Introduction aux méthodes I.A. (Brouillon)

Marc-Michel Corsini
UFR S. & M. Université de Bordeaux 2
146 rue Léo Saignat
33076 Bordeaux Cedex (France)
e-mail: corsini@u-bordeaux2.fr

16 mars 2011

Table des matières

Première partie

Invitation à l'IA

Chapitre 1

Intelligence Artificielle : Réalité et Applications

1.1 Avis au lecteur

Le présent texte n'est qu'un brouillon nullement exempt de fautes tant syntaxiques que grammaticales. Les opinions avancées n'engagent que moi. Ceci est la version **1.2**[1] de ces notes de cours[2]. Certaines parties sont en cours de rédaction, parfois ne transparaissent que les grandes lignes (dans certains cas réduites aux têtes de chapitres), les éléments complémentaires seront donnés en cours.

L'objectif avoué est de vous sensibiliser à cette approche et vous donner envie d'approfondir le sujet. Ces notes sont, comme l'état de l'art qu'elles décrivent, provisoires. Toutes améliorations, critiques, commentaires seront les bienvenus.

1.2 Introduction

Après plus de 30 ans de recherches et peu d'applications significatives, l'IA est devenue un des thèmes porteurs de l'informatique. De nombreux industriels manifestent leur intérêt IBM avec CLP($\mathcal{R}$), Bull/Siemens/ICL avec CHIP, des entreprises spécialisées se sont créées (Delphia à Grenoble, Prologia à Marseille, Cosytec à Orsay, BIM en Belgique ...). Devant cet état de faits les réactions vont du scepticisme à l'emerveillement le plus complet.

L'objectif de l'IA est double : le premier concerne l'analyse théorique et pratique des processus cognitifs[3] ; le second s'intéresse à la réalisation (« hard »et « soft ») d'artéfacts intelligents. Pour certains, l'IA se place parmi les sciences « sociales »(psychologie, linguistique, philososphie), « naturelles »(neurobiologie, physiologie), et a une approche oscillant entre une méthode rigoureuse et un bricolage effréné. Pour d'autres, il ne s'agit que d'une branche de l'informatique au même titre que l'infographie, la combinatoire, l'informatique théorique.

Déjà le terme « intelligence artificielle »est assez flou, puisque si l'*artificiel* a été conçu par la main de l'homme (ou d'une, voire plusieurs, machine(s) conçue(s) par ...) par opposition au *naturel* ; la notion d'intelligence est quant à elle difficile à préciser : un plombier est-il plus ou moins « intelligent »qu'un mathématicien ?

Les études psychologiques ont démontré que cette appellation (non contrôlée) recouvre de nombreuses capacités très différentes :

a. pouvoir d'abstraction,
b. résolution de problèmes,

[1] version Avril 2003

[2] La première version date de 1986 où ce cours fut donné en DESS Informatique et Productique (86-92). Depuis plusieurs chapitres ont été ajoutés au grè de mes intérêts

[3] qui a trait au raisonnement et à la connaissance par la pensée

c. adaptation (rapide) à un nouvel environnement,
d. culture générale,
e. raisonnement, créativité ....

De plus, une fois connus les mécanismes permettant d'accomplir une tâche, l'homme a tendance à en nier la qualité d'intelligence.

En résumé, l'IA peut être vue comme « l'utilisation de l'ordinateur comme moyen de simulation des processus naturels ou comme support des capacités ordinairement attribuées à l'intelligence humaine ». L'objectif est donc vaste puisqu'il englobe aussi bien la capacité qu'aurait une machine à jouer aux echecs, traduire un texte, ou découvrir, sur une photo la présence de Monsieur Untel. Son but est d'analyser les comportements humains dans les domaines de la compréhension, de la perception, de la résolution de problèmes afin de les reproduire sur machine. Pour se faire une idée des champs d'applications et d'investigations (au milieu des années 80) je renvoie le lecteur à « la RECHERCHE en Intelligence artificielle » [25] et, dans une moindre mesure au premier chapitre de « Les Sciences Cognitives une introduction » [98]. Le lecteur se reportera avec bonheur aux livres « Intelligence Artificielle et Psychologie Cognitive » [19] et « à la recherche de l'I.A. » [30]. En décembre 1998, la revue *Pour la Science* a publié un numéro spécial sur l'intelligence qui offre une assez bonne vision de l'état de la recherche en ce domaine.

Dans [3], les auteurs considèrent deux conceptions différentes : la première « cognitive »est essentiellement pluri-disciplinaire et son but serait *la réalisation de programmes imitant dans leur fonctionnement l'esprit humain* ; la seconde « pragmatiste »considère que l'IA n'est pas une fin en soi mais l'élaboration de moyens pour développer des théories permettant d'améliorer notre capacité à programmer « efficacement »un ordinateur.

1.2.1 Différentes formes d'IA

Les chercheurs en IA ne sont pas tous d'accord sur ce que l'on est en droit d'attendre de ses recherches. En fait il existe deux courants principaux si l'on en juge par l'approche proposée par R. Penrose dans [74], le premier dit opérationaliste et le second les tenants de l'IA forte.
Pour les opérationalistes, l'ordinateur est réputé « penser »dès lors qu'il agit de façon telle qu'on ne puisse faire la différence avec une personne qui agit en pensant réellement. Cela ne signifie pas que l'ordinateur doit se déplacer ou ressembler ou provoquer la même sensation lorsqu'on le touche ; ces attributs n'ayant rien à voir avec ce pourquoi on a construit un ordinateur. Par contre, on exige qu'il produise des réponses « humaines »à toute question qu'il nous plaira de lui poser. L'expérimentation ne sera concluante (nous serons prêts à affirmer qu'il pense, comprend, a des sentiments, ...) que s'il n'est pas possible de faire la différence entre ses réponses et celles d'un être humain. C'est ce point de vue qui a été défendu par Alan Turing dans un article publié en 1950[4], c'est ce test que l'on appelle **Test de Turing**.

Les partisans de l'IA forte soutiennent qu'il est possible d'attribuer une certaine qualité mentale au fonctionnement logique de n'importe quel dispositif capable de calculer, même au dispositif mécanique le plus simple. Cette affirmation repose sur l'idée que toute activité mentale se ramène simplement à exécuter une suite d'opérations (algorithme). Selon eux, les algorithmes mis en œuvre dans les processus mentaux ont certes un degré de complexité très important (de loin supérieur à celui mis en jeu pour un thermostat) mais il n'y a aucune raison d'en différer au niveau des principes. L'identification faite entre états mentaux et algorithmes est loin de faire l'unanimité. Même si les travaux de Jean-Louis Krivine (portant sur le lambda-calcul comme structure logique sous-jacente de la pensée) ont eu un écho récent dans le mensuel `Science & Vie` (février 2002, pp 38–57) sous le titre raccoleur « L'intelligence dévoile enfin sa vraie nature : Toute pensée est un calcul ! ».

1.2.2 Première approche

En 1956, H. Simon, A. Newell et J.C. Shaw écrivent le premier programme doté d'intelligence : *Logic Theorist* écrit en IPL (ancêtre de Lisp), qui avait pour but le traitement de théorèmes en logique

[4] « Computing Machinery and Intelligence »dans la revue philosophique *Mind*.

symbolique. Pendant la même période, d'autres ingénieurs se penchèrent sur un vieux rêve humain : l'élaboration d'une machine jouant aux échecs[5]. Les premières tentatives ne furent pas concluantes, les ordinateurs de taille respectable (pour l'époque) n'etaient pas capables de trouver des « mats en trois coups », ni de jouer convenablement des débuts de parties. Depuis, on a fait beaucoup mieux. Vers le milieu des années 60, le mythe de l'ordinateur omni-potent était à son apogée (cf. SF [20, 56]). Les informaticiens tentèrent de traduire automatiquement en plusieurs langues des textes qui avaient trait aussi bien à la météorologie, qu'aux techniques boursières ou à la littérature classique.

La technique du mot à mot, par trop simpliste, fut abandonnée au profit d'une analyse grammaticale plus poussée mais ne permettant pas de lever toutes les ambiguités : « Time flies like an arrow »[le temps file comme une flêche] étant traduit par « les mouches à temps aiment une flêche », ce qui n'est guère explicite ....

Une approche totalement différente du problème a permis l'obtention de résultats convenables.

Au début, l'IA était tournée uniquement vers des techniques de l'informatique, depuis elle s'est ouverte à d'autres disciplines telles que la linguistique ou la psychologie, lui permettant d'obtenir quelques succès non négligeables dans le domaine de la traduction ou de le représentation de connaissances.

La résolution de problèmes, les jeux (tactiques) et la traduction automatique ont été le point de départ de son développement. Très vite, les problèmes de reconnaissance de formes, la robotique se sont greffés étendant le champ d'application de l'IA à l'ensemble des activités humaines. A l'heure actuelle de nombreux domaines de recherches sont étiquetés IA parmi ceux-ci il y a :

1. Résolutions de problèmes, systèmes experts, jeux. Regroupant en fait de nombreuses recherches sur le raisonnement et la représentation des connaissances.
2. Réseaux de Neurones formels, approche dite connexioniste.
3. Traitement du langage naturel (traduction semi-automatique).
4. Robotique.
5. Analyse des images et de la parole (reconnaissance de formes).
6. Programmation automatique.
7. C.A.O. et E.A.O. la première ayant de nombreux rapports avec le traitement d'images et la représentation des connaissances, la seconde étant plus liée à la représentation et à l'apprentissage.

Les premiers points de cette énumération sont fondamentaux à l'heure où notre société s'informatise et où les non-informaticiens sont (heureusement) légions. Leurs buts étant de proposer une interface Homme/machine la moins contraignante pour le genre humain.

Aux points précédents, il semble intéressant d'adjoindre un survol des langages de l'IA tels que Lisp (ou un de ses avatars) et Prolog (même remarque) ainsi que quelques techniques afférantes (substitution, pattern matching, backtracking, ...) et de présenter une méthodologie « moderne »de programmation : la programmation par objets ou par acteurs au travers d'un langage particulier : *SmallTalk*. Il est amusant de constater que cette approche est passé du champ de l'IA à celui de l'informatique et notament du `Génie Logiciel` avec la mise au point des langages tels que `Java, C++, Delphi`.

Par ailleurs, la fin des années 1990 a vue l'émergence du *Data Mining* (cf [59]) (appelé aussi « Knowledge Discovery in Database »noté KDD) dont l'objectif est l'extraction de connaissances dans les bases de données. Cette extraction met en jeu des techniques relevant du domaine de l'IA et des Statistiques, les outils à base de réseaux neuromimétiques, d'arbres de décision ou d'algorithmes génétiques font partie intégrante de cette approche.

Ne s'agissant que de notes de cours, nous conseillerons tout au long de ce texte une bibliographie ayant un rapport avec le sujet traité. Il est déjà possible d'identifier trois groupes de source d'information :

[5] 1770, Joueur d'échecs du baron Kempelen, remis au goût du jour par R. Houdin

Vulgarisation : en français, régulièrement des articles ayant trait à l'IA sont publiés dans des revues telles que « Pour la Science », « MicroSystèmes », « Sciences et Avenir », « La Recherche », ... En anglais, dans la revue « Byte »et « PC AI ».
Plus technique : TSI, revue française qui s'adresse aussi bien aux spécialistes, qu'aux industriels.
Spécialisé : en anglais uniquement ce qui permet une plus large diffusion ; on peut citer :
« Journal of Automated Reasonning », « Journal of Logic Programming », « Journal of Symbolic Computation », ... De même que les actes de colloques comme celui sur les systèmes experts, qui se tient annuellement en Avignon, ou les conférences internationales sur la programmation logique (ICLP), la conférence internationale sur l'IA (IJCAI).

1.3 Survol des problèmes traités

Dans cette partie, nous allons présenter diverses applications qui ressortent du domaine de l'intelligence artificielle. Certaines ont été traitées avec succès d'autres n'ont été que partiellement résolues et d'autres sont insolubles dans l'état actuel de nos compétences. Pour un traitement plus approfondi, le lecteur est renvoyé à [25, 30, 39, 57, 19, 3]. Le texte qui suit est un condensé personnel de mes lectures depuis le milieu des années 1980.

1.3.1 Jeux

Dès le début de l'IA, les informaticiens se sont intéressés aux jeux de stratégie pour montrer que les ordinateurs pouvaient faire autre chose que du calcul numérique. Les jeux constituent un excellent domaine d'applications et de bancs d'essais, de plus ils n'ont pas un véritable enjeu social. On trouvera dans [71, 24] de nombreux articles tant sur les jeux de stratégie que sur les méthodes informatiques utilisées pour obtenir une stratégie gagnante.

Les jeux de stratégie mettent les protagonistes en situation de compétition où il leur faut raisonner sur les réponses possibles des adversaires et, au bout du compte, envisager un certain nombre de coups amis et ennemis. Ces jeux répondent à des critères bien particulier :

1. Des règles précises qui décrivent l'ensemble des coups légaux.
2. Une règle qui, pour une situation donnée, indique que la partie est terminée.
3. Une règle qui permet, lorsque la partie est finie, de déterminer le ou les gagnants.

Certains jeux impliquent en fait peu de stratégie et font surtout appel à la mémoire, ainsi le Scrabble version duplicatée[6] ne fait appel à aucune stratégie, pour gagner il faut simplement faire plus de points que ses adversaires. Le programme ATHENA de S. Chapleau et G. Lapalme (1983), joue mieux que les champions du monde en tournoi duplicate. à l'aide d'un dictionnaire (bien structuré), il trouve souvent la solution optimale.

D'autres jeux sont trop simplistes pour être réellement intéressants. C'est notamment le cas lorsque l'espace de tous les coups possibles est suffisamment restreint pour être énuméré rapidement ; dans ce cas l'ordinateur, s'il est bien programmé, n'omettra aucune solution et trouvera donc la solution optimale. Pour certains jeux, tel le jeu de NIM[7], on connait une solution algorithmique simple. Il existe, malgre tout, des jeux pour lesquels il n'y a pas de solution algorithmique comme par exemple les jeux de damier (échec, dames, backgammon, othello, go ...) et les jeux de cartes (bridge, poker). Pour les jeux sur damier, la taille de l'espace des états possibles est considérable, pour les échecs il y aurait 10^{120} coups possibles. Une recherche exhaustive est donc exclue ; la solution choisie est une exploration partielle de l'espace — représenté sous forme d'arbre ou plutôt de graphe orienté sans cycle (DAG) — et pour une profondeur déterminée. De ce fait on est amené à introduire une fonction d'évaluation qui, pour les échecs, dépend des pièces et de la mobilité du jeu comme par exemple :

$$\text{Score} = \text{Matériel} + (\text{Coefficient} * \text{Mobilité})$$

[6] tous les joueurs possèdent le même tirage
[7] jeu d'allumettes à deux joueurs où celui qui retire la dernière a perdu

L'importance d'une telle fonction a été mise en évidence dès 1949, lorsque E. Slater publia des statistiques sur des parties jouées par des grands maîtres. Les résultats indiquaient que la mobilité du vainqueur augmente au dépend de celle de son adversaire. Il est évident que plus la mobilité tend vers zéro, plus le mat est proche. D'autres critères entrent en ligne de compte, comme par exemple la position d'une pièce par rapport à sa position d'origine, l'existence de sa dame, si un "roque" a eu lieu etc. D'autre part, il est évident que plus la fonction d'évaluation sera complexe, plus le temps de calcul sera important, et moins il sera possible d'étudier de coups en un laps de temps donné. Les programmes classiques recherchent dans l'arbre le meilleur coup possible en fonction de la réplique de l'adversaire et ce jusqu'à une certaine profondeur de l'arbre. Cette procédure est connue sous le nom de **MINIMAX** [57]. à chaque niveau de l'arbre correspond le joueur qui a le trait (aux échecs, une fois sur deux ce sont les blancs). à une situation donnée correspond un ensemble de coups légaux, à chaque coup est associé un nœud avec sa valuation (dépendant de la fonction choisie). La procédure cherche dans l'arbre la branche la plus valuée associant à l'adversaire une valuation négative. En pratique cet algorithme est couteux en temps, le nombre de solutions augmentant exponentiellement avec la profondeur de l'arbre. C'est pourquoi, on utilise la technique de **Alpha–Béta** [57, 25] fondée sur le fait que certaines branches conduisent à des situations pires ou identiques à celles déjà étudiées. à titre d'exemple nous proposons l'étude du **tic-tac-toe** version simplifiée du **morpion** jeu bien connu des lycéens.

Exemple 1.3.1

Le tic-tac-toe se joue à deux joueurs sur un damier 3x3. Le but étant d'aligner 3 pions de sa couleur symbolisée par **X** *et 0. Au départ le trait est à* **X**. *La partie se termine soit lorsque les 9 coups ont été joués, soit s'il y a un gagnant. Grâce à la procédure d'alpha-béta on constate*

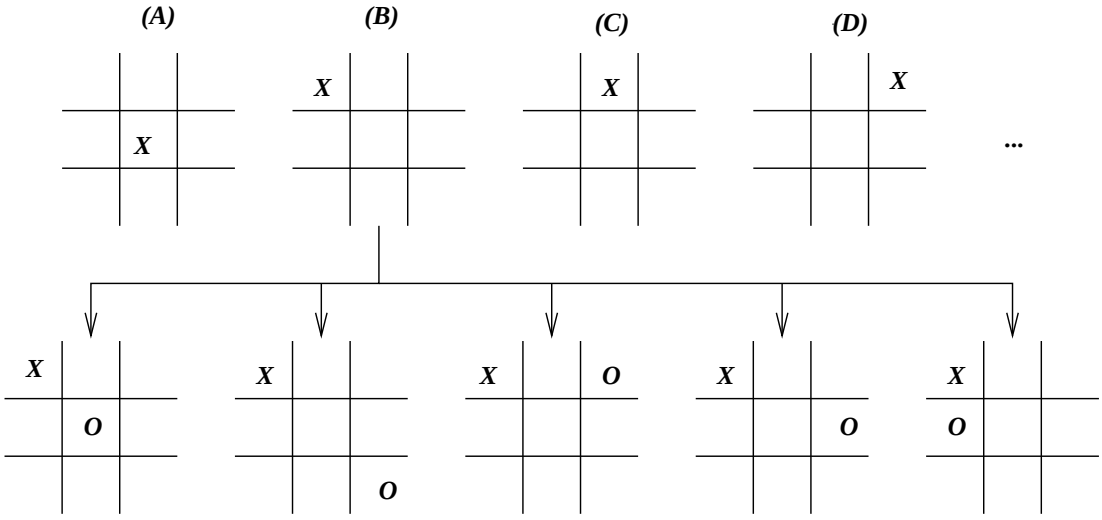


Figure 1.1 – Représentation partielle de l'espace des solutions

qu'au niveau 1 de la figure 1.1, seuls les cas **A**, **B** *et* **C** *sont distincts, les six autres cas peuvent être obtenus par rotation de* **B** *ou* **C**. *De* **A** *on aboutit à trois cas distincts, de* **B** *ou* **C** *cinq coups différents peuvent être joués par 0. Ce qui réduit l'arbre de 9 ! (soit 362880) nœuds en théorie à (2+5+5)*7 ! (soit 60480).* ◃

Exemple 1.3.2

Imaginons une autre situation où l'espace de recherche est valué par une certaine fonction de coût, comme dans la figure 1.2. On suppose, dans cet exemple, que la profondeur d'exploration maximale est fixée à trois. On évalue les situations aux feuilles, et on suppose que l'adversaire utilise la même fonction de coût et que, ce faisant, il cherche lui aussi à maximiser son gain, donc à minimiser le vôtre. Cette idée est dûe à Oscar Morgenstern et John Von Neumann (1945).

Si on évalue les gains possibles à partir des quatre coups jouables **A**, **B**, **C** *et* **D** *en affectant*

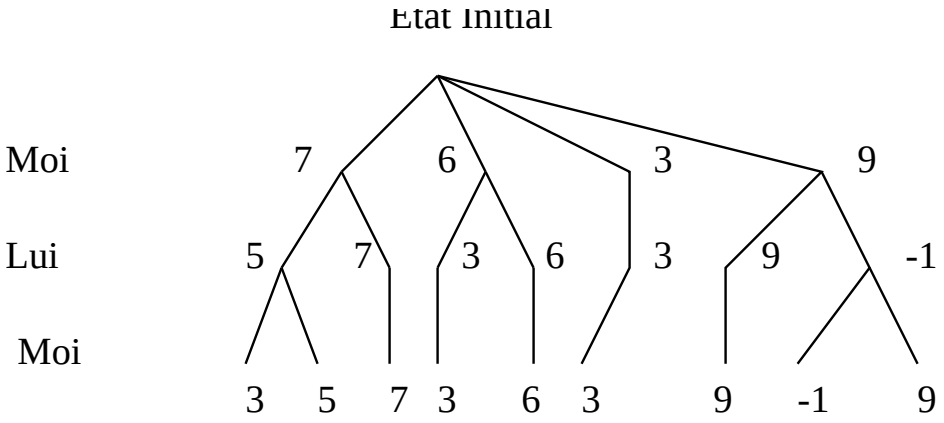


FIGURE 1.2 – Un arbre valué

un coefficient négatif pour le joueur Lui, *on trouve :*

A {3, 5, 7}

B {3, 6}

C {3}

D {9, 9, -1}

Il faut alors choisir la solution qui conduit au gain maximum tout en évitant une perte trop importante. De façon évidente on voit que le choix ***C*** *est moins bon que les choix* ***A*** *et* ***B*** *— le plus mauvais score est le même dans les trois cas, mais pas l'espérance de gain. La situation* ***D*** *est la plus "juteuse", mais aussi la plus risquée. Le programme se doit donc de connaître la force de son adversaire, et utiliser des méthodes de la théorie des jeux (se rapporter aux cours sur les probas et autres, ou le chapitre 6 de [71]), ou bien apprendre en étant confronté à différents adversaires (voire au même sur plusieurs parties), lire à ce propos "l'ordinateur joue au poker" dans [71].* ◃

Du fait d'une profondeur d'exploration limitée, l'ordinateur est sujet à *l'effet d'horizon*, que l'on peut traduire approximativement par : "un coup peut être optimal à la profondeur n, mais pas à la profondeur $n + k, k > 0$". De plus ces programmes jouent au "coup par coup", c'est-à-dire qu'il recalcule des solutions, non seulement entre deux parties, mais aussi entre deux "traits". En fait ils utilisent ce que l'on appelle la "force brute", c'est-à-dire leur puissance de calcul, plutôt qu'une quelconque aptitude à raisonner. Ce type de programmes arrive à battre entre 80% et 90% des joueurs (cf [24] numéro 35).

Sur des machines plus puissantes, outre les fonctionnalités précédentes, les programmes peuvent élaguer l'arbre d'évaluation en élaborant des plans, comme dans le cas des échecs essayer d'attaquer une pièce non défendue ou encore réaliser une "fourchette". Dans ce cas, seules les branches pertinentes pour l'aboutissement du plan sont évaluées. Ce type de programme s'apparente alors aux SE tels que ROBIN de J. Pitrat (1977) ou PARADISE de D. Wilkins (1979).

1.3.2 Ordinateur et jeux

Aux echecs (cf [24] numéro 17, pp 23–25), le programme/ordinateur qui fait référence en la matière est "Deep Blue" produit par IBM, si vous souhaitez en savoir plus sur le match qui l'a opposé à Kasparov, le mieux est d'aller visiter le site web `http://www.chess.ibm.com/meet/html/d.3.2.html`.

Aux dames, le nombre de coups légaux est plutôt faible, du fait des blocages et des prises obligatoires. Le programme de Dames écrit par A. Samuel (1959) a battu de grands joueurs, en tenant compte de certains paramètres tels que le contrôle d'une position, de plus il mémorise des situations déjà analysées. Il a accès (rapide) à une bibliothèque de 180 000 parties et la possibilité de modifier

dynamiquement sa fonction d'évaluation. Environ, 1 an après E. Jensen et T. Truscott ont écrit un programme utilisant des techniques similaires, qui est toujours considéré comme l'un des meilleurs joueurs de dames américain (cf [24] numéro 5, pp 32–39).

Le programme de backgammon BKG 9.8 (cf [24] numéro 8, pp 12–16), écrit par H. Berliner de l'Université de Pittsburg a vaincu le champion du monde en titre en 1979, le programme de G. Tesauro [91] basé sur un réseau de neurones et un apprentissage par renforcement est un autre exemple de programmes ayant montré des performances analogues.

Les jeux de cartes posent un problème différent, la représentation par arbre n'est plus la solution idéale, puisqu'en général on ne connait pas la (ou les) main(s) adverse(s). Il serait stupide d'étudier toutes les possibilités. On est alors amené à utiliser des connaissances sur le jeu lui-même. Il existe des SE pour le bridge (JOSEPHINE écrit par B. Faller du LRI d'Orsay), la phase des enchères étant codifiée, et au moins un programme pour le poker (écrit par Waterman) qui utilise une base de connaissances d'une vingtaine de règles, lui indiquant s'il doit suivre, relancer ou abandonner pendant les enchères. Pour cela, il tient compte de divers paramètres comme la valeur du pot, la valeur de la main, l'inclinaison qu'a l'adversaire à bluffer. Ce dernier paramètre ayant été évalué expérimentalement grâce aux parties antérieures. Ce programme est de plus capable de bluffer lui-même, sans aucun signe apparent de nervosité ...

1.4 Langage Naturel

Le traitement des langues naturelles touche plusieurs domaines tels que : la traduction automatique, l'analyse et la compréhension de textes, l'interrogation de bases de données, l'enseignement, les interfaces homme/machine. Pour en savoir plus, le lecteur est renvoyé aux publications suivantes [76, 25, 33, 82, 68].

Dès le début de l'informatique, l'homme a été confronté au problème de la communication avec les machines, la puissance de l'ordinateur étant contrebalancée par l'obligation d'utiliser un codage fastidieux — manipulation de clefs sur le frontal de l'ordinateur, ou l'écriture binaire [8]. De ces contraintes sont nés les langages de programmation et les dispositifs de traduction en binaire (seule information réellement intelligible pour un ordinateur) ; c'est-à-dire compilateur et/ou interpréteur. Reste qu'un langage de programmation, aussi évolué soit-il, n'est pas une langue naturelle. Les linguistes ont établi depuis fort longtemps que les langues humaines fonctionnent selon trois modes : *syntaxique*, *sémantique* et *pragmatique*. Le mode syntaxique correspond à l'adéquation entre une phrase d'un langage donné et la grammaire du dit langage. De cette étape, on extrait la structure de la phrase. Le mode sémantique permet de traduire cette structure en une formule supposée exprimer le sens du texte initial. Le mode pragmatique cherche à modifier le résultat de l'analyse sémantique en prenant en considération des éléments extérieurs au texte analysé. L'un des exemples classiques, extrait du chapitre 6 dans [25], « Pouvez-vous me dire l'heure qu'il est ? », est une question qui appelle une réponse par oui ou par non. Les conventions sociales, qui déconseillent les formes trop impératives, conduisent la personne à qui s'adresse la question, à interpréter la question comme une forme polie pour connaître l'heure. La pragmatique détermine l'ensemble des rapports possibles entre objets ou concepts intervenant dans le monde réel : une table peut supporter une tasse, l'inverse ne se produisant que très rarement.

Si les deux premiers modes sont effectivement implémenter, très peu de logiciels intègrent la troisième étape.

N. Chomsky a introduit la notion de grammaire de constituants ou grammaire générative, dans laquelle on a des règles de réécriture. Ainsi une version (outrageusement) simplifiée de la langue française pourrait être :

```
Texte  --> Phrase, Texte
Texte  --> Phrase; Texte
Texte  --> Phrase. Texte
Texte  --> Phrase.
```

[8] utilisant uniquement des 0 et des 1

```
Phrase --> GN GV
GN     --> Art Nm
GN     --> Art Nm Adj
GV     --> V GN
```

Où `Art` permet l'accès aux articles *le*, *la*, ... ; `Nm` permet d'accéder aux noms ...

Lorsque l'ordinateur travaille en mode syntaxique, il vérifie que le texte qu'il analyse possède bien l'une des structures que la grammaire lui impose.

Chomsky a montré que la meilleure représentation correspondait à un arbre mettant en évidence à la fois la décomposition de la phrase en constituants et l'appartenance de ces constituants à des catégories déterminées. Bien entendu, ce modèle a été raffiné pour prendre en compte la notion de verbes transitifs, intransitifs, indiquant un mouvement, de noms correspondants à des objets animés ou non. Cependant, si cette technique permet d'éviter des constructions comme « Le président marche la table », elle n'est pas suffisante pour déterminer si dans la phrase « les menaces de licenciement des directeurs », le mot « directeurs »est associé à « menaces »ou à « licenciement ». Ces constatations ont conduit Chomsky (et d'autres) à la notion de niveaux dans les langues naturelles, qui sont utilisés dans les grammaires transformationnelles. Outre leur complexité, peu compatible avec un traitement automatique, des phrases comme :
« Léon permet à son amie de venir »
et
« Léon promet à son amie de venir »
ont une stucture identique, alors que dans un cas c'est l'amie de Léon qui bouge et dans l'autre c'est Léon (lui-même) qui se déplace. La dissociation de la syntaxe et de la sémantique, a depuis été abandonnée. On associe à un mot du dictionnaire, différents *attributs* sémantiques (ou *traits* sémantiques), permettant d'en connaître le sens comme dans l'exemple décrit dans la figure 1.3.

Les traits sont nécessaires à la compréhension d'un texte et pour la traduction (semie) automatique.

1.4.1 Compréhension ou bluff ?

En 1966, J. Weizenbaum a écrit un logiciel (ELIZA), simulant l'activité d'un psychiatre[9] utilisant une thérapie « non directive ». Le programme donnait l'impression de poser les bonnes questions au malade, il s'agissait en fait d'une analyse syntaxique sur la dernière phrase du patient, et dont les mots clefs (nouveaux) étaient utilisés dans une forme interrogative. Si aucun mot nouveau n'etait utilisé, quelques phrases « bateaux »telles que « pourriez-vous être plus clair ? », « precisez votre pensée »etaient utilisées. La supercherie fut découverte par l'intermédiaire d'un autre programme (PARANOIA) fonctionnant sur le même mode mais qui simulait un patient. Le programme PARANOIA a été écrit par K. Colby de l'université de Stanford en 1970. Voici un court extrait du dialogue (adapté en français) :

```
...
P: Je ne comprends pas vos motivations
E: Pourquoi ne comprenez-vous pas mes motivations ?
P: Dois-je vraiment les comprendre ?
E: Pourquoi dites vous cela maintenant ?
P: Parce que c'est comme ça
E: Supposez que ce ne soit pas comme ça
P: Et alors ?
E: Je ne suis pas sûr de vous comprendre complètement
...
```

Il n'y a dans ces sytèmes aucune recherche sémantique, en fait ils ne comprennent rien.

Actuellement, il existe des systèmes capables de répondre à des questions exprimées dans une langue naturelle, pourvu que ce soit sur un domaine extrèmement restreint. Le problème est loin

[9] On trouvera sous Emacs (21.2) une version de ce programme dans le menu `Help`

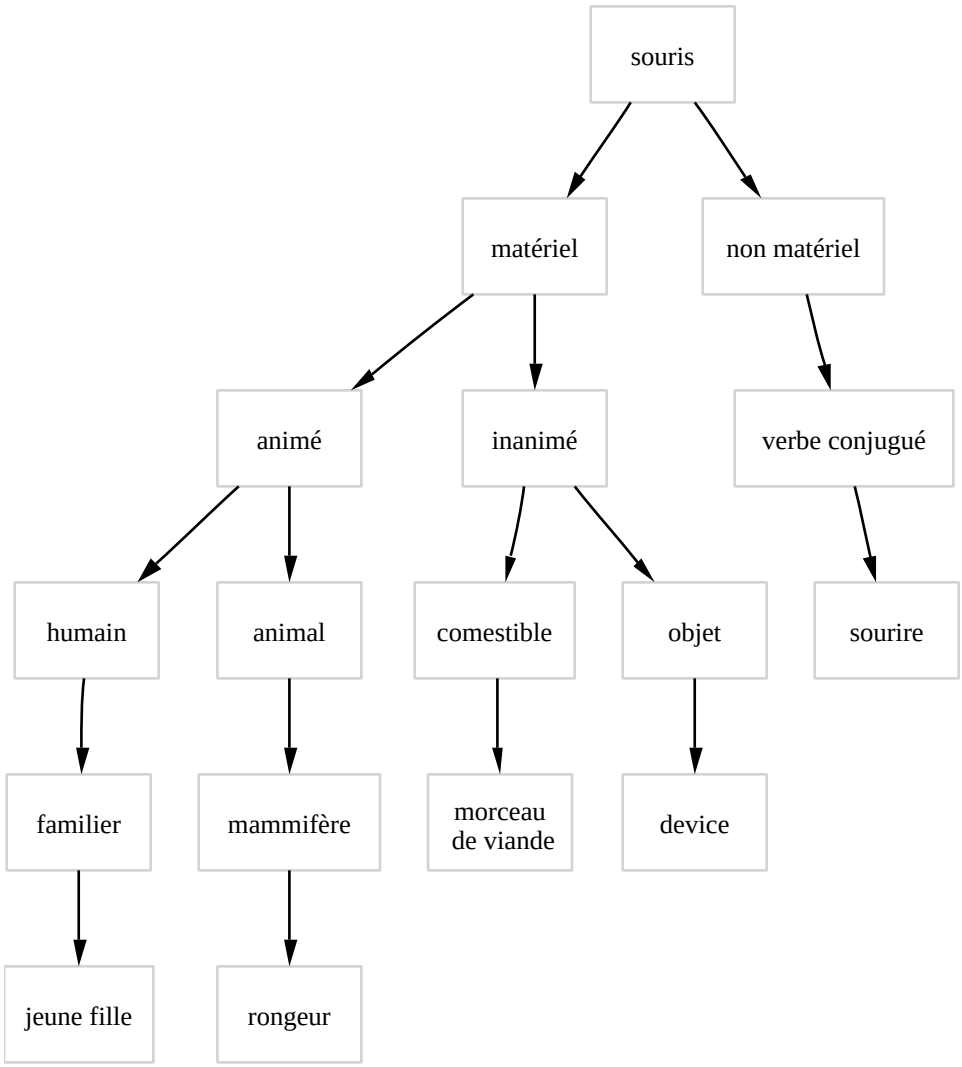


FIGURE 1.3 – Mot et traits sémantiques associés

d'être simple, de nombreux mots sont sujets à de multiples interprétations. Pour compliquer le tout il n'existe pas de méthodes formelles permettant de s'assurer que quelqu'un à totalement compris quelque chose — comment un enseignant peut-il être certain que son enseignement a été *totalement* compris. Des systèmes de compréhension de langues naturelles sont dores et déjà commercialisés, citons par exemple l'interrogation de bases de données, la recherche documentaire (Spirit de la société Systex) ou la commande d'automatismes.

1.4.2 Traduction automatique

Force est de constater qu'il s'agit d'une nécessité, on peut citer par exemple : bulletin météorologique au Canada (bilinguisme oblige). En juin 1985, on comptait 1800 traducteurs au service de la CEE, la moitié du budget de fonctionnement était consacré à la traduction. Les pays comme le Japon ou TaïWan sont de gros exportateurs de produits manufacturés, les notices techniques et mode d'emploi pour les pays importateurs sont une obligation.

La difficulté énorme de cette tâche est illustrée par la fable suivante. Un « savant américain »ayant construit un traducteur universel prit contact avec un sénateur dans l'espoir d'obtenir quelques subsides. Le sénateur demanda à voir cette merveille. Pour tester le système il proposa de faire traduire d'anglais en chinois la devise suivante : « out of sight, out of mind »[10] (loin des yeux, loin du cœur) ; après quelques minutes la machine imprima une feuille remplie d'idéogrammes. Le sénateur emerveillé, ne lisant pas le chinois, demanda que la machine fasse la traduction inverse. Un peu plus tard il obtint la réponse suivante : « invisible idiot »[11]

L'idée de faire effectuer la traduction par une machine fut émise par Weaver et Booth (1949) à l'université de Washington, dès la sortie des premiers ordinateurs. Les machines n'étaient alors considérées que comme de gros dictionnaires (même pas portatifs), et la traduction étant réduite à une opération de mot-à-mot[12]. En 1964, le gouvernement américain demanda un rapport sur la traduction automatique. Ce rapport (connu sous le nom de ALPAC) établissait que la traduction automatique n'avait aucun avenir que ce soit à court ou à moyen terme. Cela stoppa net la recherche dans la plupart des pays (fautes de subsides).

Certains résultats ayant été obtenus depuis, la traduction (semie) automatique est de nouveau un sujet de recherche dynamique [33]. La traduction doit, pour être fiable, travailler sur les trois axes : syntaxique, sémantique et pragmatique.

Exemple 1.4.1

Traduction anglais/français :

« I love you »l'analyse syntaxique (anglaise) indique que la phrase est du type Sujet + Verbe + Complément. à partir de la syntaxe anglaise, de la phrase et d'un dictionnaire on obtient les correspondances suivantes :

I	=	*je / moi*
love	=	*amour / aimer*
you	=	*tu / te / toi / vous*

La syntaxe française permet d'ordonner les constituants différemment, les contraintes liées à la grammaire française (conjugaison) permettent de construire les deux alternatives : « Je vous aime »et « Je te aime ». La sémantique et la pragmatique indique que le verbe aimer induit un rapport familier entre le sujet et le complément plutôt qu'un rapport de politesse, le pronom « te »sera préféré, enfin les contraintes grammaticales obligeront l'élision du pronom donnant finalement la forme « Je t'aime ». ◁

Le système Sytran a été l'un des premiers systèmes de traduction opérationnel (année 60), la plus grande partie de l'information nécessaire est prise dans les dictionnaires bilingues. En plus des lexiques de mots et d'expression courantes (avec attributs sémantiques et pragmatiques [13], on ajoute des lexiques spécialisés permettant de cibler exactement le vocabulaire spécifique au document. L'anglais technique est très différent de l'anglais shakespearien.

[10] dans un dictionnaire anglais–français : sight se traduit par vue et mind par esprit.

[11] dont la traduction en français est immédiate ...

[12] « the spirit is willing but the flesh is weak »(l'esprit est consentant mais la chair est faible) devenant « l'alcool est prêt mais la viande est avariée »

[13] en France on « mange la soupe »tandis qu'au Japon on « boit la soupe ».

Actuellement, la traduction n'est pas automatique mais assistée par ordinateur (T.A.O.) on y distingue plusieurs phases :

1. saisie du texte source,
2. contrôle syntaxique et orthographique, un opérateur devant compléter ou corriger les mots non reconnus par le système,
3. traduction en fonction des différents lexiques chargés,
4. le texte obtenu apparaît à l'écran, les mots non reconnus sont placés tels quels dans le texte, mais à leur place dans la langue cible, par exemple si « you »n'est pas reconnu, la phrase « I love you »sera transformée en « Je you aime »,
5. l'opérateur intervient pour donner la touche finale à la traduction.

Lorsqu'un mot nouveau est introduit dans le lexique, toutes les occurences sont modifiées dans le texte. Les performances des systèmes T.A.O. offrent un gain de l'ordre de 80% en temps de traduction.

Il existe deux types de systèmes T.A.O. :

direct : chaque langue nécessite un dictionnaire vers toutes les autres langues. Offrant les avantages d'une traduction directe et plus efficace, et l'inconvénient, pour un système traitant N langues, auquel on ajoute une nouvelle langue, d'ajouter 2N dictionnaires.

indirect : utilisation d'une langue « pivot ». Offrant des facilités d'extension (2 dictionnaires pour une nouvelle langue), et les inconvénients d'une double traduction (passage obligé vers le langage pivot) auquel s'ajoute la difficulté de trouver la langue pivot. C'est pourquoi cette approche n'est utilisée qu'au sein de langues appartenant à la même famille (exemple indo-européenne).

1.4.3 Laboratoires français

Différents laboratoires traitent cette question épineuse, les plus importants (liste non ordonnée et non exhaustive) sont :

- GETA, Université de Grenoble
- GIA, Université de Marseille
- Université de Paris VI
- LADL, Université de Paris VII
- Cap Sogeti Innovation
- CNET Lannion

1.5 Reconnaissance de formes

La reconnaissance de formes est l'un des domaines les plus importants de l'IA, la notion de formes est à prendre au sens large (formes visuelles, sonores ou tactiles), son domaine d'application : la robotique, les interfaces homme/machine.

1.5.1 Vision

Nous allons aborder trois activités : l'analyse d'objets (qui est de fait la méthodologie générale), la reconnaissance de caractères et enfin l'analyse de scènes en 3 dimensions (3D). Ces deux derniers domaines étant des applications directes de la première activité.

Analyse d'objets

Détermination de l'objet, caractérisation, comparaison entre l'image connue et l'image vue au niveau du pixel[14]. On peut considérer dans ce processus trois phases distinctes :

- prétraitement : simplification de l'image visualisée,
- extraction des primitives : caractéristiques principales de l'objet,
- classification.

L'opération de prétraitement englobe l'extraction des contours i.e. la détection de courbe continue, l'analyse de continuité, le seuillage qui correspond à choisir un niveau de gris, tout pixel plus foncé sera codé par **1**, tout pixel plus clair étant codé par **0**.

L'extraction des primitives consiste à coder l'image au moyen de descripteurs pertinents ; pertinence dépendant du type de traitement ultérieur. Par exemple on peut s'intéresser à la surface, au périmètre, au nombre de trous, à la dimension du rectangle circonscrit, ... On rajoute aussi des descripteurs de position et d'orientation comme le centre de gravité, les angles, ...

La classification est en fait le processus de reconnaissance proprement dit, il s'agit de comparer l'image visualisée avec la connaissance du système. En général on utilise l'une des deux méthodes suivantes :

- méthode du plus proche voisin, c'est aussi la plus simple et la plus longue. Il faut comparer les objets un à un ; ce qui est rendu d'autant plus difficile que les contraintes initiales seront lâches (objet arrivant dans n'importe quel sens, superposition possible, ...).
- méthode de décision par arbre binaire, cette technique demande de choisir un nombre optimal de critères discriminants (descripteurs pertinents), c'est la technique la plus couramment utilisée dans le cadre de la reconnaissance de caractères.

Les recherches menées en synthèse d'images sont faites en étroite collaboration avec l'analyse d'un objet digitalisé (problème de stockage, représentation de l'image, ...).

Reconnaissance de caractères

Actuellement plusieurs types de lecteurs optiques sont disponibles sur le marché (lecteur codes barre, OCR[15], ...) depuis ceux spécialisés dans la reconnaissance d'une ou plusieurs polices de caractères jusqu'aux lecteurs aptes à apprendre n'importe quoi.

L'intérêt évident étant le gain de temps (comparons simplement ma vitesse de frappe pour retranscrire ce texte dactylographié et un OCR du marché fourni avec un scanner à 135€). Auquel s'ajoute le fait que les supports de masse (magnétique, optique, ...) ont des durées de vie nettement supérieur à celle du papier, les recherches documentaires, le traitement du courrier dans les centres postaux, ...

Le premier lecteur optique avec reconnaissance de caractères date de 1955, les recherches sur le sujet furent abandonnées par les grosses sociétés (IBM, Control DATA) lorsqu'au début des années (19)60 parurent les premiers résultats sur la synthèse vocale (et le mythe de la « paperless society »). Seuls quelques constructeurs spécialisés continuèrent à travailler sur le sujet (et heureusement vu l'état de l'art en synthèse vocale au seuil du troisième millénaire). En 1974, R. Kurzweil a l'idée de concevoir un lecteur optique permettant aux non-voyants de lire des textes classiques. La machine associe lecture optique et synthèse vocale pour restituer le texte original, en 1985 la KDEM 4000 est l'un des systèmes les plus performants.

La reconnaissance nécessite une squelettisation du caractère. Une fois simplifié, la représentation peut se faire en utilisant le code de Freeman basé sur le fait qu'un pixel possède au plus 8 voisins (numéroté de 0 à 7) permettant ainsi un codage sur un octet.

[14] cellule élémentaire d'une image digitalisée ; abréviation de « picture element ».
[15] « Optical Character Recognition ».

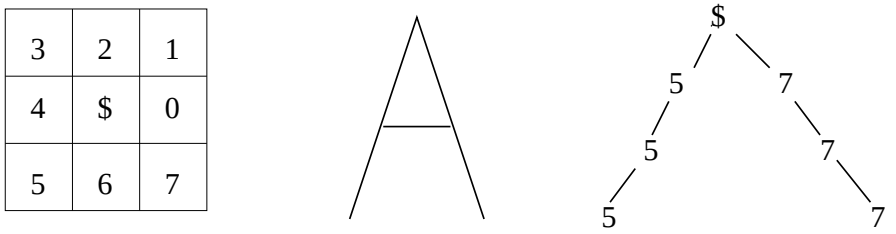
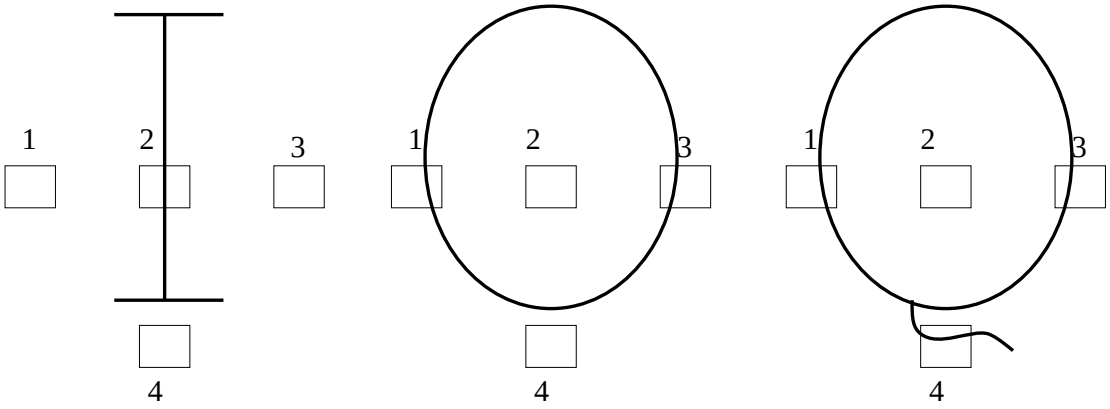


FIGURE 1.4 – Exemple de codage d'un A

On peut aussi utiliser des descripteurs tels que verticale, horizontale, courbe, ... qui sont indépendants de la police de caractères utilisée, la reconnaissance pouvant alors s'effectuer à partir de masques partiels tels que ceux décrits dans la figure 1.5 ou par des heuristiques, comme la distribution aléatoire de droites dans le plan avec étude statistique du nombre d'intersection avec le caractère (cf figure 1.6). Dans cette figure on remarque que la lettre **B** a plus d'intersection avec les droites que la lettre **A**.



Masques	1	2	3	4
I	0	1	0	0
O	1	0	1	0
Q	1	0	1	1

FIGURE 1.5 – Masques partiels Lettres I, O, Q

Pour en savoir plus
Le lecteur désireux d'étendre ces connaissances sur le domaine est renvoyé aux parutions suivantes :
- Le chapitre 3 [25, pp. 77-98], « La vision des robots ».
- Le chapitre 4 [25, pp. 99-118], « La reconnaissance de l'écriture ».

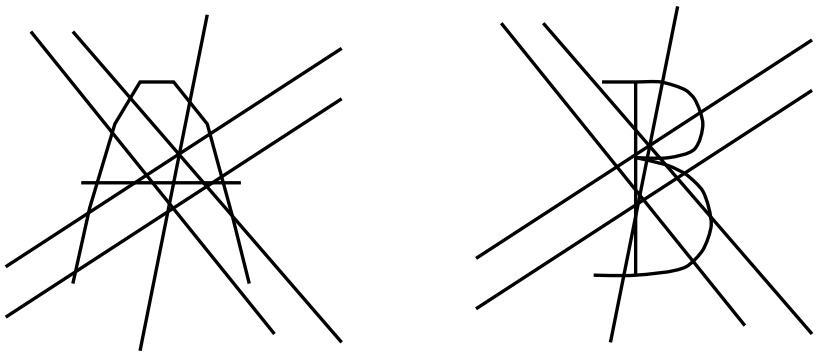


FIGURE 1.6 – Droites aléatoires

Analyse de scènes

La difficulté dans l'analyse de scènes 3D réside dans le fait que toute l'information n'est pas accessible directement problème de faces cachées, que l'on rencontre aussi en synthèse d'images (cf figure 1.7).

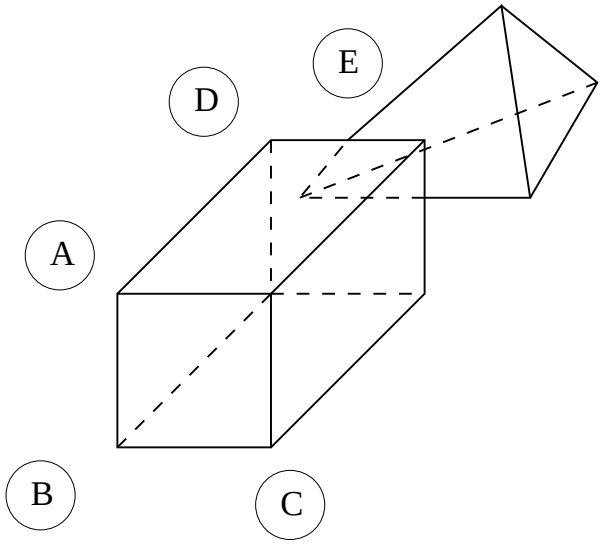


FIGURE 1.7 – Une scène 3D

On cherche dans ce cas à détecter les points caractéristiques, par exemple, dans le cas de volumes simples (cf figure 1.7), les *angles* ou les *jonctions* de droites entre les différents contours (cf figure 1.8. À partir de ces informations l'ordinateur (ou le robot) doit décomposer la scène en éléments simples.

1.5.2 Parole

L'analyse de la reconnaissance vocale a donné lieu à de nombreux travaux depuis plus de 25 ans, les applications ne se restreignent pas uniquement aux ordinateurs. Déjà certains systèmes (simples voire simplistes) équipent des véhicules de haute gamme, on peut imaginer que d'ici peu les voitures réagiront à des ordres, que le téléphone pourrait composer automatiquement des numéros — dans ce cas on peut imaginer que la personne utilisant un composeur vocal associe un mot clef à un numéro donné, puis n'a plus qu'a citer ce mot, pour que le numéro soit appelé (la société Thomson CSF a

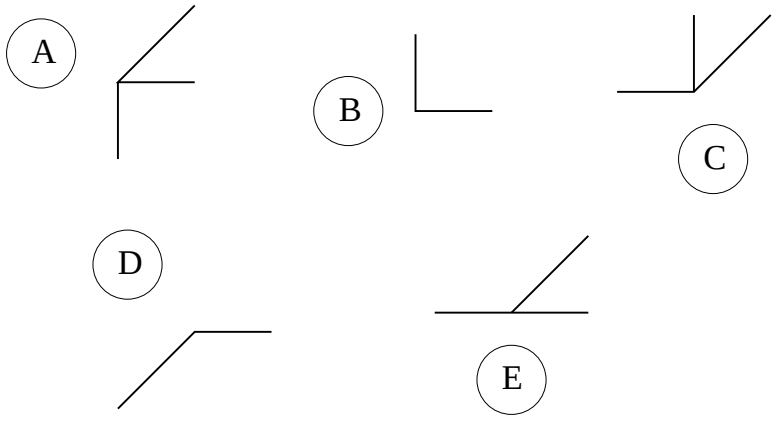


FIGURE 1.8 – Points caractéristiques de la figure 1.7

d'ailleurs développé et fabriqué de tels composants). Le KATALAVOX de M. Kempf (19)85 utilisé pour commander un appareil chirurgical à l'aide des quatre commandes directionnelles « up », « down », « left »et « right ».

La reconnaissance vocale repose sur une relation bi-univoque entre un ensemble de sons appris (entendus, mémorisés) et les sons perçus ultérieurement. L'élément de base est le phonème (au lieu du pixel) ; le français comporte quelques 33 phonèmes distincts. En réalité on travaille sur les diphonèmes ($33 \times 33 \approx 1000$), et de plus ce diphonème est affecté par le phonème précédent et le phonème suivant soit au total quelques 10^6 combinaisons.

Quelques laboratoires Le CERFIA (Toulouse), le CNET (Lannion), le CRIN (Nancy) et l'ENSERG (Grenoble) sont les plus connus en France.

Pour en savoir plus
- Le chapitre 5 de [25, pp. 119-148] « La reconnaissance de la parole ».
- Le chapitre 6 de [25, pp. 149-176] « Des machines qui comprennent notre langue ».

1.6 Matériel pour l'IA

1.6.1 Les ordinateurs

Il existe des langages spécifiques à l'IA [42, 40, 67, 21] de la même manière qu'il existe des langages pour le calcul scientifique (FORTRAN [61]) ou pour la gestion (COBOL [53]). Certaines opérations de base de ces langages peuvent être cablées, c'est-à-dire que des circuits logiques spécialisés sont utilisés pour réalisé ces opérations, permettant une plus grande efficacité de traitement. Dans les machines LISP (chapitre 7) (principal constructeur SYMBOLICS) les fonctions d'accès aux éléments d'une liste sont cablées ; pour les ordinateurs japonais de 5^{eme} génération, qui sont des machines PROLOG (chapitre 8) se sont les mécanismes de substitution qui sont privilégiés.

Le tableau de la figure 1.9 est un résumé d'une étude de l'Université de Stanford, 1985 et publié dans la revue MICRO-SYSTEMES de février 1987

1.6.2 Automatique, Robotique, Productique

Ce ne sont pas, à proprement parler, des matériels de l'IA mais ils en sont l'une de ces meilleures raisons d'être.

Compagnie	Machines	Prix($)	Sites	Type	Applications
TEKNOWLEDGE	Xerox Symbolics DEC-VAX	45 000 80 000	70	orienté règles	diagnostic
INTELLICORP	Xerox LMI Symbolics HP 9000/300	60 000	110	règles + frames	diagnostic simulation conduite de simulation
INFERENCE	Symbolics LMI DEC TI Explorer	85 000	70	règles	divers
CGF	Symb. LMI Perg, VAX TI Explorer	50 000	25	orienté objets	CAD/CAM CAE
SOFTWARE	Apollo Sun Tektronix VAX IBM/PC-XT	7 000 25 000 4 000	40	orienté règles + frames outil statistique	planification des missions militaires diagnostic
SMART SYSTEM TECHNOLOGY	Symbolics VAX Xerox	6 000	-	règles	diagnostic

FIGURE 1.9 – Tableau des machines d'IA et des stations de travail (1985)

Le robot a fait son apparition (en tant que tel) dans les ateliers des années (19)50. Il s'agissait de machine outils commandées par des bandes de papier perforées, le mot robot trouve son origine dans une (mauvaise) traduction d'une œuvre de K. Capeck (R.U.R.[16] 1920) et est tiré d'un mot tchèque signifiant travailleur, son titre de noblesse lui fut donné par Isaac Asimov [6][17] et ses célèbres lois de la robotique[18] :

1. Un robot ne peut porter atteinte à un être humain ni, restant passif, laisser cet être humain exposé au danger.

2. Un robot doit obéir aux ordres donnés par les êtres humains, sauf si de tels ordres sont en contradiction avec la première loi.

3. Un robot doit protéger son existence dans la mesure ou cette protection n'est pas en contradiction avec la première ou la deuxième loi.

Le premier robot à usage industriel apparaît en 1962, et en France aux alentours des années 1970 (usine Renault). Avant de voir ce dont ils sont maintenant capable, faisons un saut dans le passé.

Un peu d'histoire

La plus ancienne trace littéraire se trouve dans le livre XVIII de l'Iliade, dans lequel on peut lire " Alors ...il sortit en clopinant, appuyé sur un baton épais et soutenu par deux jeunes filles. Ces dernières étaient faites en or à l'exacte ressemblance de filles vivantes ; elles étaient douées de raison, elles pouvaient parler et faire usage de leurs muscles, filer et accomplir les besognes de leur état ... ".

