec l'échantillon S, qui n'est, en général, pas le plus petit DFA compatible avec S.

La version de l'algorithme RPNI présentée à la figure 2.9 est celle proposée par Parekh et Honavar [PH97].

Algorithme RPNI
Entrée : $S = S_+ + S_-$
Début
  $A = PT(S_+)$
  $\pi = \{\{0\}, \{1\}, \ldots, \{N-1\}\}$
  Pour $i = 1$ à $N - 1$
    Pour $j = 0$ à $i - 1$
      $\pi' = \pi \setminus \{B(i,\pi), B(j,\pi)\} \cup \{B(i,\pi) \cup B(j,\pi)\}$
      $A' = A/\pi'$
      $\pi'' = determinise(A')$
      $A'' = A'/\pi''$
      Si A'' est consistant avec S_-
      Alors
        $A = A''$
        $\pi = \pi''$
      Fsi
    Fpour
  Fpour
  retourner A
Fin

FIG. 2.9 - L'algorithme RPNI.

8. L'algorithme proposé indépendamment par Lang [Lan92] est une variante de RPNI.

L'algorithme calçule d'abord l'automate arbre préfixe A pour l'ensemble des mots positifs de l'échantillon. La partition initiale π correspond à la partition canonique de A. À chaque étape, l'algorithme essaye de raffiner la partition courante en fusionnant deux états de l'automate. Si l'automate résultant de cette fusion est compatible avec les exemples négatifs, cette partition est conservée en tant que partition courante et la procédure continue le raffinement. La procédure `determinise(`A`)` calcule la partition permettant de déterminiser l'automate A. En effet, une dérivation de A par une partition quelconque π' peut engendrer un automate non déterministe. Cet algorithme parcourt effectivement l'ensemble des automates compatibles avec S_+ en profondeur d'abord : sélectionner une partition signifie descendre d'un niveau dans le treillis d'automates. L'algorithme `RPNI` a une complexité en temps en $O(p^3n)$ avec $p = ||S_+||$ et $n = ||S_-||$.

Cet algorithme retourne l'automate canonique d'un langage L' consistant avec l'échantillon S. Pour un échantillon quelconque, l'automate retourné n'est pas, en général, le plus petit DFA compatible avec S. Cependant, pour tout langage L, il est possible de construire un échantillon représentatif $S_L^{\texttt{RPNI}}$ tel que si l'échantillon d'apprentissage S passé à l'algorithme `RPNI` contient $S_L^{\texttt{RPNI}}$ alors l'automate généré est exactement $A(L)$.

Définition 2.2.6 *Soit L un langage régulier, soit $A(L) = (Q, \{q_0\}, F, \Sigma, \delta)$ son automate canonique. Un échantillon $S = S_+ + S_-$ relatif à L est* représentatif de L pour `RPNI` *s'il satisfait les deux conditions suivantes :*

- *S_+ est structurellement complet relativement à $A(L)$,*
- *$\forall\ u \in SP(L), \forall\ v \in N(L)$ tel que $\delta(q_0, u) \neq \delta(q_0, v)$ alors $\exists\ w \in \Sigma^*$ tel que soit $uw \in S_+$ et $vw \in S_-$ soit $uw \in S_-$ et $vw \in S_+$*

La première propriété assure que l'automate $A(L)$ appartient à $Lat(PT(S_+))$ (voir théorème 2.2.3, page 69). La seconde propriété interdit la fusion de deux états distincts de $L(A)$. Une telle fusion ferait passer l'algorithme sur un niveau inférieur à celui sur lequel se trouve $A(L)$ dans le treillis, et empêcherait éventuellement l'algorithme de trouver $A(L)$.

Dans le pire des cas, Garcia et Oncina ont montré que la taille maximale d'un tel échantillon est en $O(|A(L)|^2)$.

Étant donnée l'existence, pour tout langage L, d'un échantillon représentatif pour `RPNI`, cet algorithme peut donc être utilisé comme algorithme d'identification à la limite par présentation complète d'exemples (par application du théorème 2.2.1 (page 66)). La version incrémentale de cet algorithme présente cependant un défaut. En effet, il ne permet pas de réutiliser l'hypothèse d'une étape i pour construire l'hypothèse de l'étape $i + 1$. En d'autres termes, il n'est pas suffisant de ne considérer que le nouvel exemple tiré à l'étape $i + 1$ et donc la totalité de l'échantillon doit être prise en compte à chaque étape. Dupont propose cependant une version incrémentale de `RPNI` [Dup96]. Dans [dlHOV96], De la Higuera, Oncina et Vidal montrent qu'il existe d'autres parcours possibles dans le treillis d'automates afin d'atteindre l'identification à la limite et proposent l'algorithme `DFA`*`infer`*.

Cette section a montré que l'identification de langages réguliers par présentation complète d'exemples est possible. Nous nous proposons d'étudier, dans la partie suivante, le cas où seuls des exemples positifs relatifs au langage cible sont présentés à l'apprenant. Rappelons que ce paradigme trouve un écho dans l'apprentissage naturel puisque, selon les psychologues, l'apprentissage de la langue maternelle ne se fait que par exemples positifs.

2.2.3 Identification à la limite par présentation positive complète

Rappelons d'abord l'un des résultats fondamentaux de Gold [Gol67] : une classe de concepts contenant tous les concepts finis plus -au moins- un concept infini n'est pas identifiable à la limite par exemples positifs seuls[9]. La classe des langages réguliers $\mathcal{R}eg$ satisfaisant cette propriété, elle n'est par conséquent pas identifiable à la limite par exemples positifs seuls.

Notons que le théorème 2.2.1 (page 66) ne peut effectivement pas s'appliquer dans ce cas, car le plus petit automate consistant avec tout échantillon positif de mots est l'automate universel. Il s'agit d'un cas typique de sur-généralisation.

Outre les méthodes heuristiques qui ont été proposées [FGHR69, Mic80], plusieurs travaux s'attachent à rechercher des sous-classes de $\mathcal{R}eg$ qui soient identifiables à la limite par exemples positifs seuls.

C'est précisément le cas d'Angluin [Ang82a] qui présente une famille d'algorithmes identifiant à la limite par exemples positifs seuls les classes de langages k-réversibles (voir section 2.1.3, page 57). Les algorithmes qu'elle décrit se basent sur des principes similaires à ceux que l'on utilise dans le cas d'identification par exemples positifs et négatifs. Nous verrons qu'ici encore, trouver un plus petit automate qui soit consistant avec un échantillon caractéristique relatif à un langage cible, permet d'identifier à la limite ce langage.

Cas des langages 0-réversibles : l'algorithme ZR

Pour une question de simplicité, considérons d'abord la famille la plus simple de langages réversibles, c'est-à-dire celle des langages 0-réversibles. L'algorithme ZR (voir figure 2.10, page suivante) proposé par Angluin [Ang82a] prend en entrée un échantillon de mots S_+ et retourne en un temps quasi linéaire par rapport $\|S_+\|$ un automate 0-réversible reconnaissant l'ensemble des mots de S_+.

Le principe (très simple) de l'algorithme consiste à fusionner l'automate arbre préfixe construit à partir de l'échantillon tant que l'on n'a pas un automate 0-réversible. Les ensembles $s[B,a]$ (resp. $p[B,a]$) contiennent les états a-successeurs (resp. a-prédécesseurs) du bloc B. Remarquons qu'au départ ces ensembles contiennent au plus un élément étant donnée la structure d'arbre de l'automate initial ; cette propriété sur les ensembles s et p sera assurée par la suite. La liste LISTE contient l'ensemble des paires d'états à fusionner pour assurer la 0-réversibilité de l'automate. Au départ, on sait que tous les états terminaux de l'automate doivent être fusionnés, d'où l'initialisation de la liste. Tant que cette liste n'est pas vide, il existe éventuellement des états empêchant la 0-réversibilité de l'automate. Après une fusion, la procédure s-UPDATE(B_1,B_2,a) (resp. p-UPDATE(B_1,B_2,a)) met LISTE à jour en y ajoutant le couple $(s[B_1,a],s[B_2,a])$ (resp. $(p[B_1,a],p[B_2,a])$) si $s[B_1,a]\cup s[B_2,a]$ (resp. $p[B_1,a]\cup,p[B_2,a]$) contient deux éléments; en effet, dans ce cas, les blocs B_1 et B_2 empêcheraient l'automate (resp. l'automate miroir) d'être déterministe. De plus, la procédure s-UPDATE(B_1,B_2,a) (resp. p-UPDATE(B_1,B_2,a)) définit $s[B_1\cup B_2,a]$ (resp. $p[B_1\cup B_2,a]$) comme étant égal à $s[B_1,a]$ (resp. $p[B_1,a]$) s'il n'est pas vide et à $s[B_2,a]$ (resp. $p[B_2,a]$) sinon. Ce nouvel ensemble a donc au plus un élément qui est un des représentants du bloc considéré.

Angluin montre que l'algorithme ZR calcule la partition π la plus fine telle que l'automate A_0/π est 0-réversible et que le langage $L(A_0/\pi)$ est le plus petit langage 0-réversible contenant S_+.

9. Gold donne le nom de *classe super-finie* à ce type de classes.

Algorithme ZR
Entrée: S_+ : un ensemble non vide de mots
Début
* Initialisation
Soit $A_0 = PT(S_+) = (Q_0, \{q_0\}, F_0, \delta_0)$
Soit π_0 la partition canonique de Q_0
Soit LISTE une liste vide de paires d'états
Pour $a \in \Sigma$
Pour $q \in Q_0$
$s[\{q\}, a] = \delta_0(q, a)$
$p[\{q\}, a] = \delta_0^r(q, a)$
Fpour
Fpour
Soit q' un élément de F_0
Pour $q \in F_0 - \{q'\}$
Ajouter à LISTE la paire (q', q)
Fpour

* Fusions
Soit $i = 0$
Tant que LISTE non vide
Dépiler (q_1, q_2) de LISTE
$B_1 \leftarrow B(q_1, \pi_i)$
$B_2 \leftarrow B(q_2, \pi_i)$
Si $B_1 \neq B_2$
Alors
$\pi_{i+1} \leftarrow (\pi_i \setminus \{B_1, B_2\}) \cup \{B_1 \cup B_2\}$
Pour $a \in \Sigma$
s-UPDATE(B_1, B_2, a)
p-UPDATE(B_1, B_2, a)
Fpour
$i \leftarrow i + 1$
Fsi
Ftq

* Résultat
Retourner A_0/π_i
Fin

FIG. 2.10 - L'algorithme ZR.

Problème de sur-généralisation

Jusqu'à présent, l'algorithme ZR ne peut pas être utilisé comme algorithme d'identification des langages 0-réversibles par exemples positifs. En effet, étant donné un échantillon positif S_+ relatif à un langage L, rien n'assure que $L(ZR(S_+)) = L$. De plus, l'apprentissage se faisant à partir d'exemples positifs seuls, on peut être confronté au problème de sur-généralisation. Dans le cas de l'apprentissage des langages réguliers, nous rappelons qu'il y a sur-généralisation lorsque le langage inféré est un langage strictement plus grand que le langage cible (pour une étude approfondie de la sur-généralisation dans le cas de l'apprentissage des langages voir [Ang80]). Remarquons que dans le cas des langages 0-réversibles, le cas extrême de sur-généralisation conduirait à générer le langage Σ^* lui-même, celui-ci étant 0-réversible. Afin de pallier ce problème, Angluin montre de manière constructive qu'il existe pour tout langage 0-réversible L un échantillon caractéristique S_L, tel que L est le plus petit langage 0-réversible contenant S_L.

Remarque 2.2.3 Comme nous l'avons déjà indiqué, pour des raisons de sur-généralisation, dans le cas de l'apprentissage par exemples positifs seuls, il n'est pas raisonnable de se satisfaire du plus petit automate consistant avec l'échantillon ; l'automate universel répondant toujours à ce critère. Par contre, exiger d'inférer le plus petit langage appartenant à la classe de concepts permet de prouver l'apprenabilité à la limite de la classe considérée. Notons encore que ce principe ne peut pas être appliqué à la classe $\mathcal{R}eg$ puisque le plus petit langage consistant avec un échantillon positif est l'échantillon lui-même.

Notation :
Soit $A = (Q, \{q_0\}, F, \Sigma, \delta)$ un automate déterministe. Soit $q \in Q$. On note $u(q)$, le plus petit mot $u \in \Sigma^*$ tel que $\delta(q_0, u) = q$. On note $v(q)$, le plus petit mot $v \in \Sigma^*$ tel que $\delta(q, v) \in F$.

Définition 2.2.7 *Soit L un langage 0-réversible et soit $A(L) = (Q, \{q_i\}, \{q_f\}, \Sigma, \delta)$ son automate canonique. L'*échantillon A_0-caractéristique *du langage L, noté S_L^0, est l'ensemble de mots défini par $S_L^0 = \{u(q)v(q) \mid q \in Q\} \cup \{u(q)av(q') \mid q, q' \in Q, a \in \Sigma, q' = \delta(q, a)\}$.*

Propriété 2.2.1 (Angluin, [Ang82a]) *Soit L un langage 0-réversible et soit S_L^0 son échantillon A_0-caractéristique. Le langage L est le plus petit langage 0-réversible contenant S_L^0.*

Exemple :
Soit le langage 0-réversible L_{ex5} constitué des mots contenant un nombre pair de 0 et un nombre pair de 1 ; $L_{ex5} = \{u \mid |u|_0 = 2 * l, |u|_1 = 2 * n$ avec $l, n \in I\!N\}$. L'automate canonique de L_{ex5} est représenté à la figure 2.11 (page suivante). L'échantillon A_0-caractéristique du langage L_{ex5} est l'ensemble $S^0_{L_{ex5}} = \{\varepsilon, 00, 11, 0101, 0110, 1010\}$. ◇

À la différence des échantillons caractéristiques qui ont été précédemment définis, un échantillon caractéristique pour un langage 0-réversible est défini indépendamment de tout algorithme. Cependant par définition des échantillons A_0-caractéristiques , si L est un langage 0-réversible alors ZR(S_L^0) retourne l'automate canonique de L. Le qualificatif d'échantillon A_0-caractéristique peut donc s'appliquer relativement à l'algorithme ZR.

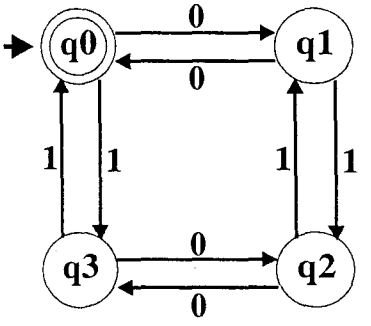


FIG. 2.11 - Automate canonique du langage $L_{ex5} = \{u \mid |u|_0 = 2 * l, |u|_1 = 2 * n,$ avec $l, n \in I\!N\}$.

Identification à la limite

Étant donnée l'existence d'un tel échantillon caractéristique, une version incrémentale de l'algorithme ZR, notée $\mathtt{ZR}_\infty$, peut donc être utilisée pour identifier à la limite la classe des 0-réversibles par présentation complète d'exemples positifs. En effet, dans ce paradigme, tout exemple positif d'un langage cible 0-réversible L est présenté au bout d'un nombre fini d'étapes. Il existe donc une étape pour laquelle la totalité des mots de S_L^0 sera fournie à l'algorithme qui proposera alors l'automate $A(L)$. La sur-généralisation est de ce fait évitée. Notons de plus qu'à chaque étape i du processus, $\mathtt{ZR}_\infty$ utilise l'hypothèse H_{i-1} inférée à l'étape précédente pour construire la nouvelle hypothèse. Ainsi, le nouvel exemple est intégré à l'hypothèse H_{i-1}, ce qui donne un automate H'_{i-1}. L'algorithme ZR est ensuite lancé sur ce nouvel automate afin de construire H_i.

Cas des langages k-réversibles

Angluin a étendu ses travaux aux cas des langages k-réversibles ($k \geq 0$ fixé) et propose l'algorithme k-RI qui prend en entrée un ensemble de mots S_+ et retourne, en un temps $O(k\|S_+\|^3)$, le plus petit langage k-réversible contenant S_+. Le principe général de cet algorithme est le même que celui de ZR, c'est-à-dire trouver la partition π la plus fine de l'automate $PT(S_+)$ telle que $PT(S_+)/\pi$ soit k-réversible. L'algorithme est un peu plus complexe à décrire puisque la fusion de deux blocs repose sur la condition de non déterminisme de l'automate ou de non déterminisme à l'horizon k de l'automate miroir.

Une version incrémentale de k-RI peut être utilisée afin d'identifier à la limite tout langage k-réversible. Le cas de sur-généralisation est encore évité grâce à la définition d'échantillon A_k-caractéristique d'un langage k-réversible L.

Notation :
Soit un automate $A = (Q, I, F, \Sigma, \delta)$ et soit $q \in Q$. On note L_q^k l'ensemble des k-meneurs de l'état q. Soit $x \in \Sigma^*$. On note $u(q, x)$ le plus petit mot $u \in \Sigma^*$ tel que $\delta(q_0, ux) = q$.

Définition 2.2.8 *Soit L un langage k-réversible et $A(L) = (Q, \{q_0\}, F, \Sigma, \delta)$ son automate canonique. L'*échantillon A_k-caractéristique *du langage L, noté S_L^k, est l'ensemble de mots défini par $S_L^k = L^{\leq k} \cup \{u(q,x)xv(q) \mid q \in Q\, x \in L_q^k\} \cup \{u(q,x)xav(q') \mid q, q' \in Q, a \in \Sigma, x \in L_q^k, q' = \delta(q,a)\}$.*

Propriété 2.2.2 (Angluin, [Ang82a]) *Soit L un langage k-réversible et soit S_L^k son échantillon A_k-caractéristique. Le langage L est le plus petit langage k-réversible contenant S_L^0.*

Remarque 2.2.4 Soit L un langage k-réversible. Par définition des langages réversibles, L est aussi $(k+1)$-réversibles. Par construction, il est évident que l'échantillon A_k-caractéristique de L n'est pas (toujours) égal à l'ensemble A_{k+1}-caractéristique de L. Prenons par exemple le langage $L_{ex5} = \{u \mid |u|_0 = 2 * l, |u|_1 = 2 * n, \text{ avec } l, n \in I\!N\}$ (voir figure 2.11, page précédente). Ce langage est 0-réversible et par conséquent il est également 1-réversible. L'échantillon A_0-caractéristique de L_{ex5} est $S^0_{L_{ex5}} = \{\varepsilon, 00, 11, 0101, 0110, 1010\}$ alors que l'échantillon A_1-caractéristique de L_{ex5} est $S^1_{L_{ex5}} = \{\varepsilon, 00, 11, 0000, 0011, 0101, 1001, 1010, 010001, 101101\}$. Cette remarque est importante puisque cela implique une certaine indépendance entre l'échantillon caractéristique et l'algorithme. Ainsi, si l'on utilise l'algorithme $k+1$-RI avec l'échantillon A_k-caractéristique d'un langage k-réversible, le langage généré a de grandes chances de ne pas être le langage cible.

Quelque soit $k \geq 0$, Angluin a donc montré que la classe $\mathcal{R}ev_k$ est identifiable à la limite par exemples positifs seuls. C'est un beau résultat dans la mesure où ces classes de langages contiennent un grand nombre de langages réguliers « connus » (voir section 2.1.5, page 63). Notons que c'est la « connaissance » de la propriété de k-réversibilité du langage cible qui permet à l'algorithme d'apprendre. Il semble donc que l'utilisation d'informations sur des caractéristiques algébriques relatives aux automates cibles augmente le pouvoir d'inférence des algorithmes.

Classe des langages réversibles

La question qui se pose à ce stade est de savoir si la classe $\mathcal{R}ev$ des langages réversibles est identifiable à la limite par exemples positifs. Malheureusement la réponse à cette question est non. En effet, quel que soit L un langage fini, il existe $k \geq 0$ tel que L est k-réversible (voir proposition 2.1.1, page 63). La classe $\mathcal{R}ev$ est par conséquent une classe super-finie de langages, ce qui lui interdit d'être identifiable à la limite par exemples positifs. Savoir que le langage cible est réversible n'est donc pas une information suffisante pour apprendre par exemples positifs seuls. Cependant, cette classe en tant que sous-classe de $\mathcal{R}eg$ est évidemment identifiable à la limite par présentation complète d'exemples. Angluin propose un algorithme de consistance RI dont la version incrémentale permet d'identifier à la limite la classe $\mathcal{R}ev$. L'algorithme RI prend en entrée S, un échantillon d'exemples positifs et négatifs, et trouve la plus petite valeur de k telle qu'il existe un langage k-réversible consistant avec S. Cet algorithme retourne alors le plus petit langage k-réversible convenable. RI tourne en temps $O(\|S\|^3(k+1)^2)$ $(k \leq \|S\|)$.

Nous venons de présenter des travaux montrant que des sous-classes de la classe $\mathcal{R}eg$ sont identifiables à la limite par exemples positifs seuls, bien que $\mathcal{R}eg$ ne le soit pas. Kobayashi et Yokomori [KY97] proposent d'apprendre approximativement la classe $\mathcal{R}eg$ par exemples positifs au moyen d'une classe d'hypothèses identifiable à la limite par exemples positifs. Ils montrent que la classe $\mathcal{R}ev_k$ permet effectivement une telle approximation. Remarquons que ce type de travaux élargit le modèle de Gold en définissant la notion d'apprentissage approximatif à la limite ; on pourra trouver dans [Muk94] la définition d'un tel modèle.

2.3 PAC-Apprentissage des Langages Réguliers

Nous abordons dans cette section, l'étude de l'apprenabilité de la classe $\mathcal{R}eg$ dans le modèle de Valiant (voir section 1.3, page 32). Dans le contexte du modèle de Gold, il s'agissait de savoir si cette classe de langages peut être identifiée à la limite, c'est-à-dire s'il existe un algorithme qui génère exactement le langage cible sous l'hypothèse que tout exemple est présenté en un nombre fini d'étapes. Quoique la question de temps de calcul ne se pose pas dans ce contexte (le processus d'identification à la limite étant infini), les algorithmes de consistance sur lesquels l'apprentissage s'appuie doivent retourner une réponse en un temps raisonnable (*ie* polynomial) afin de ne pas pénaliser le temps de réponse à chaque étape. La question de la complexité de l'identification à la limite des DFAs a été étudiée par Pitt [Pit89]. Cependant, les tentatives de définitions qu'il propose semble par trop restrictives. Dans [dlH97], de la Higuera étudie également cette question dans le cadre de l'identification à partir de données fixées.

Dans le modèle de Valiant, un problème équivalent se pose quant à la définition de la poly-apprenabilité dans le cas des langages. En effet, dans le cas des langages, les exemples relatifs à un concept donné peuvent être de longueur différentes (en nombre éventuellement infini), ce qui n'est pas le cas, par exemple, pour les formules booléennes. Cette particularité pose un réel problème pour définir la polynomialité des algorithmes d'apprentissage. La définition que nous avons présentée (voir définition 1.3.4, page 34) inclut dans la mesure de complexité la longueur du plus long exemple lu par l'algorithme d'apprentissage. Il est important de remarquer que ceci pose un problème pour certaines distributions de probabilité pour lesquelles des exemples arbitrairement longs peuvent être tirés (la distribution de Solomonoff-Levin en est un exemple). Pitt [Pit89] propose une autre définition de poly-PAC apprenabilité dans laquelle la classe des DFAs est dite poly-PAC apprenable si toutes les restrictions du langage cible pour une borne donnée sur la longueur des chaînes sont apprenables. Cette définition ne paraît pas non plus complètement satisfaisante. Il semble plus naturel de ne pas prendre en compte la longueur des exemples dans le calcul de la complexité et donc de ne considérer que les paramètres de confiance et de précision ainsi que la taille de la cible.

Quoiqu'il en soit, nous continuons dans cette section à utiliser la définition donnée au chapitre précédent. Notre problème consiste donc à étudier si la classe $\mathcal{R}eg$ associée à l'ensemble de représentations $\mathcal{DFA}$ est poly-PAC-apprenable. Une première piste consiste à étudier s'il existe un algorithme d'Occam polynomial pour $(\mathcal{R}eg,\mathcal{DFA})$; si un tel algorithme existe, le théorème 1.3.3 (page 38) prouverait que la classe $\mathcal{R}eg$ est poly-PAC-apprenable. Malheureusement, un tel algorithme n'est pas envisageable étant donné le théorème 2.2.2 (page 66) (on pourra se reporter au chapitre 6 de [Nat91] pour plus de détails à ce sujet). Cette inexistence d'algorithme d'Occam pour la classe $\mathcal{R}eg$ n'est pas suffisante pour affirmer que $\mathcal{R}eg$ n'est pas poly-PAC-apprenable. Cependant, les résultats de Pitt et Warmuth [PW90, PW93] montrent que sous certaines hypothèses cryptographiques, la classe $\mathcal{R}eg$ associée à l'ensemble de représentations $\mathcal{DFA}$ n'est pas poly-PAC-apprenable.

Dès lors, il semble légitime de se demander si le choix de la représentation par les DFAs est judicieux et si un autre ensemble de représentations, plus « puissant », ne permettrait pas de rendre ces problèmes praticables (*ie* polynomiaux). Kearns et Valiant [KV94] montrent cependant que sous certaines hypothèses cryptographiques, un tel changement de représentations ne modifie en rien la non poly-PAC-apprenabilité de $\mathcal{R}eg$. Rappelons que ces résultats sont essentiellement dus aux « pires des cas », lorsque langage cible et échantillon d'apprentissage sont choisis de manière à rendre impraticable l'apprentissage [Ang78]. Lang montre

en effet empiriquement, qu'en moyenne, c'est à dire lorsque la cible et l'échantillon sont générés aléatoirement, l'apprentissage approximatif des DFAs est possible en temps raisonnable [Lan92].

Il semble donc indispensable d'assouplir le modèle en fournissant des informations supplémentaires à l'apprenant. Pour ce faire, une solution consiste à autoriser l'apprenant à poser certains types de questions[10] à l'enseignant. Cependant, les différentes requêtes imaginables n'ont pas toutes la même puissance et ne permettent pas toutes d'atteindre la poly-PAC-apprenabilité. Par exemple, la requête `APPARTIENT` qui consiste à demander si un exemple est ou non positif n'est pas suffisante lorsqu'elle est utilisée seule. En effet, Angluin montre que ce type de requête permet l'apprentissage approximatif des DFAs si en plus un échantillon d'entrée bien adapté à la cible est passé à l'apprenant [Ang82b]. Angluin [Ang87] propose d'autoriser les deux types de questions suivants :

- « Est-ce que l'exemple e est un exemple positif pour la cible? ». L'enseignant répond simplement par oui ou par non.
- « Est-ce que l'hypothèse h est équivalente au concept cible? ». L'enseignant répond soit par l'affirmative soit par la négative et fournit alors un contre-exemple, c'est-à-dire un exemple positif pour h mais non reconnu par la cible ou l'inverse.

Formellement, on parle d'*oracle d'appartenance* pour traiter le premier type de questions et d'*oracle d'équivalence* pour le second. Ces oracles seront par la suite notés `APPARTIENT`(e) et `EQUIVALENT`(h). L'association de ces deux types d'oracles forment ce qu'Angluin appelle un *enseignant minimal adéquat*[11]. Intuitivement, on peut donc imaginer qu'un enseignant doit au moins répondre à ces deux types de questions s'il espère un apprentissage de la part de ses élèves. Précisons qu'au niveau de la complexité en temps, l'appel à l'un de ces oracles coûte une unité de temps.

L'algorithme L* (*Learner*) [Ang87] prend en entrée un échantillon fini S relatif à un langage cible $f \in \mathcal{R}eg$ et retourne **exactement** le concept cible f. Au cours de l'apprentissage, cet algorithme peut s'adresser à chacun des deux oracles décrits précédemment. Sa complexité en temps est polynomiale par rapport à la taille de l'automate cible et à la longueur du plus long contre-exemple fourni par l'oracle d'appartenance. Dans la suite, nous allons étudier cet algorithme et voir dans quelle mesure il peut être utilisé dans le modèle d'apprentissage PAC.

2.3.1 Présentation de l'algorithme L*

L'idée principale sur laquelle repose l'algorithme L* est la propriété d'*invariance-droite* des automates déterministes :

Théorème 2.3.1 *(Invariance-droite) Soit un automate déterministe* $A = (Q, \{q_0\}, F, \Sigma, \delta)$. *Soient deux chaînes* $u, v \in \Sigma^*$. *S'il existe une chaîne* $w \in \Sigma^*$ *telle que* $\delta(q_0, uw) \in F$ *et* $\delta(q_0, vw) \notin F$ *alors* $\delta(q_0, u) \neq \delta(q_0, v)$.

Définition 2.3.1 *Soit un ensemble de mots* $E \subset \Sigma^*$. *L'ensemble* E *est dit* préfixe clos *si quel que soit* $u \in E$, *chaque préfixe de* u *appartient à* E. *L'ensemble* E *est dit* suffixe clos *si quel que soit* $u \in E$, *chaque suffixe de* u *appartient à* E

10. Dana Ron [Ron95] emploie le terme « d'apprenant actif » pour signifie cette prise d'initiatives rendue possible à l'apprenant.

11. En anglais *minimally adequate teacher.*

Intuitivement, l'algorithme L* va construire progressivement un DFA en ajoutant petit à petit des états de manière à assurer cette propriété d'invariance-droite. Pour ce faire, l'algorithme maintient une *table d'observation* décrivant l'automate en cours de construction. Chaque ligne de la table correspond à un état de l'automate, chaque colonne correspond à un mot lu. Les labels des lignes (c'est-à-dire des noms d'état) décrivent des chemins permettant d'atteindre cet état depuis l'état initial ; l'ensemble de ces mots, noté S, doit être préfixe-clos afin d'assurer que tous les états de l'automate sont effectivement décrits. Les labels des colonnes sont des mots de Σ^* qui permettent de distinguer deux états différents de l'automate; l'ensemble de ces mots, noté E, est suffixe-clos afin d'assurer la description de chaque transition possible. Étant données une ligne s et une colonne e, la cellule $Table[s, e]$ contient la valeur booléenne 1 si et seulement si $\delta(s, e) \in F$. L'état initial est l'état dont le label est ε. Un état s est terminal si la cellule $Table[s, \varepsilon]$ est égale à 1. Afin de décrire complètement l'automate, il est utile d'ajouter un certain nombre de lignes en fin de table. Celles-ci sont labellées par les mots de l'ensemble $S.\Sigma \setminus S$ et permettent de décrire les transitions[12] de l'automate pour tous les états et toutes les lettres.

Par la suite, pour $s \in \Sigma^*$, on notera $ligne(s)$ la suite ordonnée des cellules apparaissant sur la ligne de label s.

Définition 2.3.2 *Une table d'observation est dite* close *si* $\forall\ s \in (S.\Sigma) \setminus S, \exists\ s' \in S$ *tels que* $ligne(s) = ligne(s')$.

Intuitivement, une table close assure que tout état d'arrivé d'une transition existe effectivement dans l'ensemble des états.

Définition 2.3.3 *Une table d'observation est dite* consistante *si* $\forall\ s, s' \in S$ *tel que* $ligne(s) = ligne(s')$, $\nexists a \in \Sigma$ *tel que* $ligne(sa) \neq ligne(s'a)$.

La consistance de la table est donc directement liée à la propriété d'invariance-droite. En effet, si deux lignes sont égales, cela signifie qu'elles décrivent le même état et dans ce cas, il est impératif que les états d'arrivée par une transition labellée par une même lettre soient les mêmes.

Exemple :
La table d'observation suivante décrit l'automate reconnaissant le langage $L_{ex5} = \{u \mid |u|_0 = 2 * l, |u|_1 = 2 * n, l, n \in I\!N\}$ (voir figure 2.11, page 75).

$S = \{\varepsilon, 0, 1, 11, 01, 011\}$,
$S.\Sigma = \{00, 10, 110, 111, 010, 0110, 0111\}$ et
$E = \{\varepsilon, 0, 1\}$.

12. Du moins celles qui ne le sont pas encore.

	ε	0	1	
ε	1	0	0	$= q_0$
0	0	1	0	$= q_1$
1	0	0	1	$= q_3$
11	1	0	0	$= q_0$
01	0	0	0	$= q_2$
011	0	1	0	$= q_1$
00	1	0	0	$\delta(q_1, 0) = q_0$
10	0	0	0	$\delta(q_3, 0) = q_2$
110	0	1	0	$\delta(q_0, 0) = q_1$
111	0	0	1	$\delta(q_0, 1) = q_3$
010	0	0	1	$\delta(q_2, 0) = q_3$
0110	1	0	0	$\delta(q_1, 0) = q_0$
0111	0	0	0	$\delta(q_1, 1) = q_2$

◇

La figure 2.12 (page suivante) présente l'algorithme L* tel qu'il est proposé dans [Ang87]. Celui-ci initialise la table avec un état initial dont le label est ε ; les transitions à partir de cet état et pour chaque lettre de l'alphabet sont également calculées. Ensuite, l'algorithme rend la table d'observation consistante et close de sorte qu'elle corresponde effectivement à un automate déterministe minimal. Dans ce cas, l'automate décrit par la table est proposé à l'oracle d'équivalence. Si celui-ci répond que l'hypothèse est valable, l'algorithme s'arrête. Dans le cas contraire, c'est qu'il existe un mot mal classé par l'automate. Ce contre-exemple est ajouté à S de sorte qu'il décrive un chemin vers un nouvel état de l'automate et le processus recommence.

La preuve de correction de cet algorithme peut être trouvée dans [Ang87] ainsi que dans le chapitre 6 de [Nat91].

2.3.2 Utilisation de l'algorithme L* dans le modèle PAC

Tel qu'il a été présenté, l'algorithme L* permet d'inférer l'automate canonique d'un langage cible au moyen de deux types d'oracles. Son utilisation reste pour le moment indépendante de tous les contextes d'apprentissage précédemment décrits.

Cependant, Angluin montre que dans un cadre probabiliste, un tirage aléatoire d'exemples peut remplacer l'oracle d'équivalence [Ang87, Ang88]. Pour ce faire, il suffit de tirer un certain nombre d'exemples (au moyen de l'oracle d'exemples). Si l'hypothèse est consistante avec l'ensemble des exemples fournis, cela correspond à une réponse positive de l'oracle d'équivalence. Dans le cas contraire, c'est-à-dire s'il existe au moins un exemple mal classé par l'automate hypothèse, celui-ci joue alors le rôle de contre-exemple qui aurait été retourné par l'oracle d'équivalence. Notons que le nombre d'exemples à tirer dépend des paramètres de précision (ε) et de confiance (δ) demandés.

Angluin montre que cette nouvelle version de l'algorithme, nommée L_a^*,(*Approximate Learner*), est effectivement un algorithme de poly-PAC-apprentissage avec oracle. La classe $\mathcal{R}eg$ est donc polynomialement PAC-apprenable si l'on autorise l'algorithme d'apprentissage à avoir recours à un oracle d'appartenance. Afin de pallier quelque peu le problème précédemment évoqué de la dépendance entre la complexité de l'algorithme et la longueur du plus long exemple lu, Angluin propose de modifier quelque peu l'algorithme L_a^*. Cette nouvelle version

Algorithme L*
Entrée:
Début
* Initialisation
Soit $S \leftarrow \{\varepsilon\}$
Soit $E \leftarrow \{\varepsilon\}$
$Table[\varepsilon, \varepsilon] =$ APPARTIENT(ε)
Pour $a \in \Sigma$
$Table[a, \varepsilon] =$ APPARTIENT(a)
Fpour

* Boucle principale
Répéter
Tantque *Table* n'est pas close ou n'est pas consistante
Si *Table* n'est pas consistante
Alors
Trouver deux chaînes s_1 et s_2 de S, $a \in \Sigma$ et $e \in E$
tq $ligne(s_1) = ligne(s_2)$ et $Table[s_1 a, e] \neq Table[s_2 a, e]$
$E \leftarrow E \cup \{ae\}$
Pour $s \in S \cup S.\Sigma \setminus S$
$Table[s, ae] =$ APPARTIENT(sae)
Fpour
Fsi
Si *Table* n'est pas close
Alors
Trouver $s_1 \in S$ et $a \in \Sigma$ tq $\nexists\ s \in S$ tq $ligne(s_1 a) = ligne(s)$
$S \leftarrow \{s_1 a\}$
Pour $b \in \Sigma$
Pour $e \in E$
$Table[s_1 ab, e] =$ APPARTIENT$(s_1 abe)$
Fpour
Fpour
Fsi
Ftq
Calculer l'automate A correspondant à *Table*
Si pas EQUIVALENT(A)
Alors
Soit t le contre-exemple
$S \leftarrow S \cup Pr(\{t\})$
Pour $s \in Pr(\{t\})$
Pour $e \in E$
$Table[s, e] =$ APPARTIENT(se)
Pour $a \in \Sigma$
$Table[sa, e] =$ APPARTIENT(sae)
Fpour
Fpour
Fpour
Fsi
Tantque pas $EQUIVALENT(A)$

* Résultat
Retourner A
Fin

FIG. 2.12 - L'algorithme L*.

tire un échantillon préliminaire afin de déterminer une longueur l telle qu'avec la probabilité au moins $(1 - \delta/2)$ le poids des mots de longueur supérieure à l est est au plus $\varepsilon/2$. L'algorithme $\mathtt{L}_a^*$ se déroule alors comme précédemment décrit à la différence que seuls les exemples de longueur inférieure à l sont traités ; les valeurs des paramètres de confiance et de précision sont ici $\delta/2$ et $\varepsilon/2$. Il est alors possible de montrer qu'un échantillon de taille polynomiale en $1/\delta$ et $1/\varepsilon$ permet à l'algorithme de retourner avec une probabilité au moins $(1 - \delta)$ une hypothèse h qui $\varepsilon/2$-approxime une $\varepsilon/2$-approximation de la cible.

2.4 Conclusion

Dans ce chapitre, nous avons étudié l'apprentissage automatique pour le cas particulier des langages réguliers, connus sous le nom d'inférence grammaticale.

Pour le modèle de Gold, il existe plusieurs algorithmes d'apprentissage à la limite dans le protocole dit de données fixées. Ceux-ci sont basés sur l'existence d'échantillons caractéristiques pour le langage cible. Nous avons également abordé l'inférence à la limite par exemples positifs seuls. Dans ce cas, seules des sous-classes de langages réguliers sont apprenables, également grâce à l'existence d'échantillons caractéristiques.

Dans le modèle de Valiant classique, la classe des langages réguliers n'est pas apprenable et ce, quel que soit l'ensemble de représentations utilisé. Ce résultat négatif est cependant contourné en assouplissant le modèle, c'est-à-dire en aidant l'algorithme au moyen d'un oracle d'appartenance. Cette modification du paradigme de Valiant apporte donc un réel pouvoir supplémentaire d'apprentissage.

Enfin, nous n'avons pas étudié le cas du modèle de Li et Vitányi pour l'apprentissage des langages réguliers. Notons simplement que dans ce modèle, la classe des automates simples 0-réversibles est apprenable [LV91]. Un automate 0-réversible à n états est simple si tout état appartient à un chemin menant de l'état initial à un l'état final de complexité de Kolmogorov en $O(\log n)$. Bien que cette classe de langages semble quelque peu artificielle dans sa définition, il s'agit d'un premier résultat d'apprenabilité selon un modèle dérivé de PAC pour une classe de langages réguliers ; précisons que nous n'avons connaissance d'aucun résultat sur la PAC apprenabilité de la classe des langages 0-réversibles.

Chapitre 3

Un Nouveau Modèle d'Apprentissage : le Modèle PACS

Dans le chapitre 1 (page 25), nous avons présenté trois modèles d'apprentissage. Chacun de ces modèles formalise un certain nombre d'idées intuitives que l'on peut se faire de l'apprentissage naturel : échec possible, approximation de la cible, présentation d'exemples les plus simples possibles ...

Jusqu'à présent, le modèle de Li et Vitányi semble être un bon candidat pour modéliser l'apprentissage naturel en répondant à l'ensemble de ces attentes. Cependant, la notion de simplicité « absolue » d'exemples que ce modèle intègre n'est pas satisfaisante. Ainsi, à aucun moment, le concept cible n'est pris en compte afin de « bien » choisir les exemples. Nous proposons dans ce chapitre un nouveau modèle d'apprentissage, appelé *modèle PACS*[1]. Dans celui-ci, la mesure de simplicité d'exemples est faite par rapport au concept cible. Dans une première section, nous présenterons formellement ce nouveau modèle d'apprentissage ; un théorème du type rasoir d'Occam sera également prouvé. Dans les trois sections suivantes, nous nous attacherons à montrer l'apprenabilité de trois classes de concepts connues. Enfin, dans la dernière section, nous présenterons un ensemble de résultats généraux concernant ce nouveau modèle ; cette section nous donnera également l'occasion d'émettre quelques critiques relatives à ce modèle.

3.1 Présentation du Modèle PACS

3.1.1 Idées intuitives

Reprenons l'exemple de l'enseignant désireux d'apprendre la notion de « carré » à ses élèves (voir section 1.1.4, page 28). Nous allons comparer les méthodes pédagogiques de deux professeurs : Monsieur **Absolu** et Monsieur **Relatif**.

La figure 3.1 (page suivante) présente le type d'exemples que monsieur Absolu propose à ses élèves. Les exemples positifs (*i.e.* les carrés) apparaissent en noir, les contre-exemples (*i.e.* les non carrés) sont dessinés en gris.

La figure 3.2 (page suivante) concerne les exemples proposés par Monsieur Relatif.

Monsieur Absolu n'a dessiné que des figures géométriques dont les coordonnées des sommets sont des valeurs entières. Il était en effet plus simple de dessiner ainsi l'ensemble des

1. La plupart des résultats de ce chapitre ont été publiés dans [DdG96].

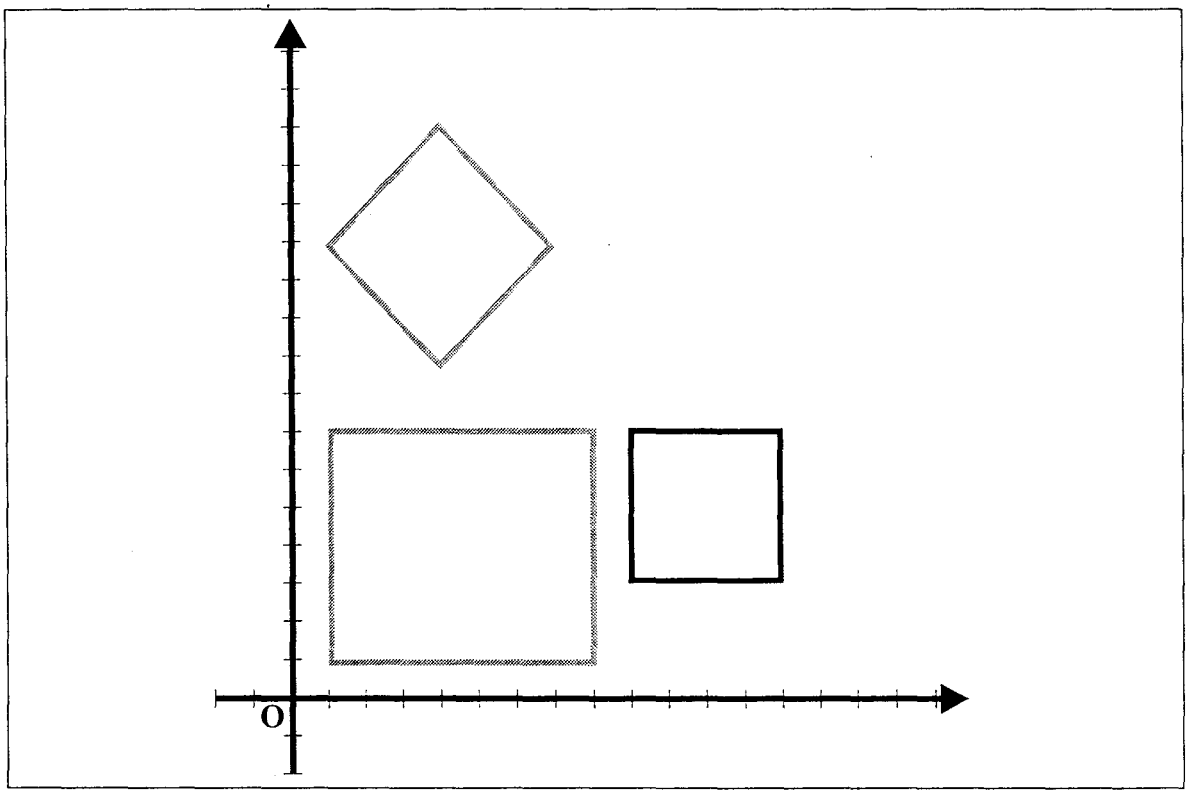


FIG. 3.1 - Quelques exemples de Monsieur Absolu.

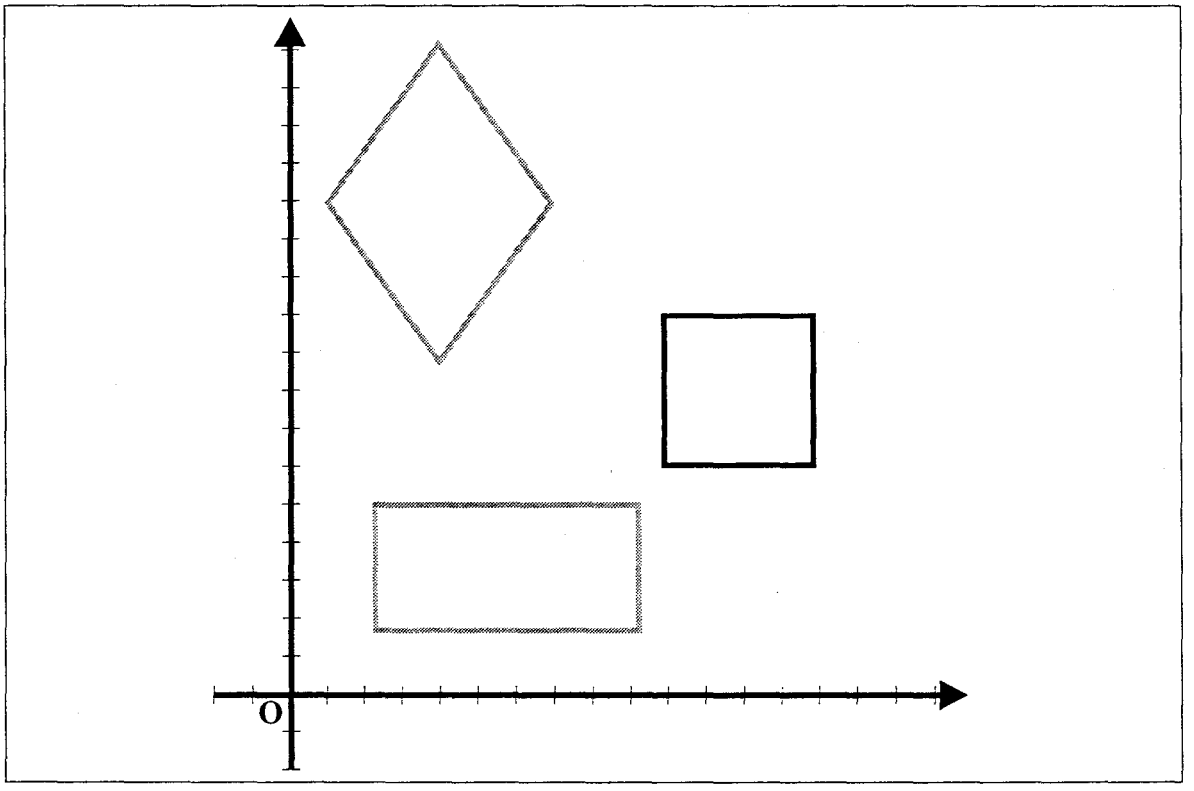


FIG. 3.2 - Quelques exemples de Monsieur Relatif.

exemples. Cependant, il paraît difficile de distinguer entre carrés et non carrés car Monsieur Absolu n'a pas pris la peine de choisir des contre-exemples suffisamment éloignés des carrés. Ainsi trouve-t-on, par exemple, un rectangle dont la longueur n'est que très faiblement supérieure à la largeur.

Pour Monsieur Relatif par contre, il était important de bien choisir les exemple par rapport à la notion de carré. Ainsi, le rectangle qu'il propose ne peut en aucun cas être confondu avec un carré. Monsieur Relatif sait que l'apprentissage sera plus facile s'il présente, pour chaque type de figures, des représentants caractéristiques de ceux-ci. Remarquons que cette idée de représentants caractéristiques est relative au but à atteindre ; en effet, parions que monsieur Relatif aurait choisi des exemples complètement différents si l'objet du cours avait été « les rectangles dont la longueur est le double de la largeur ».