Avant les robots étaient les automates. C'est un barbier grec du III^{e} siecle avant J.C. qui construisit le premier automate. Inventeur de l'orgue hydrolique, il est le premier à réaliser des oiseaux

[16] Rossum Universal Robots (Robots universels de Rossum). Comme Frankestein, Rossum avait découvert le secret pour fabriquer des hommes articulés

[17] recueil de nouvelles écrites entre 1940 et 1948

[18] apparues dans "cycle fermé" deuxième nouvelle du recueil [6]

siffleurs et autres figurines se déplaçant sous l'action de l'eau. Plus tard Philon d'Alexandrie mis en pratique les découvertes d'Archimède [19], et construisit des automates pondéraux, hydroliques et pneumatiques. Ceux-ci sont mûs par écoulement de fluide et chute de poids. Ils n'effectuent qu'un nombre limité d'actions, mais bougent sans que les mécanismes responsables soient visibles. C'est au I^{er} siècle avant notre ère qu'apparaissent les automates avec rupture de séquence — une chevillette délogée par la chute d'un poids provoquait le changement de comportement — qui étaient plus "vivants". Pendant le moyen-âge, les automates évoluent avec l'apparition de l'horlogerie. L'époque des automates modernes commence avec Descartes (1630). La Pascaline (1645) fait des additions et des soustractions, la machine de Leibnitz (1646-1716) multiplie. Ces deux engins introduisent l'emploi des registres intermédiaires et font des reports automatiques de retenues.
En 1738, Vaucanson réalise un joueur de flûte, androïde assis exécutant rigoureusement les mêmes actions qu'un véritable flûtiste. L'air actionnant l'instrument sort de la bouche de l'automate, les lèvres modulent l'air et les doigts obturent librement les trous ad-hoc. En 1860, Farcot invente le servo-moteur.

Et maintenant . . .

La première définition officielle de robot date de 1978, elle est dûe aux japonais "un robot est une machine polyvalente, capable d'agir sur son environnement, et adptable à un environnement changeant".

On distingue trois types de machines :

1. mécanique : levier, treuil, grue qui reçoivent et restituent l'énergie mécanique.
2. énergétique : transforme une forme d'énergie en une autre, typiquement la machine à vapeur.
3. programmable : exemple type l'ordinateur.

Les deux premières catégories sont des prolongements de notre système musculaire, alors que la troisième peut être vue comme une extension de notre système nerveux.

Toutes consomment de l'information, mais l'automate est programmé de façon séquentielle alors que le robot est capable d'accomplir plusieurs fonctions et de les réaliser de manière autonome.

La cybernétique est la science qui s'occupe des robots, elle tente de reproduire les fonctions humaines comme l'intelligence, l'apprentissage et l'adaptation. Les travaux de Wiener [73] ont donné naissance à des animaux cybernétiques vérifiant le schéma suivant :

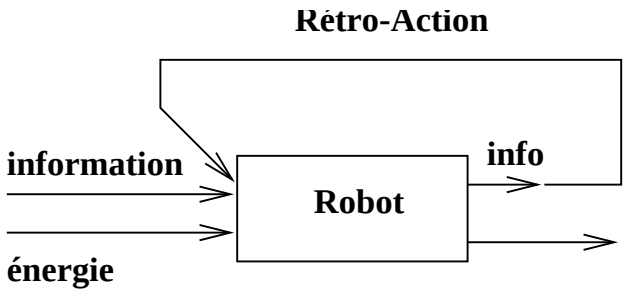


FIGURE 1.10 – Rétro-Action

On considère qu'il existe deux types de rétro-action : l'une *positive* dont l'objectif est d'atteindre un but ; la seconde *négative* qui a pour but le retour à un état d'équilibre, il s'agit d'une régulation de système type homéostat.

On parle aussi de machines comportementales, la plupart des robots ne sont pas anthropomorphes (exple la marche sur 2 pattes et plus difficile que le déplacement sur roues). Souvent un robot se réduit à un bras manipulateur + des doigts/pinces pour la préhension. Parfois il est doté d'un système de

[19] Eurekâ !

vision, mais si celui-ci est placé au-dessus du bras, on est confronté à des problèmes de parallaxe, de qualité de résolution, de transformation de coordonnées qui n'existeraient pas s'il était situé au niveau de la pince.

Les robots sont classés en 6 catégories :

1. Robots manipulateurs (RM), à commandes manuelles (aucune autonomie),
2. RM à séquences fixes
3. RM à séquences variables, compatibles avec plusieurs hypothèses de travail,
4. Robots programmables par apprentissage, dotés d'une mémoire contenant plusiseurs séquences,
5. Robots à commandes numériques, dépendance vis-à-vis de l'informatique,
6. Robots avec capteurs sensoriels.

Seuls ceux appartenant à la dernière catégorie sont considérés comme pouvant avoir un comportement intelligent [20], cette nouvelle génération (productique) peut communiquer avec l'homme. Ces robots peuvent se déplacer, sont capables de détecter leur défaillances et de les signaler (si ce n'est d'y remédier). La robotique fait partie du milieu industriel, il existe même au Japon une usine de robots construisants des robots (et ça marche ...).

En 1985, est parue l'estimation suivante :

Pays	Nombre de robots
France	2 300 / 2 700
USA	13 000
Japon	21 000 / 25 000
RFA	6 600
GB	2 600
Suède	2 400
Italie	2 600

La réalisation qui m'a le plus impressionnée est le robot pianiste du Pr. Kato, qui possède (le robot pas le Pr.) un système de vision lui permettant de lire une partition musicale (achetée en ville) en temps réel, tandis que les mains articulées jouent (bien) le morceau déchiffré. Chaque bras possède 7 degrés de liberté définissant les mouvements du poignet, du coude et de l'épaule. Chaque bras est contrôlé par un ordinateur indépendant et 16 micro-processeurs.

Pour en savoir plus
- Le chapitre 3 de [25] "La vision des robots", pages 77–98.
- Le chapitre 13 de [25] "Les robots mobiles autonomes", pages 335–364.

1.6.3 Réseaux de neurones, Connexionisme

Le but de ce matériel n'est pas la simulation du cerveau, en effet, les réseaux de neurones formels ne sont pas un modèle de notre structure cérébrale mais plutôt une métaphore. Le cerveau est constitué d'un réseau extrèmement dense de cellules nerveuses (neurones), de chaque cellule partent plusieurs dizaines de branches (dendrites), l'influx nerveux passe d'un neurone à l'autre par le biais de synapses. L'influx est de nature électrique et se transmet par un processus chimique. Le neurone collecte, au niveau des dendrites les signaux électriques et propage l'information à ses voisins si le seuil de sensibilisation est atteint.

Chaque élément constitutif d'un réseau de neurones formels réalise un traitement simle, dont l'intérêt réside dans l'émergence de propriétés globales à l'ensemble du système (le neurone formel est au

[20] On s'accorde en général à penser que l'intelligence n'existe que dans le cadre d'un (ou plusieurs) échange(s) avec l'extérieur.

réseau, ce que la fourmi est à la fourmilière). Chaque composant fonctionne plus ou moins indépendemment de ses voisins, la structure globale étant fortement parallèle avec un indice de connexions très élevé. De fait, l'objectif des réseaux de neurones artificiels est d'essayer de récupérer tout ou partie des fonctionnalités d'un cerveau telles que :

- architecture massivement parallèle,
- calcul et mémoire massivement distribués,
- capacité d'apprentissage,
- capacité de généralisation,
- capacité d'adaptation,
- forte tolérance aux pannes,
- faible consommation énergétique.

Les ordinateurs actuels (fin du 20ème siècle) outrepassent largement les aptitudes du cerveau dans le domaine du calcul et de la manipulation de symboles. Cependant les êtres humains, peuvent sans effort, résoudre des problèmes complexes (ou mal définis) dans le domaine de la perception (reconnaissance d'un visage dans une foule, reconnaissance d'un individu plusieurs (dizaine d') années plus tard, ... On peut considérer que la nature des problèmes résolus dépend fortement du type d'architecture des modèles computationnels utilisés. Les domaines où les architectures neuronales sont particulièrement efficaces sont :

- l'association,
- la classification,
- la discrimination,
- l'estimation.

L'histoire des réseaux de neurones formels remontent au milieu des années 1940. On doit à Mc Culloch et Pitts la première présentation des neurones formels, et à Hebb la première règle empirique pour la modification des poids synaptiques (et par conséquent l'apprentissage dans ce type de système). Le perceptron de Rosenblatt (1958) est le plus célèbre système utilisant une architecture voisine. En 1969 Minsky et Pappert ont montré mathématiquement les limitations de ce genre d'appareil ; ce qui stoppa les recherches jusqu'en 1982. À cette date, J. Hopfield, éminent physicien a remis au goût du jour cette approche en proposant une solution au problème du voyageur de commerce grâce à un modèle que depuis lors on appelle réseau de Hopfield. De plus en plus de chercheurs venant d'horizons très diverses s'intéressent à ce domaine (neurobiologie, mathématiques, informatique, électronique, biologie). Le lecteur est renvoyé à la partie IV et plus particulièrement au chapitre 14 page 154 pour une introduction plus précise au domaine.

Autres applications

Il existe d'autres applications intégrant des composants tirant partie de techniques développées en IA. Parmi celles-ci on trouve les appareils à commandes floues basés sur la logique floue (cf le chapitre 6 partie II), citons, entre autres l'appareil photos Nikon F90 [62] qui utilise un calculateur numérique, le Canon H800 camescope commercialisé en (19)90 qui utilise une mise au point basée sur 13 règles floues, certains aspirateurs (Mitsubishi, Samsung) réputés pour avoir une consommation énergétique inférieure d'environ 40%. L'industrie automobile utilise des composants flous (General Motors a équipé son modèle Saturn d'une transmission floue, Nissan a breveté des systèmes flous de freinage antidérapage, de transmission floue, d'injection de carburant). L'un des systèmes les plus complexes est un modèle réduit d'hélicoptère réalisé par M. Sugeno de l'Institut de Technologie de Tokyo, dans lequel le gouvernail de profondeur, l'aileron, la commande des gaz et le palonnier réagissent à 13 commandes vocales floues telles que "monter", "atterrir", "faire du sur-place"

(cette dernière action étant l'une des plus difficiles pour un pilote humain). Plus hypothétique, dans un article de la revue "PC Expert", numéro d'août 1998, on trouve une présentation prospective des machines à l'aube du 3éme millénaire. Ces machines intègreraient les interfaces virtuelles (3D, commande vocale, ...) réagiraient en intégrant des informations sur l'environnement (affichage de résultat sur le terminal visuel le plus proche de l'interlocuteur humain). Bref, utiliseraient des techniques qui sortent actuellement des laboratoires de recherche en Intelligence Artificielle.

Deuxième partie

Logiques et Raisonnement

L'objectif de cette partie est de donner envie aux lecteurs d'approfondir ce champ qui est souvent mal connu si ce n'est mal dominé [45, 72]. Nous étudierons, outre la logique propositionnelle (base des autres logiques) et la logique des prédicats, passage obligé avant d'aborder la programmation logique (PROLOG ou l'un de ses avatars), diverses logiques, que nous ne ferons qu'effleurer telles que la logique modale, la logique floue, la logique intuitionniste, ... Pour une étude plus détaillée, nous renvoyons le lecteur aux livres suivants qui, il faut l'admettre sont assez inégaux, tant par leur contenu que par leur lisibilité, je les ai classé par thèmes :

Livres traitant de logique propositionnelle
[94, 60, 81, 92, 45, 32, 80, 55, 5, 54, 78]

Livres traitant de logique des prédicats
[94, 60, 81, 92, 45, 32, 80, 55, 5, 54]

Livres traitant de logique floue
[12, 95, 38, 52]

Autres thèmes connexes
[50, 63, 101, 9, 43, 18, 69, 93, 45, 48, 46]

L'un des buts de l'IA est de concevoir des systèmes capables de reproduire le comportement humain dans ses raisonnements [58, 45]. De ce fait, l'étude du raisonnement avec pour objectif sa mécanisation est un thème majeur de la discipline, tant du point de vue de la recherche fondamentale qu'appliquée. Le raisonnement se faisant sur des connaissances, il nous faut, dans un premier temps, définir les concepts englobés et les problèmes liés à leurs représentation.

La connaissance intègre différentes formes de savoir : objets, règles heuristiques. La méta-connaissance (connaissance sur la connaissance) en est une des formes importantes car elle est liée à la façon d'utiliser la connaissance, aux stratégies du raisonnement et à l'acquisition de nouvelles connaissances (obtenues par raisonnement ou par apprentissage). D'une manière générale, les connaissances peuvent être permanantes ou temporaires, il peut aussi s'agir d'une hypothèse émise pendant un raisonnement et qui attend d'être (in)validée.

Représentation des connaissances

Pour que le raisonnement puisse s'effectuer, il est nécessaire de préciser l'*Univers du Discours*, ce qui signifie que tout ce qui n'est pas connu comme étant vrai et, ipso facto, considéré comme faux (hypothèse dite du monde clos, qui est mise en œuvre dans un langage comme PROLOG cf chapitre 8 de la partie III). Pour mécaniser le raisonnement il faut :

1. Une structure de données. Plusieurs approches sont possibles, nous survolerons les représentations logiques, réseaux sémantiques, règles de production et frames.
2. Une ou plusieurs procédures d'exploration.
3. Une représentation claire (être capable de faire correspondre à un élément de l'Univers du Discours, une représentation), puissante et suffisante.
4. Une simulation efficace en temps et en espace.

On pourra aussi se reporter à l'article de S. Pinson [75] sur la représentation des connaissances dans les systèmes experts, ou au livre de A. Bonnet [11] pour une vue plus générale.

Représentation logique
L'Univers du Discours est représenté par un ensemble de formules logiques, les règles sont le *modus ponens* qui peut s'exprimer par $(((p \supset q) \wedge p) \supset q)$, et le *modus tollens* qui s'exprime en logique par $(((p \supset q) \wedge \neg q) \supset \neg p)$.

Ainsi le langage PROLOG fournit un mode de représentation et des mécanismes de raisonnement totalement intégrés.

Réseau sémantique
Un réseau sémantique (voir le chapitre 11 de [11] ainsi que [65] page 256 et suivantes) est un graphe étiqueté où les nœuds sont les objets (concepts) et les arcs étiquetés les relations. De nombreux systèmes sont issus de cette approche, tel que KL-ONE dont la conception date de 1970. Ce langage est à la frontière entre réseaux sémantique et frames. Un réseau exprime une hiérarchie et offre une solution à la structuration des données (cf figure 1.11). Un nœud peut hériter par *subsumption*[21] des propriétés d'un autre nœud. Ce concept est présent dans les langages orientés objets (cf chapitre 11 de la partie III).

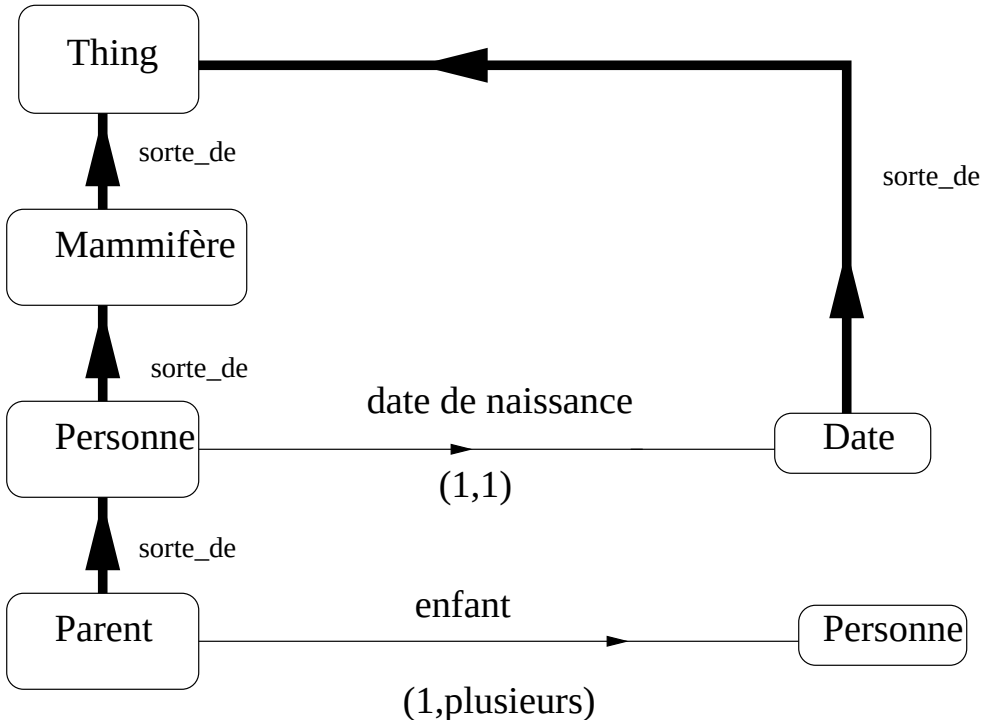


FIGURE 1.11 – Réseau sémantique

Les réseaux sémantiques ont été mis en œuvre dans le cadre des SE, en particulier dans PROSPECTOR. L'idée d'utiliser ce formalisme est attribué à Quillian[22] (1968). Considérons une phrase comme `Médor sent Mirza` qui peut se représenter comme dans la figure 1.12. Cette approche peut être raffinée en considérant que chaque concept (ici `Médor`, `Mirza`) est relié à sa famille d'appartenance par un lien `est_un` donnant lieu à la représentation de la figure 1.13. Cette approche peut encore une fois être améliorée en introduisant des liens entre concepts par le biais de la relation `sorte_de`. Dans ce cadre une phrase telle que `Médor est un basset qui est un chien` se représente comme dans la figure 1.14. Dans ce cadre, les propriétés peuvent se transmettre des concepts les plus élevés (dans la hérarchie) ves les fils (au sens de la relation `sorte_de`). Malheureusement, on peut consta-

[21] Un nœud A subsume un nœud B s'il est plus général (Mammifère vs Personne dans la figure 1.11)
[22] « Semantic Memory »in *Semantic Information Processing*, M. Minsky (ed.), MIT Press, pp 227–270

ter le manque de standardisation du formalisme et le risque d'associations abusives en suivant les chaînages induits par le réseau.

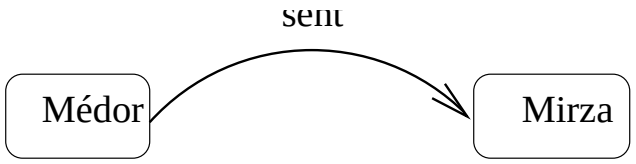


FIGURE 1.12 – Médor sent Mirza

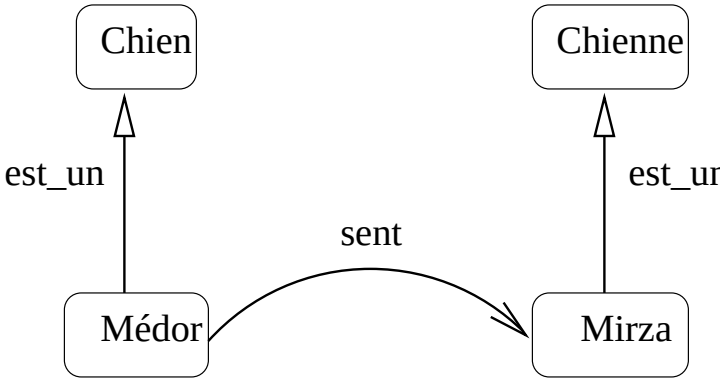


FIGURE 1.13 – Relation `est_un`

Il est certain qu'un accroissement de la complexité du formalisme augmente sa puissance d'expressions et de raisonnement, cependant il est nécessaire, comme dans toute application informatique, de faire un compromis entre complexité des structures de données et complexité des algorithmes utilisés pour leur exploitation.

Règle de production
À la suite de l'émergence des SE (cf chapitre 12 de la partie IV) les règles de production ont acquis leur lettre de noblesse (en particulier grâce à MYCIN, voir le chapitre 12 de [11]). Une règle de production est de la forme :

`Si` partie-si `Alors` partie-alors (α).

La « partie-si »est une formule logique qui doit être vérifiée pour que la règle soit déclenchée. La « partie-alors »peut correspondre à un fait (ajout ou retrait) ou à une hypothèse émise par le système, au déclenchement d'une autre règle ; α est un coefficient traduisant l'incertitude associée à la règle ; suivant le domaine dans lequel on travaille on parle de coefficient de vraissemblance, de facteur de plausibilité ou de certitude, de degré de vérité.

Le noyau des systèmes à base de connaissances reposant sur des règles de production comprend trois parties :

1. une base de règles (connaissance permanante),
2. un moteur d'inférences (mécanisme de raisonnement) qui peut être de différentes natures,
3. une base de faits (mémoire de travail, connaissance temporaire)

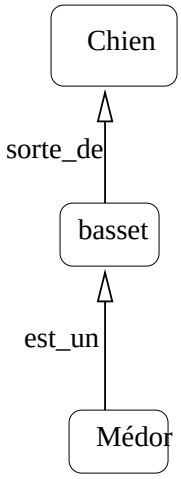


FIGURE 1.14 – Relation `sorte_de`

En plus, de tels systèmes se doivent d'intégrer une interface Homme/Machine performante. Nous n'en dirons pas plus pour l'instant, ces concepts étant approfondis dans le chapitre 12 de la partie IV de ce même document. On peut cependant noter qu'il existe des différences notables entre règle de production (Si p alors q) et formule logique ($p \supset q$). D'une part, la relation de causalité n'est pas nécessaire dans le cas de formules logiques et d'autre part l'adjonction du coefficient de vraissemblance dans le cadre des règles de productions. Pour conclure ce paragraphe, on constate que de tels systèmes pêchent par une faible capacité à organiser les connaissances, c'est pourquoi on les couple (dans le cas de systèmes industriels) avec des systèmes de frames.

Frames
M. Minsky (1975) a introduit cette approche dans le cadre de la vision par ordinateur. Ce travail est directement inspiré des travaux menés en psychologie cognitive sur la mémoire chez l'homme (F. C. Bartlett 1932) et visant à proposer un modèle de représentation d'expériences passées pour résoudre un problème nouveau. R. Schank (1977) s'est inspiré de ces travaux pour la compréhension des langues naturelles. Depuis, cette approche a été englobée dans les langages orientés objets (voir à ce propos le livre de G. Masini [65]). Un frame est une structure de données qui représente un objet typique ou une situation stéréotype, c'est en fait un réseau hierarchisé de nœuds et de relations. On peut retenir parmi les propriétés essentielles des frames, l'existence de valeur par défaut pour un attribut (slot), permettant la notion de raisonnement révisable ; de contraintes que doivent satisfaire les attributs et qui sont propagées au cours du raisonnement ; de procédure (méthode) qui se déclenche lorsque la valeur d'un attribut est requise. Comme dans un réseau sémantique, un mécanisme d'héritage est mis en œuvre.

Conclusion

Il existe de nombreuses possibilités de représenter des connaissances, mais hélas aucune n'est la panacée universelle. Toutes ont des avantages et des inconvénients qu'il est nécessaire d'apprécier en fonction de l'application souhaitée. Cependant, certaines représentations peuvent être couplées. En définitive, le choix final reposera sur le degré de compromis souhaité entre efficacité (puissance de raisonnement, puissance d'expression et de représentation) et clarté.

Le raisonnement en IA

Le raisonnement est une activité intentionnelle à la différence d'autres telles que la perception. On considère le raisonnement comme une activité élaborée [72, 58, 45] pouvant se définir comme un enchaînement d'énoncés conduit en fonction d'un but qui peut suivant le cas être de démontrer, justifier ou expliquer.

Le raisonnement est souvent considéré sous plusieurs angles tels que :

- le raisonnement formel, basé sur la manipulation *syntaxique* d'énoncés symboliques à l'aide de règles. Une telle le approche se trouve dans le raisonnement logique que nous expliciterons dans les chapitres 3 et 4 de la partie II. Il regroupe le raisonnement inductif et déductif.

- le raisonnement par analogie, qui est très courant chez l'homme mais qui est mal défini (et donc délicat à mettre en œuvre).

- le raisonnement par abstraction et généralisation est répandu chez l'homme mais mal défini, ce type de raisonnement, dans le cadre de simulation sur machine peut être mis en œuvre par le biais d'héritage, et est en relation avec la classification (voir à ce propos le chapitre 8 de [45]).

Dans son article « Logique et induction : un vieux débat », J.-P. Ganascia propose la distinction suivante : l'*induction* conclut du particulier à l'universel selon le principe de généralisation : ce qui convient à plusieurs choses, convient aussi aux autres choses du même genre. Tandis que l'*analogie* conclut de la ressemblance particulière entre deux choses à la ressemblance totale selon le principe de spécification.

Il est possible de distinguer deux approches relevant de l'induction. La première est *conjecturale* dans le sens où les conclusions auxquelles on peut aboutir n'ont pas un degré de certitude absolue (comme « le moineau est un oiseau, le moineau vole ; donc tous les oiseaux volent »). La seconde est *certaine*, c'est notamment le cas pour le principe utilisé dans un raisonnement par récurrence.

Chapitre 2

Vous avez dit Logique ?

La logique classique intègre, généralement deux parties, d'une part la logique propositionnelle (parfois aussi appelé « calcul des énoncés ») dont la brique de base est la *proposition* représentant soit un évènement soit un fait, elle étudie toutes les associations possibles ; d'autre part la logique des prédicats, dont l'élément fondamental est la relation et les termes internes aux propositions.

Je souhaiterai donner un (double) avertissement aux lecteurs en forme d'introduction, le premier de Jean Largeault (cf préface du livre de B. Ruyer [81]), « l'étude de la logique n'a jamais aidé quiconque à mieux raisonner », et le second de B. Ruyer (opus cité) « on pourrait définir la logique comme la science des règles qui légitiment l'emploi du mot **donc** ».

Avant d'attaquer l'étude du calcul des énoncés, il semble important d'introduire quelques notions fondamentales telles que :

Paradoxe
Le barbier de la ville rase tous les hommes qui ne se rasent pas eux-même, et seulement ceux-là.
Le barbier se rase-t-il ?

Tiers exclu
Une porte est ouverte ou fermée. On peut bien entendu raffiner le concept d'ouverture (en introduisant la notion d'entre-ouverte), mais cela ne change en rien la propriété qui peut s'énoncer alors : Une porte est entr'ouverte ou ne l'est pas.

Vrai et Démontrable
Considérons une affirmation G de la forme « G n'est pas démontrable ». Et faisons l'hypothèse que G est fausse, par hypothèse G est donc non démontrable, d'où l'on tire que G est vraie[1], ce qui nous mène à une contradiction. Donc G est vraie et G est non démontrable.

Changement de niveau
La notion de méta-règle permet de décrire des objets appartenant à un niveau inférieur.

Exemple 2.0.1

Soit l'ensemble des entiers naturels. Une méta-règle pourrait être : « le plus petit entier qui s'écrive (en français) en deux mots », il s'agit du nombre 17 (dix-sept). ◃

Considérons la méta-règle : « Trouver l'ensemble des entiers qui ne peuvent pas s'écrire en moins de quinze mots ». Le plus petit nombre appartenant à cet ensemble est 1 297 297. Mais comme c'est le plus petit, on peut le définir en quatorze (14) mots : « (1) le (2) plus (3) petit (4) nombre (5) ne (6) pouvant (7) pas (8) s' (9) écrire (10) en (11) moins (12) de (13) quinze (14) mots ». Ce nombre ne fait donc pas parti de l'ensemble cherché, en itérant le processus on peut établir que l'ensemble recherché est l'ensemble vide !.

[1] la notion de vérité repose sur la notion de démontrabilité. Quelque chose qui n'est vrai ne peut en aucune façon être démontrable.

Dans la même veine, certains artistes tels que René Magritte (1898-1967) (cf le tableau[2] reproduit dans la partie gauche de la figure 2.1 et visible sur le site `http://www.virtuo.be/gallery/gallery/af.html` ou Maurits Cornelius Escher (1898-1972) (cf le tableau[3] reproduit dans la partie droite de la figure 2.1 visible sur le site web `http://www.mclink.it/personal/MC0006/escher/escher40.htm`) se sont intéressés à l'auto-référence.

Ceci n'est pas une pomme

Mains se dessinant (1948)

FIGURE 2.1 – Magritte et Escher : auto-référence

Pour terminer, je souhaiterai souligner un point qui me semble important quant à l'intérêt de la logique pour l'informatique en générale et la programmation en particulier.
La logique est imporatnte da,s le cadre de l'étude des propriétés de programmes, la validation de programmes et l'étude de la sémantique des langages de programmation. Hoare (1969) a défini un système axiomatique simple permettant de transformer les raisonnements basés sur la méthode des assertions de Floyd en déductions formelles. Cependant, ces logiques ne sont utilisables que dans le cadre de la programmtion séquentielle. La programmation parallèle [7] repose sur la notion de logique temporelle dont l'introduction est due à Pnuelli (1977). Les programmes parallèles font appel à la notion de « contexte intensionnel »qui met en défaut le principe de *composionalité*. Selon ce principe, le sens d'une expression est le résultat de la combinaison de ces sous-expressions. La logique temporelle fait partie des logiques dites *modales* car elle nécessite des opérateurs dont la sémantique est proche du langage usuel : « la prochaine fois », « jusqu'à », « avant », « après », « toujours », ...

Nous allons dans les chapitres suivants voir différents systèmes logiques, nous commencerons par le système le plus simple et le plus classique qui est celui du « calcul des énoncés ». Nous aborderons les notions de démonstration, de démantique, de syntaxe, de systèmes formels. Nous verrons

[2] (C) All reproduction rights of Magritte's works are reserved. C. Herscovici, Brussels - Belgium.
[3] All M.C. Escher works (c) Cordon Art-Baarn-the Netherlands. All rights reserved.

ensuite que ce système est extrêmement limité, et nous serons amenés à l'étendre de différentes manières.

Chapitre 3

Logique propositionnelle

3.1 Première approche

Définitions tirées du petit larousse [70] : « Science du raisonnement en lui même, abstraction faite de la matière à laquelle il s'applique et de tous processus psychologiques ».
La logique mathématique est quant à elle définie comme : « une théorie scientifique des raisonnements excluant les processus psychologiques mis en œuvre et qui se divise en

1. Calcul des propositions.
2. Calcul des prédicats. »

Toujours dans le même dictionnaire on trouve : « la logique constitue une langue, i.e. un système de symboles et de variables liés par des opérateurs qui déterminent la structure interne des propositions et les relations entre les propositions. »

La logique remonte à Aristote (IV^e siècle avant JC) qui posa les bases du syllogisme que les philosophes scholastiques formalisèrent au Moyen-âge. C'est au XIX^e siècle avec Bolzano, Boole et de Morgan que la logique devient mathématique. Frege (1848–1925) est le fondateur de la logique formelle, théorisée par Russel (1872–1970) et Wittgenstein (1889–1951).

Avant de commencer je vous propose le petit exercice suivant :

Exercice 3.1
Tâche de sélection de Wason :

1. On considère des cartes - portant d'un côté une lettre, de l'autre un chiffre, dont seule une face est visible, et on cherche à vérifier la règle suivante : « S'il y a un R d'un côté, il y a un 2 de l'autre ». Quelles sont les cartes à retourner si l'on voit 4 cartes marquées respectivement par R, J, 2 et 8.
2. On considère des enveloppes, dont seule une face est visible, et on cherche à vérifier la règle suivante : « Si la lettre est cachetée alors elle porte un timbre sur l'autre face ». Quelles sont les lettres à retourner si l'on voit 4 enveloppes telles que : cachetée, non cachetée, affranchie, non affranchie.

▹

3.1.1 Grèce antique

Platon et Aristote sont les deux grandes figures de la Grèce antique. Dont l'attitude philosophique est diamétralement opposée. Pour Platon, la connaissance est donnée originellement à l'homme qui doit simplement se ressouvenir, et dont le célèbre dialogue dit du Menon est une illustration. Pour Aristote, l'âme est un cahier sur lequel le monde s'inscrit où plutôt sur lequel nous inscrivons le monde qui nous parvient au cours de la vie.

La syllogistique est au cœur de l'approche d'Aristote (Grèce Antique). Aristote définit la notion de *termes* tels que « homme », « science », « qui peut être enseigné ». À partir de deux termes, on construit une *proposition*, les termes sont reliés à l'aide d'une *copule* c'est-à-dire un verbe. Par exemple : `La science est une vertu` ; `Toute science est « qui peut être enseigné »`, ce que l'on peut écrire `Toute science peut être enseignée`.

De plus parmi les propositions on distingue plusieurs catégories : *universelle*, *particulière* ou *individuelle*. Qui plus est, ces propositions peuvent être : *affirmative* ou *négative*. Pour se repérer, on a pris l'habitude d'indiquer le type de proposition à l'aide d'une des quatre lettres suivantes :

(A) Affirmative universelle : « Tout B est A ».

(I) Affirmative particulière : « Quelques M sont N ».

(E) Universelle négative : « Nul C n'est A ».

(O) Particulière négative : « Quelques D ne sont pas N ».

Une fois les propositions établies, il convient de donner les règles permettant de passer avec certitude de deux propositions à une troisième. C'est l'objet de la syllogistique que de déterminer ces règles qui prennent la forme de *syllogisme*.

Définition 3.1
Un *syllogisme* est un raisonnement qui contient 3 propositions : la *majeure*, la *mineure* et la *conclusion*. Telles que, la conclusion est déduite de la majeure par l'intermédiaire de la mineure. La majeure et la mineure forment la *prémisse*.

Tout le travail d'Aristote a été de balayer l'ensemble des combinaisons possibles de trois propositions construites sur trois termes suivant les différentes modalités A, E, I, O et à distinguer celles qui étaient universellement valides. C'est au cours du Moyen-âge que ces figures valides ont été nommées au moyen d'un système mnémotechnique utilisant les quatre voyelles.

Exemple 3.1.1
Quelques exemples de syllogismes :

S-1
(Majeure) : Tous les hommes sont mortels, (Mineure) : Tous les grecs sont des hommes, (Conclusion) : Socrate est un grec, donc Socrate est mortel.

S-2
(Majeure) : Tous les chiens ont quatre pattes, (Mineure) : Quelques animaux sont des chiens, (Conclusion) : Quelques animaux ont quatre pattes.

S-3
(Majeure) : Tout ce qui est rare est cher, (Mineure) : Un cheval bon marché est rare, (Conclusion) : Un cheval bon marché est cher.

Le premier syllogisme (S-1) est appelé Barbara *car il utilise trois propositions affirmatives universelles (A). Le second syllogisme (S-2) est appelé* Darii *car il utilise une majeure de type A et deux propositions (mineure et conclusion) de type I. Le dernier syllogisme (S-3) n'est pas considéré comme valide car le sens associé à* `rare` *a changé entre la majeure et la mineure.* ◃

Exercice 3.2
Voici la liste des syllogismes identifiés au Moyen-âge. Ils reposent sur la notion de figure qui indique la position du terme moyen qui disparaît au cours du raisonnement :

1. GH, FG, FH.
2. HG, FG, FH.
3. GH, GF, FH.
4. HG, GF, FH.

Où G désigne le terme moyen et les deux autres termes étant symbolisés par les lettres F et H.

1. Première figure : Barabara, Darii, Celarent.
2. Seconde figure : Ferio, Cesare, Camestres
3. Troisième figure : Festino, Baroco, Darapti(*), Felapton(*), Disamis, Datisi, Bocardo, Ferison.
4. Quatrième figure : Bamalip(*) (Barbari), Dimaris (Dimatis), Camenes (Calentes), Fesapo(*) (Fespamo), Fresison.

Les figures marquées d'une étoile ont changés de statut depuis le Moyen-âge. Pour ces quatres figures, donner un exemple et indiquer s'ils sont ou non universellement valides lorsque l'on considère les interprétations suivantes :

1. Une proposition telle que « Tout F est G »se traduit maintenant par : « S'il existe des x qui ont la propriété F, alors ils ont la propriété G » ; elle est donc **vraie** si aucun x n'a la propriété F.
2. Pour les logiciens du Moyen-âge, elle signifiait plutôt : « Il existe des x qui ont la propriété F, et tous ces x ont la propriété G ».

▹

L'objectif de la logique propositionnelle (ou calcul des propositions) est d'étudier les ennoncés qui sont vrais ou faux.

Exemple 3.1.2

E-1 Un quart d'heure avant sa mort, il était encore vivant.

E-2 Si [S'il pleut alors la route est mouillée] Alors [Si la route n'est pas mouillée Alors il ne pleut pas].

E-3 La terre tourne.

On détecte plusieurs propositions : « Il pleut », « La route est mouillée ». Les trois ennoncés précédents procèdent de logiques différentes : E-1 est une vérité de langage, il faut connaitre le sens des mots pour en déterminer la vérité ; E-2 est une vérité logique, on peut changer les propositions, on reste avec un ennoncé vrai ; E-3 est une vérité factuelle (vérité de fait) ici physique.
Seul le deuxième ennoncé (E-2) relève du calcul des propositions. ◃

3.2 Langage

La logique est un *langage* qui contient des règles et des signes. Ce langage est constitué d'un système de symboles et de variables liés par des opérateurs qui déterminent la structure interne des propositions et les relations existants entre ces différentes propositions.

On considère un alphabet $\mathcal{A}$ constitué de 3 sous-ensembles disjoints :

1. $\mathcal{P} = \{p, q, r, \ldots\}$ un ensemble fini ou dénombrable de variables proositionnelles, éventuellement indicées.
2. Ponct = { (,) }
3. Conn = $\{\neg, \wedge, \vee, \supset, \equiv\}$, ces connecteurs se lisent respectivement `non`, `et`, `ou`, `implique`, `équivalent à`. Nous reviendrons ultérieurement (section 3.4) sur la signification à accorder à ces différents connecteurs, et aux différences notables avec le langage naturel.

à partir de cet alphabet, on construit des mots qui appartiennent à $\mathcal{A}^*$. Sur ces mots, on définit un sous ensemble noté $\mathcal{F}$ constituant les formules propositionnelles.

Définition 3.2
L'ensemble des formules propositionnelles $\mathcal{F}$ construites sur $\mathcal{P}$ est le plus petit ensemble tel que :

- Si $F \in \mathcal{F}$ alors $\neg F \in \mathcal{F}$
- Si F et G sont des formules alors $(F \wedge G)$, $(F \vee G)$, $(F \supset G)$ et $(F \equiv G)$ appartiennent à $\mathcal{F}$.

De cette définition, on peut tirer plusieurs remarques. La première est que l'on a $\mathcal{P} \subseteq \mathcal{F}$, la seconde que $\mathcal{F}$ est bien définie, car $\mathcal{A}^*$ vérifie la définition 3.2, que $\mathcal{P}$ appartient à l'intersection de tous les ensembles vérifiant cette définition.

Il est cependant possible de définir les formules propositionnelles par induction de la façon suivante :

Définition 3.3
1. $\mathcal{F}_0 = \mathcal{P}$
2. $\mathcal{F}_{n+1} = \mathcal{F}_n \cup \{\neg F \,/\, F \in \mathcal{F}_n\} \cup \{(F \bowtie G) \,/\, F, G \in \mathcal{F}_n, \bowtie \in \{\wedge, \vee, \supset, \equiv\}\}$

De la définition 3.3 on constate que la suite $(\mathcal{F}_n)_{n\in\mathbb{N}}$ est croissante. Nous allons maintenant voir que les définitions 3.2 et 3.3 définissent le même objet.

Proposition 3.2.1
$$\mathcal{F} = \bigcup_n \mathcal{F}_n$$

preuve : Nous allons démontrer la proposition en établissant la double inclusion :

- $$\mathcal{F} \supseteq \bigcup_n \mathcal{F}_n$$
  Il suffit d'établir la propriété par récurrence sur n
- $$\mathcal{F} \subseteq \bigcup_n \mathcal{F}_n$$
  Pour cela il suffit de montrer que
  $$\bigcup_n \mathcal{F}_n$$
  vérifie la définition 3.2. Comme par ailleurs, $\mathcal{F}$ est le plus petit ensemble vérifiant cette définition, la démonstration en découle immédiatement.

□

Les lois logiques sur lesquelles reposent le calcul des ennoncés sont :

- Le principe d'identité : $(f \equiv f)$
- Le principe de non contradiction : $\neg(f \wedge \neg f)$
- Le principe du tiers exclus : $(f \vee \neg f)$
- Le principe de la double négation : $(\neg\neg f \equiv f)$

Définition 3.4
La profondeur d'une formule F est le plus petit entier n tel que $F \in \mathcal{F}_n$. On peut associer à une formule logique, son arbre syntaxique (les nœuds internes sont les connecteurs logiques, les feuilles sont les variables propositionnelles), la profondeur de la formule est aussi la profondeur de l'arbre (ou sa hauteur, cf les cours d'algorithmique de Licence MASS [26], voir aussi le rappel donné en annexe B page 171).

Pour illustrer la définition 3.4 rien ne vaut quelques petits exemples, on se reportera à la figure 3.1 pour les représentations des formules sous forme d'arbres syntaxiques.

EXEMPLE 3.2.1

- *p a la profondeur 0* ($p \in \mathcal{P} \stackrel{\text{def}}{=} \mathcal{F}_0$)
- *$\neg p$ a la profondeur 1*
- *$((\neg p \vee q) \wedge r)$ est de profondeur 3*

◁

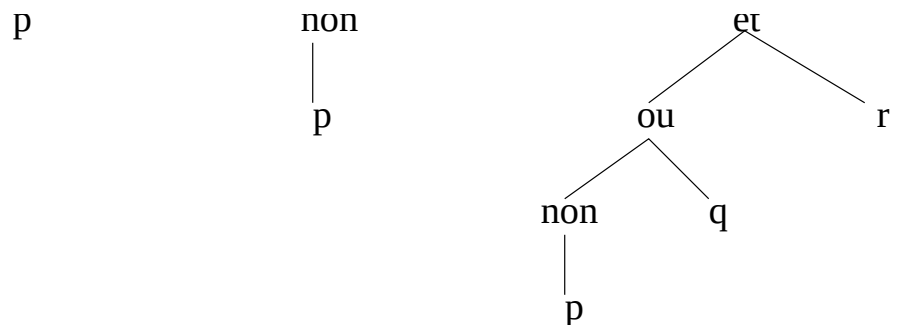


FIGURE 3.1 – Arbres Syntaxiques

DÉFINITION 3.5
On dira qu'une formule G est une sous-formule d'une formule F, si l'arbre syntaxique de G est un sous-arbre de l'arbre syntaxique de F.

Si nous reprenons les formules de l'exemple 3.2.1, on constate que p est une sous formule de $\neg p$ qui est elle-même une sous formule de $((\neg p \vee q) \wedge r)$. On voit cependant que la définition ne considère pas que $(q \wedge r)$ est une sous-formule de $((\neg p \vee q) \wedge r)$.

3.3 Théorie des modèles et Théorie de la démonstration

Lorsque l'on désire étudier une formule logique, on peut le faire selon deux angles que sont la théorie des modèles et la théorie de la démonstration. La première a pour but d'établir un mécanisme sémantique d'évaluation des formules, il s'agit d'une formalisation de la notion intuitive que nous avons de la vérité ou de la fausseté d'une phrase en fonction de ses constituants. La seconde est un mécanisme purement syntaxique d'évaluation. Il s'agit, dans ce cadre, d'appliquer des règles mécaniques sans s'intéresser au sens véhiculé par les composants. Les règles sont appelées, *règles d'inférence* et portent sur des *jugements* que l'on souhaite prouver mécaniquement - dans ce cadre un jugement porte sur la véracité d'une formule.

3.4 Sémantique dans le calcul des propositions

Le rôle de la *syntaxe* est de différencier une formule d'un mot quelconque de $\mathcal{A}^*$. La tâche dévolue à la *sémantique* est d'attribuer une *signification* aux énoncés, plus particulièrement une valeur de vérité.

Une proposition est soit vraie, soit fausse. On peut alors lui attribuer un domaine sémantique que l'on notera {`vrai`, `faux`} ou {0, 1} (dans ce cas, 0 représente le `faux` et 1 le `vrai`). Interpréter une formule revient à lui associer une valeur de vérité. Une sémantique est une loi compositionnelle qui dépend de ses constituants. Les connecteurs logiques sont considérés comme vérifonctionnels puisque une table de vérité définit complètement leur comportement. Ci-après, nous donnons la table de vérité associée à chaque connecteur :

p	q	$\neg p$	$(p \wedge q)$	$(p \vee q)$	$(p \supset q)$	$(p \equiv q)$
0	0	1	0	0	1	1
0	1	1	0	1	1	0
1	0	0	0	1	0	0
1	1	0	1	1	1	1

Avant d'aller plus loin, il semble nécessaire de comparer la sémantique attribuée aux connecteurs logiques et leur signification en langage naturel (français, anglais, ...).

Et logique vs et français
Le `et` logique est commutatif, ce qui n'est pas le cas en français puisqu'il peut y avoir un lien de causalité entre les deux propositions. On pourra, pour s'en assurer, comparer les deux phrases suivantes : « Il prit peur et le tua », « Il le tua et prit peur ». Dans la même veine on pourra étudier les deux phrases : « Elle eut un enfant et se maria »et « Elle se maria et eut un enfant ».

Ou logique vs ou français
Très souvent le `ou` français a une connotation d'exclusivité (les deux propositions ne peuvent être vraies simultanément), alors que le `ou` logique est par nature inclusif. à titre d'exemple on pourra considérer les phrases : « fille ou garçon », « fromage ou dessert », « le pistolet ou la corde », ...
Pour exprimer le `ou exclusif` en logique on utilisera au choix l'opérateur `xor` avec une table de vérité ad-hoc, correspondant à l'une des formules logiques $((\neg p \wedge q) \vee (\neg q \wedge p))$; $((\neg p \vee \neg q) \wedge (p \vee q))$.

Implication logique vs implication française
L'implication française qui se traduit par l'usage de la syntaxe « si hypothèse alors conclusion »indique une relation de causalité entre hypothèse et conclusion. L'implication logique[1] $(p \supset q)$ est en fait une notation commode pour exprimer l'alternative[2] $(q \vee \neg p)$ qui montre qu'aucune relation n'est exigée entre les deux propositions (ou formule logique) représentant l'hypothèse et la conclusion.

3.5 Puissance d'expression

Nous allons maintenant établir que les cinq connecteurs ne sont pas nécessaires pour exprimer les formules logiques. Il existe d'autres ensembles de connecteurs offrant toute la puissance d'expression de la logique propositionnelle. Ces ensembles seront appelés *base* en regard de la logique[3]. Pour s'assurer qu'un ensemble de connecteurs forme une base, on se restreindra à la comparaison des tables de vérité de chaque expression. Dans la suite nous utiliserons le signe $=$[4] entre deux expressions, pour indiquer qu'elles ont même table de vérité, la construction et la vérification étant laissées au lecteur à titre d'exercice.

3.5.1 Bases

Nous développerons ici l'étude de la base $\{\neg, \supset\}$, les autres étant laissées à titre d'exercice.

$$\begin{array}{lcl} (p \vee q) & = & (\neg p \supset q) \\ (p \wedge q) & = & \neg(p \supset \neg q) \\ (p \equiv q) & = & \neg((p \supset q) \supset \neg(q \supset p)) \end{array}$$

[1] On dit aussi « implication matérielle ».

[2] Pour s'assurer de la correction de cette expression, il suffit de construire les tables de vérité associées à chacune des expressions et de vérifier que les résultats sont identiques.

[3] Dans [60] on parle d'ensemble adéquat

[4] Le signe = appartient au méta-langage et non au langage propositionnel. Dire que $F = G$ c'est dire que $F \equiv G$ est une tautologie.

On peut établir que cet ensemble est minimal, pour cela il faut prouver qu'un connecteur ne peut pas s'exprimer en fonction de l'autre. En fait il suffit de prouver que $\neg p$ ne peut s'exprimer par le biais du connecteur $\supset$. Comme $(p \supset p)$ est vraie quelque soit la valeur de p, la démonstration est immédiate.

Exercice 3.3

1. Démontrer que $\{\neg, \wedge, \vee\}$ forme une base en regard de la logique. Cette base n'est pas minimale, puisque les ensembles $\{\neg, \vee\}$ et $\{\neg, \wedge\}$ sont aussi des bases.
2. Démontrer que l'opérateur ternaire `ite` (si-alors-sinon) défini par la table de la figure 3.2 forme une base.

x	y	z	`ite`(x, y, z)
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1
x	y	z	$((x \wedge y) \vee (\neg x \wedge z))$

Figure 3.2 – Table de vérité de l'opérateur `ite`

3. Démontrer que l'opérateur binaire $\uparrow$ (`non-et`[5]) qui vérifie $(p \uparrow q) = \neg(p \wedge q)$ forme une base.
4. Démontrer que l'opérateur binaire $\downarrow$ (`non-ou`[6]) qui vérifie $(p \downarrow q) = \neg(p \vee q)$ forme une base.

▹

Remarque : *Les deux connecteurs $\uparrow$ et $\downarrow$ sont les seuls opérateurs binaires ayant la propriété d'être une base.* •

3.5.2 Formules et interprétations

Avant de commencer l'étude des interprétations que l'on peut associer à une formule, il semble utile de regarder plus précisément, quelles sont les fonctions à 2 variables que l'on peut définir. Comme chaque variable peut prendre 2 valeurs, il y a donc 4 couples distincts de valeurs, à chaque couple on peut associer (on dit aussi assigner) 2 réponses (0 ou 1), il y a ainsi 16 fonctions à deux variables possibles qui sont résumées dans la table de la figure 3.3.

p	q	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}	f_{16}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	1	0	0	1	1	0	1	0	1	0	0	1	1	0	1
1	0	0	0	1	0	1	0	1	1	0	0	1	0	1	0	1	1
1	1	0	0	0	1	0	1	1	1	0	0	0	1	0	1	1	1

Figure 3.3 – 16 fonctions à 2 variables

à partir de la figure 3.3, on reconnaît les identités suivantes : $f_1(p, q) = 0$ (fonction uniformément fausse), $f_2(p, q) = (\neg p \wedge q)$, $f_3(p, q) = (p \wedge \neg q)$, $f_4(p, q) = (p \wedge q)$, $f_5(p, q) = (p \equiv \neg q) = (\neg p \equiv$

[5]Cet opérateur, aussi appelé barre de Sheffer, correspond aux portes NAND des circuits logiques.
[6]Cet opérateur, aussi appelé barre de Sheffer duale, correspond à une porte NOR.

$q) = \neg(p \equiv q)$, $f_6(p,q) = q$, $f_7(p,q) = p$, $f_8(p,q) = (p \vee q)$, $f_9(p,q) = (p \downarrow q)$, $f_{10}(p,q) = \neg p$, $f_{11}(p,q) = \neg q$, $f_{12}(p,q) = (p \equiv q)$, $f_{13}(p,q) = (p \uparrow q)$, $f_{14}(p,q) = (p \supset q)$, $f_{15}(p,q) = (q \supset p)$, $f_{16} = 1$. La fonction f_1 est appelée **contradiction** dont l'expression type est $(p \wedge \neg p)$, tandis que la fonction f_{16} est appelée **tautologie** dont l'expression type est $(p \vee \neg p)$. Il en résulte que la négation d'une contradiction est une tautologie et réciproquement.

Définition 3.6
Un fonction d'interprétation[7] (ou interprétation) est une application **i** qui à toute variable propositionnelle p associe une valeur de vérité.

On étend facilement cette notion aux formules, par le biais des tables de vérité. Cette extension noté **I** est aussi une interprétation.

Définition 3.7
Une interprétation qui rend une formule F vraie est appelée un *modèle* de F.

Remarque : *Une formule possédant k variables propositionnelles admet 2^k interprétations différentes.* •

Exemple 3.5.1
Prenons la formule $F = (p \vee q)$, et considérons l'interprétation I vérifiant, $I[\![p]\!] = 1, I[\![q]\!] = 0$, c'est un modèle de F, puisque $I[\![F]\!] = 1$. ◁

Définition 3.8
Un *littéral* est soit une variable propositionnelle, soit la négation d'une variable propositionnelle. En d'autre terme, si $p \in \mathcal{P}$, alors p et $\neg p$ sont des littéraux, le premier sera dit *positif* et le second *négatif*.

Une interprétation détermine une partition de l'ensemble des littéraux, en deux sous-ensembles notés L_0 et L_1, chaque ensemble contenant exactement un élément de la paire $(p, \neg p)$. Par exemple, si on considère l'interprétation I de l'exemple 3.5.1, on a $\{p, \neg q\} \subseteq L_1$ et $\{\neg p, q\} \subseteq L_0$. Une conséquence immédiate de cette définition est qu'une interprétation vérifie les trois points suivants :

1. $\mathrm{I}[\![F]\!] = 1 - \mathrm{I}[\![\neg F]\!], \forall F \in \mathcal{F}$.
2. $\mathrm{I}[\![F \vee G]\!] = \max(\mathrm{I}[\![F]\!], \mathrm{I}[\![G]\!]), \forall F, G \in \mathcal{F}$.
3. $\mathrm{I}[\![F \wedge G]\!] = \min(\mathrm{I}[\![F]\!], \mathrm{I}[\![G]\!]), \forall F, G \in \mathcal{F}$.

Définition 3.9
Une formule sera dit sémantiquement consistante (ou plus simplement *consistante*) si elle admet un modèle. Une formule qui n'est pas consistante est dite *inconsistante*. Une formule est dite *valide*[8] si elle est toujours vraie. Enfin une formule qui n'est ni valide, ni inconsistante sera dite *contingente*.

Afin de maitriser le vocabulaire précédent, je vous enjoins à résoudre le petit exercice suivant :

Exercice 3.4

1. Trouver des formules simples et utilisant la base $\{\neg, \vee, \wedge\}$ (au sens où elles mettent en jeu peu de connecteurs logiques, et peu de variables propositionnelles) qui soient : valides, inconsistantes, consistantes mais non valides, contingentes.
2. Trouver une formule valide utilisant comme connecteur principal l'implication logique ($\supset$).

▷

[7] On parle parfois de fonction de valuation.
[8] On dit aussi que c'est une tautologie.

Notation : On utilisera la notation $\models F$ pour indiquer qu'une formule F est une tautologie.
Par extension, si $\mathcal{E}$ est un ensemble de formules, on notera $\mathcal{E} \models F$ pour indiquer que toute interprétation I modèle de $\mathcal{E}$ est modèle de F. On dit aussi que « F est une conséquence logique de $\mathcal{E}$ ».
Lorsque l'ensemble de formules $\mathcal{E}$ se réduit à une seule formule, on omet les accolades. Ainsi $\{G\} \models F$ est noté $G \models F$. ◇

Remarque :

- *$F \models G$ revient à dire que $(F \supset G)$ est une tautologie.*
- *$\{F_1, F_2, \ldots, F_k\} \models G$ revient à dire que*

$$\models (\bigwedge_{i=1}^{k} F_i \supset G)$$

•

Dire qu'un ensemble $\mathcal{E}$ de formules est cohérent (ou consistant), c'est dire que tous les élements de $\mathcal{E}$ admettent le même modèle (ou encore qu'il existe un modèle pour la conjonction des formules de $\mathcal{E}$). Enfin, dire que F est conséquence logique d'un ensemble de formules $\mathcal{E}$, c'est établir que $\mathcal{E} \cup \{\neg F\}$ est incohérent (ou inconsistant). Cette dernière approche est à la base de la *déduction* qui consiste à déterminer si une formule F est conséquence logique d'un ensemble de formules $\{F_1, F_2, \ldots, F_k\}$.

Les questions que l'on peut vouloir résoudre sont :

1. Déterminer si un mot $m \in \mathcal{A}^*$ est une formule. Il s'agit d'un problème trivial.
2. Déterminer si une formule $F \in \mathcal{F}$ est
   - consistante,
   - valide,
   - contingente.

   sont des problèmes qui sont soient difficiles, soient fastidieux.

On peut résoudre ces questions par les *arbres sémantiques* ou par des *algorithmes de réduction* que nous allons maintenant étudier.

Arbres sémantiques

Soit $\mathcal{E}$ un ensemble fini ou dénombrable de propositions.

Définition 3.10
Un arbre sémantique est un arbre binaire tel que :

- chaque arc est étiqueté par un littéral,
- issu d'un même nœud les arcs sont étiquetés par un littéral et son opposé,
- dans une branche donnée (chemin depuis la racine), on ne rencontre au plus qu'une seule occurence d'un littéral,
- dans aucune branche on ne trouve un littéral et son opposé.

Remarque : *Si $\mathcal{E}$ est fini, l'arbre est fini. Si $\mathcal{E}$ est infini, l'arbre est infini (une branche au moins est infinie).* •

Remarque : *L'ordre des variables influe sur la construction de l'arbre sémantique. Rechercher l'ordre permettant de construire l'arbre sémantique de taille minimum (le moins de sommets possible) est un problème difficile.* •

à partir de cette construction, il est possible de définir une interprétation partielle au niveau des nœuds en fonction du chemin depuis la racine. Cette interprétation n'est pas définie pour une variable $p \in \mathcal{E}$, si aucun des littéraux p et $\neg p$ n'appartient au chemin. On dira, que l'arbre sémantique est *complet* si chaque feuille de l'arbre est une interprétation totale. En 1950, Quine a établi que *si toute extension possible d'une interprétation partielle attribue la même valeur de vérité à une formule, il est inutile de continuer à développer le sous-arbre.*

On peut définir deux représentations graphiques (équivalentes) pour un arbre sémantique. La première découle directement de la définition 3.10, la seconde adopte la représentation suivante :

- un nœud est étiqueté par une variable propositionnelle,
- un arc est étiqueté 1, si le littéral est pris positivement et 0 sinon.

La figure 3.4 donne les deux représentations possibles pour la formule $(p \vee q)$.

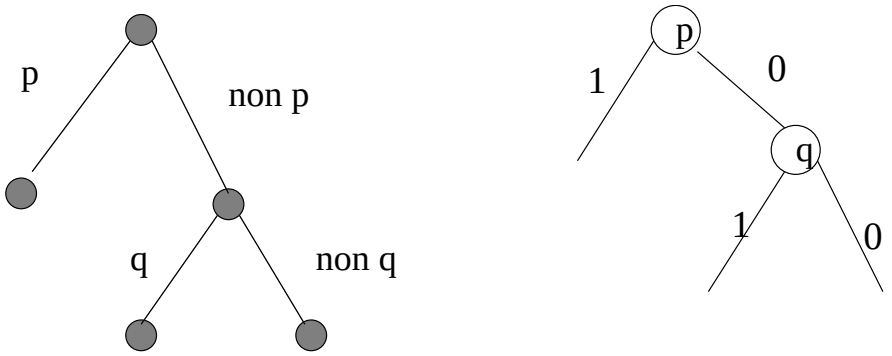


FIGURE 3.4 – Arbre sémantique de $(p \vee q)$

EXERCICE 3.5

- Construire l'arbre sémantique associé à la formule $((((p \wedge q) \supset r) \wedge (p \wedge q)) \supset (p \supset r))$. En prenant comme ordre sur les variables, l'ordre de rencontre dans la formule.
- Construire l'arbre sémantique associé, dans le cas où l'ordre est r, p, q.

▹

Algorithme de réduction

Cette méthode correspond à ce que l'on appelle en mathématiques une preuve par l'absurde, cette approche est particulièrement efficace dans le cas où la formule utilise un grand nombre de connecteurs $\supset$. Nous allons illustrer cette approche au travers d'un exemple. Avant cela, nous introduisons le concept de *substitution uniforme*.

DÉFINITION 3.11
Le principe de la substitution uniforme consiste à remplacer, dans une formule logique, toutes les occurences d'une proposition par une formule, on notera $F_{[p/f]}$ cette opération, où F est la formule sur laquelle on applique la substitution, p est la proposition remplacée et f la formule de remplacement.

Le principe de la substitution uniforme permet de générer autant de tautologies que l'on veut. En effet, si F est une tautologie, si p est une proposition et si f est une formule alors $F_{[p/f]}$ est une tautologie.

Exemple 3.5.2
Considérons la formule $F = (p \vee \neg p)$ et la formule $f = (q \wedge \neg r)$. F est une tautologie comme l'est aussi $F_{[p/f]} = ((q \wedge \neg r) \vee \neg(q \wedge \neg r))$ ◁

Exemple 3.5.3
Soit la formule $F = (((p \wedge q) \supset r) \supset (p \supset (q \supset r)))$.

- *Posons $F_1 = ((p \wedge q) \supset r)$, et $F_2 = (p \supset (q \supset r))$. Et faisons l'hypothèse que $I[\![F]\!] = 0$. Grâce au principe de substitution, on établit que $I[\![F_1]\!] = 1$ et $I[\![F_2]\!] = 0$. D'où l'on tire que $I[\![p]\!] = 1$ et $I[\![(q \supset r)]\!] = 0$, c'est-à-dire que $I[\![q]\!] = 1$ et $I[\![r]\!] = 0$.*
- *Remplaçons ces valeurs dans F_1, on aboutit à $I[\![F_1]\!] = 0$, ce qui est contraire aux hypothèses.*
- *On en déduit que F est une tautologie.*

◁

Il est possible de construire des formules telles que la méthode par arbres sémantiques et l'algorithme de réduction ne sont pas plus efficaces qu'un algorithme purement énumératif. Le problème de savoir si une formule est valide est un problème NP-complet (pour en savoir plus sur le sujet, nous renvoyons le lecteur à P. Wolper [102] et au cours de Recherche Opérationnelle de la maîtrise MASS [28]).

3.5.3 formes duales

Définition 3.12
Soit une formule F ne contenant que des connecteurs pris dans la base $\{\neg, \vee, \wedge\}$, et soit $\overline{F}$ la formule obtenue à partir de F en interchangeant les connecteurs $\wedge$ et $\vee$ et en remplaçant les littéraux par leurs opposés. $\overline{F}$ est la forme `duale` de F.

On a le théorème suivant :

Théorème 3.1
Pour toute formule $F \in \mathcal{F}$ construit sur la base $\{\neg, \vee, \wedge\}$, on a $\overline{F} = \neg F$.

preuve : La démonstration se fait par induction sur le nombre de connecteurs apparaissant dans la formule F. □

Corrolaire 3.1.1 Soient $F, G, H \in \mathcal{F}$ des formules construites sur la base $\{\neg, \vee, \wedge\}$, on a les propriété suivantes :

1. Si $F = G$ alors $\overline{F} = \overline{G}$
2. Si $F = (G \vee H)$ alors $\overline{F} = (\overline{G} \wedge \overline{H})$
3. Si $F = (G \wedge H)$ alors $\overline{F} = (\overline{G} \vee \overline{H})$.

3.5.4 formes normales

On identifie deux types de formes normales : les formes normales conjonctives (`fnc`) et les formes normales disjonctives (`fnd`). Que l'on peut définir de la manière suivante :

Définition 3.13
Une *forme normale conjonctive* est soit un littéral, soit une conjonction de formules écrites comme disjonctions de littéraux.
Une *forme normale disjonctive* est soit un littéral, soit une disjonction de formules écrites comme conjonctions de littéraux.

On notera que toute formule peut s'écrire, indifféremment en `fnc` ou en `fnd` cette propriété est basée sur les équivalences suivantes :

PROPRIÉTÉ 3.5.1 La négation offre les équivalences telles que $\neg(f \vee g) = (\neg f \wedge \neg g)$, $\neg(f \wedge g) = (\neg f \vee \neg g)$, $\neg\neg f = f$. Les deux premières égalités sont connues sous le nom de loi de *de Morgan*
Les connecteurs `ou` et `et` sont associatifs et distributifs l'un par rapport à l'autre, ainsi on a $((f \vee g) \vee h) = (f \vee (g \vee h))$, $((f \wedge g) \wedge h) = (f \wedge (g \wedge h))$, $(f \wedge (g \vee h)) = ((f \wedge g) \vee (f \wedge g))$, $(f \vee (g \wedge h)) = ((f \vee g) \wedge (f \vee h))$.
Les connecteurs `et` et `ou` sont absorbants au sens où l'on $(p \vee (p \wedge q)) = p$, $(p \wedge (p \vee q)) = p$.

Grâce à ces propriétés, il est possible de simplifier des expressions et de trouver ainsi des valeurs de vérité plus aisément. Ces simplifications sont aussi utilisées lorsque les formules sont mises sous forme clausale afin de préparer leur résolution par le biais de la méthode de Robinson que nous verrons dans la section 3.6.2.

Algorithme de normalisation

Il existe deux méthodes pour mettre une formule quelconque sous forme normale, la première passe par une suite d'équivalence (au sens du signe = tel que défini page 41) en remplaçant le membre gauche par le membre droit. Cet méthode peut se résumer par les règles suivantes :

`Suppression de` $\equiv$
$(F \equiv G) = ((F \supset G) \wedge (G \supset F))$
`Suppression de` $\supset$
$(F \supset G) = (\neg F \vee G)$
`Déplacement de` $\neg$ `à l'intérieur des formules`
$\neg (F \vee G) = (\neg F \wedge \neg G)$
$\neg (F \wedge G) = (\neg F \vee \neg G)$
$\neg \neg F = F$
`Distributivité des opérateurs` $\vee$ `et` $\wedge$
$(F \wedge (G \vee H)) = ((F \wedge G) \vee (F \wedge H))$
$(F \vee (G \wedge H)) = ((F \vee G) \wedge (F \vee H))$

La seconde méthode consiste à déterminer la fonction booléenne associée, puis à construire une formule sous forme normale. Les étapes peuvent se résumer de la manière suivante :
`(1) On détermine les interprétations I telles que` $I[\![f]\!] = 1$
`(2) à chaque` I_k `telle que` $I_k[\![f]\!] = 1$ `est associée une formule` g_k `qui est une conjonction de littéraux`
`(3) La formule g est finalement obtenue par la disjonction des` g_k`.`

Afin de clarifier ces deux méthodes considérons l'exercice suivant de normalisation.

EXERCICE 3.6
Soit à normaliser la formule $\neg(p \equiv (q \supset r))$. Par la première méthode on obtient :
$\neg((p \supset (q \supset r)) \wedge ((q \supset r) \supset p))$
$\neg((\neg p \vee (\neg q \vee r)) \wedge (\neg(\neg q \vee r) \vee p))$
$(\neg(\neg p \vee (\neg q \vee r)) \vee \neg((\neg\neg q \wedge \neg r) \vee p))$
$((\neg\neg p \wedge \neg(\neg q \vee r)) \vee (\neg(\neg\neg q \wedge \neg r) \wedge \neg p))$
$((\neg\neg p \wedge (\neg\neg q \wedge \neg r)) \vee ((\neg\neg\neg q \vee \neg\neg r) \wedge \neg p))$
$((p \wedge (q \wedge \neg r)) \vee ((\neg q \vee r) \wedge \neg p))$
$((p \wedge (q \wedge \neg r)) \vee ((\neg q \wedge \neg p) \vee (r \wedge \neg p)))$
qui est une `fnd`.

Par la seconde méthode on peut construire la table de vérité de la formule est on trouve :

p	q	r	$(q \supset r)$	$(p \equiv (q \supset r))$	f	
0	0	0	1	0	1	$\neg p \wedge \neg q \wedge \neg r$
0	0	1	1	0	1	$\neg p \wedge \neg q \wedge r$
0	1	0	0	1	0	
0	1	1	1	0	1	$\neg p \wedge q \wedge r$
1	0	0	1	1	0	
1	0	1	1	1	0	
1	1	0	0	0	1	$p \wedge q \wedge \neg r$
1	1	1	1	1	1	

La `fnd` que l'on trouve est donc $((\neg p \wedge \neg q \wedge \neg r) \vee (\neg p \wedge \neg q \wedge r) \vee (\neg p \wedge q \wedge r) \vee (p \wedge q \wedge \neg r))$. Elle semble à priori différente de la précédante. Nous allons voir que ces deux formes sont équivalentes : à partir des lignes 1 et 2 on peut construire la formule $(\neg p \wedge \neg q)$, à partir des lignes 2 et 4 on peut construire la formule $(\neg p \wedge r)$, la formule 7 restant isolée. La disjonction des interprétations nous redonne la forme normale trouvée précédemment.
On aurait pu aussi construire l'arbre sémantique, en utilisant l'algorithme de Quine vu dans la section 3.5.2, et construire la `fnd` en prenant la disjonction des interprétations qui valent 1 au niveau des feuilles de l'arbre. ▹

Exercice 3.7
Comment construire, à moindre coût, une `fnc` à partir d'une `fnd` (et récirpoquement) ? ▹

3.5.5 Foncteur booléen associé à une formule

L'approche de la logique du point de vue de fonctions booléennes est fondamentale pour la modélisation de circuits logiques. Nous ne développerons dans cette partie que les bases de l'approche, sans aborder, par exemple les notions de treillis, de tableaux de Karnaugh ou de simplification de fonctions booléennes. Les lecteurs intéressés sont renvoyés à [48, 5].

On se place ici dans le cadre où on a un nombre fini de variables propositionnelles. à une formule F portant sur n variables propositionnelles, on peut associer un foncteur booléen $f_F : \{0,1\}^n \longrightarrow \{0,1\}$. On dira que deux formules F et G sont équivalentes si les foncteurs f_F et f_G sont égaux. On dispose alors du théorème suivant :

Théorème 3.2
Soit $\mathcal{E} = \{p_1, p_2, \ldots, p_n\}$ un ensemble fini de variables propositionnelles. Toute fonction f de $\{0,1\}^n \longrightarrow \{0,1\}$ peut être représentée par une formule booléenne F portant sur $p_1, \ldots, p_n$ de telle sorte que pour toute interprétation I on ait $f(I) = I[\![F]\!]$.

La preuve de ce théorème s'établit par récurrence sur n.

preuve :

- $n = 1$. Il existe 4 fonctions possibles représentées dans la table suivante :

p	f_1	f_2	f_3	f_4
0	0	0	1	1
1	0	1	0	1
	$(p \wedge \neg p)$	p	$\neg p$	$(p \vee \neg p)$

- supposons la propriété vraie jusqu'au rang $n-1$, et considérons $\mathcal{E} = \{p_1, p_2, \ldots, p_n\}$. à toute interprétation I sur $\{p_1, \ldots, p_{n-1}\}$ on peut faire correspondre $\bar{\mathrm{I}}$ une interprétation sur $\mathcal{E}$. Notons f_0 et f_1 les restricteurs de f au cas $\bar{\mathrm{I}}[\![p_n]\!] = 0$ et $\bar{\mathrm{I}}[\![p_n]\!] = 1$. Ainsi si f correspond à une formule $F(p_1, \ldots, p_n)$, f_0 à $G(p_1, \ldots, p_{n-1})$ et f_1 à $H(p_1, \ldots, p_{n-1})$, on obtient que $F = ((\neg p_n \wedge G) \vee (p_n \wedge H))$

□

Afin de clarifier notre propos, considérons l'exemple ci-dessous

Exemple 3.5.4
Considérons la formule $F(p_1, p_2, p_3)$ qui vaut 1, lorsqu'il y a exactement deux variables propositionnelles sur trois à 1, sa table de vérité est la suivante :

p_1	p_2	p_3	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

La fomule $G(p_1, p_2)$ correspond aux lignes ayant un 0 dans la colonne de p_3, la formule $H(p_1, p_2)$ aux lignes ayant un 1 dans la colonne de p_3. On voit que $G(p_1, p_2) = (p_1 \wedge p_2)$ et $H(p_1, p_2) = ((\neg p_1 \wedge p_2) \vee (p_1 \wedge \neg p_2))$. La formule $F(p_1, p_2, p_3) = ((\neg p_3 \wedge G(p_1, p_2)) \vee (p_3 \wedge H(p_1, P_2)))$. ◁

Corrolaire 3.2.1 Toute formule F peut se ramener à une `fnc` ou à une `fnd`.

La preuve découle directement du théorème 3.2 et des propriétés d'associativité, de commutativité et de distributivité sur les connecteurs `et` et `ou`.

3.6 Formes clausales

Nous allons dans cette partie voir une nouvelle représentation de formules booléennes appelée forme clausale. Puis nous étudierons deux méthodes de résolution, la première est basée sur la syntaxe (Robinson 1965), la seconde sur la sémantique (Davis et Putnam 1960).

Définition 3.14
Une clause est une disjonction d'un nombre fini de littéraux, que l'on écrira sous forme ensembliste. Il n'existe qu'une clause inconsistante qui sera notée dans la suite $\{\}$ ou $\square$. Un ensemble de clauses est une conjonction de clauses.

Remarque : *Toute formule pouvant se mettre en `fnc` il est facile de la mettre sous forme clausale. Il ne s'agit en fait que d'une autre forme de présentation.* •

Définition 3.15
Un modèle pour un ensemble de clauses est un modèle pour chacune des clauses constituantes.
Un ensemble de clauses est dit *cohérent* s'il admet un modèle.
Un ensemble de clauses est dit *incohérent* s'il n'admet pas de modèle.

La résolution à partir de formes clausales considère un ensemble de clauses C et une formule F et a pour but de déterminer si $C \models F$, ou en d'autre terme si $C \cup \{\neg F\}$ est inconsistant.

3.6.1 Simplification dans les clauses

Avant d'aborder les méthodes de résolution de formules sous forme clausale, il est bon de savoir nettoyer ces expressions. Pour cela on applique les règles suivantes.

Proposition 3.6.1
1. Si une clause contient un littéral et son opposé, on l'enlève.
2. Dans une clause, on supprime les multiples occurences d'un littéral.
3. Si une clause c_i contient une clause c_j, on supprime c_i. On dit que c_j *subsume* c_i.

Il nous faut maintenant établir que ces règles sont valides, pour cela nous allons démontrer que si I est un modèle pour un ensemble de clauses, c'est aussi un modèle pour l'ensemble réduit. Il faudra aussi démontrer que tout modèle de l'ensemble de clauses réduit, est un modèle de l'ensemble initial.

preuve : Soit $C = \{c_1, \dots c_k\}$ un ensemble de clauses, une clause $c_i \in C$ est de la forme $\{l_{1_i}, \dots l_{k_i}\}$. Un modèle pour un ensemble de clauses est un modèle pour chacune des clauses constituantes. Par conséquent, un modèle d'un ensemble de clauses C_0 est un modèle pour tout sous-ensemble de C_0.

1. Une clause contenant un littéral et son opposé est une tautologie (vraie quelque soit l'interprétation choisie).
2. Une clause étant un disjonction de littéraux, et le connecteur $\vee$ étant absorbant, supprimer au sein d'une clause les occurences multiples d'un littéral n'influe pas sur le modèle.
3. Soient $c_i, c_j \in C$ telles que $c_j \subseteq c_i$, et soit I un modèle de c_j cela signifie qu'il existe $q_j \in \{1_j, \dots k_j\}$ tel que $\mathrm{I}[\![l_{q_j}]\!] = 1$, comme $l_{q_j} \in c_i$, I est un modèle de c_i.

Ces 3 points démontrent que tout modèle de l'ensemble réduit est un modèle de l'ensemble initial. □

Exemple 3.6.1

Trouver la forme clausale simplifiée de la formule $\neg(p \equiv (q \supset r))$. *Après simplification on obtient :*

$((p \wedge (q \wedge \neg r)) \vee ((\neg q \vee r) \wedge \neg p))$

$((p \vee ((\neg q \vee r) \wedge \neg p)) \wedge ((q \wedge \neg r) \vee ((\neg q \vee r) \wedge \neg p)))$

$((p \vee \neg q \vee r) \wedge (p \vee \neg p) \wedge (((q \wedge \neg r) \vee (\neg q \vee r)) \wedge ((q \wedge \neg r) \vee \neg p)))$

$((p \vee \neg q \vee r) \wedge (p \vee \neg p) \wedge (q \vee \neg q \vee r) \wedge (\neg r \vee \neg q \vee r) \wedge (q \vee \neg p) \wedge (\neg r \vee \neg p))$

Ce qui donne l'ensemble de clauses suivant :

$C = \{\{p, \neg q, r\}, \{p, \neg p\}, \{q, \neg q, r\}, \{\neg r, q, r\}, \{q, \neg p\}, \{\neg r, \neg p\}\}$

Les clauses 2, 3 et 4 sont des tautologies ; l'ensemble final réduit est donc :

$$\{\{p, \neg q, r\}, \{q, \neg p\}, \{\neg r, \neg p\}\}$$

◁

3.6.2 Méthode de résolution de Robinson

Définition 3.16

Soient deux clauses c_1 et c_2 et soit l un littéral tel que $l \in c_1$ et $\neg l \in c_2$. La résolvante est obtenue en ajoutant les littéraux de c_1 et de c_2 et en faisant disparaître les occurences (positive et négative) de l, plus formellement $r = (c_1 \backslash \{l\}) \cup (c_2 \backslash \{\neg l\})$.

La propriété suivante démontre que chercher un modèle pour deux clauses c_1 et c_2, c'est chercher un modèle pour r.

Proposition 3.6.2 Soient deux clauses $c_1 = \{l_{1_1}, \dots, l_{k_1}\}$ et $c_2 = \{l_{1_2}, \dots, l_{k_2}\}$ supposons (sans perte de généralité) que $l_{1_1} = \neg l_{1_2}$. Alors $\{c_1, c_2\} \models \{l_{2_1}, \dots, l_{k_1}, l_{2_2}, \dots, l_{k_2}\}$

preuve : Soit I un modèle de $\{c_1, c_2\}$, il y a deux cas à considérer. Soit $\mathrm{I}[\![l_{1_1}]\!] = 1$, donc $\mathrm{I}[\![l_{1_2}]\!] = 0$, comme I est un modèle de c_2, il existe un $i \in 2_2..k_2$ tel que $\mathrm{I}[\![l_i]\!] = 1$ d'où I est un modèle de r. Le deuxième cas, symmétrique du premier est $\mathrm{I}[\![l_{1_1}]\!] = 0$, donc $\mathrm{I}[\![l_{1_2}]\!] = 1$; par le même type de raisonnement, on aboutit à la même conclusion. Donc on a établi la proposition. □

Exemple 3.6.2

Considérons 4 clauses $c_1 = \{p, q, r\}, c_2 = \{\neg p, q\}, c_3 = \{\neg q, r\}, c_4 = \{\neg r, p\}$. *à partir de* c_1 *et* c_2, *on peut construire* $c_{12} = \{q, r\}$ *en combinant ensuite avec* c_3 *on obtient* $c_{123} = \{r\}$. *Si on combine ensuite avec* c_4 *on aboutit à* $c_{1234} = \{p\}$. *Enfin en combinant de nouveau avec* c_2 *on trouve* $c_{12342} = \{q\}$. *Ce qui revient à dire que* $\{c_1, c_2, c_3, c_4\} \models (p \wedge (q \wedge r))$ *ou que* $\{c_1, c_2, c_3, c_4\} \cup \{\{\neg p, \neg q, \neg r\}\}$ *est incohérent*. ◁

On constate que si le principe de cette résolution est simple, il n'en demeure pas moins qu'elle est longue et fastidieuse, car elle nécessite de combiner toutes les clauses. Néanmoins le processus peut être amélioré, en évitant la création de tautologies, en suivant une résolution linéaire, ou en restreignant le type de clauses (clauses de Horn[9], par exemple). La résolution sera dite linéaire si la résolvante construite à la $k+1$ème étape utilise la résolvante trouvée à l'étape k. Certaines de ces améliorations ont permis la mise en place du langage PROLOG (voir les chapitres 4 et 8).

3.6.3 Méthode de résolution sémantique

Dans ce cadre, établir la cohérence d'un ensemble de clauses, revient à chercher un modèle, c'est-à-dire assigner une valeur de vérité à chaque variable propositionnelle. Ce principe d'affectation est un problème que l'on peut présenter sous forme récursive :
(1) choisir un littéral
(2) lui assigner la valeur `vrai`
(3) à partir de là 3 situations sont envisageables :

1. L'affectation rend vraie toutes les clauses. La résolution se termine, un modèle a été trouvé.
2. L'affectation provoque une incohérence. Une clause ne contient que des littéraux affectés à `faux`. Il faut soit changer l'affectation, soit revenir sur une affectation antérieure. Lorsque toutes les alternatives ont été épuisées l'ensemble de clauses est dit incohérent.
3. Il n'y a pas d'incohérence et certaines clauses ne sont pas encore vraies. Le processus d'affectation doit continuer.

Remarque : *Cette méthode revient grosso modo à la méthode élaborée par Quine sur les arbres émantiques (voir section 3.5.2).* •

Considérons l'exemple suivant :

EXEMPLE 3.6.3
Soit $C = \{\{\neg p\},\{p,q\},\{p,r\},\{p,\neg s\}\}$ et la formule $F = ((p_1 \wedge \neg p) \vee \neg p_1)$. On construit le nouvel ensemble de clauses $C \cup \{\neg F\} = C \cup \{\{\neg p_1, p\},\{p_1\}\} = C'$.
On pose $I[\![p_1]\!] = 1$, cette affectation ne pose pas de problème pour C, pour que C' admette un modèle il faut que $I[\![p]\!] = 1$ et que C admette un modèle. Ce n'est pas le cas à cause de la clause $\{\neg p\}$. Il faut revenir sur un choix antérieur, si $I[\![p]\!] = 0$ alors C' est incohérent. Il faut revenir sur l'affectation de p_1 et choisir $I[\![p_1]\!] = 0$, ce qui rend C' incohérent. Donc on établit que $C \models F$. ◁

Nous allons maintenant étudier l'algorithme proposé par Davis et Putnam (1960) et proposer quelques heuristiques (astuces ne modifiant pas la validité du processus mais améliorant sa rapidité).

3.6.4 Algorithme de Davis et Putnam

Cette partie est directement inspirée de la thèse de doctorat d'A. Rauzy [78].

DÉFINITION 3.17
Soit C un ensemble de clauses réduit. On note $T_{l_1,l_2,\ldots,l_n}(C)$ l'ensemble de clauses obtenu à partir de C en supprimant les clauses contenant l_i ou les occurences opposées de l_i, pour tout $i \in 1..n$.

EXEMPLE 3.6.4
Soit $C_0 = \{\{a,b,c\},\{\neg a,b\},\{\neg b,c\},\{\neg c,a\}\}$.
$T_a(C_0) = \{\{b\},\{\neg b,c\}\}$; $T_{abc}(C_0) = \emptyset$; $T_{\neg a,b}(C_0) = \{\{c\}\}$. ◁

Remarque : *L'ordre des opérations n'influe pas sur le résultat, on a les égalités suivantes :*

$$T_{abc}(C) = T_a(T_{bc}(C)) = T_{bc}(T_a(C))$$

•

[9] Clause de Horn : clause ayant au plus un littéral positif.

PROPRIÉTÉ 3.6.3 Soit C un ensemble de clauses réduit. Un modèle de $C \cup \{l_1\} \cup \{l_2\} \cup \{l_n\}$ est un modèle de $T_{l_1,l_2,...,l_n}(C)$.

preuve : Soit I un modèle de $C \cup \{l_1\} \cup \{l_2\} \cup \{l_n\}$. Par définition, I est un modèle de C et vérifie $I[\![l_1]\!] = I[\![l_2]\!] = \ldots = I[\![l_n]\!] = 1$. I rend vraie toutes les clauses de C qui contienne un l_i (par subsumtion entre autre), dans toutes les autres clauses de C, on peut supprimer les occurences négatives des l_i, les clauses résiduelles appartiennent à T (par construction). □

EXEMPLE 3.6.5
Soit $C_0 = \{\{a, b, c\}, \{\neg a, b\}, \{\neg b, c\}, \{\neg c, a\}\}$, et considérons $T_{\neg a,b}(C_0) = \{\{c\}\}$. Soit I telle que $I[\![\neg a]\!] = I[\![b]\!] = I[\![c]\!] = 1$ alors I est un modèle de C_0. ◁

PROPRIÉTÉ 3.6.4 Soit C un ensemble de clauses réduit, et soient l_1, l_2, $\ldots l_n$ des littéraux distincts deux à deux et tels que $\nexists i \neq j$ avec l_i et l_j opposé. Alors l'ensemble $T_{l_1,l_2,...,l_n}(C)$ est équivalent à $T^*_{l_1,l_2,...,l_n}(C)$ obtenu par subsumption.

PROPRIÉTÉ 3.6.5 Soit C un ensemble de clauses réduit, et soient l_1, l_2, $\ldots l_n$ des littéraux distincts deux à deux et tels que $\nexists i \neq j$ avec l_i et l_j opposé. Alors $T^*_{l_1,l_2,...,l_n}(C)$ et $C \cup \{l_1\} \cup \{l_2\} \cup \{l_n\}$ ont un même modèle.

preuve : T^* et T admettent le même modèle. □

PROPRIÉTÉ 3.6.6 Soit C un ensemble de clauses et soit p une variable propositionnelle apparaissant dans C. C est cohérent si et seulement si $T^*_p(C)$ ou $T^*_{\neg p}(C)$ est cohérent.

preuve : Puisque p est une variable apparaissant dans C, un modèle de C est une extension d'un modèle qui rend p ou $\neg p$ vrai. Donc soit $C \cup \{p\}$ est cohérent, soit $C \cup \{\neg p\}$ est cohérent. On peut alors établir (grâce à la propriété 3.6.5) que soit $T^*_p(C)$ est cohérent, soit $T^*_{\neg p}(C)$ est cohérent. La réciproque s'établissant par la même propriété. □

On a donc un moyen constructif pour établir la cohérence d'un ensemble donné de clauses, c'est ce moyen qui est l'algorithme de Davis et Putnam.

```
Soit  C un ensemble de clauses réduit
Si  C = ∅ alors  C est cohérent
Sinon Si  C contient □ alors  C est incohérent
      Sinon
        on choisit une variable propositionnelle p et
        Si T*p( C) et T*¬ p( C) sont incohérents
        Alors  C est incohérent.
        Fsi
      Fsi
Fsi
```

EXEMPLE 3.6.6
Soit l'ensemble de clauses $C_0 = \{\{p, q\}, \{p, r\}, \{\neg p, q\}, \{\neg q\}\}$.
Choisissons la variable p, on construit alors :
$T^*_p(C_0) = \{\{q\}, \{\neg q\}\}$; $T^*_{\neg p}(C_0) = \{\{q\}, \{r\}, \{\neg q\}\}$
Choisissons la variable q, on construit alors :
$T^*_{p,q} = \{\square\}$; $T^*_{p,\neg q} = \{\square\}$ qui sont tous les deux incohérents, donc T^*_p est incohérent.
$T^*_{\neg p,q} = \{\{r\}, \square\}$; $T^*_{\neg p,q} = \{\square, \{r\}\}$ qui sont tous les deux incohérents donc $T^*_{\neg p}$ est incohérent.
Donc C_0 est incohérent. ◁

Un exemple de codage d'un ensemble de clauses

Il est possible de représenter un ensemble de clauses par le biais d'une matrice creuse, dont les lignes correspondent aux clauses et les colonnes aux littéraux.

Exemple 3.6.7

Regardons le codage de l'ensemble $\{\{p, q, \neg r\}, \{q, s\}, \{\neg p, q\}, \{\neg r, s\}, \{p, \neg r\}\}$.

$$\begin{pmatrix} p & q & \neg r & \\ & q & & s \\ \neg p & q & & \\ & & \neg r & s \\ p & & \neg r & \end{pmatrix}$$

◁

Par rapport à l'algorithme, sélectionner une variable, revient à sélectionner une colonne. Choisir un littéral c'est mettre un 1 à sa place et un 0 à la place de sa forme négative. Simplifier une clause c'est supprimer toutes les lignes où il y a un 1.

Exemple 3.6.8

En continuant l'exemple 3.6.7, et en supposant que l'on ait choisi le littéral p *on obtient :*

$$\begin{pmatrix} 1 & q & \neg r & \\ & q & & s \\ 0 & q & & \\ & & \neg r & s \\ 1 & & \neg r & \end{pmatrix}$$

Ce qui donne après simplification :

$$\begin{pmatrix} q & & s \\ q & & \\ & \neg r & s \end{pmatrix}$$

En prenant le littéral q*, après simplification on obtient :*

$$\begin{pmatrix} \neg r & s \end{pmatrix}$$

Si on choisit ensuite le littéral $\neg r$*, le processus s'arrête (suppression de toutes les lignes), si par contre on choisit le littéral* r*, on se retrouve avec une matrice 1x1 où l'on est forcé de prendre le littéral* s*, sous peine d'incohérence.* ◁

Quelques heuristiques

Parmi les heuristiques possibles :

1. Privilégier le choix des variables propositionnelles correspondant à un littéral isolé dans une clause.
2. Choisir les littéraux apparaissant dans les clauses les plus courtes.
3. Choisir les littéraux qui n'apparaissent que positivement (resp. négativement), et parmi ceux-ci ceux qui sont présents le plus souvent.

à titre d'exemple, prenons l'ensemble de clauses $\{\{p, q, \neg r\}, \{q, s\}, \{\neg p, q\}, \{\neg r, s\}, \{p, \neg r\}\}$. Sans heuristique, on choisit les littéraux par ordre alphabétique.
Choisir p : $\mathrm{T}^*_p = \{\{q\}, \{\neg r, s\}\}$
Choisir q : $\mathrm{T}^*_{p,q} = \{\{\neg r, s\}\}$
Choisir r : $\mathrm{T}^*_{p,q,r} = \{\{s\}\}$
Choisir s : $\mathrm{T}^*_{p,q,r,s} = \emptyset$
On a donc trouver un modèle de l'ensemble initial, pour trouver tous les modèles, il faut continuer le processus.

Si par contre on applique les heuristiques.
Choisir q : $\mathrm{T}^*_q = \{\{\neg r, s\}, \{p, \neg r\}\}$

Choisir $\neg r$: $T^*_{q,\neg r} = \emptyset$
On a donc trouver un modèle de l'ensemble initial, bien entendu si l'on cherche tous les modèles le processus doit se poursuivre, mais le calcul demeure plus rapide.

3.7 Quelques erreurs à éviter

Lors de vos études vous avez manipulé la notion de *contraposée* et de *réciproque* de formule où le connecteur principal était l'implication $\supset$. Soit $p \supset q$ une implication, la contraposée de $p \supset q$ est $\neg q \supset \neg p$, la réciproque étant $q \supset p$. La contraposée véhicule la même information que l'implication initiale, ce qui n'est pas le cas de la réciproque. Pour cela, nous allons vérifier, grâce aux tables de vérité les différences et ressemblance entre les trois formules :

p	q	$p \supset q$	$\neg q \supset \neg p$	$q \supset p$
0	0	1	1	1
0	1	1	1	0
1	0	0	0	1
1	1	1	1	1

$$\begin{array}{lcl} p \supset q & = & \neg p \vee q \\ \neg q \supset \neg p & = & \neg\neg q \vee \neg p \\ q \supset p & = & \neg q \vee p \end{array}$$

De la formule $(\neg q \supset \neg p) \wedge q$ on ne peut pas déduire p. De même si $p \supset q$ est faux, on ne peut pas en déduire $q \supset p$.

3.8 Séquent

Un séquent est un couple $(\mathcal{F}, F)$, il est vrai pour une interprétation I si $\forall G \in \mathcal{F}, \mathrm{I}[\![G]\!] = 1$ implique $\mathrm{I}[\![F]\!] = 1$.
Un séquent est *valide* s'il est vrai pour toute interprétation I, dans ce cas on écrit $\mathcal{F} \models F$.
Un séquent est *prouvable*, on le note $\mathcal{F} \vdash F$, s'il est construit en utilisant un nombre fini de fois les règles suivantes :

R-1 Utilisation d'une hypothèse : si $F \in \mathcal{F}$ alors $\mathcal{F} \vdash F$.

R-2 Augmentation d'une hypothèse : si $G \notin \mathcal{F}$ et $\mathcal{F} \vdash F$ alors $\mathcal{F} \cup \{G\} \vdash F$.

R-3 Détachement ou *modus ponens* : si $\mathcal{F} \vdash F \supset F'$ et $\mathcal{F} \vdash F$ alors $\mathcal{F} \vdash F'$.

R-4 Synthèse : si $\mathcal{F}, G \vdash F$ alors $\mathcal{F} \vdash G \supset F$.

R-5 Double négation : $\mathcal{F} \vdash F$ si et seulement si $\mathcal{F} \vdash \neg\neg F$.

R-6 Raisonnement par l'absurde : si $\mathcal{F}, F \vdash F'$ et $\mathcal{F}, F \vdash \neg F'$ alors $\mathcal{F} \vdash \neg F$.

Un démonstration d'un séquent prouvable $\mathcal{F} \vdash F$ est une suite finie de séquents prouvables de la forme $\mathcal{F}_i \vdash F_i$ dont le dernier est $\mathcal{F} \vdash F$ telles que tout séquent est obtenu à partir de ceux qui le précèdent. Le premier séquent est donc nécessairement une règle d'utilisation d'une hypothèse.

Exemple 3.8.1
Soit à démontrer $\emptyset \vdash p \supset (p \supset p)$. *La démonstration pourra se faire de la façon suivante :*

$p \vdash p$	*R-1*
$\emptyset \vdash p \supset p$	*R-4*
$p \vdash p \supset p$	*R-2*
$\emptyset \vdash p \supset (p \supset p)$	*R-4*

◁

Exercice 3.8
Prouver $p \vdash \neg p \supset q$. ▷

3.9 Théorie de la démonstration

Comme nous l'avons présenté succinctement dans la setion 3.3, la théorie de la démonstration est purement syntaxique, elle repose sur l'utilisation de règles mécaniques pour aboutir à la démonstration.

Définition 3.18
Une règle d'inférence se présente sous la forme :

$$\frac{j_1 j_2 \dots j_n}{j}$$

Dans laquelle $j_1, \dots, j_n$ sont des jugements qui forment les *prémisses* de la règle, j est un jugement qui est la *conclusion* de la règle.

Une telle règle exprime que l'on peut *inférer* la conclusion j si on déjà été capable de produire les prémisses $j_1, \dots, j_n$. Dans le cas où $n = 0$, la règle est appelé un *axiome* et indique que la conclusion peut être produite directement. Si les jugements concernent des formules, il sera possible de produire de nouvelles formules en ne s'appuyant que sur les axiomes et sur les règles d'inférence. Ces formules seront appelées *théorèmes* et pourront à leur tour être utilisés pour produire de nouveaux théorèmes. La théorie de la démonstration repose donc sur un système axiomatique et sur des règles d'inférence, on parle aussi de système formel. Il est bien entendu souhaitable que ces systèmes axiomatiques soient en bonne correspondance avec une interprétation sémantique.

3.9.1 Système axiomatique pour le calcul propositionnel

Définition 3.19
Une formule F est un théorème, noté $\vdash F$, si F est un axiome ou bien si F est obtenue à partir de l'application d'une règle d'inférence sur d'autres théorèmes.

On utilise trois schémas d'axiomes et une règle d'inférence pour formaliser le calcul propositionnel :

A-1 $F \supset (G \supset F)$

A-2 $(F \supset (G \supset H)) \supset ((F \supset G) \supset (F \supset H))$

A-3 $(\neg G \supset \neg F) \supset (F \supset G)$

Cet ensemble est le plus petit ensemble (au sens de la cardinalité) que l'on puisse utiliser.
La règle d'inférence est la règle du *modus-ponens* que l'on peut exprimer sous la forme :

$$\frac{\vdash F, \vdash F \supset G}{\vdash G}$$

3.9.2 Démonstration

Définition 3.20
Une *démonstration* d'un théorème F est une suite finie $(F_1, F_2, \dots, F_n, F)$ où chaque F_i est soit un axiome, soit un théorème obtenu par l'application d'une règle d'inférence sur les théorèmes F_j, F_k avec $j, k < i$. On écrit $\vdash F$ pour indiquer que F est un théorème.

Comme on le voit, cette définition corespond à la notion intuitive de démonstration c'est-à-dire un enchaînement fini d'inférences formelles.

3.10 Propriétés

Nous allons, dans cette partie, montrer que dans le cadre du calcul propotionnel les deux théories (modèles et démonstration) sont équivalentes. Cette adéquation repose sur un certain nombre de propriétés que nous allons rapidement survoler.

3.10.1 Adéquation

Étant donnés, un langage et une sémantique, on s'intéresse à la relation entre preuve et modèle.

Définition 3.21 (Adéquation)
Une logique est dite *fondée* ou *adéquate*[10] si tout théorème $\vdash F$ est une formule valide $\models F$.

On peut démontrer par induction que le calcul propositionnel est adéquat.

3.10.2 Consistance

Cette notion indique que si l'on peut démontrer un théorème, on ne peut pas démontrer sa négation. Il s'agit d'une conséquence directe de la notion d'adéquation.

Définition 3.22 (Consistance)
Une logique est dite consistante (syntaxiquement) s'il n'existe aucune formule F du langage telle que l'on puisse avoir $\vdash F$ et $\vdash \neg F$.

On peut démontrer que le calcul propositionnel est consistant.

3.10.3 Complétudes

On distingue deux types de complétude liant syntaxe et sémantique. La complétude faible qui garantit que les règles suffisent pour démontrer toutes les formules valides ; la complétude forte exprime que la relation de conséquence valide correspond à la notion de dérivabilité syntaxique. Il existe une troisième notion de complétude, la complétude syntaxique qui exprime que pour toute formule logique F il est possible de démontrer $\vdash F$ ou $\vdash \neg F$.

Plus formellement on a :

Définition 3.23 (Complétude forte)
Une logique est dite *fortement complète* si la relation de conséquence valide correspond à celle de dérivabilité : Soit $\mathcal{E}$ un ensemble de formules, si $\mathcal{E} \models F$ alors $\mathcal{E} \vdash F$.

Définition 3.24 (Complétude faible)
Une logique est dite *faiblement complète* si toute formule valide est un théorème : Si $\models F$ alors $\vdash F$.

Définition 3.25 (Complétude syntaxique)
Une logique est dite *syntaxiquement complète* si pour toute formule F on a soit $\vdash F$ soit $\vdash \neg F$.

3.10.4 Décidabilité

Le problème de la décidabilité est une question qui touche tous les systèmes formels, elle repose d'une part sur une notion mécanisable du calcul de la réponse à une question et d'autre part sur la finitude du processus computationnel. Plus formellement :

Définition 3.26 (Décidabilité)
On dit qu'une logique est *décidable* quand on connaît une procédure mécanique permettant d'établir en un temps fini si une formule F donnée est, ou n'est pas un théorème.

Cette notion sera reprise dans le cadre général des systèmes formels et de la notion de calculabilité.

[10] Les anglophones utilisent le terme de « sound ».

3.11 Exercices

Dans cette partie, je vous propose quelques exercices mettant en pratique les divers concepts que nous avons abordés au cours de ce chapitre.

Exercice 3.9
Que pensez-vous des syllogismes suivants :

1. (M) « S'il pleut Noé boit », (m) « Il pleut », (c) « Noé boit ».
2. (M) « Tous les moineaux sont mortels », (m) « Tous les pigeons sont mortels », (c) « Tous les oiseaux sont mortels ».
3. (M) « Tous les A sont B », (m) « Tous les B sont C », (c) « Tous les A sont C ».
4. (M) « Aucun A n'est B », (m) « Quelques C sont B », (c) « Aucun A n'est C ».
5. (M) « Tous les banquiers sont des athlètes », (m) « Aucun conseiller n'est banquier », (c) « Quelques athlètes ne sont pas conseiller ».

▹

Exercice 3.10

- Montrer que la formule $((p \supset q) \supset r)$ n'est pas identique à la formule $(p \supset (q \supset r))$. Trouver un exemple en français.
- Montrer que les formules $((p \equiv q) \equiv r)$ et $(p \equiv (q \equiv r))$ sont identiques.

▹

Exercice 3.11
Soit p la proposition « il pleut », q la proposition « il y a des nuages ». Écrire en français $p \supset q$, sa contraposée, sa réciproque, la contraposée de la réciproque. Quelles implications sont vraies ? ▹

Exercice 3.12
Quelle est votre conclusion pour les quatre situations suivantes :

1. $(((p \supset q) \wedge p) \supset ?)$
2. $(((p \supset q) \wedge q) \supset ?)$
3. $(((p \supset q) \wedge \neg p) \supset ?)$
4. $(((p \supset q) \wedge \neg q) \supset ?)$

▹

Exercice 3.13
Tiré de B. Ruyer [81] :
Arthur, Barnabé et Casimir sont soupçonnés d'avoir peint en vert le chat de la voisine. Ils font les déclarations suivantes :

Arthur : Barnabé est le coupable, Casimir est innocent ;

Barnabé : Si Arthur est coupable, Casimir aussi ;

Casimir : Je suis innocent, mais au moins l'un des deux autres est coupable.

Transcrire les trois déclarations dans le langage de la logique en posant :
a = « Arthur est coupable », b = « Barnabé est coupable », c = « Casimir est coupable ». Puis, à l'aide d'une table de vérité, répondre aux questions suivantes :

1. Si Casimir a menti, que dire de la déclaration d'Arthur.

2. Si Casimir a menti, que dire de la déclaration de Barnabé.
3. En supposant que tous ont dit la vérité, qui est coupable, qui est innocent.
4. En supposant que tous sont coupables, qui a menti, qui a dit la vérité.
5. En supposant que tout innocent dit la vérité et que tout coupable ment, qui est innocent, qui est coupable.
6. Que répondre à la question : « En supposant que les innocents ont menti et que les coupables ont dit la vérité, qui est innocent, qui est coupable ».

▹

Exercice 3.14

Tiré de R. Smullyan [84]

Pour vider ses prisons, un roi impose aux prisonniers de choisir entre deux cellules. Sur chacune d'entre elles une pancarte est apposée sur laquelle on peut lire :

cellule 1 Une cellule au moins contient une princesse

cellule 2 L'autre cellule contient une princesse

Par ailleurs on sait que « l'inscription sur la cellule 1 est vraie si et seulement si elle contient une princesse »et que « l'inscription sur la cellule 2 et vraie si et seulement si elle contient un tigre ».

1. Peut-on avoir un tigre dans chaque cellule
2. Quelle est votre cellule de prédilection

On considèrera qu'on ne peut avoir dans une même cellule un tigre et une princesse. Résoudre ces questions à l'aide d'une table de vérité en posant : a = « un tigre dans la cellule 1 »et b = « un tigre dans la cellule 2 ». ▹

Exercice 3.15

Cinq amis (A)ndré, (B)arnabé, (C)asimir, (D)ésiré et (L)udovic sont au restaurant où il mange du bœuf. On cherche à déterminer, à partir des assertions ci-dessous, quel(s) condiment(s) ils ont choisi.

1. A prend du sel, ssi B ou L prend du sel et de la moutarde.
2. A prend de la moutarde, ssi C ou D prend un seul condiment.
3. B prend du sel, ssi C ne prend que du sel ou que de la moutarde.
4. B prend de la moutarde, ssi D ne prend ni sel ni moutarde ou A prend du sel et de la moutarde.
5. C prend du sel, ssi B ne prend qu'un des deux condiments ou A n'en prend aucun.
6. C prend de la moutarde, ssi D prend du sel et de la moutarde ou L prend du sel et de la moutarde.
7. D prend du sel, ssi B ne prend ni sel ni moutarde ou C prend du sel et de la moutarde.
8. D prend de la moutarde, ssi L ou A ne prennent aucun condiment.
9. L prend du sel, ssi B ou D ne prennent aucun condiment.
10. L prend de la moutarde, ssi C ou A ne prennent aucun condiment.

▹

Exercice 3.16

Trouver la forme clausale simplifiée de la formule $((p \supset (q \supset r)) \supset ((p \wedge s) \supset r))$. ▹

Exercice 3.17

En prenant la formule de l'exercice précédent appliquer les différentes techniques de résolution exposées au cours de ce chapitre. ▹

Exercice 3.18
Construire les tables de vérité des formules $p \supset (p \supset p)$ et $(p \supset p) \supset p$. Qu'en déduisez vous ? ▹

Exercice 3.19
Que pensez vous des assertions suivantes :

A-1 Pour être séduisant, il suffit de porter un nœud papillon ; Alfred n'est pas séduisant donc Alfred n'a pas de nœud papillon.

A-2 Pour réussir cet exercice, il faut garder la tête froide ; Vous ne perdez pas la tête donc Vous ne ratez pas l'exercice.

A-3 Si mon raisonnement est valide, je ne raterai pas cet exercice ; Ou bien je rate mon exercice, ou bien la logique n'est pas facile ; La logique n'est pas facile donc Mon raisonnement n'est pas valide.

A-4 Il faut et il suffit qu'il pleuve pour que l'escargot sorte ; l'escargot sort donc Il pleut.

▹

Exercice 3.20
Deux personnes se promène dans un édifice, elles arrivent dans un couloir où trois portes sur lesquelles une fleur de couleur est stylisée.
Une seule proposition est vraie :

P1-1 Appuyez sur la fleur jaune ou bleue

P1-2 Appuyez sur la fleur bleue ou rouge

P1-3 Appuyez sur la fleur jaune

Quelle porte choisissez-vous ?

Continuant leur promenade ils arrivent à un embranchement de trois couloirs.
Une seule proposition est fausse :

P2-1 Prenez le couloir de droite ou du milieu

P2-2 Ne prenez ni le couloir de gauche ni celui du milieu

P2-3 Ne prenez ni le couloir de droite, ni celui du milieu

Quel couloir choisissez-vous ?

Un petit peu plus loin ils trouvent quatre escaliers.

P3-1 L'escalier montant conduit à un cul de sac

P3-2 Deux de ces assertions sont fausses

P3-3 L'un des trois escaliers descendants mênent à l'étape suivante

P3-4 Au plus de deux ces assertions sont vraies

Justifiez pourquoi il faut prendre l'escalier montant.

Vous aboutissez à une salle où sont disposées quatre statuettes qui sont dans l'ordre « connaissance », « sagacité », « fécondité », « prospérité », il faut en sélectionner deux parmi quatre :

P4-1 Soit la connaissance, soit la prospérité ne doivent pas être choisie

P4-2 Soit la sagacité, soit la prospérité ne doivent pas être choisie

P4-3 Ne choisissez pas deux statuettes voisines

▹

EXERCICE 3.21

Alfred, 30 ans ; Baptiste, 27 ans ; Charly, 33 ans sont accusés d'avoir participé à plusieurs forfaits. On sait qu'ils sont les seuls à pouvoir être responsables, et on ne dispose que d'informations (vraies) provenant de sensitifs.

Bijouterie La peine encourue est de 6 ans, on dispose de quatre affirmations :

A-1 Si Charly est innocent, Alfred est coupable.

A-2 Si Alfred est coupable, il a agi avec un et un seul complice.

A-3 Si Baptiste n'a pas participé à l'attaque, Charly non plus.

A-4 S'il y a deux responsables, Alfred est l'un d'eux.

Banque La peine est de 7 ans, on a quatre affirmations :

B-1 Si Baptiste est coupable, Charly aussi.

B-2 Pour les hold-up de banques, Alfred ne fait jamais équipe avec Charly.

B-3 Si Alfred est coupable et Baptiste hors du coup alors Charly est coupable.

B-4 Charly n'a pas pu faire ce genre de boulot tout seul.

Pharmacie La peine est de 4 ans, trois affirmations :

C-1 Si Alfred est coupable, Baptiste est innocent.

C-2 Si Baptiste est coupable, il a un et un seul complice.

C-3 Si Charly est coupable, Alfred et Baptiste aussi

Q-1 Mettre chaque assertions en `fnc`

Q-2 En utilisant soit la méthode de Robinson, soit la méthode de Davis et Puttnam déterminer « qui est coupable de quoi ? »

Q-3 Quelle est la drôle de conséquence que l'on peut tirer ?

Cette enquête policière est tirée de J. & S. numéro 3. ▹

EXERCICE 3.22

Vous devez embarqué pour la planète QUOLGIE, mais pour vérifier votre aptitude à survivre sur cette planète vous devz subir un dernier test. Vous êtes accompagné d'un robot, R, qui dit toujours la vérité.

- La première étape de ce test est de localiser la salle de contrôle. R « Tout ce que je peux vous dire, c'est qu'au moins une indication est juste et au moins une est fausse. »

  P-1 Pas de contrôle porte P-2.

  P-2 Pas de contrôle ici.

  P-3 Contrôle ici.

- La seconde étape sera de localiser la salle d'enregistrements. R « Suivez les indications, il y en a au plus une d'exacte. »

  Q-1 Enregistrement ici.

  Q-2 Pas d'enregistrement ici.

  Q-3 Pas d'enregistrement porte Q-2.

- Troisième étape trouver le secteur de douane. R « Les tubes d'accès au secteur de douanes sont munis de portiers intégrés. Chacun d'entre eux ne peut proférer plus d'un mensonge. »

  T-1 Pas d'accès à la douane ici, l'inspection dure 10'.

  T-2 Pas d'accès à la douane par le tube T-1, l'inspection dure 2'.

  T-3 Pas d'accès aux douanes ici, accès par le tube T-2.

- Dernière étape, localiser la porte 7. R « Méfiez vous de l'une des portes en permanance, elle ment. Envers l'autre soyez confiants, elle ne dit que la vérité. Quant à la dernière au mensonge elle joint la vérité."

  A La porte 7 n'est pas ici, c'est la porte B.

  B La porte 7 n'est pas ici, c'est la porte C.

  C La porte 7 ce n'est pas la porte B, c'est la porte A.

Cet exercice est tiré de J. & S. numéro 4. ▹

3.12 BDD, un codage efficace de formules booléennes

La présentation faite ici des BDD est inspirée de [29], dont certains paragraphes sont repris tels quels.

Un BDD (Binary Decision Diagram) qui en français se dira Diagrammes Binaires de Décision, est un codage efficace de la table de vérité d'une formule booléenne. Ce codage est canonique modulo un ordre sur les variables de la formule et même si la taille du BDD associé à une formule est exponentielle, dans le pire cas, elle est, pour un certain nombre de formules assez réduite. On peut, par ailleurs, effectuer toutes les opérations logiques usuelles directement sur les BDD – le coût de cette opération est, dans le pire cas, le produits des tailles des BDD utilisés. Un des intérêts majeurs de cette représentation étant de rendre triviaux le test de satisfaisabilité et le problème de l'équivalence de deux formules. Bryant, dans [15], présente un état de l'art sur cette structure de données ainsi que de nombreuses applications.

On a vu qu'une formule F pouvait se décomposer en $(x \wedge F_{x=1}) \vee (\neg x \wedge F_{x=0})$ (voir la section 3.5.5 page 48), ce qui en introduisant l'opérateur si-alors-sinon dont la table de vérité est donnée figure 3.2 sera noté

$$\texttt{ite}(x, F_{x=1}, F_{x=0})$$

3.12.1 Représentation graphique

Soit la formule $(a \vee b) \wedge (c \vee d)$ et l'ordre lexicographique sur les variables la figure 3.5 donne l'arbre syntaxique complet, tandis que la figure 3.6 donne le codage sous forme BDD de la même formule.

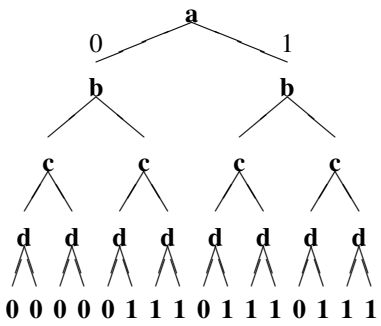


FIGURE 3.5 – Arbre Syntaxique associé à $(a \vee b) \wedge (c \vee d)$

3.12.2 Opérations sur les BDD

DÉFINITION 3.27

Regardons comment on peut manipuler directement les BDD avec des connecteurs logiques :

- Soit A = ite(x,A_1,A_2) et B = ite(y,B_1,B_2) deux formules booléennes données sous forme de BDD, et soit $\bowtie \in \{\wedge, \vee, \supset, \equiv\}$.

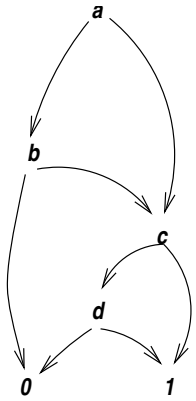


FIGURE 3.6 – BDD associé à $(a \vee b) \wedge (c \vee d)$

1. Si x = y, alors (A $\bowtie$ B) = ite(x,$A_1 \bowtie B_1$, $A_2 \bowtie B_2$).
2. Sinon, on peut supposer, sans perte de généralité que x < y, c'est-à-dire, que x n'apparait pas dans une sous-formule de B. Dans ce cas (A $\bowtie$ B) = ite(x,$A_1 \bowtie$ B,$A_2 \bowtie$ B).

- Soit A = ite(x,A_1,A_2), $\neg$ A $\neq$ ite(x,A_2,A_1)

EXERCICE 3.23

1. Établir que si une formule F = `ite`(x,A,B) alors $\neg$F = `ite`(x,$\neg$A,$\neg$B)
2. Étant donné un ensemble de variable $V = \{v_1, v_2, v_3, v_4, v_5, v_6\}$, on cherche à construire le BDD associé à la formule

$$F = (v_1 \wedge v_2) \vee (v_3 \wedge v_4) \vee (v_5 \wedge v_6)$$

   (a) Pour l'ordre $v_1 < v_2 < v_3 < v_4 < v_5 < v_6$
   (b) Pour l'ordre $v_1 < v_3 < v_2 < v_5 < v_4 < v_6$

   Astuce on construira les BDD de manière symbolique c'est-à-dire en utilisant l'opérateur `ite`.