Nos deux professeurs ont le même but mais leurs méthodes d'enseignement sont différentes. Monsieur Absolu choisit ses exemples en fonction de leur complexité intrinsèque (ici la facilité qu'il aura à dessiner des polygones dans un repère orthonormé), mais cette mesure de complexité se fait indépendamment du but à atteindre. Aussi, les exemples qu'il présente ne permettront probablement pas à ses élèves de comprendre ce qu'est effectivement un carré. Monsieur Relatif, quant à lui, préfère donner des exemples suffisamment représentatifs en gardant à l'esprit qu'il faut caractériser les propriétés des carrés. Monsieur Relatif arrivera probablement à ses fins.

Le modèle Simple-PAC, proposé par Li et Vitányi, peut être assimilé à la méthode pédagogique de monsieur Absolu. Les exemples sont effectivement simples, mais cette notion de simplicité absolue n'est pas toujours adaptée au concept à apprendre. Le modèle que nous allons présenter dans la suite se rapproche quant à lui des méthodes de Monsieur Relatif. Ce modèle formalise la propriété consistant à favoriser la présentation d'exemples simples par rapport à la cible. Ce type d'exemples étant susceptible de faciliter la recherche des caractéristiques propres à celle-ci. Notons que cette idée d'un enseignant guidant ses élèves au moyen d'un bon jeu d'exemples se trouve déjà dans quelques autres modèles d'enseignement [GK91, JT92, GM93].

Ce nouveau modèle dérive du modèle PAC de la même manière que le modèle de Li et Vitányi. En fait, dans son énoncé, ce modèle est très proche de celui de Li et Vitányi à la différence que la mesure de simplicité se fera relativement au concept cible. Par la suite, nous utiliserons le terme *PACS* (PAC avec exemples Simples) pour désigner ce modèle.

Avant de présenter formellement ce nouveau modèle, il convient de définir la notion de simplicité d'exemple par rapport à un concept. Il sera alors possible de définir une mesure de probabilité privilégiant ce type d'exemples.

3.1.2 Distribution universelle de Solomonoff-Levin relative à un concept

La mesure de Solomonoff-Levin utilisée dans le modèle Simple-PAC s'appuie sur la notion de complexité de Kolmogorov d'une chaîne (voir section 1.4.1, page 41). Cette mesure est telle que les chaînes peu complexes, c'est-à-dire fortement compressibles ou encore pouvant être générées par un programme court, ont une forte probabilité d'être tirées.

En ce qui nous concerne, nous voulons favoriser le tirage des chaînes (*i.e.* des exemples) pouvant être facilement décrites en utilisant une connaissance préalable (*i.e.* la description du concept).

La complexité de Kolmogorov d'un objet x relative à un objet y, notée $K(x \mid y)$, formalise cette notion de simplicité relative (voir section 1.4.1, page 41). En effet, supposons que y soit

la description d'un concept, si x peut être facilement décrit en utilisant éventuellement la connaissance de y, alors x est un exemple simple relativement à y. De plus, si l'information apportée par y est effectivement utile pour décrire x, c'est-à-dire si x et y partagent de l'information, alors on peut penser que la chaîne x est susceptible de caractériser certaines propriétés de y.

Remarque 3.1.1 Il convient cependant de ne pas considérer toute chaîne x ayant une faible complexité $K(x \mid y)$ comme étant caractéristique de y. En effet, le théorème 1.4.4 (page 43) établit que $K(x \mid y) \leq K(x) + O(1)$. Aussi, une chaîne x intrinsèquement simple, c'est-à-dire pouvant être générée sans aucune connaissance *a priori* a une complexité $K(x)$ faible, ce qui implique qu'elle a également une faible complexité par rapport à y ; intuitivement, rien n'oblige un programme à utiliser de l'information supplémentaire y pour générer x. Pour autant, la chaîne x peut n'avoir aucun rapport avec y, et n'être donc absolument pas caractéristique de cette dernière.

Avant de présenter la mesure de probabilité de Solomonoff-Levin relative à un concept, il convient de montrer le théorème suivant :

Théorème 3.1.1 *Soit une machine de Turing préfixe universelle U. Si r est une chaîne de Σ^* alors*

$$\sum_{x \in \Sigma^*} 2^{-K_U(x|r)} \leq 1$$

Preuve :
La preuve de cette proposition s'appuie sur une technique connue consistant à faire correspondre à toute chaîne u le segment demi-ouvert $[0.u, 0.u + 2^{-|u|}[$ (écrit en binaire) sur l'axe $[0, 1[$. Par exemple, à la chaîne $u = 01$, on associe le segment $[0.01, 0.01 + \frac{1}{2^2}[= [\frac{1}{2}, \frac{1}{2} + \frac{1}{4}[$. Si u_1 et u_2 sont deux chaînes non préfixes l'une de l'autre, les segments qui leur sont respectivement associés sont disjoints. En effet, comme $\sum_{i=1}^{\infty} \frac{1}{2^{n+i}} = \frac{1}{2^n}$, on a $[0.u, 0.u + 2^{-|u|}[= [0.u, 0.u111\ldots[$. Par conséquent, si une chaîne v est telle que $0.v \in [0.u, 0.u + 2^{-|u|}[$, alors u est un préfixe de v. Soit $Prog(U)$ l'ensemble des programmes de la machine de Turing U considérée. On peut associer à chaque programme p de $Prog(U)$ un segment $[0.p, 0.p + 2^{-|p|}[$. L'ensemble $Prog(U)$ étant préfixe, les segments associés à chaque programme p sont deux à deux disjoints et donc $\sum_{p \in Prog(U)} 2^{-|p|} \leq 1$. Parmi l'ensemble de ces programmes, considérons l'ensemble $Prog_{min}(U)$ des programmes de longueur minimale calculant un x connaissant r c'est-à-dire $Prog_{min}(U) = \{p \in Prog(U) \mid \exists\, x, r \in \Sigma^* \text{ tq } U(p, r) = x \text{ et } \forall\, p' \in Prog(U) \text{ tq } U(p', r) = x \text{ alors } |p| \leq |p'|\}$. Cet ensemble est évidemment inclus dans $Prog(U)$, et si $p \in Prog_{min}(U)$ avec $U(p, r) = x$ alors $|p| = K(x \mid r)$. L'ensemble $Prog_{min}(U)$ est par conséquent également préfixe d'où le théorème. □

Définition 3.1.1 *La* mesure de probabilité de Solomonoff-Levin associée à U relativement à r, *notée* $\mathbf{m}_r(x)$, *est telle que :* $\mathbf{m}_r(x) = \lambda_r 2^{-K_U(x|r)}$, *où* λ_r *satisfait*

$$\lambda_r \sum_{x \in \Sigma^*} 2^{-K_U(x|r)} = 1$$

Il est possible de montrer que $\sum_{x\in\Sigma^*} 2^{-K_U(x|r)} < 1$ et par conséquent $\lambda_r > 1$ (voir chap. 4 de [LV93]).

Cette mesure de probabilité privilégie les chaînes fortement compressibles par la machine U lorsque celle-ci dispose de l'information r.

De plus, étant donnée une chaîne $r \in \Sigma^*$, pour toute paire de machines de Turing U et U' satisfaisant le théorème d'invariance (voir théorème 1.4.1, page 42), il existe une constante $k_{U,U'}$ telle que pour toute chaîne $x \in \Sigma^*$

$$2^{-k_{U,U'}} \leq \mathbf{m}_r(x)/\mathbf{m}'_r(x) \leq 2^{k_{U,U'}},$$

avec $\mathbf{m}_r$ (respectivement $\mathbf{m}'_r$) la distribution de Solomonoff-Levin associée à U (resp. à U') relativement à r.

Pour la suite, on suppose fixée une machine de Turing préfixe universelle U et on note $K(x \mid r) = K_U(x \mid r)$ et $\mathbf{m}_r(x) = \lambda_r 2^{-K(x|r)}$.

3.1.3 Le modèle PACS

Nous pouvons maintenant définir formellement le modèle d'apprentissage PACS. Comme nous l'avons déjà indiqué, la définition de ce modèle d'apprentissage est similaire à celle du modèle de Li et Vitányi, puisqu'il suffit de remplacer la mesure de probabilité $\mathbf{m}$ par la mesure $\mathbf{m}_r$.

On rappelle que $l_{min}(f)$ représente la longueur d'une plus petite représentation du concept f.

Définition 3.1.2 *Soit un algorithme* A *prenant en entrée un paramètre de précision* $\varepsilon \in]0,1]$, *un paramètre de confiance* $\delta \in]0,1]$ *et un entier* l. A *est un* algorithme d'apprentissage PAC avec exemples simples *(algorithme de PACS-apprentissage) pour la classe de concepts* F *associée à l'ensemble de représentations* R *si pour tout concept* f *de* F *et pour tout nom* r *de* f *avec* $|r| \leq l$, A *a recours à un oracle d'exemples* $Exemples_{X,\mathbf{m}_r}(f)$ *et sort la représentation d'un concept* h *de* F, *telle que avec la probabilité au moins* $1-\delta$, $Erreur_{\mathbf{m}_r,f}(h) \leq \varepsilon$.

Définition 3.1.3 *Une classe de concepts* F *est* PAC apprenable avec exemples simples *(PACS-apprenable) dans l'ensemble de représentations* R *s'il existe un algorithme d'apprentissage PACS pour* F *dans* R.

Une condition de polynomialité peut être exigée pour l'algorithme d'apprentissage afin d'assurer que le processus est praticable.

Définition 3.1.4 *Une classe de concepts* F *est* polynomialement-PACS-apprenable *(poly-PACS-apprenable) dans* R *si elle est PACS-apprenable dans* R *et si* A *l'algorithme d'apprentissage tourne en temps polynomial par rapport à* $1/\varepsilon$, $1/\delta$ *et* l.

Comme dans le cas du modèle de Li et Vitányi, la complexité en temps ne dépend pas de la longueur du plus long exemple tiré. Dans le cas d'exemples de longueur non uniforme (*e.g.* les mots), ici encore, un exemple simple par rapport à la cible peut être arbitrairement long. Par conséquent, l'algorithme d'apprentissage doit se contenter de ne lire que les exemples (ou des préfixes) de taille polynomiale en $1/\varepsilon, 1/\delta$ et l.

Il est important de remarquer que dans ce modèle, la mesure de probabilité utilisée pour tirer les exemples dépend effectivement du concept à apprendre, de la même manière que le

professeur adapte ses exemples en fonction du concept qu'il a l'intention d'enseigner. Comme il a été dit précédemment, cette distribution de probabilité favorise non seulement les exemples intrinsèquement simples mais aussi les exemples « caractéristiques » du concept. Dès lors, on peut imaginer que l'essentiel de la tâche de l'apprenant consistera à retrouver, parmi l'ensemble d'exemples, ceux qui sont effectivement représentatifs du concept et d'en dégager les caractéristiques communes.

3.1.4 Exemple de PACS-apprenabilité

Afin d'illustrer le principe de ce nouveau modèle, reprenons le cas de l'apprentissage de la classe des singletons (voir section 1.5, page 46). Rappelons que, dans cet exemple, le domaine est $I\!N$ et la classe de concepts $F = \{\{i\} \mid i \in I\!N\}$. De plus, cette classe F est naturellement associée à l'ensemble de représentations R qui n'est autre qu'un codage binaire des entiers. Cette classe de concepts est évidemment apprenable dans le modèle PACS (la PAC-apprenabilité de cette classe étant une condition suffisante de PACS-apprenabilité). L'algorithme d'apprentissage donné à la figure 3.3 est très semblable à celui utilisé dans le modèle PAC.

Algorithme `Apprend_Singletons_4`
Entrée : ε, δ, l
Début
  Tirer un échantillon S de taille suffisante en utilisant la distribution $\mathbf{m}_r$
  Si S contient un exemple positif $e = (i, 1)$
  Alors $h \leftarrow \{i\}$
  Sinon $h \leftarrow \{j\}$ tel que $(j, 0) \notin S$
  Fsi
  Sortir h
Fin

FIG. 3.3 - Algorithme de poly-PACS-apprentissage des singletons sur $I\!N$

Ici encore, les commentaires relatifs au PAC-apprentissage peuvent s'appliquer et il suffit que la taille de l'échantillon tiré soit supérieure à $\frac{2}{\varepsilon}\log(\frac{1}{\delta})$ pour assurer le PACS-apprentissage. Afin de comprendre plus précisément les différences qui existent entre les modèles PAC, simple-PAC et PACS, nous nous proposons d'étudier dans quels cas, l'apprentissage exact de la cible est réalisé. Une telle comparaison entre les deux premiers modèles a déjà été réalisée de manière intuitive à la section 1.5 (page 46), nous la formalisons ici.

Soit P la distribution de probabilité utilisée lors de l'apprentissage dans le modèle considéré. Soit $f = \{i\}$ le concept cible. La probabilité que l'exemple i n'apparaisse pas dans l'échantillon S est $(1 - P(i))^{|S|}$. On peut considérer que l'apprentissage exact a lieu seulement si i est fourni à l'algorithme (le cas où i ne fait pas partie de l'échantillon et que l'algorithme d'apprentissage choisit $j = i$ est supposé improbable). La probabilité que l'algorithme apprenne exactement la cible est donc égale à $1 - (1 - P(i))^{|S|}$. Étudions ce que cela signifie

dans chacun des trois modèles:

- **Apprentissage PAC**: la distribution de probabilité P utilisée est quelconque. L'apprentissage exact est atteint avec la probabilité $1-(1-P(i))^{|S|}$. Celle-ci dépend uniquement du poids attribué par P à i. Ainsi, plus la distribution de probabilité P favorise i, plus l'apprentissage a de chance d'être exact.

- **Apprentissage Simple-PAC**: la distribution utilisée dans ce modèle est $\mathbf{m}$. La probabilité que l'hypothèse retournée soit exactèment la cible est donc $1-(1-\mathbf{m}(i))^{|S|}$. Par conséquent, c'est la complexité intrinsèque de i qui conditionne l'apprentissage exact. Plus i est simple, plus la probabilité d'apprendre exactement le concept f est grande.

- **Apprentissage PACS**: la distribution de probabilité utilisée dans ce modèle est $\mathbf{m}_r$ avec r un nom pour f. La probabilité que l'apprentissage soit exact est donc égale à $1-(1-\mathbf{m}_r(i))^{|S|}$. Un nom r pour f n'est autre qu'un codage de i, l'exemple unique de f. Ceci implique évidemment que $K(i \mid r)$ est faible, c'est-à-dire que $\mathbf{m}_r(i)$ est grande et ce, quelle que soit la complexité intrinsèque de i. Ainsi, que i soit un entier complexe (construit par exemple en tirant à pile ou face chacun des bits le constituant) ou non, la probabilité que l'apprentissage soit exact est la même dans les deux cas. L'apprentissage exact est par conséquent bien plus probable dans ce modèle que dans les deux autres.

3.1.5 Un théorème du rasoir d'Occam

Avec le théorème d'Occam (voir théorème 1.3.3, page 38), le modèle de Valiant dispose d'un outil permettant de donner une condition suffisante afin de prouver l'apprenabilité d'une classe de concepts. Un algorithme d'Occam est, rappelons-le, basé sur l'heuristique selon laquelle une hypothèse assez courte et consistante avec l'échantillon est probablement une bonne approximation de la cible. Nous présentons dans cette partie, une variante de ce théorème pour le modèle PACS. Ce théorème affirme que s'il est possible de construire, pour tout concept d'une classe F, un échantillon représentatif de celui-ci, et s'il existe un algorithme ayant le comportement d'un algorithme d'Occam lorsque l'échantillon d'entrée contient un échantillon caractéristique alors F est PACS-apprenable. Nous verrons également que le PACS apprentissage basé sur un tel algorithme d'Occam permet d'assurer l'indépendance de l'apprenabilité d'une classe de concepts par rapport à la machine de Turing utilisée.

Avant d'énoncer le théorème d'Occam pour notre modèle, nous montrons préalablement un certain nombre de lemmes qui aideront à prouver le théorème annoncé.

La définition suivante formalise la propriété de simplicité pour un échantillon. Par la suite, les échantillons représentatifs devront satisfaire cette condition. Intuitivement, un échantillon est simple si tous les exemples qu'il contient peuvent « être décrits simplement » lorsque l'on connaît une représentation du concept.

Définition 3.1.5 *Soient X un domaine et F une classe de concepts associée à l'ensemble de représentations R. Soient une constante c, un concept f de F et r un nom de f. Un échantillon $S \subseteq X \times \mathbb{B}$ est* c-simple pour (f,r) *si et seulement si* $\forall\ (x,b) \in S, K(x \mid r) \leq c\log(|r|)$.

Le lemme suivant donne une borne sur la taille d'un échantillon c-simple.

Lemme 3.1.2 *Soit F une classe de concepts associée à l'ensemble de représentations R. Soit $f \in F$ avec $r \in R$ un nom de f. Si S est un échantillon c-simple pour (f,r) alors $|S| \leq 2|r|^c$.*

Preuve :
La preuve est assez évidente. Par définition, pour toute chaîne x de l'échantillon S, il existe un programme de longueur inférieure ou égale à $l = c\log(|r|)$ qui, connaissant r, calcule x. Le nombre de programmes de longueur au plus l est inférieur ou égal à $2^{l+1} = 2^{c\log(|r|)+1} = 2|r|^c$, d'où le lemme proposé. □

Le lemme suivant montre que si l'on tire un échantillon de taille suffisante (mais polynomiale) selon la distribution $\mathbf{m}_r$, on a de bonnes chances de tirer l'intégralité d'un échantillon c-simple pour f de nom r.

Lemme 3.1.3 *Soient f un concept d'une classe F associée à un ensemble de représentations R, r un nom de f et S_c un échantillon c-simple pour (f, r). Soit $\delta \in [0, 1[$. Si S est un échantillon tiré selon la distribution $\mathbf{m}_r$ avec $|S| \geq |r|^c(\ln(1/\delta) + \ln(2|r|^c))$ alors la probabilité que $S_c \subseteq S$ est supérieure ou égale à $1 - \delta$.*

Preuve :
La preuve se base essentiellement sur le fait que S_c contient peu d'éléments ayant chacun une grande probabilité d'être tiré. Soit $x \in S_c$. Par définition de S_c, $K(x \mid r) \leq c\log(|r|)$ donc $\mathbf{m}_r(x) \geq \lambda_r |r|^{-c}$. De plus $\lambda_r \geq 1$ donc $\mathbf{m}_r(x) \geq |r|^{-c}$. La probabilité que x ne soit pas tiré lors de N tirages est inférieure à $(1 - |r|^{-c})^N$. De plus, d'après le lemme 3.1.2 (page précédente), la taille de S_c est inférieure ou égale à $2|r|^c$, donc la probabilité qu'au moins un exemple de S_c ne soit pas tiré (c'est-à-dire que $S_c \not\subseteq S$) est inférieure à $2|r|^c(1 - |r|^{-c})^N$. Nous voulons que N soit suffisamment grand pour que cette probabilité soit bornée par δ. C'est-à-dire,

$$2|r|^c(1 - |r|^{-c})^N \leq \delta$$

En prenant le logarithme de chaque côté de l'inégalité,

$$\ln(2|r|^c) + N\ln(1 - |r|^{-c}) \leq \ln(\delta)$$

Soit,

$$N\ln(1 - |r|^{-c}) \leq \ln(\delta) - \ln(2|r|^c)$$

En utilisant l'approximation $\ln(1 - \alpha) \leq -\alpha$, il suffit que,

$$-|r|^{-c}N \leq \ln(\delta) - \ln(2|r|^c)$$

Soit en divisant par $-|r|^{-c} < 0$,

$$N \geq \frac{1}{-|r|^{-c}}(\ln(\delta) - \ln(2|r|^c))$$

En simplifiant, on obtient,

$$N \geq |r|^c(\ln(1/\delta) + \ln(2|r|^c))$$

ce qui termine la preuve. □

La définition qui suit présente la notion d'algorithme de PACS-Occam. Ce type d'algorithme est assez proche des algorithmes d'Occam présentés à la section 1.3.3 (page 37).

Conjointement aux algorithmes de PACS-Occam, nous définissons également la notion d'échantillon représentatif. Ces deux objets sont, en effet, étroitement liés puisqu'un échantillon est qualifié de représentatif s'il est simple et s'il permet à l'algorithme de PACS-Occam de sortir une hypothèse satisfaisante. L'algorithme de PACS-Occam quant à lui voit son comportement conditionné uniquement par le fait que l'échantillon qui lui est fourni contient un échantillon représentatif.

Définition 3.1.6 *Soit F une classe de concepts associée à l'ensemble de représentations R. Un algorithme* B *prenant en entrée un échantillon S et un entier l est un algorithme de* PACS-Occam *pour (F, R) s'il existe une constante c, une constante $0 < \alpha < 1$, deux entiers a et b et un polynôme p tels que pour tout concept f de nom r avec $|r| \leq l$, s'il existe un échantillon S_r c-simple pour (f, r), tel que si $S_r \subseteq S$ alors* B *sort un nom s d'un concept g tel que :*

- *g est consistant avec S,*
- *$|s| \leq a|S|^{\alpha}|r|^{b}$,*
- B *tourne en temps $p(|S|, l)$.*

Notons qu'un algorithme d'Occam classique est aussi un algorithme de PACS-Occam. Il est cependant plus facile de définir des algorithmes de PACS-Occam. En effet, la difficulté ou l'impossibilité à trouver, pour certaines classes de concepts, un algorithme d'Occam classique ayant un comportement satisfaisant quel que soit l'échantillon d'entrée est ici contournée en exigeant de l'algorithme de PACS-Occam un comportement valable que si certains exemples particuliers font partie de l'échantillon. On peut considérer que la contrainte forte du modèle PAC (apprentissage sous toutes les distributions de probabilité possibles) se retrouve dans le théorème d'Occam classique : celui-ci doit fonctionner quel que soit l'échantillon. Dans le modèle PACS par contre, les exemples simples du concept cible sont privilégiés par l'utilisation de la mesure $\mathbf{m}_r$. Ce « relâchement » sur la distribution de probabilité se traduit dans le théorème d'Occam PACS par l'utilisation d'un algorithme moins exigeant ne devant fonctionner correctement que dans le cas où l'échantillon contient un ensemble d'exemples particulier. Nous énonçons maintenant ce nouveau théorème d'Occam.

Théorème 3.1.4 *Soit F une classe de concepts associée à l'ensemble de représentations R. S'il existe un algorithme de PACS-Occam pour (F, R) alors F est poly-PACS-apprenable dans l'ensemble de représentations R.*

Avant de prouver ce théorème, il convient de montrer que lorsqu'il est utilisé avec un échantillon convenable (*i.e* contenant un échantillon caractéristique et de taille suffisante), l'algorithme B a un comportement répondant à l'heuristique d'Occam, à savoir que le concept court qu'il retourne a de bonnes chances d'être une hypothèse valable.

Lemme 3.1.5 *Soit P une distribution de probabilité, soient $\varepsilon \in]0, 1]$ et $\delta \in]0, 1]$. Soient f un concept, r un nom pour f et $l \geq |r|$. Si S est un échantillon tiré selon P avec $S_r \subseteq S$ et $|S| \geq [(\ln(1/\delta) + al^b + 1)/\varepsilon]^{\frac{1}{1-\alpha}}$, alors* B *avec les entrées S et l sort le nom d'un concept g tel que avec une probabilité supérieure ou égale à $1 - \delta$, $Erreur_{P,f}(g) \leq \varepsilon$.*

Preuve :
Le principe de la preuve est le même que dans le cas classique (*i.e.* cas du modèle PAC) et

repose sur le fait que si le nombre de concepts non valides est faible, alors on peut espérer les écarter avec un petit nombre d'exemples.

Soit f un concept de nom r. Par hypothèse $S_r \subseteq S$, donc l'algorithme B retourne une hypothèse courte d'un concept consistant avec S. Nous allons évaluer quelle doit être la taille de S pour que toutes les hypothèses courtes n'approximant pas f soient éliminées.

Considérons l'ensemble H_r des concepts g n'approximant pas f mais ayant un nom suffisamment court:

$$H_r = \{g \in F \mid Erreur_{P,f}(g) > \varepsilon \text{ et tel que } \exists\ s \in R(g) \text{ avec } |s| \leq a|S|^\alpha |r|^b\}$$

Soit $g \in H_r$. La probabilité qu'un exemple labellé pour f soit consistant avec g est inférieure à $1-\varepsilon$. La probabilité que N exemples labellés pour f soient consistants avec g est donc inférieure à $(1-\varepsilon)^N$. Soit $N = |S|$. Il est évident que la cardinalité de H_r est inférieure à $2^{aN^\alpha l^b+1}$ (nombre maximal de représentations de longueur au plus $aN^\alpha |r|^b$). Par conséquent, la probabilité qu'un échantillon de taille N soit consistant avec un des concepts de H_r est inférieure à $2^{aN^\alpha l^b+1}(1-\varepsilon)^N$. Les concepts de H_r sont susceptibles d'être retournés par B, nous voulons que N soit suffisant pour que la probabilité qu'un tel cas arrive soit inférieure à δ. C'est-à-dire,

$$2^{aN^\alpha l^b+1}(1-\varepsilon)^N \leq \delta$$

En passant au logarithme naturel de chaque côté de l'inégalité,

$$aN^\alpha l^b + N\ln(1-\varepsilon) \leq \ln(\delta) - 1$$

En divisant par $N^\alpha > 0$,

$$al^b + N^{1-\alpha}\ln(1-\varepsilon) \leq N^{-\alpha}(\ln(\delta) - 1)$$

Comme $N^{-\alpha}(\ln(\delta) - 1) \geq (\ln(\delta) - 1)$, il suffit que,

$$al^b + N^{1-\alpha}\log(1-\varepsilon) \leq \log(\delta) - 1$$

Soit,

$$N^{1-\alpha}\ln(1-\varepsilon) \leq \ln(\delta) - 1 - al^b$$

En utilisant l'approximation $\ln(1-\varepsilon) \leq -\varepsilon$, il suffit que,

$$-\varepsilon N^{1-\alpha} \leq \ln(\delta) - 1 - al^b$$

En divisant de chaque côté par $-\varepsilon < 0$,

$$N^{1-\alpha} \geq \frac{\ln(\delta) - 1 - al^b}{-\varepsilon}$$

C'est-à-dire, après simplification,

$$N \geq \left\lceil \frac{\ln(1/\delta) + 1 + al^b}{\varepsilon} \right\rceil^{\frac{1}{1-\alpha}}$$

Ce qui termine la preuve. □

Remarque 3.1.2 Dans cette preuve, nous n'avons absolument pas eu recours aux propriétés de S_r. Il fallait cependant obliger S_r à apparaître dans S afin d'assurer que B retourne une hypothèse courte.

Nous passons maintenant à la preuve du théorème d'Occam (théorème 3.1.4 (page 91)) proprement dite.

Preuve (du théorème 3.1.4 (page 91)) :

Afin de prouver que la classe F est poly-PACS-apprenable dans l'ensemble de représentations R s'il existe un algorithme de PACS-Occam polynomial, montrons que l'algorithme A qui suit est un algorithme polynomial de PACS-apprentissage pour F.

Algorithme de PACS-apprentissage A
Entrée : ε, δ, l
Début
Soit $N \geq max\{l^c(\ln(2/\delta) + \ln(2l^c)), [(\ln(2/\delta) + al^b + 1)/\varepsilon]^{\frac{1}{1-\alpha}}\}$
Tirer un échantillon S en appelant N fois l'oracle $Exemple_{X,\mathbf{m}_r}(f)$
Lancer B(S, l) sur $p(N, l)$ étapes
Si B retourne le nom d'un concept g
Alors Retourner le résultat de B
Sinon retourner une hypothèse quelconque
Fsi
Fin

Prouvons que l'algorithme A est effectivement un algorithme de PACS-apprentissage. Premièrement, d'après la définition d'un algorithme de PACS-apprentissage, on peut considérer uniquement le cas où $l \geq |r|$. On a alors $N \geq l^c(\log(2/\delta)+\log(2l^c)) \geq |r|^c(\log(2/\delta)+\log(2|r|^c))$. D'après le lemme 3.1.3 (page 90), la probabilité que $S_r \subseteq S$ est donc supérieure à $1 - \delta/2$. Ensuite, d'après le lemme 3.1.5 (page 91) et l'hypothèse sur N, si $S_r \subseteq S$, alors l'algorithme B sort le nom d'un concept g en moins de $p(N, l)$ étapes tel que avec la probabilité au moins $1 - \delta/2$, $Erreur_{\mathbf{m}_r,f}(g) \leq \varepsilon$. Par conséquent, A sort le nom d'un concept g de F tel que avec la probabilité au moins $(1 - \delta/2)^2 \geq 1 - \delta$, $Erreur_{\mathbf{m}_r,f}(g) \leq \varepsilon$. L'algorithme A est donc un algorithme de PACS-apprentissage pour la classe F. De plus, il est évident que cet algorithme tourne en temps polynomial en $1/\varepsilon, 1/\delta$ et l puisque le nombre d'exemples tirés est borné par un tel polynôme et seul un nombre polynomial d'étapes sur B est lancé.

En conclusion, la classe F est poly-PACS-apprenable pour la machine de Turing considérée. □

Remarque 3.1.3 La valeur de l est connue de l'algorithme d'apprentissage, il s'agit de la longueur maximale d'une représentation de concepts manipulés par l'algorithme. Cette valeur est utile à deux niveaux. D'abord, elle permet de borner la complexité en temps pour l'algorithme B. En effet, celui-ci doit au moins construire une représentation de longueur au maximum l. Ensuite, cette valeur l intervient dans le calcul de la taille minimale de S. En effet, le nombre de concepts appartenant à la classe augmente de la même manière que la valeur de l ; la taille de l'échantillon doit alors être plus grande afin de pouvoir différencier deux concepts.

Nous l'avons déjà dit, ce théorème est une variante du théorème d'Occam établi pour le modèle PAC. L'algorithme de PACS-Occam B utilisé ici est moins exigeant que dans le cas

classique puisqu'on attend de lui un comportement satisfaisant (*i.e* qu'il retourne en un temps polynomial une description courte d'un concept consistant avec l'échantillon) uniquement si l'échantillon contient un ensemble d'exemples particuliers. Cet ensemble d'exemples S_r constitue un *échantillon représentatif* pour le concept f et relativement à l'algorithme B. C'est cet échantillon représentatif qui « capte » les particularités de notre modèle, à savoir que les exemples simples de la cible doivent être passés à l'apprenant. Ici, les exemples peu complexes relativement à la cible font partie de l'échantillon représentatif.

Le modèle PACS définit l'apprenabilité d'une classe de concepts F, étant donnée une machine de Turing (celle utilisée pour tirer les exemples). En fait, rien n'empêche l'algorithme d'apprentissage d'avoir recours à un mécanisme particulier de la machine (*i.e* une procédure interne) l'aidant à satisfaire les contraintes d'apprentissage. Ainsi, la classe F peut être prouvée PACS-apprenable pour une machine de Turing particulière sans pour autant l'être pour toutes les machines de Turing. Notons que cette particularité de dépendance de l'apprentissage vis à vis de la machine de Turing utilisée se retrouve également dans le modèle de Li et Vitányi sans pour autant être clairement mentionnée par ceux-ci. Le théorème suivant donne une condition suffisante d'indépendance vis à vis de la machine de Turing préfixe. En fait, ce théorème établit que si une classe de concepts est montrée PACS-apprenable par l'utilisation d'un algorithme de PACS-Occam (c'est-à-dire en utilisant le théorème 3.1.4 (page 91)) alors l'indépendance vis à vis de la machine sera assurée.

Théorème 3.1.6 *Soit F une classe de concepts associée à l'ensemble de représentations R. S'il existe un algorithme de PACS-Occam pour (F, R), alors la poly-PACS-apprenabilité de F est indépendante de la machine de Turing préfixe universelle.*

Preuve :
Le théorème d'Occam PACS (théorème 3.1.4 (page 91)) montre que s'il existe un algorithme d'Occam pour (F, R) alors la classe F est poly-PACS-apprenable sur la machine de Turing préfixe universelle U fixée *a priori*. Considérons une autre machine de Turing préfixe universelle, disons U'. En utilisant le théorème d'invariance 1.4.1 (page 42), il existe une constante $C_{U,U'}$ telle que pour toutes chaînes x et r, $K_{U'}(x \mid r) \leq K_U(x \mid r) + C_{U,U'}$. Par conséquent, pour toute chaîne $x \in S_r$, $K_{U'}(x \mid r) \leq c\log(|r|) + C_{U,U'}$. De plus, on peut supposer, sans perdre de généralité, que $\log(|r|) \geq 1$, et donc $\forall\ x \in S_r, K_{U'}(x \mid r) \leq (c + C_{U,U'})\log(|r|)$. En posant $c' = c + C_{U,U'}$, l'ensemble des hypothèses relatives à l'algorithme de PACS-Occam (voir définition 3.1.6, page 91) sont satisfaites en utilisant la constante c' à la place de c. □

De la même manière que le théorème d'Occam classique a permis de montrer l'apprenabilité de quelques classes dans le modèle de Valiant, la variante PACS de celui-ci peut être utilisée comme outil permettant de prouver la PACS-apprenabilité de certaines classes de concepts. Pour ce faire, il « suffira » de montrer que pour tout concept de la classe, il est possible de construire un échantillon représentatif relativement à un algorithme de PACS-Occam qu'il conviendra également de définir.

Dans la suite, nous nous proposons d'étudier la PACS-apprenabilité de trois classes de concepts particulières. Ainsi, nous nous intéresserons successivement à la classe des DNF, des langages 0-réversibles puis enfin à celle des langages k-réversibles.

3.2 PACS-Apprenabilité des DNF

Rappelons avant tout que la classe des formules k-DNF est poly-PAC-apprenable (voir section 1.3.4, page 39) et que celle des formules $\log n$-DNF est poly-simple-PAC-apprenable (voir section 1.4.3, page 45).

La classe des formules k-terms-DNF c'est-à-dire contenant l'ensemble des formules DNF d'au plus k monômes (k étant fixé) n'est pas poly-PAC-apprenable en termes de k-terms-DNF sous la condition $RP \neq NP$ (voir section 1.3.4, page 39).

Dans cette section, nous allons montrer que la classe des formules DNF est poly-PACS-apprenable.

Soit n un entier fixé. Le domaine considéré est $X = \mathbb{B}^n$. Remarquons que X est un domaine fini. Un concept est une formule booléenne d'arité n, c'est-à-dire $f : \mathbb{B}^n \mapsto \mathbb{B}$. Une telle formule f peut aussi être vue comme un sous-ensemble $\hat{f}$ de $\mathbb{B}^n$ avec $x^n \in \hat{f}$ si et seulement si $f(x^n) = 1$.

Avant tout, il convient de préciser quel type de représentations des formules DNF nous allons utiliser. Les représentations de concepts tiennent une place importante dans notre modèle puisque la taille de celles-ci interviennent dans les mesures de complexité. Par conséquent, il est fondamental de fixer un système de représentations adapté à la classe de concepts étudiée, c'est-à-dire suffisamment proche des représentations classiques que l'on a de ces concepts.

Définition 3.2.1 *Soit n un entier. Soit R un ensemble de représentations associé à la classe $\mathcal{DNF}$ des formules booléennes sur n variables. R est un* système de représentations naturel *de la classe $\mathcal{DNF}$ si :*

1. *il existe un algorithme* T *tel que* $\forall\ f \in \mathcal{DNF}, \forall\ r \in R(f)$, $T(r) = \{m_1, m_2, \ldots, m_q\}$ *tel que* $f = \bigvee_{i=1}^{q} m_i$,
2. $\forall\ f \in \mathcal{DNF}, \forall\ r \in R(f)$, $n \leq |r| \leq O(nq)$.

Théorème 3.2.1 *Soit n un entier. Soit R un ensemble de représentations naturellement associé à la classe $\mathcal{DNF}$ sur $\mathbb{B}^n$. La classe $\mathcal{DNF}$ est PACS-apprenable polynomialement dans R.*

Preuve :
La preuve s'appuie sur l'utilisation du théorème d'Occam pour notre modèle (théorème 3.1.4 (page 91)). Nous allons dans un premier temps définir S_r ainsi qu'un algorithme B. Nous montrerons ensuite que les hypothèses du théorème d'Occam sont satisfaites.

Soit f une formule booléenne DNF. Soit $r \in R(f)$ avec $r = m_1 \vee \ldots \vee m_q$. Par la suite, si m est un monôme apparaissant dans r, nous écrirons $m \in r$. Pour chaque monôme $m \in r$, on définit les deux vecteurs exemples 0_m et 1_m tels que 0_m (respectivement 1_m) satisfait m et affecte la valeur 0 (resp. 1) à toutes les variables n'apparaissant pas dans m. Par exemple, si $m = x_1 x_3 \overline{x_4}$ et $n = 5$, $0_m = 10100$ et $1_m = 11101$. Il est clair que chacun de ces vecteurs est un exemple positif pour le concept f. Étant donnés un concept f et $r \in R(f)$, on définit $S_r = \{(0_m, 1) \mid m \in r\} \cup \{(1_m, 1) \mid m \in r\}$.

Soit l'algorithme B suivant (cet algorithme est fortement inspiré de l'algorithme d'apprentissage des formules $\log(n)$-DNF proposé par Li et Vitányi [LV91]) :

Algorithme B
Entrée : S un échantillon d'exemples labellés
Début
* Initialisation
Soit POS l'ensemble des exemples positifs de S
Soit NEG l'ensemble des exemples négatifs de S
Pour chaque paire d'exemple e et e' dans POS
construire un monôme contenant
x_i si e et e' ont un 1 à la position i
$\overline{x_i}$ si e et e' ont un 0 à la position i
et ne contenant ni x_i ni $\overline{x_i}$ sinon
Fpour
Parmi les monômes construits, effacer ceux impliquant des exemples de NEG

* Construction « gloutonne » d'une représentation
Soit r_h la somme des monômes restants
Soit r_g la formule nulle
Tant que $POS \neq \emptyset$
Pour chaque $m \in r_h$
Soit POS_m l'ensemble des exemples de POS satisfaisant m
Fpour
Soit m le premier monôme tel que $Card(POS_m)$ est maximal
$r_g \leftarrow r_g \vee m$
$POS = POS - POS_m$
Ftq

* Résultat
Retourner r_g une représentation de l'hypothèse g
Fin

Étant donné un concept cible f, nous avons donc défini un algorithme de consistance B ainsi qu'un échantillon particulier S_r. Montrons que toutes les hypothèses du théorème 3.1.4 (page 91) sont satisfaites.

- **Il existe une constante c telle que pour tout $f \in \mathcal{DNF}$ de nom r, l'échantillon S_r est c-simple.** Il s'agit de montrer qu'il existe une constante c telle que pour tout $f \in \mathcal{DNF}$ de nom r on a $\forall\ x \in S_r,\ K(x \mid r) \leq c\log(|r|)$. Supposons que $x = 0_m$ pour $m \in f$. Il est évident qu'il existe un programme simple, ne dépendant que de la constante n, générant x à partir d'une représentation de m. Par conséquent $\exists\ c_1$ tel que $K(x \mid m) \leq c_1$. Il en est de même pour $x = 1_m$ avec c_1' à la place de c_1.

  Maintenant, étant donné que le système de représentations R associé à $\mathcal{DNF}$ est naturel, il existe un algorithme T générant un ensemble M de q monômes m_i tel que $f = \bigvee_i m_i$. Ainsi, pour générer un monôme m quelconque de cette représentation, il est suffisant d'avoir la valeur i de la position de m dans M, avec $i \leq q \leq |r|$. Il existe donc une constante c_2 telle que $K(m \mid r) \leq c_2\log(|r|)$. Par conséquent, en utilisant l'inégalité 2 du théorème 1.4.4 (page 43), on a $\forall\ x \in S_r, K(x \mid r) \leq K(x \mid m) + K(m \mid r) \leq$

$c_2 \log(|r|) + \max(c_1, c_1')$. Sans perdre de généralité, on peut supposer que $\log(|r|) \geq 1$ et donc $K(x \mid r) \leq (\max(c_1, c_1') + c_2) \log(|r|)$, ce qui prouve le résultat attendu.

- **L'algorithme B retourne le nom d'un concept consistant avec S.** Aucun monôme du concept g calculé par B ne satisfait d'exemple négatif de S. En effet, ceux-ci sont effacés de l'ensemble des monômes possibles. De plus, pour tout exemple positif de l'échantillon, il existe un monôme le satisfaisant (partie « gloutonne ») de l'algorithme.

- **L'algorithme B tourne en temps polynomial lorsque $S_r \subseteq S$.** La construction de la formule r_h peut être implantée en temps $O(n|S|^3)$ et r_h contient au plus $|S|^2$ monômes. Supposons que $S_r \subseteq S$. Dans ce cas, tous les monômes de r apparaissent dans r_h. À chaque étape i de la construction de r_g, la taille de l'ensemble POS_m sélectionné est au moins $|POS(i)|/|r|$ (avec $POS(i)$ représentant l'ensemble POS à l'étape i). En effet, la formule f (et donc h) couvre POS et r la représentation de f a au plus q monômes (avec $q \leq |r|$). Le nombre d'itérations utiles à la construction de r_g est donc au plus $|r| \log(|POS|) \leq |r| \log(|S|)$. Le nombre d'étapes utiles à la construction de chaque POS_m est inférieur à $n|S|^3 \leq |r|.|S|^3$ puisque r_h contient au maximum $|S|^2$ monômes et chaque exemple est composé de n bits. Donc, lorsque $S_r \in S$, l'algorithme B tourne en temps $O(|r|^2|S|^3 \log(|S|))$, c'est-à-dire qu'il existe un p tel que B peut être implanté pour tourner en temps $p(|S|, l)$ avec $l \geq |r|$.

- **Lorsque $S_r \subseteq S$, il existe un nom « court » pour le concept g retourné par l'algorithme B.** Le concept g est construit dans la partie « gloutonne » de l'algorithme. Nous avons vu que dans cette partie de l'algorithme, si $S_r \subseteq S$, le nombre d'itérations est au plus $|r| \log(|S|)$. Or, à chaque itération, on ajoute exactement un monôme à r_g ; par conséquent, r_g contient au plus $|r| \log(|S|)$ monômes. Il existe donc $\alpha \in]0, 1]$, ainsi que deux entiers $a = 1$ et $b = 1$ tels que le nom r_g de g retourné par B satisfait $|r_g| \leq a|S|^\alpha |r|^b$.

  Remarquons que c'est la partie gloutonne de l'algorithme B qui assure cette contrainte. La formule r_h calculée jusque là est en effet une hypothèse consistante avec l'échantillon, elle pourrait par conséquent être retournée par l'algorithme. Cependant, afin que B puisse prétendre à être un algorithme de PACS-Occam, la représentation de l'hypothèse qu'il retourne doit être suffisamment courte, d'où l'utilité de la partie gloutonne.

Nous avons montré que l'algorithme B est un algorithme de PACS-Occam qui se comporte convenablement lorsque l'échantillon d'entrée contient S_r ; le théorème 3.1.4 (page 91) peut être appliqué, ce qui termine la preuve. □

Nous nous proposons d'étudier dans la suite l'apprenabilité des langages réversibles (voir la section 2.1.3 (page 57) pour une présentation détaillée de cette classe de langages). Pour une question de clarté et de simplicité, nous nous focaliserons d'abord sur la classe des langages 0-réversibles. Les résultats seront ensuite étendus aux classes de langages k-réversibles.

3.3 PACS-apprentissage des Langages Réversibles

Dans cette partie, nous supposerons que l'ensemble des représentations R associé à la classe des langages réguliers est un codage d'automates déterministes. Ainsi, pour $L \in \mathcal{R}eg$, le seul

nom de R pour L est le codage binaire de $A(L)$, l'automate canonique de L. Notons qu'un automate déterministe à n états peut être codé au moyen d'une chaîne binaire de longueur $O(n\log(n))$. Pour ce faire, il suffit de coder l'ensemble des transitions de l'automate ainsi que la liste des états terminaux. On peut supposer que l'état initial est toujours le premier état. Chaque transition est représentée sous la forme: $q_1 a q_2$ avec q_1 l'état initial de la transition, q_2 l'état d'arrivée et a la lettre de l'alphabet Σ labellant la transition. La longueur du codage d'une transition est donc en $O(\log(n)\log(|\Sigma|))$. De plus, le nombre de transitions est borné par $|\Sigma|n$. Le nombre d'états terminaux quant à lui est borné par n et le codage permettant de décrire l'état i comme état terminal est de longueur $O(\log(n))$. De plus, si l'automate canonique d'un langage L a n états alors on peut supposer que $n \leq |R(L)|$.

Comme indiqué dans les chapitres précédents, on considère l'ordre standard sur les mots de Σ^*, c'est-à-dire si $\Sigma = \{0,1\}$, l'énumération des mots de Σ^* selon cet ordre est:

$$\varepsilon, 0, 1, 00, 01, 10, 11, 000, \ldots$$

Si $u, v \in \Sigma^*$, $u < v$ (respectivement $u \leq v$) signifie que le numéro d'ordre de u est inférieur (resp. inférieur ou égal) à celui de v. Tout ensemble non vide de mots possède un *plus petit mot* qui est le mot dont le numéro d'ordre est inférieur à tous les autres, c'est-à-dire si $E \subseteq \Sigma^*$, $\min(E) = u$ tel que $u \in E$ et $\forall\, v \in E - \{u\}, u < v$.

3.3.1 Cas des langages 0-réversibles

La section 2.1.3 (page 57) du chapitre précédent présentait un certain nombre de résultats concernant l'apprenabilité de la classe des langages 0-réversibles. Cette classe est identifiable à la limite par exemples positifs [Ang82a]. La classe des langages simples-0-réversibles est simple-PAC-apprenable [LV91]. Cependant, nous ignorons si la classe des langages 0-réversibles est ou non PAC-apprenable.

Notons $\mathcal{R}ev_0$ la classe des langages 0-réversibles et $\mathcal{R}ev_0^{\leq n}$ celle des langages 0-réversibles dont l'automate canonique a au plus n états.

Dans la suite de cette section, nous allons d'abord montrer que pour tout n, la classe $\mathcal{R}ev_0^{\leq n}$ est poly-PACS-apprenable par exemples positifs. Nous prouverons ensuite que la classe $\mathcal{R}ev_0$ est PACS-apprenable par exemples positifs seuls en temps usuellement polynomial.

On rappelle que pour tout langage $L \in \mathcal{R}ev_0$, avec $A(L) = (Q, \{q_0\}, \{q_f\}, \Sigma, \delta)$, il existe un échantillon caractéristique noté S_L^0 (voir définition 2.2.7, page 74), tel que L est le plus petit langage 0-réversible contenant S_L^0. De plus, sur l'entrée S vérifiant $S_L^0 \subseteq S \subseteq L$, l'algorithme ZR proposé par Angluin retourne l'automate canonique de L.

Notation :
Soit $A = (Q, \{q_0\}, F, \Sigma, \delta)$ un automate déterministe. Soit $q \in Q$ un état quelconque de A. On note $u_A(q)$ le plus petit mot labellant un chemin menant de l'état initial q_0 à l'état q. On note $v_A(q)$ le plus petit mot labellant un chemin menant de l'état q à un état final. Formellement, $u_A(q) = \min(\{u \in \Sigma^* \mid \delta(q_0, u) = q\})$ et $v_A(q) = \min(\{v \in \Sigma^* \mid \delta(q, v) \in F\})$.

Avant de passer aux résultats de PACS-apprenabilité, il convient de remarquer que:

Proposition 3.3.1 *Soit L un langage 0-réversible. Si l'automate canonique de L, noté $A(L) = (Q, \{q_0\}, \{q_f\}, \Sigma, \delta)$, contient n états, alors S_L^0, l'échantillon caractéristique de L, contient au maximum $(1+|\Sigma|)n$ chaînes et toutes les chaînes de S_L^0 ont une longueur inférieure à $2n+1$.*

Preuve :
Pour une question de simplicité, nous utiliserons la notation $u(q)$ (resp. $v(q)$) à la place de $u_{A(L)}(q)$ (resp. $v_{A(L)}(q)$). L'échantillon caractéristique d'un langage 0-réversible L est défini par: $S_L^0 = \{u(q)v(q) \mid q \in Q\} \cup \{u(q)av(q') \mid q, q' \in Q, a \in \Sigma, q' = \delta(q, a)\}$ (voir définition 2.2.7 , page 74). En ce qui concerne la borne sur le nombre de mots de S_L^0, nous avons donc,

$$|S_L^0| = |\{u(q)v(q) \mid q \in Q\}| + |\{u(q)av(q') \mid q, q' \in Q, a \in \Sigma, q' = \delta(q, a)\}|$$

Avec,

$$|\{u(q)v(q) \mid q \in Q\}| \leq |Q| = n$$

Et,

$$|\{u(q)av(q') \mid q, q' \in Q, a \in \Sigma, q' = \delta(q, a)\}| \leq |Q|.|\Sigma| = n.|\Sigma|$$

Ce qui conduit finalement à,

$$|S_L^0| \leq (1 + |\Sigma|)n$$

Le résultat concernant la borne sur la longueur des mots de S_L^0 est tout aussi évident. En effet, la longueur maximale des mots de S_L^0 est inférieure à la longueur maximale des mots de l'ensemble $\{u(q)av(q') \mid q, q' \in Q, a \in \Sigma, q' = \delta(q, a)\}$, c'est-à-dire $|u(q)| + |v(q)| + 1$. De plus, la longueur d'un mot minimal menant d'un état de l'automate à un autre état est inférieure à n ; donc $|u(q)| \leq n$ et $|v(q)| \leq n$. On obtient ainsi la borne proposée. □

Nous en arrivons maintenant au premier résultat de PACS-apprenabilité pour la classe des langages 0-réversibles,

Théorème 3.3.1 *Pour tout n, la classe $\mathcal{R}ev_0^{\leq n}$ est poly-PACS-apprenable par exemples positifs et l'apprentissage est probablement exact*[2].