▹

De la définition précedente, on peut tirer un algorithme de construction des BDD

```
algorithme eval(op(F,G))
début
 Si F=0 ou F=1 ou G=0 ou G=1 alors
    Utiliser la table de l'opérateur
 Sinon
    Choisir x dans F et dans G
    U := eval(op(F_{x=1},G_{x=1}))
    V := eval(op(F_{x=0},G_{x=0}))
    Si U=V alors retourner U
    Sinon
      ajouter_trouver(ite(x,U,V))
    Fsi
 Fsi
fin
```

EXEMPLE 3.12.1

Soit les formules $F = (a \vee b)$ et $G = (b \vee c)$, on veut construire $(F \wedge G)$. La formule F

s'exprime $\texttt{ite}(a,1,\texttt{ite}(b,1,0))$*, et la formule* G *par* $\texttt{ite}(b,1,\texttt{ite}(c,1,0))$*. L'appel à la procédure* $\texttt{eval}((F \wedge G))$ *donne la séquence suivante :*

```
Choisir a
U := eval((1 ∧ ite(b,1,ite(c,1,0))))
     soit : ite(b,1,ite(c,1,0))
V := eval((ite(b,1,0) ∧ ite(b,1,ite(c,1,0))))
     soit : choisir b
            U := eval((1 ∧ 1))
                 soit : 1
            V := eval((0 ∧ ite(c,1,0)))
                 soit : 0
            ite(b,1,0)
ite(a,ite(b,1,ite(c,1,0)),ite(b,1,0))
```

◁

Cette structure de données est utilisée dans le langage TOUPIE qui est une implémentation du μ-calcul et que nous survolerons dans le chapitre 10. Il est temps maintenant d'aborder la logique des prédicats.

Chapitre 4

Logique des prédicats

On trouvera une très bonne introduction dans [92, pp 38-70], et dans une moindre mesure dans [55, chap. 3].

La logique propositionnelle est limitée, elle ne peut traiter des syllogismes de la forme :

(p) Tout homme est mortel

(q) Socrate est un homme

(r) Socrate est mortel

Qui peu ou prou se traduirait en calcul des énoncés par $\mathtt{p} \wedge \mathtt{q} \supset \mathtt{r}$.
Alors que la formule correcte devrait être :

$$[\forall x(\text{homme}(x) \supset \text{mortel}(x)) \wedge \text{homme}(\text{Socrate})] \supset \text{mortel}(\text{Socrate})$$

La logique des prédicats, aussi appelée « logique du premier ordre », introduit un nouvel objet : le *prédicat*.

Exemple 4.0.2

grand-pere(marcel,albert) ◁

Le résultat du prédicat est un booléen, i.e. à valeur dans { `vrai`, `faux` }. Dans le cas où le résultat est `vrai`, il y a évaluation des arguments incomplètement spécifiés.

Dans une fonction, tous les arguments sont spécifiés, l'évaluation de la fonction retourne une valeur. Un prédicat, peut-être vu comme la relation existant entre une fonction (munie de ses paramètres) et le résultat retourné.

Exemple 4.0.3

En SCHEME, *(fact 3)* $\rightsquigarrow$ *6*
Peut se traduire `fact_relation(3,6)` $\rightsquigarrow$ `vrai`.
`fact_relation` *simule une fonction lorsqu'elle est utilisée comme :*
`fact_relation(3,X)` $\rightsquigarrow$ *vrai, X=6.* ◁

Les relations sont réversibles, c'est-à-dire que l'on peut fournir le résultat et chercher quelle est la valeur initiale correspondante :

Exemple 4.0.4

fact_relation(X,6) $\rightsquigarrow$ *vrai, X=3* ◁

4.1 Calcul des prédicats, le point de vue formel

Il nous faut définir les objets manipulés par ce calcul, de la même façon que nous avons défini un langage propositionnel.

Il y a :

1. Des connecteurs logiques (comme pour la logique propositionnelle).
2. Des quantificateurs, universel et existentiel.
3. Des constantes relationnelles (prédicats), par exemple grand-pere.
4. Des variables.
5. Des constantes, par exemple 1, 2, albert, marcel dans grand-pere(albert,marcel).
6. Des constantes fonctionnelles, par exemple : « cons », « car », « cdr ».

Nous allons maintenant fixer les notations pour chaque objet de ce langage :

Notation :

1. Les variables seront notées `x`, `y`, `z` éventuellement indicées.
2. Les prédicats seront `p`, `q`, `r` éventuellement indicés.
3. Les constantes seront `a`, `b`, `c` éventuellement indicées.
4. Les constantes fonctionnelles `f`, `g`, `h` éventuellement indicées.
5. Les connecteurs logiques principalement utilisés seront $\neg$, $\vee$, $\wedge$, $\supset$, $\equiv$.
6. Les deux connecteurs seront $\forall$, $\exists$
7. Les constantes booléennes 1 (ou `vrai`) et 0 (ou `faux`).
8. Les symboles de ponctuations « (», « , », «) ».

◇

DÉFINITION 4.1
Un *terme* est soit :

- une constante,
- une variable,
- une constante fonctionnelle suivie d'une suite de termes $t_1,\ldots,t_k$ entre parenthèses. La constante fonctionnelle est appelée *foncteur principal*.

Un terme sera dit *clos* s'il ne contient aucune variable.

Quelques exemples de termes
EXEMPLE 4.1.1
a ; x ; f(x,a,x,g(a),f(a,b)) ◃

DÉFINITION 4.2
On appelle *arité* d'un terme `t`, le nombre d'arguments de ce terme.

Si on reprend les exemples précédents on obtient :

terme	arité
`a`	0
`x`	0
`f(x,a,x,g(a),f(a,b))`	5

DÉFINITION 4.3
Une *forme prédicative* est un terme particulier dont le foncteur principal est un prédicat. L'arité est définie de manière similaire.

Exemple 4.1.2
p, (resp. q(f(a,b),x,y,g(x,a))) est une forme prédicative d'arité 0 (resp. 4) ◁

Définition 4.4
L'ensemble des formules *atomiques* contient les formes suivantes :

- $t_1 = t_2$, où les t_i sont des termes.
- une forme prédicative est une formule atomique.

Définition 4.5
Une *formule* est définie de manière inductive par :

1. Les constantes booléennes sont des formules.
2. Une formule atomique est une formule.
3. $\neg$A est une formule si A est une formule.
4. Si A et B sont des formules alors (A $\bowtie$ B) est une formule pour $\bowtie \in \{\vee, \wedge, \supset, \equiv\}$.
5. Si x est une variable et A une formule ne contenant pas $\forall x$ B ou $\exists x$ B , alors $\forall x$ A est une formule et $\exists x$ A aussi.

Cette dernière règle peut être relâchée mais se pose alors des problèmes qui ne sont pas l'objet de ce cours.

Les variables ont en Logique le même rôle qu'en Algèbre ou en Analyse, elles servent de marqueur de place. Ainsi dans l'expression mathématiques $\lambda x \lambda y.x \times y$ on aurait pu remplacer x par u et y par v sans changer en rien l'expression, par contre l'expression suivante est différente : $\lambda u \lambda u.u \times u$. Revenons à la Logique est considérons la formule :

$$\forall x(\text{homme}(x) \supset \text{mortel}(x))$$

La variable x est *liée* ; la liaison est introduite par la première occurence de x qui suit immédiatement un quantificateur. Une autre façon de présenter, consiste à considérer une formule comme une chaîne de caractères, on dira qu'une variable x est liée dans une formule Φ, s'il existe une sous-chaîne de Φ qui soit une formule de la forme $\forall x\,\phi$ ou $\exists x\,\phi$. La *portée* d'une quantification est la formule à laquelle la quantification se rapporte. La première occurence de x est dite *quantifiée*. Une variable sera dite *libre* si elle n'est ni liée, ni quantifiée.

Exemple 4.1.3
Éxaminons les formules suivantes :

- $\forall x\, p(x,a) \supset \exists x\, q(x)$
- $\forall x\, p(x,a) \supset q(x)$

Dans la première formule, la deuxième occurence de x est liée par le quantificateur universel, tandis que la quatrième occurence est liée par le quantificateur existentiel, on aurait pu écrire $\forall x\, p(x,a) \supset \exists y\, q(y)$. Dans la seconde, la deuxième occurence de x est liée par le quantificateur universel, tandis que la troisième occurence de x est libre ($q(x)$ n'est pas dans la portée du quantificateur), dans ce cas il est préférable de la réécrire sous la forme $(\forall y\, p(y,a) \supset q(x))$. ◁

4.2 Sémantique

Une interprétation pour le calcul des prédicats est un triplet I= $\langle D_I, I_c, I_v \rangle$ avec :

1. $D_I \neq \emptyset$, c'est le domaine d'interprétation
2. I_c une fonction qui associe à toute constante fonctionnelle f d'arité n $I_c(f) : D_I^n \longrightarrow D_I$ et qui à toute constante prédicative p d'arité m $I_c(p) : D_I^m \longrightarrow \{0,1\}$

3. I_v une fonction qui, à toute variable associe une valeur dans D_I.

On peut maintenant définir I

Définition 4.6
Une interprétation $I = \langle D_I, I_c, I_v \rangle$ vérifie les points suivants :

- Si x est une variable libre, $\mathrm{I}[\![x]\!] \stackrel{\text{def}}{=} I_v(x)$.
- Si f est une fonctionnelle d'arité n et si $t_1, \ldots, t_n$ sont des termes alors $\mathrm{I}[\![f(t_1, \ldots, t_n)]\!] \stackrel{\text{def}}{=} I_c(f)(\mathrm{I}[\![t_1]\!], \ldots, \mathrm{I}[\![t_n]\!])$.
- Si p est une constante prédicative d'arité m et si $t_1, \ldots, t_m$ sont des termes alors $\mathrm{I}[\![p(t_1, \ldots, t_m)]\!] \stackrel{\text{def}}{=} I_c(p)(\mathrm{I}[\![t_1]\!], \ldots, \mathrm{I}[\![t_m]\!])$.
- Si t_1 et t_2 sont des termes alors $\mathrm{I}[\![t_1 = t_2]\!] \stackrel{\text{def}}{=} 1$, si $\mathrm{I}[\![t_1]\!] = \mathrm{I}[\![t_2]\!]$ et 0 sinon.
- Si A et B sont des formules, alors $\mathrm{I}[\![\neg A]\!]$, $\mathrm{I}[\![(A \bowtie B)]\!]$ avec $\bowtie \in \{\vee, \wedge, \supset, \equiv\}$, dont la sémantique est identique à celle du calcul des énoncés.

Pour ce qui est de l'interprétation des quantificateurs, on doit préalablement introduire une notation :
Si I est une interprétation de domaine D_I, si $d \in D_I$ alors $\mathrm{I}_{x/d}$ désigne l'interprétation J telle que :

1. $D_J = D_I$
2. $J_c = I_c$
3. $J_v(x) = d$ et $J_v(y) = I_v(y)$ pour toute variable libre $y \neq x$.

Dans ce cas les règles d'interprétation sont données par :

- $\mathrm{I}[\![\forall x A]\!] = 1$, si $\mathrm{I}_{x/d}(\mathrm{A}) = 1$ pour tout élément d de D_I.
- $\mathrm{I}[\![\exists x A]\!] = 1$, si $\mathrm{I}_{x/d}(\mathrm{A}) = 1$ pour au moins un élément d de D_I.

On dira qu'une formule A du calcul des prédicats est *vraie pour une interprétation I* lorsque l'on a $\mathrm{I}[\![A]\!] = 1$.

On a, dans le calcul des prédicats les mêmes notions de formules valides (vraies pour toute interprétation), inconsistantes (vraies pour aucune interprétation) et consistantes (ou contingentes) (vraie pour au moins une interprétation). Si A est une formule, un *modèle* de A est une interprétation I telle que $\mathrm{I}[\![A]\!]=1$. L'introduction des quantificateurs entraîne certaines relations avec les connecteurs logiques. La figure 4.1 est un résumé de ces correspondances :

$(\forall x\, A \wedge \forall x\, B)$	$\equiv$	$\forall x\, (A \wedge B)$
$(\forall x\, A \vee \forall x\, B)$	$\supset$	$\forall x\, (A \vee B)$
$\forall x\, (A \supset B)$	$\supset$	$(\forall x\, A \supset \forall x\, B)$
$\forall x\, (A \equiv B)$	$\supset$	$(\forall x\, A \equiv \forall x\, B)$
$\exists x\, (A \vee B)$	$\equiv$	$(\exists x\, A \vee \exists x\, B)$
$\exists x\, (A \wedge B)$	$\supset$	$(\exists x\, A \wedge \exists x\, B)$
$\exists x (A \supset B)$	$\equiv$	$(\exists x\, A \supset \exists x\, B)$
$\forall x \neg A$	$\equiv$	$\neg \exists x\, A$
$\forall x \forall y\, A$	$\equiv$	$\forall y \forall x\, A$
$\exists x \exists y\, A$	$\equiv$	$\exists y \exists x\, A$
$\exists x \forall y A$	$\supset$	$\forall y \exists x\, A$

Figure 4.1 – Schémas avec quantificateurs

Exercice 4.1
Pour chacune des formules de la figure 4.1 qui met en jeu le connecteur $\supset$ en tant que connecteur principal, montrez que l'on ne peut avoir $\equiv$. **Idée :** on prendra le domaine des entiers et les opérations arithmétiques de base. Ainsi pour établir la dernière formule, on pourra considérer :

$$\forall x \, \exists y \, (x \times y = 1)$$

que l'on comparera à

$$\exists x \, \forall y \, (x \times y = 1)$$

▹

4.3 Substitution et instantiation

Une *substitution* est une application de l'ensemble des variables dans l'ensemble des termes. Une substitution σ est finie si $\sigma(x) \neq x$ pour un nombre fini de variables x. Nous ne considèrerons dans la suite de ce texte que des substitutions finies, et le qualificatif sera donc omis dans la suite.

Définition 4.7
On représente une substitution σ par l'ensemble de paires de la forme x/t où $\sigma(x) = t$.
On étend la notion de substitution au terme de la façon suivante :
Soit $\sigma = \{x/f(y), z/g(y, y)\}$ et soit $t = h(g(x, y), f(a, z))$, $\sigma(t) = h(g(f(y), y), f(a, g(y, y)))$.
La composée de deux substitutions σ_1 et σ_2 est la fonction $\sigma_2 \circ \sigma_1$ définie par :

$$(\sigma_2 \circ \sigma_1)(t) = \sigma_2(\sigma_1(t))$$

Un terme t_1 est une instantiation d'un terme t s'il existe une substitution σ_1 telle que

$$\sigma_1(t) = t_1$$

On supposera, par la suite que les substitutions sont idempotentes, c'est-à-dire que quelque soit la substitution σ utilisée on aura :

$$\sigma \circ \sigma = \sigma$$

4.4 Unificateurs

On introduit maintenant la notion d'unification, soit A et B deux termes, s'il existe une certaine substitution θ telle que $\theta(A) = \theta(B)$ on dira que θ est un *unificateur* de A et B, ou encore que A et B sont *unifiables*. Un unificateur de A et B sera dit mgu [1] s'il est plus général que tout autre unificateur de A et B. De fait, on a le théorème de **Robinson** dont une preuve se trouve dans [4] :

Théorème 4.1
Il existe un algorithme (appelé algorithme d'unification) qui produit, pour deux termes, un mgu, s'ils sont unifiables, ou un message indiquant l'absence d'un unificateur.

4.4.1 Algorithme d'unification

Choisir, de manière non-déterministe, une équation dans l'ensemble des
équations et choisir l'opération idoine :
(1) $f(s_1, \ldots, s_k) = f(t_1, \ldots, t_k)$
remplacer par les équations $s_1 = t_1, \ldots, s_k = t_k$
(2) $f(s_1, \ldots, s_k) = g(t_1, \ldots, t_l)$ avec $f \neq g$
arreter avec échec
(3) $x = x$

[1] Pour des raisons de commodités on gardera l'abreviation anglo-saxone désignant un unificateur le plus général.

supprimer l'équation
(4) $t = x$ avec t pas une variable
remplacer par l'équation $x = t$
(5) $x = t$ où t n'est pas x et x possède une autre occurence dans l'ensemble des équations.
si x apparaît dans t arreter avec échec
sinon appliquer la substitution $\{ x/t \}$ sur toutes les autres équations.

Comme dans le cadre de la logique propositionnelle (voir section 3 page 36) on va pouvoir définir la notion de *littéral* (qui ici sera soit une forme prédicative soit sa négation), la notion de *clause* (comme disjonction de littéraux) et les notions de forme conjonctive (resp. disjonctive) normale. Une *forme de **Skolem*** est une formule qui ne comporte plus de quantificateurs existentiels. On dira qu'une formule est sous forme *clausale* si c'est une forme conjonctive normale sous forme de Skolem. à partir de là, on va pouvoir définir la notion de résolution dans le cadre du calcul des prédicats.

4.4.2 Règle de résolution

Cette règle étend la règle de la définition 3.16 page 50 que l'on a vu dans le cadre de la logique propositionnelle.

Définition 4.8
Soient deux clauses c_1 et c_2 et soit l_1 un littéral tel que $l_1 \in c_1$ et $\neg l_2 \in c_2$ et $\exists\theta$ un mgu de l_1 et l_2. La résolvante est obtenue en ajoutant les littéraux de c_1 (aux quels on a appliqué θ) et de c_2 (aux quels on a appliqué θ) et en faisant disparaître les occurences (positive et négative) de l_1 et l_2, plus formellement $r = (\theta(c_1)\backslash\{l_1\}) \cup (\theta(c_2)\backslash\{\neg l_2\})$.

Exemple 4.4.1
Soit $c_1 = \{\neg p_1(a, x), p_2(x)\}$ *et* $c_2 = \{p_1(y, b), q_1(c, y)\}$, *alors la résolvante est :* $r = \{p_2(b), q_1(c, a)\}$ ◁

Il est maintenant possible de définir la méthode de résolution, identique à la règle de résolution prés. Pour montrer qu'une formule F est conséquence logique d'un ensemble de clauses C (c'est-à-dire que $C \models F$) il suffit de démontrer que $C \cup \{\neg F\}$ est inconsistant.

Exemple 4.4.2
Soit C l'ensemble de 8 clauses suivantes :

1. $p(x, y) \vee \neg q(x) \vee \neg r(z, x) \vee \neg r(z, y)$
2. $q(a)$
3. $q(b)$
4. $q(c)$
5. $r(a, b)$
6. $r(b, c)$
7. $r(a, c)$
8. $p(a, d)$

Et soit la formule $p(b, c)$, *est-ce que* $C \models p(b, c)$ *? pour cela on rajoute une neuvième clause :*

$$\neg p(b, c)$$

On peut alors faire tourner l'algorithme de résolution, pour simplifier la tâche, on se restreindra à éliminer toujours le littéral le plus à gauche, ce qui donne :

à partir de 9 et 1, on obtient :	10. $\neg q(b) \vee \neg r(z, b) \vee \neg r(z, c)$
En prenant 10 et 3, on crée :	11. $\neg r(z, b) \vee \neg r(z, c)$
Avec 11 et 5, on tire :	12. $\neg r(a, c)$
Qui avec 7 nous conduit à :	13. $\square$

On a donc établi que $C \cup \{\neg p(b, c)\}$ *est inconsistant, et donc que* $C \models p(b, c)$. ◁

La séquence choisie n'est bien entendue pas unique, mais plus grave on peut établir que : *l'algorithme ne se termine pas toujours. En effet, selon la stratégie employée pour choisir les clauses on peut engendrer indéfiniment les mêmes résolvantes.*

L'algorithme précédent est intéressant sur le plan théorique mais inefficace sur le plan pratique, la stratégie utilisée en PROLOG est basée sur la *SL-résolution* développée par Kowalski et Kuhener[2].

4.5 De la logique des prédicats à Prolog

Pour des raisons d'efficacité, PROLOG se restreint aux formules mises sous forme de clause de **Horn**. Il s'agit d'une restriction sévère, puisque toute formule du premier ordre ne peut être mise dans ce format.

DÉFINITION 4.9
Une clause de Horn est une clause possédant au plus un littéral positif.

L'avantage de ce format est qu'une clause de Horn est équivalente à une formule dont le connecteur principal est $\supset$ et où les littéraux à gauche de l'implication sont sous forme conjonctive. De plus les variables dans la partie gauche sont quantifiées existentiellement, alors que les variables en partie droite sont quantifiées universellement.

EXEMPLE 4.5.1

$$\forall x \forall y \forall z (gp(x, y) \vee \neg p(x, z) \vee \neg m(z, y))$$

est équivalent à la formule :

$$\forall x \forall y \exists z ((p(x, z) \wedge m(z, y)) \supset gp(x, y))$$

◁

D'autre part, on sait que tout ensemble consistant de clauses de Horn admet un modèle minimal, au sens de l'inclusion des ensembles de littéraux interprétés à vrai.

[2] « Linear Resolution with Selection Function »a été publié dans Artificial Intelligence vol 2, 1971, 227–260.

Chapitre 5

Logique non standard

Nous allons dans ce chapitre survoler rapidement différents systèmes logiques relativement proches de la logique classique. Nous terminerons par les logiques modales, qui introduisent de nouveaux quantificateurs. Ce chapitre reprend la présentation de [3, chap. 8], et de [7, chap. 2].

5.1 Logiques faibles

Il existe de nombreux systèmes logiques proches de la logique classique et qui repose sur une interprétation plus *faible* des notions de négation et d'implication. Il est possible de définir ces systèmes à partir de chaque connecteur pris séparément. De fait, cette approche est même nécessaire puisque les différents opérateurs n'entretiennent pas de relation permettant de les définir les uns par rapport aux autres.

5.1.1 La logique absolue : A

La logique `A` utilise un certain nombre de schémas d'axiomes pour définir les connecteurs $\supset, \wedge, \vee$, l'opérateur $\neg$ n'est pas défini, la négation n'existe pas de ce système qui porte le nom de *logique absolue*.

A-1 $F \supset (G \supset F)$

A-2 $(F \supset (G \supset H)) \supset ((F \supset G) \supset (F \supset H))$

A-3 $(F \wedge G) \supset F$

A-4 $(F \wedge G) \supset G$

A-5 $(F \supset G) \supset ((F \supset H) \supset (F \supset (G \wedge H)))$

A-6 $(F \supset (F \vee G))$

A-7 $(G \supset (F \vee G))$

A-8 $(F \supset H) \supset ((G \supset H) \supset ((F \vee G) \supset H))$

5.1.2 La logique positive : P

La logique `A` ne permet pas d'obtenir tous les théorèmes (sans négation) de la logique classique. La logique *positive* `P` adjoint aux schémas de `A` le schéma suivant :

$$F \vee (F \supset G)$$

5.1.3 La logique minimale : M

Cette logique *minimale* intègre tous les schémas de `A` et rajoute deux schémas pour la négation. La logique `M` introduit la notion de *réfutabilité*. Une formule F est dite « réfutable »si $\neg F$ est prouvable. Les deux schémas à introduire sont :

M-1 $\neg F \supset (F \supset \neg G)$

M-2 $(F \supset \neg F) \supset \neg F$

Dans le système `M` on a le théorème suivant liant négation et implication :

THÉORÈME 5.1
Soit p une proposition alors $\vdash p \supset \neg\neg p$

Remarque : *Dans `M`, la réciproque du théorème précédant est fausse.* •

5.1.4 La logique intuitionniste : J

Le système `J` est une extension du système `A` auquel on adjoint les deux schémas suivants :

J-1 $\neg F \supset (F \supset G)$

J-2 $(F \supset \neg F) \supset \neg F$

Tout théorème de `M` est un théorème de `J`, l'axiome M-1 étant une conséquence triviale de l'axiome J-1. L'interprétation associée à la négation est « est absurde ». Pour retrouver la logique classique, il suffit d'adjoindre au système `J`, le schéma supplémentaire du système `P`.

5.2 Logiques modales

On peut faire remonter les origines de la logique modale chez Aristote ou au Moyen-âge, cependant on s'accorde à situer sa naissance dans les travaux de C. Lewis qui aborde la notion d'« implication stricte », notée $>$. Il a introduit cette notion suite à l'étide de ce que l'on nomme les « paradoxes »de l'implication matérielle :

$$F \supset (G \supset F) \text{ et } \neg F \supset (F \supset G)$$

Ces deux formules sont des théorèmes de la logique classique (démonstrations laissées aux lecteurs) et qui peuvent se lire :

- Si F est vraie alors de n'importe quoi on peut déduire F.
- Si F est faux alors de F on peut déduire n'importe quoi.

C. Lewis a introduit la notion d'implication stricte pour éviter de déduire n'importe quoi d'un énoncé faux. Le connecteur « $>$ »exprime que « si F implique G alors G découle de F »et Lewis le définit en faisant intervenir la notion de *possibilité*, notée $\Diamond$. Il pose :

DÉFINITION 5.1 (IMPLICATION STRICTE ET POSSIBILITÉ)

$$(F > G) \stackrel{\text{def}}{=} \neg\Diamond(F \wedge \neg G)$$

C'est-à-dire qu'il est « impossible que F soit `vrai` et G `faux` lorsque F implique strictement G »

Les premiers systèmes modaux élaborés par Lewis visent donc à donner une logique à l'implication stricte mais ils amènent implicitement une logique de la *possibilité* et de son dual la *nécessité*, notée $\Box$.

Définition 5.2 (Nécessité)

$$\Box F \stackrel{\text{def}}{=} \neg\Diamond\neg F$$

Que l'on lit « F est nécessaire »ou, de manière équivalent « $\neg F$ est impossible ». De ces deux définitions il découle :

$$\Box(F \supset G) = (F > G) \tag{5.1}$$

Nous allons établir l'équation 5.1 en nous appyant sur les définitions 5.2 et 5.1, sur la notion d'implication matériel (section 3.4) et de l'effet de la négation sur la disjonction.

$$\Box(F \supset G) = \neg\Diamond\neg(F \supset G) = \neg\Diamond\neg(\neg F \vee G) = \neg\Diamond(F \wedge \neg G) = (F > G)$$

On appelle « logique aléthique »une logique modale possédant les modes « possibilité »et « nécessité ». Il existe d'autres modalités, comme la « connaissance »de la *logique épistémique* ou l'« obligation »de la *logique déontique*[1].

C. Lewis a étudié plusieurs systèmes aléthiques qu'il a baptisé système 1, système 2, ... et dont les plus connus sont les systèmes `S4` et `S5`. Depuis quelques dizaines d'années une nouvelle nomenclature est apparue qui consiste à désigner les systèmes par le nom des axiomes les caractérisant. Par exemple, le système `S4` est appelé `KT4`. Avant d'étudier quelques sytèmes modaux, nous allons introduire quelques définitions préalables.

5.2.1 Langage

Le langage de la logique modale est construit de la même manière que celui du calcul des énoncés (cf chapitre 3 section 3.2) auquel on adjoint l'ensemble des modes : Mod = { $\Diamond$, $\Box$}.

Définition 5.3
L'ensemble des formules modales $\mathcal{F}_M$ construites sur $\mathcal{P}$ est le plus petit ensemble tel que :

- Si $F \in \mathcal{F}_M$ alors $\neg F \in \mathcal{F}_M$, $\Diamond F \in \mathcal{F}_M$, $\Box F \in \mathcal{F}_M$
- Si F et G sont des formules alors $(F \wedge G)$, $(F \vee G)$, $(F \supset G)$ et $(F \equiv G)$ appartiennent à $\mathcal{F}_M$.

Définition 5.4 (Modalités)
On appelle *modalité* tout mot construit sur l'alphabet $\{\neg, \Diamond, \Box\}$

Nous allons voir plusieurs exemples de modalités
Exemple 5.2.1

- *Le mot vide est une modalité*
- $\neg$ *est une modalité*
- $\Diamond$*est une modalité*
- $\Box$*est une modalité*
- $\Diamond^{\alpha}\Box^{\beta}$ *est une modalité*.

◁

Définition 5.5 (modalités équivalentes, distinctes)
On dira que deux modalités φ ey ψ sont *équivalentes* si $\varphi F \equiv \psi F$ est un théorème pour toute formule F.
On dira que deux modalités φ et ψ sont *distinctes* si elles ne sont pas équivalentes.

[1] Pour en savoir plus, le lecteur pourra consulté `http://logique.uqam.8m.com/histoire11.htm` ou la section B.3.

Chapitre 6

Logique floue

Troisième partie

Langages de l'IA

Chapitre 7

Langages Applicatifs

Lisp [77, 17] est né aux alentours des années 60. A cette époque on ne comptait que très peu de langages, et tous n'en était qu'à leur début. Actuellement ce langage existe sur toutes les machines de la plus humble (micro-ordinateur) à la plus grosse, sous de très nombreuses versions (MacLisp, CommonLisp, Xlisp, Vlisp, FranzLisp, Le_Lisp, ...).

Conçu au MIT (Massachussets Institute of Technology) par J. Mc Carthy [67], Lisp devait, à l'origine, être une extension de Fortran [61]. Le but étant d'ajouter à Fortran[1], la possibilité de manipuler des symboles, notamment pour faire de la dérivation et/ou de l'intégration formelle de fonctions.

Très vite, Mc Carthy décida de considérer Lisp comme un langage à part entière (Lisp est acronyme de "List Processing Language"). Toute expression symbolique étant exécutée sous forme de listes. Il fit également la remarque qu'un programme peut lui même être présenté comme une expression symbolique ; ainsi il est fréquent qu'un programme écrit en LISP s'*auto-modifie* en cours d'éxécution. Qui plus est, ce langage s'appuie sur une des réalisations fondamentales de la logique : le *lambda calcul* de Church qui permet d'exprimer formellement la sémantique de l'évaluation de fonctions mathématiques (et informatiques).

SCHEME est un dialecte de LISP créé par G. Lewis, G. Sussman et J. Sussman, dans le but d'obtenir un langage clair avec une sémantique claire et comportant très peu de façons pour construire une expression particulière. SCHEME offre tout à la fois :

1. Des techniques de programmations impératives.
2. Des techniques de programmations applicatives.
3. La possibilité d'envoyer des messages.

Le lecteur intéressé se reportera aux livres et manuels suivants pour approfondir ses connaissances [83, 1, 2, 36, 89]. Il existe plusieurs implémentations sous `Linux` pour ma part j'utilise principalement `scm` de Aubrey Jaffer [44] et `Guile` [86] de la `Free Software Foundation` [85].

7.1 Introduction

SCHEME est à la confluence de deux approches, d'une part celle de LISP et d'autres part celle d'ALGOL et de ses descendants. De LISP SCHEME a hérité de sa puissance méta-linguistique, d'un typage automatique, d'une interface interactive, d'une gestion automatique de la mémoire et de la représentation des programmes en tant que données. De la tradition issue d'ALGOL la notion de structure de blocs, et de la liaison léxicale des variables. Il intégre de plus la récursion terminale, le passage par valeurs des paramètres et aucune contrainte sur l'ordre d'évaluation des paramètres.

[1] langage très utilisé dans le domaine du calcul numérique

7.2 Langage

7.2.1 Lambda Calcul

Lisp et ses avatars repose sur la théorie du λ-calcul introduit par Church. On pourra considérer dans ce qui suit le λ-calcul comme une notation fonctionnelle :

$$\left.\begin{array}{rccc} f: & D_1 & \longrightarrow & D_2 \\ & x & \mapsto & f(x) \end{array}\right\} \lambda x.f(x)$$

Ainsi la fonction qui élève un nombre au carré pourra s'écrire : $\lambda x.x^2$ et son application à une valeur : $(\lambda x.x^2)2 \rightsquigarrow 4$.

Autre principe important la curryfication qui consiste à considérer une fonction à n paramètres comme la composition d'une fonction à $(n-1)$ paramètres et d'une fonction à un paramètre (fonction unaire), cette opération est appelée *curryfication* du nom de H. Curry.

Exemple 7.2.1

$$\left.\begin{array}{rccc} f: & R \times Z & \longrightarrow & R \\ & (x,y) & \mapsto & x^y \end{array}\right\} \lambda x.\lambda y.x^y$$

En d'autres termes, une fonction de $D_1 \times D_2 \times \ldots \times D_n \longrightarrow D$ *sera vue comme une fonction de* $D_1 \longrightarrow (D_2 \longrightarrow \ldots (D_n \longrightarrow D)\ldots)$. ◁

Si E est une fonction à deux arguments, l'expression Exy peut être lue $E'y$ où E' est la fonction à un argument (Ex), dont le résultat est aussi une fonction à un argument. On dira que E est une *fonctionnelle*.

Les trois sections suivantes sont directement inspirées du polycopié [13] "Modèles et Techniques de programmation" de l'Université de Bordeaux I.

Langage

Le λ-calcul est défini par :

- Un nombre infini de noms de variables
- Un quantificateur noté λ
- Un ensemble d'éléments de ponctuation "(", ".", ")"

On notera les variables par des minuscules, et les λ-expressions à l'aide de majuscule.

Les termes du λ-calcul sont construits de manière très simple, à l'aide deux opérations l'*application* et l'*abstraction*.

Définition 7.1
L'ensemble des termes du λ-calcul est le plus petit ensemble tel que :

1. Les variables $x, y, z, \ldots$ sont des termes.
2. Si U et V sont des termes, alors (UV) est un terme (application).
3. Si x est une variable, et T un terme, alors $\lambda x.T$ est un terme (abstraction).

Variables libres, liées et Substitutions

Les occurences libres d'une variable x dans un terme T sont définies par induction sur la longueur du terme :

1. Si $T = x$, l'occurence de x est libre dans T.
2. Si $T = (UV)$, les occurences libres de x dans T sont les occurences libres de x dans U et V.

3. Si $T = \lambda y.U$ alors

   - si $x = y$ alors, x n'a pas d'occurence libre dans T, on dira que x est liée dans T.
   - si $x \neq y$ alors, les occurences libres de x dans T, sont les occurences libres de x dans U.

Une variable x est libre dans T si elle possède au moins une occurence libre dans T (on notera $x \in \mathbf{Free}(T)$). Un terme T est clos, s'il n'a pas de variable libre.

L'un des avantages du λ-calcul est de pouvoir modéliser simplement le résultat de l'application d'une fonction, en effet on peut voir l'application de $\lambda x.M$ sur un terme N comme la substitution des occurences libres de x dans M par N. On notera

$$M[x/N]$$

cette opération. Cependant cette opération ne se fait pas sans contrainte si N contient des variables libres qui sont liées dans M. Dans ce cas, on doit préalablement renommer les variables liées de M qui sont en "conflit" avecles variables libres de N. De manière plus formelle on a :

Définition 7.2

$$\begin{array}{rcll} x[x/N] & = & N & \\ y[x/N] & = & y & y \neq x \\ (PQ)[x/N] & = & P[x/N]Q[x/N] & \\ (\lambda x.P)[x/N] & = & \lambda x.P & \\ (\lambda y.P)[x/N] & = & \lambda y.P[x/N] & y \neq x, y \notin \mathbf{Free}(N) \\ (\lambda y.P)[x/N] & = & \lambda z.(P[y/z][x/N]) & y \neq x, z \notin \mathbf{Free}(N) \cup \mathbf{Free}(P) \end{array}$$

Règles de Conversion

L'évaluation d'une expression dans le λ-calcul passe par une suite de réécriture utilisant l'une des trois règles suivantes de réduction ou de conversion :

β-réduction :

$$(\lambda x.M)N \rightsquigarrow_\beta M[x/N]$$

α-conversion : si y n'est pas libre dans M,

$$\lambda x.M \rightsquigarrow_\alpha \lambda y.M[x/y]$$

η-conversion : si x n'est pas libre dans M,

$$(\lambda x.Mx) \rightsquigarrow_\eta M$$

Stratégies d'évaluation

Il existe plusieurs façons d'effectuer la suite de transformation lorsque l'on évalue une λ-expression, le théorème de **Church-Rosser** garantit que lorsqu'elle existe, il y a une unique forme normale (forme irréductible) et que quelque soit la séquence que l'on choisisse on y aboutira fatalement [2]. Le principe de la stratégie `d'appel par nom` consiste à évaluer en premier les expressions les plus à gauche parmi les plus externes, ainsi l'évaluation de $(\lambda x.x * x)((\lambda x. - x)1)$ serait :
$\rightsquigarrow_\beta ((\lambda x. - x)1) * ((\lambda x. - x)1)$
$\rightsquigarrow_\beta -1 * ((\lambda x. - x)1)$
$\rightsquigarrow_\beta -1 * -1$

[2] sous réserve, bien entendu qu'elle existe

7.2.2 Syntaxe

SCHEME fait un usage abondant (les détracteurs diront excessif [3]) de parenthèses qui servent à délimiter les S-expressions.
Le langage est constitué d'une part, d'atomes parmi lesquels on trouve : les nombres (entiers et flottants), les opérateurs (arithmétiques), les identificateurs (soient prédéfinis, soient utilisateurs), quelques constantes système (notament la liste vide) et les chaînes de caractères qui sont encadrés par des guillemets. Et d'autres part les S-expressions qui sont une séquence d'objets séparés par des espace et encadrée par des parenthèses. Une S-expression utilise la notation préfixe, c'est à dire que le premier élément de la liste est l'opérateur qui sera appliqué aux autres objets vus comme argument.

SCHEME fonctionne comme un interpréteur, pour chaque commande tapée, le système en fait l'évaluation, dans la suite nous utiliserons $\rightsquigarrow$ pour indiquer les différentes étapes d'une évaluation en SCHEME. Le symbole > sera utilisé pour indiquer que le système est prêt à recevoir une nouvelle commande de l'utilisateur.

EXEMPLE 7.2.2

```
> 3
⇝ 3
> (+ 4 5)
⇝ 4 + 5
⇝ 9
>
```

Imaginons que l'on ait défini les fonctions `carre` *et* `cube` *ainsi que les valeurs* `a = 3` *et* `b = 5` *et que l'on souhaite calculer* $a^2 + b^3$

```
> (+ (carre a) (cube b))
⇝ (+ 9 125)
⇝ 134
```

◁

7.2.3 Quel système

Il existe de très nombreuses implémentations de SCHEME accessibles gratuitement sur la toile, parmi celles que j'ai utilisé à différentes périodes : `scm, umb-scheme, guile` [44, 86]. Depuis la popularisation de `Linux` et de certains systèmes de fenêtrage, les distributions embarquent en standard `umb-scheme` et `guile`.

7.3 Éléments du langage

Il est possible d'introduire de nouveaux objets en SCHEME grâce à la commande `define` qui prend deux paramètres, le premier est le nom du nouvel objet, le second étant la valeur que l'on souhaite lui affecter :

EXEMPLE 7.3.1

```
> (define a 3)
> (define b 5)
> (define c (+ (* a a ) (* b b b)))
> a
3
> b
5
> c
```

[3] Ils ont d'ailleurs donnés un sens particulier à l'acronyme LISP i.e. Lot of Insipide and Stupid Parenthesis

```
134
> (define b 6)
> b
6
> c
134
```

On peut, en suivant le même schéma introduire de nouvelles fonctions, ainsi les fonctions `carre` et `cube` de l'exemple 7.2.2 seront créées par :

EXEMPLE 7.3.2
```
> (define (carre x) (* x x))
> (define (cube x) (* (carre x) x))
```

7.3.1 Particularités

En SCHEME les commentaires commencent par un point virgule `;` et vont jusqu'à la fin de la ligne. SCHEME permet par ailleurs de manipuler des symboles, pour éviter d'évaluer une expression, on utilise `quote` de la manière suivante : `(quote symbole)`. Comme, par ailleurs, on est souvent amené à utiliser le processus de "quotation", on a la possibilité d'utiliser une abréviation en faisant précéder l'expression à ne pas évaluer par une apostrophe `'`. La fonction `eval` permet de contrecarrer l'effet de l'apostrophe. On ne peut utiliser que des objets qui sont préalablement définis, l'ordre des définitions n'est pas pertinents en SCHEME à la différence des langages de la famille ALGOL. On trouvera dans l'annexe C quelques éléments importants du langage qui n'ont pas leur place dans le texte principal.

EXEMPLE 7.3.3
```
> (define a 3) ;; a est un entier
> (define b (+ a 3)) ;; b vaut la même valeur que a + 3
> (define c 'b) ;; c a pour valeur le symbole b
> (define d (quote b)) ;; d a pour valeur le symbole b
> a
3
> b
6
> c
b
> (eval c)
6
> (eval d)
6
```

Il existe plusieurs opérateurs permettant de tester l'égalité de deux expressions :

`=`	pour comparer deux expressions arithmétiques
`eq?`	pour comparer deux expressions retournant un symbole
`eqv?`	pour deux expressions retournant, un nombre, un symbole, un booléen
`equal?`	idem que `eqv?` mais en plus procédure et liste

Bien entendu, plus un opérateur est général, moins il sera efficace par rapport à un opérateur spécialisé.

7.3.2 Conditionnelles

On identifie deux conditionnelles, l'une correspondant au classique si-alors-sinon, l'autre pour les conditionnelles à choix multiples. La première est faite sur le schéma `(if test partie_si partie_sinon)`,

la seconde étant définie par : `(cond (test_1 sequence_1) (test_2 sequence_2) ...(test_n sequence_n) (else sequence_recup))` la dernière branche (celle du `else`) étant faclcutative.

EXERCICE 7.1

1. Écrire la fonction factorielle définie par :

$$\begin{cases} f(0) &= 1 \\ f(n) &= n * f(n-1) \quad \forall n > 0 \end{cases}$$

2. Écrire la fonction de fibonacci définie par

$$\begin{cases} f(1) &= 1 \\ f(2) &= 1 \\ f(n) &= f(n-2) + f(n-1) \quad \forall n > 2 \end{cases}$$

▹

7.3.3 Récursivité terminale

Dans l'exercice 7.1 on s'aperçoit que le calcul n'est pas forcément très efficace, c'est pourquoi on introduit une technique de programmation appelée "récusrivité terminale", dont l'objectif est d'avoir calculer le résultat de la fonction lorsque l'on atteint le dernier appel récursif. Pour cela on a besoin d'introduire une fonction auxiliaire qui prend en charge le calcul et le passage des résultats intermédiaires :

EXEMPLE 7.3.1

```
(define (fact n)
;; factaux est la fonction auxiliaire assurant la recursivite terminale
 (define (factaux p q)
  (if (= p 1) q (factaux (- 1 p) (* p q))))

;; programme principal qui agit en fonction de la valeur de n
 (cond ((or (= n 0) (= n 1) 1))
       ((> n 2) (factaux n 1))
       (else (error "La factorielle n'est pas definie pour n < 0"))
       )
)

(define (fib+ n)
  (define (fibaux p a1 a2)
    (cond ((= p 0) a1)
          ((= p 1) a2)
          (else (fibaux (- p 1) a2 (+ a1 a2)))))
  (fibaux n 1 1))
```

◃

EXERCICE 7.2

Écrire une fonction (voire plusieurs) qui permettent de :

1. faire la somme des n premiers termes,
2. faire la somme des carrés des nombres pris dans un intervalle,
3. faire la somme des cubes des nombres impairs pris dans un intervalle,
4. faire le développement limité à l'ordre n de la fonction cosinus.

▹

On trouvera dans l'annexe C une correction possible de cet exercice.

7.3.4 Scheme et les listes

Commençons par clarifier la notion de Listes (ou de S-expressions [4])

Définition 7.3
Une liste est de l'une des formes suivantes :

1. ( ) : la liste *vide*.
2. $(o_1 \dots o_n), n \geq 1$ où o_i est soit un atome soit une liste $\forall i \in [1, n]$.

On considère deux types d'opérateurs sur les listes, d'une part les constructeurs (de fait il en existe deux principaux, mais un seul est générique `cons`) et les destructeurs (au nombre de deux `car` et `cdr`). L'objectif d'un constructeur étant à partir d'éléments du langage de construire une S-expression, alors que le rôle d'un destructeur est d'accéder aux éléments constitutifs d'une liste.

Définition 7.4
On peut définir les trois opérateurs de la façon suivante :

`cons`	Atome × List	⟶	List
`car`	List	⟶	Atome ∪ List
`cdr`	List	⟶	List

Avec l'axiome suivant : Si $l \in$ List alors `cons`(`car`(l), `cdr`(l)) = l

Exemple 7.3.2

```
> (car '((a b) (c d)))
⇝ (a b)
> (cdr '((a b) (c d)))
⇝ ((c d))
```
◃

Exercice 7.3
On considère les trois listes suivantes :

```
l1 = ((a b) a b)
l2 = (a)
l3 = (b)
```

Calculer le `car` et `cdr` pour chaque liste, construire `l1` à partir de `l2` et `l3` ▹

Il existe un autre constructeur de listes, noté `list`, qui à partir de n objets retourne une liste constituée de ces n objets. De fait il s'agit d'un sucre syntaxique qui évite d'avoir à écrire n `cons`.
Exemple 7.3.3

```
(list 'a 'b 'c) ⇝ (a b c)
(cons 'a (cons 'b (cons 'c '()))) ⇝ (a b c)
```
◃

Il existe quelques prédicats, fonctionnelles et opérateurs utiles à connaitre lorsque l'on manipule des listes :

Définition 7.5

1. `null?` qui prend en paramètre une liste et renvoie **vrai** si la liste est vide
2. `pair?` qui prend en paramètre un objet et renvoie **vrai** si l'objet est une liste non vide
3. `eqv?` qui prend en paramètre deux objets et qui renvoie **vrai** s'ils sont identiques.
4. `#t` est l'atome booléen **vrai**.

[4] on utilisera indifférement l'une des deux appellations daans la suite de ce texte.

5. `#f` est l'atome booléen **faux**.

6. `cadr` est une abréviation pour obtenir le premier éléments de la fin d'une liste (c'est le `car` du `cdr`). De fait, en SCHEME une séquence quelconque de `a` et `d` pris entre un `c` et un `r` est une séquence de quelconque de `car` et de `cdr`.

On trouvera sur `http://swissnet.ai.mit.edu/~jaffer/Scheme.html` une version hypertexte de [1], qui donne accès aux opérateurs standards que l'on doit trouver dans une implémentation SCHEME conforme.

EXERCICE 7.4
Étant donnée une liste de trois éléments :

1. réaliser une permutation circulaire, c'est-à-dire, le premier élément devient le second, le second devient le troisième et le troisième le premier.
2. généraliser à une liste de longueur n connue
3. généraliser à une liste quelconque.

▹

EXERCICE 7.5

1. Écrire la fonction `concat`, qui, étant données deux listes `l1` et `l2`, calcule une troisième liste `l3` qui contient, dans l'ordre les éléments de `l1` suivis des éléments de `l2`. Cette opération est connue comme la concaténation de deux listes. à titre d'exemple :
   `(concat '((a b) a b) '(1 2 3 (4 5))) ⇝ ((a b) a b 1 2 3 (4 5))`
2. Calculer la longueur `longueur` d'une liste i.e. le nombre d'éléments qui la constituent. Si on considère l'exemple précédent, on vérifie que :
   `(longueur (concat l1 l2)) = (+ (longueur l1) (longueur l2)) ⇝ 7`
3. Calculer la profondeur `pf` d'une liste définie comme le nombre maximum d'imbrications de parenthèses. Si on reprend l'exemple donné pour la concaténation, on doit vérifier que :
   `(pf (concat l1 l2)) = max((pf l1), (pf l2)) ⇝ 2`
4. Écrire une fonction `elem?` qui permet de savoir si un objet appartient à une liste. Toujours dans l'exemple pris au début de cet exercice, on vérifiera que :
   `(elem? '5 (concat l1 l2)) = (or (elem? '5 l1) (elem? '5 l2)) ⇝ #t`

▹

7.3.5 Environnement

Les objets en Scheme sont définis dans un environnement, et leurs valeurs dépendent fortement de l'environnement courant. Il est possible de modifier cet environnement grâce à la fonctionnelle `set!` qui correspond peu ou prou à l'affectation dans les langages impératifs, la syntaxe est alors : `(set!var val)`.

Il est par ailleurs possible de définir des variables locales qui ne se réfèrent qu'à l'environnement courant, par l'intermédiaire de `let`, dont la syntaxe est :
`(let ((var_1 val_1) ...(var_k val_k)) corps_utilisant_var_i)`.
Grosso modo, cela revient à considérer que les couples `(var_1 val_1) ...(var_k val_k)` forment l'environnement dans lequel `corps_utilisant_var_i` sera évalué, par ailleurs, il n'y a pas d'effet de bord d'une variable sur l'autre.

EXEMPLE 7.3.4

```
>(define a 5)
>(define b 6)
>(define c
     (let ((a 24) (b (+ a 5)) (x (+ a b)))
       (* (+ x a) b)))
>a
↝ 5
>b
↝ 6
>c
↝ 350
```

◁

Pour obtenir des effets de bord, il faudra utiliser un "`let` séquentiel" obtenu par l'instruction `let*`.

EXEMPLE 7.3.5

```
>(define a 5)
>(define b 6)
>(define c
     (let* ((a 24) (b (+ a 5)) (x (+ a b)))
       (* (+ x a) b)))
>(define d
     (let* ((b (+ a 5)) (x (+ a b)))
       (* (+ x a) b)))
>a
↝ 5
>b
↝ 6
>c
↝ 2233
>d
↝ 200
```

◁

7.3.6 Quelques exercices

Cette section est une série d'exercices de taille moyenne, afin de mettre en pratique les différentes notions vues jusqu'ici. Il y a trois exercices, l'un est une gestion de compte bancaire, le second est le calcul de dérivées formelles, le troisième et dernier exercice est la réalisation en SCHEME d'algorithmes de gestion des arbres binaires vu en cours d'Algorithmique et Structures de Données en Licence MASS.

EXERCICE 7.6

On souhaite réaliser la gestion de compte bancaire, pour cela on voudrait écrire une fonction `cc` qui permette :

1. D'initialiser un compte par une commande du type : `(define Paul (cc 10000))`, créant un compte courant pour Paul de 10000 FF.
2. On veut pouvoir effectuer des virements, des retraits et connaitre le solde d'un compte courant de la manière suivante :

   ```
   > (Paul 'virement 500)
   ↝ 10500
   > (Paul 'retrait 800)
   ↝ 9700
   > (Paul 'solde)
   ↝ 9700
   ```

▹

EXERCICE 7.7

On souhaite faire la dérivation formelle d'une fonction arithmétique par rapport à une variable. On aimerait de plus obtenir une forme simplifiée de cette dérivée dans laquelle on prenne en compte les éléments neutre de l'addition et de la multiplication. ▹

EXERCICE 7.8

On décide de représenter un arbre binaire (muni d'une relation d'ordre >) par le biais de listes SCHEMEAinsi l'arbre binaire de recherche dans lequel on a stocké successivement les valeurs 5, 2, 6, 4 sera représenté par la liste `(5 (2 () (4)) (6))`. On veut :

1. Créer un ABR (arbre binaire de recherche).
2. Ajouter un élément dans l'arbre.
3. Retirer un élément dans un ABR, tout en gardant la propriété d'ABR.
4. Savoir si un élément est dans l'arbre.
5. Obtenir la liste ordonnée des éléments dans l'arbre.
6. Connaître la hauteur de l'arbre.
7. Connaître ses longueurs (LCE, LCI et LC).

▹

Chapitre 8

Programmation Logique

8.1 Le langage Prolog

PROLOG : PROgrammer en LOGique

PROLOG [21, 14, 10] est un langage de programmation basé sur la logique du premier ordre. Unification et retour-arrière (en anglais “backtracking”) en sont les deux mécanismes principaux. Il existe de nombreuses implémentations de ce langage, je vous en propose deux parmi celles auxquelles on peut accéder gratuitement : `Gnu-Prolog` [31] et `SWI-Prolog` [87] de la `Free Software Foundation` [85].

Développé dans le courant des années 70–75 à l’université de Marseille comme un outil pratique de la programmation logique par Roussel et Colmerauer. PROLOG a connu un intérêt croissant durant la dernière décennie du 20ème siècle du fait que les Japonais l’ont sélectionné comme langage noyau des ordinateurs de "$5^{ème}$ génération" (terminologie pompeuse à but publicitaire, dont les résultats n’ont pas été à la hauteur des espérances).

Il est, du point de vue de l’utilisateur, clair, lisible, concis, et permet une grande rapidité d’écriture.

C’est un langage particulièrement adapté aux systèmes sur les langages naturels, les systèmes experts, et à de nombreux domaines de l’IA symbolique.

Les structures de données PROLOG sont en général des arbres. Un nombre illimité de types peuvent être utilisés, sans avoir à être déclarés séparément. Pour manipuler de telles données Lisp ou Scheme utilise les notions de “constructeur” et de “destructeur”, PROLOG quant a lui fournit un mécanisme plus général : le *pattern-matching* utilisé dans le cadre de l’unification.

Un calcul en PROLOG peut être vu comme une suite d’appels dirigés de procédures. Du fait de l’importance des “procédures” il est nécessaire d’avoir plus de flexibilité que dans les autres langages, l’utilisation d’une procédure peut varier d’un appel à l’autre, en effet les paramètres peuvent suivant le cas être :
– des paramètres de retour (résultats),
– des paramètres d’entrée (données) et cela n’a pas besoin d’être prédéterminé, cette propriété permet aux procédures d’être multi-directionnelles.

De plus les procédures doivent permettre, dans certains cas, de générer des solutions différentes à chaque appel, cette propriété est appelée *non-déterminisme* et prend effet au sein d’un mécanisme plus complexe : le *backtracking*.

DÉFINITION 8.1
Un programme est déterministe s’il ne fournit qu’une solution. S’il fournit plusieurs réponses (solutions), il est non-déterministe.

Le système PROLOG peut être vu comme un constucteur de preuves pour un programme donné, il y aura *succès* si une preuve peut être établie et *échec* sinon. L’échec signifiant non pas que le programme est **faux** mais que le système ne peut établir une preuve en utilisant les informations (le

programme constitué de faits et de règles) à sa disposition. On dit aussi que PROLOG fait l'hypothèse du *monde clos*.

Un programme PROLOG est un mélange de faits et de règles de calculs auxquels on accède de façon identique, c'est ce qui justifie l'intérêt porté à PROLOG pour les langages d'interrogation de bases de données (e.g. `DataLog`).

La syntaxe PROLOG dont nous allons donner quelques aperçus est celle de PROLOG *d'Edimbourg* [21], un standard de fait.

8.1.1 Syntaxe et définition

Un programme PROLOG est une suite de *clauses*. Chaque clause comprend une *tête* et un *corps*. Le corps est une suite, éventuellement vide, de *buts*, ainsi une clause peut être écrite comme :

```
P :- Q,R,S.
```

où `P` est la tête de clause ; `Q, R, S` forment le corps de la dite clause.

```
P.
```

est aussi une clause, elle est qualifiée d'*unitaire*, du fait que le corps est vide.

Pour comprendre la signification d'une clause il suffit de l'interpréter comme un moyen raccourci de décrire une phrase d'un langage naturel (français, ...). Une clause non unitaire telle que :

```
P :- Q,R.
```

peut être lue "`P` est vrai si `Q` et `R` le sont." ou encore "pour que `P` soit vrai il faut que `Q` et `R` le soient".

Une clause unitaire revient donc à une affirmation.

Tête et buts d'une clause sont des *termes* appelés *littéraux*.

En PROLOG le `:-` se lit *si*, les virgules séparant les littéraux correspondent à un `et` logique, et le point termine toujours une clause.

On peut définir les termes de façon inductive :
un terme est soit un *terme_de_base*, soit un *terme_composé* ou *structure*.
Un terme_de_base est soit une *variable*, soit un *atome*, soit un *entier* :

- Une variable est un identificateur commençant par une majuscule, ou un sous-ligné "_" ainsi
  – `X,Pere,_papa,_` sont des variables
  – `_` a une signification particulière que nous verrons ultérieurement.
- Un atome est un identificateur qui n'est ni une variable ni un entier.
- Un entier s'écrit au moyen des chiffres de 0 à 9.

Un terme_composé est constitué d'un *foncteur* (en anglais functor) et d'une liste de un ou plusieurs termes appelés *arguments*.
Un foncteur est un atome caracterisé par son *arité* (i.e. le nombre de ses arguments), ainsi le terme_composé dont le foncteur est `pere`, d'arité 2 et ayant pour arguments `jean` et `jacques` s'écrit :

pere(jean,jacques).

Le foncteur d'un littéral est appelé un *prédicat* :

```
(c1) conc([],L,L).
(c2) conc(cons(X,L0),L1,cons(X,L2)):-conc(L0,L1,L2).
```

1. `c1, c2` sont des clauses,
2. `conc` est un prédicat,

3. `cons(X,L)` est un terme_composé dont le foncteur est `cons`,

4. `[]` est un atome,

5. `L` est une variable,

6. `conc(L0,L1,L2)` est un (le) but appartenant au corps de `c2`

Pour comprendre un programme il faut attacher une signification (*sémantique*) à chaque prédicat, cette signification devra être la même pour toutes les références faites à ce prédicat, i.e. le prédicat est une *relation* qui lie entre eux les arguments du terme considéré.

```
pere(jean,pascale).
pere(paul,jacques).
pere(andre,marc-michel).
pere(luc,adeline).
pere(andre,dominique).
pere(andre,isabelle).
```

Le prédicat `pere` pourrait avoir la signification "est le père de " ainsi `pere(paul,jacques)` sera interprété comme "paul est le père de jacques" et devra être interprété de cette façon tout au long de *ce* programme afin d'éviter des interprétations farfelues telles que "adeline est le père de luc".

Une *procédure* PROLOG est définie comme l'ensemble des clauses qui ont même prédicat et arité, par exemple la procédure `pere`/2 (prédicat `pere`, arité 2) est composée de six clauses.

8.1.2 PROLOG par la pratique

Une *base de faits* est un ensemble d'informations, dans un premier temps nous ne considèrerons que des clauses unitaires. Supposons qu'on veuille représenter la carte d'un restaurant. On peut le faire de la façon suivante :

```
/*                          MENU                                    */

hors_d_oeuvre(crudites).
hors_d_oeuvre(assiette_anglaise).
hors_d_oeuvre(salade_nicoise).
hors_d_oeuvre(salade_greque).

plat_principal(saucisses,lentilles).
plat_principal(roti_porc,puree).
plat_principal(omelette,champignons).
plat_principal(steack,frites).

dessert(fromage).
dessert(fruits).
dessert(yaourt).
dessert(glace).
```

Les commentaires sont placés entre `/*` et `*/` .

Supposons que ce texte soit stocké dans le fichier `menu.pl` (le suffixe `pl` a été choisi pour indiquer que le fichier contenait un source PROLOG). Voici un exemple de session possible :

```
18:02:55 firmin-MM-bash:  cprolog
C-Prolog version 1.5
?- consult('menu.pl').
menu.pl consulted 512 bytes 0.0166667 sec.

yes
?- listing.

hors_d_oeuvre(crudites).
hors_d_oeuvre(assiette_anglaise).
hors_d_oeuvre(salade_nicoise).
hors_d_oeuvre(salade_greque).

plat_principal(saucisses,lentilles).
plat_principal(roti_porc,puree).
plat_principal(omelette,champignons).
plat_principal(steack,frites).

dessert(fromage).
dessert(fruits).
dessert(yaourt).
dessert(glace).

yes
?-
```

CProlog est un interpréteur Prolog sous Unix. Le prompt du système est `?-`, pour charger le fichier il faut utiliser la *requête* `consult('menu.pl').` La requête `listing.` permet d'avoir la liste des clauses connues par le système Prolog à un instant donné. Par convention, dans la suite du texte nous utiliserons le symbole `?-` pour indiquer qu'il s'agit d'une requête présentée à Prolog.

À partir de cette carte, plusieurs types de questions (requête) peuvent être posées :

vérification :
Il s'agit simplement de vérifier que telle ou telle chose est dans la carte (que tel ou tel fait appartient à la base). Imaginons que l'on veuille savoir s'il est possible de manger des sardines ; on pose la question :

?- hors_d_oeuvre(sardines).

réponse :	no ~

~ symbolisera la position du curseur.
Comment le système a-t-il résolu le problème posé ?
Il a tout simplement parcouru toute la base de faits de haut en bas (dans l'ordre où les clauses ont été écrites) et a essayé de faire correspondre le but courant (la requête) avec une tête de clause, comme la correspondance n'a pu se faire il a renvoyé l'atome `no` puis a positionné le curseur à la ligne suivante. Le fait qu'il est répondu par la négative signifie qu'il n'a pu établir la véracité du fait à prouver.

?- hors_d_oeuvre(crudite).

réponse :	no ~

`crudites` est un hors d'œuvre mais pas `crudite`, méfiez-vous des fautes d'orthographe ! !

Le système PROLOG fait correspondre deux termes, ceux-ci ne seront considérés comme identiques que si leurs orthographes sont identiques.

?- hors_d_oeuvre(fruits).

réponse :	no ~

Le système ne s'est pas rendu compte qu'un fruit n'est pas un hors d'œuvre son fonctionnement est toujours le même.

?- dessert(fruits).

réponse :	yes ~

questions :
On n'a aucune idée sur le menu et on voudrait connaître les diverses entrées proposées par le restaurateur. On va alors utiliser des variables :

?- hors_d_oeuvre(Entree).

réponse :	Entree=crudites ~

Le curseur, symbolisé par ~, reste sur la ligne courante, attendant de nouvelles instructions. Il y en a deux possibles soit on tape <cr> (i.e. touche return) soit on tape ; <cr> (un point-virgule suivi de return). Dans le premier cas cela signifie que la réponse nous satisfait, dans l'autre que l'on voudrait connaître (si elle existe) une autre solution.

?- hors_d_oeuvre(Entree).

réponse : réponse :	Entree=crudites <cr> yes ~

?- hors_d_oeuvre(Entree).

réponse : réponse :	Entree=crudites ; <cr> Entree=assiette_anglaise ~

On peut bien entendu itérer le processus, lorsque toutes les solutions ont été trouvées le système répond no.

?- dessert(Fromage).

réponse : réponse : réponse : réponse : réponse :	Fromage=fromage ; <cr> Fromage=fruits ; <cr> Fromage=yaourt ; <cr> Fromage=glace ; <cr> no ~

Composition de buts :
Nous allons maintenant regarder l'ordre d'évaluation des requêtes en PROLOG. Imaginons, que nous

souhaitions constituer un repas avec un hors d'œuvre, un plat principal et un dessert ; nous pouvons poser au système la requête suivante :

?- hors_d_oeuvre(Entree), plat_principal(Plat,Legume), dessert(Dessert).

réponse :	Entree = crudites Plat = saucisses Legume = lentilles Dessert = fromage ~

Comme on peut le constater, la requête réussit si la conjonction des buts la composant admet une solution. L'algorithme de résolution peut-être décrit de la façon suivante :

```
procedure Resoudre(Q : requete)
debut
 Si Q est un but unique alors chercher une solution
 Sinon
   debut
     Q est de la forme Q1,Q2,...,Qn
     P := Q2,...,Qn
     Resoudre(Q1)
     Resoudre(P)
   fin
fin
```

Comme on le voit, les buts sont résolus de gauche à doite (en anglais left to right). La requête réussit si tous les buts sont satisfaits. Intéressons-nous à un convive voulant une entrée, un plat avec de la purée et un dessert.

?- hors_d_oeuvre(Entree), plat_principal(Plat,puree), dessert(Dessert).

réponse :	no ~

Lorsque la requête initiale réussit, il est possible de demander s'il existe d'autres alternatives :

réponse :	Entree = crudites Plat = saucisses Legume = lentilles Dessert = fromage ; <cr>
réponse :	Entree = crudites Plat = saucisses Legume = lentilles Dessert = fruits ; <cr>
réponse :	Entree = crudites Plat = saucisses Legume = lentilles Dessert = yaourt ; <cr> ~

Lorsque l'on demande d'autres solutions le système PROLOG remet en cause son dernier choix, c'est ce que l'on appelle le retour arrière (en anglais *backtracking*). S'il ne reste plus aucune alternative, il remet en cause la solution obtenue pour le but précédent. Le parcours se fait donc en profondeur d'abord (en anglais depth first search).

Si on cherche à re-satisfaire un but, il faut remettre toutes les variables dans l'état où elles se trouvaient avant que le but ait été satisfait. Puis on parcourt la base de faits à partir de la position du marqueur. Comme précédement il peut y avoir échec ou succès.

8.1.3 Structures de données

L'arborescence est une structure fondamentale de PROLOG, elle permet de représenter de façon aisée les termes-composés ainsi `pere(jean,jacques)` correspondra à la figure 8.1.

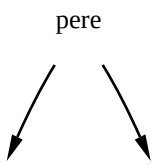


FIGURE 8.1 – Un arbre PROLOG

La liste est une autre structure privilégiée du langage. Une liste PROLOG est formée d'éléments séparés par des virgules et entourés de crochets `[` et `]`, de plus une liste a la possibilité d'être hétérogène à la différence de nombreux langages.

EXEMPLE 8.1.1

`[bonjour,marc]` *est une liste constituée de deux atomes,*
`[[PROLOG],est,[[un]],[langage,[de],[[programmation]]]]` *est une liste de quatre éléments dont seul le deuxième est un atome.* ◁

En LISP, il existe deux primitives `car` et `cdr` qui jouent le rôle de destructeurs de listes, et un constructeur privilégié `cons`. En PROLOG on dispose d'un mécanisme beaucoup plus puissant : l'*unification*. Ainsi pour obtenir la tête et/ou la queue d'une liste, il suffit d'écrire celle-ci sous la forme :

`[TETE|QUEUE]`

`[]` symbolise la liste vide (`nil`, `()` en sont les équivalents « lispiens »).

EXEMPLE 8.1.2

Écrivons le programme qui permet de savoir si un élément appartient à une liste.
Un élément appartient à une liste si c'est la tête de la liste (son premier élément) sinon on regarde s'il est dans la queue de la liste.

```
(a) elt(X,[X|_]).
(b) elt(X,[_|L]):-elt(X,L).
```
◁

Cet exemple nous permet d'illustrer par la même occasion l'utilisation un peu particulière du symbole _ (sous-ligné), qui est là pour indiquer l'existence d'un objet unique dont la valeur est inutile pour la suite des calculs. Il permet de plus un gain de mémoire puisqu'il autorise le système à ne pas reserver de place pour cette variable.

L'espace de recherche permet de bien appréhender le fonctionnement d'un programme donné. La figure 8.2 représente celui associé à la requête : `?- elt(1,[2,3,1,1,4]).`

Dans la figure 8.2 l'atome `fail` signifie l'échec de la déduction. PROLOG effectue automatiquement un retour-arrière quand il recontre cet atome et ce jusqu'à obtenir un succès symbolisé par □, à ce moment une réponse est retournée il s'agit du *mgu* (l'unificateur le plus general), et l'interpréteur attend un ordre "`<cr>`" ou "`;<cr>`".

8.1.4 Écrire en PROLOG

Dans cette section nous allons essayer de voir comment il est possible d'écrire de manière systématique des petits bouts de code en PROLOG, pour cela on cherche, dans un premier temps la version SCHEME puis on traduira étape par étape pour aboutir à la version PROLOG.

EXEMPLE 8.1.3

Commençons par la concaténation de deux listes, en SCHEME *cela donne :*

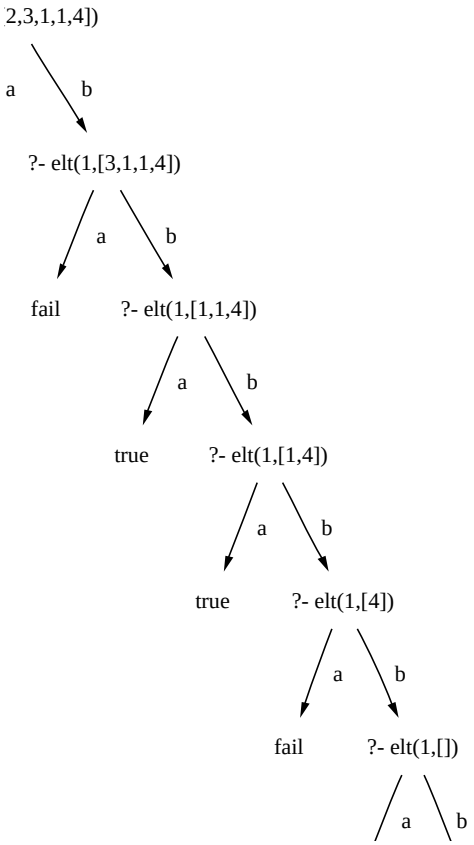


FIGURE 8.2 – Espace de recherche

```
(define (conc l1 l2)
 (if (null? l1) l2
  (cons (car l1) (conc (cdr l1 l2))))
)
```

PROLOG *étant un langage relationnel, on a besoin d'introduire un paramètre supplémentaire qui sera le résultat du calcul. Ensuite, on peut voir les clauses constitutives du programme, comme les différents cas de l'évaluation. En d'autres termes nous auront deux clauses, l'une pour le cas où* `l1` *est vide, l'autre lorsque* `l1` *n'est pas vide. Une première version serait :*

```
conc(L1, L2, L3) :- L1=[], L2=L3.
conc(L1, L2, L3) :- L1=[X | L11], L3=[X | conc(L11,L2)].
```

Bien sûr, nous sommes encore loin d'un code PROLOG. *Regardons la première clause. Grâce au mécanisme d'unification du langage on peut la réécrire en :*

```
conc([],L,L).
```

Dans la deuxième, nous avons déjà utilisé la notion d'unification puisque nous avons remplacé les notions de `car` *et* `cdr` *par leur équivalent, pourtant l'écriture de* `L3` *n'est pas encore correcte, la notion de fonction étant absente de* PROLOG, *nous devons remplacer la queue de* `L3` *par une forme plus orthodoxe :*

```
conc([X | L1], L2, L3) :- conc(L1,L2,L4), L3=[X | L4].
```

et maintenant la version définitive :

```
conc([],L,L).
conc([X | L],L2,[X | L3]) :- conc(L,L2,L3).
```

◁

EXEMPLE 8.1.4

Dans ce deuxième exemple, nous allons écrire l'inversion d'une liste, d'une part de manière naïve (en utilisant la concaténation), puis de manière plus efficace grâce à la récursivité terminale. Commençons par les écritures en SCHEME.

```
;; version naïve de l'inversion
(define (nrev l)
 (if (null? l) l
  (conc (nrev (cdr l)) (list (car l))))
)

;; version basée sur la récursivité terminale
(define (rev l)
 (define (aux afaire fait)
  (if (null? afaire) fait
   (aux (cdr afaire) (cons (car afaire) fait)))
 )
 (aux l '())
)
```

Nous laissons de côté la vérification de la validité de nos programmes en SCHEME *pour nous concentrer sur leur traduction en* PROLOG. *Et tout d'abord la version de* `nrev`.

```
nrev([],[]).
nrev([X | L],conc(nrev(L),[X])).
```

à nouveau, comme dans le mode relationnel, la fonctionnelle de la deuxième clause doit être remplacée par un littéral dont l'argument résultat sera le résultat de l'inversion ; d'où l'écriture suivante :

```
nrev([],[]).
nrev([X | L],L2) :- nrev(L,L1), conc(L1,[X],L2).
```

Pour la version récursive terminale, nous appliquons la même démarche afin d'aboutir à la forme suivante :

```
rev(L1,L2) :- local(L1,[],L2).
local([],L,L).
local([A | Faire], Fait, Res) :- local(Faire, [A | Fait], Res).
```

◁

Terminons cette partie avec un exemple de parcours de graphe.

EXEMPLE 8.1.5

On cherche à établir s'il existe un chemin entre deux sommets d'un graphe. Dans un premier temps, nous devons coder le graphe. Afin de clarifier l'exposé, supposons que l'on ait $G = \langle\{a,b,c,d\},\{(a,b),(b,c),(c,d),(c,a)\}\rangle$. *Pour coder le graphe, il suffira de coder les arcs, ce qui donnera :*

```
/* arc(source,but) */
arc(a,b).
arc(b,c).
arc(c,d).
arc(c,a).
```

On peut alors définir la notion de chemin, en indiquant qu'un chemin entre deux sommets existe si, soit il existe un arc entre ces deux sommets, soit il existe un sommet intermédiaire, tel que l'on ait un arc du sommet initial vers ce sommet intermédiaire, et un chemin de ce dernier vers le but fixé. En d'autres termes :

```
path(X,Y) :- arc(X,Y).
path(X,Y) :- arc(X,Z), path(Z,Y).
```

◁

Tout semble aller pour le mieux, hélas, si nous insérons une boucle dans le graphe, notre programme risque de ne plus terminer, supposons que l'on rajoute une boucle sur b, et que l'on insère cette information au milieu de notre base de faits :

```
/* arc(source,but) */
arc(a,b).
arc(b,b).
arc(b,c).
arc(c,d).
arc(c,a).
```

Si nous cherchons un chemin entre a et d, nous n'obtiendrons jamais de réponse, dommage .... Une première idée, consiste à se souvenir par quel sommet nous sommes passés :

```
path(X,Y,[]) :- arc(X,Y).
path(X,Y,[Z | L]) :- arc(X,Z), path(Z,Y,L).
```

Cette mémorisation n'est pas suffisante puisqu'aucun contrôle sur les sommets mémorisés n'est effectué. Il faudrait que l'on puisse connaître les sommets visités, afin d'éviter d'y retourner. Mais pour cela nous devons introduire quelques nouvelles notions ...

8.1.5 Le coupe choix

Le coupe choix (en anglais « cut ») est un autre mécanisme fondamental de PROLOG. Dans un programme PROLOG le coupe choix est représenté par le symbole !. Il permet d'indiquer au système quels sont les choix qui ne sont pas à reconsidérer lors d'un retour arrière. Le cut a deux effets importants :

- Exécution plus rapide, puisque certaines options sont bloquées.
- Gain d'espace mémoire, du fait du nombre moins important de renseignements à conserver (les points de choix sont supprimés).

Le coupe choix peut même s'avérer indispensable au fonctionnement d'un programme, soit pour lui éviter de boucler, soit lorsqu'il s'agit d'un logiciel assez volumineux (au sens de l'arbre de recherche) pour supprimer des points de choix inutiles.

Le cut peut être vu comme un but qui réussit toujours, mais qui empêche de revenir sur les choix antérieurs qui ont été effectués lors de la résolution.

Voyons ensemble quelques exemples de son utilisation.

Limitation des choix :
On veut que la procédure que l'on écrit soit déterministe.

EXEMPLE 8.1.6

Supposons que l'on veuille écrire un prédicat qui retourne le prédécesseur d'une valeur d'entrée s'il existe, et `erreur` *sinon :*

```
pred(X,erreur):-X=<0.
pred(X,X1):-X1 is X-1.
```

Si on écrit ce programme et qu'on lance la requête :

```
?- pred(0,M).
```

réponse :	`M=erreur ; <cr>`
réponse :	`M=-1 ; <cr>`
réponse :	*no*
	~

Alors qu'en remplacant la première clause par :

```
pred(X,erreur) :- X=<0,!.
```

On obtient, pour la même requête :

réponse :	`M=erreur ; <cr>`
réponse :	*no*
	~

qui correspond à ce que l'on souhaitait. ◁

Position du cut :

Dans cette partie, nous allons regarder quels sont les effets du cut en terme d'espace de recherche, en fonction de sa position dans une conjonction de littéraux. Considérons trois versions très proches d'une même procédure sans signification particulière.

```
p1([],a).
p1([X|L],M) :- p1(L,M).
p1([],b).

p2([],a).
p2([X|L],M) :- !, p2(L,M).
p2([],b).

p3([],a).
p3([X|L],M) :- p3(L,M),!.
p3([],b).
```

Nous allons visualiser cet espace sous forme de trace, outil qui existe dans tout système PROLOG.

```
?- p1([1,2],M).
   (1) 1 Call: p1([1,2],_5) ?
   (2) 2 Call: p1([2],_5) ?
   (3) 3 Call: p1([],_5) ?
   (3) 3 Exit: p1([],a)
   (2) 2 Exit: p1([2],a)
   (1) 1 Exit: p1([1,2],a)

M = a ;
   (3) 3 Back to: p1([],_5) ?
   (3) 3 Exit: p1([],b)
   (2) 2 Exit: p1([2],b)
   (1) 1 Exit: p1([1,2],b)

M = b ;
```

```
   (3) 3 Back to: p1([],_5) ?
   (3) 3 Fail: p1([],_5)
   (2) 2 Fail: p1([2],_5)
   (1) 1 Fail: p1([1,2],_5)

no
?- p2([1,2],M).
   (1) 1 Call: p2([1,2],_5) ?
   (2) 2 Call: p2([2],_5) ?
   (3) 3 Call: p2([],_5) ?
   (3) 3 Exit: p2([],a)
   (2) 2 Exit: p2([2],a)
   (1) 1 Exit: p2([1,2],a)

M = a ;
   (3) 3 Back to: p2([],_5) ?
   (3) 3 Exit: p2([],b)
   (2) 2 Exit: p2([2],b)
   (1) 1 Exit: p2([1,2],b)

M = b ;
   (3) 3 Back to: p2([],_5) ?
   (3) 3 Fail: p2([],_5)

no
?- p3([1,2],M).
   (1) 1 Call: p3([1,2],_5) ?
   (2) 2 Call: p3([2],_5) ?
   (3) 3 Call: p3([],_5) ?
   (3) 3 Exit: p3([],a)
   (2) 2 Exit: p3([2],a)
   (1) 1 Exit: p3([1,2],a)

M = a ;

no
```

Comme on peut le constater la procédure `p1` permet d'obtenir les deux réponses `M=a` et `M=b`. Il y a, en effet deux solutions à la requête `p1([],M)`. Pour la procédure `p2`, on obtient les deux mêmes solutions, le coupe-choix n'a aucune influence, sur l'appel `p2([],M)`. Son effet ne se fait sentir que sur les alternatives portant sur une liste non vide. Par contre, la procédure `p3` se comporte différemment, le gel est effectué sur tous les sous-buts ayant emprunté la deuxième clause. La procédure `p3` est déterministe, si le premier argument est une liste non vide.

Optimisation des calculs :
Considérons la procédure `conc` suivante :

```
conc([],L,L).
conc([X|L],L1,[X|L2]):-conc(L,L1,L2).
```

Cette procédure peut être utilisée de multiples façons, soit on lui donne deux listes à concaténer (le résultat se trouvant dans la troisième liste), soit on lui peut lui demander de générer plusieurs sous-listes d'une liste.
Nous ne considèrerons ici que le premier cas, la procédure est alors déterministe du fait qu'une liste ne peut être simultanément vide et non vide. Cette propriété est connue d'un utilisateur humain, pas du système i.e. il conservera des points de choix inutiles (alternatives qui seront stockées dans diverses piles), pour permettre au système de minimiser l'occupation mémoire il suffit de mettre un coupe choix dans les corps de clauses.

```
conc1([],L,L):- !.
conc1([X|L],L1,[X|L2]):- conc1(L,L1,L2).
```

Le fait d'avoir rajouté ce « cut »interdit à conc1 de jouer le rôle de générateur de listes à la différence de conc. Dans l'exemple 8.1.7 nous présentons une session utilisant conc et conc1.

EXEMPLE 8.1.7

```
?- conc([1,2],[a,b],R).
R=[1,2,a,b];
no

?- conc1([1,2],[a,b],R).
R=[1,2,a,b];
no

?- conc(L,M,[1,2,a,b]).
L=[]
M=[1,2,a,b];

L=[1]
M=[2,a,b];

L=[1,2]
M=[a,b];

L=[1,2,a]
M=[b];

L=[1,2,a,b]
M=[];
no

?- conc1(L,M,[1,2,a,b]).
L=[]
M=[1,2,a,b];
no

?- conc(X,[Y,Y|L],[1,2,2,1,4,3,a,x,x,x,a,s]).
X=[1]
Y=2
L=[1,4,3,a,x,x,x,a,s];

X=[1,2,2,1,4,3,a]
Y=x
L=[x,a,s];

X=[1,2,2,1,4,3,a,x]
Y=x
L=[a,s];

no

?- conc1(X,[Y,Y|L],[1,2,2,1,4,3,a,x,x,x,a,s]).
X=[1]
Y=2
L=[1,4,3,a,x,x,x,a,s];

no
```

8.1.6 Negation as failure

On est parfois amené en Prolog à considérer un succès lorsqu'une requête a échoué. C'est pourquoi, PROLOG permet d'utiliser le prédicat `fail`. Malheureusement, le fonctionnement par défaut d'un système PROLOG, est d'effectuer un retour-arrière lorsqu'il rencontre un échec, l'utilisation du cut, va nous permettre de geler cette possibilité. Imaginons que nous souhaitions détecter si un élément est absent d'une liste, on pourra alors écrire :

```
notin(X,L) :- elt(X,L),!, fail.
notin(_,_).
```

Si on essaye de donner une sémantique à ces clauses, on s'aperçoit que la première génèrera un échec, si l'élément apparaît effectivement dans la liste, la seconde clause est là pour récupérer justement (avec succès) les cas où l'élément est absent de la liste. L'inconvénient majeur de cette approche est de nous forcer à écrire une version positive et une version négative pour les prédicats. C'est pourquoi nous utiliserons, le *méta-prédicat* suivant :

```
non(X) :- call(X),!, fail.
non(_) .
```

Il est à noter, que suivant la version de PROLOG utilisée ce méta-prédicat est déjà défini, mais comme sur le marché, on ne trouve que des implémentations en langue anglaise, je vous conseille d'utiliser la forme francisée telle que définie plus haut, et ce afin d'avoir des programmes qui fonctionnent indépendamment du logiciel utilisé.

8.1.7 Prolog et l'arithmétique

Tout langage qui se respecte doit permettre de faire des calculs, malheureusement ceci ne se fait pas sans quelques acrobaties si le langage est spécialisé pour certains types de traitement. Les programmeurs COBOL savent ce qu'il en coûte de vouloir faire une addition. Le langage PROLOG dans sa version de base est sujet aux mêmes douloureuses contorsions.

8.1.8 Arithmétique selon Péano

Le langage PROLOG manipule des termes, l'idée consiste à exprimer l'arithmétique comme un algèbre de termes, et ce grâce à l'arithmétique de Péano qui ne considère que deux objets, `0` et le foncteur `s` pour successeur. Dans ce cadre on peut définir, l'addition par la procédure :

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).
```

Pour la multiplication, l'opération est un rien plus complexe.

```
mult(0,_,0)
mult(s(0),X,X).
mult(s(X),Y,Z) :- mult(X,Y,W), add(Y,W,Z).
```

Les autres opérations arithmétiques, telles que la division entière et la soustraction se révèle particulièrement délicates à implémenter. On pourrait imaginer d'introduire les entiers négatifs par le biais d'un foncteur `p` pour le prédecesseur, mais on serait alors obligé d'effectuer une normalisation (simplification) des termes pour éviter des séquences de `s` et `p`. En effet le terme `s(s(p(s(p(p(s(0)))))))` correspond au terme simplifié `s(0)`.

8.1.9 Arithmétique et prédicat extra-logique

L'alternative à l'approche précédente consiste à introduire dans le langage des prédicats extra-logiques, au sens où les variables manipulées dans ce contexte ne peuvent plus évoluer par le biais de l'unification. Les opérateurs arithmétiques classiques (+, *, /, -) interviennent sont manipulés par l'intermédiaire du prédicat `is` qui correspond à l'affectation dans les langages impératifs. Ce prédicat est déterministe, non réversible et s'utilise de la façon suivante :

Var_Libre `is` Expression_arithmétique_close

L'inconvénient est que l'on a un prédicat déterministe qui provoque un échec si les conditions d'évaluation ne sont pas réunies (membre gauche une variable libre, membre droit une expression complètement évaluée), ainsi les expressions :

```
5 is 2 + 3.
5 is 2 + X.
```

provoqueront un échec.

Dans l'exemple suivant, nous allons voir comment calculer la longueur d'une liste.

Exemple 8.1.8

```
longueur([],0).
longueur([X|L],Y) :- longueur(L,Z), Y is Z + 1.
```

Il est important de noter que l'on n'aurait pas pu écrire sous la forme suivante :

```
longueur([],0).
longueur([X|L],Y) :- Y is Z+1, longueur(L,Z).
```

De plus, comme le prédicat `is` *n'est pas réversible, il est impossible de générer, à l'aide de cette version, une liste de longueur 5.* ◁

Exercice 8.1

Écrire un programme qui permette de vérifier, pour une liste donnée sa longueur, et pour une longueur donnée, une liste ayant le bon nombre d'éléments. ▷

Pour résoudre cette difficulté, on a cherché très tôt à étendre le langage, et dans le cadre de l'arithmétique, on s'est tourné vers la notion de contraintes (voir la section 8.3 pour une première approche).

8.2 Exercices

Il ne semble pas raisonnable de vouloir apprendre un langage de programmation sans se donner les moyens de le pratiquer, l'objectif de cette section est de vous faire programmer quelques petits exemples jouets.

Exercice 8.2

à partir des éléments donnés, trouvez une solution au problème abordé à la fin de l'exemple 8.1.5 page 95. ▷

Exercice 8.3

On écrira l'algorithme de Bellmann (recherche de plus court chemin) en Prolog, dont on rappelle l'algorithme tel qu'il est vu dans le module de Recherche Opérationnelle de la filière MASS.

Soit un réseau R=(X,A,d) sans circuit, s un sommet initial
début
  $S \longleftarrow \{s\}$
  $\pi(s) \longleftarrow 0$
  $\mathcal{A}(s) \longleftarrow \epsilon$
  Tant qu'il existe $x \notin S$ dont tous les prédécesseurs sont dans S faire
    $\pi(x) = \text{Min}_{\{a \ / \ T(a)=x\}}[\pi(I(a)) + d(a)]$
    soit $\overline{a}$ tel que $\pi(x) = \pi(\overline{(a)}) + d(\overline{a})$
    $A(x) = \overline{a}$
    $S \longleftarrow S \cup \{x\}$
  Fait
fin

▷

Exercice 8.4

Trois missionnaires et trois cannibales se trouvent sur le bord d'une rivière et disposent d'une barque à deux places. Il faut transporter tout le monde (sain et sauf) sur l'autre rive. La condition à respecter est la suivante : « si le nombre de cannibales excède le nombre de missionnaires sur une rive, alors les missionnaires présents sur cette rive sont dévorés ».

Ce problème est à rapprocher de la recherche d'un chemin dans un labyrinthe, on revient en arrière si la solution choisie mène à un échec partiel.
En effet, on a une condition initiale, et un but à atteindre, il suffit de trouver la suite des états menant de l'un à l'autre. Il y aura échec partiel si on retombe dans un état antérieur. La suite des états sera stockée dans une liste, étant donné qu'il est plus facile d'ajouter un élément en tête de liste, il faudra, lorsqu'on aura atteint l'état final, inverser la liste des solutions.

Un état sera défini par un triplet (concrétisé par une liste de 3 éléments) indiquant respectivement le nombre de missionnaires, le nombre de cannibales qu'il y a sur la rive où se trouve la barque.
`ga` sera l'abréviation pour "rive gauche" et `dr` correspondra à "rive droite". Ainsi `[2,2,ga]` signifie que deux missionaires et deux cannibales sont sur la rive gauche avec la barque.

Une étape de calcul liera deux états consécutifs viables, cette relation sera définie en fonction du nombre d'individus qu'il y a sur la rive considérée et du nombre de cannibales et de missionnaires transférés d'une rive à l'autre.

`traverse` : est la relation qui, à partir du nombre de cannibales et du nombre de missionnaires, va fournir le nombre et la catégorie des personnes transférées.
`barque` : est la contrainte sur la capacité de transfert de la barque.
`viable` : permet de savoir si pour une rive la situation est viable ou non.
`essai` : va chercher à rajouter une nouvelle étape si on n'est pas dans l'état final caractérisé par `[3,3,ga]`.
`canib_missio` : est le prédicat initial par lequel on accède au programme. ▹

Exercice 8.5

Il s'agit ici de résoudre un problème de coloration de graphe. Étant donné un graphe non orienté, l'objectif est d'associer à chaque sommet une couleur, sous la contrainte que deux sommets adjacents (i.e. il existe une arête les reliant) possèdent des couleurs différentes. Par ailleurs, on sait que si le graphe est planaire, 4 couleurs suffisent pour colorier le graphe. On considèrera, le graphe suivant $G = \langle [1, 15], \{i, j \text{ tq } j = 1 + i\} \cup \{i, j \text{ tq } i = 2k, j = k\} \cup \{i, j \text{ tq } j = 2k + 1, i = (k \bmod 15) + 1\} \rangle$ ▹

8.3 Les domaines finis dans Prolog

L'idée d'étendre le langage par le biais de contraintes, s'est fait jour dès que les utilisateurs se sont aperçus que l'algèbre des termes ne permettait pas d'exprimer simplement des algorithmes mettant en jeu des calculs numériques. Certes, il est toujours possible de faire appel dans un programme à un programme écrit dans un autre langage, mais le souhait des utilisateurs était de pouvoir garder la même fléxibilité, le même type de résolution (logique) quelque soit la nature du problème abordé. Les premiers travaux ont été menés dans le but de fondre dans un même formalisme, programmation logique et programmation fonctionnelle. Ces langages hybrides (LisLog, FunLog, LogLisp, ...) étaient en fait des extensions plus ou moins réussies d'un des deux formalismes et ne satisfaisaient aucun des spécialistes des deux communautés. Au début des années 1980, une nouvelle approche qui tend à englober les travaux de la communauté "CSP" issue de la Recherche Opérationnelle, considère Prolog sous l'aspect d'un langage basé sur la Logique et muni d'un résolveur ad-hoc pour les termes. L'idée consiste à garder le langage, mais à y intégrer d'autres modules plus à même de résoudre des équations sur d'autres objets (numériques par exemple), la PLC (Programmation Logique par Contraintes) est née. Après des débuts difficiles, la communauté Prolog a totalement adhéré à ce concept.

8.3.1 Les méthodes de résolution

Dans ce paragraphe, nous survolerons les différents algorithmes de résolution de problèmes de satisfaction de contraintes. Une excellente présentation de ces techniques est donnée dans [96]. Cet ouvrage est *la* réference sur l'introduction de variables prenant valeur sur des domaines finis au sein de Prolog. Le lecteur désireux d'explorer ce domaine pourra se reporter avec bonheur aux auteurs suivants : Pascal van Hentenryck [96, 97], Joxan Jaffar [49], Thom Frühwirth [37] et Luděk Matyska [66].

Vocabulaire

Un problème de satisfaction de contraintes (en anglais CSP pour Constraints Solving Problem) est la donnée :

- D'un ensemble de variables $\{x_1, x_2, \ldots, x_n\}$ et pour chaque variable x_i d'un domaine de variation D_i qui est l'ensemble **fini** des valeurs que x_i peut prendre.
- D'un ensemble de contraintes $\{C_1, C_2, \ldots, C_m\}$ portant sur les x_i, chaque C_j décrivant implicitement un sous-ensemble de $D_1 \times D_2 \times \ldots \times D_n$.

Cette définition est très large et permet d'appréhender un grand nombre de CSP. En pratique, les domaines introduits dans Prolog sont l'algèbre de Boole et la programmation linéaire en nombres entiers.

Résoudre un tel problème revient à déterminer les affectations des variables qui satisfont les contraintes, ou, autrement dit, déterminer l'intersection des sous-ensembles de $D_1 \times \ldots \times D_n$ que décrivent les contraintes. Les méthodes de résolution varient bien entendu en fonction du domaine considéré. Cependant, il est possible de présenter ces techniques en restant à un niveau très général.

Afin d'illustrer cette présentation, nous appliquerons ces méthodes sur l'exemple suivant (utilisant des contraintes sur des nombres entiers) :

$x_1 \in \{0, 2, 5\}$ $x_2 \in \{0, 1, 2\}$ $x_3 \in \{3, 4, 5\}$ $x_4 \in \{1, 2, 3\}$

$x_1 + x_2 \geq 3$
$x_1 + x_3 \leq 7$
$x_3 + x_3 \geq 5$
$x_3 \leq x_4$

Toutes les méthodes que nous présentons ici sont énumératives, c'est-à-dire qu'elles recherchent les solutions du CSP en affectant les variables aux différentes valeurs de leurs domaines de définitions. Il existe d'autres méthodes de résolution, mais elles sont en général dédiées à un domaine particulier et ne sont pas forcément plus efficaces que les méthodes énumératives. L'affectation des variables se fait en construisant un arbre de démonstration, c'est-à-dire un arbre dont les nœuds sont étiquetés par les x_i (une variable apparaissant au plus une fois dans chaque branche) et dont les arêtes partant d'un nœud étiqueté par la variable x sont étiquetées par les différents élements de son domaine de variation.

Generate and test

La première méthode de résolution c'est le "générer et tester", méthode naïve consistant à construire une affectation $x_1 \leftarrow v_1, \ldots, x_n \leftarrow v_n$ de toutes variables puis à vérifier si elle satisfait les contraintes. C'est la méthode de Prolog "pur".

Sur notre exemple, l'algorithme de generate-and-test parcourt donc les $3 \times 3 \times 3 \times 3 = 81$ affectations possibles pour découvrir la seule solution du système ($x_1 = 2, x_2 = 2, x_3 = 3, x_4 = 3$).

Standard backtracking

Une amélioration substantielle consiste à tester, après chaque affectation, si les contraintes dont toutes les variables sont instanciées sont vérifiées. C'est le backtracking standard.

Sur notre exemple, après avoir affecté x_1 et x_2 à 0, l'algorithme constate que la contrainte $x_1+x_2 \geq 3$ n'est pas vérifiée et ne parcourt donc pas le sous-arbre de recherche correspondant. Cela permet de réduire le nombre d'affectations étudiées de 81 à 25.

Dans la mesure où l'on s'arrête avant d'avoir affecté toutes les variables, il est intéressant de choisir dynamiquement l'ordre dans lequel on affecte ces variables. Il existe de nombreuses heuristiques pour guider ce choix. Nous ne pouvons nous livrer ici à une étude exhaustive de ces heuristiques. Toutefois, on peut souligner que les critères les plus souvent retenus sont la taille des domaines de variations (mis à jour dynamiquement) — choisir la variable qui a le plus petit domaine — et le nombre de contraintes dans lesquelles apparaît la variable — choisir la variable la plus contrainte. Ces heuristiques permettent d'augmenter considérablement les performances des algorithmes (nous avons pu constater des facteurs 1000 et plus pour la résolution de contraintes booléennes).

Forward checking

Le forward-checking (regarder en avance) consiste à vérifier, après chaque affectation, toutes les contraintes dans lesquelles la variable affectée apparaît. Cela permet de réduire les domaines de variations des variables non encore affectées qui apparaissent dans ces contraintes.

Sur notre exemple, après avoir affecté la variable x_1 à la valeur 5, la contrainte $x_1 + x_3 \leq 7$ permet de restreindre le domaine de x_3 à l'ensemble vide. Ce qui évite de parcourir le sous-arbre de recherche correspondant (et évite 8 affectations). Avec le forward-checking, on ne fait plus 13 affectations.

Looking ahead

Le look-ahead checking consiste à vérifier après chaque affectation que toutes les contraintes sont encore satisfiables (par d'autres moyens que l'affectation des variables). Dans sa généralité, ce programme est trop ambitieux pour être réalisable en pratique. On peut cependant faire un look-ahead checking partiel consistant par exemple à mettre à jour les minimums et les maximums de chaque domaine de variation (pour la programmation en nombres entiers).

Sur notre exemple, cette mise à jour consisterait, avant toute affectation à remarquer que la contrainte $x_3 \leq x_4$ implique que la valeur maximum que peut prendre la variable x_3 est forcément inférieure à $max(D_4)$, et donc que $x_3 = x_4 = 3$. Partant, la contrainte $x_2 + x_3 \geq 5$ nous donnerait la valeur de x_2 (2), la contrainte $x_1 + x_3 \leq 7$ réduirait le domaine de x_1 à $\{0, 2\}$, et finalement, la contrainte $x_1 + x_2 \geq 3$ nous donnerait l'unique valeur possible pour x_1 (2).

Comparaison

La progression generate-and-test, standard-backtracking, forward-checking, (partial) looking-ahead décrit le traitement de plus en plus important effectué sur le système de contraintes après chaque affectation. Le choix de la méthode dépend du domaine et de l'efficacité des traitements : "il ne faut pas faire rentrer par la fenêtre ce que l'on fait sortir par la porte".

Les tables suivantes indiquent les performances obtenues sur le problème des reines avec la surcouche de `Sicstus`.

standard backtracking			
reines	solutions	affectations	temps cpu
8	92	2056	67
9	352	8393	279
10	724	35538	1499

forward checking			
reines	solutions	affectations	temps cpu
8	92	1360	50
9	352	5399	180
10	724	19744	855

8.3.2 Le problème des N-reines

Dans cette section, nous mettons en œuvre les differents algorithmes présentés plus haut sur le problème des N-reines. Il s'agit de placer, dans chaque colonne d'un échiquier $N \times N$ une reine de telle sorte qu'elle ne soit pas en prise avec les autres (i.e. ni sur la même ligne, ni sur une même diagonale). Afin d'illustrer notre propos, nous considérons le cas où $N = 4$. Le problème peut se ramener au système d'équations suivants :

$$\begin{cases} x_i \in D_i & \forall i, 1 \leq i \leq 4 \\ D_i = \{1, 2, 3, 4\} & \forall i, 1 \leq i \leq 4 \\ x_i \neq x_j & \forall i, j, 1 \leq i < j \leq 4 \\ x_i \neq x_j + (j - i) & \forall i, j, 1 \leq i < j \leq 4 \\ x_i \neq x_j - (j - i) & \forall i, j, 1 \leq i < j \leq 4 \end{cases}$$

Où, x_i correspond à la ligne occupée par la reine de la colonne i. La situation initiale peut être visualisée graphiquement par la figure 8.3 :

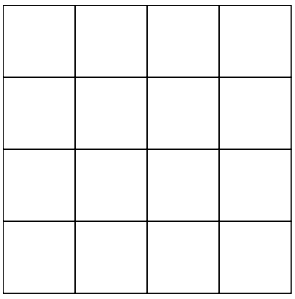


FIGURE 8.3 – Échiquier initial

Quant à la première solution elle est décrite par la figure 8.4 :

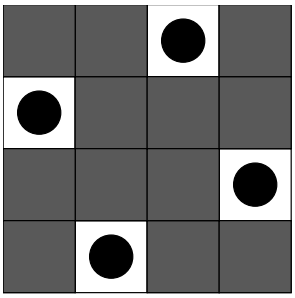


FIGURE 8.4 – Première solution

Nous nous restreindrons au deux premières itérations de chaque algorithme.

generate-and-test :
à la première itération, l'algorithme choisit pour x_1 la première valeur du domaine, $v_1 = 1$. à la deuxième itération, il choisit la première valeur pour x_2, soit $v_2 = 1$.

standard-backtracking :
à la première itération, l'algorithme choisit pour x_1 la première valeur du domaine, $v_1 = 1$. Comme aucune autre affectation n'a été faite aucun contrôle n'est effectué (cf. figure 8.5).

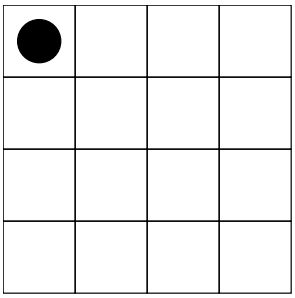


FIGURE 8.5 – Algorithme standard-backtracking

à la deuxième itération, il prend pour x_2, la première valeur, $v_2 = 1$ et vérifie que cette affectation est compatible avec celles sur x_1. L'équation $x_1 \neq x_2$ permet de rejeter cette affectation. L'algorithme standard-backtracking choisit alors, $v_2 = 2$, qui à son tour est rejetée à cause de $x_1 \neq x_2 + 1$. Ensuite, l'algorithme sélectionne $v_2 = 3$ qui ne pose pas de problème.

forward-checking :
à la première itération, l'algorithme choisit pour x_1 la première valeur du domaine, $v_1 = 1$. La propagation des contraintes, conduit à réduire les domaines comme suit : $D_2 = \{3, 4\}$, $D_3 = \{2, 4\}$, $D_4 = \{2, 3\}$. Ce qui, d'un point de vue graphique correspond à la situation de la figure 8.6 :

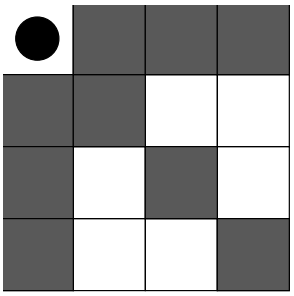


FIGURE 8.6 – Algorithme forward-checking

à la deuxième itération, l'algorithme sélectionne pour x_2 la valeur $v_2 = 3$, puis il propage les contraintes et obtient le système visualisé figure 8.7 :

L'affectation est rejetée, car $D_3 = \emptyset$. L'algorithme de forward-checking choisit alors $v_2 = 4$ ce qui donne la nouvelle situation (cf. figure 8.8) :

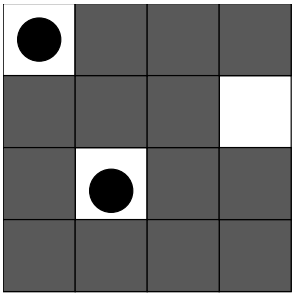


FIGURE 8.7 – Algorithme forward-checking : $v_2 = 3$

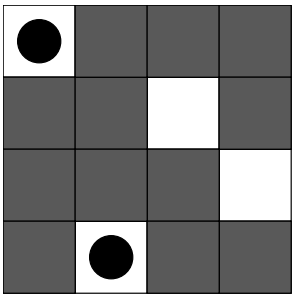


FIGURE 8.8 – Algorithme forward-checking : $v_2 = 4$

looking-ahead :
L'algorithme choisit pour x_1 l'affectation $v_1 = 1$, puis il propage les contraintes. Celles correspondant au forward-checking, sont valides. L'algorithme doit ensuite étudier les ensembles de contraintes suivants :

$$\begin{cases} l & = & 2 \\ x_2 & \neq & x_3 \\ x_2 & \neq & x_3 + 1 \\ x_2 & \neq & x_3 - 1 \\ x_2 & \neq & x_4 \\ x_2 & \neq & x_4 + 2 \\ x_2 & \neq & x_4 - 2 \end{cases}$$

Ce premier ensemble de contraintes est satisfaisable.
Puis, il considère le système :

$$\begin{cases} l & = & 3 \\ x_2 & \neq & x_3 \\ x_2 & \neq & x_3 + 1 \\ x_2 & \neq & x_3 - 1 \\ x_3 & \neq & x_4 \\ x_3 & \neq & x_4 + 1 \\ x_3 & \neq & x_4 - 1 \end{cases}$$

Ce système n'étant pas satisfaisable, $v_1 = 1$ est rejeté, il choisit alors $v_1 = 2$, après vérification la valeur est gardée. Ce qui est représentée par la figure 8.9 :

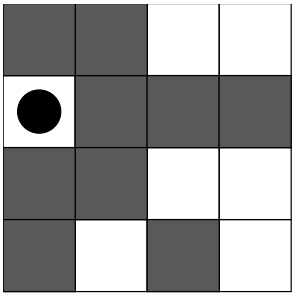


FIGURE 8.9 – Algorithme looking-ahead

à la deuxième itération, le système choisit pour x_2, la seule valeur possible, soit $v_2 = 4$, puis il propage les informations, ce qui a pour effet de réduire les domaines $D_3 = \{1\}$ et $D_4 = \{1, 3\}$ (cf. figure 8.10) :

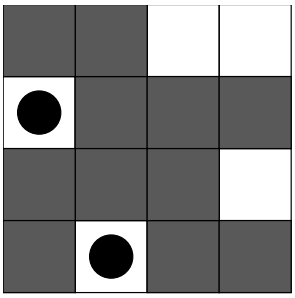


FIGURE 8.10 – Algorithme looking-ahead : $v_2 = 4$

Comme on peut le constater sur cet exemple, plus l'algorithme est complexe moins il essaye d'alternatives sans avenir. Mais, comme toujours, plus un algorithme est complexe, plus les calculs mis en jeu sont lourds. Nous allons voir maintenant d'autres approches permettant d'améliorer l'efficacité de l'exploration.

Backtracking intelligent

Les techniques de retour-arrière intelligent permettent de revenir non pas au dernier point de choix (à la dernière affectation), mais à l'affectation qui est cause de l'échec.

Sur notre exemple, après avoir affecté x_1 à 5, x_2 à 0 et essayé les trois valeurs possibles pour x_3, on aurait pu se rendre compte que l'affectation de la variable x_2 n'était pour rien dans l'échec et revenir directement sur l'affectation de x_1 à 5.

Nous n'en dirons pas plus ici, ces techniques, dans le cadre des contraintes ne sont pas largement répandues. Le lecteur intéressé est renvoyé aux thèses de doctorat de Philippe et Christian Codognet [23, 22].

Branch-and-Bound

La plupart des problèmes que les utilisateurs de PROLOG avec contraintes souhaitent pouvoir résoudre sont des problèmes d'optimisation combinatoire. Il s'agit de déterminer non seulement si un système de contraintes donné admet des solutions, mais encore la ou les solutions optimales selon certains critères (en général, une fonction linéaire des variables).

Les algorithmes décrits dans les paragraphes précédents permettent d'effectuer ce type de traitement : il suffit de noter chaque solution et de ne retenir que la ou les meilleures. On peut cependant améliorer ce mécanisme en estimant (minorant) le coût des solutions constructibles à partir de chaque affectation partielle. Ainsi, si le coût de toute solution constructible en dessous d'un certain nœud de l'arbre de recherche est supérieur à celui d'une solution trouvée précédement, on pourra éviter de parcourir le sous-arbre correspondant.

Enfin, on peut modifier la stratégie d'exploration de l'arbre (jusque là en profondeur d'abord) afin de tirer le meilleur parti de la remarque précédente. Le point de reprise (ou de backtracking) n'étant donc plus forcément le dernier point de choix.

Cette technique est dite de séparation et évaluation (Branch-an-Bound), elle est traitée plus en profondeur dans le cours de Recherche Opérationnelle de la maîtrise MASS [28].

8.4 Bilan

Comme nous avons essayé de le montrer dans la partie précédente, l'introduction des contraintes en Programmation Logique nécessite de remplacer l'unification par une procédure de décision déterminant si une contrainte ou un ensemble de contraintes est satisfaisable. Cette procédure de décision est appelée *résolveur de contraintes* (en anglais "constraints solver"). Ce résolveur est dit *complet* s'il est capable de décider de la satisfaisabilité de n'importe quel ensemble de contraintes appartenant au domaine choisi. Nous verrons que ce n'est pas le cas pour CLP($\mathcal{R}$). Qui plus est afin d'être efficace, il faut que ce résolveur soit incrémental, c'est-à-dire que lorsque l'on rajoute une contrainte dans l'entrepôt des contraintes (traduction approximative de "constraints store"), il utilise les calculs/informations antérieurs.

8.5 Contraintes Numériques : CLP($\mathcal{R}$)

L'introduction des contraintes est un gain manifeste dans la puissance d'expression du langage, pour vous en convaincre, revenons sur la manipulation des entiers dans le cadre de l'arithmétique de Peano afin de permettre une certaine réversibilité (ce que ne permet pas le prédicat extra-logique `is` de PROLOG) :

```
add(0,N,N):-!.
add(s(X),Y,s(Z)):- add(X,Y,Z).
```

La requête `:- add(X,Y,Z), add(X,Y,s(Z)).` provoque un "core dump" à l'exécution, alors qu'en CLP($\mathcal{R}$) qui intègre un solveur arithmétique la requête `X+Y=Z, X+Y=Z+1` produit la réponse (espérée) `No`.

Il existe deux types de contraintes numériques, les premières dites linéaires sont traitées efficacement, les secondes dites non-linéaires sont plus délicates.

8.5.1 Contraintes linéaires

On autorise toutes les opérations arithméthiques usuelles, mais dans une multiplication on ne permet qu'une variable au plus, de même dans une division, le dénominateur est une constante. Les opérateurs de comparaison sont $\{=, \neq, <, >, \leq, \geq\}$.

Satisfaisabilité

Décider de la satisfaisabilité peut se faire en temps polynomial, un résultat de 1979 du à Kachian (pour l'égalité seulement), et son extension à $\{\neq, >, <\}$ connu depuis au moins 1988 (Lassez–Maher).

Algorithmes :
Plusieurs candidats, d'une part le simplex, qui est exponentiel dans le pire cas, mais polynomial en moyenne, d'autre part l'algorithme de Kachian, voire de Karmakan (1988) est polynomial mais le rendre incrémental n'est pas chose aisée.

Méthode choisie
Il s'agit le plus souvent d'une généralisation du simplex que l'on peut décomposer de la manière suivante :

1. Pour les équations $a + a_1x_1 + \ldots + a_nx_n = 0$, on choisit une variable x_1 par exemple et on transforme l'équation en
$$x_1 = \frac{-(a + a_2x_2 + \ldots + a_nx_n)}{a_1}$$

2. Les inégalités $t_1 \geq t_2$ (resp. $t_1 \leq t_2$), sont réécrites $t_1 - s^+ = t_2$ (resp. $t_1 + s^+ = t_2$), s^+ est appelée *variable non–négative*. Ces variables intermédiaires sont astreintes à prendre des valeurs positives ou nulles. Dans ce cas, on peut utiliser l'élimination gaussienne lorsqu'il y a au moins une variable arbitraire. Dans le cas où seules des variables non-négatives sont présentes, l'élimination gaussienne n'est plus suffisante car on peut générer des valeurs négatives. Se contenter de vérifier l'absence de ce problème n'est pas suffisant car il peut y avoir de nombreuses équations de ce type, qui de plus doivent être consistantes.

Équations avec uniquement des variables non-négatives

Décider de la satisfaisabilité d'un tel ensemble d'équations doit se faire de façon incrémentale. L'idée derrière le simplex est de maintenir les contraintes sous forme soluble. On dira qu'un ensemble d'équations est sous forme soluble s'il peut se réécrire en :

$$\begin{cases} y_1 &= b_1 + a_{11}x_1 + \ldots + a_{1m}x_m \\ \ldots & \\ y_n &= b_n + a_{n1}x_1 + \ldots + a_{nm}x_m \end{cases}$$

Où les $y_1, \ldots y_n$ sont les variables de *base*, les x_i les variables *non-base* et les b_i sont non-négatives.
On introduit des variables artificielles et on applique un pivot pour trouver une base minimisant la valeur des variables artificielles.

Les contraintes sont satisfaisables si et seulement si le résultat de la minimisation est 0.

L'incrémentalité : lorsque l'on rajoute une nouvelle équation à un ensemble sous forme soluble s'obtient en opérant comme suit :

1. Les variables de base dans la nouvelle contrainte sont remplacées par le membre droit correspondant.

2. On ajoute une nouvelle variable artificielle pour obtenir une base initiale.

3. On cherche à minimiser cette nouvelle variable.

à nouveau la satisfaisabilité est assurée si la minimisation vaut 0.

Différence et inégalité stricte
Pour les inégalités strictes de la forme $t_1 > t_2$ (resp. $t_1 < t_2$) il suffit de les réécrire sous la forme $t_1 \geq t_2 \wedge t_1 \neq t_2$ (resp. $t_1 \leq t_2 \wedge t_1 \neq t_2$). Reste alors à gérer les différences et à trouver une forme soluble adaptée. L'idée naturelle consistant à prendre la forme

$$0 \neq b + a_1x_1 + \ldots + a_nx_n$$

avec au moins une des valeurs de $\{b, a_1, \ldots, a_n\}$ non nulle et les x_i n'étant pas des variables de base. Malheureusement, *il est possible de mettre sous forme soluble un système non satisfaisable*.
Exemple 8.5.1
Considérons le système :

$$\begin{cases} y_1 & = & x_1 - x_2 \\ y_2 & = & x_2 - x_1 \\ 0 & \neq & x_1 - x_2 \end{cases}$$

En prenant les deux premières équations on obtient $y_1 + y_2 = 0$ *d'où l'on tire* $x_1 = x_2$. ◁

En fait une forme soluble sera valide s'il n'existe pas de constante cachée c'est-à-dire de variable astreinte à prendre une valeur unique.

Vérifier si de telle constante cachée n'existe pas dépend du système. On peut, par exemple, remarquer que si on suppose que le sysème était valide à l'instant t, il faut, pour qu'il ne soit plus valide au temps t+1, que la variable nouvellement introduite soit une des variables astreintes à prendre une valeur unique.

8.5.2 Contraintes non linéaires

Considérons le produit de deux nombres complexes $x_1 + iy_1$ et $x_2 + iy_2$ qui a pour valeur $x_1x_2 - y_1y_2 + i(x_1y_2 + x_2y_1)$ que l'on peut écrire en CLP($\mathcal{R}$) sous la forme

```
complex(X1,Y1,X2,Y2,R1,R2) :-
  R1 = X1*X2 - Y1*Y2,
  R2 = X1*Y2 + X2*Y1.
```

et interrogeons le système

```
2 ?-  complex(1,2,3,4,X,Y).

Y = 10
X = -5

*** Yes

3 ?-  complex(1,2,X,Y,A,B).

Y = 0.2*B - 0.4*A
X = 0.4*B + 0.2*A

*** Yes

4 ?- complex(X1,2,X2,4,-5,10).

X1 = -0.5*X2 + 2.5
3 = X1 * X2

*** Maybe

5 ?- complex(X1,2,X2,4,-5,10), (X1=1; X1=2; X1=3; X1=4).