Preuve :
Ici encore la preuve s'appuie sur le théorème d'Occam (théorème 3.1.4 (page 91)). Nous allons décrire un algorithme de PACS-Occam ainsi qu'un échantillon représentatif étant donné un langage de $\mathcal{R}ev_0^{\leq n}$. L'ensemble des hypothèses du théorème 3.1.4 (page 91) sera ensuite vérifié. Soit un entier n, L un langage de $\mathcal{R}ev_0^{\leq n}$ et r un nom pour L. On définit $S_r = S_L^0$ et on propose comme algorithme d'Occam l'algorithme B suivant:

Algorithme B
Entrée : S
Constante connue : n
Début
Soit S' l'ensemble des exemples positifs de S de longueur au plus $2n + 1$.
Lancer ZR sur l'entrée S'
Sortir un nom pour ZR(S')
Fin

2. On parle d'apprentissage probablement exact lorsque l'hypothèse inférée est égale à la cible en cas de succès du processus (c'est-à-dire avec la probabilité au moins $1 - \delta$).

Montrons que toutes les conditions du théorème 3.1.4 (page 91) sont satisfaites:

- **Il existe une constante c telle que pour tout $L \in \mathcal{R}ev_0^{\leq n}$ de nom r, l'échantillon S_r est c-simple.** Montrons qu'il existe une constante c telle que $\forall\ x \in S_r$, $K(x \mid r) \leq c\log(|r|)$. Soient $L \in \mathcal{R}ev_0^{\leq n}$ et n_0 le nombre d'états de l'automate canonique de L ($n_0 \leq n$). Soit $x \in S_r$. Pour générer x, il est suffisant d'avoir S_r ainsi que l'index i de x dans cet ensemble. D'après la proposition 3.3.1 (page 98),

$$|S_r| \leq (1+|\Sigma|)n_0 \leq (1+|\Sigma|)n$$

Donc en posant $c_1 = (1+|\Sigma|)$,

$$i \leq c_1 n$$

Par conséquent, $K(x \mid S_r) \leq \log(c_1) + \log(n)$. De plus, par définition de S_r, il existe un algorithme construisant cet échantillon directement à partir de l'automate canonique de L (c'est-à-dire r) ; d'où,

$$K(S_r \mid r) = O(1)$$

D'après l'inégalité 2 du théorème 1.4.4 (page 43), on a,

$$K(x \mid r) \leq K(x \mid S_r) + K(S_r \mid r) + O(1)$$

D'où,

$$K(x \mid r) \leq \log(n) + O(1)$$

D'après les hypothèses sur R, $n \leq |r|$ donc,

$$K(x \mid r) \leq \log(|r|) + O(1)$$

En supposant que $\log(|r|) \geq 1$, il existe donc une constante c telle que,

$$K(x \mid r) \leq c\log(|r|)$$

Pour la suite on considère $L \in \mathcal{R}ev_0^{\leq n}$ et n_0 le nombre d'états de l'automate canonique de L.

- **Si $S_r \subseteq S$, l'algorithme `B` retourne le nom d'un concept consistant avec S.** Si $S_r \subseteq S$, alors $S_r \subseteq S'$ puisque S' est l'ensemble des mots positifs de longueur inférieure ou égale à $2n+1$ (condition vérifiée par l'ensemble des mots de S_r d'après la proposition 3.3.1 (page 98) et $n_0 \leq n$). Or, l'algorithme `ZR` retourne l'automate canonique reconnaissant L si l'échantillon d'entrée (S' ici) contient l'échantillon $S_L^0 = S_r$. L'hypothèse retournée est donc consistante avec L et par conséquent avec $S \subseteq L$.

- **L'algorithme `B` tourne en temps polynomial lorsque $S_r \subseteq S$.** La construction de l'échantillon S' se fait évidemment en temps polynomial par rapport à $|S|$. L'algorithme `ZR` tourne en temps quasi-linéaire par rapport à $\|S'\|$ (voir section 2.2.3, page 72). Par construction, les mots de S' ont une longueur inférieure ou égale à $2n+1$, d'où $\|S'\| \leq (2n+1)|S'| \leq (2n+1)|S|$. La complexité en temps de l'algorithme `B` est donc polynomial par rapport à $|S|$.

- **Lorsque $S_r \subseteq S$, il existe un nom « court » pour le concept L' retourné par l'algorithme B.** Si $S_r \subseteq S'$, l'algorithme ZR retourne l'automate canonique du langage L. La vérification de cette hypothèse devient alors évidente.

Chaque condition du théorème 3.1.4 (page 91) étant vérifiée, la poly-PACS-apprenabilité de la classe $\mathcal{R}ev_0^{\leq n}$ est prouvée. Enfin, l'apprentissage est probablement exact puisque l'algorithme ZR retourne l'automate canonique de L si $S_r \subseteq S$; or ceci arrive avec la probabilité $1-\delta$ d'après le lemme 3.1.3 (page 90) concernant les échantillons c-simples. □

Le théorème précédent montre que si l'on connaît une borne sur le nombre d'états de l'automate canonique du langage 0-réversible cible, celui-ci peut être appris exactement dans notre modèle avec la probabilité $1-\delta$.

Le théorème que l'on se propose de montrer dans la suite ne fait plus d'hypothèse sur la taille de l'automate canonique de la cible. Cela permet ainsi de montrer la PACS-apprenabilité de la classe $\mathcal{R}ev_0$ elle-même. Cependant, étant donnée l'absence d'hypothèse sur la taille des représentations des objets de la classe, on ne peut prétendre à un résultat aussi fort que précédemment. Ainsi, la complexité en temps de l'apprentissage devient ici *usuellement polynomiale*, c'est-à-dire que l'algorithme tourne en temps polynomial avec la probabilité $1-\delta$ (*i.e.* dans les cas de succès).

Théorème 3.3.2 *La classe $\mathcal{R}ev_0$ des langages 0-réversibles est PACS-apprenable en temps usuellement polynomial par exemples positifs seuls. L'apprentissage est probablement exact.*

À la différence de la preuve du théorème 3.3.1 (page 99), nous ne pouvons pas ici nous contenter d'utiliser l'algorithme ZR pour définir un algorithme de PACS-Occam. En effet, notre définition d'apprenabilité impose de ne traiter que des exemples de longueur polynomiale par rapport à la taille de la cible. Par conséquent, nous allons définir directement un algorithme de PACS-apprentissage qui itère sur le nombre d'états du concept cible. Avant de passer à la preuve de ce théorème, il convient de montrer un ensemble de lemmes « techniques ». Ces lemmes ont pour but de montrer qu'il existe des mots (ou des préfixes de mots) de longueur polynomiale par rapport à la taille de la cible permettant d'abandonner une hypothèse de taille inférieure à celle de la cible (lemmes 3.3.5 (page suivante), 3.3.6 (page suivante) et 3.3.7 (page 104)). Le lemme 3.3.8 (page 106) établit quant à lui qu'en tirant un ensemble d'exemples de taille suffisante, ces mots particuliers ont une grande chance d'être tirés.

Lemme 3.3.3 *Soit $A = (Q, \{q_0\}, \{q_f\}, \Sigma, \delta)$ un automate 0-réversible. Si $u_1 w \in L(A)$ et $u_2 w \in L(A)$ alors $\delta(q_0, u_1) = \delta(q_0, u_2)$.*

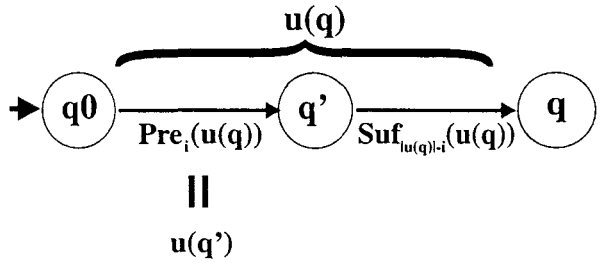
Preuve :
La preuve de ce lemme est immédiate en utilisant le lemme 2.1.8 (page 63) avec $k = 0$. □

Le lemme suivant donne une propriété sur les préfixes du plus petit mot permettant d'atteindre un état quelconque d'un automate depuis l'état initial.

Lemme 3.3.4 *Soit $A = (Q, \{q_0\}, F, \Sigma, \delta)$ un automate déterministe à n états. Soit $q \in Q$. Si q' est un état de Q tel que $\exists\, i \leq |u_A(q)|$ et $q' = \delta(q_0, Pre_i(u_A(q)))$ alors $Pre_i(u_A(q)) = u_A(q')$.*

Preuve :

Graphiquement, ce résultat peut être représenté comme suit :



Notons $u = u_A(q)$. Supposons qu'il existe un mot u' tel que $u' < Pre_i(u)$ et $\delta(q_0, u') = \delta(q_0, Pre_i(u)) = q'$. Remarquons que,

$$u = Pre_i(u)Suf_{|u|-i}(u)$$

Nous avons évidemment,

$$\delta(q_0, u'Suf_{|u|-i}(u)) = q$$

Or par hypothèse $u' < Pre_i(u)$ donc,

$$u'Suf_{|u|-i}(u) < Pre_i(u)Suf_{|u|-i}(u) = u$$

Ceci contredit le fait que u est le plus petit mot menant de q_0 à q et prouve donc le lemme. □

Pour les lemmes qui suivent, nous considérons L et L' deux langages 0-réversibles avec $L' \subset L$. Soient S_L^0 et $S_{L'}^0$ les échantillons caractéristiques de L et L'. Les automates canoniques respectifs de ces langages sont $A = (Q, \{q_0\}, \{q_f\}, \Sigma, \delta)$ et $A' = (Q', \{q_0'\}, \{q_f'\}, \Sigma, \delta')$. Soit n le nombre d'états de A et n' celui de A'. Ici encore, par souci de simplicité, $u(q)$ (resp. $u'(q)$) représente $u_A(q)$ (resp. $u_{A'}(q)$) et $v(q)$ (resp. $v'(q)$) représente $v_A(q)$ (resp. $v_{A'}(q)$).

Lemme 3.3.5 *Sous les hypothèses précédemment énoncées,* $\exists\ u \in S_L^0$ *tel que* $u \notin S_{L'}^0$.

Preuve :
Ce lemme est une conséquence directe de la définition des échantillons caractéristiques. On rappelle qu'un échantillon caractéristique S_L^0 d'un langage 0-réversible L est tel que L est le plus petit langage 0-réversible contenant S_L^0. Par conséquent L' est le plus petit langage 0-réversible contenant $S_{L'}^0$; de même L est le plus petit langage 0-réversible contenant S_L^0. Supposons qu'un tel mot u n'existe pas, alors $S_L^0 \subseteq S_{L'}^0$. Dans ce cas, le plus petit langage contenant S_L^0 est L', ce qui contredit les hypothèses. □

Lemme 3.3.6 *Sous les hypothèses précédemment énoncées,* $(S_L^0 - S_{L'}^0) \cap L' = \emptyset$.

Preuve :
Nous allons montrer que s'il existe un mot appartenant à $(S_L^0 - S_{L'}^0) \cap L'$ alors ce mot appartient aussi à $S_{L'}^0$, ce qui prouvera le lemme par l'absurde. Soit $u \in (S_L^0 - S_{L'}^0) \cap L'$. Le mot u appartient à S_L^0 donc u est soit de la forme $u(q_1)v(q_1)$ avec $q_1 \in Q$ soit de la forme $u(q_1)av(q_2)$ avec $q_1, q_2 \in Q, a \in \Sigma, q_2 = \delta(q_1, a)$.

- **Cas 1 : u est de la forme $u(q_1)v(q_1)$ avec $q_1 \in Q$.** Le mot u appartient à L' donc il existe un état $q'_1 \in Q'$ tel que $q'_1 = \delta'(q'_0, u(q_1))$ et $\delta'(q'_1, v(q_1)) = q'_f$.
  Comme $u'(q'_1)$ est la plus petite chaîne menant de q'_0 vers q'_1 on a,

$$u'(q'_1) \leq u(q_1)$$

  De plus, le mot $u'(q'_1)v(q_1)$ appartient à L' et donc aussi à L étant donné que $L' \subseteq L$. On a donc $u = u(q_1)v(q_1) \in L$ (par hypothèse sur u) et $u'(q'_1)v(q_1) \in L$; en appliquant le lemme 3.3.3 (page 101) sur ces deux chaînes on a donc,

$$\delta(q_0, u(q_1)) = \delta(q_0, u'(q'_1)) = q_1$$

  Par définition de $u(q_1)$,

$$u(q_1) \leq u'(q'_1)$$

  Et donc finalement,

$$u(q_1) = u'(q'_1)$$

  Maintenant, la chaîne $v'(q'_1)$ étant la plus petite chaîne menant de q'_1 vers q'_f on a,

$$v'(q'_1) \leq v(q_1)$$

  Le mot $u'(q'_1)v'(q'_1) = u(q_1)v'(q'_1)$ appartient à L', il appartient donc également à L. Par définition de $v(q_1)$,

$$v(q_1) \leq v'(q'_1)$$

  Ceci implique finalement que,

$$v(q_1) = v'(q'_1)$$

  Par conséquent, $u = u(q_1)v(q_1) = u'(q'_1)v'(q'_1)$. Par construction de $S_{L'}^0$, u appartient donc à cet échantillon, ce qui contredit l'hypothèse de départ et prouve le lemme pour ce premier cas.

- **Cas 2 : u est de la forme $u(q_1)av(q_2)$ avec $q_1, q_2 \in Q, a \in \Sigma, q_2 = \delta(q_1, a)$.** Le mot u appartient à L', il existe donc deux états $q'_1, q'_2 \in Q'$ tels que $\delta'(q'_0, u(q_1)) = q'_1$, $\delta'(q'_1, a) = q'_2$ et $\delta'(q'_2, v(q_2)) = q'_f$. Le même raisonnement que précédemment montre que,

$$u(q_1) = u'(q'_1)$$

  Et que,

$$v(q_2) = v'(q'_2)$$

  Par construction de $S_{L'}^0$, le mot $u = u(q_1)av(q_2) = u'(q'_1)av'(q'_2)$ appartient donc à $S_{L'}^0$ ce qui contredit ici encore l'hypothèse de départ.

Ce qui termine la preuve. □

Les deux lemmes précédents montrent qu'il existe au moins un mot différenciant S_L^0 de $S_{L'}^0$ et qu'un tel mot n'appartient pas à L'. Le lemme suivant établit que pour de tels mots, il est suffisant de regarder les $2n'$ premières lettres pour décider de leur non appartenance à L'. On rappelle que $Pr(L')$ est l'ensemble des préfixes du langage L' (voir section 2.1.2, page 55).

Lemme 3.3.7 *Sous les hypothèses précédemment énoncées, si $u \in S_L^0 - L'$ alors soit $|u| < 2n'$ et $u \notin L'$, soit $|u| \geq 2n'$ et $Pre_{2n'}(u) \notin Pr(L')$.*

Preuve :
Soit $u \in S_L^0 - L'$. Si $|u| < 2n'$, il n'y a rien à prouver. Supposons maintenant que $|u| \geq 2n'$. Le mot u appartient à S_L^0, il est donc de la forme $u = u(q)v(q)$ avec $q \in Q$ (le cas où $u = u(q_1)av(q_2)$ avec $q_1, q_2 \in Q, a \in \Sigma$ et $q_2 = \delta(q_1, a)$ est similaire). Soit $w = Pre_{2n'}(u)$. Nous allons montrer par l'absurde que w ne peux pas être préfixe d'un mot de L'. Pour ce faire, supposons que w est le préfixe d'un mot de L', c'est-à-dire qu'il existe un mot $v \in \Sigma^*$, tel que $wv \in L'$. Montrons qu'alors $|u| = |u(q)| + |v(q)| < 2n'$.

- **Montrons d'abord que $|u(q)| \leq n' - 1$.** Comme $|w| = 2n'$ et que w est le préfixe d'un mot de L', la chaîne $Pre_{n'}(w)$ labelle un chemin passant deux fois par un même état de A'. Il existe donc un mot s de longueur inférieure à $n' - 1$ tel que,

$$\delta'(q_0', s) = \delta'(q_0', Pre_{n'}(w))$$

Comme $|w| = 2n'$, il est évident que,

$$w = Pre_{n'}(w)Suf_{n'}(w)$$

D'où,

$$\delta'(q_0', sSuf_{n'}(w)v) = \delta'(q_0', wv)$$

Comme $wv \in L'$, on a même,

$$\delta'(q_0', sSuf_{n'}(w)v) = \delta'(q_0', wv) = q_f'$$

On a donc $sSuf_{n'}(w)v \in L'$ et comme $L' \subset L$, ce mot appartient aussi à L. Supposons que $|u(q)| \geq n'$. Dans ce cas, étant donné que $w = Pre_{2n'}(u(q)v(q))$,

$$Pre_{n'}(u(q)) = Pre_{n'}(w)$$

Comme $wv \in L$, cela implique que,

$$Pre_{n'}(u(q))Suf_{n'}(w)v \in L$$

Par application du lemme 3.3.3 (page 101) sur les mots $sSuf_{n'}(w)v$ et $Pre_{n'}(u(q))Suf_{n'}(w)v$, on obtient,

$$\delta(q_0, s) = \delta(q_0, Pre_{n'}(u(q))) = q_1$$

D'après le lemme 3.3.4 (page 101), on a de plus,

$$Pre_{n'}(u(q)) = u(q_1)$$

Mais, $|s| \leq n' - 1$ et $|u(q_1)| = n'$ donc $s < u(q_1)$ ce qui est impossible par définition de $u(q_1)$. L'hypothèse selon laquelle $|u(q)| \geq n'$ n'est donc pas possible et donc $\boldsymbol{|u(q)| \leq n' - 1}$.

- **Montrons maintenant que $|v(q)| \leq n' - 1$.** Intéressons-nous à la deuxième partie du mot w, c'est-à-dire $w' = Suf_{2n'-|u(q)|}(w)$ (notons que $w = u(q)w'$). On vient de montrer que $|u(q)| \leq n' - 1$ et comme $|w| = 2n'$, on a donc,

$$|w'| > n'$$

Soit $q'_1 = \delta'(q'_0, u(q))$. Comme $wv \in L'$ on a,

$$\delta'(q'_1, w'v) = q'_f$$

La chaîne $Pre_{n'}(w')$ labelle un chemin depuis q'_1 et passant deux fois par un même état de A' puisque A' contient n' états. Par conséquent, il existe un mot t de longueur inférieure ou égale à $n' - 1$, labellant un chemin depuis q'_1 vers $\delta(q'_1, Pre_{n'}(w'))$. On a donc,

$$\delta(q'_1, tSuf_{|w'|-n'}(w')v) = \delta'(q'_1, w'v) = q'_f$$

Et même,

$$\delta(q'_0, u(q)tSuf_{|w'|-n'}(w')v) = \delta'(q'_0, u(q)w'v) = q'_f$$

Ainsi, le mot $u(q)tSuf_{|w'|-n'}(w')v$ appartient à L' et par conséquent aussi à L (puisque $L' \subset L$).

Il en est de même pour le mot $u(q)w'v$ qui peut aussi s'écrire $u(q)Pre_{n'}(w')Suf_{|w'|-n'}(w')v$.

Par application du lemme 3.3.3 (page 101) sur les chaînes $u(q)tSuf_{|w'|-n'}(w')v \in L$ et $u(q)Pre_{n'}(w')Suf_{|w'|-n'}(w')v \in L$ on a,

$$\delta(q_0, u(q)t) = \delta(q_0, u(q)Pre_{n'}(w'))$$

Soit encore étant donné que $\delta(q_0, u(q)) = q$,

$$\delta(q, t) = \delta(q, Pre_{n'}(w'))$$

Supposons que $|v(q)| \geq n'$. Comme $|w'| > n'$ et $w = Pre_{2n'}(u(q)v(q)) = u(q)w'$, on a alors,

$$Pre_{n'}(v(q)) = Pre_{n'}(w')$$

Soit $q_1 = \delta(q, t)$, nous avons alors

$$q_1 = \delta(q, t) = \delta(q, Pre_{n'}(v(q)))$$

Et par conséquent,

$$\delta(q, tSuf_{|v(q)|-n'}(v(q))) = \delta(q, Pre_{n'}(v(q))Suf_{|v(q)|-n'}(v(q))) = \delta(q, v(q)) = q_f$$

Or $|t| \leq n' - 1$ donc,

$$|tSuf_{|v(q)|-n'}(v(q))| \leq n' - 1 + |v(q)| - n'$$

Soit encore,

$$|tSuf_{|v(q)|-n'}(v(q))| \leq |v(q)| - 1$$

Il existe donc un mot de longueur inférieure à celle de $v(q)$ labellant un chemin entre q et q_f. Ceci est impossible par définition de $v(q)$, et donc $\boldsymbol{|v(q)| \leq n' - 1}$.

Finalement, nous avons montré que si w est préfixe d'un mot de L' alors $|u| = |u(q)| + |v(q)| \leq 2n' - 2$, ce qui contredit l'hypothèse initiale. Donc le préfixe w de longueur $2n'$ de u ne peut pas être un préfixe d'un mot de L', ce qui termine la preuve. □

D'après le lemme 3.3.7 (page 104), il est suffisant de lire au plus les $2n'$ premières lettres d'un mot de $S_L^0 - L'$ pour détecter que celui-ci n'appartient pas à L'. Le lemme suivant établit que dans un échantillon de L, de taille polynomiale et tiré selon la distribution de Solomonoff-Levin relative à L, la probabilité de trouver un tel mot est grande.

Lemme 3.3.8 *Sous les hypothèses énoncées, il existe un mot $u \in S_L^0 - L'$ tel que $\mathbf{m}_r(u) \geq \lambda/(n' \log^2(n'))$ où r est un nom pour L et λ est une constante dépendant uniquement de la machine de Turing préfixe de référence U. Il existe de plus un polynôme q ne dépendant que de U tel qu'un échantillon S tiré selon $\mathbf{m}_r$ et de taille supérieure à $q(n', 1/\delta)$ contient un mot $u \in S_L^0 - L'$ avec la probabilité au moins $1 - \delta/2^{n'}$.*

Preuve :
Étant donné L, l'échantillon S_L^0 est, par définition, calculable. D'après le lemme 3.3.5 (page 102), il existe au moins un mot appartenant à S_L^0 et pas à $S_{L'}^0$. De plus, il existe un tel mot u dans les $(|\Sigma| + 1)n' + 1$ premiers mots de S_L^0 puisque $|S_{L'}^0| \leq (|\Sigma| + 1)n'$ (voir proposition 3.3.1, page 98). Ce mot u n'appartient pas à L' d'après le lemme 3.3.6 (page 102).
Montrons que pour un tel mot u, nous avons $K(u \mid r) \leq \log(n') + 2\log(\log(n')) + O(1)$ où r est un nom pour L. Pour générer u, il est suffisant de disposer de l'échantillon caractéristique S_L^0 et de l'index i de u dans celui-ci. Aussi, d'après le théorème 1.4.3 (page 42),

$$K(u \mid S_L^0) \leq K(i) + O(1) \leq \log(i) + 2\log(\log(i)) + O(1)$$

Le mot u se trouve dans les $(|\Sigma| + 1)n' + 1$ premiers mots de S_L^0, donc en supposant que $n' \geq 1$,

$$i \leq (|\Sigma| + 1)n' + 1 \leq (|\Sigma| + 2)n'$$

D'où

$$K(u \mid S_L^0) \leq \log(n') + 2\log(\log(n')) + O(1)$$

De plus, par définition de S_L^0, le programme construisant S_L^0 connaissant r a une longueur constante donc,

$$K(S_L^0 \mid r) \leq O(1)$$

On trouve finalement que,

$$K(u \mid r) \leq K(u \mid S_L^0) + K(S_L^0 \mid r) + O(1) \leq \log(n') + 2\log(\log(n')) + O(1)$$

Par conséquent, il existe λ tel que,

$$\mathbf{m}_r(u) \geq \lambda/(n' \log^2(n'))$$

Utilisons cette inégalité pour montrer, de manière classique, l'existence du polynôme q. Supposons que l'on tire un échantillon S selon la distribution $\mathbf{m}_r$. La probabilité de ne pas sortir une telle chaîne u, en un tirage, est inférieure à $1 - \lambda/(n' \log^2(n'))$. La probabilité de ne pas sortir une telle chaîne u en N tirages est inférieure à $(1 - \lambda/(n' \log^2(n')))^N$. Cherchons la valeur minimale de N pour que cette probabilité soit au plus égale à $\delta/2^{n'}$. Il faut que,

$$\left[1 - \frac{\lambda}{n' \log^2(n')}\right]^N \leq \frac{\delta}{2^{n'}}$$

En prenant le logarithme naturel de chaque côté de l'inégalité,

$$N \ln\left(1 - \frac{\lambda}{n' \log^2(n')}\right) \leq \ln\left(\frac{\delta}{2^{n'}}\right)$$

En utilisant l'approximation $\ln(1 - \varepsilon) \leq -\varepsilon$, il suffit que,

$$-N \frac{\lambda}{n' \log^2(n')} \leq \ln\left(\frac{\delta}{2^{n'}}\right)$$

En divisant par $-\lambda/(n' \log^2(n')) < 0$,

$$N \geq \frac{\ln\left(\frac{\delta}{2^{n'}}\right)}{-\frac{\lambda}{n' \log^2(n')}}$$

D'où en simplifiant,

$$N \geq \frac{1}{\lambda}(n' \ln(2) + \ln(\frac{1}{\delta}))(n' \log^2(n'))$$

Ce qui prouve l'existence du polynôme $q(n', 1/\delta)$. □

En résumé, au moyen de ces lemmes nous avons montré qu'il existe un mot u propre à l'échantillon caractéristique de L, tel qu'il suffit de lire au plus ses $2n'$ premières lettres pour vérifier sa non appartenance au langage L'. De plus, si l'on tire un échantillon de taille suffisante mais polynomiale, un tel mot u a de bonnes chances d'être tiré. Intuitivement, ce type de mots va caractériser les différents langages les uns par rapport aux autres. L'idée de l'algorithme d'apprentissage proposé dans la suite est de faire une succession d'hypothèses sur la taille de l'automate cible puis de confronter l'automate généré à un échantillon test contenant avec une grande probabilité un tel mot caractéristique qui permettra, éventuellement, d'abandonner l'hypothèse en cours (simplement en lisant le préfixe de ce mot).

Cet algorithme est présenté dans la preuve du théorème 3.3.2 (page 101):

Preuve (du théorème 3.3.2 (page 101)) :
Soient un langage 0-réversible L de nom r et n le nombre d'états de l'automate canonique de L. La preuve du théorème 3.3.1 (page 99) établit que les échantillons S_L^0 sont c-simples. Par la suite, p représente le polynôme tel que si l'on tire un échantillon S selon $\mathbf{m}_r$ de taille

supérieure à $p(n, 1/\delta)$ alors $S_L^0 \subseteq S$ avec une probabilité supérieure à $1 - \delta$ (voir lemme 3.1.3 , page 90).

Considérons l'algorithme de PACS-apprentissage qui suit :

Algorithme de PACS-apprentissage A
Entrée : δ
Début
Soit $S \leftarrow \emptyset$
Soit $n' \leftarrow 1$
Soit $NEX \leftarrow 0$
Tant que pas Fin_du_monde
Faire $p(n', 2/\delta) - NEX$ appels à $Exemple_{\mathbf{m}_r}(L)$
Ajouter les nouveaux exemples à S
Soit S' l'ensemble des exemples positifs de S de longueur inférieure ou égale à $2n' + 1$
Soit A' l'automate retourné par ZR(S')
Si A' a n' états
Alors
Faire $q(n', 2/\delta)$ appels à $Exemple_{\mathbf{m}_r}(L)$
Soit $TEST$ l'ensemble des nouveaux exemples
Si Le préfixe de longueur $2n'$ de tous les mots de $TEST$
sont des préfixes de mots acceptés par A'
Alors Retourner A' et s'arrêter
Fsi
Fsi
$NEX \leftarrow p(n', 2/\delta)$
$n' \leftarrow n' + 1$
Ftq
Fin

Montrons que cet algorithme est bien un algorithme d'apprentissage probablement correct dans notre modèle. Soient L le langage cible et n le nombre d'états de l'automate canonique de L. Remarquons d'abord que tout automate A' calculé par ZR est tel que $L(A') \subseteq L$ puisque $S' \subseteq L$ et ZR(S') retourne le plus petit langage 0-réversible contenant S'. L'algorithme A peut sortir un automate déterministe A' à n' états avec $n' < n$ et $L(A') \subset L$. Dans ce cas, il n'y a pas de mots vérifiant le lemme 3.3.7 (page 104) dans $TEST$; ceci arrive avec une probabilité inférieure à $\delta/2^{n'+1}$ d'après le lemme 3.3.8 (page 106). La probabilité que l'algorithme A retourne un DFA A' à n' états avec $n' < n$ est inférieure à $\sum_{i<n} \delta/(2^{i+1}) \leq \delta/2$. Par conséquent, la probabilité que n' atteigne la valeur n est supérieure à $1 - \delta/2$. De plus, quand $n' = n$, si $S_L^0 \subseteq S$, alors l'algorithme A sort un nom pour L (apprentissage exact). La probabilité que $S_L^0 \subseteq S$ quand $n' = n$ est supérieure à $1 - \delta/2$ (puisque l'échantillon a une taille supérieure à $p(n, 2/\delta)$). Finalement, l'algorithme A retourne un nom pour L avec la probabilité supérieure à $(1 - \delta/2)^2 \geq 1 - \delta$. Il est alors facile de vérifier que dans ce cas, A tourne en temps polynomial en $1/\delta$. □

3.3.2 Cas des langages k-réversibles

L'ensemble des résultats concernant les langages 0-réversibles peuvent être étendus aux classes de langages k-réversibles (k étant fixé). Ainsi, la classe des langages k-réversibles dont l'automate canonique a un nombre d'états borné est poly-PACS-apprenable. Sans l'hypothèse sur la taille des automates canonique, les langages k-réversibles sont apprenables en temps usuellement polynomial dans notre modèle. La plupart des preuves sont semblables[3] à celles données dans le cas des langages 0-réversibles. Aussi, nous ne donnerons dans cette section que les idées conduisant aux preuves. Le lecteur intéressé par les preuves complètes pourra se reporter à l'annexe A (page 197).

Soit $\mathcal{R}ev_k$ la classe des langages k-réversibles, $\mathcal{R}ev_k^n$ celle des langages k-réversibles dont l'automate canonique contient exactement n états et enfin $\mathcal{R}ev_k^{\leq n}$ la classe des langages k-réversibles dont l'automate canonique contient au plus n états.

Soit $L \in \mathcal{R}ev_k$. On rappelle qu'il est possible de construire un échantillon caractéristique S_L^k tel que L est le plus petit langage k-réversible contenant S_L^k. On rappelle également qu'Angluin décrit un algorithme k-RI qui retourne l'automate canonique du plus petit langage k-réversible contenant l'échantillon positif passé en entrée.

Théorème 3.3.9 *Pour tout n, $\mathcal{R}ev_k^{\leq n}$ est PACS-apprenable polynomialement et l'apprentissage est probablement exact.*

Idée de preuve :
La preuve est strictement la même que dans le cas des langages 0-réversibles et s'appuie sur le théorème d'Occam pour notre modèle (théorème 3.1.4 (page 91)). Ici encore, les échantillons caractéristiques S_L^k jouent le rôle d'échantillons représentatifs S_r. La taille d'un tel échantillon est bornée par $|\Sigma|^k.(|\Sigma|+1).n+\frac{1-|\Sigma|^{k+1}}{1-|\Sigma|}$. L'algorithme d'Occam B est identique à celui proposé pour le cas des langages 0-réversible en remplaçant l'appel à ZR par l'appel à k-RI. La borne sur la longueur des exemples de S est ici $2n+k+1$. □

Théorème 3.3.10 *La classe $\mathcal{R}ev_k$ des langages k-réversibles est PACS-apprenable par exemples positifs seuls en temps usuellement polynomial et l'apprentissage est probablement exact.*

Idée de preuve :
La preuve, comme dans le cas des langages 0-réversibles s'appuie sur le fait qu'il est possible, avec une forte probabilité, de tirer un exemple caractéristique permettant de distinguer entre deux langages k-réversibles. À chaque étape de l'algorithme, celui-ci suppose que la cible est un concept de $\mathcal{R}ev_k^n$ avec n de plus en plus grand. Ainsi, à chaque étape, un échantillon S est tiré et l'algorithme appelle k-RI avec cet échantillon S. L'hypothèse retournée par k-RI est ensuite évaluée sur un autre échantillon TEST. Si le test est satisfait, l'hypothèse est retournée sinon le processus recommence avec la valeur $n+1$. □

3. Quoique un peu plus complexes de par la présence de k.

3.3.3 Conclusion

L'ensemble des résultats concernant l'apprenabilité des langages k-réversibles dans le modèle PACS est satisfaisant puisqu'il n'existe pas, à notre connaissance, d'équivalent dans le modèle de Valiant.

Le modèle de Li et Vitányi a permis de montrer l'apprenabilité des classes de langages simples-0-réversibles mais la notion de simplicité introduite pour définir cette classe semble artificielle ; cette classe est de fait trop restreinte.

Cependant, l'ensemble de ces résultats suppose connue la valeur de k et la question de savoir si la classe des langages réversibles est ou non PACS-apprenable par exemples positifs seuls reste ouverte. Dans la suite, nous verrons que la classe des langages réguliers est PACS-apprenable, ce qui implique que la classe des langages réversibles est PACS-apprenable dans $\mathcal{R}eg$.

3.4 Résultats Généraux sur le Modèle PACS

Cette section a pour but d'évoquer certains résultats récents relatifs à notre modèle. D'abord, nous verrons que la classe des langages réguliers est PACS-apprenable. Ensuite, nous présenterons brièvement deux propriétés (dues à Castro et Guijarro [CG98]) établissant des relations entre d'autres modèles d'apprentissage et le notre. Enfin, un dernier paragraphe présentera notre point de vue sur le modèle PACS.

3.4.1 PACS-apprentissage des langages réguliers

Un certain nombre de classes de concepts ont été prouvées PACS-apprenables. Parmi celles-ci, on trouve notamment les classes de langages k-réversibles qui sont PACS-apprenables par exemples positifs seuls. Rappelons que le résultat est d'autant plus intéressant que ces classes ne sont pas connues PAC-apprenables. Dès lors, la question qui vient naturellement à l'esprit est : « est-ce que la classe des langages réguliers est PACS-apprenable?[4] ». Parekh et Honavar [PH97] ont étudié cette question et propose une réponse affirmative dans le cas où le nombre d'états de l'automate cible est connu. Ils proposent d'utiliser pour ce faire l'algorithme `RPNI` [OG92] (voir section 2.2.2, page 70) en tant qu'algorithme d'Occam PACS. Un premier résultat concerne l'apprenabilité de la classe $\mathcal{R}eg^{\leq n}$, c'est-à-dire la classe des langages réguliers dont l'automate canonique a au plus n états (n fixé) :

Théorème 3.4.1 (Parekh et Honavar, [PH97]) *Pour tout n, la classe $\mathcal{R}eg^{\leq n}$ est poly-PACS apprenable et l'apprentissage est probablement exact.*

Idée de preuve :
La preuve repose sur le théorème d'Occam pour notre modèle (voir théorème 3.1.4, page 91). L'algorithme d'Occam utilisé est l'algorithme `RPNI`. Rappelons que cet algorithme retourne en temps polynomial, l'automate canonique du langage cible L si l'échantillon d'entrée contient un échantillon représentatif de L pour `RPNI` (voir définition 2.2.6, page 71). Ce type d'échantillon est montré c-simple, il est par conséquent utilisé comme échantillon représentatif au

4. Rappelons que la classe des langages réguliers n'est pas apprenable polynomialement dans le modèle PAC sous certaines hypothèses cryptographiques, et ce quel que soit le système de représentations [KV94].

sens PACS-Occam. . □

Un deuxième résultat proposé par Parekh et Honavar concerne la PACS-apprenabilité de la classe $\mathcal{R}eg$, c'est-à-dire sans contrainte sur le nombre d'états des automates canoniques des cibles :

Théorème 3.4.2 (Parekh et Honavar, [PH97]) *La classe des langages réguliers $\mathcal{R}eg$ est poly-PACS-apprenable.*

Idée de preuve :
L'algorithme permettant d'obtenir cette généralisation est comparable à notre algorithme permettant d'apprendre la classe $\mathcal{R}ev_k$. Ainsi, l'algorithme d'apprentissage proposé recherche une hypothèse valable pour une valeur de n (le nombre d'états de la cible) de plus en plus grande. Pour chaque résultat retourné par `RPNI`, un échantillon de test est utilisé pour évaluer cette hypothèse. □

Ce résultat tente de répondre à une question de Pitt [Pit89] demandant si la classe $\mathcal{DFA}$ est PAC-apprenable lorsque les exemples sont tirés avec une distribution uniforme ou une autre distribution simple et connue. Nous contestons cependant ce théorème. En effet, la preuve proposée par Parekh et Honavar est très similaire à la preuve du théorème 3.3.2 (page 101) concernant la classe des langages 0-réversibles à la différence qu'il n'existe pas, dans ce cas, de lemme similaire aux lemmes techniques (lemmes 3.3.5 (page 102) à 3.3.7 (page 104)). Lors de l'évaluation d'une hypothèse par un ensemble test de mots, l'algorithme ne peut donc pas se contenter d'utiliser des préfixes de longueur polynomiale des mots tests. Aussi, la longueur du plus long exemple lu lors du test de validité doit être prise en compte dans le calcul de la complexité de l'algorithme[5]. Rappelons que ceci pose un problème dans la mesure où un mot tiré au moyen de $\mathbf{m}_r$ peut être arbitrairement long. Précisons que le fait de prendre en compte la longueur du plus long mot tiré dans le calcul de la complexité de l'algorithme, permet de définir des algorithmes de complexité non-polynomiale ; de cette manière, il devient possible de prouver l'apprenabilité de n'importe quelle classe de concepts. Le théorème proposé par Parekh et Honavar ainsi que sa preuve sont par conséquent justes mais sont utilisés dans une variante du modèle PACS qui nous semble sans grand intérêt car trop permissif.

3.4.2 Rapports entre le modèle PACS et d'autres modèles

L'idée principale du modèle PACS consistant à privilégier les exemples adaptés au concept cible n'est pas nouvelle. D'autres modèles définissent des ensembles d'apprentissage relatifs à la cible afin d'aider l'apprenant ; citons par exemple [MS91, GK91, GM93]. Ces ensembles d'apprentissage peuvent être rapprochés de nos échantillons caractéristiques. Le modèle proposé par Goldman et Mathias par exemple suppose que l'enseignant construit un ensemble d'apprentissage adapté à la cible. D'autres exemples sont ajoutés à cet ensemble par une tierce personne (appelé *adversaire*) de manière à empêcher la collusion entre l'apprenant et l'enseignant. Le modèle demande alors à l'apprenant d'identifier exactement la cible à partir de cet ensemble d'exemples.

5. Cette manière de définir le modèle d'apprentissage PACS est à rapprocher de la définition du modèle PAC proposé par Pitt [Pit89].

Récemment, Denis et Gilleron ont proposé un nouveau modèle appelé *modèle PAC avec enseignant*[6] [DGS97, DG97]. Ici encore, l'enseignant utilise une représentation du concept cible afin de définir un ensemble d'apprentissage. Le modèle est défini de la même manière que le modèle PAC classique à la différence que seules sont considérées les distributions n'affectant pas la probabilité nulle aux exemples de l'ensemble d'apprentissage. Notons que dans ce modèle, la complexité en temps de l'algorithme d'apprentissage dépend également de la plus petite probabilité d'un exemple de l'ensemble d'apprentissage selon la distribution considérée. Remarquons que le problème de la collusion est ici évité de par le fait que le modèle n'est pas restreint à une seule distribution de probabilité (une telle distribution de probabilité est qualifiée de *distribution enseignante*[7]). Une version du théorème d'Occam est également établie pour ce modèle. Lorsque l'ensemble d'apprentissage ne contient que des exemples simples par rapport au concept cible (c'est-à-dire simplement descriptible en connaissant le concept) le modèle est dit *PAC avec enseignants simples*. Notons que tout ensemble d'apprentissage calculable est également simple (voir proposition 2 de [DG97]). Dès lors, des rapports entre ce modèle et le modèle PACS sont établis et notamment :

Théorème 3.4.3 (Denis et Gilleron, [DG97]) *Soit F une classe de concepts associé à l'ensemble de représentations R. Si C est PAC apprenable avec enseignants simples dans R alors C est PACS apprenable dans R.*

Un certain nombre de classes de concepts telles que les listes de décisions, les formules DNF ont été montrées PAC apprenables avec enseignant simple. Il en est de même pour la classe $\mathcal{R}eg^{\leq n}$ associée à l'ensemble de représentations DFA. La classe des langages k-réversibles est également montrée PAC apprenable avec enseignant simple et par exemples positifs seuls. En utilisant le théorème 3.4.3 , il est donc possible de montrer que l'ensemble de ces classes sont également PACS apprenables. Ces résultats sont un autre moyen pour montrer la PACS apprenabilité des classes étudiées dans ce chapitre.

Dans un récent rapport, Castro et Guijarro s'intéressent aux relations existantes entre le modèle PACS et deux autres modèles d'apprentissage [CG98]. Ainsi, ils montrent d'abord le théorème suivant :

Théorème 3.4.4 (Castro et Guijarro, [CG98]) *Soit F une classe de concepts associée à l'ensemble de représentations R. Si F est apprenable polynomialement en utilisant un oracle d'équivalence et un autre d'appartenance, alors F est PACS-apprenable.*

Le principe de la preuve repose essentiellement sur le fait que l'information passée lors de requêtes d'appartenance ou d'équivalence a une complexité de Kolmogorov faible. Intuitivement, ce résultat renforce les idées de base qui ont conduit à formaliser le modèle PACS. En effet, la distribution de probabilité $\mathbf{m}_r$ modélise un enseignant qui adapte ses exemples en fonction de la cible, de la même manière que l'on peut se représenter les deux oracles comme étant un professeur répondant le mieux possible aux questions de ses élèves.

Castro et Guijarro présentent un second résultat établissant une relation entre le modèle simple-PAC et le modèle PACS. Ainsi, sous certaines conditions raisonnables, si une classe F est PACS-apprenable alors sa restriction aux concepts ayant une représentation simple est simple-PACS-apprenable (voir [CG98] pour plus de détails). Ce résultat traduit le fait que l'on

6. En anglais, *PAC learning under helpful distribution.*
7. En anglais *helpful distribution.*

peut transformer la condition de simplicité des exemples (du modèle PACS) en une hypothèse de simplicité des concepts. Intuitivement, on peut imaginer que s'il est possible d'enseigner un cours au moyen d'exemples adaptés, les éléments les plus simples de ce cours peuvent être appris au moyen d'exemples intrinsèquement peu complexes.

3.4.3 Point de vue et critique sur le modèle

Le modèle PACS présente plusieurs avantages. D'abord, il formalise un certain nombre d'idées *a priori* relatives à l'apprentissage et notamment l'adéquation des exemples au concept cible. Il permet de montrer l'apprenabilité de nombreuses classes de concepts non triviales et non « artificielles » (à la différence du modèle simple-PAC).

Cependant, il subsiste deux problèmes majeurs qui empêchent ce modèle d'être complètement satisfaisant : le premier est lié à la non calculabilité de la distribution $\mathbf{m}_r$; le second est connu sous le terme de collusion. Un troisième inconvénient à souligner est l'absence de résultat négatif dans ce modèle, c'est-à-dire de classe montrée non PACS-apprenable. Remarquons qu'il est très probable que ce dernier inconvénient soit directement lié au fait que ce modèle est trop peu contraignant, et donc à l'existence d'algorithmes d'apprentissage collusifs pour des classes aussi importantes que $\mathcal{R}eg$.

Non calculabilité de $\mathbf{m}_r$

Le premier de ces problèmes se rencontre déjà dans le modèle de Li et Vitányi : il s'agit de la non calculabilité de $\mathbf{m}_r$. De ce fait, le modèle PACS reste essentiellement un modèle théorique, puisque l'on ne connaît pas, à ce jour, de mesure de probabilité permettant d'approcher au mieux $\mathbf{m}$ et $\mathbf{m}_r$. Le modèle d'apprentissage PAC avec enseignant proposé par Denis et Gilleron [DG97] permet de se passer de la mesure $\mathbf{m}_r$ et de sa non calculabilité, puisque ce modèle est défini pour toutes les distributions de probabilité enseignantes et non plus uniquement au moyen de $\mathbf{m}_r$.

Problème de collusion

Le deuxième problème posé par le modèle PACS est d'ordre plus « philosophique ». En effet, étant donnés un concept f et sa représentation r, le problème vient de la possibilité de « coder » la chaîne r sous forme d'exemple (positif ou non) $\hat{r}$ et de « décoder » $\hat{r}$ en r en temps polynomial. Si c'est le cas, alors $\hat{r}$, le codage de r, est un exemple représentatif pour le concept f. Il existe alors un algorithme d'« apprentissage » qui décode chacun des exemples de l'échantillon en concept et qui teste la consistance de ce concept avec l'échantillon. Ce phénomène connu sous le nom de *collusion* reste encore mal compris ; à ce jour, il n'existe pas de définition pleinement satisfaisante de la collusion.

La figure 3.4 (page 115) présente un algorithme d'apprentissage collusif type. Précisons que d'autres types d'algorithmes peuvent être considérés par certains comme étant également collusifs bien que cette propriété ne fasse pas l'unanimité de la communauté.

Classiquement, l'échantillon S est tiré au moyen de la distribution de probabilité fixé *a priori*. Dans le cas de l'apprentissage PACS, il s'agit donc de la distribution $\mathbf{m}_r$ avec r une représentation du concept cible.

Dans un premier temps, nous allons montrer qu'un tel algorithme ne permet pas d'apprendre les langages réguliers par exemples positifs seuls.

Proposition 3.4.1 *La classe des langages réguliers $\mathcal{R}eg$ associée à l'ensemble de représentations $\mathcal{DFA}$ n'est pas apprenable au moyen d'un algorithme collusif type par exemples positifs seuls.*

Preuve :
Considérons l'algorithme collusif type de la figure 3.4 (page suivante), avec k fixé. Nous allons montrer que quelle que soit la valeur de k, il existe une infinité de langages qui ne peuvent pas être inférés par l'algorithme collusif type en ne considérant que les exemples positifs. Plus précisément, nous allons montrer qu'il existe une infinité de langages pour lesquels il n'existe pas de codage possible. Soit ϕ la fonction de codage, c'est-à-dire une fonction prenant en entrée un langage régulier et retournant un ensemble fini d'au plus k mots; soit formellement :

$$\begin{array}{rccl} \phi: & \mathcal{R}eg & \rightarrow & \{P \subseteq \Sigma^* \mid |P| \leq k\} \\ & L & \mapsto & \phi(L) \end{array}$$

avec,

$$\phi(L) \in L \ \textbf{(cond 1)}$$

et,

$$L_1 \neq L_2 \Rightarrow \phi(L_1) \neq \phi(L_2) \ \textbf{(cond 2)}$$

La condition (cond. 1) assure que le codage du langage est bien un exemple positif de celui-ci, la condition (cond. 2) quant à elle permet à tout code d'être décodé sans ambiguïté.

Parmi les langages réguliers, on trouve les langages finis à un seul mot. Étant donnée la condition (cond. 1), tout langage L de ce type doit être codé par $\phi(L) = \{s\}$ avec $L = \{s\}$. Par conséquent, toutes les parties à un seul élément sont utilisées pour coder les langages à un seul mot. Les langages à deux mots sont également des langages réguliers. Soit $L' = \{s_1, s_2\}$ un tel langage. La condition (cond. 2) implique que L ne peut être codé uniquement par $\{s_1\}$ ou par $\{s_2\}$ car ces codes sont déjà utilisés pour coder deux langages à un seul mot. Par conséquent, $\phi(L) = \{s_1, s_2\}$. L'ensemble des codages des langages à deux mots utilise donc tous les codes à deux éléments. Le même raisonnement pour tous les langages d'au plus k mots implique que tous les codes d'au plus k éléments sont utilisés pour coder ceux-ci.