X2 = 3
X1 = 1

*** Retry? y

*** No
```

Explications :
Lorsque le système rencontre une équation non linéaire, celle-ci est retardée jusqu'à ce que l'une

des deux situations suivantes soient rencontrées :

1. L'équation est rendue linéaire par d'autres contraintes.
2. Le calcul se termine.

Lors des deux premières requêtes le système s'est retrouvé dans la première situation, dans la troisième requête aucun élément supplémentaire n'est intervenu, le système indique son incompétence à résoudre plus avant le problème. à noter que la réponse `Maybe` peut aussi bien correspondre à un système ayant une infinité, une seule ou pas de solution.

Sémantique opérationnelle de CLP($\mathcal{R}$) revisitée :

8.5.3 Précision numérique

On s'intéresse à la suite récurrente

$$u_{n+1} = u_n + R(u_n - u_n^2)$$

Suite qui possède un comportement chaotique lorsque $R > 2.7$. Nous allons évaluer les termes de cette suite au moyen de 5 versions différentes, et nous allons regarder les implémentations dans les langages CLP($\mathcal{R}$), C, Maple et Scheme.

8.5.4 CLP($\mathcal{R}$)

```
eq1(R,X,(R+1)*X-R*(X*X)).
eq2(R,X,(R+1)*X-(R*X)*X).
eq3(R,X,((R+1)-R*X)*X).
eq4(R,X,R*X+(1-R*X)*X).
eq5(R,X,X+R*(X-X*X)).

itere(R,I,X1,X2,X3,X4,X5):- I<101,!,
  eq1(R,X1,Y1),
  eq2(R,X2,Y2),
  eq3(R,X3,Y3),
  eq4(R,X4,Y4),
  eq5(R,X5,Y5),
  affich(I,X1,X2,X3,X4,X5),
  itere(R,I+1,Y1,Y2,Y3,Y4,Y5).

itere(_,_,_,_,_,_,_).

affich(I,X1,X2,X3,X4,X5) :-
   (I=1 ; I=20 ; I=50 ; I=80 ; I=100),!,
   dump([I,X1,X2,X3,X4,X5]).
affich(_,_,_,_,_,_).



go :- itere(3,0,0.5,0.5,0.5,0.5,0.5).
```

I	X1	X2	X3	X4	X5
1	1.25	1.25	1.25	1.25	1.25
20	0.418895	0.418895	0.418895	0.418895	0.418895
50	0.0675671	0.063747	0.0599879	0.0614907	0.0615029
80	0.553038	0.817163	0.0219522	1.3104	0.0184394
100	6.71271e-05	0.119457	1.2565	0.658047	1.04282

8.5.5 C

```
#include <stdio.h>

#define R 3

main()
int i,j=0,k;
double x[]=.5,.5,.5,.5,.5;
double valeur[5][5];
int index[]=  1,20,50,80,100 ;


for(i=0;i<101;i++)
  if(i==index[j])

      for(k=0;k<5;k++)
        valeur[k][j]=x[k];
      j+=1;

  x[0]=(R+1)*x[0]-R*(x[0]*x[0]);
  x[1]=(R+1)*x[1]-(R*x[1])*x[1];
  x[2]=((R+1)-R*x[2])*x[2];
  x[3]=R*x[3]+(1-R*x[3])*x[3];
  x[4]+=R*(x[4]-x[4]*x[4]);


for(i=0;i<5;i++)

    printf("I=    for(k=0;k<5;k++)
      printf("X    printf("
n");
```

Execution sur une Sparc

I	X1	X2	X3	X4	X5
1	1.25	1.25	1.25	1.25	1.25
20	0.418895	0.418895	0.418895	0.418895	0.418895
50	0.0675671	0.063747	0.0599879	0.0614907	0.0615029
80	0.553038	0.817163	0.0219522	1.3104	0.0184394
100	6.71271e-05	0.119457	1.2565	0.658047	1.04282

Execution sur PC-Linux gcc (egcs 2.91.57)

I	X1	X2	X3	X4	X5
1	1.25	1.25	1.25	1.25	1.25
20	0.418895	0.418895	0.418895	0.418895	0.418895
50	0.0618953	0.0618953	0.0618953	0.0618953	0.0618953
80	1.25982	1.25982	1.25982	1.25982	1.25982
100	0.190393	0.190393	0.190393	0.190393	0.190393

8.5.6 MAPLE

```
> r:=3:u[0]:=0.5:v[0]:=0.5:w[0]:=0.5:
> for i from 0 to 100 do
     u[i+1]:= (r+1)*u[i] - r*u[i]*u[i];
     v[i+1]:=((r+1) -r*v[i])*v[i];
     w[i+1]:=w[i]+r*(w[i]-w[i]*w[i]);
  od :
```

I	X1	X3	X5
1	1.25	1.25	1.25
20	.418832780	.4189314629	.418901139
50	.08563346423	.001233040819	.700132281
80	.3375262195	.001590142946	1.333078292
100	.1866691722	.3570344050	1.087487672

8.5.7 Scheme

La version Scheme utilisée est `scm 5b2`

```
;; les différentes implémentations possibles

(define (v1 old r)
  (- (* (+ r 1) old)
     (* r (* old old))))

(define (v2 old r)
  (- (* (+ r 1) old)
     (* r old old)))

(define (v3 old r)
  (* (- (+ r 1) (* r old)) old))

(define (v4 old r)
  (+ (* r old) (* (- 1 (* r old)) old)))

(define (v5 old r)
  (+ old (* r (- old (* old old)))))

;; La version itérée

(define (iter num fun val r)
  (if (= num 1) (fun val r) (iter (- num 1) fun (fun val r) r)))
```

I	X1	X3	X5
1	1.25	1.25	1.25
20	0.418895025025972	0.418895025010612	0.418895025013753
50	0.0675670954719196	0.0599878799198289	0.0615028941943347
80	0.553038460686109	0.0219521769792697	0.0184394042940725
100	6.71270949128302e-5	1.25649567629486	1.04282303339008

Chapitre 9

Logo

9.1 Présentation rapide de *Logo

Dans [79], l'auteur a proposé un langage StarLogo pour permettre la réalisation d'expériences dans des micro-mondes massivement parallèles, pour cela il s'est fortement inspiré des automates cellulaires où des règles simples appliquées aux cellules peuvent conduire à des organisations très complexes. Désirant que l'utilisateur expérimente par le biais d'activité, il a choisi d'étendre le langage Logo afin de permettre la manipulation de plusieurs tortues en parallèles. L'approche Logo (prônée par Seymour Papert) a démontré que le langage était (relativement) accessible pour des non-informaticiens. Mais si les « tortues »peuvent représenter à peu-près n'importe quel type d'objet dans le monde : une colonie de fourmis, un bouchon de la circulation, ..., le langage n'offre pas un certains nombres des mécanismes nécessaires à la mise en œuvre de comportements de groupe (suivi de gradient entre autres). D'où la définition d'un nouveau langage *Logo ($\star$ pour les systèmes parallèles) avec la gestion de plusieurs tortues pouvant inter-agir soit avec d'autres tortues, soit avec le monde et ce de manière totalement asynchrone.
Le monde est défini par un ensemble de « patches »qui disposent de différentes variables. Un opérateur `ask` permet de commander soit aux patches soit aux tortues d'exécuter des morceaux de codes en parallèle. Deux implémentations en Java sont disponibles l'une StarLogo est accessibles à partir de `www.media.mit.edu/starlogo/`, l'autre NetLogo à partir de `ccl.northwestern.edu/netlogo/`. Ma préférence est allée à NetLogo qui semble être plus robuste à l'utilisation.

9.2 Eléments de programmation

Comme nous venons de le voir il existe plusieurs dialectes de ce langage, mais les mécanismes sous-jacents sont extrêmement voisins, il est bien entendu hors de propos de donner le lexique des commandes utilisateurs, nous renvoyons lelecteur aux excellents tutoriaux accessibles avec les implémentations pré-citées. En StarLogo, on distingue deux entités l'observateur et les tortues. En NetLogo l'approche est plus uniforme, et à notre préférence, c'est pourquoi nous suivrons la démarche adoptée par cette implémentation, mais nous insistons sur le fait que les idées sous-jacentes et la syntaxe ne diffèrent que très peu de celles utilisées en StarLogo.

9.2.1 Agents

Le monde est constitué d'agents. Ces agents obéissent à des instructions. Chaque agent agit/inter-agit avec le monde, et ce parallèlement aux autres agents placés dans l'environnement. NetLogo distingue trois type d'agents `turtles`, `patches`, `observer`. Les `turtles` se déplacent dans le monde qui est bi-dimensionnel et divisé en une grille de `patches`. Chaque patch est un carré sur lequel une tortue peut se déplacer suivant une certaine direction. L'observateur n'est pas localisé. Au démarrage, il n'y a pas de tortues, l'observateur a la possibilité de creer des tortues, les patches aussi. Les patches possèdent des coordonnées, le centre du monde est de coordonnées 0 0. Ces coordonnées croissent

en allant vers le nord ou vers l'est. La taille du monde est paramétrable. Les tortues sont elles aussi localisées mais dans un espace continu. Le monde n'est pas « borné »mais est vu comme un tore.

9.2.2 Procédures et fonctions

Une procédure débute par le mot clef `to` suivi du nom de la procédure et se termine par le mot clef `end`. Une fonction (renvoie une valeur) elle débute par le mot clef `to-report`, se termine par `end` et **doit** contenir une commande `report`. Les commandes sont soient des mots du lexique, soient des appels de procédures (ou fonctions). Un bloc de commandes est encadré par des crochets. Un point virgule est un commentaire est à une portée limitée à la ligne en cours.

EXEMPLE 9.2.1

```
; cette procédure demande à toutes les tortues d'exécuter la séquence
; avance d'une case puis tourne à droite d'un angle de moins de 45
; degrés et tourne à gauche d'un angle de moins de 45 degrés

to go
 ask turtles [
   fd 1 ; fd signifie forward
   rt (random 45) - (random 45) ; rt = right turn
 ]
end
```

◁

9.2.3 Variables

Les variables peuvent être globales, locales à une procédure, propre à un agent. Certaines variables sont prédéfinies comme, par exemple, les coordonnées des agents (pour les tortues `xcor, ycor`, pour les patches `pxcor, pycor`, leur couleur `color`, pour les tortues leur direction `heading`, ou leur identificateur `who`.

Pour déclarer une variable globale, on utilise le mot clef `globals`, pour une variable locale le mot clef `locals`, pour une variable associée à un agent, on utilise le type de l'agent suffixé par `-own`.

EXEMPLE 9.2.2

```
globals [clock]           ; une horloge
turtles-own [age vitesse] ; les tortues ont un age et une vitesse
patches-own [food]        ; les patches ont une qte de nourriture

to-report factorielle [n]  ; calcule la factorielle de n
 locals [res i]            ; i un index, res le resultat
 set res 1
 set i 2
 if n < 0 [report -1]      ; erreur
 while [i < n][            ; tant que i<n faire
   set res res * i         ; res := res * i
   set i i + 1             ; i   := i + 1
 ]
 report res                ; renvoie le resultat
end                        ; fin de la fonction
```

◁

9.2.4 Ask

Probablement la commande la plus importante, elle permet de demander à un ensemble d'agents d'exécuter, dans un ordre quelconque une séquence d'instructions. Elle sert aussi à s'adresser à un agent en particulier en utilisant conjointement l'un des suffixes modificateurs `-at`, `-from`

EXEMPLE 9.2.3

```
turtles-own [energy speed]
patches-own [food]
to setup
 ca ; clear all
 crt 100 ; creation de 100 tortues
 ask turtles [
   set color red ; elles deviennent rouge
   rt random 360 ; elles ont une directio aleatoire
   set energy (random 100) ; energie entre 0 et 99
   set speed  random 10    ; vitesse entre 0 et 9
   fd speed ; elles avancent en fonction de leur vitesse
 ]
 ask patches [
   set food random 10 ; on met une qte aleatoire de nourriture
   scale-pcolor green food 0 10 ; la couleur de base est le vert
                                ; l'intensite depend de food
 ]
end

to oneAction
 ask turtle 0 [fd 1] ; la tortue 0 avance d'un pas
 ask turtle 1 [set color yellow] ; la tortue 1 jaunit
 ask turtle 2 [                  ; demande à la tortue 2
   ask patch-at 1 0 [            ; de demander au patch a sa gauche
       set food food - 1         ; de diminuer sa nourriture de 1
       scale-pcolor green food 0 10 ; d'adapter sa couleur
   ]
 ]
```

◁

9.2.5 Groupe d'agents

Comme leur nom l'indique il s'agit de groupes d'agents qui peuvent être aussi bien des tortues que des patches mais pas des deux. On a déjà vu deux groupes d'agents les `turtles` et les `patches`. On a la possibilité de discriminer dans ces deux ensembles des sous-ensemble particulier, par exemple à l'aide des commandes `turtles-here` ou `turtles-at`, le premier correspond au sous-ensembles des tortues qui sont sur le patch courant ou bien celles qui se trouvent sur un patch distant. On peut utiliser l'opérateur `with` pour sélectionner les agents ayant une certaine propriété. Ci-après on pourra trouver quelques exemples :

EXEMPLE 9.2.4

```
turtles with [color = red] ; les tortues rouges
patches with [food > 0]    ; les patches ayant de la nourriture
patches with [pxcor > 0]   ; les patches à droite de l'écran
turtles in-radius 3        ; toutes les tortues à moins de 3 patches
turtles with [(xcor > 0) and (ycor > 0) and (pcolor = green)]
; les tortues dans le premier quart d'écran sur un patch vert
```

◁

Une fois défini un groupe d'agents on peut utiliser l'un des trois opérateurs suivants `ask` pour les faire agir ; `any` pour savoir si l'ensemble est vide ; `count` pour connaître la cardinalité de l'ensemble sélectionné. Il existe d'autres opérateurs possibles tels que `random-one-of` pour en prendre un au hasard, `max-one-of, min-one-of`, ...

9.2.6 Espèces

On peut particulariser les tortues en espèces (ce qui revient à creer des groupes de tortues) permettant ainsi de modéliser des comportements différents - très utile pour la modélisation d'un système de type prédateurs-proies. Comme il ne s'agit que d'une discrimination syntaxique (le système implémente en fait les tortues avec une variable `breed`) on peut à tout moment faire passer une tortue d'une espèce à une autre.

EXEMPLE 9.2.5

```
breeds [chat souris]
chat-own [ ... ]
souris-own [ ... ]

ask random-one-of chat [set breed souris] ; un chat devient souris
```

◁

9.2.7 Listes

*Logo propose la liste comme structure de données, une liste est une suite de valeurs entre crochets. On dispose d'accesseurs tels que `first,last` qui renvoie respectivement le premier et le dernier éléemnt d'une liste. On a la possibilité d'ajouter un élément dans une liste `fput,lput` le premier pour ajouter en tête, le second pour ajouter en queue. On peut accéder ou modifier une valeur dans la liste, savoir si la liste est vide ou pas, enlever un ou plusieurs éléments. La liste joue en *Logo le rôle habituellement dévolue aux tableaux. Le programme donné en annexe E en fait un usage important pour les gênes et leur altération.

9.3 Conclusion

Il existe une interface utilisateur basé sur Java qui offre la possibilité à l'utilisateur d'inter-agir avec le programme. Du point de vue programmation aucune difficulté majeure, c'est à mon avis une excellente introduction pragmatique à un environnement multi-agents, bien plus facile à utiliser que des outils comme `swarm`, certes l'implémentation Java limite les performances mais pour faire du maquettage je trouve l'outil très agréable d'autant qu'il est multi-plateformes (merci Java). Que vous choisissiez l'implémentation StarLogo ou NetLogo n'a aucune espèce d'importance, la philosophie étant la même passer d'un environnement à l'autre n'est pas pénalisant. Par ailleurs Logo ayant la réputation (établie) d'être facilement accessible à un non-informaticien, je ne peux que vous enjoindre à l'essayer dans l'une de ces versions.

Chapitre 10

Toupie ou le μ-calcul sur les domaines finis

Chapitre 11

Langage Orienté Objets

Les langages orientés objets [65] ont changés de statut entre la première rédaction de ce cours (1986) et maintenant (1998). En 1986, les langages orientés objets et leurs avatars (langages de classes, de frames ou d'acteurs) faisaient partie de l'IA, désormais ils appartiennent au bagage de l'informaticien et sont partie intégrante des cours de Génie Logiciel (citons `C++`, `Delphi`, `Java`) et sont même intégrés dans les cours de second cycle (Informatique ou MASS). Après un bref aperçu des concepts majeurs qui sont utilisés, je vous convie à une introduction du langage `Smalltalk` [16, 40] qui fut l'un des premiers en ce domaine.

Idéalement, un programmeur devrait pouvoir disposer des facilités suivantes :

- Outils de coordination pour de nombreux types de données,
- Environnement de programmation perfectionné,
- Outils pour un prototypage rapide,
- Outils pour représenter des connaissances hétérogènes,
- Outils pour exploiter des bases de connaissance et des bases de données,
- Outils pour la programmation parallèle.

L'objet regroupe une partie statique, un ensemble de données et une partie dynamique, un ensemble de procédures manipulant ces données. Dans le meilleur cas, les données sont *privées* : l'objet est défini par son comportement et non par sa structure. L'objet est muni d'une *interface* qui spécifie comment il inter-agit avec le milieu extérieur. Le seul moyen de communiquer avec lui est d'utiliser les procédures ou *méthodes* de son interface. C'est l'objet lui-même qui est responsable de la manière dont l"exploitation des données privées est effectuée. L'appel d'une méthode est plutôt vu comme une requête ou un *message* de l'appelant demandanr l'exécution de certaines opérations.

On distingue habituellement dans ce type de programmation différents groupes de langages :

- langage de classes,
- langage de frames,
- langage d'acteurs.

DÉFINITION 11.1 (OBJET)
L'objet regroupe un ensemble de couples (variable,valeur) appelée respectivement *variable d'instance* et *valeur d'instance* qui définissent son *état* et des *méthodes* qui constituent son *comportement*. Une méthode est un couple (sélecteur,procédure).

EXEMPLE 11.0.1
Considérons un cercle C1 défini par
centre : (3,4)

rayon : *10*
périmètre : $2 \times \pi \times$ *rayon*
distance : *méthode permettant le calcul de la distance entre deux cercles.* ◁

EXEMPLE 11.0.2
Considérons un rectangle R1 défini par
longueur : *4*
largeur : *2*
angle : *90*
périmètre : $2 \times$ *(longueur + largeur)* ◁

Comme on peut le voir sur ces deux exemples simplistes, la programmation par objets autorise la surcharge (la méthode périmètre est différente pour les deux objets rectangle et cercle).

DÉFINITION 11.2 (ENVOI DE MESSAGE)
Appliquer une méthode à un objet s'appelle un envoi de message. La syntaxe informelle pourrait être :
`envoi(<receveur>,<selecteur>,<arg1>,...,<argn>)`, où n représente le nombre de paramètres nécessaires au selecteur.

Ainsi `envoi(C1,périmètre)` permet de calculer le périmètre du cercle C1 ; tandis que si C2 est un autre cercle l'ordre `envoi(C1,distance,C2)` permet le calcul de la distance entre les deux cercles C1 et C2.

DÉFINITION 11.3 (CLASSE)
À tout objet est associée une classe spécifiant sa structure et son comportement. Le comportement est le dictionnaire des méthodes applicables à l'objet.

EXEMPLE 11.0.3
La classe `cercle` *définit les variables d'instance {centre, rayon} et les méthodes {périmètre, distance}.* ◁

DÉFINITION 11.4 (INSTANCIATION)
Les objets sont créés dynamiquement à partir de leur classe, la syntaxe informelle pourrait être : `new(<classe>,<val1>,...,<valk>)`, où k représente le nombre de variables d'instance.

EXEMPLE 11.0.4
On peut créer C1 en utilisant l'ordre `new(cercle,(3,4),10)` ◁

L'instantiation définit la relation **est_un** entre l'objet et la classe dont il est issu.

DÉFINITION 11.5 (HÉRITAGE)
Il consiste à enrichir une classe pour en créer une autre, l'intérêt étant une factorisation du code et donc un gain de place dans l'application.

EXEMPLE 11.0.5
Un carré est un cas particulier de rectangle pour lequel longueur et largeur sont identiques. ◁

L'héritage définit la relation **sorte_de** entre la classe enrichie et la classe initiale.

Il est possible à partir de ces deux relations de construire un *graphe d'héritage* tel que celui de la figure 11.1.

Lorsque l'on veut appliquer une méthode à l'objet `Q`, le système va voir si elle est spécifique à la classe `carré`, sinon en remontant dans le graphe d'héritage il va consulter la classe `rectangle`.

DÉFINITION 11.6 (MÉTA-OBJET)
Cette notion apparaît lorsqu'une classe est considérée comme un objet ; il s'agit alors d'une classe permettant de créer de nouvelles classes.

11.1 SmallTalk

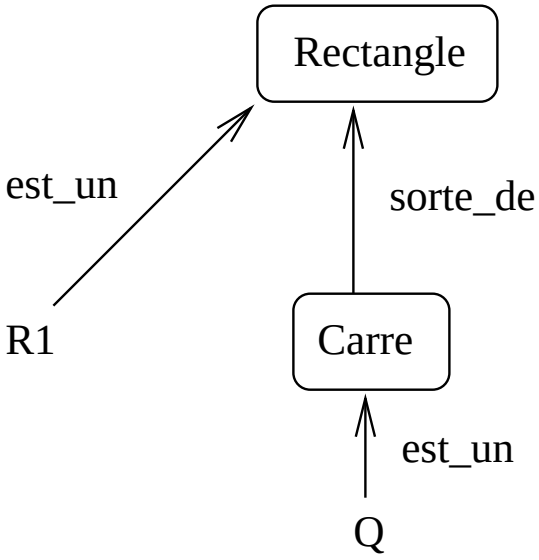


FIGURE 11.1 – Un graphe d'héritage

Quatrième partie

Techniques de l'IA

Chapitre 12

Une application : les systèmes experts

Les systèmes experts (SE) l'un des aspects les plus à la mode de l'intelligence artificielle (IA). Ils fleurissent depuis les années 70, ce sont en fait des programmes qui manipulent un grand nombre de connaissances sur un domaine particulier, ils sont construits pour raisonner à partir de cet ensemble d'informations, et mener à bien une tâche alors qu'il n'existe pas d'algorithme connu pour y parvenir.

Dans certains domaines des résultats significatifs ont déjà été obtenus, mais les espoirs sont à la hauteur des difficultés rencontrées. Le développement des SE est en grande partie dû à l'échec relatif des techniques de l'informatique classique pour des problèmes où la résolution fait intervenir une grande quantité de connaissances.

Diagnostiquer une panne d'ordinateur, une maladie infectieuse, localiser une nappe de pétrole, ou même jouer au poker ne peuvent être résolus sans faire appel à de très nombreuses connaissances, qui plus est, spécifiques du domaine traité.

A l'heure actuelle les SE semblent être la moins mauvaise solution pour résoudre ce type de problèmes. Ce sont des systèmes spécialisés, qui utilisent une base de connaissances acquise auprès d'un expert (humain) du domaine.

Un SE n'est qu'une *aide* pour résoudre quelques difficultés.

En général un SE utilise des règles de production ayant la forme suivante :

```
Si Hyp1 & .. & Hypn
Alors Conc1 &.. & Concm ($\alpha$)
```

Le coefficient $\alpha, 0 \leq \alpha \leq 1$, qui est parfois omis, correspond au niveau de confiance que l'on a de la règle

DÉFINITION 12.1
Un SE est constitué de :

1. Une base de faits contenant les données courantes.
2. Une base de règles de la forme <Hypotheses, Conclusions>.
3. Un moteur d'inférences qui interprète les règles à partir des faits.

12.1 Fonctionnement du SE et définitions

fonctionnement :

1. détections des règles possibles,
2. choix d'une règle,
3. application, puis déductions, puis mise à jour de la base de la base de faits.

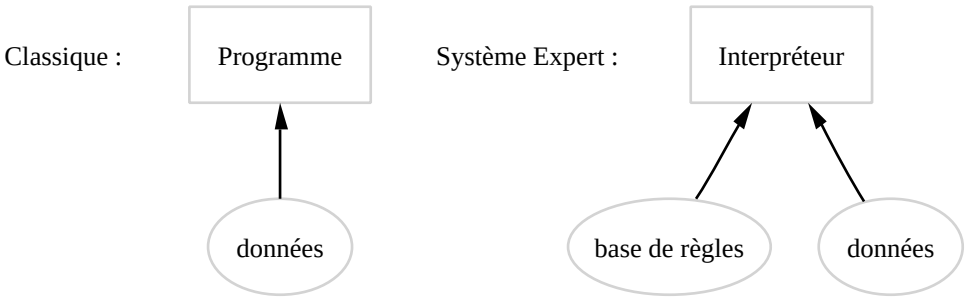


FIGURE 12.1 – Programmation classique versus système expert

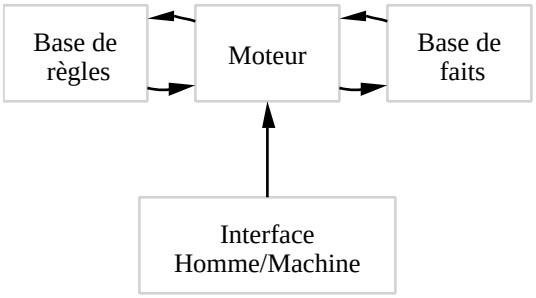


FIGURE 12.2 – Aspect extérieur d'un SE

définitions :

Le moteur d'inférences est la « partie informatique »qui met de l'ordre dans les faits « en vrac »de la base. Son fonctionnement est de deux sortes :

1. *chaînage avant* : il part des faits pour aboutir aux conclusions par filtrage des conditions d'une règle sur les faits (exemple on a certains symptomes et on cherche les affections possibles).
2. *chaînage arrière* : il part du but à atteindre et applique les règles à rebours (exemple : il y a une panne et on cherche à détecter les causes afin de les supprimer).

L'ordre d'un SE est directement lié aux types d'informations traitées, cette notion découle directement de la logique où 0 $\Longleftrightarrow$ propositionnel et 1 $\Longleftrightarrow$ des prédicats.

a. ordre 0 : le système ne prend pas en compte les variables, une règle n'est constituée que de faits (ex `si plumes alors oiseau` qui se lit « s'il a des plumes alors c'est un oiseau »).

b. ordre 1 : utilisation de variables à l'intérieur des règles (ex `si humain(X) alors mortel(X)`, qui se lit « si X est humain alors X est mortel »).

Le moteur d'inférences sert à déclencher les règles et à les enchaîner les unes aux autres pendant la resolution qui correspond à la détection puis au déclenchement de la ou des règles selectionnées.

Problèmes rencontrés :

1. Mise au point du moteur : chaînage avant, chaînage arrière ou chaînage mixte.
2. Écriture des règles.
3. Inefficacité en temps d'exécution
4. Difficulté pour trouver toutes les règles (nombre minimal)

12.2 En résumé

Un SE a :

1. une « grosse masse »de connaissances dans le domaine traité,
2. de hautes performances comparables à des experts humains, restreint à un seul domaine,

c. une certaine utilité, disponibilité, amortissement possible.

Un SE doit :

1. être transparent i.e.,
   (a) modification aisée des connaissances,
   (b) explication de chaque pas de deductions ;
2. avoir une bonne interface Homme/Machine.

La figure 12.3 montre le schéma conceptuel du fonctionnement d'un SE.

12.3 Quelques applications

Principalement dans le domaine de la médecine et de la biologie :
– diagnostic,
– aide à la prescription thérapeutique,
– suggestions de plans d'expérimentation,
– surveillance de malades sous soins intensifs.

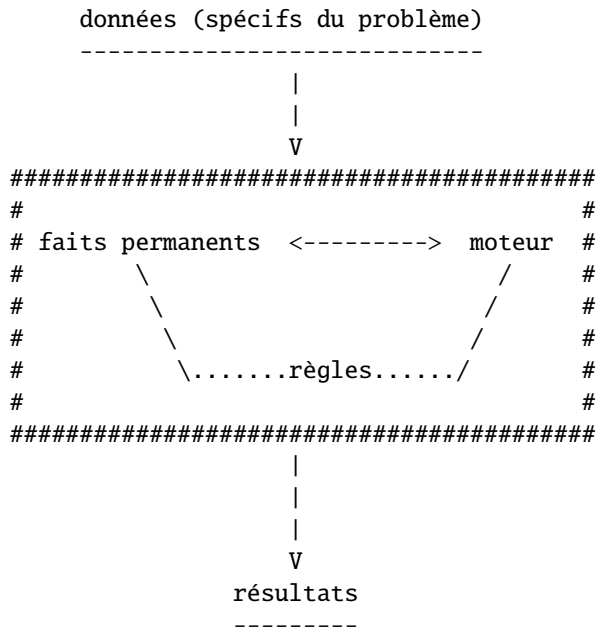


FIGURE 12.3 – Schéma conceptuel d'un SE

12.3.1 Quelques exemples de SE existants pour le milieu médical

Le but de l'énumération qui suit est de montrer le particularisme des SE dévelopés en médecine :

- *mycin* 1976
  aide aux diagnostics et aux traitements pour les infections bactériennes du sang.
- *casnet* 1976
  système d'aide au traitement et au diagnostic du glaucome de l'œil en tenant compte de son évolution.
- *centaur* 1979
  interprète les resultats de tests sur des patients atteints de maladies pulmonaires.
- *guidon* 1979
  système d'aide à l'enseignement de la médecine.
- headmed 1976
  traitement pharmacologique en psychiatrie.
- *iris* 1976
  consultation médicale.
- *internist* 1975
  SE en médecine interne.
- *medico* 1976
  ophtalmologie/visite médicale.
- *oucocin* 1981
  traitement du cancer.

- *puff* 1979
  diagnostics infections pulmonaires.
- *sam* 1981
  maladies vasculaires (diagnostic+traitement).
- *vm* 1979
  aide à la réanimation.

12.3.2 La trilogie chez les SE

Mycin est le modèle de très nombreux SE tant dans le milieu médical que dans d'autres domaines comme la prospection pétrolière ou encore la construction en bâtiments.
Dendral quant à lui est célèbre du fait de son ancienneté (les premières recherches ayant eu lieu en 1964), il est spécialisé dans l'analyse des molécules chimiques.
Prospector enfin est celui qui certainement est à l'origine de l'engouement actuel pour les SE dans le monde industriel car il a permis la découverte au Canada d'un gisement de molybdène alors que les experts humains ne l'avaient pas detecté.

12.4 Chaînage avant

Pour le chaînage avant, le moteur d'inférences utilise la base de faits. La gestion en est assurée par deux fonctions, l'une qui va ajouter un fait démontré, l'autre qui va rechercher si un fait a été établi.

Il s'agit ici d'un moteur en chaînage avant fonctionnant par saturation, tant que de nouvelles informations peuvent être rajoutées on boucle sur la base de règles

12.5 Chaînage arrière

Le moteur travaille ici sur plusieurs informations, d'une part les faits qui sont connus comme étant vrais, d'autre part ceux connus comme étant faux, et enfin les faits que le système doit établir (terminaux), lorsque le système est bloqué, il demande à l'utilisateur de statuer sur certains faits, avant de relancer le calcul.

Ces différents algorithmes décrits dans les figures 12.4 12.5 sont tirés de [34].

12.6 Solution en Prolog

Ci-après je vous donne le code de deux petits systèmes experts écrits en Prolog. Le premier fonctionne par saturation sur un exemple jouet, que je vous convie à tester :

12.7 Cohérence des bases de connaissances

Problème difficile que celui de définir et vérifier la cohérence d'une base de connaissances et celle d'un ensemble de règles d'un système expert. On peut aborder le problème sous différents angles que nous n'allons qu'effleurer. L'objectif n'étant pas d'apporter des solutions mais de vous sensibiliser au problème.

Une première approche est la validation expérimentale, qui consiste, comme pour tout programme infomatique à vérifier que ce que l'on a écrit correspond bien au problème à traiter. Les méthodes les plus souvents utilisées sont la simulation, les tests ou des méthodes statistiques.

Une autre approche est la validation formelle, dans ce cas on cherche à définir un modèle de la cohérence d'une base et ensuite vérifier que ce modèle est en adéquation avec la base. On peut identifier deux tendances majeures :

```
Procédure CHAINAGE-AVANT
début
 Répéter
  DEDUIT := DECLENCHER(ensemble de règles)
 Jusqu'à non DEDUIT
fin

Fonction TEST-SI(règle) : {Vrai, Faux }
début
 TEST := Vrai
 Pour tout élément de la partie SI de la règle faire
  Tant que TEST faire
    Si l'élément est dans la base alors TEST := Vrai
    Sinon TEST := Faux
    fsi
  fait
 fpour
 retourner(TEST)
fin

Fonction APPLICATION(règle) : {Vrai, Faux }
début
 APPLIQUE := Faux
 Pour tout élément de la partie ALORS de la règle faire
  Si l'élément n'appartient pas à la base alors
   APPLIQUE := Vrai
   Ajouter l'élément dans la base
  fsi
 fpour
 retourner(APPLIQUE)
fin

Fonction DECLENCHER(ensemble de règles) : { Vrai, Faux }
début
 DECLENCHE := Faux
 Pour toutes les règles faire
  Tant que non DECLENCHE faire
    Si TEST-SI(règle) alors DECLENCHE := APPLICATION(règle) fsi
  fait
 fpour
 retourner(DECLENCHE)
fin
```

FIGURE 12.4 – algorithme du chainage-avant

```
Procédure CHAINAGE-ARRIERE(liste de faits terminaux)
début
 Si la liste est vide alors Arrêt
 Sinon
  Soit F, le premier élément de la liste
  Si VERIF(F) alors
   AFFICHE(F vérifié)
  fsi
  CHAINAGE-ARRIERE(reste de faits terminaux)
 fsi
fin

Fonction VERIF(H) : { Vrai, Faux }
début
  VERIFIE := Faux
  Si H appartient à la base de faits alors VERIFIE := Vrai
  Sinon
    Si H appartient à la base des faits faux
    Alors VERIFIE := Faux
    Sinon
      construire l'ensemble des règles déduisant H
      Si l'ensemble est vide Alors
         demandez H à l'utilisateur
         Si H est Vrai Alors
            ajouter H aux faits vrais
            VERIFIE := Vrai
         Sinon
            ajouter H aux faits faux
            VERIFIE := Faux
         fsi
      Sinon
         Pour toutes les règles R de l'ensemble faire
              Si TEST-SI(R) alors
                 VERIFIE := Vrai
                 APPLICATION(R)
                 Arrêt
              fsi
         fpour
      fsi
    fsi
  fsi
  retourner(VERIFIE)
fin

Fonction TEST-SI(R) : { Vrai, Faux }
début
  TEST := Vrai
  Pour tout élément E de la partie SI de R faire
      Si non VERIF(E) alors
         TEST := Faux
         Arrêt
      fsi
  fait
  retourner(TEST)
fin
```

FIGURE 12.5 – algorithme de chaînage arrière

```
/*
  mini mini-systeme-expert
  dont le source provient de mini&micro
  chainage avant + saturation
*/
deduc(Fait):-regle(Cond,Conc),verif(Cond,Fait),not verif(Conc,Fait),
             affic(Conc),ajout(Conc,Fait,Nouv_fait),!,deduc(Nouv_fait).
deduc(_).

verif(X,Y) :- inter(X,Y,X).
ajout(X,Y,Z) :- conc(X,Y,Z).

conc([],L,L).
conc([X|L1],L2,[X|L3]):-conc(L1,L2,L3).

elt(X,[X| _]).
elt(X,[_|L]):-elt(X,L).

inter([],_,[]).
inter([X|L],L1,[X|L2]):-elt(X,L1),!,inter(L,L1,L2).
inter([_|L1],L2,L3):-inter(L1,L2,L3).

affic([X|L]):-write(deduit(X)),nl,!,affic(L).
affic([]):-nl,!.

regle([possede(poils,X)],[mammifere(X)]).
regle([possede(plumes,X)],[oiseau(X)]).
regle([mange(herbe,X)],[herbivore(X)]).
regle([mange(viande,X)],[carnivore(X)]).
regle([herbivore(X),carnivore(X)],[absurde]).
regle([herbivore(X),carnivore(Y)],[mange(X,Y)]).
regle([oiseau(X),carnivore(Y)],[mange(X,Y)]).

/*
?- deduc([mange(herbe, vache), mange(viande, loup), possede(plumes,autruche)]).

deduit(oiseau(autruche))
deduit(herbivore(vache))
deduit(carnivore(loup))
deduit(mange(vache, loup))
deduit(mange(autruche, loup))

*** Yes
*/
```

FIGURE 12.6 – Chainage avant par saturation

- La première, une fois le modèle déterminé, consiste à vérifier que le système construit des bases de faits cohérentes. Cette vérification pouvant être locale, on vérifie, pour chaque nouveau fait établi, que celui ci ne viendra pas entâcher la base d'incohérence. Deux possibilités s'ouvrent alors, l'une statique, l'autre dynamique. En mode dynamique, le contrôle s'effectue en modifiant la base de faits pour que celle-ci reste cohérente à tout instant. En mode statique, le contrôle est effectué, mais aucune modifiation n'est envisagée, un tel système se contente en général d'envoyer un message d'erreur à l'utilisateur. Le gros avantage (que ce soit dans le cas dynamique ou statique) et de permettre de concevoir un module indépendant de la nature des concepts manipulés, et qui soit rattaché au moteur d'inférences. On parle, dans ce cas d'une *validation par les faits*.
- La seconde, identifiée comme une *validation par les règles* est plus liée au problème de l'acquisition des connaissances. Il s'agit alors de vérifier qu'un ensemble de règles est cohérent, une manière de traiter ce problème est de fournir une compilation incrémentale des règles.

Reste un problème de définition, qu'entend-t-on par cohérence, la définiton qui vient en premier à l'esprit est celle-ci « un système est cohérent s'il ne contient pas, ou s'il ne peut pas établir un fait et son contraire ». Malheureusement cette définition n'est pas adéquate pour les SE car :

- On ne manipule que très rarement la notion de négation pour un fait.
- La négation logique n'existe pas. On se place le plus souvent dans l'hypothèse du monde clos, c'est-à-dire qu'un fait est faux s'il n'est pas démontrable et non si sa négation a été explicitement établie.
- Deux faits peuvent être incohérents sans qu'ils soient la négation l'un de l'autre, exemple « il fait beau »et « il pleut ».

Ce qui sous-entend que la notion de cohérence est intrinsèquement liée avec la nature de la connaissance manipulée. Et donc seul l'expert est à même de déterminer si deux faits sont incohérents.

12.8 Complétude

Dans le cadre du chaînage avant, on est confronté à un autre problème, celui de la complétude. C'est-à-dire, comment garantir qu'un tel système peut trouver *toutes* les informations déductibles à partir d'un ensemble initial.

Considérons un système possédant les 3 règles suivantes :

```
règle 1 :
Si A == 0 ALORS (B = 1 ET D = 2)

règle 2 :
Si A != 0 ALORS (B = 2 ET D = 1)

règle 3:
SI C == 1 ALORS (B = 1)
```

Supposons que l'on sache au départ que `(C = 1)`, on va pouvoir alors déduire, grâce à la règle 3, que `(B = 1)`. Le système s'arrête, alors qu'il est évident, que puisque `(B = 1)` il ne peut valoir 2, et donc que A vaut 0 et **D vaut 2**. Il est vital, pour mettre ce type de raisonnement en œuvre que les faits initiaux soient cohérents, en effet on ne peut appliquer le raisonnement précédent, si on a `C = 1 ET A = 1`. Tout cela repose sur l'usage exclusif de la règle du **modus ponens**, alors que dans un univers clos on peut utiliser de la même façon la règle du **modus tolens** (ou contraposée). Il est difficle de mettre en œuvre la « méta règle suivante » :

SI (A implique B) est **vrai** ALORS (non B implique non A) est **vrai**

Si on mettait au point un algorithme capable de gérer ce type de règle, on augmenterait sensiblement les temps d'exécution. C'est pourquoi certains travaux ont étudié, comment à partir d'un ensemble de règles, construire un nouvel ensemble intégrant toutes les contraposées. Si nous reprenons l'exemple précédent la base de règle on devrait rajouter 5 nouvelles règles :

```
SI B != 1 ALORS (A != 0)
SI B != 2 ALORS (A = 0)
SI B != 1 ALORS (C != 1)
SI D != 2 ALORS (A != 0)
SI D != 1 ALORS (A = 0)
```

Ce qui amène au simple constat d'une explosion combinatoire de la base de règles, et de plus rend impossible la mise en place de bases plus complexes mettant en jeu des coefficients de vraisemblance.

12.9 Extensions possibles

Dans cette partie, nous allons voir deux extensions simples permettant de définir des systèmes mieux à même de résoudre des problèmes réels. La première consiste à étendre les propostions dans le cadre d'une logique avec contraintes finies. L'autre va permettre de modifier le comportement tout ou rien des informations manipulées.

12.9.1 Domaine fini

12.9.2 Incertitude et Imprécision

Lorsque l'on est confronté à la réalisation de systèmes experts manipulant des connaissances, on bute souvent sur la notion de raisonnement approximatif. D'une part, la notion d'*incertain* traite du cas où les éléments de connaissance ont une valeur de vérité connue avec plus ou moins de précision, d'autre part la notion d'*incertain* où les éléments de connaissance ont un **contenu** imprécis. Les exemples suivants sont tirés de [45] :

Incertitude
1. Gustave va probablement être rçu à son examen
2. Prendre le métro doit être une bonne solution
3. Si le malade tousse, il y a présomption de bronchite

Imprécision
1. Gus mesure entre 1m70 et 1m80
2. La température est d'environ 24^o C
3. Si le malade tousse beaucoup, lui donner un peu de sirop

Coefficients de vraisemblance

La démarche la plus souvent utilisée dans les SE découle des travaux sur MYCIN (1976) qui possède un mécanisme empirique de gestion des facteurs de certitude.
Les prémisses d'une règle sont des conditions élémentaires notées cond_i reliées par des connecteurs $\wedge$ (ET) et $\vee$ (OU). La partie conclusion peut, quant à elle, être très variée : introduction d'un nouveau fait, émission d'une hypothèse, déclenchement d'une action, ...

On se restreint dans la suite au cas où la conclusion est un fait ou ne hypothèse (de diagnostic, par exemple) et où la règle est incertaine. Ce dernier point introduit un coeeficient numérique associé à la règle **cr** permettant d'indiquer que l'application de cette règle est une inférence faible. Ce coefficient va influencer la confiance associée à la conclusion, les cond_i pouvant elles aussi être incertaines. Ce coefficient pourra servir à sélectionner une règle parmi d'autres, il sera alors vu comme un coefficient d'« importance ».

Supposons que les cond_i au moment où l'on déclenche la règle soit affectées d'un coefficient de vraisemblance $\mathbf{cc}_i$. La phase d'évaluation des prémisses va avoir pour but de calculer un coefficient de certitude **cp** qui dépend des $\mathbf{cc}_i$ et des connecteurs utilisés. On décide, ensuite, si on peut ou non appliquer la règle candidate. Pour cela on compare la valeur de **cp** avec un seuil de déclenchement pré-établi pour la règle. Si la règle est déclenchée, on associe à la conclusion un coefficient **cv** qui dépendra de **cp** et **cr**. Deux situations pouvant alors se présenter :

1. Le fait déduit est nouveau, on l'intègre dans la base de fait avec le coefficient **cv** calculé.

2. Le fait était déjà présent affecté d'un coefficient $\mathbf{cv}_0$, on remplace $\mathbf{cv}_0$ par $f(\mathbf{cv}_0, \mathbf{cv})$.

Dans le cas de MYCIN, les coefficients varient dans l'intervalle $[-1, +1]$ avec l'interprétation suivante : -1 sûrement faux, $+1$ sûrement vrai, 0 indéterminé.

EXEMPLE 12.9.1

Soit la règle : (0.8) SI $cond_1 \wedge (cond_2 \vee cond_3)$ *ALORS conclusion. Où,* ***cr****=0.8. Supposons de plus que* $\mathbf{cc_1} = 0.85$, $\mathbf{cc_2} = 0.65$, $\mathbf{cc_3} = 0.75$ *le* ***cp*** *calculé sera min(0.85,max(0.65,0.75))=0.75 et supposons que le seuil pour cette règle soit de 0.7. Comme* ***cp*** *dépasse la valeur du seuil, la règle est déclenchée, on calcule alors* $\mathbf{cv_1}$ *par* $\mathbf{cp} \times \mathbf{cr}$. *Supposons, encore, que la conclusion soit connue avec un coefficient de vraissemblance à 0.75 on pourra alors définir le nouveau coefficient de vraissemblance de la conclusion à l'aide de la formule :* $\mathbf{cv} = \mathbf{cv_0} + \mathbf{cv_1} - (\mathbf{cv_0} \times \mathbf{cv_1}) = 0.75 + 0.56 - (0.75.0.56) = 0.89$

Bien entendu, on peut avoir des coefficients de vraissemblances de signe différent. De fait MYCIN *met en jeu les règles de calcul suivantes :*

$$\begin{cases} \mathbf{cv_0}, \mathbf{cv_1} & \geq 0 & \mathbf{cv_0} + \mathbf{cv_1} - (\mathbf{cv_0} \times \mathbf{cv_1}) \\ \mathbf{cv_0}, \mathbf{cv_1} & < 0 & \mathbf{cv_0} + \mathbf{cv_1} + (\mathbf{cv_0} \times \mathbf{cv_1}) \\ \mathbf{cv_0}, \mathbf{cv_1} & \text{sinon} & (\mathbf{cv_0} + \mathbf{cv_1})/1 - \min(|\mathbf{cv_0}|, |\mathbf{cv_1}|) \end{cases}$$

◁

REMARQUE : *Si on a les valeurs* $+1$ *et* -1, *on aboutit à une incohérence et des erreurs dans le calcul du nouveau coefficient de vraissemblance.* •

REMARQUE : *L'ordre dans lequel sont évalués les deux coefficients de vraissemblance, n'influe pas sur les formules de mise à jour des coefficients.* •

Ce type de calcul a été mis en œuvre avec succès dans le cadre du diagnostic médical, de conseils en placements financiers, d'identification de causes de pannes. En fait on mêle deux types de coefficients, d'une part au niveau « méta »les confiances dans les règles, d'autre part au niveau des éléments constitutifs de la base de connaissance. On ne peut plus parler de cohérence, ni de complétude au niveau logique. On n'a plus qu'une cohérence de type sémantique qui n'est, malheureusement pas vérifiable automatiquement.Le dernier problème avec ce type de raisonnement est que l'on « renforce »des opinions qui ne sont pas totalement indépendantes. Trois points sont à étudier dans ce type de démarche :

1. l'adéquation des faits et des prémisses,

2. la propagation de l'incertitude,

3. la combinaison de l'incertitude dans le cas de conclusions multiples.

Calcul Bayesien

Dans cette partie, on introduit rapidement la notion de calcul Bayesien (probabiliste) pour traiter les notions d'incertitudes. On notera **P**(x) la probabilité de l'évènement x. Classiquement, on définit **P**(vrai) = 1, **P**(faux) = 0, $\mathbf{P}(p \vee q) = \mathbf{P}(p) + \mathbf{P}(q) - \mathbf{P}(p \wedge q)$. Si q est conséquence logique de p alors **P**(q) $\geq$ **P**(p).

Lorsque les variables du domaine sont de nature aléatoire ou statistique, il semble naturel de s'appuyer sur un calcul de probabilités. Nous supposerons dans la suite que les hypothèses sont connues et en nombre fini. Il est alors possible de définir les notions suivantes :

DÉFINITION 12.2

Probabilité indépendante : $\mathbf{P}(p \wedge q) = \mathbf{P}(p)\ \mathbf{P}(q)$.

Définition 12.3
Probabilité conditionnelle : $\mathbf{P}(H \mid E) = \mathbf{P}(H \wedge E) \,/\, \mathbf{P}(E)$.

Définition 12.4 (formule de Bayes (1763))
$\mathbf{P}(H \mid E) = \mathbf{P}(H)\, \mathbf{P}(E \mid H) \,/\, \mathbf{P}(E)$.

Cette formule fonctionne si $\mathbf{P}(E)$ peut être estimée et si on a une seule hypothèse. On peut étendre la formule à plusieurs hypothèses, si on suppose que les H_i sont mutuellement exclusives, à l'aide de la formule suivante :

$$\mathbf{P}(H_i \mid E) = \frac{\mathbf{P}(E \mid H_i)\mathbf{P}(H_i)}{\sum_{j=1}^{n} \mathbf{P}(E \mid H_j)\mathbf{P}(H_j)}$$

$$\sum_{j=1}^{n} \mathbf{P}(H_j) = 1$$

Si E est de la forme $E_1 \wedge \ldots \wedge E_l$, la formule reste valide, mais on est alors confronté au problème de l'estimation de toutes les combinaisons possibles des E_k pour l'hypothèse H_i ; n hypothèses, l évènements conduisent à 2^{n+k} situations soit $2^n - 1$ probabilités à priori et $(2^l - 1)2^n$ probabilités conditionnelles. C'est pourquoi, on supposera, en général les E_k indépendants vis à vis des H_i i.e.

$$\mathbf{P}(E_k \wedge E_1 \mid H_i) = \mathbf{P}(E_k \mid H_i)\mathbf{P}(E_1 \mid H_i)$$

d'où la formule :

$$\mathbf{P}(H_i \mid E_1 \wedge \ldots E_l) = \frac{\mathbf{P}(E_1 \mid H_i) \ldots \mathbf{P}(E_l \mid H_i)\mathbf{P}(H_i)}{\sum_{j=1}^{n} \mathbf{P}(E_1 \mid H_j) \ldots \mathbf{P}(E_l \mid H_j)\mathbf{P}(H_j)}$$

Prospector (1981) dont le développement s'étala de 1974 à 1983, et qui fit appel à 9 experts différents pour un coût de l'ordre de 30 hommes année, est un système expert de type Bayesien. Nous allons décrire dans ce qui suit le mécanisme d'inférences et les calculs mis en jeu dans ce cadre, avant d'en voir les limitations.

Mécanisme inférentiel : Le raisonnement consiste à déterminer s'il y a lieu d'effectuer un forage pour trouver les minerais recherchés à partir de l'examen des données. Puisque le système est de type Bayesien, on doit considérer d'une part les probabilités à priori de trouver les minerais et d'autre part, les probabilités d'observer certaines caractéristiques étant donnée la présence de tel ou tel minerais. Le système peut alors en déduire la probabilité de présence du minerais. Le moteur d'inférence s'appuye sur un double mécanisme :

1. application du théorème de Bayes,
2. manipulation des degrés d'incertitude affectant les assertions sur lesquelles s'appuye une hypothèse donnée.

Calcul : Soit la règle « SI E ALORS H »à laquelle est adjoint un couple de coefficient (λ_S, λ_N). λ_S est appelé coefficient de suffisance et λ_N coefficient de nécessité.

L'observation E est une conjonction d'observation $E = (\bigwedge_{i=1}^{n} E_i)$ et H est une hypothèse, objets auxquels on affecte des probabilités à priori $\mathbf{P}(E_i)$ et $\mathbf{P}(H)$. Si les E_i sont indépendants on a $\mathbf{P}(E) = \mathbf{P}(E_1) \ldots \mathbf{P}(E_n)$, sinon $\mathbf{P}(E) = \min_i(\mathbf{P}(E_i))$. La vraissemblance, à priori de H est définie par :

$$\begin{aligned}
\mathbf{O}(H) &= \frac{\mathbf{P}(H)}{(1-\mathbf{P}(H))} \\
\lambda_S &= \frac{\mathbf{P}(E \mid H)}{\mathbf{P}(E \mid \neg H)} \\
\lambda_N &= \frac{\mathbf{P}(\neg E \mid H)}{\mathbf{P}(\neg E \mid \neg H)}
\end{aligned}$$

$\mathbf{O}(H)$, λ_S, λ_N sont fournis par l'expert lors de la construction du système.

Si $\lambda_S \gg 1$ la règle indique que la présence de E rend H fortement vraisemblable.
Si λ_S est proche de 0 la règle indique que E est suffisant pour que $\neg H$ soit vraisemblable.
Si $0 < \lambda_N \ll 1$ la règle indique que E est nécessaire pour H car l'absence de E rend H invraisemblable.
λ_S et λ_N sont indépendants mais reliés par $\mathbf{P}(E \mid H)$ et $\mathbf{P}(E \mid \neg H)$, on a en effet :

$$\begin{aligned}
\mathbf{P}(E \mid H) &= \lambda_S(1-\lambda_N) \quad / \quad (\lambda_S - \lambda_N) \\
\mathbf{P}(E \mid \neg H) &= (1-\lambda_N) \quad / \quad (\lambda_S - \lambda_N)
\end{aligned}$$

La vraisemblance, à postériori de H est établie par :

$$\begin{aligned}
\mathbf{P}(H \mid E) &= \lambda_S \mathbf{O}(H) \\
\mathbf{P}(H \mid \neg E) &= \lambda_N \mathbf{O}(H)
\end{aligned}$$

Si la connaissance de E est partielle (E'), on fait l'hypothèse simplificatrice selon laquelle l'influence de E' sur H passe par E et la probabilité à postériori est donnée par

$$\mathbf{P}(H \mid E') = \beta \mathbf{P}(H \mid E) + (1-\beta)\mathbf{P}(H \mid \neg E)$$

On peut prendre $\beta = \mathbf{P}(E \mid E')$ mais cela suppose que l'on ait $\mathbf{P}(E \mid E') = \mathbf{P}(E) \Longrightarrow \mathbf{P}(H \mid E') = \mathbf{P}(H)$, ce qui n'est jamais le cas en pratique. On peut résoudre le problème en prenant (comme dans Prospector) $\mathbf{P}(H \mid E')$ une fonction linéaire par morceaux qui respecte cette coïncidence.

Limitations de l'approche probabiliste : Plusieurs limitations voient le jour lorsque l'on choisit d'implémenter une telle approche :

1. l'approche Bayesienne se fonde sur 3 postulats qui ne sont pas réalistes :
   - hypothèses mutuellement exclusives,
   - exhaustivité
   - indépendance conditionnelle de l'observation dans les hypothèses.
2. elle exige de pouvoir estimer toutes les probabilités à priori,
3. les contraintes sur les valeurs des probabiliés (en particulier, la somme doit être égale à 1) rendent difficile l'évolution de la base de connaissance,
4. la propagation des probabilités est coûteuse en temps,
5. ne sait pas gérer l'ignorance.

Autres approches probabilistes

Il est possible de choisir d'autres modèles de calcul des probabilités, par exemple on pourrait prendre la mesure **g** définie de la manière suivante :

$$\begin{array}{lcl} \mathbf{g}(\text{vrai}) & = & 1 \\ \mathbf{g}(\text{faux}) & = & 0 \\ \mathbf{g}(q) & \geq & \mathbf{g}(p), \text{ si } q \text{ est conséquence logique de } p \\ \mathbf{g}(p \vee q) & = & \min(\mathbf{g}(p), \mathbf{g}(q)) \\ \mathbf{g}(p \wedge q) & = & \max(\mathbf{g}(p), \mathbf{g}(q)) \end{array}$$

12.9.3 Possibilités et raisonnement possibiliste

On définit deux mesures l'une dite de nécessité et notée **N**, l'autre de possibilité notée Π. Ces mesures introduites par Zadeh (1978) sont définies par les équations :

$$\begin{array}{lcl} \Pi(p \vee q) & = & \max(\Pi(p), \Pi(q)) \\ \mathbf{N}(p \wedge q) & = & \min(\mathbf{N}(p), \mathbf{N}(q)) \end{array}$$

elles vérifient l'ensemble des propriétés suivantes :

PROPOSITION 12.9.1

$$\begin{array}{lcl} \mathbf{N}(p) & \leq & \Pi(p) \\ \mathbf{N}(p) & = & 1 - \Pi(\neg p) \\ \mathbf{N}(p) + \mathbf{N}(\neg p) & \leq & 1 \\ \Pi(p) + \Pi(\neg p) & \geq & 1 \\ \min(\mathbf{N}(p), \mathbf{N}(\neg p)) & = & 0 \\ \max(\Pi(p), \Pi(\neg p)) & = & 1 \end{array}$$

Par ailleurs on a :

1. $\Pi(p) < 1 \Longrightarrow \mathbf{N}(p) = 0$
2. $\mathbf{N}(p) > 0 \Longrightarrow \Pi(p) = 1$

L'intervalle $[\mathbf{N}(p), \Pi(p)]$ caractérise l'incertitude sur p.

En matière de probabilité, $\mathbf{P}(p) = 1$ signifie que p est *certain* alors que $\Pi(p) = 1$ ne signifie pas que l'évènement est certain car l'évènement contraire $\neg p$ peut avoir une possibilité de 1 si $\mathbf{N}(p) = 0$. En revanche, $\mathbf{N}(p) = 1$ qui implique que $\Pi(p) = 1$ et $\Pi(\neg p) = \mathbf{N}(\neg p) = 0$ qualifie l'évènement p de certain.

La connaissance de $\mathbf{N}(p) < 1$ et $\mathbf{N}(\neg p) < 1$ correspond à une information incomplète, tandis que $\mathbf{N}(p) > 0$ et $\mathbf{N}(\neg p) > 0$ traduit une incohérence d'autant plus forte que le minimum des deux est proche de 1 ; p et $\neg p$ se rapprochent de la certitude. L'ignorance totale se traduisant par $[\mathbf{N}(p), \Pi(p)] = [0, 1]$.

EXEMPLE 12.9.2

MYCIN *(1976) développé à partir de 1974, possède un moteur qui fonctionne par chaînage arrière en recherchant les maladies possibles à partir des symptômes. Soit* $\mathbf{P}(H \mid E)$ *la probabilité d'un diagnostic* H *compte tenu d'un ensemble de symptômes* E*. Soit* $\mathbf{P}(H)$ *la probabilité à priori d'un diagnostic* H *sans référence à une observation, le théorème de Bayes permet de réactualiser la probabilité de* H *à chaque observation. Pour simplifier :* $\mathbf{P}(H \mid E)$ *est traitée comme une règle* `SI E ALORS H` *avec un coefficient de confiance* $\mathbf{CF}(H, E)$ *fourni par l'expert lors de l'élaboration du système et en accord avec leur expérience clinique. Lorsque* $\mathbf{P}(H \mid E) = \mathbf{P}(H)$*,* E *n'a pas d'incidence sur* H *et* $\mathbf{CF}(H, E) = 0$*, si* $\mathbf{P}(H \mid E) > \mathbf{P}(H)$ *la présence de* E *accroît la probabilité de* H *; c'est l'inverse lorsque la valeur est inférieure. Pour distinguer ces deux cas on définit deux quantités notées* $\text{MB}(H, E)$ *(croyance*[1]*) et* $\text{MD}(H, E)$ *(non-croyance*[2]*) en supposant*

[1] belief
[2] disbelief

une relation linéaire dans les intervalles $[0, \mathbf{P}(H)]$ *et* $[\mathbf{P}(H), 1]$. *La quantité* $\mathbf{P}(H)$ *étant supposée distinctes de* 0 *et* 1. *On a ainsi :*

- *Pour* $\mathbf{P}(H \mid E) \geq \mathbf{P}(H)$:
  $\mathrm{MB}(H, E) \geq 0$ *et* $\mathrm{MD}(H, E) = 0$.
- *Pour* $\mathbf{P}(H \mid E) \leq \mathbf{P}(H)$:
  $\mathrm{MD}(H, E) \geq 0$ *et* $\mathrm{MB}(H, E) = 0$.
- *Le facteur de certitude* $\mathbf{CF}(H, E)$ *est relié à ces deux quantités par :* $\mathbf{CF}(H, E) = \mathrm{MB}(H, E) - \mathrm{MD}(H, E)$. *Il varie entre* -1 *et* $+1$. *Comme au moins l'une des deux quantités est nulle, l'expert n'a besoin de fournir qu'une valeur numérique pour chiffrer sa confiance dans la règle* `SI E ALORS H`

On dispose aussi des règles déjà présentées dans l'exemple 12.9.1 page 134. ◁

Conclusion

En notant $\Pi(p \mid q)$ la possibilité d'avoir p sachant q on peut définir :

1. $\Pi(p \mid q) = 1 - \mathrm{MD}(p, q)$
2. $\Pi(\neg p \mid q) = 1 - \mathrm{MB}(p, q)$
3. ce qui est cohérent avec :

$$\begin{cases} \mathrm{MB}(\neg p, q) & = & \mathrm{MD}(p, q) \\ \min(\mathrm{MB}(p, q), \mathrm{MD}(p, q)) & = & 0 \end{cases}$$

L'un des coefficients est toujours nul, il suffit à l'expert de fournir un seul coefficient pour chaque règle. Positif ce coefficient ira dans le sens de la confiance de la règle ; négatif, il ira dans le sens de la défiance. Il s'agit de facteurs de certitude de nature empirique, rendant difficile la vérification du bon emploi des formules dans la combinaison des facteurs. Malgrè tout cette technique à fait ses preuves dans la pratique.

12.9.4 Logique floue

On s'intéresse maintenant au cas où l'on traite des propositions à caractères floues ou d'assertions floues dans leur expression. Pour en savoir plus sur ces différents concepts, je vous renvoie aux livres suivants qui m'ont servi à élaborer cette partie [12, 95, 38, 45] et à l'introduction des concepts de base abordés dans le chapitre 6 de la partie II de ce pavé.

12.10 Conclusion

Dans cette dernière partie nous allons faire le point sur les éléments clefs à considérer lorsque l'on décide de réaliser un SE.

12.10.1 Conditions pour la mise en « chantier »

La règle peut se résumer par :

Le développement d'un système expert doit être possible, justifié et approprié

Possible :

- s'il existe des experts du domaine,
- si les experts sont capables d'expliciter leurs méthodes,
- si les experts sont d'accord sur la façon de résoudre un problème, c'est-à-dire s'il existe une théorie reconnue par tous (pas de SE sur la divination, les horoscopes, le loto, la diététique, ...),
- si les données sont cognitives et non expérimentales, on ne doit pas faire d'expériences pour résoudre le problème,
- le problème ne doit pas être trop complexe,
- le domaine doit être précis, restreint,
- la résolution ne doit pas faire appel au sens commun.

Justifié :

- la méthode actuelle est coûteuse,
- pas d'experts humains disponibles,
- expertise pouvant disparaître, ou ayant disparue,
- expertise en milieu hostile.

Approprié :

- nature symbolique raisonnement/données,
- compléxité importante (car il y a un coût important du développement (cf PROSPECTOR 30 hommes année),
- problème de bonne taille et ayant une valeur pratique (pour l'entreprise),
- modulaire pour rester de taille raisonnable.

12.10.2 Phases indispensables

Elles sont au nombre de 5 : identification, conceptualisation, formalisation, implémentation, tests.

Identification :

- caractéristiques du problème,
- personnes participant au développement,
- ressources nécessaires,
- but/objectif à atteindre.

Conceptualisation :

- l'expert et le cogniticien décident des concepts, relations, mécanismes de contrôle nécessaires,
- exploration des sous-problèmes,
- granularité de la résolution.

Formalisation :
• choix du langage permettant de mettre en œuvre les concepts définis dans la phase précédente.

Implémentation :

- écriture informatique,
- structures de données, règles d'inférences, contrôle,
- réalisation d'un prototype.

Tests :

- évaluation des performances sur des problèmes types,
- le système prend-il des décisions agréées par les experts,
- règles d'inférences correctes, consistantes, complètes,
- explications adéquates,
- tests recouvrant le domaine,
- le système est-il utile,
- le système est-il rapide,
- l'interface est-elle conviviale, adéquate, robuste aux erreurs, aux mauvaises expressions.

12.10.3 Différents types de systèmes

Dans cette partie nous essayons de définir la nature des prototypes réalisables en termes d'efficacité, de temps de développements, de taille (nombre de règles et de faits), de nombre de problèmes pour lesquels le système apporte une réponse.

- **Prototype de démonstration** : ne traite qu'une partie du problème. Sert à montrer la faisabilité de l'approche. Il contient entre 50 et 100 règles, résoud correctement 1 ou 2 cas, nécessite de 1 à 3 mois de développement.
- **Prototype de recherche** : de taille moyenne, ayant des performances crédibles sur un certain nombre de tests. Il est fragile, et peut se tromper si on sort du cadre des problèmes qu'il sait résoudre, trop peu tester il peut même se tromper sur des cas prévus. Il contient entre 200 et 500 règles, résoud correctement un grand nombre de problèmes classiques, nécessite de 1 à 2 ans de développement.
- **Prototype du domaine** : de taille moyenne ou grande, abondamment testé sur des cas réels, à peu prés fiable, ayant une interface pratique et servant aux utilisateurs. Il a de 500 à 1000 règles, résoud très bien de nombreux cas classiques, nécessite de 2 à 3 ans de développement.
- **Prototype de production** : programme de démonstration, très largement testé, réimplanté dans un langage rapide et en optimisant le stockage. Il contient de 500 à 1500 règles, prend des décisions rapides et efficaces, nécessite de 2 à 3 ans de développement.
- **Système commercial** : rare, en général s'est un prototype de production utilisé en milieu commercial. Il contient plus de 3000 règles, atteint de bonnes conclusions dans plus de 90% des cas, nécessite de 5 à 6 ans de développements intensifs.

Chapitre 13

Algorithmes génétiques

Les algorithmes génétiques sont des algorithmes d'exploration fondés sur les mécanismes de la sélection naturelle et de la génétique. Ils sont basés sur les principes de survie de structures les mieux adaptées et les échanges d'information. À chaque génération, un nouvel ensemble de créatures artificielles (codées sous forme de chaînes de caractères) est construit à partir des meilleurs éléments de la génération précédente. Bien que reposant fortement sur le hasard (et donc sur un générateur de nombres aléatoires) ces algorithmes ne sont pas purement aléatoires.

Ce chapitre se veut une introduction de cette approche, la littérature étant extrêment prolyxe, nous renvoyons le lecteur d'une part sur le forum usenet `comp.ai.genetic` et sa FAQ. La meilleure production en français est sans conteste le livre de David Goldberg qui commence à dater (une dizaine d'années) [41]. Le lecteur anglophone pourra aussi reporter avec bonheur sur les articles introductifs suivants [8, 88, 99].

13.1 Introduction

L'appellation Algorithmes Evolutionnaires (EAs) est un terme générique pour décrire les approches informatiques reposant sur les principes de l'évolution comme éléments clefs de leurs fonctionnements. Il existe un grand nombre d'algorithmes évolutionnaires : algorithmes génétiques, programmation évolutive, stratégies évolutives, systèmes de classifieurs et programmation génétique. Toutes ces approches ont en commun la simulation de l'évolution de structures individuelles au travers de méthodes de *sélection*, *mutation* et *reproduction*. Ces processus reposant sur la *performance* des structures individuelles à leur *environnement*. Plus précisément, les EAs maintiennent une population de structures, qui évoluent en accord avec des règles de sélection et d'autres opérateurs, appelés opérateurs de recherche. Chaque individu possède une mesure de son adéquation avec l'environnement : la *fitness*. La reproduction s'appuye sur cette mesure pour favoriser la prolifération des individus les mieux adaptés. La recombinaison et la mutation perturbent ces individus fournissant par ce biais des heuristiques pour l'exploration de nouvelles solutions. Bien que relativement simple ces algorithmes sont suffisants pour assurer des mécanismes de recherche robustes et puissants. Le pseudo-code suivant montre le fonctionnement d'un EA :

```
// initialisation du temps
t := 0
// initialisation aléatoire de la population
initPopulation(P,t)
// evaluation de la fitness individuelle
evaluate(P,t)
// un critere d'arret dependant du temps, de la fitness
Loop
 Exit when done
 t := t + 1
```

```
 Q := selection(P,t)
 recombinaison(Q)
 mutation(Q)
 evaluate(Q,t)
 P := survivants(P,Q,t)
EndLoop
```

Pour les algorithmes génétiques, les individus sont représentés par leurs chromosomes (sous forme de chaîne de caractères). La programmation évolutive, conçue initialement par J. Fogel en 1960, est une stratégie d'optimisation stochastique. À la différence des algorithmes génétiques, elle ne repose pas sur un codage particulier des individus (on peut l'appliquer directement sur les structures que l'on veut faire évoluer - réseaux de neurones par exemple), elle s'intéresse aux relations existant entre parents et enfants plutôt qu'à chercher à émuler des opérateurs génétiques particuliers. Un système de classifieurs peut çetre vu comme un système cognitif qui réagit à son environnement (à mettre en parallèle avec l'approche animat). Pour cela, on dispose d'un environnement, de capteurs permettant de connaître l'environnement (ce qui s'y passe), d'effecteurs (pour agir) et d'un système fonctionnant comme une boîte noire fonctionnant comme un système de production (règle `si-alors`). Les informations provenant des capteurs sont interprétées comme des messages, les règles réagissent à ces messages, et font des enchères, la meilleure enchère est sélectionnée, la règle correspondante est choisie (déclenchée) et la partie `alors` renvoie les informations aux effecteurs. Lorsqu'une règle est choisie, elle paye une contribution aux règles qui ont participé aux enchères mais qui n'ont pas été sélectionnées. La programmation génétique est une extension des algorithmes génétiques aux programmes. C'est-à-dire que les individus sont des programmes exprimées sous forme d'arbres syntaxiques, les langages de la famille LISP se prêtent particulièrement bien à cette approche. L'opérateur de croisement agit en échangeant des sous-arbres, le lecteur interessé est renvoyé à la page de Koza `www.genetic-programming.com/johnkoza.html`, ou au site `www.genetic-programming.org`.

Des informations plus précises se trouve dans la faq de `com.ai.genetic`.

13.2 Objectif

Les algorithmes génétiques sont des algorithmes d'exploration fondés sur la sélection naturelle et la génétique. Ils utilisent les principes de la survie des structures les mieux adaptées, les échanges d'informations pseudo-aléatoires. Ils reposent de manière intensive sur le hasard mais ne sont pas purement aléatoires. Développés à l'origine par J. Holland et son équipe au sein de l'université du Michigan [47]. La recherche des algorithmes génétiques a pour but : l'amélioration de la robustesse et l'équilibre entre la performance et le coût nécessaire à la survie dans un environnement. Ils sont utilisés comme alternative à l'optimisation de fonction lorsque les méthodes "classiques", telles que les méthodes indirectes qui recherchent à atteindre les extrema locaux en résolvant des sytèmes d'équations et les méthodes directes par suivi de gradient ou par énumération, ne sont pas applicables ou ont échoué. Ce qui différencie les AG des autres approches peut s'exprimer en quatre points :

1. Les AG utilisent un codage des paramètres et non les paramètres eux-mêmes.
2. Les AG travaillent sur une population de points et non sur un point particulier.
3. Les AG n'utilisent que les fonctions étudiées, pas leur propriété (telle que la dérivabilité ou autre).
4. Les AG utilisent des règles de transitions probabilistes et non déterministes.

13.3 Fonctionnement simplifié

Les mécanismes mis en jeu sont très simples, on dispose d'une population de chaînes de caractères sur un alphabet que nous supposerons binaire. À chaque individu est associé un score (fitness),

on en profite pour calculer la somme de ces scores ainsi que les valeurs maximale, minimale et moyenne. Trois opérateurs vont être utilisés : un opérateur de sélection (`select`) dont le rôle est de choisir les individus les plus adaptés (ceux possédant un score élevé), un opérateur d'appariement (`cross-over`), enfin un opérateur de mutation (`mutation`). L'algorithme génétique peut s'exprimer sous la forme suivante :

```
t := 0
P := initPopulation
Loop
 evaluate(P,t)
 Exit when critere d'arret
 Q := select(P,t)
 R := cross-over(Q)
 P := mutation(R)
 t := t+1
EndLoop
```

Nous allons revenir maintenant sur les différents constituants des algorithmes génétiques. Avant de pouvoir utiliser un AG, il nous faut définir un codage adapté au problème (les *chromosomes*), ainsi qu'une fonction (*fitness*) qui va caractériser l'adéquation de la solution (représentée par son chromosome) au problème.

13.3.1 Chromosomes

On suppose dans cette approche qu'une solution du problème peut etre représentée par un ensemble de paramètres, ces paramètres sont regroupés sous la forme d'une chaîne de caractères. J. Holland a le premier montré pourquoi une représentation sous forme binaire était efficace, bien qu'il existe d'autres approches, nous nous restreindrons dans ce qui suit à un alphabet binaire.
En génétique, un ensemble de paramètres représentés par un chromosome particulier est appelé son *génotype*, le *phénotype* est quant à lui une instance particulière d'un génotype. La fonction d'évaluation dépend du phénotype.

13.3.2 Fitness

Cette fonction est déterminée en fonction du problème à résoudre et du codage choisi pour les chromosomes. À un chromosome particulier, elle attribue une valeur numérique, qui est supposée proportionnelle à l'intérêt de l'individu en tant que solution. Dans le cas d'une optimlisation de fonction, la fitness est de façon triviale la fonction que l'on cherche à optimiser, mais cela n'est pas toujours le cas. Ainsi dans l'exemple que nous traiterons dans la section 13.5, nous nous intéressons à la survie d'un organisme cherchant à accomplir une tâche (recherche d'objets et évitement d'obstacles) ; dans ce cas la fonction d'évaluation sera une composition de l'âge atteint par la bestiole et du nombre d'objets trouvés, pénalisé proportionnellement au nombre d'obstacles rencontrés.

13.3.3 Selection

Pour cette phase, nous supposons connu : les individus de la population au temps t, la fonction d'évaluation. Il va falloir trouver une manière de chosir les individus répondant le mieux à notre problème. Il s'agira donc de privilégier les individus ayant un score au-dessus de la moyenne, et de pénaliser ceux en-deça de cette moyenne. Plusieurs approches sont possibles :

1. La roue de la fortune : On associe à chaque individu A_i son évaluation f_i, on constitue une roue dont la valeur maximale est donnée par $\sum_i f_i$. On applique alors l'algorithme suivant :

```
function Wheel(int p){
  sigma := ∑_i  f_i
  S = ∅
  for(j = 0 ; j < p ; j++){
   k = 0 ;
```

```
    x = random(sigma);
    y = sigma;
    for(i = 0; i<n && y > x; i++){
      y = y - f_i;
      k = i;
    }
    S = S ∪ { A_k };
  }
}
```

2. Méthode des fractions : On effectue le calcul en deux étapes, dans un premier temps, chaque individu est sélectionné autant de fois que la valeur de la partie entière du rapport entre son score f_i et du score moyen de la population $\overline{f} = \frac{\sum_i f_i}{n}$. Dans un second temps on constitue une roulette à partir des parties décimales de chaque $\frac{f_i}{\overline{f}}$.

3. Méthode élitiste : on force la population sélectionnée à contenir les k meilleurs individus de la population, puis on complète par l'une des méthodes précédentes.

4. Recalage de la fitness : en fait cette méthode est un préalable aux autres méthodes, elle consiste à recalibrer les scores effectifs soit par compression (pour éviter que les éléments les meilleurs soient trop privilégiés), soit par décompression (pour favoriser les éléments les meilleurs). Soit par une méthode de "recentrage", typiquement il s'agit d'ordonner les individus en fonction de leur score et ensuite on sélectionne les différents éléments soit par une stratégie linéaire, soit une stratégie exponentielle. D'une manière générale ces méthodes sont utilisées après étude mathématique des valeurs de fitness et en fonction d'un objectif qui est soit la convergence accéléree (que l'on priviligiera principalement en fin d'évolution, soit au contraire en vue d'éviter une convergence prématurée et donc principalement lors des premières générations).

13.3.4 Cross-Over

Le cross-over est une méthode permettant de construire de nouveaux individus à partir de leurs parents. Il existe deux grandes approches, le cross-over simple qui consiste à choisir un point de croisement compris entre 1 et $(l-1)$ où l est la longueur de la chaîne et à recombiner les enfants comme étant le début d'un chromosome et la fin de l'autre. Par exemple, si l'on considère les deux chromosomes suivants `00110011` et `10101010` et que l'on choisisse comme point de croisement la valeur 3, on obtiendra les deux nouveaux individus suivants : `001 01010` et `101 10011`. La seconde approche est le cross-over double, qui consiste à sélectionner deux points de croisements et à échanger les parties intermédiaires. Ainsi en reprenant l'exemple précédent et en considérant les points 3 et 6 on obtient les deux nouveaux individus : `001 010 11` et `101 100 10`. À noter que le cross-over simple peut être considéré comme un cross-over double si l'on "voit" les chromosomes comme des chaînes circulaires, et le deuxième point de croisement comme étant en position l.

13.3.5 Mutation

La mutation est appliquée individuellement à tous les "enfants" après le croisement. Elle a pour but d'altérer aléatoirement une information avec une faible probabilité (de l'ordre de 0.001). L'intérêt de la mutation et de pouvoir explorer aléatoirement une autre portion de l'espace de recherche, garantissant ainsi que tous les points de l'espace ont une probabilité non nulle d'être essayer.

13.3.6 Convergence

Si l'AG a été correctement implémenté, la population évolue petit à petit au cours des générations successives et l'on constate que le score moyen de la population évolue globalement vers l'optimum de la fonction à maximiser. La *convergence* est la progression vers cet optimalité. Un gêne sera dit comme ayant convergé si 95% de la population possède cette caractéristique. La population a convergé lorsque tous les gênes ont convergé.

13.4 Quelques applications

Les algorithmes génétiques ont été appliquer avec succès à la recherche de solution dans un jeu de stratégie, l'optimisation de réseaux de neurones [103, 104], l'optimisation de fonctions (travaux de De Jong), la simulation de cellules biologiques, la reconnaissance de formes. L'étude d'animat au moyen de classifieurs.

13.5 Algorithme génétique pour animat en *Logo

Cette partie développe une application en NetLogo pour l'évolution d'agents cognitifs dans un environnements simplifiés. Il s'agit de la reprise d'une expérience menée par Dan Kunkle dont un descriptif en anglais se trouve sur `www.rit.edu/~drk4633/dmt/`. Après avoir décrit brièvement l'approche proposée, nous donnerons des extraits de la réalisation effectuée. On trouvera en annexe E le programme complet, et dans le chapitre 9 quelques éléments de programmation.

13.5.1 Description sommaire

On dispose d'un environnement (sous la forme d'un tore) pavé d'agents immobiles que l'on appelle *patches* capables de contenir de la nourriture représentée par une variable numérique entière `food`, des obstacles modélisés par une couleur (rouge) ou des objectifs modélisé par une couleur (bleu). Dans cet environnement des bestioles *turtles* se déplacent. Ces bestioles sont caractérisées par un niveau d'énergie, modélisé par une variable numérique entière `energy`, une vitesse `speed`, une vision modélisée par sa distance et notée `vision`. Ces tortues disposent d'autres variables sur lesquelles elles ne peuvent pas agir : leur `age`, leur adéquation à l'environnement `score`, la liste des endroits qu'elles ont visités et où il y avait un objectif `lgoals`, la liste des endroits visités où il y avait un obstacle `lmines`, la distance parcourue au cours de leur vie `dist`, leur génôme `genome`. Ces bestioles ont des capteurs leur permettant de détecter la nourriture, les buts et les obstacles.
L'objectif est d'obtenir par sélection (via algorithmes génétiques) des bestioles capables de survirvre dans ce type d'environnement en collectant le plus d'objectifs possibles et en évitant au mieux les obstacles.
Pendant leur "vie", les bestioles doivent évaluer à chaque pas de temps l'état dans lequel elle se trouve. Cet état dépend d'informations provenant de l'environnement par le biais de leurs capteurs. Une fois déterminer leur état, elles effectuent les actions correspondant à cet état et codées dans leur chromosome. Ces actions n'influent que sur les caractéristiques suivantes :

1. Leur direction de déplacement.
2. Leur vitesse de déplacement.
3. Leur vision (qui influe sur la performance de leurs capteurs).

Un certain nombre d'actions ont été prédéfinies : `nope` ne fait rien ; `speedup` augmente de 1 la vitesse de déplacement (en fait le nombre de pas maximum autorisé au cours d'une unité de temps) ; `slowdown` décrémente de 1 la vitesse de déplacement ; `resetspeed` arrêt ; `head-food` s'oriente vers la case voisine (dans un rayon limité par `vision`) ayant la plus grande quantité de nourriture ; `random-heading` choisit aléatoirement une nouvelle direction ; `half-turn` fait un demi-tour sur elle-même ; `head-towards-friends` s'oriente dans la direction où il y a le plus de congénères ; `head-towards-mine` s'oriente vers la direction offrant le plus de mines ; `head-towards-smell` s'oriente en fonction du gradiant d'odeur associé à la nourriture ; `head-towards-goal` oriente vers la zone possédant le plus de buts ; `increase-vision` augmente la vision d'un cran ; `decrease-vision` fonction inverse de la précédente ; `run-away- goal, mine, friends` recherchent la direction opposé à l'opération `head-towards-`.

Puis, en fonction de leur vitesse (un entier) elle répète la séquence :

1. Faire un pas

2. Evaluer leur santé
3. Se nourrir

Faire un pas, affecte leur niveau énergétique (paramètre fixé par l'utilisateur), l'évaluation de leur santé à pour but de savoir si elle meurt ou reste en vie, l'absorbtion de nourriture augmente leur énergie d'une quantité fixé par l'utilisateur.

L'évaluation des bestioles se fait de la manière suivante :

1. On initialise un monde avec les ressources positionnées aléatoirement.
2. On initialise la population de bestioles à des positions aléatoires et avec un génome aléatoire.
3. La simulation s'arrête lorsqu'il ne reste plus aucune bestiole en vie.
4. On évalue la performance de chaque bestiole, on calcule la moyenne et les valeurs maximales et minimales

Le chromosome d'un individu est donc une liste correspondant à chaque état possible dans lequel pourra se trouver la bestiole (dans cet exemple il y a 64 états possibles) et pour chaque état on associe une liste d'actions possibles (le nombre maximum est fixé par l'utilisateur). La fonction de fitness doit tenir compte des différents paramètres à optimiser, c'est la fonction `fitness-value` qui ici cherche à privilégier le nombre de buts différents trouvés, l'âge atteint, la distance parcourue et la quantité de nourriture absorbé, tout en pénalisant le nombre d'obstacles rencontrés et la période d'immobilité.

13.5.2 Quelques méthodes

Dans cette partie nous allons décrire quelques méthodes :

run-one-generation permet de faire une simulation avec la population de tortues. Tant que l'on n'a pas atteint le maximum d'itérations et qu'il y a encore des tortues en vue. On fait la mise-à-jour des patches, on diffuse l'odeur de la nourriture, on demande au tortues vivantes d'agir. Si une tortue est cachée `hidden?` elle change d'espèce, sinon on évalue l'effet de l'âge.

init-pool-genome initialise la variable globale `pool-genome` l'élément en position i correspond au génôme de la tortue ayant l'identificateur `who` = i.

make-pool-fitness calcule le score minimum et maximum de fitness pour tous les génômes, et stocke tous les scores dans la variable `pool-fitness`, le ième score correspond au ième génôme dans `pool-genome`.

select-A implémente une méthode de sélection par roue de la fortune.

select-B implémente une méthode de sélection par fractions.

generate-pool prend en paramètre une liste de numéro de tortues (une tortue peut être présente en plusieurs exemplaires), une liste d'index (de 1 à n), et renvoie la nouvelle population de génômes. Supposons que la première liste contienne les informations suivantes `[1 1 2 3 3 4]` la seconde liste étant `[1 2 3 4 5 6]`, on pioche aléatoirement les valeurs `1` et `5` dans la deuxième liste, cela a pour effet de choisir de croiser les tortues 1 et 3. Lorsque l'on rappellera cette méthode on aura les valeurs de listes suivantes : `[1 1 2 3 3 4]` et `[2 3 4 6]`. À chaque itération la liste est diminuée de deux, et une même instance ne pourra pas être choisie plusieurs fois.

cross-over effectue le croisement entre les génômes des deux numéros de tortues passées en paramètres, le résultat du croisement étant stockés dans le troisième argument.

mutation Elle s'effectue sur le `pool-genome` qui servira à initialiser les tortues pour la prochaine simulation. La mutation consiste à chosir une action et à la remplacer par une autre. La taille du génôme n'est pas altérée.

13.6 Un algorithme génétique en Scheme

Cette partie développe une application en Scheme d'un algorithme génétique recherchant une configuration de 32 interrupteurs. Le programme complet est fourni en annexe E. Explorer toutes les possibilités de l'espace correspond à essayer 2^32 configurations soit `4 294 967 296`. L'implémentation proposée n'a pas pour but d'être efficace, juste de montrer comment mettre en œuvre des AG en SCHEME. Un chromosome est une paire pointée dont le premier champ est un vecteur de 32 bits (correspondant aux différentes positions des interrupteurs) le second champ est un vecteur de deux valeurs, la première est la fitness comprise entre 0 et 100, le second est un rééchelonnage de la fitness pour privilégier les scores au-desus de la moyenne. La population est représentée par une liste, dont le premier élément est un vecteur de deux composantes correspondant à la somme des fitness individuelles, les autres éléments étant des paires individu score. Nous allons maintenant, brièvement décrire les fonctions définies dans le programme

Variables globales On utilise trois variables globales `laSolution` qui est la configuration à trouver ; `secret` qui est le nombre d'interrupteurs augmenté du nombre d'interrupteurs en position `On` dans `laSolution` ; `chromoSz` le nombre d'interrupteurs (ici 32) utilisés dans une configuration.

true-fitness prend en paramètre un chromosome est renvoie sa fitness calculée en comptant le nombre d'interrupteurs ayant la même position que la solution, augmenté du nombre de fois où les interrupteurs sont sur la position `On`=1. Par exemple si on suppose que `chromoSz` = 4, que la configuration à trouver est `1001` on aura `secret` = 4 + 2 = 6. Si on évalue la configuration `1000` on obtiendra 3 + 1 = 4.

eval-fitness ramène le score entre 0 et 100 en calculant $\frac{\texttt{true-fitness}*100}{\texttt{secret}}$. Ainsi, en reprenant l'exemple précédent on trouve 66.66.

build-chromosome-alea génère un vecteur aléatoire de `chromoSz` bits.

mutation altère un bit quelconque dans la configuration.

simple-cross-over prend trois paramètres, 2 individus à croiser et la taille de ces individus. Elle retourne une liste contenant les deux vecteurs croisés.

double-cross-over effectue un croisement en deux points, et en échangeant les parties centrales des chromosomes.

cross-over prend en entrée le type de croisement qui peut être 1 ou autre, et les deux chromosomes à croiser. Elle retourne le résultat du croisement.

cross-list prend en entrée un type de croisement est une liste de chromosomes, elle renvoie la liste augmentée des croisements effectués entre les individus $2i$ et $2i + 1$ pour i variant de 1 à taille de la liste / 2.

init-population prend en paramètre la taille de la population est génère le nombre d'individus demandés tout en calculant, au fur et à mesur la fitness de la population.

make-stat prend en entrée une population et renvoie un quadruplet contenant, le meilleur individu, le score minimum, le score moyen, le score maximum.

affiche-resultat prend en paramètre une liste de quadruplets obtenus par `make-stat`, et un nombre et affiche autant de quadruplet que la valeur du nombre.

select choisit parmi la population un certain nombre d'individus (quantité passée en paramètre) pour constituer le pool génétique qui sera ensuite transmis au cross-over et à la mutation.

one-run est le programme qui permet de faire une itération de l'algorithme génétique, il a besoin de connaître la population, la taille de la population (pour éviter un re-calcul), le type de cross-over, la probabilité de mutation, si on veut faire une version optimisée de l'algorithme, et le meilleur chromosome. Si on est en mode "optimisé", on a une stratégie élitiste, qui consiste à préserver le meilleur élément, et pour la sélection on s'appuye sur la fitness modifiée (dans laquelle on a augmenté les écarts entre les différentes fitness) - cette modification est effectuée dans `build-next-population`.

ag est la fonction principale du programme, elle prend en paramètre la taille de la population, le nombre d'itérations, le type de croisement à effectuer pour construire les descendants, la probabilité de croisement (comprise entre 0 et 10), si on veut effectuer une version optimisée du calcul, et enfin le nombre d'affichage que l'on veut en sortie. Elle contrôle les différents paramètres et elle fait appel aux fonctions `one-run`, `make-stat` et à la fin `affiche-resultat`.

Exemple 13.6.1

Ce petit exemple a pour but de montrer une execution type du programme

```
12:21:59 hobbes-mmc-bash:  guile -l ag-simple.scm
done
guile> (ag 100 50 1 0 0 2)
; un point de croisement, pas de mutation, pas d'optimisation
-----------------------
(#(0 1 1 1 1 1 1 0 1 1 0 0 1 1 1 1 0 1 0 1 0 1 1 1 1 0 1 1 1 1 1 1) .
 #(88.4615384615385 57.6923076923077))
iterate : 50
minimum : 42.0
moyenne : 62.0
maximum : 88.0
-----------------------
(#(0 1 1 1 1 1 1 0 0 1 0 0 1 1 1 1 0 1 0 1 0 1 1 1 1 0 0 1 1 1 1 1) .
 #(88.4615384615385 61.5384615384615))
iterate : 49
minimum : 42.0
moyenne : 61.0
maximum : 88.0
-----------------------
guile> (ag 100 50 1 5 0 2)
; un point de croisement, mutation 1/2, pas d'optimisation
-----------------------
(#(0 1 0 0 1 1 1 1 0 0 1 0 1 0 1 1 1 1 0 0 0 1 0 1 1 0 1 1 1 0 1 1) .
 #(84.6153846153846 71.1538461538462))
iterate : 50
minimum : 53.0
moyenne : 70.0
maximum : 84.0
-----------------------
(#(1 1 0 1 1 1 1 1 0 0 1 0 1 0 1 1 1 1 0 0 0 1 0 1 1 1 1 0 1 0 1 1) .
```