Par conséquent, il n'existe plus de code disponible pour les langages contenant plus de k mots et l'algorithme de la figure 3.4 (page suivante) n'est donc pas un algorithme d'apprentissage des langages réguliers par exemples positifs. □

Algorithme `Apprentissage_collusif`
Constante : k
Début
Soit S un échantillon de taille suffisante
Considérer toutes les parties de S à k exemples (soit $C^k_{|S|}$ parties)
Décoder chaque partie (soit $C^k_{|S|}$ hypothèses)
Choisir et retourner l'hypothèse la meilleure
Fin

FIG. 3.4 - Algorithme collusif type.

Un raisonnement similaire permet de montrer le même résultat pour les langages 0-réversibles, à savoir :

Proposition 3.4.2 *La classe des langages 0-réversibles $\mathcal{R}ev_0$ associée à l'ensemble de représentations $\mathcal{DFA}$ n'est pas apprenable au moyen d'un algorithme collusif type par exemples positifs seuls.*

Preuve :
La preuve est similaire à celle concernant les langages réguliers, c'est-à-dire qu'ici encore, il s'agit de montrer qu'il existe une infinité de langages pour lesquels il n'existe pas de codage. La fonction de codage utilisée ici est définie de la même manière que dans la preuve précédente. Rappelons d'abord qu'un langage est 0-réversible si et seulement si son automate canonique est 0-réversible. De plus, il est clair que pour tout $p \leq 0$, il existe au moins un langage 0-réversible contenant exactement p mots.

Tous les langages contenant un seul mot sont évidemment 0-réversibles. Pour les mêmes raisons que dans la preuve précédente, toutes les parties de Σ^* à un seul élément sont utilisées pour coder tous ces langages. Un raisonnement similaire à celui utilisé dans la preuve précédente montre que tout langage L 0-réversible contenant p mots (avec $p \leq k$) est codé par une partie contenant exactement p éléments. En effet, tout sous-langage de L est 0-réversible et ne peut donc servir à coder L. On a alors deux choix pour coder L : soit utiliser une partie de moins de p éléments de Σ^* mais ne codant pas un langage 0-réversible, soit coder L par la partie de p éléments contenant tous les mots de L. La première alternative ne peut être retenue puisque dans ce cas, la condition (cond 1) n'est plus valable. Aussi, L est-il coder par lui même.

Il s'en suit que tous les langages à plus de k éléments ne peuvent plus être codés, ce qui montre donc que les langages 0-réversibles ne sont pas apprenables par un algorithme collusif type par exemples positifs seuls. □

Les deux propositions qui précèdent montrent que les résultats de PACS apprenabilité pour ces classes, présentés dans ce chapitre, conservent tout leur intérêt. Il en est de même pour la classe des DNF pour laquelle il est possible de montrer qu'un tel algorithme n'est pas applicable.

Par contre, si l'on ne se restreint pas aux seuls exemples positifs, la classe des langages réguliers est poly-PACS-apprenable au moyen d'un algorithme collusif. Rappelons d'abord que pour tout DFA minimal A, on peut construire (simplement) un échantillon $S_A^{\mathtt{RPNI}}$ tel que si $S \supseteq S_A^{\mathtt{RPNI}}$ alors l'algorithme RPNI avec S comme entrée retourne exactement l'automate A. La figure 3.5 présente un algorithme collusif de PACS-apprentissage pour les langages réguliers (on suppose que A est l'automate canonique du langage cible).

Algorithme Apprend_reguliers_collusif
Entrée : δ
Début
Soit S_1 un échantillon de taille suffisante pour être δ-sûr que le code de A soit tiré
Soit S_2 un échantillon de taille suffisante pour être δ-sûr que $S_A^{\mathtt{RPNI}}$ soit entièrement tiré
Décoder les éléments de S_1 et conserver les automates minimaux
Conserver ceux qui sont cohérents avec S_2 (A se trouve parmi eux)
Pour tous les automates B restants,
ne conserver que ceux pour lesquels $S_B^{\mathtt{RPNI}}$ est inclus dans S_2
Fin

FIG. 3.5 - Algorithme collusif apprenant les langages réguliers dans le modèle PACS.

Cet algorithme retourne effectivement l'automate canonique A. En effet, supposons que l'algorithme retourne un automate B différent de A. Par construction, B est cohérent avec l'échantillon S_2 et $S_B^{\mathtt{RPNI}}$ est inclus dans S_2. Par conséquent $\mathtt{RPNI}(S_2) = B$ (propriété de l'algorithme RPNI). Mais on doit également avoir $\mathtt{RPNI}(S_2) = A$, on aboutit donc à une contradiction et donc $B = A$.

Il est assez remarquable de constater que l'algorithme RPNI n'est jamais utilisé dans cet algorithme collusif ; c'est la raison pour laquelle il s'agit d'un algorithme collusif. C'est le seul fait qu'il existe qui permet de conclure. Notons également que cet argument peut être repris tel quel dans le modèle de Goldman et Mathias [GM93].

Pareck et Honavar ont proposé un algorithme de PACS-apprentissage des langages réguliers basé sur l'algorithme RPNI (voir théorème 3.4.2, page 111). La longueur du plus long exemple lu devait cependant être prise en compte dans la complexité en temps de l'algorithme d'apprentissage. L'algorithme que nous venons de présenter prouve que la classe des langages réguliers est effectivement poly-PACS-apprenable[8] mais au moyen d'un algorithme collusif.

Raisonnablement, de tels algorithmes ne peuvent évidemment pas être qualifiés d'algorithmes d'apprentissage. En fait, on est en droit de se demander s'il n'y a pas collusion dès lors que l'échantillon contient un échantillon représentatif ; en effet, dans de nombreux cas, toute l'information relative à la cible est contenue dans un tel échantillon. La seule différence par rapport à ce qui vient d'être évoquée réside dans le fait que l'information est alors distribuée sur de nombreux exemples et non concentrée sur un seul. Mais comment distinguer ces deux cas? Comment distinguer un enseignant pédagogue qui adapte ses exemples à la cible de l'enseignant peu scrupuleux qui cache la réponse parmi les informations afin d'aider

8. Sans modifier la définition de notre modèle.

un étudiant complice? Enfin, est-ce qu'un modèle d'apprentissage dans lequel la collusion est possible doit pour autant être qualifié de non raisonnable, dès lors que l'on n'utilise pas cette possibilité dans la définition des algorithmes?

Conclusion à la Première Partie

Dans cette première partie, nous avons défini le formalisme relatif à la théorie de l'apprentissage automatique. Un certain nombre de modèles théoriques « classiques » ont été présentés ainsi que leurs caractéristiques propres. Chacun d'eux tente de modéliser l'idée que l'on se fait de l'apprentissage. Toute la difficulté vient du fait que l'ensemble des connaissances que l'on a de ce domaine n'est qu'une liste d'intuitions établie en observant le comportement de l'humain en état d'apprentissage. Quoiqu'il en soit, il apparaît qu'un certain nombre de qualités telles que la rapidité d'apprentissage ou bien encore le droit à l'erreur ne peuvent pas ne pas être prises en compte dans la modélisation d'un tel acte. Les différents modèles qui ont été étudiés présentent leurs avantages et leurs inconvénients qui, suivant le point de vue sous lequel on les considère, font de ceux-ci leur force et leur faiblesse.

Il semble que la notion d'échantillon caractéristique occupe une place importante dans l'apprentissage humain. Lorsque cette notion est prise en compte dans la formalisation de l'apprentissage, il apparaît que le potentiel d'apprentissage du modèle s'en trouve renforcé. Cette idée a permis de définir un nouveau modèle d'apprentissage appelé modèle PACS.

Lorsque l'on étudie l'apprenabilité d'une classe de concepts dans un modèle donné, le problème principal consiste à trouver un algorithme d'apprentissage. L'existence d'un tel algorithme permet alors de montrer que la classe est effectivement apprenable dans le modèle considéré. La plupart du temps, la complexité de l'algorithme est également évaluée, afin d'établir la praticabilité de l'apprentissage. Nous disposons alors d'une borne maximale sur le nombre d'exemples à tirer afin d'assurer l'apprentissage de tout concept de la classe. Cependant, cette borne n'est qu'une valeur calculée[9] pour le « pire des cas ». Mais, qu'en est-il de la taille moyenne de l'échantillon utile? Remarquons que cette question ne se pose pas que dans le cas de l'apprentissage PAC (et dérivés). En effet, il peut être légitime d'évaluer, en moyenne, le nombre d'étapes nécessaire avant qu'un algorithme d'inférence à la limite ne se stabilise sur la bonne hypothèse. Dès lors, un certain nombre de questions arrivent naturellement à l'esprit, telles que: « Y'a-t-il des exemples permettant d'apprendre plus vite? », « Quel est le nombre minimal d'exemples utiles? », « Certaines distributions sur les exemples favorisent-elle plus l'apprentissage que d'autres? », « Existe-t-il des distributions calculables pouvant remplacer dans certains cas la distribution universelle? » etc...

Ce type de questions n'est, malheureusement, que très peu étudié dans le cadre de l'apprentissage automatique. Il s'agit pourtant de questions fondamentales si l'on espère, un jour, implanter de vrais algorithmes qui apprennent. Il semble que ce sont des questions auxquelles il est difficile de répondre. Aussi, une idée serait de les aborder de manière empirique, c'est-à-dire en testant effectivement les algorithmes sur des jeux de données « réels ».

C'est pour tenter de répondre à ce type de questions « pratiques » que nous avons créé

9. Souvent grossièrement.

une plate-forme d'évaluation d'algorithmes d'apprentissage. La partie suivante a pour objet de présenter cette plate-forme. Notre idée est de fournir un outil capable de donner des éléments de réponses, d'ouvrir de nouvelles pistes, de générer des intuitions qui pourront, par la suite, être étudiées de manière théorique. Il s'agit donc d'un atelier d'expérimentations implantant les objets propres à l'apprentissage et offrant une totale liberté sur le paramétrage de ceux-ci afin de réfuter ou de renforcer une idée *a priori* relative à l'apprentissage.

Deuxième partie

Une Plate-forme Générique d'Expérimentations pour l'Inférence

Chapitre 4

<PEpIn> : Principes Généraux

<PEpIn> (**P**late-forme d'**E**xpérimentations pour l'**In**férence) est un ensemble d'éléments génériques permettant de créer ses propres applications d'expérimentation dans le cadre de l'inférence inductive. Il s'agit d'un ensemble de classes C++ implantant la plupart des « objets » utilisés dans le cadre de l'apprentissage automatique. Au moyen de cette librairie, il devient possible de définir son propre environnement de travail (domaine d'apprentissage, protocole à utiliser, etc...) afin de concevoir un « atelier d'expérimentations » adapté à ses besoins. Au moyen de <PEpIn>, nous avons ainsi implanté une application permettant de lancer des expériences relatives à l'inférence grammaticale (voir chapitre 6, page 161). Ce premier chapitre expose les choix que nous avons adoptés afin de concevoir une telle plate-forme. D'abord, le cahier des charges listera l'ensemble des attentes d'un tel outil. Ensuite, nous présenterons <PEpIn> au niveau conceptuel et verrons la manière dont nous avons répondu au cahier des charges. Enfin, dans la dernière section, nous nous intéresserons plus particulièrement au noyau du système.

4.1 Cahier des Charges

La première difficulté pour établir le cahier des charges de <PEpIn> vient du fait qu'il ne s'agit nullement d'une application « prête à l'emploi » mais plutôt d'une librairie générique permettant de créer des applications pour l'expérimentation ayant trait à l'inférence inductive. En d'autres termes, <PEpIn> est une bibliothèque définissant l'ensemble des éléments de la théorie de l'apprentissage tels que ceux définis au chapitre 1 (page 25) (*e.g.* domaine, concepts, exemples labellés, etc...). La création d'une telle bibliothèque « abstraite » est possible puisque quel que soit le contexte[1] considéré, les dépendances entre ces éléments sont toujours les mêmes, seules diffèrent les entités qu'ils représentent et les manipulations sous-jacentes possibles. Par exemple, dans le contexte des langages formels, un item est un mot alors que dans le contexte des formules booléennes, un item est un n-uplet de booléens. Quoiqu'il en soit, dans les deux contextes, un item est un élément du domaine qui permet de construire un exemple labellé. La conception de la plate-forme réside alors essentiellement dans l'analyse et la modélisation des rapports existants entre les intervenants de la théorie de l'apprentissage.

1. Par *contexte* on entend le cadre théorique auquel appartient l'ensemble des objets étudiés. Les langages formels et les formules booléennes sont deux exemples de contexte.

4.1.1 Définitions préliminaires

Afin de formaliser au maximum notre discours, il convient de définir quelques peu le vocabulaire que nous utiliserons par la suite. Certaines définitions peuvent sembler superflues, voire artificielles ; cependant elles permettent de bien distinguer entre les objets « abstraits » relatifs à la terminologie de l'apprentissage et les objets « tangibles » d'un contexte particulier.

Définition 4.1.1 *Un* rôle *est une fonction qu'occupe un objet d'un contexte particulier dans le cadre de l'apprentissage théorique. Les différents rôles possibles sont : le domaine, la classe de concepts, la classe d'hypothèses et l'ensemble des représentations. On note chacun de ces rôles respectivement* $\hat{X}$, $\hat{F}$, $\hat{H}$ *et* $\hat{R}$.

Définition 4.1.2 *Un* OTA *(Objet de la Théorie de l'Apprentissage) est un élément constitutif de la terminologie de l'apprentissage. Les OTAs qui seront considérés dans la suite sont : les domaines, les items, les concepts, les classes de concepts, les classes d'hypothèses, les ensembles de représentations, les exemples, les échantillons, les distributions de probabilité, les oracles, les algorithmes d'apprentissage et les modèles d'apprentissage.*

Remarquons que tout OTA ne définit pas pour autant un rôle ; en effet, certains OTAs n'ont pas de correspondance dans les contextes étudiés. Ceux-ci se définissent alors soit à partir d'autres OTAs (*e.g.* les items qui sont des éléments du domaine), soit *de nihilo* c'est-à-dire en tant qu'objet propre du monde de l'apprentissage, sans dépendance particulière avec le contexte étudié (*e.g.* les modèles d'apprentissage).

Définition 4.1.3 *L'ensemble des OTAs forme l'*univers abstrait d'apprentissage. *L'univers abstrait d'apprentissage est noté* $U =< \hat{X}, \hat{F}, \hat{H}, \hat{R} >$ *avec* $\hat{X}, \hat{F}, \hat{H}$ *et* $\hat{R}$ *représentant les différents rôles précédemment évoqués (voir définition 4.1.1).*

Définition 4.1.4 *Un* univers d'apprentissage relatif au contexte $\mathcal{D}$ *est un univers d'apprentissage dans lequel chaque rôle est tenu par un élément particulier du contexte* $\mathcal{D}$. *On note un tel univers* $U(\mathcal{D}) =< \hat{X} = X, \hat{F} = F, \hat{H} = H, \hat{R} = R >$ *avec* X, F, H, R *des objets du contexte* $\mathcal{D}$.

Au vue de ces définitions, nous pouvons reformuler la vision que l'on a de <PEpIn> et des applications qu'elle permet de créer.

4.1.2 Reformulation du problème

L'objectif d'une plate-forme comme <PEpIn> est, par conséquent, la modélisation et l'implantation de l'univers abstrait d'apprentissage. D'une part, cette bibliothèque se doit de définir les rapports qui existent entre les différents OTAs, et ce, quels que soient les objets futurs auxquels on attribuera un rôle. D'autre part, elle doit fournir un *run-time* permettant d'implanter le processus d'apprentissage lui-même. En résumé, disposer d'une plate-forme générique comme <PEpIn>, c'est disposer de l'implantation de $U =< \hat{X}, \hat{F}, \hat{H}, \hat{R} >$.

L'objectif d'une telle bibliothèque est donc d'être associée à un contexte particulier afin de construire des applications relatives à l'inférence inductive dans ce (ou ces) contextes. Pour ce faire, il faut avant tout supposer que le contexte considéré est lui-même modélisé et implanté dans une seconde librairie. La création de l'application consiste alors principalement

en l'attribution des rôles à certaines entités du contexte. Un petit travail de programmation supplémentaire sera également nécessaire afin, entre autre, d'écrire le programme principal gérant l'application. La figure 4.1 illustre la création d'une telle application pour un contexte particulier $\mathcal{D}$.

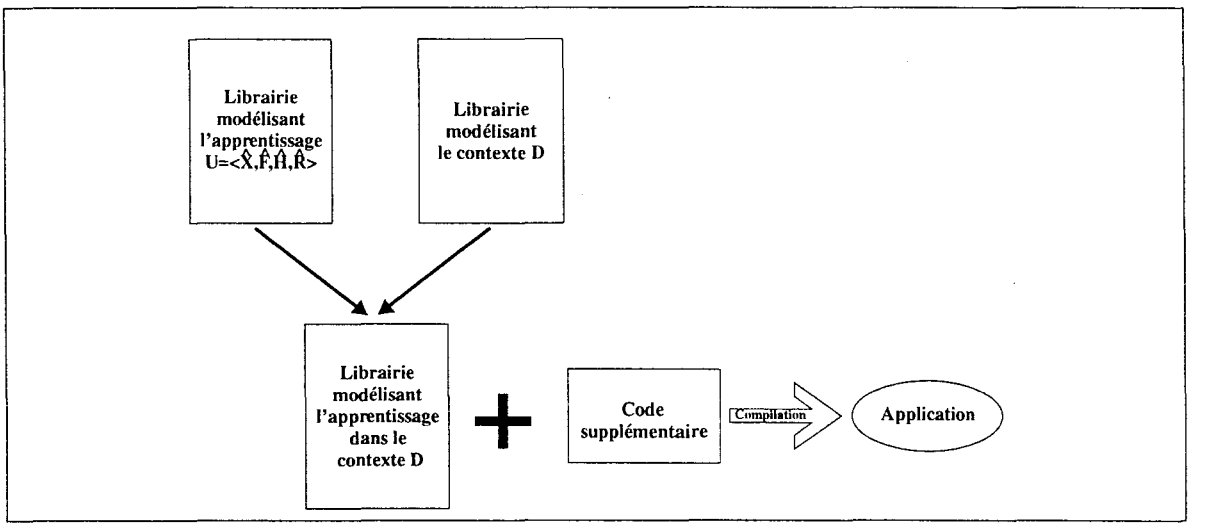


FIG. 4.1 - Création d'une application particulière à partir de <PEpIn> et d'une librairie modélisant un contexte.

La mise en relation de la librairie <PEpIn> avec la librairie modélisant un contexte particulier $\mathcal{D}$ permet de créer implicitement une troisième librairie en attribuant les rôles définis dans $U =< \hat{X}, \hat{F}, \hat{H}, \hat{R} >$ à certains objets X, F, H, R du contexte $\mathcal{D}$. La librairie ainsi créée modélise finalement l'univers d'apprentissage $U(\mathcal{D}) =< \hat{X} = X, \hat{F} = F, \hat{H} = H, \hat{R} = R >$. La figure 4.2 (page suivante) illustre les créations de deux univers d'apprentissage particuliers, l'un relatif au contexte des langages réguliers et l'autre à celui des formules booléennes. Le principe de **généricité** demandé à la plate-forme est représenté au moyen de « trous » comblés par des « briques » dépendantes du contexte considéré.

Jusqu'à présent, nous n'avons fait qu'énoncer le principe de généricité attendue de la plate-forme. Étant donnée cette caractéristique, toutes les fonctionnalités offertes par <PEpIn> se retrouveront dans les applications créées à partir de cette librairie. Aussi, il convient de bien définir les attentes de ces applications afin de les intégrer au mieux dans la plate-forme elle-même.

Dans la suite, nous établissons le cahier des charges des applications générées à partir de <PEpIn>. Nous allons considérer une telle application, construite pour un contexte $\mathcal{D}$ quelconque.

4.1.3 Étude d'une application particulière

Nous considérons un contexte quelconque $\mathcal{D}$ et l'application construite pour l'univers d'apprentissage $U(\mathcal{D}) =< \hat{X} = X, \hat{F} = F, \hat{H} = H, \hat{R} = R >$.

L'objectif premier d'une telle application est d'apporter des éléments de réponse, de manière empirique, à des questions relatives à l'apprentissage de la classe de concepts F. En premier lieu, elle doit permettre l'évaluation d'algorithmes d'apprentissage pour la classe F. Cependant, restreindre l'étude de l'apprentissage d'une classe de concepts à la seule analyse

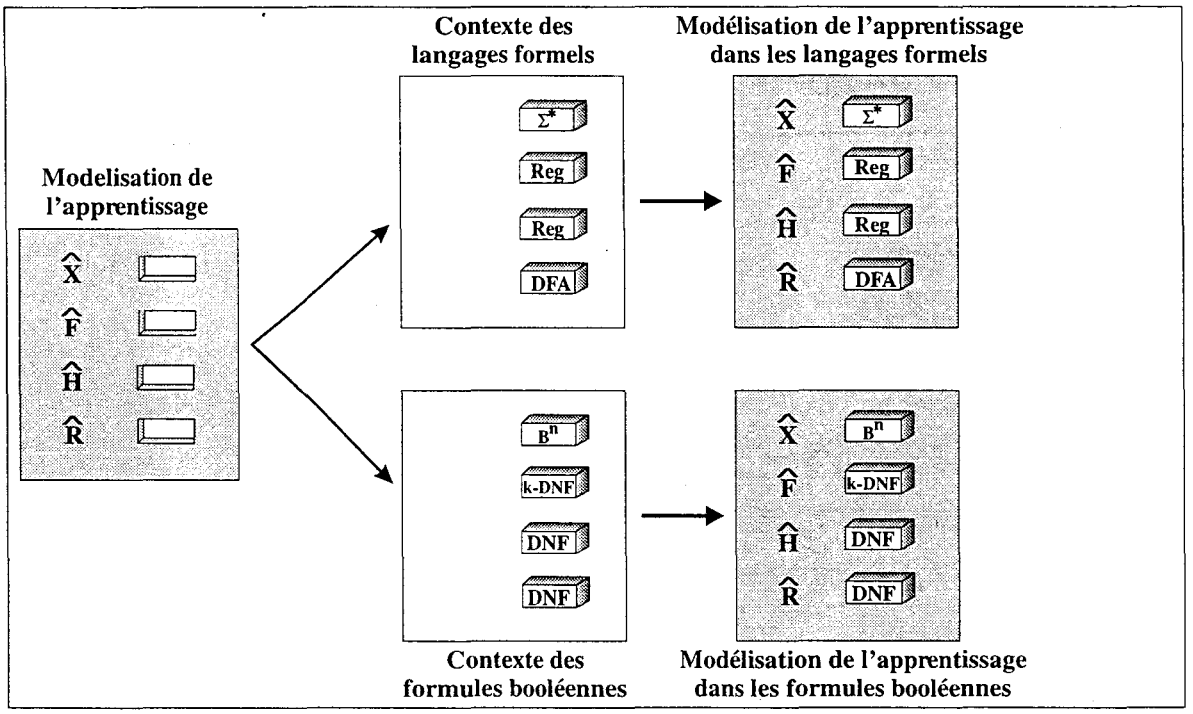


FIG. 4.2 - Distribution des rôles pour l'apprentissage à des objets de deux univers.

des algorithmes d'inférence est par trop limité et ne suffit pas à exprimer les attentes d'un tel outil. En effet, un algorithme d'apprentissage n'a d'existence que dans le cadre d'un modèle d'apprentissage. Ce modèle d'apprentissage fixe, d'une part le ou les critères de succès de l'apprentissage et d'autre part les attentes de la part de l'algorithme suivant la présentation des exemples. Aussi, étudier le comportement d'un tel algorithme s'accompagne de fait d'une étude de l'environnement dans lequel celui-ci est plongé (distribution des exemples, connaissance d'informations *a priori*, ...). Finalement, nous attendons d'une telle application qu'elle permette l'étude de l'ensemble de l'environnement d'apprentissage, c'est-à-dire non seulement de l'algorithme d'inférence (pertinence, efficacité en moyenne) mais aussi de l'influence de la distribution des exemples ou bien du critère de succès.

Il s'agit donc bien d'un *atelier d'expérimentations pour l'apprentissage*. Il doit permettre au chercheur de se forger des idées, de confirmer ou d'infirmer une intuition faite *a priori*. Le principe est simplement de donner à l'expérimentateur la possibilité de faire un grand nombre d'expériences d'apprentissage, en jouant sur de multiples paramètres. Disposant d'un grand nombre de données, il lui sera alors possible de les analyser afin d'établir des résultats (en général d'ordre statistique) répondant ainsi **empiriquement** à ses questions.

Nous listons ci-après, un certain nombre de questions types qui pourraient requérir l'utilisation d'une telle application afin d'y répondre.

- « Quel est le nombre moyen d'exemples permettant d'atteindre un succès d'apprentissage? »: ce nombre moyen d'exemples utiles est évidemment fonction des différents paramètres possibles (distribution de probabilité, confiance, précision). Dans le paradigme de Gold, cette question se reformule en « quel est le nombre moyen d'étapes pour que l'apprentissage exact soit atteint? ».

- « Existe-t-il des distributions de probabilité particulières favorisant l'apprentissage de la classe de concepts étudiée? »,
- « Existe-t-il un échantillon particulier indispensable à l'apprentissage? »,
- « Tel algorithme d'apprentissage est-il, dans certains cas d'utilisation, plus efficace que tel autre? ».

Remarquons que toutes ces questions ne sont pas indépendantes les unes des autres; les réponses aux unes impliquant certaines réponses aux autres.

Répondre par l'expérience à ce type de questions n'est pas sans intérêt pour le théoricien. En effet, nous espérons que cela pourra ouvrir quelques pistes afin par exemple de:

- Montrer que certaines classes de concepts sont « PAC-apprenables » sous des distributions de probabilité particulières,
- Améliorer les performances « en moyenne » de certains algorithmes d'apprentissage,
- Définir des échantillons caractéristiques pour un algorithme d'apprentissage donné,
- Trouver des heuristiques permettant de remplir les conditions d'apprentissage pour une classe non apprenable.

Les objectifs de la plate-forme générique sont évidemment les mêmes que ceux des applications créés à partir de celle-ci. <PEpIn> doit donc intégrer l'ensemble des outils permettant d'atteindre ces buts, définis à un niveau « méta », c'est-à-dire sans considérer un contexte en particulier.

Nous énonçons maintenant quelques qualités que l'on est en droit d'attendre de <PEpIn>, afin d'en faire une véritable plate-forme générique d'apprentissage pour le chercheur.

4.1.4 Qualités et caractéristiques attendues

Rapport à la théorie de l'apprentissage

Cette plate-forme a pour but d'être utilisée dans le cadre de recherches sur l'apprentissage automatique. Son intérêt principal est d'aider le chercheur dans la découverte de nouvelles propriétés d'apprentissage, dans la formulation de nouveaux modèles ou bien encore dans l'élaboration d'heuristiques. Aussi, étant donné qu'elle s'adresse à une communauté ayant son propre vocabulaire, ses propres matériaux, il est important que chacun reconnaisse ceux-ci dans l'implantation qui en est faite. Les objets de la théorie de l'apprentissage doivent donc pouvoir être identifiés facilement dans l'implantation. Remarquons que par nature, certains OTAs ne peuvent être implantés ou demandent à être adaptés pour une représentation sur machine. Il convient cependant de pouvoir retrouver une « trace » de chacun d'eux dans la librairie <PEpIn> afin de faciliter leur instanciation dans un contexte particulier.

Intégration des modèles

Il semble séduisant de pouvoir implanter les modèles classiques de la théorie de l'apprentissage, et en particulier les modèles de Gold et de Valiant. Cependant, espérer une telle prouesse est de l'ordre de l'utopie. En effet, en ce qui concerne le modèle de Gold, celui-ci

définissant l'acte d'apprentissage comme un processus infini, son implantation *stricto sensu* est impossible. En ce qui concerne le modèle PAC, c'est la condition d'apprenabilité sous toutes les distributions de probabilité possibles qui empêche d'implanter ce modèle tel quel. Nous proposerons par la suite une technique permettant de « simuler » l'un et l'autre de ces deux modèles (voir section 4.3, page 133).

Évolutivité

La qualité de généricité demandée à la plate-forme concerne l'adaptation possible de celle-ci à plusieurs contextes différents. La qualité d'évolutivité consiste à permettre l'implantation progressive de nouveaux algorithmes d'apprentissage sans avoir à modifier quoi que ce soit au niveau de la plate-forme elle-même. Cela suppose donc une certaine indépendance entre le noyau de la plate-forme et les éléments dépendant directement du contexte considéré. La propriété d'évolutivité doit être assurée pour d'autres types d'objets comme par exemple les distributions de probabilité. Remarquons que celles-ci font partie intégrante de <PEpIn> (en tant qu'OTA). Il doit cependant être possible de définir d'autres types de distributions de probabilité que celles fournis par défaut. Notons que le but n'est pas de créer un langage de conception d'algorithmes d'apprentissage ou de définition de distributions de probabilité. De tels nouveaux éléments devront, par conséquent, être programmés au même niveau que l'application elle-même. Il s'agit donc bien d'intégrer l'idée que <PEpIn> et ses applications dérivées ont la possibilité d'évoluer sans modifier l'existant. Il n'est donc pas possible[2] de créer « dynamiquement » des algorithmes d'apprentissage ou de distributions de probabilité au cours de l'utilisation d'une application.

Automatisme et performance

Les deux dernières contraintes que l'on impose sont dues essentiellement à l'utilisation expérimentale qui va être faite des applications. L'exploitation des résultats d'un apprentissage sera basée principalement sur des analyses statistiques de ceux-ci. Cela implique donc d'une part de disposer d'une masse importante de données résultant d'un grand nombre de versions d'une même expérience et d'autre part de pouvoir travailler sur des objets significatifs. Les applications devront donc permettre le lancement automatique d'expériences de manière à disposer des données attendues sans pour autant monopoliser un expérimentateur humain. De plus, une fois paramétrée, une expérience doit éventuellement pouvoir s'exécuter sans intervention humaine. La machine devra donc jouer à la fois le rôle de l'enseignant et celui de l'apprenant. Enfin, il n'est pas envisageable de restreindre les études à des « cas d'école » ; c'est pourquoi il sera important de pouvoir manipuler des données (exemples et concepts) de taille importante, sans pour autant pénaliser les performances du système.

4.1.5 Synthèse

La plate-forme d'apprentissage est donc une librairie de classes génériques C++, modélisant l'ensemble des éléments de la théorie de l'apprentissage automatique. Ces classes sont destinées à être utilisées conjointement avec une librairie implantant un contexte particulier (univers des langages réguliers, univers des formules booléennes ...) afin de décrire une instance particulière de l'univers d'apprentissage. Une application d'« atelier d'expérimentations

2. Du moins dans la version actuelle de <PEpIn>.

sur l'apprentissage » dans le contexte considéré peut alors être écrite de manière très simple (en général un programme principal gérant le déroulement de l'application).

Dans la suite, nous allons considérer l'ensemble des objectifs et des caractéristiques évoqués dans ce cahier des charges afin de présenter l'analyse de notre plate-forme. Remarquons que seule la partie concernant la plate-forme d'apprentissage sera décrite. Le chapitre 6 (page 161) sera l'occasion d'exposer l'association de la plate-forme à un contexte particulier : celui des langages réguliers.

4.2 Analyse du Projet

L'analyse qui est faite ici est une analyse orientée objet ; elle est basée sur O.M.T. (Object Modeling Technique), la méthode de conception orientée objet due à Graddy Booch [Boo94].

Notre propos n'est pas ici de redonner les principes et les techniques de la conception orientée objet. Le lecteur non familier avec ce domaine pourra consulter [AD91], [Boo94] ou [Mul97]. Rappelons simplement quelques définitions de base propres à la programmation par objets (ces définitions sont tirées de [AD91]).

4.2.1 Rappel de définitions relatives à la programmation objet

- Une *classe* est une entité génératrice d'une famille d'objets, ses *instances*, pour lesquels elle définit la structure (au moyen de ses *attributs*) et le comportement (au moyen de ses *méthodes*).
- Une *instance* est un *objet*. C'est un élément d'une classe dont l'état est caractérisé par les valeurs de ses attributs.
- L'*héritage* est le mécanisme permettant le partage d'attributs et d'opérations entre les classes. Il est basé sur des relations hiérarchiques. Une *sous-classe* hérite de toutes les propriétés de sa classe mère et y ajoute ses propres propriétés.

Remarque 4.2.1 Afin d'éviter toute confusion entre le vocabulaire utilisé dans la terminologie de la programmation objet et celui utilisé dans les autres domaines, nous noterons les termes relatifs à la programmation objet en italique. Par exemple « *classe A* » représente la classe A au sens programmation objet, alors que « classe A » représente la classe A au sens mathématique du terme.

4.2.2 Première analyse

À la figure 4.3 (page suivante), nous proposons un premier modèle objet de <PEpIn>. Celui-ci n'est qu'une représentation brute du formalisme de l'apprentissage. Il repose essentiellement sur les définitions de base des différents OTAs et de certaines relations qui existent entre eux (voir section 1.1.2, page 26).

Ce premier modèle demande à être commenté. Remarquons d'abord que nous n'y avons fait figurer ni le modèle d'apprentissage, ni l'algorithme d'apprentissage. En effet, une étude approfondie de ces deux objets est indispensable afin de les intégrer ; une telle analyse sera faite par la suite.

Ensuite, il est clair que certaines relations, pourtant bien définies entre les objets, n'apparaissent pas sur le graphe ; on peut citer par exemple les relations ensemblistes entre les

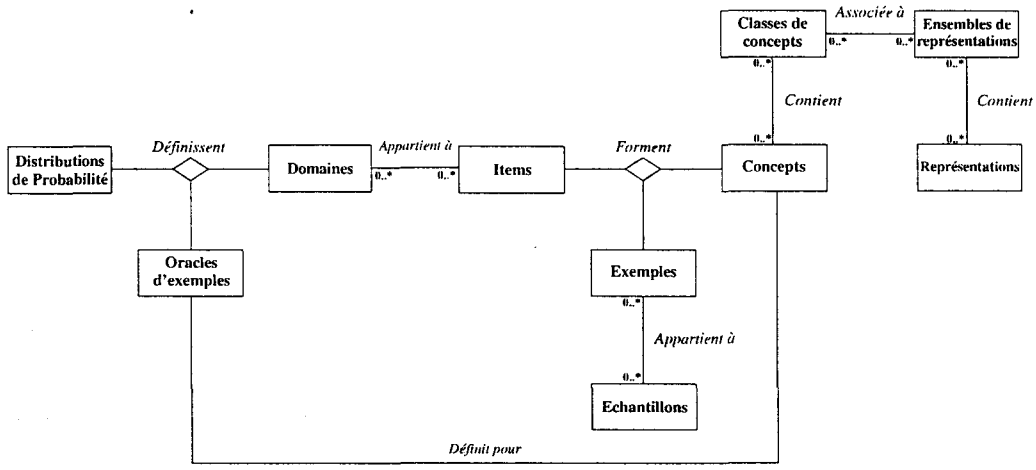


FIG. 4.3 - Un premier modèle objet de <PEpIn>.

concepts et le domaine. En fait, ces relations ont été volontairement supprimées car elles n'ont que peut d'intérêt dans le problème qui nous occupe ; en effet, nous faisons l'hypothèse *a priori* que le contexte considéré définit correctement ses objets et donc, par exemple, qu'un concept est bien un sous-ensemble d'un domaine.

Constatons encore que ce graphe ne présente que des associations entre les *classes* mais aucune relation d'héritage entre elles. C'est un fait assez rarissime dans une analyse objet pour le souligner mais des relations d'héritage seront introduites par la suite notamment lorsque le contexte sera intégré à l'environnement d'apprentissage.

Enfin, remarquons que l'OTA « classe d'hypothèses » n'est pas modélisé ici. En effet, une classe d'hypothèses est une classe de concepts à part entière. Il s'agit donc simplement d'une *instance* de la *classe* `Classes_de_concepts` au même titre que la classe des concepts cibles.

L'analyse directe qui vient d'être faite n'est, d'évidence, pas suffisante. Elle a l'avantage d'être simple et de présenter un premier niveau de relations entre nos objets mais demande à être raffinée.

4.2.3 Raffinement de l'analyse

`Classes_de_concepts` ***versus*** `Concepts`

La question que nous nous posons ici est « Est-il utile de disposer à la fois d'une *classe* `Classes_de_concepts` et d'une *classe* `Concepts` ? ». Par définition, la *classe* `Concepts` représente la famille des *objets* concepts. Une telle *classe* étant définie, il est alors possible de construire certains concepts particuliers. Dans le cas de l'apprentissage, nous n'avons à aucun moment besoin de décider de l'appartenance d'un concept à une classe particulière. Seules la construction et l'utilisation de concepts particuliers connus comme appartenant à une classe sont utiles dans le problème de l'apprentissage. La *classe* `Classes_de_concepts` est par conséquent superflue. Finalement, c'est la *classe* `Concepts` qui joue le rôle de l'OTA « classe de concepts » alors qu'une *instance* de cette *classe* tient lieu de concept.

Notons que le même raisonnement peut être tenu en ce qui concerne les *classes* `Ensembles_de_representation` et `Representations`, ce qui conduit à ne plus considérer la première de ces deux *classes*.

`Domaines et Items`

Naturellement, nous pouvons nous poser la même question au sujet de la *classe* `Domaines` par rapport à la *classe* `Items` ; c'est-à-dire ne pourrait-on pas « fusionner » ces deux éléments en un seul, la *classe* `Items` jouant alors le rôle des domaines, un item étant dans ce cas simplement une *instance* de cette *classe*. Ici, nous choisissons pourtant de conserver ces deux entités telles quelles. En effet, la *classe* `Domaines` dispose d'une relation d'association avec les *classes* `Distributions_de_probabilite` et `Oracles` qui n'est pas présente au niveau de la *classe* `Items`. En fait, il doit être possible de construire effectivement des *instances* du type `Domaines` afin de pouvoir munir celles-ci d'une distribution de probabilité.

`Representations` ***versus*** `Concepts`

Tel que défini à la section 1.1.2 (page 26), un ensemble de représentations doit permettre de coder tout élément de la classe de concepts. En fait, l'ensemble des représentations est introduit dans la théorie de l'apprentissage afin que les algorithmes d'apprentissage puissent manipuler de manière effective les concepts. Ainsi, un concept est un objet mathématique alors qu'une représentation de concept est un objet informatique utilisable au sein d'une machine. Il est évident qu'il est impossible de construire effectivement une instance de la *classe* `Concepts` puisqu'il ne s'agirait alors que d'une représentation, d'un codage de celle-ci et les *classes* `Concepts` et `Representations` joueraient dans ce cas le même rôle. Aussi, nous décidons de supprimer l'une de ces deux *classes*. En toute logique, c'est la *classe* `Concepts` qui devrait disparaître. Cependant, afin d'assurer une lisibilité maximale, c'est cette *classe* que nous conservons ; plus précisément, nous conservons la *classe* `Representations` mais nous la renommons `Concepts`.

`Distributions_de_probabilite`

Nous définissons la *classe* `Distributions_de_probabilite` en tant que *classe abstraite*. Celle-ci est donc destinée à être dérivée afin d'implanter effectivement la distribution de probabilité désirée. Par exemple, on pourra créer une *classe* fille `Distributions_uniformes` qui implante les distributions de probabilité uniformes entre deux bornes. Une *instance* d'une telle classe permettra, par la suite, de générer aléatoirement des entiers selon la mesure de probabilité envisagée.

Prise en considération de l'OTA « Modèle d'apprentissage »

À présent, portons notre attention sur l'élément clé de notre étude : le modèle d'apprentissage. Un modèle d'apprentissage est un objet mathématique fixant les règles d'apprenabilité d'une classe de concepts. Un tel objet semble, par nature, très difficile à implanter, étant donné son côté très « abstrait ». Cependant, les définitions des différents modèles d'apprentissage convergent sur un point : il doit exister un algorithme d'apprentissage pour la classe de concepts dans le modèle. En fait, un modèle d'apprentissage fixe des règles auxquelles les algorithmes d'apprentissage doivent répondre. Deux types de règles sont envisagés :

- **Des règles *comportementales*** qui définissent la façon dont l'algorithme doit effectuer certaines actions (tirage des exemples, arrêt du processus etc...). Par exemple, dans le cas du modèle de Gold, l'algorithme d'apprentissage doit tourner indéfiniment. Dans

le modèle PAC, les exemples doivent être tirés selon une distribution de probabilité quelconque mais fixée au début du processus d'apprentissage.

- **Des règles d'*évaluation*** établissant les contraintes attendues sur les résultats ; il s'agit essentiellement des critères de succès de l'apprentissage.

Notre ambition n'est évidemment pas de pouvoir déterminer automatiquement si un algorithme donné est ou non un algorithme d'apprentissage pour une classe de concepts sous un modèle fixé[3]. En fait, il s'agit de pouvoir modéliser les différentes règles relatives à un modèle puis d'étudier l'algorithme d'apprentissage dirigé par ces règles. Aussi, nous introduisons une *classe* `Modeles_d_apprentissage` en tant qu'ensemble de telles règles ; une *instance* de cette *classe* sera donc un ensemble de règles fixant le comportement du processus d'apprentissage ainsi que celles déterminant si l'apprentissage a ou non abouti. Il convient de remarquer qu'une *instance* de cette *classe* n'est pas à proprement parlé un modèle d'apprentissage complet mais plutôt un cas particulier de celui-ci.

Nous verrons par la suite (voir section 4.3, page suivante) comment simplifier cette classe qui reste jusqu'à présent très abstraite.

Prise en considération de l'OTA « Apprenant »

Un « apprenant » est un algorithme particulier construisant un concept à partir d'exemples labellés. Suivant les principes de la conception orientée objet, nous intégrerons l'algorithme d'apprentissage en tant que *méthode* de construction d'objets du type `Concepts`. Plus précisément, nous créons une *classe* `Concepts_Apprenables` qui dérive de la *classe* `Concepts` ; une *instance* de cette *classe* est simplement un concept construit au moyen de l'algorithme d'apprentissage considéré. Cette *classe* est associée à la *classe* `Modele_d_apprentissage` dans la mesure où d'une part un algorithme d'apprentissage doit se conformer aux règles comportementales définies dans un tel modèle et d'autre part un concept ainsi construit doit pouvoir être évalué par le modèle.

Remarquons que cette façon de concevoir les concepts apprenables inclut *de facto* la notion de classe d'hypothèses. En effet, une classe d'hypothèses est, nous l'avons vu, une classe de concepts particulière. Cette manière de définir la hiérarchie des *classes* peut choquer en ce sens que dans la théorie de l'apprentissage la classe des hypothèses est en générale au moins égale à celle des concepts. Il convient de comprendre que cette modélisation, caractéristique de la conception objet, n'interdit pas d'expérimenter l'apprentissage faible[4] ; en effet, pour ce faire, il suffira simplement de restreindre le choix des concepts cibles à ceux appartenant effectivement à la classe de concepts étudiée. La *classe* `Concepts` modélise, quant à elle, la représentation d'une classe de concepts plus importante incluant celle des hypothèses.

La *classe* `Sessions_d_apprentissage`

Nous introduisons une nouvelle *classe* appelée `Sessions_d_apprentissage`. Comme son nom l'indique, une *instance* de cette *classe* représente le processus d'apprentissage lui-même, après avoir fixé un concept cible, un algorithme d'apprentissage et un modèle d'apprentissage. Une session d'apprentissage permet donc d'étudier l'algorithme d'inférence pour un concept particulier de la classe de concepts considérée *a priori* comme apprenable dans le modèle

3. La plupart du temps, la tâche est évidemment irréalisable.
4. C'est-à-dire dans lequel la classe d'hypothèses est plus grande que la classe de concepts.

fixé. Remarquons encore que c'est au travers de tels objets que l'expérimentateur aura la possibilité de récupérer l'ensemble des résultats concernant un cas d'apprentissage. La section 4.3 détaillera cette classe.

4.2.4 Fonctionnalités attendues des *objets*

Afin d'assurer la communication entre les différents objets que l'on vient d'énumérer, nous présentons les fonctionnalités que chacun doit obligatoirement inclure. Au niveau de la plate-forme générique, il s'agit essentiellement d'opérations abstraites, c'est-à-dire non utilisables en l'absence de contexte. Ces fonctions sont par conséquent sémantiquement définies mais dépendantes du contexte dans lequel la plate-forme sera intégrée.

Gestion des items

Un domaine est un ensemble dénombrable d'items. Les items sont supposés être comparables selon un ordre dépendant de leur nature; nous supposons donc l'existence d'un opérateur de comparaison `operateur<` entre deux items. Afin de permettre la modélisation d'une distribution de probabilité sur un domaine, il doit être possible de générer le i^e item d'un domaine donné (la valeur de i étant le rang auquel on trouve l'item selon l'ordre défini sur ceux-ci). La *méthode* `GetItem(i)` de la *classe* `Domaines` a pour fonction de générer le i^e item du domaine.

Utilisation des exemples et des échantillons

Un exemple est un item labellé pour un concept particulier. On demande par conséquent à la *classe* `Concepts` d'implanter la *méthode* `contient(item)` donnant le label de l'item considéré pour le concept sur lequel elle est appliquée.

Le but d'un oracle est de distribuer des exemples labellés pour un concept donné. C'est le rôle de la *méthode* `GetExemples` intégrée à la *classe* `Oracles`.

Enfin, la *classe* `Concepts_apprenables` doit permettre de construire de nouveaux concepts à partir d'exemples ou d'échantillons, selon un algorithme d'apprentissage; ceci est fait au moyen des deux *méthodes* `apprend(exemple)` et `apprend(echantillon)`. Il convient également de pouvoir comparer des concepts entre eux afin, par exemple, de tester le succès d'un apprentissage exact. La *classe* `Concepts` doit par conséquent fournir un tel opérateur de comparaison. Remarquons que nous plaçons cet opérateur au niveau de la *classe* `Concepts` et non simplement au niveau de la *classe* `Concepts_apprenables` afin de permettre la comparaison entre un concept cible et une hypothèse.

4.2.5 Nouveau modèle objet

Nous intégrons l'ensemble des considérations établies précédemment dans le nouveau modèle conceptuel de la plate-forme (voir figure 4.4page suivante).

4.3 Conception du Système

Cette section a pour but de présenter de quelle manière les éléments de la plate-forme vont s'activer afin de simuler le processus d'apprentissage. Nous nous proposons d'abord d'étudier plus précisément les règles à partir desquelles un modèle d'apprentissage se définit. Ensuite,

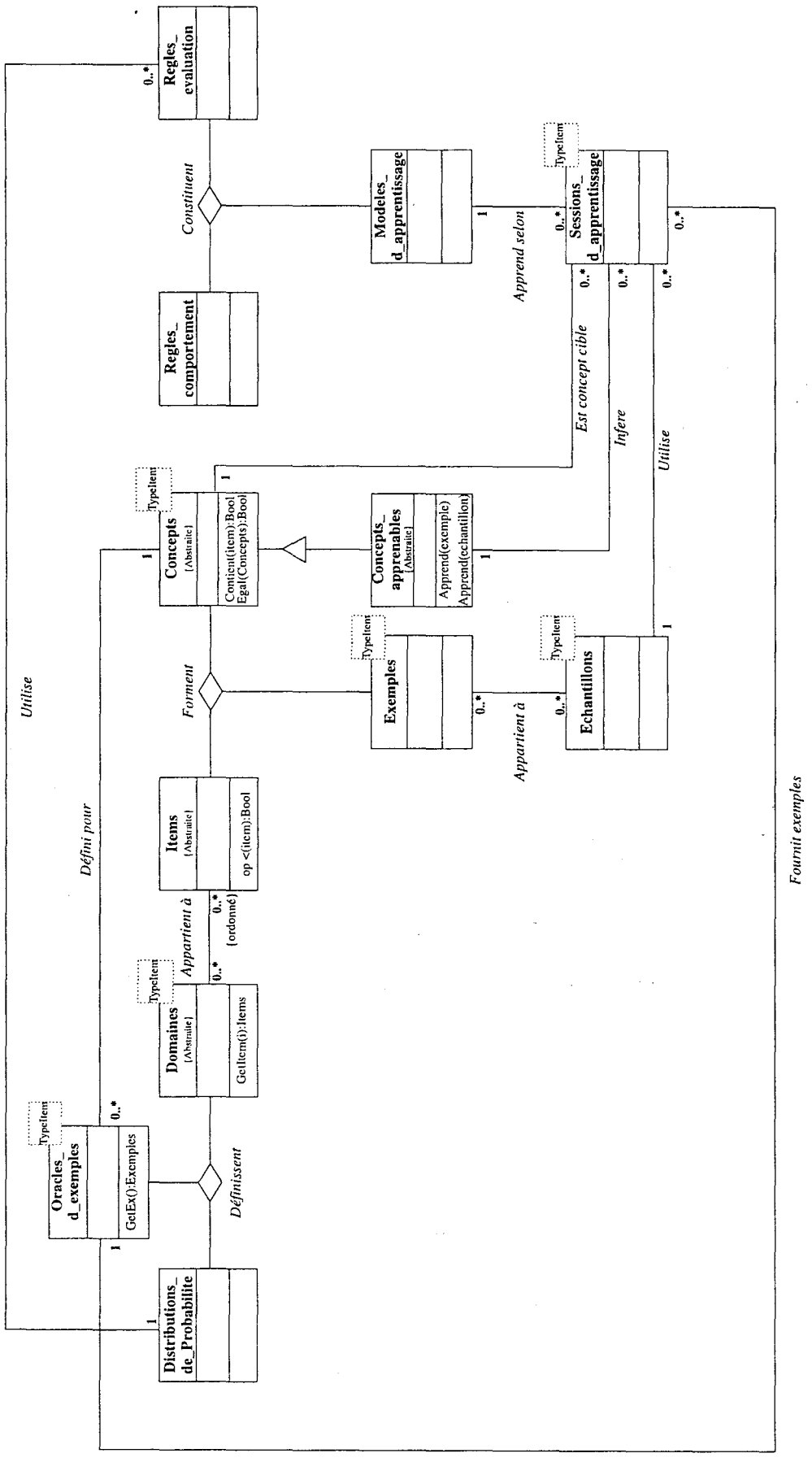


FIG. 4.4 - Nouveau modèle objet de <PEpIn>.

nous présentons le *run-time* de la plate-forme, c'est-à-dire le constituant actif d'une session d'apprentissage.

4.3.1 Nouvelle approche des modèles d'apprentissage

Dans ce qui précède, la *classe* `Modeles_d_apprentissage` restait relativement imprécise. Elle était constituée de règles auxquelles les algorithmes d'apprentissage devaient se conformer afin de rentrer dans le cadre du modèle d'apprentissage considéré. En fait, nous l'avons déjà indiqué, lors d'une expérience, il ne s'agit pas pour nous de pouvoir représenter exactement le modèle d'apprentissage étudié mais plutôt une instance de celui-ci. D'abord, un modèle d'apprentissage définit de quelle manière les exemples seront présentés à l'apprenant. Ceci est parfaitement faisable en instanciant un oracle sur le domaine au moyen d'une distribution de probabilité. Ensuite, il convient de pouvoir déterminer à quelle(s) condition(s) le processus d'apprentissage doit s'arrêter. Nous introduisons par conséquent une nouvelle *classe* appelée `Conditions_d_arret`. Les *instances* de ces *classes* représentent un opérateur décidant de l'arrêt du processus d'apprentissage suivant différents critères intégrés à l'*objet* `Conditions_d_arret`. Enfin, la *classe* `Criteres_de_succes` permet de définir la manière dont un concept hypothèse doit être évalué afin de déterminer si l'on est dans un cas de succès d'apprentissage ou non.

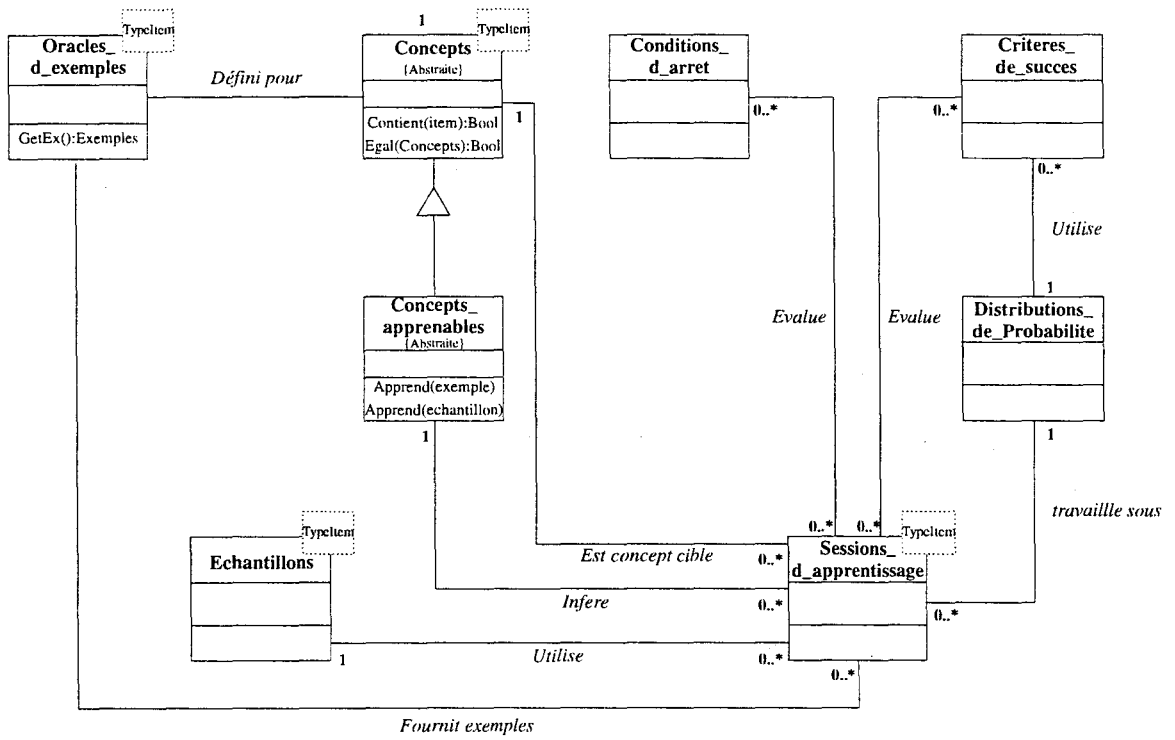


FIG. 4.5 - La *classe* `Sessions_d_apprentissage`.

Nous modifions une fois de plus notre modèle objet afin d'y faire paraître les différentes *classes* que nous venons d'introduire. De plus, afin de simplifier le travail d'implantation, nous supprimons la *classe* `Modeles_d_apprentissage`. Nous considérons maintenant que c'est une session d'apprentissage qui définit l'ensemble des éléments à prendre en compte dans un

processus particulier d'apprentissage. La figure 4.5 (page précédente) montre la partie modifiée du modèle objet.

4.3.2 Le run-time

Le run-time de la plate-forme est la méthode de la classe `Sessions_d_apprentissage` qui simule le processus d'apprentissage, après que l'ensemble des intervenants ait été défini.

Nous décidons, dans un premier temps, de séparer l'algorithme d'apprentissage en deux parties. Pour ce faire, nous nous appuyons sur le processus d'apprentissage dit par *données fixées* proposé par Gold (voir section 2.2.1, page 65). Il s'agit de considérer qu'un algorithme d'apprentissage est constitué d'une part d'un procédé d'acquisition d'exemples et d'autre part d'un algorithme de consistance. Pour nous, le rôle d'algorithme de consistance sera tenu par l'algorithme dit d'apprentissage offert par la *classe* `Concepts_apprenables`. L'algorithme d'apprentissage proprement dit sera, quant à lui, le run-time présenté dans un pseudo langage à la figure 4.6 (page suivante).

Dans certains cas, il est possible que l'algorithme de consistance disponible dans la *classe* `Concepts_apprenables` ne soit pas de type incrémental, c'est-à-dire qu'il ne permette pas de raffiner petit à petit le concept H au moyen de nouveaux exemples. Dans ce cas, le nouveau concept hypothèse construit à chaque étape ne peut pas être une simple modification de celui de l'étape précédente. Il faut alors utiliser un algorithme de consistance utilisant l'intégralité de l'échantillon déjà tiré. L'algorithme correspondant à ce type de run-time est présenté à la figure 4.7 (page suivante).

Remarquons que cet algorithme d'apprentissage incrémental est plus général que celui introduit par Gold puisque ici l'arrêt du processus est régi par les conditions d'arrêt définies dans la session d'apprentissage.

4.4 Évaluation des Possibilités Offertes par la Plate-Forme

Nous nous proposons dans cette section de reprendre quelques unes des attentes émises dans le cahier des charges (voir section 4.1, page 123) et d'évaluer si la proposition de <PEpIn> que nous venons de faire permet d'y répondre.

Remarquons d'abord que cette modélisation définit explicitement la plupart des objets de la théorie de l'apprentissage. À part les modèles d'apprentissage qui, nous l'avons vu, sont trop abstraits pour pouvoir être effectivement intégrés tels quels, tous les OTAs sont proprement décrits dans notre modèle objet.

L'évolutivité est rendue possible par le principe d'héritage offert par la programmation objet. Ainsi, qu'il s'agisse d'algorithmes d'apprentissage ou de distributions de probabilité, il sera toujours possible d'ajouter de nouvelles classes héritant des classes mères abstraites afin d'étendre et d'adapter <PEpIn> selon les besoins de chacun.

4.4.1 Possibilités offertes par le run-time

La conception du run-time que nous proposons permet d'assurer deux qualités importantes pour la plate-forme.

```
Soit S un échantillon vide
Soit H une hypothèse vide
Faire
    Tirer un exemple e
    Ajouter e à l'échantillon S
    H = H.apprend(e)
    Évaluer l'hypothèse H
Tant que pas arret()
```

FIG. 4.6 - Run-time « incrémental » de la plate-forme <PEpIn>.

```
Soit S un échantillon vide
Faire
    Tirer un exemple e
    Ajouter e à l'échantillon S
    H = apprend(S)
    Évaluer l'hypothèse H
Tant que pas arret()
```

FIG. 4.7 - Run-time « non-incrémental » de la plate-forme <PEpIn>.

Le run-time assure l'automatisme du système

Nous avons vu qu'il doit être possible de lancer automatiquement des sessions d'apprentissage, c'est-à-dire qu'une fois paramétré, le système doit permettre de simuler un apprentissage sans interaction de la part de l'expérimentateur. Le run-time que nous proposons donne cette possibilité dès lors que les exemples peuvent être fournis de manière automatique à l'algorithme d'apprentissage. C'est évidemment le rôle de l'oracle d'exemples que d'assurer cette tâche.

À propos des modèles d'apprentissage

Les modèles d'apprentissage de Gold et de Valiant peuvent être « simulés » sur notre système. En ce qui concerne le modèle de Gold, il suffit d'empêcher le système de s'arrêter (en donnant comme test d'arrêt une valeur toujours fausse par exemple). Cependant, telle quelle, une session d'apprentissage n'a que peu d'intérêt. Aussi, il est possible de demander au système de s'arrêter dès lors que le concept cible a été appris exactement. Pour une expérience donnée, nous disposons alors de la valeur de la première étape à partir de laquelle l'apprenant a inféré exactement le concept cible.

En ce qui concerne le modèle de Valiant, le problème diffère quelque peu. En effet, seules certaines instances de celui-ci peuvent être simulées, c'est-à-dire qu'une expérience donnée

correspond à un cas particulier de ce modèle (la distribution de probabilité étant fixée). Cependant, il est possible de fixer les critères de succès afin d'atteindre un apprentissage approximatif. Remarquons que le paramètre de confiance δ utilisé dans le modèle PAC n'est pas modélisable dans cette plate-forme. En effet, ce paramètre doit plutôt être vu comme une valeur statistique attendue sur un grand nombre d'expériences.

4.4.2 Généricité *versus* héritage

Nous avons choisi dans la conception de notre plate-forme de définir un grand nombre de classes de manière générique (représentées sur les graphes au moyen d'un rectangle en pointillé). Cette manière de concevoir <PEpIn> nous permettra par la suite d'intégrer directement des contextes de nature diverse dans l'environnement d'apprentissage.

Remarquons qu'il était possible de traiter le caractère de généricité de notre plate-forme simplement en définissant des classes abstraites que l'on aurait dérivées dans les différents contextes. Cependant, cette manière de procéder aurait conduit à un problème bien connu de la programmation objet du à l'héritage multiple en cascade. De plus, étant donnée notre intention d'implanter notre plate-forme en C++ qui n'est pas un langage « tout objet » ce choix permet d'outrepasser quelques difficultés sur les contraintes de types à la compilation et ce sans compliquer inutilement le système.

4.4.3 Exemple d'intégration de contexte

Afin de fixer les idées sur l'utilisation de la plate-forme, nous donnons ici l'exemple de l'intégration d'un contexte relatif aux formules booléennes. Nous instancions l'ensemble des classes génériques dans ce contexte afin de disposer d'une nouvelle librairie modélisant l'apprentissage des formules booléennes.

La figure 4.8 (page suivante) illustre l'intégration de ce contexte dans l'environnement d'apprentissage. Les parties grisées représentent les *classes* provenant uniquement de la modélisation (simplifiée) du contexte des formules booléennes.

Dans cet exemple, les concepts sont représentés au moyen des formules DNF. Les items sont des vecteurs de booléens et les domaines possibles sont les différents $\mathbb{B}^n$ (avec n fixé lors d'une expérience particulière). Remarquons qu'ici les classes génériques voient leur paramètre générique `TypeItem` instancié par le type `It_VBool` qui n'est autre que l'item correspondant aux vecteurs booléens.

Un autre cas d'intégration de contexte (celui des langages réguliers) sera étudié en détail au chapitre 6 (page 161).

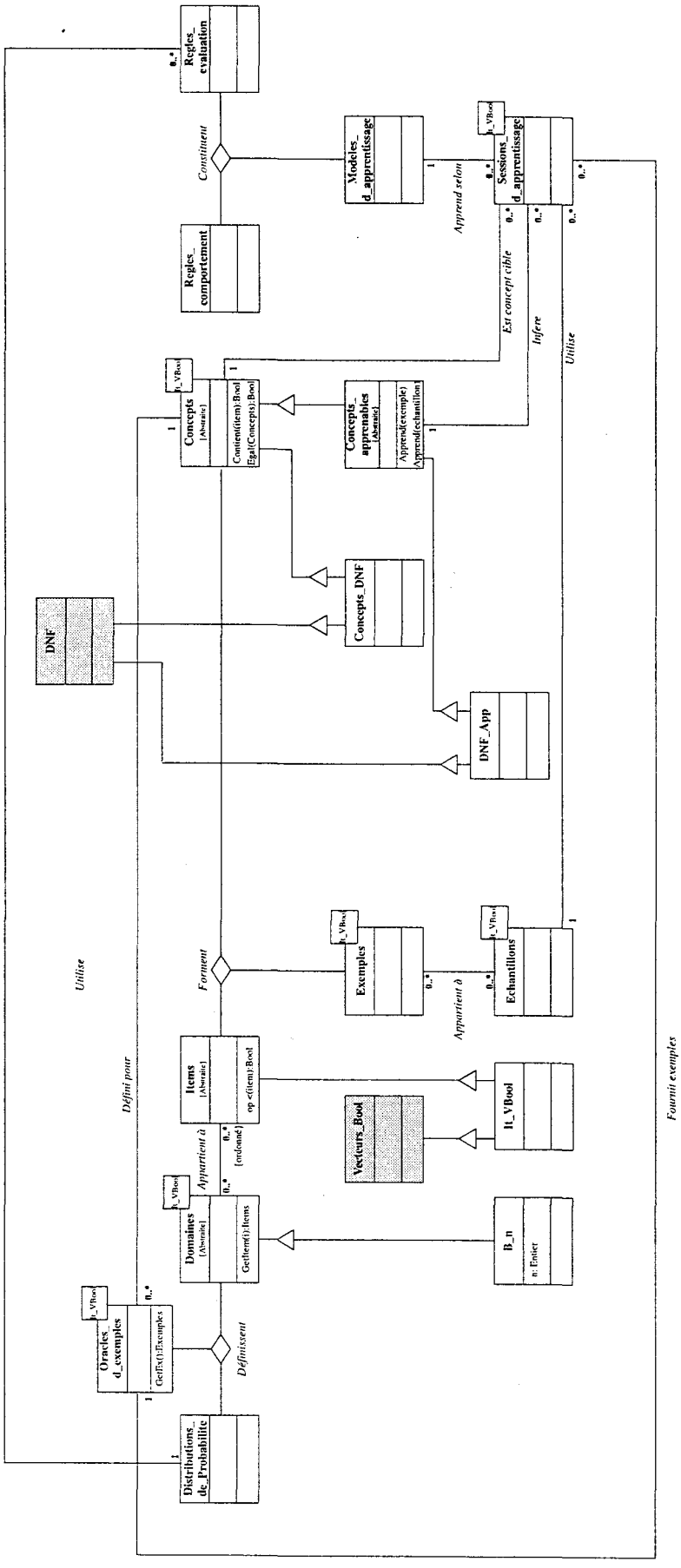


FIG. 4.8 - Intégration du contexte des formules booléennes.

Chapitre 5

Implantation de <PEpIn>

Ce chapitre d'ordre « technique » présente l'implantation que nous avons faite de <PEpIn>. L'objectif principal dans cette tâche était d'être le plus proche possible du modèle conceptuel présenté dans le chapitre précédent. De plus, l'ensemble des impératifs auxquels devait répondre la plate-forme a du être pris en compte et ce malgré quelques difficultés inhérentes au langage de programmation utilisé. Le but de ce chapitre n'est évidemment pas de rentrer dans les détails de l'implantation de <PEpIn>. Nous nous proposons d'expliquer globalement le principe d'implantation choisi ainsi que quelques difficultés auxquelles nous avons dû faire face. Les deux premières sections, assez générales, justifient le choix du langage C++ et présentent succinctement l'implantation des *classes* de base. Les deux sections suivantes s'intéressent à l'intégration de l'aléatoire et du calcul d'erreur permettant d'évaluer une hypothèse. Enfin, la dernière section reviendra sur le run-time en donnant, entre autre, quelques unes des conditions d'arrêt que nous avons intégrées.

5.1 Généralités

5.1.1 Choix du langage C++

Nous l'avons déjà dit, <PEpIn> a été implanté en C++, le langage créé par Bjarne Stroustrup [Str92]. Le choix de ce langage nous a semblé judicieux pour deux raisons principales : d'abord, il intègre l'ensemble des principes de la programmation objet. Ensuite, c'est un langage générant du code performant, ce qui est un avantage certain étant donné le temps de calcul requis par certains algorithmes d'apprentissage. Remarquons que C++ n'est pas un langage « tout objet », c'est-à-dire que certains types de base ne sont pas décrits dans le langage en tant qu'objet et qu'il est possible de ne pas utiliser l'aspect objet de celui-ci. Cependant, nous nous sommes fixés comme règle de tout implanter de manière objet de façon à tirer l'ensemble des avantages de ce style de programmation.

5.1.2 Librairie traitant des structures de données

L'implantation de <PEpIn> s'appuie également sur la bibliothèque STL (Standard Template Library) conçue par Alexandre Stepanov et Ming Lee [SM94, Fon97]. Cette librairie implante l'ensemble des structures de données « classiques » telles que les listes chaînées, les vecteurs, les piles ou bien encore les ensembles. Cette librairie a l'avantage d'être disponible pour un grand nombre de compilateurs. De plus, la mise en œuvre qui est faite de la plupart

des structures implantées est telle que les traitements sur des objets de taille importante sont très performants.

5.1.3 Généricité ou héritage

L'objectif de la plate-forme <PEpIn> consiste entre autre à décrire les différents intervenants de la théorie de l'apprentissage ainsi que les interactions entre ces éléments. Cette description en dehors de tout contexte implique par conséquent que la plupart de ces classes soient abstraites, c'est-à-dire qu'elles ne peuvent en aucun cas être instanciées telles quelles. Cependant, les définitions des classes abstraites doivent inclure l'ensemble des méthodes attendues sur les objets qu'elles représentent, sans pour autant intégrer le corps de ces méthodes (ceux-ci dépendant en général du contexte) ; de telles méthodes sont dites *virtuelles pures.*

Ceci pose un certain nombre de problèmes en C++ étant donné qu'il est impossible dans ce langage d'obtenir de l'information sur le type dynamique d'un objet.

Imaginons par exemple une classe abstraite `A` intégrant une méthode virtuelle pure `f` avec un argument du type `A`.

```
class A {
public:
  virtual f(const A& a) = 0;
};
```

Imaginons maintenant `B` une classe fille de `A` non abstraite et définissant la méthode `f` avec un argument du type `B` (c'est-à-dire également du type `A`).

```
class B: public A {
public:
  virtual f(const B& a)
  {
    /* Ici corps de la methode */
  }
};
```

Telle que définie ci-dessus, la classe `B` reste abstraite puisque la méthode `f` n'a été redéfinie que pour un sous-ensemble d'éléments du type `A`. La méthode `f` prenant comme argument un objet du type `A` n'est donc pas plus définie pour la classe `B` qu'elle ne l'est pour la classe `A`. C'est un problème difficile à résoudre en C++ dès lors que l'on veut le traiter proprement.

Dans notre cas, nous avons contourné ce problème par l'utilisation de la généricité (classes *patrons* ou *template* du C++). Nous verrons un exemple de la solution que nous proposons à la section 5.2 notamment sur la classe des items et sur celle des concepts.

5.2 Classes de Base

Les classes présentées ici sont typiquement des classes abstraites et génériques qui attendent d'être « plongées » dans un contexte particulier.

5.2.1 Les items.

Les items sont les éléments de base du domaine considéré. Un item peut être passés en paramètre à la méthode `contient` d'un concept afin d'évaluer son appartenance à ce concept. Les items permettent également de construire des exemples labellés. Il doit exister un ordre sur les items afin d'établir une numération sur ceux-ci ; cette numération sera utilisée par la suite pour implanter les distributions de probabilité sur les domaines. Un item doit par conséquent pouvoir être comparé avec un autre item du même type. Nous nous trouvons alors dans le cas du problème évoqué à la section 5.1.3 (page précédente) puisque la classe `Items` doit déclarer l'opérateur de comparaison (disons `operator<`) entre deux items de même nature. Afin de contourner ce problème, nous proposons de paramétrer la classe `Items` au moyen d'une classe définissant effectivement la représentation des items. Cette classe a, normalement, une existence propre dans le contexte considéré[1]. De cette manière, un item pourra être comparé avec la **représentation** d'un autre item. Cette solution n'est pas complètement satisfaisante au niveau de la sécurité sur les types mais, utilisée proprement, elle permet de répondre à nos exigences.

La classe `Items` est présentée ci-dessous :

```
template<class RepItem>
class Items {

public:

  /* ------------- */
  /* Constructeurs */
  /* ------------- */

  inline Items(void)
  /* Construit l'item vide */
  {};

  /* ------------------------ */
  /* Operateurs de comparaison */
  /* ------------------------ */

  virtual bool operator<(const RepItem& item) const = 0 ;

  virtual bool operator==(const RepItem& item) const = 0 ;
};
```

Afin de comprendre comment sera utilisée cette classe, nous présentons un petit exemple simple. Supposons que la classe `X` implante une représentation d'items (par exemple des vecteurs booléens). Nous supposons également que cette classe inclut les deux opérateurs de comparaison. La classe `X` est de la forme :

```
class X {
public:

  virtual bool operator<(const X& x) const
  {
```

1. Rappelons que selon le principe du génie logiciel, une librairie définissant un contexte peut être utilisée à d'autres fins que pour l'apprentissage.

```
    /* Corps de la methode */
  }

  virtual bool operator==(const X& x) const
  {
    /* Corps de la methode */
  }
};
```

Nous pouvons alors créer une classe `ItemsX` héritant à la fois de la classe `Items` et de la classe `X` :

```
class ItemsX : public Items<X>, public X {
public:

  ItemsX(void);
  /* Construit l'item vide */

  ItemsX(const ItemsX& m);


  virtual bool operator<(const X& item) const
  {
    return this->X::operator<(item);
  }

  virtual bool operator==(const X& item) const
  {
    return this->X::operator==(item);
  }
};
```

5.2.2 Les domaines

La classe des domaines est définie uniquement afin de permettre de générer le i^e item selon l'ordre défini sur ceux-ci. Cette classe sera essentiellement utilisée par les oracles afin de générer automatiquement des exemples. Rappelons que les domaines peuvent être de nature très diverses. Par exemple, dans le contexte des formules booléennes, ce peut être l'ensemble des vecteurs booléens de dimension n. Dans le contexte des langages réguliers, il peut s'agir de l'ensemble Σ^* (avec Σ un alphabet) ou bien encore l'ensemble des mots de longueur bornée.

La classe `Domaines` est déclarée comme suit :

```
template<class TypeItem>
class Domaines {
public:

  typedef unsigned int NumerosItem;

  /* ------------- */
  /* Constructeurs */
  /* ------------- */
```

```
  inline Domaines(void)
  /* Constructeur de domaine vide */
  {};

  /* -------------------- */
  /* Generation d'un item */
  /* -------------------- */

  virtual TypeItem get_item(NumerosItems i) = 0;
};
```

Précisons enfin que cette classe étant à la fois générique et abstraite, il conviendra de la dériver et d'instancier son paramètre générique.

5.2.3 Les concepts

La classe `Concepts` permet d'intégrer la représentation d'objets mathématiques qui vont jouer le rôle de concepts dans l'environnement d'apprentissage. On suppose que la librairie implantant le contexte intègre *a priori* une classe définissant cette représentation. La classe abstraite `Concepts` permet donc simplement d'utiliser cette classe de représentations en tant que concepts pour l'apprentissage. La classe `Concepts` déclare trois méthodes particulières. La première permet de comparer entre eux deux concepts ; cette méthode est indispensable à ce niveau afin d'être héritée dans les classes de concepts apprenables ce qui permet de comparer une hypothèse et un concept cible. La deuxième méthode a pour but de décider de l'appartenance d'un item à un concept en vue notamment de créer des exemples labellés. Enfin, la dernière méthode retourne la taille du concept considéré. Cette taille dépend évidemment du contexte et des données à étudier. Par exemple pour une représentation des langages réguliers au moyen des DFAs, la taille peut correspondre au nombre d'états de l'automate. Dans le cas de la représentation des formules booléennes ce peut être le nombre de variables apparaissant dans la formule. Cette information sur la taille de la représentation d'un concept sera utilisée comme donnée résultat au cours d'une session d'apprentissage ; il peut en effet être intéressant d'étudier l'évolution de la taille de représentation des différentes hypothèses inférées au cours d'un apprentissage.

Le profil de la classe `Concepts` est le suivant :

```
template <class TypeItem, class RepConcept>
class Concepts {
public:

  /* ------------- */
  /* Constructeurs */
  /* ------------- */

  inline Concepts(void)
  /* Constructeur de concept vide */
  {};

  /* --------------------------- */
  /* Comparaison de deux concepts */
  /* --------------------------- */
```

```
  virtual bool est_egal(const RepConcept& c) const = 0;

  /* ---------------------- */
  /* Appartenance d'un item */
  /* ---------------------- */

  virtual bool contient(const TypeItem& item) const = 0;

  /* --------------------------- */
  /* Taille de la representation */
  /* --------------------------- */

  virtual unsigned long taille(void) const = 0;
};
```

Remarquons qu'on retrouve ici encore le problème évoqué à la section 5.1.3 (page 142) puisqu'il faut pouvoir comparer un concept avec un autre concept en assurant certaines contraintes sur les types. Nous traitons cette difficulté de la même façon que pour le cas des `Items` en introduisant le paramètre générique `RepConcept`.

5.2.4 Exemples et échantillons

Les deux dernières classes de base que nous présentons définissent les exemples et les échantillons. Nous ne donnerons pas ici les profils de ces deux classes qui sont relativement classiques. Précisons simplement que ces deux classes génériques sont paramétrées par le type de l'item. De plus, elles n'ont pas à être dérivées puisque l'ensemble des fonctionnalités disponibles sur les exemples et les échantillons peut être défini indépendamment du contexte.

5.3 Traitement de l'Aléatoire

Les applications d'expérimentations sur l'inférence inductive créées au moyen de la plateforme <PEpIn> sont appelées à être utilisées de manière automatique. Cela signifie qu'une fois correctement paramétrée, une expérience d'apprentissage pourra se dérouler sans intervention de l'expérimentateur. Dans ce cas, la machine joue à la fois le rôle de l'apprenant (en exécutant l'algorithme d'apprentissage) et celui de l'enseignant. Afin de répondre à cette seconde tâche, le système, au travers des oracles, doit donc être capable de générer « aléatoirement » des exemples pour une cible donnée. Dans cette section, nous nous proposons d'étudier de quelle manière, nous avons intégrer cette dimension stochastique à <PEpIn>.

5.3.1 Générateur Pseudo Aléatoire

Les machines « classiques » sont par définition des machines déterministes, c'est-à-dire qu'il est possible de prévoir le comportement de n'importe quel programme et le résultat retourné par celui-ci en fonction des valeurs d'entrée. Dès lors, il devient évident qu'un véritable générateur de nombre aléatoire est impossible à implanter sur une telle machine.

Le principe consiste donc à simuler un tel générateur au moyen d'un algorithme déterministe mais générant des entiers de manière telle que sans la connaissance de cet algorithme la suite des entiers sorties semble aléatoire.

Nous avons défini une classe dont les objets instances sont de tels générateurs. L'algorithme sur lequel elle se base est celui décrit par D.H. Lehmer. Cet algorithme est très simple dans son comportement puisqu'il utilise le résultat de l'étape $n-1$ pour générer la valeur de l'étape n. La première valeur utilisée pour générer le premier nombre « aléatoire » est appelée valeur semence ; en général, on utilise comme semence une valeur sur laquelle l'utilisateur n'a pas la main (telle que l'heure système par exemple). L'algorithme de Lehmer est tel que quelque soit la semence, le nombre d'étapes avant de cycler (c'est-à-dire avant de retourner une valeur déjà générée) est très grand (supérieur à 2.10^9). Dans la suite, nous appellerons cette valeur $ValMax$.

En résumé, un objet du type `GPA` est un distributeur de nombres entiers. Les entiers pouvant être générés par un tel objet sont compris entre 0 et $ValMax$. De plus, si l'entier n a été généré à l'étape i, il sera à nouveau retourné à l'étape $i + ValMax$ mais pas avant.

5.3.2 Distributions de probabilité

La classe abstraite `DistriProba` permet de définir des mesures de probabilité sur l'ensemble $I\!N$ (ou plutôt sur l'intervalle $[0, ValMax]$). Cette classe correspond à la classe mère de toutes les classes définissant des distributions de probabilité particulières. Elle déclare deux méthodes principales : la première permet de générer de manière pseudo aléatoire un entier selon la mesure de probabilité considérée (en s'appuyant sur un objet du type `GPA`). La seconde méthode déclarée retourne la valeur de probabilité d'un entier passé en paramètre.

Le profil de la classe `DistriProba` est le suivant :

```
class Dist_Proba {
protected:

  GPA le_gpa; /* Generateur de nombres aleatoires propre a la distribution. */

  virtual double la_fonction(unsigned int n) const = 0;
  virtual double get_limite_somme(void) const = 0;

public:
  /* ------------ */
  /* Constructeur */
  /* ------------ */

  inline Dist_Proba(void);
  {};

  /* --------------------- */
  /* Calcul de probabilite */
  /* --------------------- */

  inline double proba(unsigned long n) const
  { return la_fonction(n) / get_limite_somme(); };

  /* ------------------------------------------------- */
  /* Generation aleatoire selon la distribution de proba. */
  /* ------------------------------------------------- */
  unsigned long get_rnd(void);
};
```

L'implantation d'une distribution de probabilité s'appuie sur une fonction f de $\mathbb{N}$ vers $[0,1]$. Cette fonction est supposée telle que : $\lim_{p\to\infty} \sum_{n=0}^{p} f(n) = l$. La probabilité d'un entier n est alors définie par : $P(n) = f(n)/l$. On remarque évidemment que $\sum_n P(n) = \sum_n f(n)/l = 1$; ce qui assure que P est bien une mesure de probabilité. Dans l'implantation, la méthode `la_fonction` représente f et la méthode `get_limite_somme` représente l la valeur de convergence de $\sum_n f(n)$.

Générateur de nombres aléatoires selon la distribution de probabilité

L'algorithme[2] présenté à la figure 5.1 génère un nombre aléatoire selon une distribution de probabilité construite sur la fonction f avec $\lim_{p\to\infty} \sum_{n=0}^{p} f(n) = l$. Rappelons qu'un GPA retourne une valeur comprise entre 0 et $ValMax$.

Algorithme `get_rnd_proba`
Entrée : Début
  $Alea$ = nombre aléatoire généré par un GPA
  $Alea = \frac{Alea}{ValMax} \times l$
  $Res = 0$
  $BorneSegment = f(0)$
  Tantque $Alea \geq BorneSegment$
    $Res = Res + 1$
    $BorneSegment = BorneSegment + f(Res)$
  Ftq
  Retourner Res
Fin

FIG. 5.1 - Algorithme de génération d'un nombre pseudo aléatoire selon une distribution de probabilité.

Le principe de cet algorithme consiste simplement à affecter des segments contigus de $[0,1]$ à chaque valeur de $\mathbb{N}$; la longueur d'un segment (égale à $f(n)$) est d'autant plus importante que la probabilité de l'entier qu'il représente est élevée. On recherche alors à quel segment appartient le réel $Alea$ généré de manière pseudo aléatoire au moyen d'un GPA. Remarquons que l'on assure que $Alea \leq l$ et donc que l'algorithme s'arrête puisque $\sum_n f(n)$ converge vers l (et donc la valeur de $BorneSegment$ aussi).

Implantation de distributions particulières

L'implantation de distributions de probabilité particulières est dès lors très simple puisqu'il suffit de définir la fonction f (`la_fonction`) sur laquelle se base cette mesure de probabilité ainsi que la limite de convergence de $\sum_n f(n)$.

2. Cet algorithne s'inspire de la méthode dite de « l'anamorphose » de génération de valeur d'une variable aléatoire continue.

Nous définissons par exemple la classe implantant les distributions de probabilité uniformes entre deux entiers a et b. La fonction f sous-jacente est définie par :

$$f(n) = \begin{cases} \frac{1}{b-a+1} & \text{si } n \in [a,b] \\ 0 & \text{sinon} \end{cases}$$

La limite de convergence de $\sum_n f(n)$ est égale à 1.

L'utilisation de cette distribution de probabilité dans le cas des langages réguliers est étudiée à la section 6.3.2 (page 178).

5.3.3 Oracles

Les distributions de probabilité étant définies, nous pouvons les utiliser afin de construire des oracles d'exemples. Deux types d'oracles sont envisagés. D'abord, les *oracles manuels* qui consistent simplement à demander à l'expérimentateur l'item constituant l'exemple. Dans ce cas, on peut considérer que c'est l'utilisateur lui même qui joue le rôle de l'oracle et donc de l'enseignant. Il utilise donc sa propre distribution de probabilité sur le domaine (en général, celle-ci n'est pas formalisable). Le deuxième type d'oracles implantés, appelés *oracles automatiques*, permet de générer automatiquement des exemples (positifs ou quelconques) pour un concept particulier. Un tel oracle s'appuie sur une distribution de probabilité particulière. Un oracle est également associé au domaine dans lequel les items sont tirés. Du point de vue de l'oracle, le domaine considéré est par conséquent muni de la distribution de probabilité attribuée à l'oracle ; la probabilité d'un item étant la probabilité de son numéro d'ordre. La génération pseudo aléatoire d'un item selon une distribution de probabilité consiste dons simplement à générer un entier n selon cette distribution puis à demander au domaine de retourner le n^e. Rappelons qu'il existe une méthode de la classe `Domaine` générant le n^e item d'un domaine.

Pour le cas des oracles positifs, le principe est le même à la différence que le processus de génération d'exemples est répété tant que l'exemple construit n'est pas positif pour le concept considéré. Plus précisément, soit D le domaine sur lequel sont tirés les items. Le domaine D est muni de la mesure de probabilité P. Soit c le concept pour lequel l'oracle est défini. L'algorithme de génération d'exemple positif pour c est le suivant :

Algorithme `Genere_exemples_positifs`
Début
  Faire
    Tirer $X \in D$ selon P
  Jusqu'á $X \in c$
  Retourner $(X, 1)$
Fin

Étudions dans ce cas quelle est la probabilité $P'(x)$ que l'item $x \in c$ soit retourné par l'oracle, c'est-à-dire $P'(x) = Pr(X = x)$. Soit E l'événement « Le premier item tiré appartient à f ». D'après la formule de Bayes, on a,

$$P'(x) = Pr(X = x \mid E)Pr(E) + Pr(X = x \mid \overline{E})Pr(\overline{E})$$

Avec,

$$Pr(E) = \sum_{y \in c} P(y) = P(c)$$

et,

$$Pr(\overline{E}) = 1 - P(c)$$

De plus, par définition pour les probabilités conditionnelles,

$$Pr(X = x \mid E) = \frac{Pr(X = x \cap E)}{Pr(E)} = \frac{P(x)}{P(c)}$$

Comme tous les tirages sont indépendants, on a,

$$Pr(X = x \mid \overline{E}) = P'(x)$$

Donc finalement,

$$P'(x) = P(x) + P'(x)(1 - P(c))$$

Soit encore,

$$P'(x) = \frac{P(x)}{P(c)}$$

Remarquons que dans le cas où le domaine D et le concept c ont une intersection nulle[3] cette méthode tourne indéfiniment, ce qui peut encore se traduire par $P(c) = 0$.

Il est possible d'assurer que la génération d'un exemple positif se fasse en une seule étape. Pour cela, il suffit de dériver la classe des concepts considérée de la classe `Domaine`. Il devient alors possible de définir des oracles positifs ayant pour domaine le concept cible lui-même. Alors, tout item tiré dans le domaine appartient obligatoirement au concept cible (puisque le domaine utilisé est le concept cible lui-même), ce tirage se fait bien en une étape puisque dès le premier tirage l'item est un exemple positif pour la cible. Dans ce cas, la probabilité qu'un item x soit retourné par l'oracle est $P'(x) = P(x)$.

Afin de ne pas allourdir inutilement ce chapitre, nous ne donnons pas les profils des classes d'oracles. Précisons simplement qu'il existe une classe mère abstraite et générique dont le paramètre générique est un type d'item. De plus, tout oracle est étroitement associé au concept labellant ses exemples au moyen d'une référence sur celui-ci.

5.4 Calcul d'Erreur

Nous nous intéressons dans cette section au calcul de l'erreur commise par l'algorithme d'apprentissage. Cette erreur est d'une grande importance dans le cas de l'apprentissage approximatif puisque le critère de succès en dépend. Dans les définitions des différents modèles que nous avons présentés (voir chapitre 1, page 25), l'erreur permettant d'évaluer une hypothèse est calculée pour l'ensemble des items du domaine. Rappelons (voir définition 1.3.1, page 33) que l'erreur entre deux concepts f et g sur un domaine munis de la distribution de probabilité P est définie par :

$$Erreur_{P,f}(g) = \sum_{x \in f \Delta g} P(x)$$

3. Ce qui n'arrive jamais dans le cas de la théorie de l'apprentissage.

Dans la plupart des cas, le domaine étant infini, il s'agit évidemment d'une valeur théorique qui n'est pas calculable de manière effective. Il est cependant possible d'approximer cette erreur grâce à un certain nombre de résultats statistiques.

Le calcul de l'erreur (qu'elle soit réelle ou approximée) est un « objet » purement algorithmique qui peut être implanté par une simple procédure. Cependant, afin de respecter notre volonté de tout implanter de manière objet, nous proposons trois classes définissant le calcul d'erreur. La première de ces classes est une classe abstraite mère des deux autres. Nous donnons ci-après le profil de cette classe.

```
template<class TypeItem>
class CalcErreurs {

protected:

  Oracles<TypeItem>* L_Oracle;
  double taux_erreur;

  virtual void calcule_erreur(const Concepts<TypeItem>& h) = 0;

public:

  CalcErreurs(const Concepts<TypeItem>& h,Oracles<TypeItem>* o):L_Oracle(o)
  {
    calcule_erreur(h);
  };

  /* ---------- */
  /* Accesseurs */
  /* ---------- */

  double get_erreur(void) const
  {
    return taux_erreur;
  };

  /* ------- */
  /* Clonage */
  /* ------- */

  virtual CalcErreurs<TypeItem>* clone(void) const = 0;

  /* ------------------------- */
  /* Methodes d'entrees/sorties */
  /* ------------------------- */

  virtual void affiche(ostream& flot) const = 0;

  friend ostream& operator<<(ostream& o,const G_Erreurs<TypeItem>& e);
};
```

Le calcul de l'erreur d'une hypothèse par rapport à un concept cible se fait en utilisant

un oracle associé à cette cible ; en général, il s'agit du même oracle que celui ayant servi à l'apprentissage. Rappelons qu'un oracle contient une référence à la fois sur le concept cible servant à labeller les exemples et sur la distribution de probabilité associée au domaine. Notons qu'il est indispensable que l'erreur connaisse l'oracle afin de prendre en compte le cas des oracles d'exemples positifs. Dans ce cas, seuls les exemples positifs doivent être utilisés pour calculer l'erreur (on peut imaginer que la distribution de probabilité est ramenée à sa restriction sur les exemples positifs). Enfin, remarquons que le calcul d'erreur n'est possible que dans le cas où la présentation des exemples a été faite au moyen d'un oracle automatique. En effet, si c'est à l'expérimentateur humain que revient la tâche de fournir les exemples, il n'est dans ce cas pas possible de connaître la distribution de probabilité utilisée.

La classe mère de calcul d'erreur que nous venons de présenter est dérivée en deux sous-classes implantant le calcul d'erreur réelle et le calcul d'erreur approximée. Celles-ci sont détaillées dans la suite. C'est la méthode virtuelle `calcul_erreur` qui différencie finalement les deux types d'erreur.

5.4.1 Erreur réelle

L'erreur réelle d'une hypothèse h par rapport au concept cible c et sous la distribution de probabilité P est la probabilité que h classe mal (c'est-à-dire différemment de c) un item tiré aléatoirement selon P. C'est ce type d'erreur qui est utilisé dans la description des modèles d'apprentissage pour conditionner le pouvoir prédictif des hypothèses.

Le calcul de l'erreur réelle consiste donc à sommer les poids de tous les items du domaine mal classés par l'hypothèse. Nous l'avons déjà dit, ce type d'erreur est, en général, une valeur théorique non calculable effectivement. Cependant, lorsque la distribution de probabilité est telle que seul un nombre fini d'items a une probabilité non nulle alors l'erreur réelle peut être effectivement calculée à condition de connaître la valeur du dernier item ayant une probabilité non nulle. Les distributions uniformes sur $[a, b]$ intègrent cette propriété puisque seuls les entiers compris entre les deux bornes a et b ont une probabilité non nulle.

Dans ce type de calcul, l'oracle ne doit pas être utilisé de manière traditionnelle c'est-à-dire qu'il ne doit pas servir à générer aléatoirement des exemples mais plutôt à distribuer l'intégralité des exemples ayant une probabilité non nulle. Il convient donc d'intégrer cette fonctionnalité aux oracles.

5.4.2 Erreur approximée

Le calcul d'erreur approximée sert à évaluer l'erreur réelle lorsque celle-ci ne peut être calculée exactement. Une première idée pour calculer une approximation de l'erreur sur h est d'utiliser un échantillon S de N exemples et de poser :

$$Erreur_{S,f}(h) = \frac{1}{N} \sum_{x \in S} \delta(f(x), h(x))$$

avec $\delta(f(x), h(x))$ égal à 1 si $f(x) \neq h(x)$ et égal à 0 sinon. Ce type d'erreur est souvent appelé *erreur d'échantillon* ou encore *taux d'erreur apparent*. D'après la loi des grands nombres, lorsque N, la taille de l'échantillon, augmente le taux d'erreur apparent converge vers l'erreur réelle. Cependant, pour approximer l'erreur, il convient de ne pas utiliser le même échantillon que celui ayant servi à l'apprentissage. En effet, l'hypothèse construite à partir de cet échantillon est, en général, bien adaptée à celui-ci. Le taux d'erreur apparent sur cet échantillon

d'apprentissage est par conséquent souvent proche de 0 ; on dit qu'il existe dans ce cas un *biais inductif*. Dans [Gas93], Gascuel présente en détail ce phénomène ainsi que des méthodes permettant d'utiliser un seul échantillon pour apprendre et évaluer l'erreur sur l'hypothèse.

En ce qui nous concerne, l'échantillon permettant de calculer le taux d'erreur apparent est un ensemble tiré indépendamment de l'échantillon d'apprentissage. Évidemment, la valeur du taux d'erreur apparent dépend de l'échantillon utilisé pour la calculer (c'est-à-dire que ce taux peut varier en fonction de l'échantillon tiré). Nous allons étudier ici de quelle manière les résultats statistiques permettent d'utiliser le taux d'erreur apparent afin d'approximer l'erreur réelle (le lecteur pourra se reporter au chapitre 5 de [MCM83] pour une présentation complète de ces résultats).

Soit ε l'erreur réelle. Supposons que l'on tire un échantillon test de N exemples. Supposons également que n exemples de cet échantillon sont mal classés par l'hypothèse h. La question que nous nous posons ici est : « Dans quelle mesure la fonction $Erreur_{S,f}(h) = \frac{n}{N}$ approxime-elle $Erreur_{P,f}(h) = \varepsilon$? ».

La variable aléatoire n suit une loi binomiale de paramètre ε. Rappelons que la loi binomiale décrit pour chaque valeur possible de n (comprise entre 0 et N) la probabilité d'obtenir exactement n exemples mal classés étant donné un échantillon de N exemples tirés de manière indépendante avec ε la probabilité qu'un exemple soit mal classé. Suivant cette loi nous avons donc :

$$Pr(n) = \frac{N!}{n!(N-n)!}\varepsilon^n(1-\varepsilon)^{N-n}$$

Malheureusement, la loi binomiale est difficile à manipuler avec des grands nombres. Cependant, la loi binomiale peut être approximée par la loi normale (ou loi de Gauss) pour des échantillons de taille suffisamment large. Le théorème de la limite centrale établit que :

$$\lim_{N\to\infty} Pr\left[b\sqrt{\frac{\varepsilon(1-\varepsilon)}{N}} \leq \varepsilon - \frac{n}{N} \leq a\sqrt{\frac{\varepsilon(1-\varepsilon)}{N}}\right] = \frac{1}{\sqrt{2\pi}}\int_{-a}^{-b} e^{-\frac{x^2}{2}}dx$$

L'usage est d'admettre cette égalité lorsque :

$$N\varepsilon, N(1-\varepsilon) \geq 5 \ \textbf{(cond1)}$$

et lorsque,

$$N \geq 30 \ \textbf{(cond2)}$$

,

On obtient donc l'inégalité suivante :

$$Pr\left[\varepsilon \leq \frac{n}{N} + a\sqrt{\frac{\varepsilon(1-\varepsilon)}{N}}\right] \geq \frac{1}{\sqrt{2\pi}}\int_{-a}^{+\infty} e^{-\frac{x^2}{2}}dx = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{a} e^{-\frac{x^2}{2}}dx$$

Dans cette inégalité, $a\sqrt{\frac{\varepsilon(1-\varepsilon)}{N}}$ définit la précision de l'approximation. La partie droite de l'inégalité quant à elle définit la confiance, c'est-à-dire la probabilité que $Erreur_{P,f}(h)$ appartienne à l'intervalle $Erreur_{S,f}(h) \pm a\sqrt{\frac{Erreur_{S,f}(h)(1-Erreur_{S,f}(h))}{N}}$

La valeur du paramètre a est fonction de la confiance attendue et est donnée par une table que l'on peut trouver dans tous les livres de statistiques.

Les résultats statistiques que nous venons de donner permettent de donner un algorithme calculant avec une précision et une confiance données l'approximation de l'erreur réelle pour

une hypothèse par rapport à un concept cible. Cet algorithme donné à la figure 5.2 est celui que nous avons implanté dans la classe servant à calculer les erreurs approximées.

Algorithme `calcule_erreur`
Entrée : p précision, c confiance
Début
Soit $N = 100$
Soit a la valeur donnée par la table pour obtenir
une confiance supérieure ou égale à c
Soit S un échantillon de N exemples tirés selon $Exemple_{X,P}(f)$
Soit n le nombre d'exemples de S mal classés par h
`* Estimation du nombre d'exemples a tirer`
`pour assurer la précision désirée`
$N_{Estime} = \frac{a^2 * \frac{n}{N} * (1 - \frac{n}{N})}{p^2}$
Si $N < N_{Estime}$
Alors
Tirer $N_{Estime} - N$ exemples selon $Exemple_{X,P}(f)$
Ajouter à n le nombre de nouveaux exemples mal classés par h
$N = N + N_{Estime}$
Fsi
$err = \frac{n}{N}$
$p_{calculee} = a * \sqrt{\frac{err * (1 - err)}{N}}$
`* ajustement progressif de manière à assurer les conditions`
Tantque $p_{calculee} > p$ ou **cond1** non vérifiée
Tirer 100 nouveaux exemples selon $Exemple_{X,P}(f)$
Ajouter à n le nombre de nouveaux exemples mal classés par h
$N = N + 100$
$err = \frac{n}{N}$
$p_{calculee} = a * \sqrt{\frac{err * (1 - err)}{N}}$
Ftq
retourner err
Fin

FIG. 5.2 - Algorithme d'approximation de l'erreur réelle au moyen d'un échantillon test.

Le principe de l'algorithme est le suivant : on tire d'abord un petit échantillon et l'on calcule le taux d'erreur apparent sur cet échantillon. Cette première estimation de l'erreur réelle permet alors d'évaluer le nombre de nouveaux exemples à tirer pour garantir la précision désirée. Ces nouveaux exemples sont ajoutés à l'échantillon de test et on calcule le nouveau taux d'erreur apparent sur celui-ci. On vérifie alors que les conditions permettant de dire que le taux d'erreur apparent approxime l'erreur réelle sont satisfaites. Notamment, il faut que la condition **cond1** soit vérifiée à savoir que $|S| Erreur_{S,f}(h) \geq 5$ et $|S|(1 - Erreur_{S,f}(h)) \geq 5$. Remarquons que la condition **cond2** est toujours réalisée puisque le nombre d'exemples

test tirés est, dès le début de l'algorithme, supérieur à 30. Tant que ces conditions ne sont pas satisfaites, on ajoute progressivement un petit nombre d'exemples afin d'ajuster le taux d'erreur apparent.

À cause de cette vérification des conditions, il se peut que l'algorithme prenne un temps très long (voire infini). Aussi, nous nous proposons, dans une version future de la plate-forme, d'ajouter une autre méthode de calcul d'erreur avec comme paramètres le nombre d'exemples à utiliser pour calculer l'erreur ainsi que la confiance désirée. Cette méthode utilisera alors exactement le nombre d'exemples demandé pour calculer, avec la confiance exigée, l'erreur approximée avec la meilleure précision possible.

5.5 Run Time

Cette dernière section traite de l'implantation du run-time de <PEpIn>. La difficulté dans la mise en œuvre de cette partie vient du fait qu'elle doit être indépendante du contexte auquel elle sera associée. Aussi, la présentation qui en est faite ici, insistera sur les choix que nous avons faits afin de répondre à cette exigence d'indépendance.

5.5.1 Session d'apprentissage

Une session d'apprentissage correspond à un paramétrage particulier d'un modèle d'apprentissage. Une fois entièrement paramétrée, une session d'apprentissage peut alors être lancée en suivant le run-time présenté au chapitre précédent (voir figure 4.6, page 137). Au cours de l'exécution de la session d'apprentissage, un certain nombre de résultats sont calculés. En fin de session, chacun de ces résultats est susceptible d'être utilisé par l'expérimentateur pour l'analyse de l'apprentissage. L'ensemble des résultats calculés doit donc être mémorisé dans la session d'apprentissage ; c'est le rôle des objets du type `Historique`.

Gestion de l'historique

Un historique est une liste chaînée dont chaque maillon mémorise les résultats concernant une étape du processus d'apprentissage. On rappelle que le début d'une étape correspond au tirage d'un nouvel exemple. Dans la version actuelle de <PEpIn>, les informations mémorisées à chaque étape sont : l'exemple tiré, l'hypothèse inférée et l'erreur de l'hypothèse. Celles-ci sont regroupées dans un type particulier de nom `Info_Etape` :

```
template<TypeItem>
struct Info_Etape{
  Exemples<TypeItem> l_exemple;
  Erreurs<TypeItem>* Erreur;
  Concepts_App<TypeItem>* Hypothese;
};
```

```
class A {
public:
  virtual A* clone(void) const = 0;
};

class A1 {
public:
  virtual A1* clone(void) const
  {
    return new A1();
  }
};

class A2 {
public:
  virtual A2* clone(void) const
  {
    return new A2();
  }
};

/* Programme principal */
void main(void)
{
  A1 a1;
  A2 a2;
  A* a;

  a = a1.clone(); /* *a du type A1 */
  delete a;

  a = a2.clone(); /* *a est maintenant du type A2 */
  delete a;
}
```

FIG. 5.3 - Définition de clonage sur une classe.

À ce niveau de l'implantation, il est impossible de connaître exactement le type des objets `Erreur` et `Hypothese`. En effet, le type exact de l'erreur (erreur réelle ou erreur approximée) n'est fixé qu'au moment du paramétrage de la session d'apprentissage par l'expérimentateur. Il en est de même pour le type de l'hypothèse puisque celui-ci correspond à l'algorithme d'apprentissage à utiliser. C'est la raison pour laquelle ces objets ne sont déclarés qu'en tant que pointeurs sur des classes mères abstraites. Ceci pose cependant un problème : en effet, il doit être possible de créer dynamiquement de tels éléments afin de compléter progressivement l'historique. Cette création en cours d'exécution doit prendre en compte le vrai type de chacun de ces objets. Le langage C++ n'offrant pas la possibilité d'obtenir de l'information sur le type dynamique d'un objet, il convient d'intégrer aux objets appelés à se reproduire dynamiquement une méthode dite de *clonage*. Une telle méthode ne fait rien d'autre que de construire dynamiquement un objet du même type que celui sur lequel elle est appliquée ; cependant, étant donné que cette méthode a accés au type dynamique de l'objet à reproduire, le processus peut se faire sans problème. La figure 5.3 présente le code simple d'une classe

abstraite mère A et de deux classes $A1$ et $A2$ héritant de A. Une méthode de clonage est intégrée à ces trois classes. Le programme principal présente l'utilisation d'une telle méthode.

Profil de la classe Sessions_apprentissage

Cette idée de cloner les objets dont on ne connaît pas le type exact lors de l'écriture du code se retrouve à d'autres niveaux dans la classe définissant les sessions d'apprentissage. Remarquons que cette technique de clonage libère complètement la session d'apprentissage vis à vis du contexte dans lequel l'apprentissage sera plongé. Dès lors, la définition de la classe Session_apprentissage ne pose plus de problème et nous proposons le profil (quelque peu simplifié par rapport à sa véritable implantation) de cette classe.

```
template<TypeItem>
class Sessions_Apprentissage {

protected:
  /* Elements definis en dehors de la session d'apprentissage. */
  const Concepts<TypeItem>* La_Cible;
  Oracles<TypeItem>* L_Oracle;

  Echantillons<TypeItem> L_Echantillon;

  list<Conditions_Sessions<TypeItem>* > Liste_Cond_Arret;

  bool arrete(void);

  unsigned int Num_Etape; /* Numero de l'etape en cours */

  list<Info_Etape<TypeItem>* > Historique_Hypotheses;

public:

  /* ------------- */
  /* Constructeurs */
  /* ------------- */

  inline Sessions_Apprentissage(const Concepts<TypeItem>* cible,
  Oracles<TypeItem>& o, const Concepts_App<TypeItem>* apprenant,
  const CalcErreurs<TypeItem>* calcul_erreur);

  /* -------------------------- */
  /* Ajout de conditions d'arret */
  /* -------------------------- */

  inline void put_cond_arret(const Conditions_Sessions<TypeItem>& cond);

  /* -------- */
  /* Run-time */
  /* -------- */

  void run(void);
};
```

Pour terminer la présentation de cette classe, il nous reste à présenter la façon de paramétrer la fin d'une session. Ici encore, nous voulons que de nouvelles conditions d'arrêt puissent être ajoutées par la suite sans avoir à modifier le code principal du run-time. De plus, l'expérimentateur doit avoir la possibilité de paramétrer une session d'apprentissage avec certaines conditions d'arrêt.

5.5.2 Conditions d'arrêt

Afin de pouvoir définir des conditions d'arrêt indépendamment de la session d'apprentissage, nous définissons une classe `Conditions_arret`. Cette classe est évidemment abstraite étant donnée qu'elle n'implante *a priori* aucune condition particulière. Par la suite, toute condition d'arrêt pouvant être prise en compte par une session d'apprentissage devra hériter de cette classe mère.

Le profil de cette classe est le suivant :

```
template<TypeItem>
class Conditions_arret {
protected:

public:

  /* -------------------------- */
  /* Constructeur et Destructeur */
  /* -------------------------- */

  Conditions_arret(void)
  {};

  virtual ~Conditions_arret()
  {}

  /* ---------------------------------- */
  /* Operateur general sur la condition */
  /* ---------------------------------- */

  virtual bool operator()(const Sessions_Apprentissage<TypeItem>& s) const = 0;

  /* ------- */
  /* Clonage */
  /* ------- */

  inline virtual Conditions_Sessions<TypeItem>* clone(void) const = 0;

  /* ----------------------------------- */
  /* Message correspondant a la condition */
  /* ----------------------------------- */

  virtual inline Cl_String condition2str(void) const = 0;

  /* ----------------------- */
  /* Fonction d'entree/sortie */
  /* ----------------------- */
```

```
  inline void affiche(ostream& flot) const
  {
    flot << condition2str();
  };
};
```

Une instance de cette classe (ou d'une classe dérivée) doit avoir accès à toutes les informations concernant une session d'apprentissage. En effet, en général une condition d'arrêt est fonction des résultats courants de l'apprentissage ou bien encore de l'état d'une session d'apprentissage (ex. on demande que l'apprentissage s'arrête en un nombre d'étapes maximal fixé). Pour ce faire, la classe `Sessions_Apprentissage` doit donc disposer de méthodes accesseurs permettant d'obtenir toute information sur ce qui a été fait (information contenue par exemple dans l'historique) et sur l'état actuel de la session. Remarquons qu'une instance d'une condition d'arrêt n'est pas construite *a priori* pour une session d'apprentissage particulière. Ainsi, une même condition d'arrêt peut être utilisée sur deux sessions différentes en cours d'exécution (au moyen de l'opérateur `operator()`).

Dans la version actuelle de <PEpIn>, nous avons défini 5 conditions d'arrêt différentes. Il s'agit de :

- `Conditions_NB_Etapes_Max` : cette condition impose à la session d'apprentissage de s'arrêter au bout d'un nombre d'étapes fixes (que l'apprentissage remplisse ou non les conditions de succès). Ce type de condition permet de vérifier des bornes théoriques sur la complexité en temps d'un algorithme d'apprentissage.

- `Conditions_Cible_Trouvee` : la session d'apprentissage s'arrête lorsque la cible et l'hypothèse sont égales. Ce type de condition permet de paramétrer des sessions d'apprentissage exact.

- `Conditions_NB_Hypo_Id` : lorsqu'un certain nombre d'hypothèses consécutives sont identiques, la session d'apprentissage s'arrête. Ici, on ne compare pas l'hypothèse avec la cible mais les hypothèses entre elles. L'idée qui nous a conduit à définir cette condition est de considérer que lorsque le processus d'apprentissage se stabilise assez longtemps sur une même hypothèse, celle-ci doit être le résultat final.

- `Conditions_Taille_Ech_Bornee` : en général, le nombre d'étapes ne correspond pas au nombre d'exemples différents tirés puisqu'un même exemple peut être présenté plusieurs fois à l'apprenant. Cette condition impose à la session d'apprentissage de s'arrêter lorsqu'un nombre fixé d'exemples différents ont été présentés à l'algorithme.

- `Conditions_Cible_Approchee` : cette condition permet d'implanter le cas de l'apprentissage approximatif puisque la session d'apprentissage s'arrête lorsque l'erreur calculée entre la cible et l'hypothèse est inférieure à une valeur prédéterminée.

L'expérimentateur a la possibilité de paramétrer une session d'apprentissage au moyen de plusieurs conditions d'arrêt. Chaque condition est intégrée à la session au moyen de la méthode `put_cond_arret` de la classe `Sessions_apprentissage`. La session d'apprentissage s'arrête si au moins une condition d'arrêt est vérifiée. C'est la méthode `arrete` de la classe `Sessions_apprentissage` qui est chargée, à la fin de chaque étape, de vérifier s'il faut ou non continuer le processus.

5.6 Conclusion

Ce chapitre, relativement technique, a présenté globalement l'implantation de <PEpIn> qui a été faite par notre équipe. Il faut remarquer que l'ensemble des exigences formulées dans le chapitre précédent a pu être pris en compte, parfois au prix de quelques « acrobaties » techniques. Telle quelle, la plate-forme n'est qu'une librairie de classes pour la plupart abstraites ; elle ne peut donc pas être utilisée seule. Le chapitre suivant présente en détail l'intégration d'un contexte particulier à <PEpIn> afin de construire un logiciel d'expérimentations sur l'inférence.

Chapitre 6

Intégration des Langages Réguliers

Jusqu'à présent, avec <PEpIn>, nous disposons d'une librairie modélisant la théorie de l'apprentissage de manière générique. Ainsi, pour le moment, la plate-forme n'est pas utilisable puisqu'aucun objet tangible n'est associé aux divers éléments définis dans cette librairie. Nous avons déjà abordé succinctement la manière d'intégrer des parties d'un contexte au dessus de <PEpIn> afin de créer une application manipulable. Ce chapitre revient en détail sur ce type d'intégration et présente les différentes étapes que nous avons suivies afin de créer l'application PEpIn<LR>. Cette application est un atelier d'expérimentations de l'inférence inductive dans le cadre des langages réguliers. Dans un premier temps, nous présenterons le modèle conceptuel définissant le contexte des langages réguliers. Ensuite, nous étudierons brièvement l'implantation de ce contexte sous la forme d'une bibliothèque. Enfin, nous aborderons l'intégration du contexte des langages réguliers dans la plate-forme <PEpIn> afin de créer l'application PEpIn<LR>. Cette dernière section sera également l'occasion d'étudier la définition de quelques apprenants.

6.1 Modèle Conceptuel du Contexte des Langages Réguliers

6.1.1 Cahier des charges

La bibliothèque que nous nous proposons de présenter implante un ensemble de classes permettant de représenter et manipuler les langages réguliers sous la forme d'automates finis. Par la suite, nous désignerons cette librairie sous le nom CLR (**C**ontexte des **L**angages **R**éguliers). Cette librairie étant définie indépendamment de la plate-forme <PEpIn>, elle pourra être utilisée à des fins aussi variées que l'étude expérimentale sur les automates, l'intégration dans des logiciels utilisant les langages réguliers (comme par exemple les analyseurs grammaticaux) ou bien encore le développement d'un environnement pour l'enseignement des langages formels. Cette bibliothèque peut donc être comparée à d'autres librairies du même type telles que `Grail` développée par Raymond et Wood [RW94].

Dès lors, il est légitime de se demander pourquoi nous n'avons pas simplement réutilisé une telle librairie déjà existante, d'autant plus que `Grail` a été développée en C++. Nous avons préféré concevoir notre propre librairie pour deux raisons principales : d'une part, bien qu'indépendante de la plate-forme <PEpIn>, cette librairie a pour but premier d'être utilisée en tant que contexte d'apprentissage. Nous devons par conséquent pouvoir retrouver facilement l'ensemble des objets ayant un rôle dans le cadre de l'apprentissage, ce que ne permet

pas `Grail`. D'autre part, nous voulons maîtriser l'intégralité de l'implantation (tant au niveau des structures de données qu'au niveau des algorithmes).

Finalement, la librairie CLR a pour but d'implanter le contexte des langages réguliers avec comme objectif premier d'être facilement intégrable à la plate-forme <PEpIn>. Nous nous limitons à la représentation des langages réguliers sous la forme d'automates finis ; aussi, il ne s'agit pas d'un produit aussi complet que `grail`.

Les mêmes attentes que celles formulées pour la plate-forme <PEpIn> sont exigées ici : modularité, évolutivité, performance... L'ensemble de ces qualités est généralement assuré de par la conception orientée objet du produit. La qualité de performance doit tenir une grande place dans l'implantation de cette librairie. Cette attente sera assurée d'une part au moyen des structures de données mises en œuvre et d'autre part grâce à l'efficacité exigée pour les algorithmes. Enfin, le langage C++ assure quant à lui des performances effectives sur le code généré.

6.1.2 Analyse

La figure 6.1 (page suivante) présente le modèle objet de la librairie CLR. La plupart des objets de base de la théorie des langages réguliers se retrouve dans cette modélisation de manière naturelle. Les relations définies entre les divers objets de ce modèle traduisent exactement les définitions formelles telles que celles présentées à la section 2.1.1 (page 52).

6.1.3 Les classes `Lettres` et `Alphabets`

Un alphabet est un ensemble ordonné de lettres. L'ordre sur les lettres est défini au moyen d'un opérateur de comparaison dans la classe `Lettres`. Il peut sembler superflue de définir une classe `Lettres` ; en effet, la plupart des langages de programmation intègre par défaut un type caractère que l'on pourrait se contenter d'utiliser pour représenter les lettres d'un alphabet. Dans ce cas, l'ordre existant *a priori* sur les caractères définit exactement l'ordre sur les lettres. Cependant, nous préférons redéfinir une telle classe afin de ne pas être limité aux seules lettres ne comportant qu'un unique caractère.

En tant qu'ensemble, nous attendons de la classe `Alphabets` qu'elle intègre un certain nombre d'opérations ensemblistes telles que l'union, l'intersection ou bien encore le test d'inclusion. Ces différentes méthodes ne sont cependant pas définies dans la classe `Alphabets` elle-même mais au niveau d'une classe générique `Ensembles`, mère de la classe `Alphabets`. Nous n'avons pas fait figurer la classe `Ensembles` dans le modèle objet de la figure 6.1 (page suivante), celle-ci n'étant pas, à proprement parler, une composante de base du contexte des langages réguliers mais plutôt une classe annexe.

6.1.4 La classe `Mots`

La classe `Mots` concerne les mots au sens des langages formels. Un mot est une concaténation de longueur finie de lettres appartenant à un alphabet. *A priori*, l'alphabet préexiste toujours à la création d'un mot. Un mot doit-il pour autant mémoriser une référence vers l'alphabet sur lequel il est construit ? Si tel est le cas, alors deux mots u_1 et u_2 identiques (c'est-à-dire tels que $|u_1| = |u_2| = l$ et $\forall i \in [1, l], u_{1i} = u_{2i}$) mais construits à partir de deux alphabets distincts sont représentés par deux objets présentant des caractéristiques différentes. Cette différentiation pour des objets de même valeur n'est pas satisfaisante. De plus, intégrer dans un mot une information sur l'alphabet à partir duquel il a été construit implique

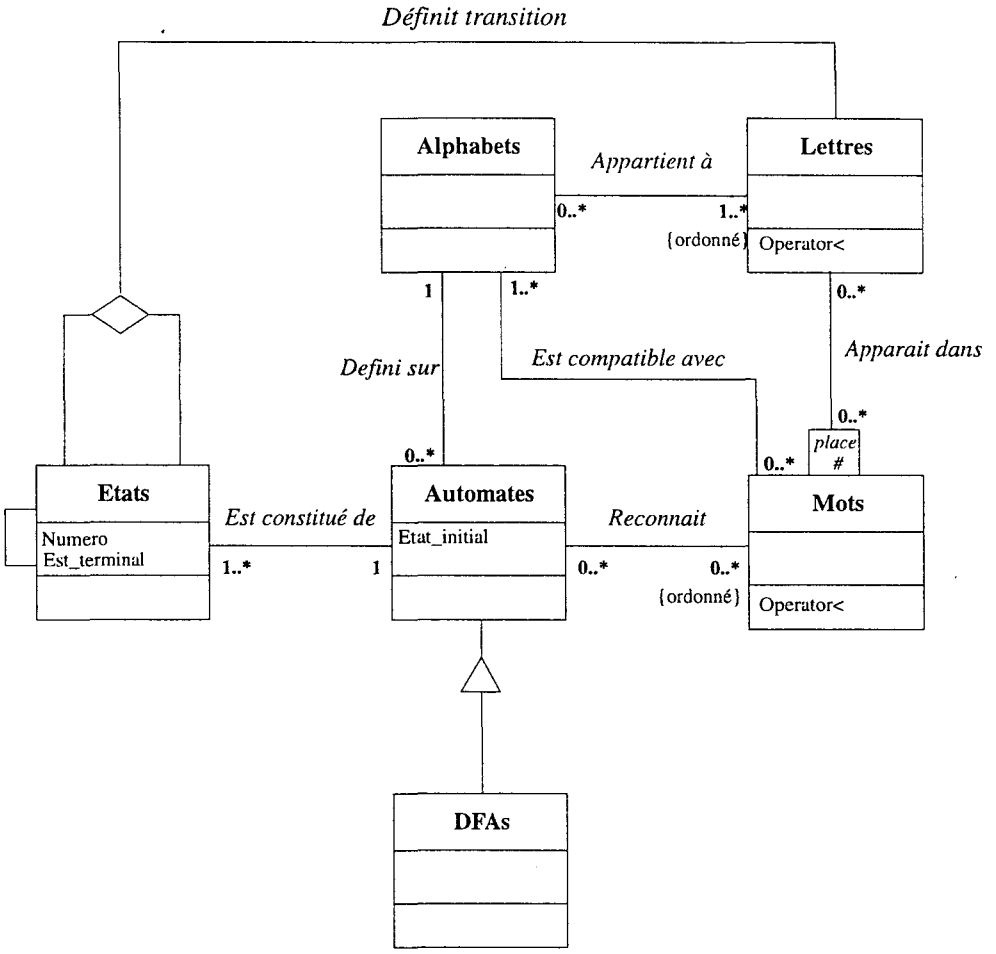


FIG. 6.1 - Modèle objet de la librairie CLR.

de fait une certaine rigidité lors de l'utilisation des mots. Par conséquent, nous choisissons de ne pas mémoriser cette information au niveau des mots. La création d'un mot est donc faite effectivement à partir d'un alphabet préexistant (ce qui permet de vérifier éventuellement l'existence de chacune des lettres composant le mot) mais une fois construit, le mot devient « indépendant » de cet alphabet.

Il convient cependant de pouvoir tester la compatibilité d'un mot avec un alphabet Σ. Ceci peut être fait simplement en construisant, par exemple, l'alphabet canonique Σ_c d'un mot (c'est-à-dire le plus petit alphabet permettant de construire ce mot) et de tester l'inclusion de Σ_c dans Σ.

Un certain nombre d'opérations spécifiques au traitement des mots dans le cadre de la théorie des langages est à prévoir. Citons par exemple le calcul des facteurs d'un mot (*e.g.* le calcul de préfixes ou de suffixes), la concaténation etc....

Dans le cadre de l'apprentissage automatique des langages réguliers, nous avons vu que les mots jouent le rôle d'items. Ne perdant pas de vue que l'utilisation première qui doit être faite de CLR est son intégration dans <PEpIn>, nous devons par conséquent définir un ordre sur les mots, au moyen de l'opérateur <. Rappelons que cet ordre sur les mots permettra de « numéroter » ceux-ci afin, par exemple, de construire les domaines (de mots) munis d'une

distribution de probabilité. L'ordre sur les mots que nous adoptons est classiquement l'ordre standard (ou lexicographique par longueur croissante), c'est-à-dire étant donné un alphabet Σ, u_1 et u_2 deux mots de Σ^*,

$$u_1 < u_2 \text{ si } \begin{cases} |u_1| < |u_2| \text{ ou,} \\ |u_1| = |u_2| = l \text{ et } \exists i \in [1,l] \text{ tel que } Pre_{i-1}(u_1) = Pre_{i-1}(u_2) \text{ et } u_{1i} < u_{2i} \end{cases}$$

6.1.5 La classe `Etats`

La conception orientée objet préconise de définir des classes spécifiques pour chaque entité jouant un rôle dans l'application. Les états sont de tels éléments. Un état n'est pas un objet manipulable de manière isolée mais c'est un constituant de base des objets automates. La classe **`Etats`** est chargée de traiter l'ensemble des fonctionnalités concernant les états. Les états contiennent un champ attribut décrivant l'ensemble des transitions les concernant.

Chaque état est caractérisé au moyen d'un numéro ; remarquons que ce numéro d'état n'a de sens qu'au niveau de l'automate lui-même. La vérification de l'unicité lors de l'attribution des numéros aux états est, par conséquent, laissée à la charge de l'automate ; en effet, seul l'automate a une vision globale des états qui le constituent.

C'est également au niveau de l'état que se situe l'information de terminalité ; il s'agit simplement d'un champ booléen prenant la valeur vraie si l'état est terminal.

6.1.6 La classe `Automates`

Par définition, un automate est un quintuplet $A = (Q, I, F, \Sigma, \delta)$. Étant donné qu'un certain nombre d'informations (terminalité, transition) est disponible au niveau des états, le profil de la classe **`Automates`** ne correspond pas exactement à la définition théorique. De plus, nous nous limitons aux automates à un seul état initial. La classe **`Automates`** contient donc trois attributs principaux qui sont : l'ensemble d'états, le numéro de l'état initial et l'alphabet de base. Insistons sur le fait que les informations concernant l'ensemble des états terminaux et les transitions sont disponibles *via* l'ensemble d'états.

La classe **`Automates`** permet de construire des automates généraux c'est-à-dire déterministes ou non. La plupart des manipulations sur les automates sont à intégrer dans cette classe. Citons par exemple l'union de deux automates, l'intersection de deux automates, la destruction des états puits ou inaccessibles, la fusion de plusieurs états ou bien encore le test de propriétés telles que le déterminisme ...

Lors de l'intégration de CLR à la plate-forme <PEpIn>, les automates joueront le rôle des représentations de concepts. Il est donc indispensable de prévoir la définition de méthodes qui permettront, par la suite, de définir les méthodes virtuelles **`contient`**, **`egal`** et **`taille`** (voir section 5.2.3, page 145). À ce niveau, ces trois fonctionnalités sont déclarées sous les noms respectifs **`reconnait`**, **`est_isomorphe`** et **`nombre_etats`**.

Afin d'assurer le mieux possible la préservation de la propriété de déterminisme sur certains automates, nous définissons une nouvelle classe ne traitant que les automates déterministes. Cette classe (décrite par la suite) propose certaines fonctionnalités utilisables uniquement sur ce type d'automates. En résumé, nous devons considérer les instances de la classe **`Automates`** comme des automates généraux, sans propriétés particulières et ne proposant donc pas l'ensemble des opérations disponibles sur les automates déterministes.

6.1.7 La classe DFAs

La classe DFAs hérite logiquement de la classe Automates. Cette classe assure que chacune de ses instances est effectivement un automate déterministe. Certaines opérations telles que la fusion d'états ne conservant pas obligatoirement la propriété de déterminisme, celles-ci doivent par conséquent être redéfinies à ce niveau (c'est-à-dire en les *surchargeant*) afin de garantir la conservation du déterminisme (par exemple, en interdisant les cas de non-conservation).

Les opérations définies uniquement pour les automates déterministes sont déclarées dans la classe DFAs. La minimalisation, par exemple, n'est pas disponible pour la classe Automates alors qu'elle l'est pour la classe DFAs.

Finalement, la classe DFAs n'est rien d'autre qu'une classe d'automates enrichies en fonctionnalités et garantissant la conservation de la propriété de déterminisme.

6.1.8 Conception de la librairie

Un certain nombre de classes annexes est nécessaire afin d'implanter le plus proprement possible le modèle objet que l'on vient de présenter ; parmi celles-ci figure la classe Ensembles que nous avons déjà évoquée précédemment. Détailler l'ensemble de ces éléments ici ne présente pas grand intérêt. Aussi, nous nous contentons de présenter succinctement les structures de données sur lesquelles repose la classe principale de CLR : Automates.

Nous rappelons deux impératifs attendus des objets automates : les traitements doivent être efficaces et la manipulation des automates depuis l'extérieur doit être naturelle. La deuxième attente impose, par exemple, que pour désigner un état particulier de l'automate on utilise son numéro ou bien encore que pour référencer une transition on utilise la représentation (i, l, j) avec i et j des numéros d'états et l une lettre de l'alphabet. La condition d'efficacité, quant à elle, oblige à utiliser des techniques d'accès aux éléments (tels que les états) très efficaces, au moyen de pointeurs par exemple. Il semble donc que ces deux contraintes soient incompatibles. Pour pallier ce problème, nous imposons par conséquent que toute opération accessible depuis l'extérieur se fasse au moyen du premier type de représentation, clair et naturel. Les manipulations internes se font quant à elle directement sur les structures de données au moyen d'accesseurs efficaces.

Les ensembles ordonnés servent non seulement pour la définition de la classe Alphabets mais aussi pour construire d'autres types d'ensembles comme les ensembles d'états. Les automates étant éventuellement non déterministes, pouvoir manipuler des ensembles d'états apporte une grande souplesse dans le traitement des algorithmes. D'après ce qui vient d'être dit dans le paragraphe précédent, il est donc envisageable de traiter non seulement des ensembles de numéros d'états, mais aussi des ensembles de pointeurs d'états.

Structures de données relatives aux états

Nous étudions ici de quelle manière une instance de la classe Etats mémorise ses transitions. Les états sont, nous le rappelons, des objets qui ne sont manipulables qu'au travers d'un automate : il n'est pas concevable de pouvoir construire et utiliser un état en dehors d'un automate. En définitive, seuls les automates opèrent directement sur les états. Les structures de données sur lesquelles sont construits les états privilégient par conséquent l'efficacité à la « clarté ».

De nombreuses opérations sur les automates demandent non seulement à utiliser les transitions de manière classique mais aussi de manière réciproques c'est-à-dire de l'état destination

vers l'état origine. Afin d'assurer une efficacité maximale pour ce type d'opérations, chaque état mémorise non seulement les transitions dont il est l'origine mais aussi les transitions pour lesquelles il est destination. De plus, la classe `Automates` permet de créer des automate non déterministes ; par conséquent, pour chaque lettre, il peut exister plusieurs transitions pour lesquelles un état est origine.

Nous proposons de mémoriser les transitions relatives à un état q au moyen de deux tableaux (un pour représenter les transitions dont l'état q est origine, l'autre pour mémoriser celles dont il est destination). Chacun de ces tableaux contient autant de cases que l'alphabet de base contient de lettres[1]. Dans chacune des cases de numéro i, on trouve alors l'ensemble des (pointeurs d') états d'arrivée (resp. de départ) pour les transitions depuis (resp. vers) l'état q et labellées par la i^e lettre de l'alphabet.

C'est à la charge de chaque état de maintenir à jour ces deux tableaux. Remarquons cependant qu'une modification de transition dans un état entraîne la plupart du temps la modification réciproque dans un autre état. Il convient donc de maintenir la consistance de l'ensemble des transitions d'états, ce qui est assuré par l'automate lui-même.

Structures de données relatives aux automates

Il ne reste plus qu'à porter notre attention sur les structures de données sur lesquelles s'appuie la classe `Automates`.

Il a déjà été précisé que la classe `Automates` contient des attributs référençant l'etat initial, ainsi que l'alphabet de base. Il ne reste donc plus qu'à étudier de quelle manière le corps de l'automate, c'est-à-dire l'ensemble des états et des transitions, est structuré.

En fait, étant donnée l'intégration des transitions au niveau des états, il est suffisant de mémoriser l'ensemble des états composant un automate. Ceci est fait simplement au moyen d'un ensemble ordonné (de pointeurs) d'états. Par l'intermédiaire des pointeurs d'états, il est possible d'accéder rapidement à un état donné. L'accès à un état connu uniquement par son numéro est quant à lui facilité par le fait que l'ensemble définissant le corps de l'automate est ordonné selon les numéros d'états.

6.2 Implantation de la Librairie des Langages Réguliers

Nous abordons dans cette partie l'implantation à proprement parler de la librairie CLR. Celle-ci a été faite en C++ et ce pour les mêmes raisons que celles nous ayant conduits à implanter <PEpIn> dans ce langage. Une fois de plus, il n'est pas question d'étudier de manière exhaustive l'ensemble des classes et des méthodes que nous avons définies ; la librairie CLR est en effet composée (pour le moment) de plus d'une quinzaine de classes différentes et d'environ 350 méthodes distribuées sur ces classes. Aussi, nous ne présentons que quelques unes de ces classes.

6.2.1 Les ensembles

Nous avons déjà parlé de l'importance des ensembles pour l'implantation de notre librairie : d'une part, il s'agit de la classe à partir de laquelle est définie la classe `Alphabets` et d'autre part les ensembles sont des structures de base pour représenter les automates.

1. Ceci montre une fois de plus l'étroite dépendance existant entre les automates et les états.

Les classes définissant les ensembles restent malgré tout des classes annexes à la librairie CLR qu'il n'était pas utile d'intégrer au modèle objet. La librairie `STL` que nous utilisons également dans l'implantation de CLR propose une classe implantant les ensembles. Cependant, celle-ci ne nous convenant pas pour diverses raisons, nous avons choisi de la redéfinir.

Les ensembles sont implantés sous la forme de deux classes génériques ; le paramètre générique représentant le type des objets éléments de l'ensemble. Au niveau interne, les ensembles ne sont rien d'autre que des listes doublements chaînées (fournies par la librairie `STL`) ; cette structure de liste présente l'avantage de faciliter l'insertion et la suppression d'éléments dans les ensembles. La première de ces classes, `F_Sets`, permet de construire des ensembles non obligatoirement ordonnés. La deuxième classe, `OF_Sets`, implante quant à elle les ensembles ordonnés ; les éléments d'un tel ensemble doivent, par conséquent, pouvoir être comparés. Il est évident que la classe `OF_Sets` hérite de la classe `F_Sets` en surchargeant un grand nombre de méthodes de celle-ci afin de prendre en compte la propriété d'ordre (ce qui optimise bien souvent les traitements qui suivront).

Une instance de l'une ou l'autre de ces classes peut être soit un ensemble classique (c'est-à-dire dans lequel chaque élément n'apparaît qu'une seule fois) ou bien un multi-ensemble (dans lequel un même élément peut apparaître plusieurs fois). Le choix du caractère ensemble ou multi-ensemble est fait à la construction de l'objet. Cette manière de décrire les ensembles permet, entre autre, de pouvoir construire facilement un ensemble à partir d'un multi-ensemble (et inversement) sans pour autant multiplier le nombre de classes[2].

Lors de l'ajout d'un élément à un ensemble `E`, la complexité en temps de l'opération peut varier en fonction de `E`. Le tableau ci-dessous présente, dans chacun des cas, la complexité maximale de cette opération (`|E|` représente le nombre courant d'éléments de l'ensemble `E`):

Classe de E	Ensemble classique	Multi-ensemble
F_Sets	$O(\|E\|)$	$O(1)$
OF_Sets	$O(\|E\|)$	$O(\|E\|)$

D'après ce tableau, il semble donc que travailler avec des ensembles ordonnés n'optimise pas les traitements. Cependant, l'insertion dans un ensemble ou un multi-ensemble ordonné est, en moyenne, plus rapide que dans un ensemble classique non ordonné. De plus, rappelons que de nombreux traitements (autres que l'insertion) dans un ensemble ordonné peuvent être optimisés en utilisant la propriété d'ordre.

Le profil (incomplet) de la classe `F_Sets` est présenté ci-dessous. Il est inutile de préciser que la plupart des opérations ensemblistes a été implantée pour ces deux classes. Lorsque cela a un sens, nous utilisons la définition sous forme d'opérateur pour définir celles-ci (*e.g.* `operator+` définit l'union entre deux ensembles et `operator*` définit l'intersection). Le profil de la classe `OF_Sets` est très similaire à celui de `F_Sets`.

```
template<class T>

class F_Sets: public list<T> {
protected:

  bool is_set;

  virtual inline iterator cherche_place(const T& elem);
```

2. Ce qui d'ailleurs poserait un réel problème lié à l'héritage multiple dit *en losange*.

```
public:

  /* ------------- */
  /* Constructeurs */
  /* ------------- */

  F_Sets(bool real_set = true);

  F_Sets(const F_Sets<T>& e, bool real_set);

  F_Sets(const F_Sets<T>& e);

  /* --------------------------------- */
  /* Insertion et destruction d'element */
  /* --------------------------------- */

  pair_iterator_bool insert(const T& elem);

  void del(const T& elem);

  inline void vide(void);

  /* ------------------------------- */
  /* Acces aux elements de l'ensemble */
  /* ------------------------------- */

  const T& operator[](unsigned int i) const;

  /* --------------------- */
  /* Methodes ensemblistes */
  /* --------------------- */

  inline unsigned int card(void) const;

  virtual bool est_inclus_dans(const F_Sets<T> &e) const;

  virtual F_Sets<T> operator+(const F_Sets<T> &e) const;

  virtual F_Sets<T> operator-(const F_Sets<T> &e) const;

  virtual F_Sets<T> operator*(const F_Sets<T> &e) const;

  virtual bool contient(const T& elem) const;
};
```

Nous ne présentons pas la classe `Alphabets` dans la mesure où celle-ci peut-être vue comme un ensemble ordonné dont le paramètre générique est du type `Lettres`.

6.2.2 Implantation de la classe `Lettres`

La classe `Lettres` est implantée entre autre afin de pouvoir définir des lettres « complexes » c'est-à-dire représentées non par un unique caractère mais par une chaîne de caractères.

Cependant, nous nous sommes limités dans un premier temps aux lettres constituées d'un seul caractère. Cette restriction nous évite d'implanter un petit analyseur syntaxique qui serait indispensable lors de la saisie de lettres complexes. Nous envisageons d'enrichir par la suite cette classe afin de répondre au cahier des charges initial. Cette évolution sera prise en compte par l'ensemble des classes de la librairie avec un minimum de modifications puisque la plupart de celles-ci ne portera qu'au sein de la classe `lettres` elle-même[3]

Pour le moment, la classe `Lettres` est donc réduite à sa plus simple expression puisqu'il ne s'agit que d'une encapsulation du type `char` (voir profil ci-dessous).

```
classe Lettres {
private :

  char la_lettre;

public :

  /* ------------- */
  /* Constructeurs */
  /* ------------- */

  Lettres(char c);

  /* ------------ */
  /* Comparateurs */
  /* ------------ */

  bool operator==(const Lettres& l) const;
  bool operator==(char c) const;

  bool operator!=(const Lettres& l) const;
  bool operator!=(char c) const;

  bool operator<(const Lettres& l) const;
  bool operator<(char c) const;

  bool operator<=(const Lettres& l) const;
  bool operator<=(char c) const;

  bool operator<(const Lettres& l) const;
  bool operator<(char c) const;

  bool operator>(const Lettres& l) const;
  bool operator>(char c) const;
```

6.2.3 Implantation de la classe `Mots`

L'implantation de la classe `Mots` ne pose aucune difficulté s'agissant à la base d'une classe relativement classique. La classe `Mots` est développée au dessus d'une classe annexe `Cl_Strings` ; celle-ci a pour objectif de traiter les chaînes de caractères de longueur non bornée. Il s'agit par conséquent d'une classe relativement générale et utile dans beaucoup d'autres

3. *A priori*, la classe `Mots` devra également subir quelques transformations.

contextes. Elle propose diverses méthodes telles que la concaténation de deux chaînes, la construction de sous-chaîne etc...

La classe `Mots` utilise donc la classe `Cl_Strings` pour mémoriser les mots sous forme de chaînes de caractères classiques. De plus, avoir défini une classe `Mots` au dessus de la classe `Cl_Strings` permet de distinguer les caractères des lettres ; ainsi, la classe `Cl_String` ne manipule que des caractères, la classe `Mots` quant à elle manipule des lettres[4]. La classe `Mots` enrichit `Cl_Strings` en implantant des méthodes spécifiques aux mots, c'est-à-dire des opérations bien définies dans le cadre de la théorie des langages (par exemple, le test de compatibilité avec un alphabet, ou encore le calcul de préfixes etc...). On implante également un opérateur de comparaison sur les mots ; rappelons que dans le cadre de l'apprentissage, les mots sont appelés à jouer le rôle d'items et qu'il est donc indispensable qu'un tel opérateur existe. Enfin, la classe `Mots` déclare un mot particulier représentant le mot vide. Celui-ci a une longueur nulle et l'utilisation de celui-ci dans les diverses méthodes suit les comportements classiques de la théorie des langages (par exemple: $\varepsilon u = u$).

Le profil de la classe `Mots` a la forme suivante.

```
class Mots : private Cl_String {
public:

  /* ---------------------- */
  /* Definition du mot vide */
  /* ---------------------- */
  static const Mots Mot_Vide;

  /* ------------- */
  /* Constructeurs */
  /* ------------- */

  Mots(void);
  /* Construit le mot vide, c'est-a-dire tel que la longueur soit nulle. */

  Mots(const Lettres& l);
  /* Construit le mot de longueur 1 et ne contenant que la lettre <l>. */

  Mots(const Mots& m);

  /* ------------------------ */
  /* Operateurs d'affectation */
  /* ------------------------ */

  Mots& operator=(const Mots& m);

  /* ---------- */
  /* Accesseurs */
  /* ---------- */

  inline unsigned int longueur() const;

  /* ---------------------------- */
```

4. Précisons que dans la version actuelle, la classe `Mots` permet également de manipuler des caractères ; ceci ne sera plus possible dans les versions prochaines lorsque la classe `Lettres` sera complétement développée.

```
  /* Operateur d'acces a une lettres */
  /* ------------------------------- */

  Lettres& operator[](unsigned int i) const;

  /* ------------------------------- */
  /* Operateurs de comparaison de mots */
  /* ------------------------------- */

  bool operator<(const Mots& m) const;

  bool operator==(const Mots& m) const;

  /* ------------------------------------- */
  /* Creation d'un mot a partir d'un autre mot */
  /* ------------------------------------- */

  Mots facteur(unsigned int pos,unsigned int l) const;

  inline Mots prefixe(unsigned int l) const;

  Mots suffixe(unsigned int l) const;

  /* ---------------------------------- */
  /* Operateurs de concatenation de mot */
  /* ---------------------------------- */

  inline Mots operator+(const Mots& m) const;

  inline Mots operator+(Lettres c) const;

  /* ---------------------------------- */
  /* Fonctions de decisions sur les mots */
  /* ---------------------------------- */

  bool est_compatible_avec(const Alphabets &a) const;

  int est_facteur_de(const Mots& m) const;

  bool est_prefixe_de(const Mots& m) const;

  bool est_suffixe_de(const Mots& m) const;
};
```

6.2.4 Implantation des classes d'automates

Les deux dernières classes que nous présentons implantent les automates. Il s'agit des classes `Auto_Gene` correspondant aux automates généraux (c'est-à-dire non obligatoirement déterministes) et `DFAs` se rapportant exclusivement aux automates déterministes.

Il n'est pas utile de présenter une fois de plus les structures de données sur lesquelles sont construits les automates (voir section 6.1.8, page 166).

La classe `Auto_Gene` est une classe très riche en méthodes. Un certain nombre de méthodes

disponibles depuis l'extérieur demande des arguments sous forme compréhensible (par exemple des numéros d'états ou bien encore des lettres) ; ces fonctions appellent alors des méthodes équivalentes mais privées (*i.e.* inaccessibles en dehors de la classe) qui traitent les structures de manière plus efficace.

L'ensemble des méthodes fournies par la classe `Auto_Gene` est utilisable aussi bien sur des automates déterministes que sur des automates non-déterministes. Celles-ci ne présentent, en général, pas de difficultés algorithmiques ou bien les algorithmes qu'elles implantent sont bien connus (voir par exemple [HU79]).

La classe `DFAs` hérite de la classe `Auto_Gene`. Cette classe présente les mêmes principes que sa classe mère si ce n'est que l'on assure que toute opération sur une instance de cette classe conserve le déterminisme de l'objet automate sur lequel elle s'applique[5]. Des algorithmes, tels que la minimalisation d'automates, sont implantés à ce niveau car ils n'ont de sens que pour les DFAs. Ici encore, les méthodes sont basées sur des algorithmes connus. Précisons que certaines méthodes définies dans la classe `Auto_Gene` sont surchargées dans la classe `DFAs` afin de prendre en compte la propriété de déterminisme et d'optimiser certains traitements.

Ci-dessous, nous ne proposons qu'une petite partie des profils des deux classes `Auto_Gene` et `DFAs` étant donnée la richesse de ces deux classes.

```
class Auto_Gene{

protected:

  Alphabets Alpha_Base;
  Ptr_States Etat_Initial;
  typedef OF_Sets<Etats> Corps;
  Corps Le_Corps;

  /* ---------------------------- */
  /* Constructeurs et Destructeur */
  /* ---------------------------- */

  Auto_Gene(const Alphabets &a,Numeros n = 1);
  /* Cree un automate a <n> etats base sur l'alphabet <a>.
     Etat initial = etat 0. */

  Auto_Gene(const Auto_Gene &a): Alpha_Base(a.Alpha_Base);
  /* Cree un automate qui est la copie conforme de <a> */

  Auto_Gene(const Auto_Gene& a, const States_Partitions& pi);
  /* Construit un nouvel automate qui est l'automate derive de <a> par <pi>
     c'est-a-dire <a>/<pi>. La partition <pi> concerne l'automate <a>, tous
     les etats sont references relativement a <a>. */

  Auto_Gene(istream& flot);
  /* Construit un nouvel automate a partir de la description
     faite dans le fichier d'entree <flot>. */

  virtual ~Auto_Gene(void);
```

5. Si ce n'est pas le cas, une erreur est déclenchée.

```
  /* Destructeur de l'automate. Les etats, proprietes de l'automate, sont
     egalement detruits. */
};

class DFAs : public Auto_Gene {

protected:

  /* ---------------- */
  /* Methodes privees */
  /* ---------------- */

public:

  /* ------------- */
  /* Constructeurs */
  /* ------------- */

  DFAs(const Alphabets &a);

  inline DFAs(const Alphabets &a, Numeros n);
  /* Cree un automate deterministe a <n> etats base sur l'alphabet <a>. */

  inline DFAs(const DFAs &a);
  /* Cree un automate deterministe qui est la copie conforme de <a> */

  DFAs(const Auto_Gene &a);

  inline DFAs(istream& flot);
};
```

Remarquons que dans notre implantation, tout automate contient son propre alphabet de base. On aurait pu imaginer qu'une simple référence sur un tel objet suffisait. Cependant, dans ce cas, il aurait été obligatoire de déclarer tout alphabet en tant que constante afin d'éviter des incohérences en cas de modification de l'alphabet après qu'un automate ait été construit dessus. Notons également que l'alphabet de base est connu dès la construction d'un automate. Ceci est obligatoire afin de connaître la taille de celui-ci et ainsi de construire les structures de données sous-jacentes ; rappelons que les tableaux de transitions attachés aux états contiennent autant de cases que la cardinalité l'alphabet. L'alphabet est utilisé au niveau interne afin de passer d'une représentation en clair d'une lettre à son indice dans l'alphabet.

La classe `DFAs` fournit un constructeur dont l'argument `a` est un automate quelconque du type `Auto_Gene`. Ce constructeur crée un nouvel automate déterministe reconnaissant exactement le même langage que `a`. L'algorithme de déterminisation est par conséquent utilisé par ce constructeur.

Enfin, il convient une fois de plus de considérer quel sera le ou les rôles joués par les automates dans le cadre de l'apprentissage. D'abord, évidemment, un automate est une représentation d'un concept langage régulier. Nous rappelons qu'il convient donc d'implanter certaines méthodes qui permettront par la suite de créer une classe `Concepts_Automates` (voir section 6.1.6, page 164).

Un autre rôle pouvant être tenu par les automates est celui de domaine. En effet, un

domaine étant un ensemble d'items, dans le contexte des langages il s'agit par conséquent d'un ensemble de mots, soit un langage. Il est donc tout à fait justifié de représenter un domaine au moyen d'un automate (dans le cas où ce domaine est un langage régulier). Notons que dans la majorité des cas, le domaine utilisé lors l'apprentissage de langages est Σ^*, avec Σ un alphabet ; le domaine peut dans ce cas être implanté au moyen de l'automate déterministe universel reconnaissant Σ^*. Dans l'optique d'utiliser les automates en tant que domaine, nous devons donner à ces objets un pouvoir de génération d'items (c'est-à-dire de mots). Pour ce faire, nous intégrons à la classe `Auto_Gene` les deux méthodes `premier_mot` et `mot_suivant` qui retournent respectivement le premier mot reconnu par l'automate considéré et le mot suivant le mot passé en argument (selon l'ordre standard sur les mots). Étant donné le possible non-déterminisme de l'automate, la complexité en temps de la méthode `mot_suivant` peut être élevée. Ce défaut est corrigé dans la classe `DFAs`. En effet, ces deux méthodes utilisent la propriété de determinisme de l'automate sur lequel elles agissent afin de retourner plus rapidement le résultat. Une troisième méthode de génération de mots est disponible dans la classe `DFAs` : `n_mot_suivant`. Cette méthode a pour fonction de retourner le n^e mot suivant le mot passé en argument et reconnu par l'automate. Cette méthode est telle que l'utiliser pour générer le n^e mot suivant un mot m est plus efficace que d'appeler n fois la méthode `mot_suivant`. Le principe de l'algorithme est donné à la figure 6.2 (page suivante) (nous nous plaçons dans le cas où le langage reconnu par l'automate A est infini).

Nous ne détaillons pas le principe de cet algorithme qui reste assez technique. Signalons simplement que le langage reconnu par A' étant fini, la génération des différents mots reconnus par celui-ci peut être implantée de façon efficace (en mémorisant la distance minimale entre chaque état et l'état terminal le plus proche). La construction de l'automate A'' peut s'appuyer sur l'automate A' déjà construit afin d'éviter des calculs déjà faits. Enfin, la génération du premier mot reconnu par un automate déterministe à p états peut être implantée en temps $O(p)$ si cet automate ne contient aucun états puits.

Enfin, il est important de constater que dans le cas des automates universels, la méthode `n_mot_suivant` peut encore être optimisée puisqu'alors, il est possible de générer directement le mot de numéro n simplement en « traduisant » n dans la base $|\Sigma|$. Nous choisissons alors de définir une nouvelle classe d'automates, appelée `Autos_Universels`, dérivant de la classe `DFAs` dans le seul but de surcharger la méthode sus-citée. Le profil de cette dernière classe est :

```
class Autos_Universels: public DFAs {
public:

  Autos_Universels(const Alphabets &a);
  /* Cree l'automate universel base sur l'alphabet <a> */

  virtual Mots n_mot_suivant(const Mots& m, NumerosMots num = 1) const;
};
```

Ces optimisations trouveront leur intérêt dans la suite, lorsque nous évoquerons le tirage aléatoire de mots dans un domaine (voir section 6.3.2, page 177).

Algorithme `n_mot_suivant`
Entrée : A un automate, m un mot, n
Début
$m_c = m$
$i = 0$
Tantque $i < n$

`* On recherche d'abord le mot parmi les mots`
`de même longueur que celle de` m_c
Construire l'automate A' reconnaissant le langage $L(A) \cap \Sigma^{|m_c|}$
Tantque $i < n$
$m_c = Suivant(A', m_c)$
$i = i + 1$
Ftq
Si $i < n$
Alors `* On passe a une longueur superieure`
Construire l'automate A'' reconnaissant le langage $L(A)^{>|m_c|}$
$m_c = Premier(A'')$
$i = i + 1$
Fsi
Ftq
Retourner m_c
Fin

FIG. 6.2 - Algorithme de génération du n^e mot suivant un mot.

6.3 Intégration de CLR à <PEpIn>

Nous abordons dans cette dernière section l'intégration de la librairie CLR à la plate-forme <PEpIn> afin de créer un atelier d'expérimentations de l'inférence inductive des langages réguliers. Cette mise en relation des deux librairies suit la démarche déjà évoquée dans les deux chapitres précédents.

6.3.1 Mise en relation des deux librairies

Dans un premier temps, il s'agit de trouver quels objets de CLR joueront les rôles clés dans l'apprentissage. Pour une question de simplicité et d'efficacité, nous choisissons de ne considérer que les automates déterministes. Tout langage régulier étant reconnaissable par un DFA, cela ne diminue en rien les potentialités de notre application. La classe `Auto_Gene` n'est donc pas directement utilisée dans <PEpIn>, bien que celle-ci soit la classe mère des classes d'automates[6]. La classe `DFAs` joue donc à la fois le rôle de représentations de concepts et de domaines. La classe `Mots` sera utilisée pour représenter les items. Insistons sur le fait que ni

6. On peut considérer que la création de la classe `Auto_Gene` apporte un plus au niveau génie logiciel afin de rendre la librairie CLR réutilisable dans d'autres situations.

la classe `DFAs` ni la classe `Mots` ne sont vues directement comme la classe des concepts ou des items. Il s'agit simplement de la représentation de ces deux éléments ; ces classes permettent donc seulement de créer les véritables classes de concepts et d'items dans le contexte des langages réguliers (voir sections 5.2 (page 142) et 5.2.3 (page 145)).

Trois nouvelles classes sont définies. Il s'agit des classes `It_Mots`, `Domaines_Mots` et `Concepts_DFAs`. La figure 6.3 présente (en partie) de quelle manière la librairie CLR est intégrée à <PEpIn> (les classes de couleur blanche sont issues de <PEpIn>, les classes en gris clair de CLR et les nouvelles classes apparaissent en gris foncé).

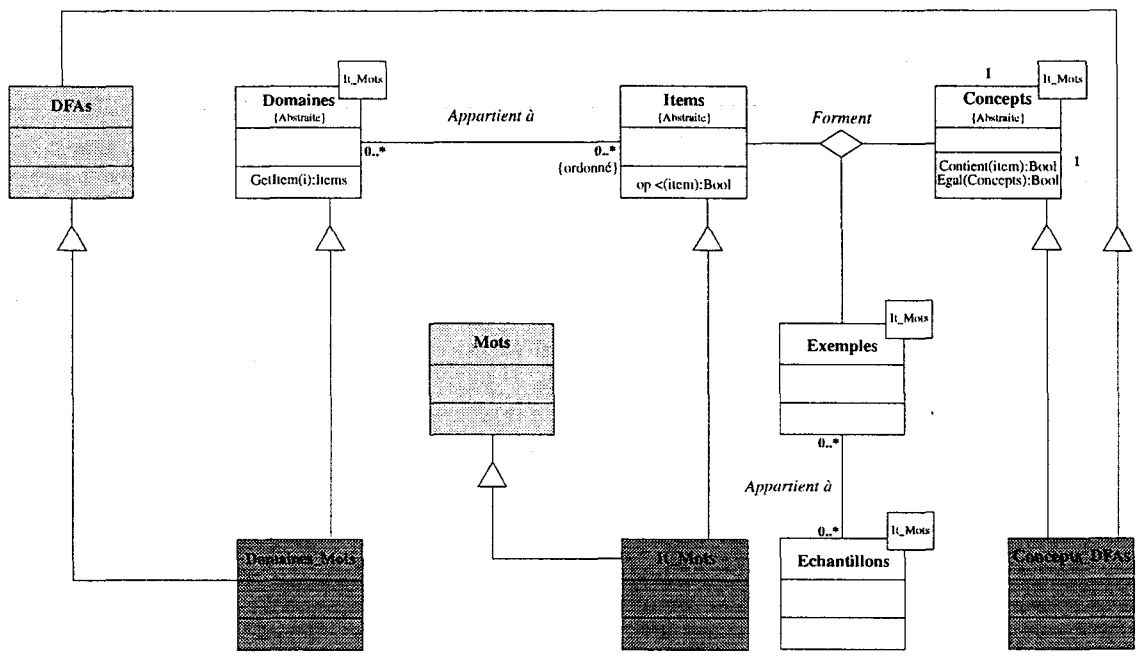


FIG. 6.3 - Intégration de la librairie CLR à la plate-forme <PEpIn>.

Il ne semble pas utile de détailler de quelle manière les nouvelles classes sont définies au niveau de l'implantation. Ces classes sont relativement pauvres puisque l'ensemble de leurs fonctionnalités existe au niveau de leur(s) classe(s) mère(s). Elles ne proposent donc que les constructeurs indispensables ainsi qu'éventuellement la surcharge des méthodes virtuelles de <PEpIn> afin d'appeler les méthodes adéquates de la classe mère issues de CLR. À titre d'exemple, nous proposons le profil de la classe `It_Mots` :

```
class It_Mots: public Items<Mots>, public Mots {

public:

  /* ------------- */
  /* Constructeurs */
  /* ------------- */

  It_Mots(void): Mots()
  /* Construit l'item mot vide */
  {};
```

```
  It_Mots(const Lettres& l): Mots(l)
  /* Construit l'item mot de longueur 1 et ne contenant que la lettre <l>. */
  {};

  Mots(const It_Mots& m): Mots(m)
  /* Constructeur de copie */
  {}

  /* ------------ */
  /* Comparateurs */
  /* ------------ */

  virtual bool operator<(const Mots& item) const
  {
    return this->Mots::operator<(item);
  }

  virtual bool operator==(const Mots& item) const
  {
    return this->Mots::operator==(item);
  }
};
```

6.3.2 Génération aléatoire de mots

Nous avons vu précédemment que les domaines de mots sont implantés sous la forme d'automates. Rappelons que les domaines munis d'une distribution de probabilité sont utilisés par les oracles afin de générer aléatoirement des exemples.

Constatons d'abord que dans le cas où l'on crée un oracle d'exemples positifs pour un automate cible A, il est tout à fait possible de demander à cet oracle de tirer les items dans le domaine A, ce qui assure que le tirage d'exemples positifs se fait en une seule étape.

Dans bien des cas, le domaine utilisé dans l'inférence grammaticale est Σ^*. Il existe une classe d'automates particulière `Autos_Universels` implantant les automates reconnaissant ces langages et fournissant une méthode très efficace de génération du n^e mot de Σ^*. Ainsi, on assure que la génération des mots ne sera pas un obstacle à l'efficacité attendue de la part des applications. Rappelons que le nombre d'exemples à tirer est souvent très important, l'efficacité en temps de ce processus doit par conséquent être maximale. Cependant, dans certaines expériences, il peut être intéressant de restreindre le domaine à un langage plus petit que Σ^*, reconnu par un automate A quelconque. Tels que définis jusqu'à présents, les oracles génèrent aléatoirement un entier suivant la distribution de probabilité puis demandent au domaine de retourner l'item de numéro d'ordre cet entier. Dans le cas des langages réguliers, la méthode `getitem(n)` de la classe `Domaine_Mots` appelle simplement la méthode `n_mot_suivant` avec comme paramètre le premier mot du reconnu par l'automate et n. Cependant, bien que relativement efficace, une telle utilisation répétée de cette méthode dégrade fortement l'efficacité du processus. En effet, à chaque tirage de nouveau mot, le processus énumère tous les mots dont le numéro est compris entre 1 et n. La méthode `n_mot_suivant` permet pourtant de générer directement le n^e mot suivant un mot reconnu par l'automate.

Optimisation pour générer un mot

L'idée pour optimiser quelque peu le processus de génération aléatoire de mots est de mémoriser tout ou une partie des mots déjà tirés. Dans le contexte des langages, nous enrichissons par conséquent la classe `Oracles<Mots>` en la dérivant en une nouvelle classe disons `Oracle_Mots`. Un oracle de ce type procède exactement de la même manière que les oracles classiques à la différence que chaque mot généré est mémorisé par l'oracle lui même avec son numéro d'indice dans le domaine. Lorsqu'il doit tirer un nouveau mot, l'oracle génère aléatoirement le numéro n du mot à retourner. Il consulte alors sa table de mots déjà sortis afin de vérifier que le mot numéro n n'a pas déjà été tiré. Si c'est le cas, il retourne directement ce mot. Sinon, il retrouve dans sa table le mot m dont le numéro est le plus grand entier inférieur à n disons n'. La génération du mot de numéro n se fait alors par l'appel à la méthode `n_mot_suivant` avec comme paramètres m et $n' - n$. Par ce moyen, nous avons pu effectivement constater un gain réel d'efficacité.

Distributions de probabilité particulières

La génération aléatoire des items dans un domaine se fait, nous l'avons vu, suivant une distribution de probabilité attachée à ce domaine. Nous étudions ici une distribution de probabilité particulière que nous avons implantée et qui semble particulièrement bien adaptée à la génération aléatoire de mots. Rappelons d'abord que la plate-forme <PEpIn> propose, à la base, une classe implantant la distribution de probabilité uniforme entre deux entiers a et b (voir section 5.3.2, page 148).

Sur le domaine Σ^*, cette distribution affecte la même probabilité à tous les mots dont l'indice (selon l'ordre lexicographique par longueur croissante) est compris entre a et b. Par exemple, en prenant $a = 1$ et $b = 5$, les mots $\varepsilon, 0, 1, 00, 01$ ont chacun une probabilité de $1/5$, les autres mots de Σ^* ont, quant à eux, une probabilité nulle.

Nous définissons une autre distribution de probabilité sur le domaine des mots dont le but est, dans certain cas, de favoriser le tirage des mots courts. Plus précisément, cette distribution de probabilité affecte la même probabilité à tous les mots d'une même longueur et la probabilité de tirer n'importe quel mot de longueur l est d'autant plus grande que la valeur de l est petite. Remarquons qu'une telle distribution de probabilité, à la différence de la distribution uniforme, affecte une valeur non nulle à tous les mots du domaine Σ^*, ce qui présente l'avantage de ne pas être restreint à une partie de Σ^*.

Une première idée pour implanter une telle distribution est d'utiliser l'« algorithme » suivant (T représente la cardinalité de l'alphabet):

1. Tirer aléatoirement la valeur de la longueur l en utilisant une distribution uniforme sur les valeurs possibles,
2. Tirer aléatoirement n, le numéro du mot à générer parmi les mots de longueur l en utilisant la distribution uniforme sur $[T^l, 2T^l - 1]$,
3. Retourner n.

Cette technique présente les défauts de ses avantages. En effet, elle est très simple à définir puisqu'elle n'utilise que les distributions de probabilité uniformes. Cependant, calculer la probabilité d'un entier n particulier ne semble pas évident. De plus, étant donné que l'étape

1 utilise une distribution uniforme pour tirer la longueur du mot, cela implique de se limiter à une longueur maximal.

Aussi, implantons-nous ce type de distribution de probabilité de manière différente en nous inspirant de la mesure de probabilité introduite par Quinlan et Rivest dans [QR89]. Nous définissons comme probabilité pour un mot $u \in \Sigma^*$, $p(u) = \left(1 - \frac{1}{r}\right)\left(\frac{1}{Tr}\right)^{|u|}$, avec r un paramètre qui privilégie d'autant plus les petits mots que sa valeur est grande. Par exemple avec $r = 2$ et $\Sigma = \{0, 1\}$, on a $p(\varepsilon) = \frac{1}{2}, p(0) = p(1) = \frac{1}{8}, p(00) = p(01) = p(10) = p(11) = \frac{1}{32} \ldots$

Cette mesure de probabilité est définie sur le domaine des mots ; en ce qui nous concerne et étant donnée l'implantation qui est faite du tirage aléatoire des items (*i.e.* les mots), nous devons définir une telle distribution de probabilité sur les entiers en prenant soin que la probabilité d'un entier particulier n soit égale à la probabilité du mot d'indice n. La longueur du n^e mot de Σ^* est égale à $\lfloor \log_T((T-1)n+1) \rfloor$ (avec T la cardinalité de Σ). En remplaçant $|u|$ par cette valeur dans la définition de p, nous définissons alors une distribution de probabilité P sur $I\!N$ répondant à nos besoins soit : $P(n) = \left(1 - \frac{1}{r}\right)\left(\frac{1}{Tr}\right)^{\lfloor \log_T((T-1)n+1) \rfloor}$. Cette mesure de probabilité sur $I\!N$ est donc implantée en tant que classe dérivant de la classe abstraite `DistriProba` en définissant la fonction f (`la_fonction`) et la limite l de $\sum_{n=0}^{p} f(n)$ de sorte que $P(n) = \frac{f(n)}{l}$ (voir section 5.3.2, page 147).

Utilisation d'automates stochastiques

Remarquons enfin que la génération aléatoire de mots peut être implantée au moyen d'*automates stochastiques* (une définition détaillée des automates stochastiques peut être trouvée au chapitre 4 de [Mic84]). Il s'agit d'automates finis dont chacune des transitions est munie d'une valeur de probabilité. La génération d'un mot se fait alors en tirant aléatoirement la transition à emprunter pour passer d'un état à un autre. Ce choix se fait évidemment en tenant compte des probabilités affectées à chacune des transitions. Pour chaque état terminal, la décision de sortir ou non se fait de manière similaire. De nombreuses équipes travaillant sur l'inférence grammaticale utilisent ce type d'outil. Pour notre part, nous n'avons pas implanté ce type d'automates. Nous pensons cependant y remédier dans une prochaine version de PEpIn<LR>.

6.3.3 Les algorithmes d'apprentissage

Jusqu'à présent, nous avons défini ce que sont les items, les domaines et les concepts dans le contexte des langages réguliers, mais nous n'avons pas encore traité le cas des algorithmes d'apprentissage. Étant donnée la conception de la plate-forme <PEpIn>, il s'agit essentiellement de définir des algorithmes de consistance, c'est-à-dire prenant en entrée un échantillon ou un exemple et retournant un concept hypothèse. Nous rappelons que le processus d'apprentissage consiste à tirer petit à petit un échantillon et à lancer, après chaque tirage, l'algorithme de consistance sur l'échantillon courant.

Dans le contexte particulier des langages réguliers, les représentations des hypothèses sont des DFAs. Les objets apprenant sont, par conséquent, des automates particuliers qui ont la propriété de se construire à partir d'un échantillon d'entrée et selon un algorithme bien défini. L'idée consiste donc à créer une classe d'*automates apprenables*[7] pour chacun des

7. Nous utilisons le terme *automate apprenable* pour désigner les classes implantant les algorithmes de

algorithmes de consistance à étudier. Chacune des classes d'automates apprenables hérite bien évidemment de la classe DFAs. Il est important de comprendre que ces classes d'automates apprenables ne font partie ni de la librairie CLR (puisque cette librairie a pour objectif d'être utilisée dans d'autres cadres que celui de l'apprentissage) ni de la librairie <PEpIn> (puisque <PEpIn> est une librairie générique adaptable à tout type de contexte et non simplement à celui des langages). Finalement, les classes automates apprenables doivent être vues comme des élements de la librairie résultant de l'intégration de CLR dans <PEpIn> au même titre que les classes It_Mots, Domaines_Mots et Concepts_DFAs.

Dans l'esprit de la conception objet, nous commençons par définir la classe DFAsApp qui est la classe mère de toutes les classes d'automates apprenables. Le profil de cette classe est le suivant :

```
class DFAsApp: public DFAs, public Concepts_apprenables<It_Mots> {
protected:

  /* Ici, la methode implantant l'algorithme de consistance */

  void construit_arbre(const Echantillons& ech);

public:

  /* ------------- */
  /* Constructeurs */
  /* ------------- */

  DFAsApp(const Alphabets& a);

  DFAsApp(const DFAsApp& a);

  DFAsApp(const Echantillons& ech);

  DFAsApp(const DFAsApp& a, const Exemples& ex);

  /* ------------------------------------------------------- */
  /* Methodes d'inference pour la classe d'automates consideree. */
  /* ------------------------------------------------------- */

  virtual void infere(const Echantillons& ech);
  /* Infere l'automate deterministe a partir de l'echantillon <ech> en
     creant l'arbre prefixe correspondant. */

  virtual void infere(const Exemples& ex);

  /* ---------------------------------- */
  /* Existance d'une version incermentale */
  /* ---------------------------------- */

  virtual bool existe_app_incremental(void) const;
```

consistance pour les DFAs.

```
  /* ---------------------- */
  /* Methode de clonage */
  /* ---------------------- */

  virtual DFAsApp* clone(void) const;
};
```

Notons d'abord que cette classe est très simple dans sa définition. L'ensemble des méthodes techniques manipulant les automates est apportée *via* l'héritage de la classe `DFAs`. En fait, la méthode la plus importante d'une classe d'automates apprenables est celle implantant l'algorithme de consistance à utiliser. Dans le cas particulier de la classe `DFAsApp` nous avons choisi d'utiliser comme algorithme de consistance `construit_arbre`. Cet algorithme a pour but de construire l'automate arbre préfixe relatif à l'échantillon d'entrée (tel que défini à la section 2.2.2 (page 67)). Ce choix d'algorithme n'est pas innocent puisque la plupart des algorithmes de consistance pour les langages réguliers commencent par construire un tel automate. Ainsi, les apprenants qui hériteront de la classe `DFAsApp` disposeront de fait de cet algorithme. La classe `DFAsApp` propose quatre constructeurs différents. Les deux premiers (dont les arguments sont respectivement `const Alphabets&` et `const DFAsApp&`) sont indispensables pour des questions inhérentes au langage $C++$. Le troisième constructeur (`DFAsApp(const Echantillons& ech)`) implantent effectivement la construction d'un automate consistant avec un échantillon en appelant la méthode `construit_arbre`. Enfin, le dernier constructeur (`DFAsApp(const DFAsApp& a, const Exemples& ex)`) permet de recopier un automate arbre préexistant et d'y ajouter un nouvel exemple.

Les deux méthodes `infere` permettent de modifier un automate `DFAsApp` déjà construit. C'est cette méthode qui est appelée à chaque étape d'une session d'apprentissage, c'est-à-dire après qu'un nouvel exemple ait été tiré. Nous avons déjà évoqué le fait que certains algorithmes de consistance (*e.g.* `RPNI`) ne sont pas utilisables de manière incrémentale, c'est-à-dire que l'hypothèse de l'etape i ne peut servir à la construction de l'hypothèse de l'étape $i+1$. Dans ce cas, la session d'apprentissage fait appel à la méthode d'inférence prenant un échantillon en paramètre. Si l'algorithme de consistance est de type incrémental, la méthode d'inférence utilisée est alors de la deuxième forme (ce type d'inférence étant en général plus performant). Précisons qu'un *objet apprenant*[8] indique à la session d'apprentissage s'il dispose ou non d'une version incrémentale d'inférence au moyen de la méthode `existe_app_incremental` ; la session d'apprentissage est alors en mesure de choisir lequel des deux run-time (voir section 4.3.2, page 136) doit être utilisé.

Nous avons implanté la plupart des algorithmes de consistance pour les langages réguliers présentés au chapitre 2 (page 51), c'est-à-dire `ZR`, k-`RI`, `TB` et `RPNI`. Nous disposons alors des classes d'automates apprenables suivantes : `DFAsApp_ZR`, `DFAsApp_kRI`, `DFAsApp_TB` et `DFAsApp_RPNI`. À titre d'exemple, nous présentons le profil de la classe `DFAsApp_ZR` :

```
class DFAsApp_ZR : public DFAsApp {
protected:

  void Z_Reversibilise(void);

public:
```

8. C'est-à-dire une instance d'une classe automates apprenables.

```
  DFAsApp_ZR(const Alphabets& a);

  DFAsApp_ZR(const DFAsApp_ZR& a);

  DFAsApp_ZR(const Echantillons& ech) : DFAsApp(ech)
  {
    Z_Reversibilise();
  };

  DFAsApp_ZR(const DFAsApp_ZR& a,const Exemples& ex): DFAsApp(a,ex)
  {
    Z_Reversibilise();
  };

  /* ------------------------------------------------------- */
  /* Methodes d'inference pour la classe d'automates consideree. */
  /* ------------------------------------------------------- */

  virtual void infere(const Echantillons& ech)
  {
    DFAsApp::infere(ech);
    Z_Reversibilise();
  };

  virtual void infere(const Exemples& ex)
  {
    DFAsApp::infere(ex);
    Z_Reversibilise();
  };

  /* ---------------------------------- */
  /* Existance d'une version incermentale */
  /* ---------------------------------- */

  virtual bool existe_app_incremental(void) const
  {
    return true;
  };

  /* ------------------ */
  /* Methode de clonage */
  /* ------------------ */

  virtual DFAsApp_ZR* clone(void);
};
```

6.3.4 L'application PEpIn<LR>

Dans ce qui précède, nous avons étudié de quelle manière la librairie CLR était intégrée à la librairie <PEpIn> afin d'affecter un acteur à chacun des rôles génériques définis dans <PEpIn>. Conformément à la figure 4.1 (page 125), nous disposons maintenant d'une

troisième librairie modélisant l'apprentissage dans le contexte des langages réguliers. Il reste alors à écrire le programme principal de l'application. Ce programme consiste essentiellement à construire les différents objets et à lancer la session d'apprentissage.

Afin de rendre l'utilisation de l'application la plus facile possible, le paramétrage d'une session d'apprentissage est fait au moyen d'un fichier de description. Dans celui-ci, l'expérimentateur fixe les divers paramètres tels que la distribution de probabilité à utiliser pour la session, le nom de l'algorithme de consistance à étudier, les différents critères definissant le succès, etc... L'automate cible est également passé à l'application sous la forme d'un fichier.

Le paramétrage et la description des automates cibles *via* des fichiers présentent plusieurs avantages parmi lesquels la possibilité de les générer automatiquement ou bien encore de créer de nouveaux paramétrages simplement en modifiant des fichiers existants. De plus, l'application retourne l'ensemble des résultats dans des fichiers qui peuvent être réutilisés en entrée. Cependant, le côté textuel de tels fichiers peut sembler rebarbatif surtout en ce qui concerne la définition des automates. C'est la raison pour laquelle une interface graphique a été développée dans le cadre d'un projet de stage en DESS. Cette interface a pour but de générer les fichiers de paramétrage mais présente un côté plus convivial pour l'utilisateur.

6.4 Conclusion

Ce chapitre a présenté de manière relativement complète la création d'une application d'expérimentations d'inférence pour le contexte particulier des langages réguliers. Il a permis de souligner l'indépendance entre la librairie modélisant un contexte particulier et la plate-forme <PEpIn>.

Ce chapitre peut être vu comme un schéma conducteur décrivant l'ensemble des opérations indispensables à la création d'une telle application : modélisation et implantation du contexte dans une librairie, intégration de cette librairie à <PEpIn> et création de l'application finale. Notons que l'ensemble de l'application PEpIn<LR> (CLR et <PEpIn> inclus) est composée d'environ 40 classes et de 400 méthodes, ce qui représente un peu moins de 30 000 lignes de code.

Enfin, le développement effectif de l'application PEpIn<LR> montre la validité de notre proposition consistant à définir séparemment la plate-forme modélisant l'apprentissage et le contexte à considérer. La conception de l'application PEpIn<LR> n'avait pas pour seul but de valider notre proposition puisque PEpIn<LR> présente un réel intérêt pour les recherches futures de notre équipe en matière d'inférence grammaticale. Le chapitre suivant se propose de présenter l'utilisation de l'application PEpIn<LR> sur un exemple simple. Ce chapitre sera l'occasion de prouver l'utilité d'un tel outil dans une démarche de recherche empirique.

Chapitre 7

Exemples d'Utilisation de PEpIn<LR>

Ce dernier chapitre, assez court, présente quelques cas d'utilisation de l'application PEpIn<LR>. Les trois chapitres précédents présentaient l'ensemble des étapes qui ont permis de créer une telle application. Ces chapitres restaient, malgré tout, assez conceptuels et techniques. Ce chapitre, beaucoup plus « pratique » a pour but de montrer la réalité de cette application. Dans un premier temps, nous présentons comment construire le fichier de paramétrage d'une session d'apprentissage ainsi que les possibilités offertes par un tel fichier. Ensuite, nous présentons un exemple d'exécution de l'application PEpIn<LR> suivi de la présentation d'une application annexe. Enfin, la dernière partie propose un cas concret d'utilisation de PEpIn<LR>, c'est-à-dire une mise en situation de PEpIn<LR> dans le cas d'une étude théorique.

7.1 Paramétrage d'une Session d'Apprentissage

Nous nous plaçons dans la situation du chercheur désirant utiliser l'application PEpIn<LR> afin, par exemple, de vérifier de manière empirique une idée intuitive. Nous avons vu que la plate-forme <PEpIn> est conçue de manière très ouverte et qu'il est donc possible d'ajouter un grand nombre de classes à celles déjà disponibles par défaut afin, par exemple, d'implanter d'autres distributions de probabilité ou encore d'autres algorithmes d'apprentissage.

Dans la suite de ce chapitre, nous considérons uniquement les outils disponibles dans la version actuelle de PEpIn<LR>. Il convient cependant de se rappeler que toute extension future de l'application sera paramétrable de manière tout aussi facile que ce qui est décrit ici.

Les paramètres d'une session d'apprentissage sont au nombre de 8 énumérés ci-après :

- **CIBLE** : le concept cible est décrit dans un fichier indépendant du fichier de paramétrage, il s'agit donc d'indiquer quel est ce fichier et dans quel répertoire le trouver.
- **NOM_EXPERIENCE** : les résultats d'une expérience sont mémorisés dans plusieurs fichiers. Ce paramètre donne simplement le nom des fichiers résultats ainsi que le répertoire dans lequel l'application doit les créer.
- **ALGO_APP** : indique quel algorithme d'apprentissage utiliser. Nous rappelons que dans sa version de base, PEpIn<LR> propose les algorithmes `ZR`, k-`RI`, `T_B` et `RPNI`.

- **ERREUR** : deux types de calcul d'erreur peuvent être utilisés : le calcul exact ou approché. Ce paramètre fixe quelle méthode l'utilisateur désire utiliser. Dans le cas du calcul d'erreur approximé, il convient également d'indiquer les valeurs de précision et de confiance. Précisons que l'erreur n'est calculable que dans le cas de l'utilisation automatique[1] de PEpIn<LR>.

- **DISTRIB_PROBA** : la distribution de probabilité définit de quelle manière doivent être tirés les exemples. Ici encore, ce paramètre n'est utilisé que dans le cas de l'utilisation automatique de PEpIn<LR>. Précisons de plus que le calcul de l'erreur est fait au moyen de cette distribution de probabilité, c'est-à-dire que le poids de chaque mot dans le calcul de l'erreur est donné par sa mesure de probabilité. Le paramétrage de la distribution de probabilité demande à être raffiné. Par exemple, pour la distribution uniforme entre deux bornes, il convient de donner la valeur de celles-ci.

- **DOMAINE** : ce paramètre fixe le domaine dans lequel les mots doivent être tirés. Ce domaine est associé à la distribution de probabilité. Le domaine peut être soit un automate, on raffine alors le paramétrage au moyen du nom du fichier décrivant cet automate, soit un alphabet Σ[2] qu'il convient de décrire en énumérant les lettres le constituant.

- **ORACLE** : trois types d'oracles peuvent être utilisés. Le premier, désigné par la valeur « `Manuel` », indique que c'est l'utilisateur qui va jouer le rôle de l'enseignant et ainsi proposer les exemples successifs (les paramètres ERREUR et DISTRIB_PROBA sont alors inutilisés). Les deux autres types d'oracles (`General` ou `Positif`) définissent une utilisation en mode automatique de l'application. Ils permettent de générer respectivement des exemples positifs et négatifs ou uniquement positifs, selon la distribution de probabilité demandée.

- **COND_ARRET** : ce dernier paramètre permet d'indiquer à quelle condition la session d'apprentissage doit s'arrêter. Nous rappelons que plusieurs conditions d'arrêt peuvent être utilisées conjointement ; la validité d'au moins une condition implique alors l'arrêt du processus.

À part les paramètres **CIBLE** et **NOM_EXPERIENCE** qui demandent comme valeur un nom de fichier, les autres paramètres doivent être utilisés avec des valeurs bien définies. Nous listons dans le tableau de la figure 7.1 (page suivante) l'ensemble des valeurs disponibles actuellement. Ce tableau indique également l'obligation ou non de fixer la valeur de chaque paramètre lors d'une utilisation en mode automatique ou manuelle ; nous rappelons que c'est le type d'oracle à utiliser qui fixe le mode d'utilisation de l'application.

Dès lors, un fichier de paramétrage s'écrit très facilement puisqu'il suffit de remplir chacun des champs paramètres (labellés par les noms des paramètres) avec les valeurs désirées. Remarquons que la lecture d'un fichier de paramétrage est rendue très claire de par l'utilisation de mots explicites.

Exemple :
La figure 7.2 (page 188) présente un fichier de paramétrage tel que ceux utilisables sur l'application PEpIn<LR>. Par ce paramétrage particulier, on demande que la cible soit l'automate

1. C'est-à-dire que l'application joue à la fois le rôle de l'enseignant et celui de l'élève.
2. Le domaine est alors Σ^*

Paramètres	Mode Automatique	Mode Manuel	Valeurs possibles	Raffinement
ALGO_APP	Obligatoire	Obligatoire	ZR T_B RPNI	Aucun
			k-RI	Valeur de k (entier)
ERREUR	Obligatoire	Inutilisé	Exact	Aucun
			Approxime	Précision (réel) Confiance (réel)
DISTRIB_PROBA	Obligatoire	Inutilisé	Uniforme	Borne inférieure (entier) Borne supérieure (entier)
			Quinlan	Valeur de r (entier)
DOMAINE	Obligatoire	Inutililisé	Alphabet	Liste de lettres (chaîne de caractères)
			Automate	Nom de fichier (chaîne de caractères)
ORACLE	Obligatoire	Obligatoire	Manuel General Positif	Aucun
COND_ARRET	Non obligatoire	Non obligatoire		Nb. conditions (réel)
			Apprentissage exact	Aucun
			Nombre etapes borne	Nb. étapes max. (entier)
			Taille echantillon bornee	Nb. exemples min. (entier)
			Nombre hypotheses consecutives identiques borne	Nb. d'hypthèses identiques maximal (entier)
			Seuil erreur	Valeur erreur max. (reel)

FIG. 7.1 - Tableau des paramètres disponibles pour PEpIn<LR>.

décrit dans le fichier `Mon_automate.auto` et que les résultats soient copiés dans des fichiers de nom `Test` (nous verrons par la suite quels sont les fichiers résultats). L'algorithme à utiliser est `ZR` (c'est-à-dire qu'on peut supposer que la cible est un automate 0-réversible). Le calcul d'erreur doit se faire de manière approximée avec une précision de 10% et une confiance de 95%. Les mots sont tirés de manière uniforme dans le domaine $\Sigma^{\leq 9}$ avec $\Sigma = \{0, 1\}$ (seuls les mots d'indice compris entre 1 et 511 ont une probabilité non nulle, c'est-à-dire les mots de longueur inférieure ou égale à 9). L'oracle fournit à l'algorithme d'apprentissage des exemples positifs construits à partir des mots tirés aléatoirement. Enfin, on demande que la session d'apprentissage s'arrête dès lors que l'hypothèse inférée est égale à la cible ou bien que le nombre d'étapes est égal à 100. ◇

```
CIBLE
/Test/Langages/Mon_automate.auto

NOM_EXPERIENCE
/Test/Resultats/Test

ALGO_APP
ZR

ERREUR
Approxime
0.1
0.95

DISTRIB_PROBA
Uniforme
1 511

DOMAINE
Alphabet
01

ORACLE
Positif

COND_ARRET
2
Apprentissage exact
Nombre etapes borne
100

# Fin de parametrage
FIN
```

FIG. 7.2 - Exemple de fichier de paramétrage.

7.2 Exemples d'Exécution de PEpIn<LR>

Le fichier de paramétrage (de nom `mes_parametres`) étant écrit, il ne reste plus qu'à lancer l'expérience au moyen de la commande :
`PEpIn_LR n < mes_parametres`

La valeur n correspond au nombre d'expériences à lancer. Une expérience correspond à une exécution de session d'apprentissage mais il est possible de demander l'exécution successive de plusieurs expériences paramétrées de la même manière. Cette facilité présente deux intérêts majeurs : d'abord cela permet de lancer un certain nombre d'expériences en « batch » sans avoir à relancer le processus. Ensuite, étant donné que pour l'ensemble des expériences le même oracle de génération d'exemples est utilisé, celui-ci n'est pas réinitialisé d'une expérience à l'autre. Ainsi, les mots déjà tirés restent mémorisés et la génération de mots dans les expériences successives peuvent bénéficier des générations précédentes de mots, ce qui permet un gain en temps pour la génération d'exemples (voir section 6.3.2 (page 178) pour plus de détails sur la génération des mots).

7.2.1 Fichiers résultats

Toute exécution d'une expérience d'apprentissage construit trois fichiers dont le nom est celui fixé par le paramètre **NOM_EXPERIENCE**. Ces trois fichiers sont différenciés par leur suffixe. Ainsi, on trouve un premier fichier contenant le rapport de l'expérience : nombre d'étapes, taille de la cible inférée, taille de l'échantillon utilisé et cause d'arrêt du processus ; ce fichier a comme suffixe « `.expr` ». Les deux autres fichiers résultats, de suffixe « `.sample` » et « `.auto` », contiennent respectivement la description de l'échantillon utilisé et la description de l'hypothèse inférée. Précisons que ces deux derniers fichiers peuvent être réutilisés en tant que données d'entrée pour des applications futures. Par exemple, le fichier décrivant l'hypothèse inférée peut devenir le fichier décrivant la cible pour une autre expérience.

Il reste à préciser que lors d'un lancement de plusieurs expériences successives paramétrées par un même fichier de paramétrage, les noms des fichiers résultats contiennent en plus le numéro de l'expérience à laquelle ils correspondent.

7.2.2 Lancement et exécution d'une expérience

Considérons maintenant le lancement d'une expérience particulière dont le paramétrage est celui donné à la figure 7.2 (page précédente). La figure 7.3 (page suivante) présente le lancement d'une (multi)-session d'apprentissage à 2 expériences avec ces paramètres. Le langage cible est le langage 0-réversible contenant tous les mots constitués d'un nombre pair de 0 et d'un nombre pair de 1.

Nous pouvons remarquer qu'au long de l'exécution, l'application retourne sur la sortie standard l'ensemble des résultats intermédiaires, avec pour chaque étape d'expérience l'erreur commise, la taille de l'hypothèse courante et l'exemple tiré à cette étape. Ces résultats intermédiaires sont essentiels pour une analyse détaillée d'expérience puisqu'ils traduisent l'évolution de l'apprentissage.

À la fin de cette exécution, nous disposons donc de 6 fichiers résultats (3 par expérience) comme précédemment décrit. La figure 7.4 (page suivante) présente par exemple le fichier `TEST1.expr` décrivant le rapport général de la première expérience.

```
PEpIn_LR 2 <mes_parametres

experience 1
Etape 1 :  Erreur = 99 % +/- 10% (95%) , Taille hypothese = 7, Exemple = (011011,POSITIF)
Etape 2 :  Erreur = 100 % +/- 10% (95%) , Taille hypothese = 13, Exemple = (00111010,POSITIF)
Etape 3 :  Erreur = 98 % +/- 10% (95%) , Taille hypothese = 9, Exemple = (0011,POSITIF)
Etape 4 :  Erreur = 97 % +/- 10% (95%) , Taille hypothese = 14, Exemple = (11000110,POSITIF)
Etape 5 :  Erreur = 97 % +/- 10% (95%) , Taille hypothese = 16, Exemple = (000000,POSITIF)
Etape 6 :  Erreur = 55 % +/- 10% (95%) , Taille hypothese = 4, Exemple = (00000000,POSITIF)
Etape 7 :  Erreur = 0 % +/- 10% (95%) , Taille hypothese = 4, Exemple = (11100010,POSITIF)

experience 2
Etape 1 :  Erreur = 99 % +/- 10% (95%) , Taille hypothese = 7, Exemple = (010111,POSITIF)
Etape 2 :  Erreur = 99 % +/- 10% (95%) , Taille hypothese = 7, Exemple = (1111,POSITIF)
Etape 3 :  Erreur = 97 % +/- 10% (95%) , Taille hypothese = 11, Exemple = (01111101,POSITIF)
Etape 4 :  Erreur = 97 % +/- 10% (95%) , Taille hypothese = 14, Exemple = (11110110,POSITIF)
Etape 5 :  Erreur = 97 % +/- 10% (95%) , Taille hypothese = 17, Exemple = (01100000,POSITIF)
Etape 6 :  Erreur = 95 % +/- 10% (95%) , Taille hypothese = 17, Exemple = (01011010,POSITIF)
Etape 7 :  Erreur = 95 % +/- 10% (95%) , Taille hypothese = 18, Exemple = (01101100,POSITIF)
Etape 8 :  Erreur = 95 % +/- 10% (95%) , Taille hypothese = 18, Exemple = (11100100,POSITIF)
Etape 9 :  Erreur = 92 % +/- 10% (95%) , Taille hypothese = 18, Exemple = (101101,POSITIF)
Etape 10 :  Erreur = 0 % +/- 10% (95%) , Taille hypothese = 4, Exemple = (10110100,POSITIF)
```

FIG. 7.3 - Exécution de deux expériences consécutives d'apprentissage.

```
NOMBRE_TOTAL_ETAPES : 7
TAILLE_ECHANTILLON : 7
TAILLE_HYPOTHESE : 4
CAUSE_D_ARRET : Cible trouvee exactement
```

FIG. 7.4 - Fichier « rapport d'une expérience ».

7.2.3 Utilisation d'algorithmes de consistance

Nous venons de décrire une expérience d'apprentissage. Dans certains cas, il semble intéressant de pouvoir lancer simplement l'un des algorithmes de consistance disponibles (à savoir `ARBRE_PREFIXE`, `ZR`, k-`RI`, `T_B` et `RPNI`) afin d'étudier le comportement d'un tel algorithme sur un échantillon particulier. Pour ce faire, nous avons écrit une petite application simple, `Echant2Auto` indépendante de PEpIn<LR> dont le seul but est de lire un échantillon en entrée et d'appliquer l'algorithme de consistance désiré sur celui-ci. Cette application retourne comme résultat l'automate construit par l'algorithme.

Dans la section précédente, nous avons indiqué que le fichier résultat décrivant l'hypothèse inférée après une session d'apprentissage peut être réutilisé en tant que fichier décrivant une cible. La même facilité est offerte au niveau des fichiers décrivant l'échantillon effectivement utilisé dans une expérience d'apprentissage. Par conséquent, ceux-ci peuvent être utilisés en tant que fichier d'entrée pour l'application `Echant2Auto`.

Afin de vérifier la stabilité, l'efficacité et la robustesse de notre application, ces algorithmes ont été testés avec les fichiers de données du concours Abbadingo. Ces fichiers ont la particularité de contenir un grand nombre d'exemples et les structures qui sont générées derrière sont, par la même, de taille importante.

Par exemple, le fichier `train.a` proposé dans le concours Abbadingo contient 4456 exemples

labellés. L'automate arbre préfixe correspondant est constitué de plus de 15000 états. Nos structures de données permettent effectivement de construire de tels automates et les algorithmes tournent parfaitement dessus. Il faut cependant préciser qu'il reste un petit travail d'optimisation à faire au niveau du temps d'exécution de certains de nos algorithmes. Ainsi par exemple, l'algorithme de Trakhtenbrot et Barzdin (T_B) que nous avons implanté est bien moins performant que celui proposé sur le site d'Abbadingo. À notre décharge, il faut préciser que l'application disponible sur le site Abbadingo ne traite que l'algorithme de Trakhtenbrot et Barzdin et n'a pas l'ambition d'être aussi général que nos propres applications.

7.3 Mise en Situation

L'application PEpIn<LR> est l'application principale permettant de lancer et d'étudier des modèles et des algorithmes d'apprentissage. Nous avons présenté dans la section précédente une autre application complémentaire de PEpIn<LR> dont le but est d'étudier le comportement d'un algorithme de consistance particulier sur un échantillon particulier. Afin de disposer d'un ensemble d'outils pratiques répondant à d'autres demandes de la part de l'utilisateur, nous avons implanter un certain nombre d'autres petites applications annexes qui peuvent être vues comme des satellites de PEpIn<LR>. Toutes ces applications sont construites au dessus des deux librairies <PEpIn> et CLR.

Nous nous proposons de décrire ici un petit scénario décrivant une idée de recherche pouvant demander l'utilisation de PEpIn<LR>. Notre but n'est pas de répondre aux questions théoriques soulevées ici mais seulement de présenter de quelle manière l'application PEpIn<LR> et ses applications satellites peuvent aider le chercheur se lançant dans une telle tâche. Précisons enfin que le problème présenté ici ainsi que la proposition d'expérience que nous donnons afin de tenter d'y répondre font partie de l'un de nos travaux de recherche actuels[3].

7.3.1 Énoncé du problème

La classe des langages 0-réversibles est prouvée PACS-apprenable (voir section 3.3, page 97). Cette classe de langages est également apprenable dans le modèle PAC avec enseignant proposé par Denis et Gilleron [DG97, DGS97]. Dans les deux cas, l'algorithme ZR est utilisé par l'algorithme d'apprentissage, en tant qu'algorithme d'Occam. Dans le modèle PAC classique, nous ne disposons d'aucun résultat d'apprenabilité sur la classe des langages 0-réversibles. Nous pensons cependant que l'algorithme ZR ne peut être utilisé en tant qu'algorithme d'Occam dans ce modèle. Il se peut pourtant qu'il existe un autre algorithme pouvant jouer ce rôle et qui permettrait alors de prouver la PAC-apprenabilité de cette classe de langages.

Dans les modèles PACS et PAC avec enseignant, la théorie donne une valeur très grande sur la taille de l'échantillon à tirer pour que l'apprentissage ait effectivement lieu. Cette valeur importante est due au fait qu'il faut assurer que l'échantillon caractéristique du langage cible soit effectivement tiré.

L'idée que l'on propose est d'étudier de manière empirique, si cette taille théorique est réellement indispensable. Nous pensons que pratiquement, avec un échantillon de taille bien

3. Ceux-ci n'en sont pourtant qu'à leur début, il nous est par conséquent impossible de présenter de manière sérieuse un quelconque résultat.

inférieure, l'apprentissage doit aboutir dans bien des cas. En fait, nous pensons que l'échantillon caractéristique défini par la théorie n'est pas le seul échantillon permettant d'assurer l'apprentissage. Aussi, serait-il intéressant d'étudier de manière empirique s'il n'existe pas d'autres types d'échantillons caractéristiques. Si tel est le cas, ceci permettrait alors peut-être de trouver un nouvel algorithme de consistance pour les automates 0-réversibles qui pourrait jouer le rôle d'algorithme d'Occam.

Nous rappelons encore que le but n'est pas ici de répondre à cette question, ni même de décrire la ou les expériences pour tenter d'y répondre mais simplement de donner une idée de la façon dont pourrait être utilisée PEpIn<LR> pour entreprendre une telle tâche.

7.3.2 Création du matériel d'expérience

Une telle démarche empirique ne peut, raisonnablement, être faite sur des « cas d'école ». Il n'est en effet pas concevable de pouvoir établir des résultats représentatifs sur des langages cibles dont l'automate canonique est de petite taille. Aussi, il convient de construire un certain nombre d'automates 0-réversibles de taille suffisante.

Nous proposons de créer ce matériel d'expérience de manière automatique afin d'avoir une réserve suffisamment conséquente de langages cibles. Il s'agit donc de construire automatiquement des automates 0-réversibles de taille assez grande.

Pour ce faire, nous proposons d'utiliser l'algorithme ZR d'Angluin sur un échantillon généré aléatoirement. Étant donnée les propriétés de l'algorithme ZR, celui-ci construit un automate canonique 0-réversible qui conviendra à nos expériences.

En fixant une taille d'échantillon suffisamment grande, l'automate ainsi généré est de grande taille. Cependant, procéder de cette manière ne permet pas, en général de construire des automates reconnaissant un grand nombre de mots. Il s'agit, la plupart du temps, d'automates reconnaissant presque uniquement les mots de l'échantillon ayant servi à sa construction. Nous imposons donc une contrainte supplémentaire aux automates ainsi générés: ceux-ci doivent généraliser l'échantillon d'entrée, soit encore compresser l'automate arbre préfixe correspondant à cet échantillon. Cette contrainte peut être satisfaite en comparant, à chaque étape du processus, la taille de l'automate généré (disons T) et celle de l'automate arbre préfixe correspondant à l'échantillon (disons T') et en imposant que le rapport $\frac{T}{T'}$ soit inférieur à une certaine valeur r fixée au départ. Ainsi, on est assuré que le langage reconnu par l'automate généré est suffisamment éloigné de celui ne contenant que l'échantillon de construction.

La figure 7.5 (page suivante) présente l'algorithme de génération aléatoire d'automates 0-réversibles. Nous avons implanté cet algorithme sous la forme d'une petite application `genere_0_rev`. Nous avons ainsi construit un ensemble d'automates 0-réversibles « non triviaux » ayant des tailles allant de 10 à 150 états.

7.3.3 Proposition d'expériences

Il nous reste à définir un petit scénario d'expérimentation destiné à tenter de répondre au problème posé au début de cette section.

Nous proposons par exemple de lancer, sur un automate particulier, une série d'apprentissages avec une distribution uniforme. Nous pouvons imposer comme critère de succès un apprentissage exact puisque dans les modèles PACS et PAC avec enseignant l'apprenabilité de la classe $\mathcal{R}ev_0$ est exacte; rappelons que nous évaluons l'apprentissage dans l'un de ces

Algorithme Genere_0_rev
Entrée : r un entier
Début
  Soit S un échantillon vide
  Faire
    Tirer aléatoirement un mot m
    Ajouter m à S
    Soit $A = \mathtt{ZR}(S)$
    Soit PT l'automate arbre préfixe de S
    Soit T la taille de A
    Soit T' la taille de A'

  Tantque $\frac{T}{T'} \leq r$
Fin

FIG. 7.5 - Algorithme de génération aléatoire d'automate 0-réversible.

modèles afin de rechercher l'existence d'un autre type d'échantillon caractéristique que celui utilisé dans la théorie. Il convient de lancer plusieurs expériences sur la même cible avec la même distribution de probabilité. D'autres expériences peuvent ensuite être lancées en utilisant la même cible mais en faisant varier l'intervalle de la distribution uniforme. Il est important d'étudier des cas de distributions uniformes affectant la probabilité nulle à tous les exemples de l'échantillon caractéristique théorique.

L'ensemble de ces expériences fournit un grand nombre de données résultats à analyser. On pourra, par exemple, étudier la taille moyenne de l'échantillon permettant d'apprendre. Ensuite, une analyse plus fine de ces échantillons, c'est-à-dire de leur contenu, devra être faite afin d'étudier par exemple, s'il existe pour chaque distribution de probabilité, un exemple ou un ensemble d'exemples indispensable à l'apprentissage. Si tel est le cas, ces exemples forment un échantillon caractéristique « pratique » pour une distribution particulière. En supposant que pour chaque distribution de probabilité nous trouvions un tel échantillon, il conviendrait, alors, d'étudier les propriétés de ceux-ci. Enfin les hypothèses établies à partir des expériences relatives à un ou plusieurs langages cibles devront être confirmées par d'autres expériences pour un plus grand nombre d'automates cibles.

Nous n'assurons pas que les quelques idées « en vrac » proposées ici suffisent à atteindre le but fixé au début. Il semble cependant que de telles expériences définissent un cadre de travail indispensable afin d'aborder un tel problème.

7.4 Conclusion

Ce dernier chapitre illustre parfaitement de quelle manière l'application PEpIn<LR> peut être utilisée, ce qu'elle permet de faire et comment elle permet d'aborder l'étude de problèmes théoriques.

L'application PEpIn<LR> répond en grande partie à l'ensemble des attentes fixées au

début de ce travail d'implantation (voir chapitre 4, page 123). Notamment, les tests que nous avons lancés sur les données du concours Abbadingo démontrent la robustesse de nos structures de données. L'objectif consistant à pouvoir traiter des données de taille importante est ici parfaitement rempli. Il reste cependant à optimiser quelques uns des algorithmes afin de gagner en efficacité.

Ce chapitre propose également un petit scénario d'expérimentation pour tenter de répondre à une question théorique. Pour notre part, les premières expériences que nous avons lancées suivant ce scénario sont tout à fait encourageantes et montrent l'utilité d'une telle application. Nous espérons que la suite de ces expérimentations, qui n'en sont encore qu'au début, nous permettra de proposer quelques réponses intéressantes au niveau théorique.

Conclusion

Dans la première partie de cette thèse, les aspects théoriques de l'apprentissage automatique ont été abordés. Après un état de l'art relativement général incluant les définitions de base de ce domaine, nous nous sommes intéressés plus particulièrement à l'apprentissage des langages réguliers. Dans ce domaine, nous avons vu qu'il reste encore de nombreuses classes non prouvées apprenables. Ce chapitre a été également l'occasion de comprendre l'intérêt des échantillons caractéristiques lors de l'apprentissage.

Enfin, nous avons proposé un nouveau modèle d'apprentissage, appelé modèle PACS, qui dérive du modèle de Valiant. Ce nouveau modèle donne une place prépondérante aux exemples simples relativement à la cible à apprendre. Nous avons prouvé l'apprenabilité de plusieurs classes non triviales dans ce modèle. Ce modèle semble cependant trop laxiste et permet, par exemple, d'apprendre certaines classes de concepts au moyen d'un algorithme collusif. L'apprentissage de la classe des langages réversibles $\mathcal{R}ev$ n'a pas été démontrée apprenable, dans aucun des modèles d'apprentissage. Nous espérons trouver un algorithme qui permette de traiter ce problème dans notre modèle.

La deuxième partie de cette thèse présente un travail de conception et d'implantation d'une plate-forme générique d'expérimentations pour l'inférence. Cette plate-forme a été effectivement implantée et a donné lieu à la création d'une application d'expérimentations de l'apprentissage dans le cadre des langages réguliers. Beaucoup de travail reste à faire pour améliorer cette plate-forme et cette application particulière. D'abord, nous espérons utiliser notre application PEpIn<LR> dans un véritable cadre de recherche, c'est-à-dire avec l'objectif de résoudre un problème théorique afin de parfaitement valider cette application. Ensuite, au niveau technique, un travail d'optimisation devra être fait afin d'améliorer le temps d'exécution de quelques algorithmes.

Nous envisageons d'étendre les possibilités de notre plate-forme en intégrant de nouveaux outils tels qu'un générateur aléatoire d'automates par exemple. Nous pensons également intégrer de nouvelles distributions de probabilité qui pourraient « simuler » la mesure de Solomonoff-Levin. Nous avons également l'idée de permettre l'utilisation non plus simplement d'une distribution de probabilité particulière mais de familles de distributions. Cette amélioration permettrait de mieux définir les modèles d'apprentissage au niveau de l'exécution et non plus de simplement avoir à faire à une instance particulière de ceux-ci.

Enfin, nous espérons utiliser notre plate-forme pour nos travaux théoriques et plus précisément pour définir de nouveaux algorithmes d'apprentissage (notamment pour les langages réversibles). Une première idée, toujours basée sur les échantillons caractéristiques, serait que l'algorithme d'apprentissage évalue l'échantillon d'apprentissage par rapport à l'échantillon caractéristique de l'hypothèse inférée.

En conclusion, nous pensons que notre plate-forme est un réel apport pour le théoricien et qu'elle permettra d'ouvrir de nouvelles perspectives de recherches en reliant la démarche

empirique aux travaux théoriques.

Annexe A

Supplément au chapitre 3

Cette annexe présente les preuves détaillées concernant les résultats de PACS-apprenabilité des langages k-réversibles (voir section 3.3.2, page 109).

A.1 Rappels et Propositions Préliminaires

On note $\mathcal{R}ev_k$ la classe des langages k-réversibles et $\mathcal{R}ev_k^{\leq n}$ celle des langages k-réversibles dont l'automate canonique a au plus n états.

Notation :
Soit $A = (Q, \{q_0\}, F, \Sigma, \delta)$ un automate déterministe. Soit $q \in Q$ un état quelconque de A. On note L_q^k l'ensemble des k-meneurs de l'état q. Soit $x \in \Sigma^*$. On note $u_A(q,x)$ le plus petit mot tel que $\delta(q_0, u_A(q)x) = q$. On note $v_A(q)$ le plus petit mot labellant un chemin menant de l'état q à un état final. Formellement, $u_A(q,x) = \min(\{u \in \Sigma^* \mid \delta(q_0, ux) = q\})$ et $v_A(q) = \min(\{v \in \Sigma^* \mid \delta(q,v) \in F\})$.

Remarque A.1.1 Si $q \in Q$ et $x \in \Sigma^*$, il existe un état $q' \in Q$ tel que $\delta(q_0, u_A(q,x)) = q'$. Par conséquent, il est évident que $u_A(q,x) = u_A(q')$. Le lemme 3.3.4 (page 101) s'applique donc aussi sur les mots $u_A(q,x)$ c'est-à-dire que si $q'' = \delta(q_0, Pre_i(u_A(q,x)))$ (avec $i \leq |u_A(q,x)|$) alors $Pre_i(u_A(q,x)) = u_A(q'')$.

Pour tout langage $L \in \mathcal{R}ev_k$, d'automate canonique $A = (Q, \{q_0\}, F, \Sigma, \delta)$, il existe un échantillon caractéristique noté S_L^k (voir définition 2.2.8, page 75), tel que L est le plus petit langage k-réversible contenant S_L^k avec

$$S_L^k = L^{\leq k} \cup \{u_A(q,x)xv_A(q) \mid q \in Q\, x \in L_q^k\} \cup \{u_A(q,x)xav_A(q') \mid q,q' \in Q, a \in \Sigma, x \in L_q^k, q' = \delta(q,a)\}$$

De plus, l'algorithme k-RI proposé par Angluin [Ang82a] sur l'entrée S retourne l'automate canonique du plus petit langage k-réversible contenant S. Si $S_L^k \subseteq S$, k-RI(S) retourne exactement l'automate canonique de L.

Proposition A.1.1 *Soit L un langage k-réversible. Si l'automate canonique de L, noté $A = (Q, \{q_0\}, F, \Sigma, \delta)$, contient n états, alors S_L^k, l'échantillon caractéristique de L, contient au maximum $|\Sigma|^k.(|\Sigma|+1).n + \frac{1-|\Sigma|^{k+1}}{1-|\Sigma|}$ chaînes et toutes les chaînes de S_L^k ont une longueur inférieure à $2n+k+1$.*

Preuve :
Pour une question de simplicité, nous utiliserons la notation $u(q)$ (resp. $v(q)$) à la place de $u_A(q)$ (resp. $v_A(q)$). L'échantillon caractéristique d'un langage k-réversible L est défini par: $S_L^k = L^{\leq k} \cup \{u(q,x)xv(q) \mid q \in Q\, x \in L_q^k\} \cup \{u(q,x)xav(q') \mid q,q' \in Q, a \in \Sigma, x \in L_q^k, q' = \delta(q,a)\}$. En ce qui concerne la borne sur le nombre de mots de S_L^k, notons d'abord que,

$$|L^{\leq k}| \leq \frac{1-|\Sigma|^{k+1}}{1-|\Sigma|}$$

De plus,

$$|\{u(q,x)xv(q) \mid q \in Q, x \in L_q^k\}| \leq n.|\Sigma|^k$$

et enfin,

$$|\{u(q,x)xav(q') \mid q,q' \in Q, a \in \Sigma, x \in L_q^k, q' = \delta(q,a)\}| \leq n.|\Sigma|^{k+1}$$

Le nombre d'éléments de S_L^k étant égal à la somme des cardinaux de ces trois ensembles, on obtient donc,

$$|S_L^k| \leq |\Sigma|^k.(|\Sigma|+1).n + \frac{1-|\Sigma|^{k+1}}{1-|\Sigma|}$$

Le résultat concernant la borne sur la longueur des mots de S_L^k est tout aussi évident. En effet, la longueur maximale des mots de S_L^k est inférieure à la longueur maximale des mots de $\{u(q,x)xav(q') \mid q,q' \in Q, a \in \Sigma, x \in L_q^k, q' = \delta(q,a)\}$, c'est-à-dire $|u(q,x)| + |x| + |v(q)| + 1$. De plus, la longueur d'un mot minimal menant d'un état de l'automate à un autre état est inférieure à n ; donc $|u(q,x)| \leq n$ et $|v(q)| \leq n$. On obtient ainsi la borne proposée. □

A.2 Preuve du Théorème 3.3.9

Théorème A.2.1 *Pour tout n, $\mathcal{R}ev_k^{\leq n}$ est PACS-apprenable polynomialement et l'apprentissage est probablement exact.*

Preuve :
La preuve s'appuie sur le théorème d'Occam (théorème 3.1.4 (page 91)). Il s'agit donc de décrire un algorithme de PACS-Occam B ainsi qu'un échantillon représentatif étant donné un langage de $\mathcal{R}ev_k^{\leq n}$. L'ensemble des hypothèses du théorème 3.1.4 (page 91) sera ensuite vérifié. Soit un entier n, L un langage de $\mathcal{R}ev_k^{\leq n}$ et r un nom pour L. On définit $S_r = S_L^k$ et on propose comme algorithme de PACS-Occam l'algorithme B suivant:

Algorithme B
Entrée : S
Constante connue : n, k
Début
  Soit S' l'ensemble des exemples positifs de S de longueur au plus $2n+k+1$.
  Lancer k-RI sur l'entrée S'
  Sortir un nom pour k-RI(S')
Fin

Montrons que toutes les conditions du théorème 3.1.4 (page 91) sont satisfaites:

- **Il existe une constante c telle que pour tout $L \in \mathcal{R}ev_k^{\leq n}$ de nom r, l'échantillon S_r est c-simple.** Montrons qu'il existe une constante c telle que $\forall\ x \in S_r$, $K(x \mid r) \leq c\log(|r|)$. Soient $L \in \mathcal{R}ev_k^{\leq n}$ et n_0 le nombre d'états de l'automate canonique de L ($n_0 \leq n$). Soit $x \in S_r$. Pour générer, x il est suffisant d'avoir S_r ainsi que l'index i de x dans cet ensemble. D'après la proposition A.1.1 (page 197),

$$|S_r| \leq |\Sigma|^k.(|\Sigma|+1).n + \frac{1-|\Sigma|^{k+1}}{1-|\Sigma|}$$

donc en posant $c_1 = |\Sigma|^k.(|\Sigma|+1)$ et $c_2 = \frac{1-|\Sigma|^{k+1}}{1-|\Sigma|}$,

$$i \leq c_1 n + c_2$$

Par conséquent, $K(x \mid S_r) \leq \log(c_1 n + c_2) \leq \log(c_1) + \log(n) + c_2'$. De plus, par définition de S_r, il existe un algorithme construisant cet échantillon directement à partir de l'automate canonique de L (c'est-à-dire r) ; d'où,

$$K(S_r \mid r) = O(1)$$

D'après l'inégalité 2 du théorème 1.4.4 (page 43), on a,

$$K(x \mid r) \leq K(x \mid S_r) + K(S_r \mid r) + O(1)$$

D'où,

$$K(x \mid r) \leq \log(n) + O(1)$$

D'après les hypothèses sur R, $n \leq |r|$ donc,

$$K(x \mid r) \leq \log(|r|) + O(1)$$

En supposant que $\log(|r|) \geq 1$, il existe donc une constante c telle que,

$$K(x \mid r) \leq c\log(|r|)$$

Pour la suite on considère $L \in \mathcal{R}ev_k^{\leq n}$ et n_0 le nombre d'états de l'automate canonique de L.

- **Si $S_r \subseteq S$, l'algorithme `B` retourne le nom d'un concept consistant avec S.** Si $S_r \subseteq S$, alors $S_r \subseteq S'$ puisque S' est l'ensemble des mots positifs de longueur inférieure ou égale à $2n+k+1$ (condition vérifiée par l'ensemble des mots de S_r d'après la proposition A.1.1 (page 197) et $n_0 \leq n$). Or, l'algorithme k-`RI` retourne l'automate canonique de L si l'échantillon d'entrée (S' ici) contient l'échantillon $S_L^k = S_r$. L'hypothèse retournée est donc consistante avec L et par conséquent avec $S \subseteq L$.

- **L'algorithme B tourne en temps polynomial lorsque $S_r \subseteq S$.** La construction de l'échantillon S' est évidemment en temps polynomial par rapport à $|S|$. L'algorithme k-RI tourne en temps $O(k\|S'\|^3)$ (voir section 2.2.3, page 75). Par construction, les mots de S' ont une longueur inférieure ou égale à $2n+k+1$, d'où $\|S'\| \leq (2n+k+1)|S'| \leq (2n+k+1)|S|$. La complexité en temps de l'algorithme B est donc polynomial par rapport à $|S|$.

- **Lorsque $S_r \subseteq S$, il existe un nom «court» pour le concept L' retourné par l'algorithme B.** Si $S_r \subseteq S'$, l'algorithme k-RI retourne un automate l'automate canonique du langage L. La preuve de cette hypothèse devient alors évidente.

Chaque condition du théorème 3.1.4 (page 91) étant vérifiée, la poly-PACS-apprenabilité de la classe $\mathcal{R}ev_k^{\leq n}$ est prouvée. Enfin, l'apprentissage est probablement exact puisque l'algorithme k-RI retourne exactement l'automate canonique de L si $S_r \subseteq S$; or ceci arrive avec la probabilité $1-\delta$ d'après le lemme 3.1.3 (page 90) concernant les échantillons c-simples. □

A.3 Preuve du Théorème 3.3.10

Théorème A.3.1 *La classe $\mathcal{R}ev_k$ des langages k-réversibles est PACS-apprenable par exemples positifs seuls en temps usuellement polynomial et l'apprentissage est probablement exact.*

La preuve de ce théorème demande qu'un certain nombre de lemmes techniques soient préalablement montrés.

Lemme A.3.2 *Soient k un entier et $A = (Q, \{q_0\}, F, \Sigma, \delta)$ un automate k-réversible. Si $u_1 w \in L(A)$ et $u_2 w \in L(A)$ avec $|w| \geq k$ alors $\delta(q_0, u_1) = \delta(q_0, u_2)$.*

Preuve :
La preuve de ce lemme est immédiate en utilisant le lemme 2.1.8 (page 63). □

Pour les lemmes qui suivent, nous considérons k un entier fixé, L et L' deux langages k-réversibles avec $L' \subset L$. Soient S_L^k et $S_{L'}^k$ les échantillons caractéristiques de L et L'. Les automates canoniques respectifs de ces langages sont $A = (Q, \{q_0\}, F, \Sigma, \delta)$ et $A' = (Q', \{q_0'\}, F', \Sigma, \delta')$. Soit n le nombre d'états de A et n' celui de A'. Soit q un état quelconque de A (respectivement de A'). Par souci de simplicité, on note L_q (resp. L_q') l'ensemble des k-meneurs de q, c'est à dire $L_q = \{u \in \Sigma^k \mid \exists\ q' \in Q$ tel que $\delta(q', u) = q\}$ (resp. $L_q' = \{u \in \Sigma^k \mid \exists\ q' \in Q'$ tel que $\delta'(q', u) = q\}$). De plus, $u(q, x)$ (resp. $u'(q, x)$) représente $u_A(q, x)$ (resp. $u_{A'}(q, x)$) et $v(q)$ (resp. $v'(q)$) représente $v_A(q)$ (resp. $v_{A'}(q)$).

Lemme A.3.3 *Sous les hypothèses précédemment énoncées, $\exists\ u \in S_L^k$ tel que $u \notin S_{L'}^k$.*

Preuve :
Ce lemme est une conséquence directe de la définition des échantillons caractéristiques. On rappelle qu'un échantillon caractéristique S_L^k d'un langage k-réversible L est tel que L est le plus petit langage k-réversible contenant S_L^k. Par conséquent L' est le plus petit langage k-réversible contenant $S_{L'}^k$; de même L est le plus petit langage k-réversible contenant S_L^k.

Supposons qu'un tel mot u n'existe pas, alors $S_L^k \subseteq S_{L'}^k$. Dans ce cas, le plus petit langage contenant S_L^k est L', ce qui contredit les hypothèses. □

Lemme A.3.4 *Sous les hypothèses précédemment énoncées,* $(S_L^k - S_{L'}^k) \cap L' = \emptyset$.

Preuve :
Nous allons montrer que s'il existe un mot appartenant à $(S_L^k - S_{L'}^k) \cap L'$ alors ce mot appartient aussi à $S_{L'}^k$, ce qui prouvera le lemme par l'absurde. Soit $u \in (S_L^k - S_{L'}^k) \cap L'$. Le mot u appartient à S_L^k donc u est soit de taille inférieure ou égale à k (*ie* $u \in L^{\leq k}$), soit de la forme $u(q_1, x)xv(q_1)$ avec $q_1 \in Q$ et $x \in L_{q_1}$ soit de la forme $u(q_1, x)xav(q_2)$ avec $q_1, q_2 \in Q, a \in \Sigma, x \in L_{q_1}, q_2 = \delta(q_1, a)$.

- **Cas 1: $u \in L^{\leq k}$.** Si le mot u appartient à L', alors par construction de $S_{L'}^k$, u appartient aussi à $S_{L'}^k$ (puisque $L'^{\leq k} \subseteq S_{L'}^k$). Ceci contredit l'hypothèse initiale et prouve donc le lemme pour ce cas.

- **Cas 2: u est de la forme $u(q_1, x)xv(q_1)$ avec $q_1 \in Q$ et $x \in L_{q_1}$.** Si le mot u appartient à L' alors il existe un état $q_1' \in Q'$ tel que $q_1' = \delta'(q_0', u(q_1, x)x)$ et $\delta'(q_1', v(q_1)) \in F$. Comme $u'(q_1', x)$ est la plus petite chaîne σ telle que $\delta'(q_0, \sigma x) = q_1'$ on a,

$$u'(q_1', x) \leq u(q_1, x)$$

De plus, le mot $u'(q_1', x)xv(q_1)$ appartient à L' et donc aussi à L étant donné que $L' \subseteq L$. On a donc $u = u(q_1, x)xv(q_1) \in L$ (par hypothèse sur u) et $u'(q_1', x)xv(q_1) \in L$. Le mot $xv(q_1)$ est de longueur supérieure ou égale à k donc en appliquant le lemme A.3.2 (page précédente), on a,

$$\delta(q_0, u(q_1, x)) = \delta(q_0, u'(q_1', x))$$

Par définition de $u(q_1)$, on a

$$u(q_1, x) \leq u'(q_1', x)$$

Et donc finalement,

$$u(q_1) = u'(q_1')$$

Maintenant, la chaîne $v'(q_1')$ étant la plus petite chaîne menant de q_1' vers tout état terminal, on a,

$$v'(q_1') \leq v(q_1)$$

Le mot $u'(q_1', x)xv'(q_1') = u(q_1, x)xv'(q_1')$ appartient à L' (puisque c'est un mot de $S_{L'}^k$), il appartient donc également à L. Par définition de $v(q_1)$,

$$v(q_1) \leq v'(q_1')$$

Ce qui implique que,

$$v(q_1) = v'(q_1')$$

Par conséquent, $u = u(q_1, x)xv(q_1) = u'(q_1', x)xv'(q_1')$. Par construction de $S_{L'}^k$, u appartient donc à cet échantillon, ce qui contredit l'hypothèse de départ et prouve le lemme pour ce deuxième cas.

- **Cas 3: u est de la forme $u(q_1,x)xav(q_2)$ avec $q_1,q_2 \in Q, a \in \Sigma, x \in Lq_1,q_2 = \delta(q_1,a)$.** Si le mot u appartient à L', il existe alors deux états $q'_1, q'_2 \in Q'$ tels que $\delta'(q'_0, u(q_1,x)x) = q'_1$, $\delta'(q'_1,a) = q'_2$ et $\delta'(q'_2, v(q_2)) \in F$. Le même raisonnement que précédemment montre que,

$$u(q_1,x) = u'(q'_1,x)$$

Et que,

$$v(q_2) = v'(q'_2)$$

Par construction de $S^k_{L'}$, le mot $u = u(q_1,x)xav(q_2) = u'(q'_1,x)xav'(q'_2)$ appartient donc à $S^k_{L'}$ ce qui contredit ici encore l'hypothèse de départ et termine la preuve.

□

Le lemme suivant montre que si u est un mot appartenant à S^k_L et non à L', alors il suffit de lire au maximum les $2n'+k$ premières lettres de u pour voir que u n'est pas un mot de L'. Ce lemme est d'une importance capitale pour assurer la polynomialité de l'algorithme d'apprentissage, puisque si un tel mot u appartient à l'échantillon, il permettra de réfuter une mauvaise hypothèse en un temps pas trop long.

Lemme A.3.5 *Sous les hypothèses précédemment énoncées, si $u \in S^k_L - L'$ alors soit $|u| < 2(n'+k)$ et $u \notin L'$, soit $|u| \geq 2(n'+k)$ et $Pre_{2(n'+k)}(u) \notin Pr(L')$.*

Preuve :
Soit $u \in S^k_L - L'$. Si $|u| < 2(n'+k)$, il n'y a rien à prouver (ce cas concerne entre autre les mots $u \in L^{\leq k}$). Supposons maintenant que $|u| \geq 2(n'+k)$. Le mot u appartient à S^k_L, il est donc de la forme $u = u(q,x)xv(q)$ avec $q \in Q, x \in L_q$ (le cas où $u = u(q_1,x)xav(q_2)$ avec $q_1,q_2 \in Q, a \in \Sigma, x \in L_{q_1}, q_2 = \delta(q_1,a)$ est similaire). Soit $w = Pre_{2(n'+k)}(u)$. Nous allons montrer par l'absurde que w ne peux pas être préfixe d'un mot de L'. Pour ce faire, supposons que w est le préfixe d'un mot de L', c'est-à-dire qu'il existe un mot $v \in \Sigma^*$, tel que $wv \in L'$. Montrons qu'alors $|u| = |u(q,x)| + |x| + |v(q)| < 2(n'+k)$.

- **Montrons d'abord que $|u(q,x)| \leq n' + k - 1$.** Comme $|w| = 2(n'+k)$ et que w est le préfixe d'un mot de L', la chaîne $Pre_{n'+k}(w)$ labelle un chemin passant deux fois par un même état de A'. Il existe donc un mot s de longueur inférieure à $n'-1$ tel que,

$$\delta'(q'_0, s) = \delta'(q'_0, Pre_{n'+k}(w))$$

Comme $|w| = 2(n'+k)$, il est évident que,

$$w = Pre_{n'+k}(w)Suf_{n'+k}(w)$$

d'où,

$$\delta'(q'_0, sSuf_{n'+k}(w)v) = \delta'(q'_0, wv)$$

Comme $wv \in L'$, on a même

$$\delta'(q'_0, sSuf_{n'+k}(w)v) = \delta'(q'_0, wv) \in F'$$

On a donc $sSuf_{n'+k}(w)v \in L'$ et comme $L' \subset L$, on a aussi $sSuf_{n'+k}(w)v \in L$. Supposons que $|u(q,x)| \geq n'+k$. Dans ce cas, comme $w = Pre_{2(n'+k)}(u(q,x)v(q))$,

$$Pre_{n'+k}(u(q,x)) = Pre_{n'+k}(w)$$

Étant donné que puisque $wv \in L$, cela implique que,

$$Pre_{n'+k}(u(q,x))Suf_{n'+k}(w)v \in L$$

Par application du lemme A.3.2 (page 200) sur les mots $sSuf_{n'+k}(w)v$ et $Pre_{n'+k}(u(q,x))Suf_{n'+k}(w)v$, on obtient,

$$\delta(q_0, s) = \delta(q_0, Pre_{n'+k}(u(q,x))) = q_1$$

D'après le lemme 3.3.4 (page 101) (au travers de la remarque A.1.1 (page 197)), on a,

$$Pre_{n'+k}(u(q,x)) = u(q_1)$$

Mais, $|s| \leq n'-1$ et $|u(q_1)| = n'+k$ donc $s < u(q_1)$ ce qui est impossible par définition de $u(q_1)$. L'hypothèse selon laquelle $|u(q,x)| \geq n'+k$ n'est donc pas possible et donc $\boldsymbol{|u(q,x)| \leq n'+k-1}$.

- **Montrons maintenant que $|v(q)| \leq n'-1$.** Nous savons maintenant que w peut s'écrire $w = u(q,x)xw'$ avec $|w'| > n'$ (puisque $|w| = 2(n'+k)$, $|u(q,x) < n'+k$ et $|x| = k$). Intéressons-nous à ce mot w' qui peut aussi s'écrire $w' = Suf_{2(n'+k)-|u(q,x)|-k}(w)$. Soit $q'_1 = \delta'(q'_0, u(q,x)x)$, comme $wv \in L'$, on a donc,

$$\delta'(q'_1, w'v) \in F'$$

La chaîne $Pre_{n'}(w')$ labelle un chemin depuis q'_1 et passant deux fois par un même état de A' puisque A' contient n' états. Par conséquent, il existe un mot t de longueur inférieure ou égale à $n'-1$, labellant un chemin depuis q'_1 vers $\delta(q'_1, Pre_{n'}(w'))$. On a donc,

$$\delta(q'_1, tSuf_{|w'|-n'}(w')v) = \delta'(q'_1, w'v) \in F'$$

Et même,

$$\delta(q'_0, u(q,x)xtSuf_{|w'|-n'}(w')v) = \delta'(q'_0, u(q,x)xw'v) \in F'$$

Ainsi, le mot $u(q,x)xtSuf_{|w'|-n'}(w')v$ appartient à L' et par conséquent aussi à L (puisque $L' \subset L$).

Il en est de même pour le mot $u(q,x)xw'v$ qui peut aussi s'écrire

$$u(q,x)xPre_{n'}(w')Suf_{|w'|-n'}(w')v$$

. En utilisant le lemme A.3.2 (page 200) sur les chaînes $u(q,x)xtSuf_{|w'|-n'}(w')v \in L$ et $u(q,x)xPre_{n'}(w')Suf_{|w'|-n'}(w')v \in L$ (qui peut s'appliquer puisque $|w'|-n' > k$) on a,

$$\delta(q_0, u(q,x)xt) = \delta(q_0, u(q,x)xPre_{n'}(w'))$$

Soit encore étant donné que $\delta(q_0, u(q,x)x) = q$,

$$\delta(q,t) = \delta(q, Pre_{n'}(w'))$$

Supposons que $|v(q)| \geq n'$. Comme $|w'| > n' + k$ et $w = Pre_{2(n'+k)}(u(q,x)xv(q)) = u(q,x)xw'$, on a alors,

$$Pre_{n'}(v(q)) = Pre_{n'}(w')$$

Soit $q_1 = \delta(q,t)$, nous avons alors,

$$q_1 = \delta(q,t) = \delta(q, Pre_{n'}(v(q)))$$

Et par conséquent,

$$\delta(q, tSuf_{|v(q)|-n'}(v(q))) = \delta(q, Pre_{n'}(v(q))Suf_{|v(q)|-n'}(v(q))) = \delta(q, v(q)) \in F$$

Or $|t| \leq n' - 1$, donc,

$$|tSuf_{|v(q)|-n'}(v(q))| \leq n' - 1 + |v(q)| - n'$$

Soit encore,

$$|tSuf_{|v(q)|-n'}(v(q))| \leq |v(q)| - 1$$

Il existe donc un mot de longueur inférieur à $v(q)$ labellant un chemin entre q et un état terminal de A. Ceci est impossible par définition de $v(q)$, et donc $\boldsymbol{|v(q)| \leq n' - 1}$.

Finalement, nous avons montré que $|u| = |u(q,x)| + |x| + |v(q)| \leq 2(n' + k) - 2$ ce qui contredit l'hypothèse initiale. Donc le préfixe w de longueur $2(n' + k)$ de u ne peut pas être un préfixe d'un mot de L'. □

Les lemmes que l'on vient d'énoncer montre donc qu'il existe des mots distinguant L et L' et appartenant à S_L^k. De plus, il est suffisant de ne lire qu'un préfixe de longueur au plus $2(n' + k)$ de ceux-ci pour décider de la non appartenance à L'. Le lemme suivant établit que si l'on tire un échantillon de taille suffisante (mais polynomiale) suivant la distribution de probabilité $\mathbf{m}_L$, alors un tel mot a de grandes chances d'appartenir à cet échantillon.

Lemme A.3.6 *Sous les hypothèses énoncées, il existe un mot $u \in S_L^k - L'$ tel que $\mathbf{m}_r(u) \geq \lambda/(n' \log^2(n'))$ où r est un nom pour L et λ est une constante dépendant uniquement de la machine de Turing préfixe de référence U. Il existe de plus un polynôme q ne dépendant que de U tel qu'un échantillon S tiré selon $\mathbf{m}_r$ et de taille supérieure à $q(n', 1/\delta)$ contient un mot $u \in S_L^0 - L'$ avec la probabilité au moins $1 - \delta/2^{n'}$.*

Preuve :
Étant donné L, par définition l'échantillon S_L^k est calculable. D'après le lemme A.3.3 (page 200), il existe au moins un mot appartenant à S_L^k et pas à $S_{L'}^k$. De plus, il existe un tel mot u dans les $|\Sigma|^k.(|\Sigma|+1).n' + \frac{1-|\Sigma|^{k+1}}{1-|\Sigma|} + 1$ premiers mots de S_L^k puisque $|S_{L'}^k| \leq |\Sigma|^k.(|\Sigma|+1).n' + \frac{1-|\Sigma|^{k+1}}{1-|\Sigma|}$ (voir proposition A.1.1, page 197). Ce mot u n'appartient pas à L' d'après le lemme A.3.4 (page 201).

Montrons que pour un tel mot u, nous avons $K(u \mid r) \leq \log(n') + 2\log(\log(n')) + O(1)$ où r est un nom pour L. Pour générer u, il est suffisant de disposer de l'échantillon caractéristique S_L^k et de l'index i de u dans celui-ci. Aussi, d'après le théorème 1.4.3 (page 42),

$$K(u \mid S_L^0) \leq K(i) + O(1) \leq \log(i) + 2\log(\log(i)) + O(1)$$

Le mot u se trouve dans les $|\Sigma|^k.(|\Sigma|+1).n' + \frac{1-|\Sigma|^{k+1}}{1-|\Sigma|} + 1$ premiers mots de S_L^k donc en supposant que $n' \geq 1$,

$$i \leq |\Sigma|^k.(|\Sigma|+1).n' + \frac{1-|\Sigma|^{k+1}}{1-|\Sigma|} + 1 \leq C_{k,|\Sigma|}n'$$

avec $C_{k,|\Sigma|}$ une constante ne dépendant que de $|\Sigma|$ et de k.

D'où,

$$K(u \mid S_L^k) \leq \log(n') + 2\log(\log(n')) + O(1)$$

De plus, par définition de S_L^k, le programme construisant S_L^k connaissant r a une longueur constante donc,

$$K(S_L^k \mid r) \leq O(1)$$

On trouve finalement,

$$K(u \mid r) \leq K(u \mid S_L^k) + K(S_L^k \mid r) + O(1) \leq \log(n') + 2\log(\log(n')) + O(1)$$

Par conséquent, il existe λ tel que,

$$\mathbf{m}_r(u) \geq \lambda/(n'\log^2(n'))$$

La preuve montrant l'existence d'un polynôme q est la même que celle donnée pour le cas des langages 0-réversibles (voir preuve du théorème 3.3.8, page 106). □

Nous pouvons à présent donner la preuve du théorème 3.3.10 (page 109). Étant donnée les lemmes qui précédent et leur similarité avec ceux établis dans le cas des langages 0-réversibles, on comprendra que la preuve qui suit est également très semblable à celle concernant les langages 0-réversibles (voir théorème 3.3.2, page 101).

Preuve (du théorème 3.3.10 (page 109)) :

Soit un langage k-réversible L de nom r et n le nombre d'états de l'automate canonique de L. La preuve du théorème A.2.1 (page 198) établit que les échantillons S_L^k sont c-simples. Par la suite, p représente le polynôme tel que si l'on tire un échantillon S selon $\mathbf{m}_r$ de taille supérieure à $p(n, 1/\delta)$ alors $S_L^0 \subseteq S$ avec une probabilité supérieure à $1-\delta$ (voir lemme 3.1.3 , page 90).

Considérons l'algorithme de PACS-apprentissage qui suit:

Algorithme de PACS-apprentissage A
Entrée : δ
Constantes connues : k
Début

Soit $S \leftarrow \emptyset$.
Soit $n' \leftarrow 1$
Soit $NEX \leftarrow 0$
Tant que pas Fin_du_monde
  Faire $p(n', 2/\delta) - NEX$ appels à $Exemple_{\mathbf{m}_r}(L)$
  Ajouter les nouveaux exemples à S
  Soit S' l'ensemble des exemples positifs de S de longueur
  inférieure ou égale à $2n' + k + 1$
  Soit A' l'automate retourné par k-RI(S')
  Si A' a n' états
  Alors
    Faire $q(n', 2/\delta)$ appels à $Exemple_{\mathbf{m}_r}(L)$
    Soit $TEST$ l'ensemble des nouveaux exemples
    Si Le préfixe de longueur $2(n' + k)$ de tous les mots de $TEST$
    sont des préfixes de mots acceptés par A'
    Alors Retourner A' et s'arréter
    Fsi
  Fsi
  $NEX \leftarrow p(n', 2/\delta)$
  $n' \leftarrow n' + 1$
Ftq
Fin

La preuve permettant de montrer que cet algorithme est bien un algorithme d'apprentissage probablement correct dans notre modèle est identique à celle donnée pour le cas des langages 0-réversibles et montre que A retourne un nom pour L avec la probabilité supérieure à $(1 - \delta/2)^2 \geq 1 - \delta$. On peut alors vérifier que A tourne en temps polynomial en $1/\delta$ puisque l'algorithme k-RI tourne en temps polynomial. □

Bibliographie

[AD91] J.P. Aubert and P. Dixneuf. *Conception et Programmation par Objets.* Masson, 1991.

[Ang78] D. Angluin. On the complexity of minimum inference of regular sets. *Inform. Control*, 39(3):337–350, 1978.

[Ang80] D. Angluin. Inductive inference of formal languages from positive data. *Inform. Control*, 45:117–135, 1980.

[Ang82a] D. Angluin. Inference of reversible languages. *J. ACM*, 29(3):741–765, July 1982.

[Ang82b] D. Angluin. A note on the number of queries needed to identify regular languages. *Inform. Control*, 51:76–87, 1982.

[Ang87] D. Angluin. Learning regular sets from queries and counterexample. *Inform. Control*, 75:87–106, 1987.

[Ang88] D. Angluin. Queries and concept learning. *Machine Learning*, 2(4):319–342, April 1988.

[AS83] D. Angluin and C.H. Smith. A survey of inductive inference: Theory and methods. *ACM Comput. Surv.*, 15(3):237–269, September 1983.

[BEHW87] A. Blumer, A. Ehrenfeucht, D. Haussler, and M. K. Warmuth. Occam's razor. *Information Processing Letters*, 24:377–380, April 1987.

[BFJ+94] A. Blum, M. Furst, J. Jackson, M. Kearns, Y. Mansour, and S. Rudich. Weakly learning DNF and characterizing statistical query learning using fourier analysis. In *Proc. 26th ACM Symposium on Theory of Computing*, pages 253–262, 1994.

[BI91] G. Benedek and A. Itai. Learnability with respect to fixed distributions. *Theoret. Comput. Sci.*, 86(2):377–389, 1991.

[Boo94] G. Booch. *Object-Oriented Analysis and Design with Applications.* Addison-Wesley, 2 edition edition, 1994.

[BP90] R. Board and L. Pitt. On the necessity of occam algorithms. In *Proc. of the 22th ACM Symposium on Theory of Computing*, pages 54–63, 1990.

[Cas95] J. Castro. A note on learning decision lists. Technical Report LSI-95-2-R, Dept. LSI, UPC, 1995.

[CB95] J. Castro and J.L. Balcàzar. Simple PAC learning of simple decision lists. In *International Workshop on Algorithmic Learning Theory*, 1995.

[CG98] J. Castro and D. Guijarro. Query, PACS and simple-PAC learning. Technical Report LSI-98-2-R, Dpt de Llenguatges i sistemes informàtics, Universitat Politéca de Catalunya, 1998.

[Cho56] N. Chomsky. Three models for the description of languages. *PGIT 2*, 3:113–124, 1956.

[DdG96] F. Denis, C. d'Halluin, and R. Gilleron. PAC learning with simple examples. In *Proc. of the 13th Symposium on Theoretical Aspects of Computer Science*, pages 231–242. Springer Verlag, 1996.

[Del93] J.P Delahaye. *Information, Complexité et Hasard.* Langue-Raisonnement-Calcul. Hermes, 1993.

[DG97] F. Denis and R. Gilleron. PAC learning under helpful distribution. In *Proceeding of ALT'97 in Lecture Notes in Artificial Intelligence*, volume 1316, pages 132–146, 1997.

[DGS97] F. Denis, R. Gilleron, and J. Simon. Apprentissage PAC avec enseignant. In *Actes de JFA 97*, 1997.

[dlH97] C. de la Higuera. Characteristic sets for grammatical inference. *Machine Learning*, 27:1–14, 1997.

[dlHOV96] C. de la Higuera, J. Oncina, and E. Vidal. Identification of DFA : data-dependant versus data-independant algorithms. In *ICGI'94*, 1996.

[DMV94] P. Dupont, L. Miclet, and E. Vidal. What is the search space of the regular inference? In *ICGI'94*, pages 25–37. Springer Verlag, 1994.

[Dup94] P. Dupont. Regular grammatical inference from positive and negative samples by genetic search: the GIG method. In *ICGI'94*, pages 236–245. Springer Verlag, 1994.

[Dup96] P. Dupont. Utilisation et apprentissage de modèles de langages pour la reconnaissance de la parole continue. Technical Report Thèse de Doctorat, ENST, Paris, 1996.

[FB75] K.S. Fu and T.L. Booth. Grammatical inference: Introduction and survey. In *IEEE Transactions on SMC*, volume 5, pages Part 1: 85–111, Part 2: 409–423, 1975.

[FGHR69] J.A. Feldman, J. Gips, J.J. Horning, and S. Reder. Grammatical complexity and inference. Technical Report Artificial Intelligence Project, Stanford University, Stanford, Californie, 1969.

[Fon97] A.B. Fontaine. *La bibliothèque standard STL du C++.* InterEdition, 1997.

[Gas93] O. Gascuel. Aspects statistiques de l'apprentissage inductif. In *Support de cours, école sur l'apprentissage automatique*, pages 33–54, 1993.

[Gin66] A. Ginzburg. About some properties of definite, reverse-definite and related automata. *IEEE Trans. Electon. Comput.*, EC-15:806–810, 1966.

[GK91] S.A. Goldman and M.J. Kearns. On the complexity of teaching. In *Proceedings of COLT'91*, pages 303–314, 1991.

[GM93] S.A. Goldman and H.D. Mathias. Teaching a smarter learner. In *Proceedings of COLT'93*, pages 67–76, 1993.

[Gol67] E.M. Gold. Language identification in the limit. *Inform. Control*, 10:447–474, 1967.

[Gol78] E.M. Gold. Complexity of automaton identification from given data. *Inform. Control*, 37(3):302–320, 1978.

[HS92] K.U. Hoffgen and H.U. Simon. Lower bounds on learning decision lists and trees. In *Proc. of the 5th ACM workshop on Computational Learning Theory*, pages 428–439. Springer Verlag, 1992.

[HU79] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, Reading, MA, 1979.

[JT92] J. Jackson and A. Tomkins. A computational model of teaching. In *Proceedings of COLT'92*, pages 319–326, 1992.

[KLPV87] M. Kearns, M. Li, L. Pitt, and L.G. Valiant. On the learnability of boolean formulae. In *Proc. 19th ACM Symposium on Theory of Computing*, pages 285–295, 1987.

[Kra49] L.G Kraft. A device for quantizing grouping and coding amplitude modulated pulse. Technical Report M. Sc. Thesis, MIT, Dpt. Elect. Eng., Cambridge, Massachusetts, 1949.

[KSS92] M. Kearns, R. Shapire, and L. Sellie. Toward efficient agnostic learning. In *Proc. of the 5th ACM workshop on Computational Learning Theory*, pages 341–352. Springer Verlag, 1992.

[KV92] M.J. Kearn and U.V. Vazirani. *An Introduction to Computational Learning Theory.* MIT Press, 1992.

[KV94] M.J. Kearns and L.G. Valiant. Cryptographic limitations on learning boolean formulae and finite automata. *Journal of the Association for Computing Machinery*, 41:67–95, 1994.

[KY97] S. Kobayashi and T. Yokomori. Learning approximately regular languages with reversible languages. *Theoretical Computer Science*, 174:251–257, 1997.

[Lan92] K.J. Lang. Random dfa's can be approximately learned from sparse uniform examples. In *Proc. of the 5th ACM workshop on Computational Learning Theory*, pages 45–52. Springer Verlag, 1992.

[LV91] M. Li and P.M.B. Vitányi. Learning simple concepts under simple distributions. *SIAM J. Comput.*, 20:911–935, 1991.

[LV92] M. Li and P.M.B. Vitányi. Inductive reasoning and kolmogorov complexity. *Journal of Computer and System Sciences*, 44:343–384, 1992.

[LV93] M. Li and P.M.B. Vitányi. *An Introduction to Kolmogorov Complexity and Its Applications.* Text and Monographs in Computer Science. Springer-Verlag, 1993.

[MCM83] R.S. Michalski, J.G. Carbonell, and T.M. Mitchell, editors. *Machine Learning: An Artificial Intelligence Approach*, volume I. Morgan Kaufmann, Los Altos, California, 1983.

[McN67] R. McNaughton. The loop complexity of pure-group events. *Inform. Control*, 11:167–176, 1967.

[MdG94] L. Miclet and C. de Gentille. Inférence grammaticale à partir d'exemples et de contre-exemples: deux algorithmes optimaux (big et rig) et une version heuristique (brig). In *JAVA94*, pages F1–F13, Strasbourg, France, 1994.

[Mic79] L. Miclet. Inférence de grammaires régulière. Technical Report Thèse de Docteur-Ingénieur, E.N.S.T, Paris,France, 1979.

[Mic80] L. Miclet. Regular inference with a tail-clustering method. In *IEEE Transactions on SMC*, volume 10, pages 737–743, 1980.

[Mic84] L. Miclet. *Méthodes structurelles pour la reconnaissance des formes.* Eyrolles, Paris, 1984.

[Mor92] H. Moravec. *Une vie après la vie.* Paris, 1992.

[MS91] S. Miyano and A. Shinohara. Teachability in computational learning. *New Generation Computing*, 8:337–347, 1991.

[Muk94] Y. Mukouchi. Inductive inference of an approximate concept from positive data. In *Proc. of the 5th workshop on Algorithmic Learning Theory*, pages 484–499. Springer Verlag, 1994.

[Mul97] P.A. Muller. *Modélisation Objet avec UML.* Eyrolles, 1997.

[Nat87] B.K. Natarajan. On learning boolean functions. In *Proc. 19th ACM Symposium on Theory of Computing*, pages 296–304, 1987.

[Nat91] B.K. Natarajan. *Machine Learning: A Theoretical Approach.* Morgan Kaufmann, San Mateo, CA, 1991.

[OG92] J. Oncina and P. Garcia. Inferring regular languages in polynomial update time. *Pattern Recognition and Image Analysis*, pages 49–61, 1992.

[PC78] T. Pao and J. Carr. A solution of syntactic induction-inference problem for regular languages. *Computer Languages*, 3:53–64, 1978.

[PH97] R. Parekh and V. Honavar. Learning DFA from simple examples. In *ICML'97*, Nashville, Tennessee, 1997.

[Pit89] L. Pitt. Inductive inference, DFAs and computational complexity. In *Analogical and Inductive Inference, Lectures Notes in Artificial Intelligence*, volume 397, pages 18–44. Springer Verlag, 1989.

[PRS63] M. Perles, M.O. Rabin, and E. Shamir. The theory of definite automata. *IEEE Trans. Electon. Comput.*, EC-12:233–243, 1963.

[PV88] L. Pitt and L. Valiant. Computational limitations on learning from examples. *J. ACM*, 35:965–984, 1988.

[PW90] L. Pitt and M.K. Warmuth. Prediction-preserving reductibility. *Journal of Computer and System Sciences*, 41(3):430–467, December 1990.

[PW93] L. Pitt and M.K. Warmuth. The minimum consistent DFA problem cannot be approximated within any polynomial. *Journal of the Association for Computing Machinery*, 40(1):95–142, January 1993.

[QR89] J.R. Quinlan and R.L. Rivest. Inferring decision trees using the minimum description length principle. *Information and Computation*, 3(80):227–248, 1989.

[Ron95] D. Ron. Automata learning and its application. Technical Report Phd, Hebrew University, Israel, 1995.

[RW94] Darrell Raymond and Derick Wood. Grail: A C++ Library for Automata and Expressions. *Journal of Symbolic Computation*, 17:341–350, 1994.

[SM94] A. Stepanov and M.Lee. The Standard Template Library. Technical Report X3J16/94-0095,WG21/No482, ISO Programming Language C++ Project, 1994.

[Str92] B. Stroustrup. *Le langage C++*. Addison-Wesley, 2 edition edition, 1992.

[TB73] B. Trakhtenbrot and Ya. Barzdin. *Finite Automata: Behaviour and Synthesis*. North Holland Pub. Comp., Amsterdam, 1973.

[Val84] L.G. Valiant. A theory of the learnable. *Commun. ACM*, 27(11):1134–1142, November 1984.

THESES

1997

06 Janvier — **Philippe MESEURE**
Modélisation de corps déformables pour la simulation d'actes chirurgicaux.

07 Janvier — **Olivier ROUSSEL**
L'achèvement des bases de connaissances en calcul propositionnel et en calcul des prédicats.

08 Janvier — **Nouredine MELAB**
Gestion de la granularité et régulation de charge dans le modèle P^3 d'évaluation parallèle des langages fonctionnels

09 Janvier — **Cédric DUMOULIN**
Dream : une mémoire partagée répartie à cohérence programmable.

13 Janvier — **Raymond NAMYST**
PM^2 : Un environnement pour une conception portable et une exécution efficace des applications parallèles irrégulières.

14 Janvier — **Philippe MERLE**
CorbaScript - Corba Web : propositions pour l'accès à des objets et services distribués.

27 Janvier — **David GALINEC**
Exécution asynchrone de programmes synchrones par transformations automatiques : application au traitement d'images temps-réel.

14 Mars — **Côme RACZY**
Commandes optimales en temps pour des systèmes différentiellement plats. Application aux commandes de satellites et de grues.

27 Mars — **Jean Jacques VANDEWALLE**
Projet OSMOSE : Mosélisation et implémentation pour l'interopérabilité de services carte à microprocesseur par l'approche orientée objet.

10 Juin — **David SIMPLOT**
Langages de mots de figures, monoïdes inversifs et langages de mots à deux dimensions

20 Juin — **Dominique SUEUR**
Algorithmes de redistribution de données. Application aux systèmes de fichiers parallèles distribués.

25 Juin

Olivier DELGRANGE

Un algorithme rapide pour une compression modulaire optimale. Application à l'analyse de séquences génétiques.

THESES

1998

09 Janvier — **David CARLIER**
Représentation permanente, coordonnée par une carte à microprocesseur, d'un utilisateur mobile.

16 Janvier — **Yves DENNEULIN**
Conception et ordonnancement des applications hautement irrégulières dans un contexte de parallélisme à grain fin.

23 Janvier — **Jaafar GABER**
Plongements et manipulations d'arbres dans les architectures distribuées.

30 Janvier — **Grégory SAUGIS**
Interfaces 3D pour le travail coopératif synchrone, une proposition.

02 Juin — **Bruno MARCHAL**
Calculabilité, Physique et Cognition..