```
 #(82.6923076923077 82.6923076923077))
iterate : 49
minimum : 48.0
moyenne : 69.0
maximum : 82.0
-----------------------
guile> (ag 100 50 1 5 1 2)
; un point de croisement, mutation 1/2, optimisation
-----------------------
(#(1 0 0 1 0 1 0 0 0 1 0 0 1 0 0 1 0 0 1 1 0 1 0 1 1 1 0 0 0 1 0 1) .
 #(98.0769230769231 3901.44378698225))
iterate : 50
minimum : 34.0
moyenne : 68.0
maximum : 98.0
-----------------------
(#(1 0 0 1 0 1 0 0 0 1 0 0 1 0 0 1 0 0 1 1 0 1 0 1 1 1 0 0 0 1 0 1) .
 #(98.0769230769231 3214.01775147929))
iterate : 49
minimum : 40.0
moyenne : 75.0
maximum : 98.0
-----------------------
```

◁

13.7 Le théorème des Schèmes

La mise en œuvre des algorithmes génétiques est relativement aisée. On commence avec un epopulation aléatoire de n chaînes, on copie les chaînes avec un biais pour privilégier la sélection des plus adaptées. On les apparient en échangeant des sous-chaînes, puis on altère certaines informations à l'aide de l'opérateur de mutation. Nous avons remarqué que ce traitement explicite des chaînes produit en fait un traitement implicite d'un grand nombre de schèmes à chaque génération. Nous allons dans cette partie analyser le développement et la disparition des schèmes dans la population, pour cela nous allons reconsidérer les différentes opérations au travers de leur impact sur les schèmes.

On suppose, dans ce qui suit que les chaînes sont constituées sur un alphabet binaire $V = \{0, 1\}$. On étend cet alphabet avec un caractère $\star$ qui remplace indifféremment n'importe quelle lettre.

DÉFINITION 13.1
Un schème de longueur l est une chaîne définie sur un alphabet quelconque $V \cup \{\star\}$.

Par exemple à la chaîne `101` on pourra lui faire correspondre les 8 schèmes suivants : `1**`, `*0*`, `**1`, `10*`, `1*1`, `*01`, `***`, `101`. De manière générale pour une chaîne de longueur l sur un alphabet de k lettres on pourra constituer k^l chaînes de valeurs différentes, et en introduisant le caractère $\star$, on pourra constituer $(k + 1)^l$ chaînes différentes. À partir d'une chaîne de longueur l, on pourra représenter 2^l schèmes distincts (pour construire un schème, on prendra pour chacune des l positions, le caractère de la chaîne ou le symbole $\star$). Enfin, dans une population de n chaînes on aura au plus $n2^l$ schèmes différents.

Les schèmes ne sont pas tous égaux : pour montrer ce résultat, nous allons dans un premier temps définir deux mesures sur les schèmes

DÉFINITION 13.2
L'ordre d'un schème H, noté o(H) est le nombre de positions instanciées.

DÉFINITION 13.3
La longueur (utile) d'un schème H, notée δ(H) est la différence entre les positions instanciées les plus extrèmes.

Ainsi pour le schème H = 1*001**1, on trouve o(H) = 5 et δ(H) = 8 - 1 = 7.

Notons m(H,t) le nombre de schèmes H dans la population au temps t. Un individu A_i est sélectionné avec une probabilité

$$p_i = \frac{f_i}{\sum_j f_j}$$

Pour une population de taille n, le nombre de représentant de H au temps t+1 est : m(H,t+1) = m(H,t) $n \frac{\text{f(H)}}{\sum_j f_j}$ où f(H) désigne la moyenne des f_i au temps t, pour les individus représentant le schème H. Si on note $\overline{f} = \frac{\sum_j f_j}{n}$ on obtient finalement :

$$\text{m(H,t+1)} = \text{m(H,t)} \frac{\text{f(H)}}{\overline{f}}$$

13.7.1 Effet de reproduction/sélection sur le nombre de schèmes

Les schèmes au-dessus de la moyenne se reproduisent, ceux qui sont en dessous de la moyenne disparaissent. Supposons que f(H) soit de la forme $\overline{f}(1 + c)$, où c est une constante indépendante du temps. On obtient ainsi :

$$\text{m(H,t+1)} = \text{m(H,t)}(1 + c)$$

en commençant à $t = 0$, on obtient :

$$\text{m(H,t)} = \text{m(H,0)}(1 + c)^t$$

Ce qui montre une croissance (resp. décroissance) exponentielle au cours du temps pour les schèmes au-dessus (resp. en dessous) de la moyenne.

13.7.2 Effet du cross-over

Considérons la chaîne A = 0111000, c'est un représentant, entre autres, des schèmes H_1 = *1****0 et H_2 = ***10**. Supposons que l'on croise A avec la chaîne A′ = 0010011 au point 3 (H_1 et H_2 ne sont pas des schèmes représentés par A′). Le croisement va détruire H_1 et préserver le schème H_2. Si on calcule les longueurs respectives des deux schèmes, on obtient : $\delta(H_1)$ = 7 - 2 = 5 ; $\delta(H_2)$ = 5 - 4 = 1.

Notons, p$-d$ la probabilité de destruction d'un schème, elle vaut δ(H) / ($l-1$) dans le cas d'un cross-over défini en un point particulier. La probabilité de survie p_s = 1 - p_d. Si maintenant on considère que le cross-over est effectué avec une probabilité p_c, la probabilité de survie devient :

$$p_s \geq 1 - p_c \frac{\delta(\text{H})}{l-1}$$

d'où l'on tire que :

$$\text{m(H,t+1)} \geq \text{m(H,t)} \frac{\text{f(H)}}{\overline{f}} [1 - p_c \frac{\delta(\text{H})}{l-1}]$$

13.7.3 Effet de mutation

Pour qu'un schème particulier H survive à une mutation, il faut que toutes ses positions instanciées survivent. Si on note p_m la probabilité d'effectuer une mutation ; la probabilité de survie pour une position particulière est 1 - p_m. Pour un schème, sa probabilité de survie est donc $(1-p_m)^{\text{o(H)}}$ comme p_m est très petit devant 1 on peut approximer la probabilité de survie par 1 − o(H) p_m.

D'où l'expression finale pour tenir compte de l'ensemble des paramètres

$$m(H,t+1) \geq m(H,t)\frac{f(H)}{\overline{f}}[1 - p_c\frac{\delta(H)}{l-1}][1 - o(H)\ p_m]$$

Ce qui en développant et en ignorant les termes du second ordre donne :

$$m(H,t+1) \geq m(H,t)\frac{f(H)}{\overline{f}}[1 - p_c\frac{\delta(H)}{l-1} - o(H)\ p_m]$$

13.8 Remarques et Conclusion

Très facile à mettre en œuvre, il existe plusieurs bibliothèques en particulier la `GALib` sur `http://lancet.mit.edu/ga/` et `EO` sur `http://eodev.sourceforge.net/`. La difficulté majeure réside dans le problème du codage des individus, même si on priviligiera un codage binaire, certains codes seront non interprétables - il faudra en tenir compte dans la fonction de fitness. Mais aussi dans le choix de la fonction de fitness - il faudra s'assurer que cette fonction est à valeur positive, on pourra être amener à transformer cette fonction pour faciliter la sélection au moyen de techniques de compression, décompression ou réalignement. Parmi les problèmes auxquels on doit faire face dans l'utilisation des AGs on peut noter :

1. Convergence prématurée : Bien que l'un des points forts des AG soit leur aptitude à s'échapper des minimas locaux, il peut arriver qu'ils convergent vers des solutions sous optimales. Cela arrive en particulier lorsque la population est trop petite, dans ce cas une solution peut apparaître rapidement et dominer alors que l'espace n'a pas été suffisamment explorer. Une méthode couramment utilisée est le ré-échelonnage de la fitness, une augmentation de la probabilité de mutation (en fait ce taux peut varier au cours du temps conjointement à une variation inverse du taux de croisement).

2. Temps d'exécution trop important : cet inconvénient peut être lié à une population trop importante ou à une fonction d'évaluation trop coûteuse. Une solution possible pour le premier problème est de diminuer la population tout en augmentant le nombre de génération - dans ce cas, il faudra intégrer le risque d'une convergence prématurée. Pour le second problème, on pourra chercher à simplifier la fonction d'évaluation - ce faisant on perdra en précision. L'autre alternative étant de combiner évaluation simplifiée et évaluation complexe au cours du temps.

Cinquième partie

Apprentissage

L'une des différences majeures entre l'homme et l'ordinateur est que l'être humain, lorsqu'il effectue une tâche quelconque s'efforce d'augmenter son efficacité. Alors qu'un ordinateur suit de manière constante l'algorithme implémenté sans possibilité d'amélioration ou d'adaptation. L'objectif de l'apprentissage automatique est d'offrir à l'ordinateur la possibilité d'adapter ses mécanismes de traitements.

Qu'est-ce qu'apprendre ?

Question difficile s'il en est à laquelle je suis bien incapable de répondre, ce que je vous propose ici est plutôt d'étudier différentes approches de l'apprentissage.

13.8.1 Vision externe

On peut essayer d'élaborer une taxonomie de l'apprentissage en fonction des méthodes classiquement utilisées :

- Apprentissage supervisé : dans ce cadre, on définit une méthode permettant au système de déterminer qu'il a commis une erreur. L'astuce, consiste à fournir au système la réponse attendue (*désirée*) pour une configuration d'entrée, le système calcule sa réponse puis l'écart éventuel entre sa réponse et celle prévue. Tout fonctionne "comme si" le programme disposait d'un tuteur lui donnant la bonne réponse. C'est l'approche que l'on rencontre en particulier pour la classe la plus répandue de réseaux de neurones : le `PMC`. Il s'agit d'un apprentissage à partir d'exemples, de résolution de problèmes de classification ou de discrimination.

- Apprentissage non supervisé : ici on laisse le système trouver des régularités dans les informations qui lui sont fournies, très utilisé dans les problèmes de formation de concepts, de regroupements.

- Apprentissage semi-supervisé : dans ce cadre, on ne donne pas la réponse attendue mais on dispose malgrè tout d'un oracle indiquant si la réponse fournie est correcte ou non. Cet oracle peut être cablé ou pas - dans ce cas le retour est obtenue en confrontant la réponse à l'environnement. Cette approche a été retenue dans le cadre de l'apprentissage par renforcement où l'agent est en constante interaction avec son environnement.

13.8.2 Vision interne

Le processus d'apprentissage (principalement symbolique) est subdivisé en deux approches fondamentales : *synthétique* et *analytique*. Dans l'approche synthétique il s'agit de créer de nouvelles connaissances (éventuellement meilleures) alors que dans l'approche analytique on s'efforcera de réorganiser les connaissances existantes (toujours dans un but d'efficacité). L'approche synthétique fait surtout appel à un raisonnement *inductif* c'est-à-dire par formulation d'hypothèses permettant de justifier les données dont on dispose ; l'approche analytique privilégiera quant à elle un raisonnement *déductif* c'est-à-dire que les données servent de prémisses pour tirer de nouvelles informations.

Chapitre 14

Réseaux de Neurones Artificiels

14.1 Introduction

L'objectif de cette partie est d'introduire les concepts de base utiles dans le cadre des réseaux de neurones artificiels, abrégés en RNA dans la suite du texte. Nous ne ferons qu'effleurer un domaine en pleine effervescence. La bibliographie sur le domaine est extrèmement vaste, pour le lecteur désireux d'en savoir plus, je le renvoie à d'autres documents que j'ai écrit par ailleurs [27], et qui sont accessibles sur le réseau à partir de l'URL `http://www.sm.u-bordeaux2.fr/~corsini/Cours/ANN/`

Les réseaux de neurones formels ne sont pas un modèle de notre structure cérébrale mais plutôt une métaphore. Chaque élément constitutif réalise un traitement simple, mais l'ensemble fait émerger des propriétés globales dignes d'étude. Grosso modo, on peut considérer qu'un neurone (formel) est à la fourmi, ce qu'un réseau est à une fourmilière. Chaque composant fonctionne/évolue indépendemment de ses voisins, la structure globale forme un système parallèle massivement connecté. L'information n'est pas localisée mais distribuée sur l'ensemble de la structure. Un réseau ne se programme pas mais est entraîné sur des données pour réaliser la tâche désirée. Parmi les utilisations majeures on trouve :

- l'association,
- la classification,
- la discrimination,
- l'estimation.

Cette démarche a été initiée par les travaux de Mc Culloch et Pitts (1943) et par ceux de Hebb (1945).

14.2 Brefs rappels historiques

Il s'agit d'une réapparition d'un concept développé à la fin des années 50-60 à la confluence de la recherche en Intelligence Artificielle et en Psychologie. L'idée sous-jacente est qu'un assemblage d'éléments simples (neurones formels) et ayant un comportement similaire est capable de résoudre des problèmes complexes. Ces éléments simples sont inspirés des neurones biologiques briques élémentaires de notre cerveau. Bien entendu, il s'agit d'une simplification extrème des neurones biologiques, dont l'idée revient à deux chercheurs McCulloch et Pitts (1943) [73].

14.3 Structure

Un réseau de neurones artificiels (RNA) peut-être assimilé à un graphe pondéré, dont les sommets sont les neurones formels (correspondant aux neurones biologiques) et les connexions pondérées

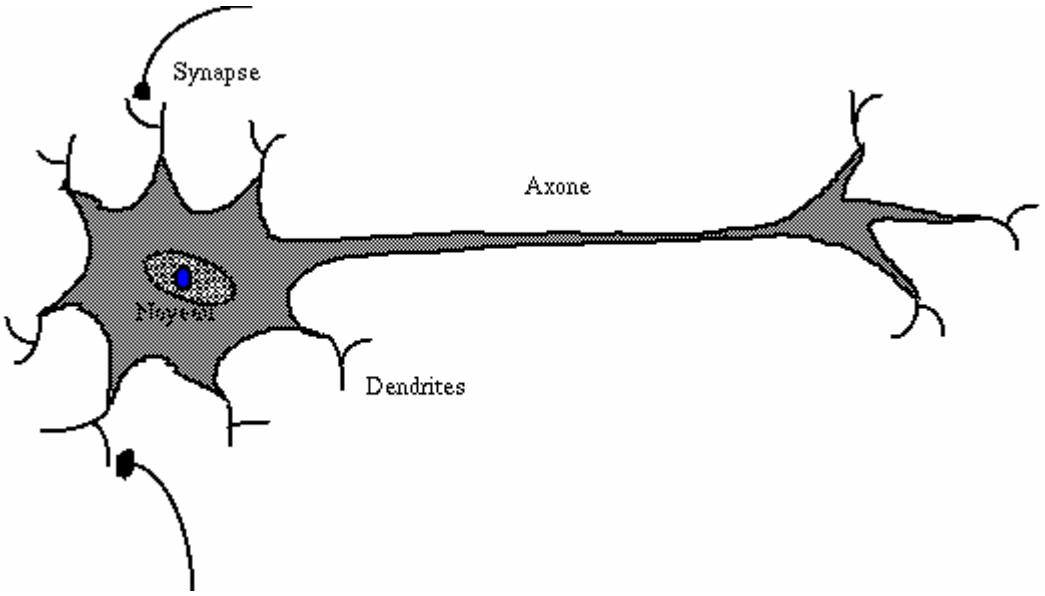


FIGURE 14.1 – neurone biologique

(représentent vaguement les connexions synaptiques). Nous allons dans cette partie décrire les deux éléments constitutifs d'un RNA et fixer le vocabulaire et les notations en vigueur dans le domaine.

14.3.1 Sommet

J'utiliserai indifféremment le terme de sommet, cellule ou neurone, étant bien entendu que je ne fais aucun amalgame entre la réalité biologique et la modélisation informatique.

Il s'agit donc d'un processeur élémentaire, muni d'une fonction de *transfert* continue ou non permetant de calculer la sortie en fonction des entrées fournies. On distingue parmi ces cellules trois catégories particulières :

1. Les cellules de la *rétine* qui servent d'entrée au système. Ce sont des sommets dont le demi-degré entrant est **0**.
2. Les cellules de *sortie* qui permettent de fournir le résultat d'un calcul au monde extérieur. Ce sont des sommets dont le demi-degré sortant est **0**.
3. Les cellules *cachées* qui, comme leur nom l'indique, sont inaccessibles à l'utilisateur.

14.3.2 Connexion

Entre deux cellules, il peut y avoir une connexion pondérée, qui peut suivant le cas être orientée ou pas. Dans le cas orienté, s'il y a une connexion de la cellule i vers la cellule j, on notera w_{ji} la pondération associée. On notera e_i l'entrée fournit par une cellule i à une cellule j. On notera s_j la sortie de la cellule j. On appelle *activation* la quantité a_j associée à la cellule j est définie par :

$$a_j = \sum_i w_{ji} e_i$$

la sortie pourra alors être calculée par :

$$s_j = f_j(a_j)$$

où f_j désigne la fonction de transfert associée au sommet j

Dans la littérature on trouve souvent :

$$s_j = f_j(\sum_i w_{ji} e_i - \theta_j)$$

où θ_j désigne le seuil associé au sommet j. Par souci de simplification, et sans rien enlever à la généralité du modèle, on pose l'existence d'une cellule 0, telle que :

$$s_0 = 1 \text{ et } w_{j0} = -\theta_j, \forall j$$

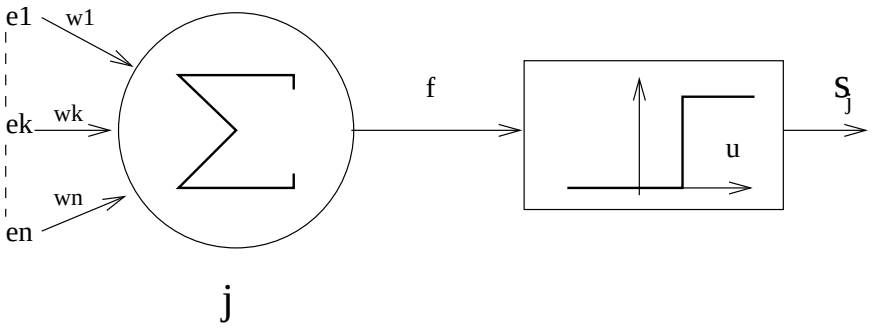


FIGURE 14.2 – Un sommet parmi d'autres

14.3.3 Fonctions de transfert

Suivant les modèles étudiés, on rencontre différentes sortes de fonction de transfert, nous donnons ci-après celles qui sont les plus courantes :

- La fonction identité : $s_j = a_j = \sum_i w_{ji} e_i$
- La fonction seuil : $\begin{cases} s_j = 0 & \text{si } a_j \leq 0 \\ s_j = 1 & \text{sinon} \end{cases}$
- Une fonction sigmoïde, c'est-à-dire, une fonction continue croissante, non constante et bornée qui approxime la fonction seuil :
$$s_j = \frac{e^{ka_j} - 1}{e^{ka_j} + 1}$$
- Une fonction probabiliste (dépendant d'une « température »T) :
$$\text{proba}(s_j = 1) = \frac{1}{(1 + e^{-a_j/T})}$$

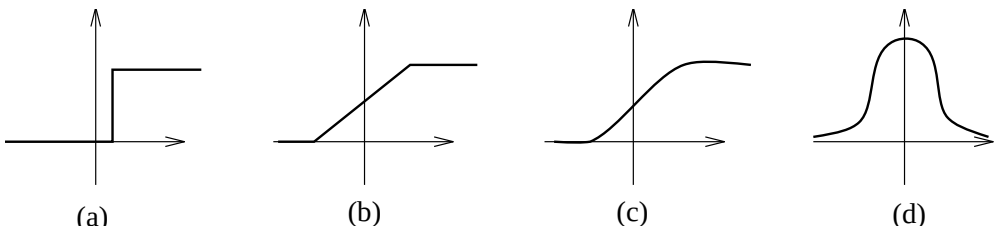


FIGURE 14.3 – fonctions de transfert

Chapitre 15

Traitement de données Symboliques

15.1 Généralités

L'accumulation d'informations par le biais des bases de données a remis au goût du jour l'intérêt pour les méthodes d'apprentissage automatique, la découverte de connaissances, l'extraction de règles. Nombre de ces méthodes sont regroupées sous l'appellation de « Data Mining »(Fouilles de Données - FD) ou Extraction des Connaissances à partir des Données (ECD) selon le terme consacré par Y. Kodratoff. Dans ce champ disciplinaire, on peut, par exemple, vouloir étudier le comportement de la clientèle à partir de données recueillies lors du passage en caisse d'une grande surface. Ces méthodes ont en commun une phase de collection/regroupement de données afin de mettre à jour certaines régularités, puis une phase consistant à extraire de nouvelles règles permettant de classer de nouvelles données.
L'objectif de ce chapitre est de mettre en évidence certains concepts clefs du DataMining sans en aborder toutes les finesses. Le lecteur intéressé pourra consulter avec bonheur les ouvrages suivants [59, 105, 100]

15.2 Data Mining : un instantanné

La montée en puissance du concept de Data Warehouse (entrepôt de données) permet de mettre à la disposition des entreprises de grands volumes de données intégrées et historisées. Bien que le stockage d'information soit primordiale (processus rendu possible par la baisse des coûts matériels) ceci reste sans valeur si l'utilisateur n'a pas à sa disposition des outils pour en tirer parti et transformer ces données en connaissance. Les outils de l'ECD s'attaquent à ce problème. Cette préoccupation est en pleine croissance. Les estimations faites par Gardner Group montre que moins de 15% des données stockées sont analysées, et que moins de 5% des données manipulées sont analysées [59]. Les besoins en analyse atteignent une croissance de 15 à 45% alors que, dans le même temps, l'accroissement des données est de l'ordre de 300%. Pour comprendre la dimension du problème, il faut savoir qu'un téra-octets de données (1000 Go ou 1 méga de Mo) est l'équivalent d'une bibliothèque de 2 millions de livres.

Nous reprenons la définition du Data Mining de [100]

Définition 15.1 (Data Mining)
Découvrir par induction des modèles intéressants et compréhensibles dur la base d'un ensemble de données.

15.3 Le problème

Soit un système physique à modéliser sur un ordinateur. À l'aide de ce modèle il est possible de tester si nous avons bien compris la façon dont les paramètres de contrôle influencent le comportement

du système. On dira que l'on dispose d'une connaissance profonde de la structure du système s'il est possible de faire une prédiction efficace au sein du système physique. Il existe de nombreuses méthodes pour construire un modèle, et celui qui est choisi dépend grandement de ce que l'on sait sur le système à modéliser.

15.4 Modélisation

L'élaboration du modèle dépend fortement de la quantité d'information (intuitive ou mathématique) qui est disponible sur le système. Si l'on dispose d'une connaissance complète des propriétés du système, il suffit de retranscrire cette connaissance directement dans le substrat disponible (par exemple une fonction mathématique ou bien un programme). La qualité du modèle produit est uniquement limitée par notre habileté à le retranscrire dans un nouveau format. Il est donc normal de supposer que les modèles associés à cette classe de problèmes sont les plus performants.
Lorsque l'on ne dispose que d'une connaissance partielle du système, l'information accessible n'est pas suffisante pour permettre de faire de bonnes prédictions. Néanmoins, des tendances ou des connaissances qualitatives du système peuvent être accessibles. Cette connaissance est augmentée par des procédures d'analyse qui extraient les informations manquantes à partir d'instance connue. Par exemple les méthodes de régression offrent de tels mécanismes. Une modélisation mathématique avec des inconnues est exprimée à partir d'information qualitative sous la forme d'équations mathématiques avec inconnues. La connaissance qualitative du comportement du système, qu'il soit quadratique, exponentiel ou linéaire s'exprime par le biais de coefficients obtenus par ces analyses.
Si l'on n'a très peu d'information sur le système, dans ce cas on ne peut même pas dérivé un modèle partiel (en pratique cela signifie que les méthodes statistiques employées n'ont pas permis de dériver un modèle adéquat) dans les cas extrèmes, il arrive que l'on ne sache même pas quels sont les paramètres clefs. L'information doit alors être extraite de l'ensemble des exemples par le biais de méthodes d'apprentissage inductives.

15.5 Le paradoxe d'Hempel

Comme nous l'avons déjà vu par ailleurs, le raisonnement à partir de cas peut être de nature *déductive* ou de nature *inductive*. Dans le premier cas, on part de prémisses pour aboutir à des conclusions, alors que dans le second on part d'exemples pour aboutir à une généralisation. Ainsi la règle :

Tous les corbeaux sont noirs

est confirmée par toute observation de corbeaux particuliers qui sont noirs. Cette règle est une implication et est donc équivalente à sa forme contraposée :

Tout ce qui n'est pas noir n'est pas corbeau

Ainsi n'importe quelle mouette blanche satisfait cette règle.

Hempel conclut : « puisqu'il y a tant d'observations qui peuvent confirmer une hypothèse folle quelconque, une confirmation qui s'appuye sur le nombre d'occurences est simplement impossible. Par conséquent l'induction est absurde. »

15.6 Un exemple de référence

Supposons que l'on souhaite trouver une classification pour les cellules suivantes

15.7 Arbres de Décision / Arbres de segmentation

Les arbres de décision sont un formalisme simple et dont la restitution est facile à lire. On constate qu'une demie-journée à une journée de formation suffit pour pouvoir confier un logiciel à base

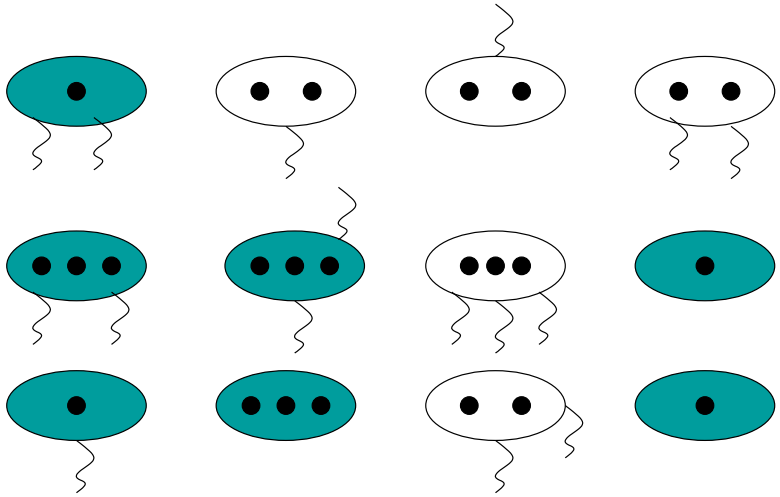


FIGURE 15.1 – Un exemple jouet

d'arbres de décisions à un ingénieur métier. Le marché a vite compris la complémentarité de ces outils avec des produits traditionnels de requêtes et de tableurs.
Deux domaines majeurs d'application pour les arbres sont :

1. La construction d'un algorithme de segmentation d'une population dont les groupes d'affectation sont connus.
2. L'affectation d'une classe à un individu à partir de certains éléments descriptifs.

15.7.1 Un petit exemple : le jeu des 20 questions

Un joueur pense à un objet (par exemple un véhicule), les autres joueurs cherchent à découvrir son identité en posant une suite de questions auxquelles on répond par oui ou par non. La réponse à une question détermine la suivante. Si les questions sont judicieusement choisie on peut alors rapidement aboutir à la solution. On remarque que de nombreuses feuilles peuvent englober les mêmes objets mais pour des raisons différentes. La croissance de l'arbre est très variée (feuilles à des profondeurs différentes). On évalue l'efficacité de l'arbre dans son ensemble en mesurant le pourcentage de réponses correctes. À chaque nœud de l'arbre, on peut mesurer :

- Le nombre d'enregistrements entrant dans le nœud.
- Comment les éléments seraient classées si le nœud était une feuille.
- Le pourcentage de réponses correctes à ce niveau.

En fait les algorithmes de construction cherchent la meilleure distribution des données entre les catégories souhaitées. Les informations ci-dessous sont tirées de [105].

15.8 Un peu de vocabulaire

Soit Ω une population d'individus ou d'objets. À cette population est associée un attribut particulier que l'on souhaite prédire - on parle, suivant les auteurs, de *variable endogène*, d'attributs à expliquer ou *à prédire* et parfois de *classe*. À chaque individu $\omega \in Omega$ on peut associer sa classe $C(\omega)$ définie par :

$$\begin{array}{rcccl} C & : & \Omega & \rightarrow & C = \{c_1, \ldots c_k\} \\ & & \omega & \mapsto & C(\omega) \end{array}$$

La détermination du modèle à prédire φ est liée à l'hypothèse selon laquelle les valeurs prises par la variable endogène ne relève pas du hasard mais de certaines caractéristiques des individus. Pour cela, l'expert du domaine établit une liste à priori d'attributs *prédictifs* ou *explicatifs* ou *variables exogènes* notées $\mathbf{X} = (X_1, \dots X_n)$. Ces variables exogènes prennent valeur dans un espace $\mathbb{E}$ sans structure mathématique sous-jacente particulière :

$$\begin{array}{rcccl} \mathbf{X} & : & \Omega & \rightarrow & \mathbb{E} \\ & & \omega & \mapsto & \mathbf{X}(\omega) = (X_1(\omega), \dots, X_n(\omega)) \end{array}$$

Dans l'exemple 15.1, $\Omega = \{o_1, \dots, o_{12}\}$, la variable à prédire est la couleur $C = \{\text{sombre}, \text{clair}\}$. Il s'agit donc de trouver un modèle de prédiction φ tel que pour un individu $\omega \in \Omega$ on puisse calculer $C(\omega)$ à partir de $\mathbf{X}(\omega)$. On souhaite que φ soit *cohérent*, i.e. si pour deux individus $\omega, \omega' \in \Omega$ on a $\varphi(\omega) = \varphi(\omega')$ alors nécessairement $C(\omega) = C(\omega')$.

En général, cette propriété ne pourra pas être vérifiée qu'à postériori. Dans la pratique, on construit φ sur un échantillon d'apprentissage $\Omega_a \subseteq \Omega$ et on se contente de vérifier sur une majorité d'individus issus d'un ensemble $\Omega_t \subseteq \Omega$ et $\Omega_t \neq \Omega_a$ que :

$$\forall \omega \in \Omega_a \cup \Omega_t \text{ on suppose connues } \mathbf{X}(\omega) \text{ et } \mathrm{C}(\omega)$$

Comme, il arrive souvent que le système ne soit pas capable de déterminer la classe d'un individu, on prend l'habitude de définir une nouvelle modalité $c_0 \notin C$, et on définit $C' = \{c_0\} \cup C$; φ est défini de $\mathcal{X}$ dans C'.

Définition 15.2
Le modèle de prédiction φ est dit cohérent si pour une majorité de cas, on a une bonne prédiction i.e.

$$\frac{|\ \{\omega \in \Omega_t\ /\varphi(\omega) \neq \mathrm{C}(\omega)\}\ |}{|\ \Omega_t\ |} \leq \epsilon$$

Où ϵ est fixé par l'utilisateur.

Si φ est jugé cohérent, on pourra calculer $\varphi(\omega), \forall \omega \in \Omega \setminus (\Omega_a \cup \Omega_t)$. Tout se résume en fait à trouver une partition de Ω par rapport à C.

Exemple 15.8.1
Reprenons l'exemple de la figure 15.1 page 159, l'ensemble $\mathcal{X}$ peut être défini par X_1 = noyaux = $\{1, 2, 3\}$ et X_2 = queues = $\{0, 1, 2, 3\}$; on supposera ici que $\Omega_a = \Omega_t = \Omega$ et on pourra représenter les éléments à l'aide d'un tableau, où les lignes seront les individus et les colonnes les variables.

	Attributs		
Individus	Noyaux	Queues	Couleur
O_1	1	2	sombre
O_2	2	1	clair
O_3	2	1	clair
O_4	2	2	clair
O_5	3	2	sombre
O_6	3	2	sombre
O_7	3	3	clair
O_8	1	0	sombre
O_9	1	1	sombre
O_{10}	3	0	sombre
O_{11}	2	2	clair
O_{12}	1	0	sombre

◁

Définition 15.3
On appelle partition engendrée par $C(\Omega) = \{c_1, \dots, c_p\}$ sur Ω la partition $2^{C(\Omega)} = \{\Omega_1, \dots, \Omega_p\}$ où $\Omega_i = \{\omega \in \Omega, C(\omega) = c_i\} = C^{-1}(c_i)$.

L'objectif de l'apprentissage est alors de construire un modèle φ permettant de partitionner Ω à partir des cariables exogènes. On comparera donc $2^{C(\Omega)}$ avec $2^{\varphi(\Omega)}$ et si elles sont identiques on pourra conclure que φ est un bon prédicteur pour C.

15.9 Profils, Mintermes, Partitions engéndrées

Dans le cas où les X_i ($i = 1, \ldots, p$) sont à valeur dans un ensemble fini discret - on parle alors de variables qualitatives, on définit les notions de *profil élémentaire*, *profil*, *minterme élémentaire* (ou profil observé) et *minterme généralisé*.

- Un profil élémentaire : tout élément de $\mathbb{E}$.
- Un profil : toute partie non vide de $\mathbb{E}$. Si $\mathbb{E}$ est un produit cartésien on utilisera la notation $\star$ pour indiquer tout élément de l'ensemble initial. Ainsi la notation $(1, \star)$ dans l'exemple de la figure 15.1, indique que l'on considère le profil correspondant à un seul noyau et un nombre quelconque de queues.
- Un minterme élémentaire : soit π un profil élémentaire, on définit $\mathbf{X}^{-1}(\pi) = \{\omega \in \Omega, \mathbf{X}(\omega) = \pi\}$
- Un minterme généralisé : soit $\Pi \subset \mathbb{E}$ un profil quelconque, on appelle minterme généralisé ou simplement minterme l'ensemble des individus de Ω appartenant au profil Π.

Nous allons voir dans l'exemple qui suit comment s'applique les différentes notions sur les cellules de la figure 15.1.

Exemple 15.9.1
$\Omega = \{o_1, \ldots, o_{12}\}$, $\mathbb{E} = \{1, 2, 3\} \times \{0, 1, 2, 3\}$.

- *Un profil élémentaire :* $(1, 1)$.
- *Un profil :* $\{1, 2\} \times \{1, 3\} = \{(1, 1), (1, 3), (2, 1), (2, 3)\}$
- *Un minterme élémentaire :* $\pi = (1, 1)$, $\{o_9\}$
- *Un minterme généralisé :* $\Pi = \{(1, 1), (1, 3), (2, 1), (2, 3)\}$, $\{o_9, o_2, o_3\}$

◁

Définition 15.4
On dira que $S = \{s_1, \ldots, s_k\}$ est une partition engendrée par $\mathbf{X}$ sur Ω si s_j est une réunion de mintermes élémentaires pour tout $j = 1, \ldots, k$.

Exemple 15.9.2
Toujours dans le cadre de notre exemple fétiche, si on pose $\Pi_1 = (2, \star), \Pi_2 = (1, \{1, 2\}), \Pi_3 = (1, \{0, 3\}) \cup (3, \star)$ *et* $s_i = \mathbf{X}^{-1}(\Pi_i)$. *On aura réaliser une partition engendrée par* $\mathbf{X}$ sur Ω ◁

Définition 15.5
On dira qu'une partition $S_1 = \{s_1^1, \ldots, s_k^1\}$ est plus fine qu'une partition $S_2 = \{s_1^2, \ldots, s_p^2\}$ si pour tout $s \in S_1, \exists s' \in S_2, s \subset s'$.

Définition 15.6
On dira qu'une partition $S = \{s_1, \ldots, s_k\}$ engendrée par des variables X_j est compatible avec la partition $2^{C(\Omega)}$ engendrée par C, si S est plus fine que $2^{C(\Omega)}$.

15.10 ID3 : un exemple

ID3 « Induction Decision Tree »ou arbre d'induction est la désignation d'un algorithme ancien de construction d'arbres, dans lequel le critère de sélection des variables est le *gain d'information*. Cet algorithme a été développée dans le milieu des années 70 et présente la stratégie la plus élémentaire. C4.5 et C5.0 sont des descendants développés par Quinlan dans les années 90/95. Une version de C5.0 est accessible sur excalibur dans le répertoire `C50` de ma « homedir ».

15.10.1 Principe général

Comme toutes les autres méthodes utilisant des arbres d'induction, ID3 élabore une suite de partitions de plus en plus fines à partir de l'ensemble Ω_a. La racine de l'arbre est l'ensemble initial. On cherche parmi p variables, celle offrant la « meilleure partition »nouvelle au sens d'un critère (à définir). Sur chaque élément de cette nouvelle partition, on itère la décomposition indépendamment de ce qu'il advient aux autres nœuds. On déclare qu'un sommet est *saturé* s'il n'existe aucune variable permettant d'engendrer localement une sous-partition. La procédure d'expansion s'arrête lorsque tous les sommets sont saturés. On adjoint souvent une autre règle à celle du critère, dont l'objectif est d'éviter de partitionner inutilement un sommet si les effectifs associés sont trop faibles pour pouvoir être considérés comme statistiquement significatifs.

15.10.2 Critère

Pour ID3, le passage d'une partition S_i à une partition S_{i+1} se fait à partir de la sélection d'une variable. Le pouvoir discriminant d'une variable est généralement exprimé par la notion de *variation d'entropie* lors du passage de S_i à S_{i+1}.

Entropie de Shannon

L'entropie de Shannon, notée h_S, est définie à partir d'un vecteur de probabilité $p_1, \ldots, p_k$ par :

$$h_S(p_1, \ldots, p_k) = -\sum_{i=1}^{k} p_i log_2 p_i$$

Comme on travaille sur des échantillons de taille finie, ces probabilités sont estimées par les fréquences empiriques. La formule devient :

$$h_S(f_1, \ldots, f_k) = -\sum_{i=1}^{k} f_i log_2 f_i$$

Par exemple, supposons qu'un nœud, s_k, corresponde à 10 individus de la classe c_1, et 20 individus de la classe c_2. L'effectif global associé au nœud est 30. Notons $f_1 = \frac{10}{30}$ et $f_2 = \frac{20}{30}$, l'entropie associée est alors

$$h_S(s_k) = -\frac{10}{30} log_2 \frac{10}{30} - \frac{20}{30} log_2 \frac{20}{30} = .92$$

15.11 Évaluations empiriques de classifieurs

Cette partie s'inspire de [105, chap. 11] et dans une moindre mesure de [90]. Le lecteur désireux d'en savoir plus se reportera donc avec bonheur sur les références ci-dessus.

15.11.1 Objet de la section

Quelque soit la méthode d'apprentissage automatique utilisée, on cherche toujours l'élaboration du meilleur prédicteur. En général cette notion du « meilleur »renvoie à la notion de taux d'erreur en généralisation. Nous allons donc étudier ce concept d'évaluation afin de proposer les stratégies les plus appropriées.

Lorsque des travaux proposent de nouveaux algorithmes d'apprentissage, après la partie usuelle consistant à positionner le problème, ils présentent un ensemble d'exemples fictifs (le plus souvent) sur lesquels les méthodes précédentes sont peu ou pas opérantes. Ils montrent ensuite sur ces mêmes exemples le fonctionnement du nouvel algorithme, qui sans surprise, possède d'excellentes performances. Pour finir, ils procèdent à une évaluation sur des bases de données réelles. Plusieurs remarques peuvent être tirées :

1. La plupart de ces nouveaux algorithmes sont le plus souvent des adaptations d'algorithmes plus anciens.
2. L'évaluation sur les données réelles n'est pas toujours faite.
3. La démarche utilisée pour mener cette (éventuelle) comparaison est de nature empirique. Elle a pour objectifs de :
   - Répondre à la question « l'algorithme A est-il, en général meilleur que l'algorithme B ? ». Cette question est loin d'être triviale, plusieurs auteurs ont montré que les carances constatées dans un domaine sont en général contrebalancées dans un autre domaine.
   - Vérifier la viabilité des réponses proposées. Il s'agit en général d'étudier le taux en re-substitution (ou taux global).
   - Analyser la fiabilité de l'algorithme. Ce qui inclut une estimation du taux d'erreur commis en généralisation mais aussi l'analyse du type d'erreur commise.
   - Sélectionner des modèles. Il s'agit en général de travailler sur le taux d'erreur mais aussi sur la complexité, le temps de réponse, la difficulté d'actualiser le modèle.

15.11.2 Critères à étudier

Selon les qualités d'un classifieur, on utilise deux critères principaux d'évaluation :

1. Le taux d'erreur en généralisation qui indique la capacité de l'algorithme à reproduire le concept à apprendre.
2. La complexité, qui dans le cadre des graphes d'induction peut se traduire soit par le nombre de feuilles, soit par le nombre de sommets. Le nombre de feuilles correspond au nombre de règles, le nombre de nœuds au nombre de propositions. Il s'ensuit que ces indicateurs recouvrent à la fois les notions de *lisibilité* et de *fiabilité*.

15.11.3 Les données tests

Ces données peuvent être regroupées en différentes catégories :

1. Les données *synthétiques* qui sont artificielles et générées à partir de fonctions. Elles sont parfaitement maîtrisées, le concept à apprendre est connu. Elles servent en général pour la mise au point de l'algorithme. Si elles sont en petites quantités, elles peuvent servir à expliquer le fonctionnement général du système. Elles peuvent aussi servir à caractériser les situations dans lesquelles une méthode est plus adaptée qu'une autre.
2. Les données *réalistes* sont artificielles mais engendrées par un modèle proche de problèmes réels. Puisqu'artificielles elles sont maîtrisables.
3. Les données *réelles* proviennent d'observations et constituent le vrai challenge pour des méthodes qui devront traiter des problèmes réels. Elles sont incontrôlées, on ne connaît ni leur niveau de bruit, ni les relations des observations manquantes avec le concept à apprendre ni le concept à apprendre (quand il existe). Pour être exploitables, il faut procéder à une première analyse :
   - l'effectif total ;
   - le nombre de classes de la variable à prédire ;
   - les distributions ;
   - le nombre d'attributs ;
   - leurs natures : qualitative, continue, mixte ;
   - la distribution des attributs ;
   - la présence ou l'absence de valeurs manquantes.

15.12 Analyse et estimation de l'erreur

L'un des critères le plus souvent utilisé est celui dit de généralisation, mais il semble nécessaire de s'intéresser aussi à la structure de l'erreur.

15.12.1 Matrice de Confusion

Il s'agit d'un tableau de contingence confrontant les classes obtenues (colonnes) et les classes désirées (lignes) pour l'échantillon. Sur la diagonale, on trouve donc les valeurs bien classées, hors de la diagonale les éléments mal classés ; la somme des valeurs sur une ligne donne le nombre d'exemplaires théoriques de la catégorie. Si les classes sont indépendantes, la position de l'erreur n'a aucune signification, si par contre les classes ne sont pas indépendantes, on peut définir une sorte de gradation dans les erreurs.

DÉFINITION 15.7

- *Taux d'erreur global en resubstitution*
  C'est le ratio entre le nombre d'objets bien classés par rapport au nombre d'objets de l'échantillon.
- *Taux d'erreur à priori*
  C'est le taux d'erreur pour chaque catégorie.
- Taux d'erreur à postériori
  C'est le taux de mauvaises classifications par rapport aux bonnes classifications pour une classe donnée.

EXEMPLE 15.12.1

Prenons un problème de classification, 3 classes A, B et C indépendantes la matrice de confusion suivante :

	A	*B*	*C*
A	*50*	*0*	*0*
B	*4*	*45*	*1*
C	*6*	*4*	*40*

On voit que l'on disposait d'un échantillon de 150 valeurs réparties équitablement entre les 3 classes, ce qui n'est pas une nécessité. La classe A est correctement discriminée, les deux autres classes posent quelques difficultés. Comme les classes sont indépendantes, cela signifie que toutes les erreurs d'attribution ont même importance (il n'est pas moins grave pour notre classifieur de croire que l'on a un objet de la catégorie A ou B, alors qu'il fallait détecter la catégorie C).
Regardons maintenant les trois indicateurs :

- *Taux global :* $(50 + 45 + 40)/150 = .90$. *On a donc un taux de reconnaissance de 90%. Cela signifie que l'on a un taux d'erreur en resubstitution de 10%*
- *Taux à priori : Si on considère chaque classe individuellement, le système ne se trompe pas sur la reconnaissance de la classe A, il fait 1 erreur sur 10 pour la classe B, et 2 sur 10 pour la classe C.*
- *Taux à postériori : Si on considère le système du point de vue des réponses fournies, lorsqu'il déclare qu'un objet appartient à la classe A, il se trompe 10 fois sur 60, il y a* $\frac{4}{60} = 6.67\%$ *de chances que ce soit un objet de la catégorie B et* $\frac{6}{60} = 10\%$ *de chances que ce soit un objet de la classe C, en d'autres termes son taux de fiabilité (pour la catégorie A est de 83.33%). Par le même type de calcul on aboutit pour B à* $\frac{4}{49} = 8.16\%$ *de chances que ce soit un objet de la classe C, soit un taux de fiabilité de 91.84%. Pour C, on trouve : 2.5% d'appartenir en réalité à la classe B*

◃

15.12.2 Estimation de l'erreur

La méthode consiste à poser :

$$q(\phi,\omega) \quad = \quad \begin{cases} 0 & \text{si} \quad \phi(\omega) \quad = \quad C(\omega) \\ 1 & \text{si} \quad \phi(\omega) \quad \neq \quad C(\omega) \end{cases}$$

Où ϕ est un prédicteur, C la variable à prédire, ω un élément de Ω.

L'erreur théorique s'écrit :

$$\epsilon = \frac{\sum_{\omega} q(\phi,\omega)}{\mid \Omega \mid}$$

Le taux d'erreur théorique est la probabilité de se tromper si on applique le classifieur à la population totale Ω malheureusement cela est rarement possible (et de toutes les façons cela serait extrèmement coûteux). C'est pourquoi on utilise le taux d'erreur en resubstitution mais ce dernier induit un biais optimiste très important. Le second taux que l'on utilise est accessible si l'on choisit un schéma apprentissage-validation. Dans ce cas on divise la population en deux sous-ensembles Ω_a et Ω_t, le premier servant à construire le classifieur, le second servant à l'évaluation.

On utilise dans ce cas la formule suivant :

$$\epsilon_{\text{validation}} = \frac{\sum_{\omega \in \Omega_t} q(\phi,\omega)}{\mid \Omega_t \mid}$$

Cette méthode possède cependant plusieurs inconvénients :

1. La variance $\frac{\epsilon_{\text{validation}}(1-\epsilon_{\text{validation}})}{|\Omega_t|}$ est trop forte
2. On ne mesure pas la variabilité induite par les échantillons de l'apprentissage Ω_a

Pour contrebalancer cela, on a souvent recours à plusieurs itérations de l'apprentissage-validation. Le principe est de répéter n fois le processus en tirant aléatoirement les échantillons, et en calculant

$$\epsilon_{\text{validation iteree}} = \frac{\sum_{i=1}^{n} \frac{\sum_{\omega \in \Omega_{t_i}} q(\phi_i,\omega)}{\mid \Omega_{t_i} \mid}}{n}$$

Malheureusement on obtient ici une variance largement sous-évaluée car elle est calculée sur une série d'échantillons non indépendants puisqu'ils partagent un certain nombre d'individus. Pour résoudre cette difficulté on préfère utiliser la validation croisée.

Validation croisée elle propose une solution au problème du recouvrement entre les fichiers de validation et la trop faible taille de l'échantillon si on n'en utilise qu'un seul exemplaire. Elle s'appuye sur le fait que plusieurs estimations à partir de portions indépendantes d'un echantillon sont plus faibles qu'une estimation sur la totalité.

validation croisée

Subdiviser l'échantillon en s parties égales
Pour chaque portion u faire
  Former $\Omega_{a,-u}$ provenant des $s - 1$ portions restantes
  Créer φ_u à partir de $\Omega_{a,-u}$
  Calculer ϵ_u à partir de $\Omega_{a,-u}$
Fait

$$\epsilon_{vc} \quad = \quad \sum_{u} \frac{\mid \Omega_{a,-u} \mid}{\mid \Omega_a \mid} \epsilon_u$$

$$\text{Var}(\epsilon_{vc}) \simeq \frac{\epsilon_{vc}\ (1\ -\ \epsilon_{vc})}{s}$$

En procédant à une étude empirique, il a été établi que la meilleure stratégie était une validation croisée avec $s = 10$. Il s'agit d'un compromis entre un biais qui diminue et une variance qui s'accroit avec l'augmentation du nombre de portions.

15.13 Comparaison des classifieurs

Il s'agit dans cette partie de répondre au mieux à la question « le classifieur issu de l'algorithme A est-il meilleur que celui issu de l'algorithme B pour cet ensemble de données ? ». Pour des classifieurs différents il n'est pas possible de contrôler les éléments influençant les performances de l'algorithme, les résultats ne sont qu'indicatifs et n'offrent pas la possibilité d'analyser les comportements. Les principales sources d'aléas à contrôler sont :

- Le fichier test, puisqu'aléatoire, il a toujours des chances d'être atypique.
- La remarque vaut aussi pour le fichier d'apprentissage d'autant plus crucial que l'algorithme est sensible à ces variations.
- La variabilité du classifieur lui-même.
- La quantité de bruit contenu dans les données influe fortement sur le classifieur final.

Les tests doivent donc tenir compte de ces sources de variations, c'est pourquoi il est nécessaire de procéder à plusieurs couples apprentissage-validation. Néanmoins, dans le cas où l'on ne peut faire qu'un seul apprentissage, il est possible d'utiliser l'un des outils suivants.

15.13.1 Apprentissage unique

Lorsque l'on ne dispose que d'un couple apprentissage-validation et de deux classifieurs φ_A et φ_B. On ne peut comparer que le taux d'erreur. Pour chaque individu ω on peut penser que la variable $q(\varphi, \omega)$ suit une loi de Bernouilli avec les paramètres Binomial$(1, \epsilon)$. Puisque l'on observe un grand nombre d'individus (supérieur à 30), on peut appliquer le théorème central limite et l'on admet que la variable ϵ_{valid} tend vers la loi normale Normal$(\epsilon, \frac{\epsilon(1-\epsilon)}{|\Omega_l|})$.

Le test de McNemar [35] s'intéresse à la structure de l'erreur. Pour cela on forme le tableau à deux lignes et deux colonnes :

$n_{\neg A,\neg B}$	$n_{\neg A,B}$
$n_{A,\neg B}$	$n_{A,B}$

$n_{\neg A,\neg B}$ désigne « le nombre de fois où φ_A et φ_B se sont trompés simultanément ».

$n_{A,\neg B}$ désigne « le nombre de fois où φ_A ne s'est pas trompé et φ_B si ».

$n_{\neg A,B}$ désigne « le nombre de fois où φ_A s'est trompé et φ_B non ».

$n_{A,B}$ désigne « le nombre de fois où ni φ_A ni φ_B se sont trompés simultanément ».

Sous l'hypothèse de performances égales entre les deux systèmes, les quantités n_{10} et n_{01} ont la même valeur espérée, qui est $\frac{n_{01}+n_{10}}{2}$. Il reste alors à former la statistique du χ^2 à 1 degré de liberté :

$$\chi^2_{\text{McNemar}} = \frac{(\mid n_{01} - n_{10} \mid -1)^2}{n_{01} + n_{10}}$$

On peut alors lire la probabilité que l'hypothèse nulle soit vraie. La table suivante reporte les informations tirées de [64].

	.250	.100	.050	.025	.010	.005	.001
$\alpha = 1$	1.3233	2.7055	3.8414	5.0230	6.6349	7.8794	10.824

Si la valeur du test de χ^2 est strictement supérieur à 3.8414 on peut rejeter l'hypothèse nulle, puisque la probabilité associée est inférieure à 5%.

15.13.2 Apprentissage répété

Lorsque l'on décide d'effectuer plusieurs répétitions, la validation croisée peut offrir un cadre simple et viable pour les tests. Les tests sur l'erreur reposent sur un test de Student sur échantillons appariés en faisant l'hypothèse que $\mid \Omega_{a,u} \mid$ est constant. Par contre pour les tests de complexité (en nombre de nœuds), on utilisera un test non parmétrique se rapprochant de l'utilisation de l'appariement des échantillons : le test de Wilcoxon.

15.14 φ_A est-il meilleur que φ_B en général ?

Pour évaluer la différence entre deux systèmes il est souhaitable d'utiliser des tests statistiques plutôt qu'une démonstration empirique au travers de plusieurs bases d'exemples. La seule démarche défendable serait d'effectuer une comparaison des méthodes sur chaque fichier, puis de créer une variable aléatoire γ_j qui prenne la valeur 1 lorsque l'algorithme A est meilleur que B sur la base j. Sous l'hypothèse nulle de performances égales, cette variable suit une loi de Bernouilli de paramètre 0.5. On peut alors effectuer un test des signes qui calcule pour J bases la quantité :

$$\Gamma = J^{-1} \sum_j \gamma_j$$

Et de voir si Γ est bien significativement supérieur à 0.5. Mais, bien que ce test soit vraiment simple, il est nécessaire pour son application de s'assurer de la représentativité des bases de données utilisées. Plus judicieux serait de limiter le domaine et la portée des fichiers utilisés. L'autre problème de ce test relève de la significativité de γ_j. Plus l'on sera permissif, plus grande sera la valeur finale de Γ.

15.15 Évaluation d'une classification probabiliste

Supposons que l'on veuille classer un élément inconnu alors que l'on dispose de k classes. Pour chaque classe on dispose d'une probabilité p_j que l'élément appartienne à la classe j. Supposons, enfin que la classe de l'élément soit i. Dans ce cas, si nos prédictions étaient correctes, on aurait :

$$a_l = \begin{cases} 0 & \text{si} \quad l \neq i \\ 1 & \text{si} \quad l = i \end{cases}$$

Il existe deux mesures pour évaluer la prédiction :

fonction de perte quadratique $\sum_{j=1}^{k} (p_j - a_j)^2$

fonction de perte informationnelle $-log p_i$

15.15.1 Le coût des erreurs

Supposons qu'un agriculteur se lance dans une mono-culture soit de maïs soit de betterave et qu'il ait évalué son gain pour chaque type en fonction du temps :

	Pluie	Soleil
maïs	1100	800
betterave	500	2000

Supposons, qu'il ait considéré que dans sa région pour la saison en cours la probabilité des événements pluie et soleil soit : P(pluie) = .6 ; P(soleil) = .4. Dans ce cas, son espérance de gain à partir de ses prédictions est :

maïs $.6 \times 1100 + .4 \times 800 = 980$

betterave $.6 \times 500 + .4 \times 2000 = 1100$

Le centre de météorologie lui propose un bulletin prévisionnel pour un coût de 100. Doit-il s'abonner ? sachant que :

$$\begin{array}{lcl} \text{P(MeteoPluie|Pluie)} & = & .8 \\ \text{P(MeteoSoleil|Pluie)} & = & .2 \\ \text{P(MeteoPluie|Soleil)} & = & .175 \\ \text{P(MeteoSoleil|Soleil)} & = & .825 \end{array}$$

La probabilité de l'événement MeteoPluie se calcule par :

$$\begin{array}{rcl} \text{P(MeteoPluie)} & = & \text{P(MeteoPluie|Pluie)P(Pluie)} + \text{P(MeteoPluie|Soleil)P(Soleil)} \\ & = & .8 \times .6 + .175 \times 4 \\ & = & .55 \end{array}$$

Tandis que l'événement Pluie sachant MeteoPluie s'obtient par :

$$\begin{array}{rcl} \text{P(Pluie|MeteoPluie)} & = & \frac{\text{P(MeteoPluie|Pluie)P(Pluie)}}{\text{P(MeteoPluie)}} \\ & = & \frac{.8 \times .6}{55} \\ & = & \frac{48}{55} \end{array}$$

Suivant le même principe on obtient la table suivante :

	MeteoPluie	MeteoSoleil
Pluie	$\frac{48}{55}$	$\frac{4}{15}$
Soleil	$\frac{7}{55}$	$\frac{11}{15}$

Avec $\text{P(MeteoSoleil)} = 1 - \text{P(MeteoPluie)} = .45$

À partir de ces valeurs, il est possible de calculer l'espérance de gain si le bulletin prévoit de la pluie :

maïs $\frac{48}{55} \times 1100 + \frac{7}{55} \times 800 = 1061$

betterave $\frac{48}{55} \times 500 + \frac{7}{55} \times 2000 = 690$

Si le bulletin prévoit du soleil :

maïs $\frac{4}{15} \times 1100 + \frac{11}{15} \times 800 = 880$

betterave $\frac{4}{15} \times 500 + \frac{11}{15} \times 2000 = 1600$

D'où l'on tire l'espérance de gain à partir du bulletin :

$$.55 \times 1061 + .45 \times 1600 = 1303.55$$

La valeur du bulletin est donc : $1303.55 - 1100 = 203.55$.

15.16 Traitement de la similarité

15.17 Analyse Formelle de Concept

Annexe A

Probabilités et Statistiques

Les modèles statistiques sont des précurseurs dans l'évaluation des différents modèles scientifiques. Afin de mieux comprendre ces modèles il est nécessaire de fournir un léger vernis sur les probabilités élémentaires et les statistiques. Cette partie est largement inspirée de [51, chap. 3].

A.1 Probabilité et estimation

Les probabilités servent à évaluer les chances que possèdent un certain évènement d'apparaître. L'idée de base est qu'une expérience peut être associée à un ou plusieurs effet/évènement. L'idée est de répéter l'expérience plusieurs fois est de comptabiliser le nombre de fois où un évènement apparaît. Si l'expérience est répétée plusieurs fois, la fréquence d'un évènement particulier est le nombre de fois où cet évènement a eu effectivement lieu, divisée par le nombre total d'expériences. Cette valeur *tend* vers une valeur particulière qui est la *probabilité* de l'évènement. En pratique, on ne peut répéter l'expérience un nombre infini de fois. En conséquence, on ne peut qu'*estimer* cette probabilité. Les statiques visent à quantifier la qualité de ces estimations.

Considérons l'expérience consistant à effectuer un lancer de dé. On peut alors déterminer le nombre de fois où l'on a obtenu un 6. Si l'on a effectué plusieurs essais (disons 3) `6,6,6`, le résultat n'est certainement pas fiable. On ne peut pas à partir de là tirer la conclusion générale suivante "avec un dé, faire un 6 est plus probable que faire un 2". Par contre on peut dire quelque chose comme "je crois avoir le truc pour faire des 6". Mais peut-être que ce n'était que de la chance ? les statistiques nous aident à quantifier l'incertitude. On peut ainsi dire : si toutes les faces ont autant de sortir, alors la probabilité que je fasse un 6 sur trois tirage est de $p\%$ ou moins. Si p est très petit, il est alors possible de rejeter l'hypothèse que le dé n'est pas pipé. C'est ce que l'on appelle l*hypothèse test*.

Il est aussi possible de placer un *Intervalle de Confiance* autour d'une valeur moyenne observée au cours d'une expérience. Connaissant les intervalles de confiance, il est possible de dire si un modèle correspond bien aux données observées.

A.2 Variables aléatoires

Une *variable aléatoire* est une variable qui modélise un évènement. Chaque valeur que peut prendre cette variable correspond à un évènement particulier auquel est associé une certaine probabilité. Reprenons l'exemple du lancer de dé. Et considérons la variable aléatoire X, si le dé n'est pas pipé on a $P(X = 1) = P(X = 2) = \ldots = P(X = 6) = \frac{1}{6}$. Si on lance deux dés, on dispose de deux variables aléatoires X et Y modélisant chacun des dés. Il est alors intéressant de définir la *probabilité conditionnelle*. $P(X = i|Y = j) = P(X = i, Y = j)/P(Y = j)$. C'est-à-dire que la probabilité de faire un 3 sachant que l'on a fait un 1 est égale à la probabilité de faire simultanément un 3 et un 1 divisée par la probabilité de faire un 1. Les variables X et Y sont dites *indépendantes* si le fait de connaître la valeur de Y n'a aucun effet sur la connaissance de X, en d'autres termes cela signifie que $P(X = i|Y = j) = P(X = i)$ Ce qui implique que $P(X = i, Y = j) = P(X = i) \times P(Y = j)$ et que

$P(X = i; Y = j) = P(X = i) + P(Y = j) - P(X = i) \times P(Y = j)$. Dans notre exemple sur les lancers de dés, les variables X et Y sont indépendantes.

Une *distribution de probabilité* est une fonction qui associe une probabilité à chaque valeur d'une variable aléatoire. Pour une variable aléatoire correspondant à un lancer de dé, chacune des 6 valeurs possède la même probabilité de $\frac{1}{6}$. C'est un exemple de distribution de probabilité uniforme.

De plus dans cet exemple la distribution est *discrète*, c'est-à-dire qu'il existe un nombre fini de valeurs possibles, chacune étant associée à une probabilité finie. Ce n'est pas toujours le cas.

A.3 Espérance et Variance

L'*espérance* d'une variable aléatoire est la somme des produits de chaque valeur que peut prendre cette variable et de la probabilité associée à cet évènement. Pour une variable discrète comme dans le cas du lancer de dé, l'espérance de X est donnée par :

$$E[X] = \sum_{i=1}^{6} P(X = i) \times i = \frac{1}{6} \times \frac{6 \times (6 + 1)}{2} = 3.5$$

Dans ce cas, l'expérance correspond à la moyenne arithmétique puisque l'on a une distribution de probabilité uniforme[1]. Si par contre la distribution n'est pas uniforme, mais que les évènements sont discrets on obtient une moyenne pondérée. Dans le cas d'une variable aléatoire continue l'espérance est donnée par :

$$E[X] = \int_{-\infty}^{+\infty} x\, P(x)\, dx$$

L'espérance est la mesure de la tendance centrale, ou "centre de masse" de la distribution de probabilité. La *variance* est une mesure toute aussi importante sinon plus. Elle sert à mesurer la dispersion de la distribution, c'est-à-dire la tendance qu'ont les valeurs de la distribution à être proche ou loin du centre de masse.

La variance est définie comme l'espérance de la différence entre chaque valeur et l'espérance. On la définit comme :

$$\text{var}[X] = E[(X - E[X])^2] = E[X^2] - E[X]^2$$

A.4 Arrangements et Combinaisons

Il est utile de connaître quelques notions simples de dénombrements lorsque l'on joue avec les probabilités. Supposons que l'on ait un alphabet de 4 lettres, et que l'on cherche le nombre de mots de 4 lettres distinctes que l'on peut former à partir de cet alphabet. Il y a 4 possibilités pour la première lettre, mais une fois celle-ci choisie, il ne reste que 3 possibilités pour la seconde lettre d'un mot, ... D'où le nombre total de mots de 4 lettres distinctes est $4 \times 3 \times 2 \times 1$, c'est le nombre d'arrangements possibles. Si on généralise à n lettres il y a alors $n!$ mots possibles.

Supposons maintenant que l'on veuille dénombrer le nombre de mots de 2 lettres, il y a 4 possibilités pour la première lettre et 3 pour la seconde, d'une manière générale, pour n lettres et des mots de $r < n$ lettres, il y a $A_n^r = \frac{n!}{(n-r)!}$ arrangements possibles.

Supposons maintenant que l'on souhaite maintenant s'intéresser aux lettres utilisées indépendemment de leur position dans le mot, si l'on reprend le premier exemple on s'aperçoit qu'il n'existe qu'un seul mot de 4 lettres. Dans le cas des mots de deux lettres, toujours sous les mêmes contraintes, on s'aperçoit qu'il n'y a que 6 combinaisons différentes (les mots "ab" et "ba" étant considérés comme identiques). D'une manière générale, pour n lettres et des mots de $r < n$ lettres, il y a $C_n^r = \frac{n!}{r! \times (n-r)!}$ combinaisons possibles. Cette formule se lit souvent "choisir r parmi n".

[1] $\sum_{i=1}^{n} i = \frac{n \times (n+1)}{2}$. Si la distribution est uniforme, la probabilité d'un évènement parmi n est égale à $\frac{1}{n}$, d'où : $\frac{1}{n} \sum_{i=1}^{n} i = \frac{n+1}{2}$

Annexe B

Résolution en Logique

Dans cette partie on trouvera les solutions aux exercices des chapitres 3 et 4

B.1 Rappel sur les arbres binaires

Un bref rappel sur la notion d'arbre binaire est fournie, le lecteur connaissant les concepts d'arbres binaires pourra sauter cette section.

B.2 Calcul des énoncés

Correction de l'exercice 3.13 page 57

Soluce :

Construisons la table de vérité à partir des variables propositionnelles a, b et c, et des déclarations respectives d'Arthur, Barnabé et Casimir :

a	b	c	Dit d'Arthur $(a \wedge \neg b)$	Dit de Barnabé $(a \supset b)$	Dit de Casimir $(\neg c \wedge (a \vee b))$
0	0	0	0	1	0
0	0	1	0	1	0
0	1	0	1	1	1
0	1	1	0	1	0
1	0	0	0	0	1
1	0	1	0	1	0
1	1	0	1	0	1
1	1	1	0	1	0

On utilisera A pour désigner ce que dit Arthur, B pour ce que dit Barnabé et C pour ce que dit Casimir.

1. On considère les lignes où $I[\![C]\!] = 0$ et on trouve $I[\![A]\!] = 0$, donc si Casimir ment, Arthur aussi.
2. On considère les lignes où $I[\![C]\!] = 0$ et on trouve $I[\![B]\!] = 1$, donc si Casimir ment, Barnabé dit la vérité.
3. Par hypothèse on a $I[\![A]\!] = I[\![B]\!] = I[\![C]\!] = 1$, c'est la troisième ligne et on trouve que $I[\![a]\!] = I[\![c]\!] = 0$ et $I[\![b]\!] = 1$, donc si tous disent la vérité Arthur et Casimir sont innocents et Barnabé est le coupable.
4. Par hypothèse, on $I[\![a]\!] = I[\![b]\!] = I[\![c]\!] = 1$, c'est la huitième ligne et on obtient $I[\![A]\!] = I[\![C]\!] = 0$ et $I[\![B]\!] = 1$, donc si tous sont coupables Arthur et Casimir mentent, Barnabé dit la vérité.

5. On ne considère que les lignes telles que I⟦a⟧ = 1 - I⟦A⟧ et I⟦b⟧ = 1 - I⟦B⟧ et I⟦c⟧ = 1 - I⟦C⟧. Seule la ligne 6 est possible, donc Barnabé est innocent, Arthur et Casimir sont coupables.
6. On ne considère que les lignes telles que I⟦a⟧ = I⟦A⟧ et I⟦b⟧ = I⟦B⟧ et I⟦c⟧ = I⟦C⟧. Aucune ligne ne correspond, l'hypothèse est contradictoire.

♡

Correction de l'exercie 3.3 page 42

Soluce :

Nous nous restreindrons à démontrer les équivalences entre les expressions n'utilisant que les connecteurs de la base pour exprimer les expressions simples de $\mathcal{F}$.

1. Ici, il suffit d'exprimer les connecteurs $\supset$ et $\equiv$, comme $\equiv$ s'exprime à partir des connecteurs $\supset$ et $\wedge$[1], il suffit de se concentrer sur le seul connecteur $\supset$. La solution est d'ailleurs donnée dans le cours à la page 41, lorsque l'on a établi que $\{\neg, \supset\}$ formait une base[2].
2. Dans le cas de l'opérateur ternaire, on a les équivalences logiques suivantes $\texttt{ite}(p, 0, 1) = \neg p$, $\texttt{ite}(p, 1, 0) = p$, $\texttt{ite}(p, 1, q) = (p \vee q)$, $\texttt{ite}(p, q, 0) = (p \wedge q)$. À partir de là, comme on sait que $\{\neg, \wedge, \vee\}$ est une base, on peut construire les expressions représentant $(p \supset q)$ et $(p \equiv q)$.
3. Pour établir que l'opérateur non-et constitue une base, il suffit de prouver que cet opérateur à la même puissance d'expression qu'une autre base. $(p \uparrow p) = \neg p$, $((p \uparrow p) \uparrow (p \uparrow p)) = \neg\neg p = p$, $(p \uparrow q) = \neg(p \wedge q) = (\neg p \vee \neg q)$. Cette dernière expression, nous donne la clef pour exprimer une expression utilisant le connecteur $\wedge$ et/ou le connecteur $\vee$. On obtient rapidement que $((p \uparrow p) \uparrow (q \uparrow q)) = (p \vee q)$ et que $((p \uparrow q) \uparrow (p \uparrow q)) = (p \wedge q)$.
4. Pour établir que l'opérateur non-ou constitue une base, il suffit de prouver que cet opérateur à la même puissance d'expression qu'une autre base. $(p \downarrow p) = \neg p$, $((p \downarrow p) \downarrow (p \downarrow p)) = \neg\neg p = p$, $(p \downarrow q) = \neg(p \vee q) = (\neg p \wedge \neg q)$. Cette dernière expression, nous donne la clef pour exprimer une expression utilisant le connecteur $\wedge$ et/ou le connecteur $\vee$.

♡

Correction de l'exercice 3.20 page 59

Soluce :

On sait que l'on doit choisir une porte et une seule. Les propriétés P1-2 et P1-3 sont opposées, on en déduit que P1-1 est nécessairement fausse puisqu'une seule proposition est vraie. Il faut donc choisir la porte rouge, ce qui a pour conséquence de rendre l'assertion P1-3 fausse et P1-2 vraie.

Dans le deuxième exercice, on s'aperçoit que P2-1 et P2-3 sont opposées, le texte nous indiquant qu'une seule proposition est fausse, P2-2 est nécessairement vraie, il faut donc chosir le couloir de droite. Ce choix rend P2-1 vraie et P2-3 fausse.

Les assertions P3-2 et P3-4, nous conduisent à la conclusion qu'il y a exactement deux assertions vraies, les deux autres étant fausses, l'escalier montant mène à l'étape suivante. Peut-on avoir une autre solution ? Intuitivement le texte ne nous donne aucun moyen de différencier les escaliers de même type, donc seul l'escalier montant peut nous conduire où nous désirons aller.

Le texte nous indique que l'on doit sélectionner deux statues parmi quatre. Il y a, à priori 6 solutions $\begin{pmatrix} 2 \\ 4 \end{pmatrix}$. La troisième assertions réduit le choix à 3 possibilités : (connaissance fécondité), (connaissance prospérité), (sagacité prospérité). Si l'on choisit de ne pas prendre connaissance (selon P4-1), P4-2 nous laisse sans solution. Si l'on choisit de laisser de côté prospérité, P4-2 nous laisse une solution. ♡

[1] Pour mémoire on $(p \equiv q) = ((p \supset q) \wedge (q \supset p))$

[2] $(p \supset q) = (\neg p \wedge q)$

Terminons par la correction de l'exercice 3.21 page 60.

Soluce :

Posons a "Alfred est coupable", b "Baptiste est coupable" et c "Charly est coupable". Résolvons les problèmes en mettant les assertions sous forme clausale. D'après le texte, la clause {a, b, c} fait partie de chaque énoncé.
Pour l'attaque de la bijouterie, les assertions se retranscrivent :

A-1 $\neg c \supset a$ dont la fnc est $c \vee a$

A-2 $a \supset (b \vee c) \wedge \neg(b \wedge c)$ dont la fnc est $\neg a \vee (b \wedge \neg c) \vee (\neg b \wedge c) = (\neg a \vee b \vee c) \wedge (\neg a \vee \neg c \vee \neg b)$

A-3 $\neg b \supset \neg c$ dont la fnc est $b \vee \neg c$.

A-4 $(b \wedge c) \supset a)$ dont la fnc est $a \vee \neg b \vee \neg c$.

D'où l'ensemble de clauses $\{\{a, b, c\}, \{a, c\}, \{\neg a, b, c\}, \{\neg a, \neg b, \neg c\}, \{b, \neg c\}, \{a, \neg b, \neg c\}\}$. Que l'on peut simplifier en $\{\{a, c\}, \{\neg a, b, c\}, \{\neg a, \neg b, \neg c\}, \{b, \neg c\}, \{a, \neg b, \neg c\}\}$. À partir des clauses c_1 et c_2 on trouve $\{b, c\}$ si l'on utilise maintenant c_4 on trouve $\{b\}$, donc Baptiste est coupable, l'ensemble réduit devient $\{\{a, c\}\{\neg a, \neg c\}, \{a, \neg c\}\}$. D'où l'on déduit $\mathrm{I}[\![a]\!] = 1$ et $\mathrm{I}[\![c]\!] = 0$. Baptiste et Alfred sont donc responsable de l'attaque de la bijouterie.
En utilisant le même cheminement, on trouve que Baptiste et Charly ont effectué l'attaque de la banque, tandis que Alfred est seul coupable dans l'attaque de la pharmacie.
Finalement, Alfred écopera de 10 ans, Baptiste de 13 ans et Charly de 7 ans. À leur sortie de prisons ils auront tous 40 ans. ♡

B.3 Logiques déontique, épistémique, doxastique

Cette partie reprend la présentation de : `http://logique.uqam.8m.com/histoire11.htm` La logique déontique est une structure logique des systèmes de valeurs, qui s'intéresse aux actions plutôt qu'aux événements. On doit son développement principalement à von Wright (1951). Il s'agit de traiter formellement énoncés formés d'expressions telles que : « il est permis de », « il est interdit de », « il est facultatif de »ou « il est obligatoire de ». Ces expressions « modalisent »les énoncés de la même façon que les modalités le font. D'ailleurs, on peut faire correspondre les premières avec les secondes.

B.3.1 Les notations déontiques

1. L'obligation (O) : elle correspond à la nécessité modale.
2. L'interdiction (I) : elle correspond à l'impossibilité modale.
3. La permission (P) : elle correspond à la possibilité modale.
4. Le facultatif (F) : elle correspond à la contingence modale.

Dans le système de von Wright, la moralité implique qu'à chaque instant du temps, il est possible de transformer le monde actuel en monde admissible ou inadmissible. Ceci fait évidemment référence à la théorie des mondes possibles et aussi à la théorie du « branching time ». Certains principes classiques s'appliquent à la logique déontique, d'autres ne le sont pas compte tenu de leur contenu sémantique.

Les axiomes de la logique déontique

- Le principe de distribution déontique : $\mathrm{P}(A \vee B) \supset (\mathrm{P}(A) \vee \mathrm{P}(B))$
- Le principe de permission : $\mathrm{P}(A) \vee \mathrm{P}(\neg A)$

Les principes classiques non applicables à la logique déontique

- Le paradoxe de l'obligation dérivée : le principe du « faux implique tout »ne s'applique pas à la logique déontique. Il est faux que : $\mathrm{O}(\neg p) \supset \mathrm{O}(p \supset q)$.
- Le paradoxe de Ross : le principe d'adjonction ne s'applique pas à la logique déontique. Il est faux que : $\mathrm{O}(p) \supset \mathrm{O}(p \vee q)$.
- La distinction entre monde actuel et monde admissible : contrairement à la logique modale, où le monde actuel peut être un monde possible, le monde actuel de la logique déontique ne peut être un monde admissible, sinon aucune action immorale n'aurait lieu dans le monde actuel. Il est faux que : $\mathrm{O}(p) \supset p$.

Plusieurs logiciens (Prior, McLaughlin) trouveront par la suite d'autres problèmes fondamentaux dans le système déontique de von Wright. Ces problèmes sont souvent dus à la sémantique du système et au contenu des concepts normatifs. La question ontologique est également cruciale : peut-on donner une signification aux énoncés normatifs sans présupposer l'existence d'un monde de « devoir-être ? »

Les développements subséquents ont été faits en ce sens, entre autres en remettant en question un principe de base du système de von Wright : la permission peut être définie (par analogie avec la possibilité) par l'absence de l'interdiction correspondante. En rejetant cette affirmation, on règle plusieurs problèmes sémantiques, mais se pose alors un problème de cohérence interne. Aussi, plutôt que de conserver cette définition de « permisssion faible », on introduit la notion de « permission forte », i.e. qu'une action sera permise dans la mesure où l'autorité concernée le permet et non uniquement parce qu'elle n'est pas interdite. Toutefois, cette nouvelle notion soulève elle aussi quelques problèmes.

La logique déontique peut être interprétée dans la perspective de la sémantique kripkienne des mondes possibles. D'ailleurs, le fait d'adopter cette sémantique permet de mettre de côté la question de savoir si les énoncés normatifs peuvent être dits vrais ou faux. Sur le plan technique, ils peuvent effectivement être vrais ou faux, en termes de compatibilité ou incompatibilité entre les mondes possibles.

B.3.2 La logique épistémique et doxastique

La théorie de la connaissance cherche entre autres à définir des concepts comme la croyance et le savoir. Par la suite, dans la mesure où une telle définition est possible, on peut penser à formaliser leurs relations mutuelles et même à en faire un système axiomatique. La logique épistémique se donne comme objectif de construire un système formel où certaines modalités représentent les concepts de croyance et de savoir. Il faut toutefois noter que le système formel n'est pas épistémique, mais peut être interprété de manière épistémique.

On doit d'abord le développement de cette logique épistémique à Hintikka (1962), puis à Goldman (1967) et enfin à Dretske (1981), qui l'a reformulée dans le cadre de la théorie de l'information. Une des contributions importantes de Hintikka, en plus d'avoir formulé les divers systèmes, consiste à considérer séparément les notions de savoir et de croyance, afin de développer une logique épistémique et une logique doxastique.

La logique épistémique (Ka)

1. Le domaine de ce système est constitué d'événements (p), d'agents (a) et des opérateurs : (K) plus ou moins équivalent à la modalité nécessaire et représentant le savoir, ainsi que (P), qui signifie qu'un événement est possible relativement au savoir de l'agent. Il s'agit d'une modalité faible qui permet la définition de (K) par la négative.

2. Plusieurs principes classiques peuvent s'appliquer à ce système, entre autres le modus ponens, la réflexivité (K$ap \supset p$) qui sous-entend la notion de vérificationnisme ainsi que la sérialité (K$ap \supset$ Pap), qui permet une ordonnance des opérateurs comme dans la logique modale.

3. La sémantique proposée par Hintikka pour la logique épistémique repose sur la notion « d'ensemble modèle », qui peut être reformulé en termes de sémantique des mondes possibles. Ces mondes possibles seront compatibles avec le monde actuel dans la mesure où leur contenu sera compatible avec le savoir dont il est question dans le monde actuel.

La logique doxastique (Ba)

1. Le domaine de ce système est constitué d'événements (p), d'agents (a) et des opérateurs : (B) plus ou moins équivalent à la modalité nécessaire et représentant la croyance, ainsi que (C), qui signifie qu'un événement est compatible avec la croyance de l'agent. Il s'agit d'une modalité faible qui permet la définition de (B) par la négative.

2. Le modus ponens est aussi applicable à ce système ainsi que la sérialité (B$ap \supset$ Cap), mais pas la réflexivité (B$ap \supset p$).

3. La sémantique de la logique doxastique réfère également à la notion kripkienne de mondes possibles.

Annexe C

Plus d'information sur Scheme

Dans cette partie, outre quelques éléments de syntaxe nécessaire à l'expérimentation du langage SCHEME on trouvera les solutions aux exercices posés dans le chapitre 7, pour plus de clartés les textes sont reportés juste avant les solutions.

C.1 Éléments du langage

On a parfois besoin d'utiliser en SCHEME des fonctionnelles pour lesquelles on ne souhaite pas donner de nom, pour cela on utilisera la forme `lambda`, qui s'écrira : `(lambda liste_param corps)`. Ainsi, la fonction identité $\lambda x.x$ s'obtiendra par `(lambda (x) x)`.
Il est parfois utile d'acceder à des commandes système, cela est prévu en utilisant la fonctionnelle `system` qui prend en argument une chaîne de caractères qui est la commande système à exécuter. Pour connaître, sous Unix, le contenu du répertoire courant, on pourra, sans sortir de SCHEME écrire : `(system "ls")`, pour visualiser le contenu d'un fichier toto.scm, on tapera : `(system "more toto.scm")`.
Il arrive parfois, que l'on souhaite mettre plusieurs instructions à la suite les unes des autres, pour cela on utilisera l'ordre `begin` en utilisant la syntaxe suivante : `(begin exp_1 ...exp_n)`, les expressions sont évaluées en séquence et la valeur de l'expression initiale est la valeur associée à la dernière expressions évaluées (`exp_n`).
On a vu qu'il était possible d'empécher l'évaluation d'une expression en faisant précéder celle-ci d'une "quote" `'` ou `(quote exp)`. Il est cependant parfois nécessaire de forcer l'évaluation d'une expression, cela peut être réalisé en utilsant la fonctionnelle `eval`, ainsi l'expression `(eval (quote exp))` aura le même effet que `exp`.

C.2 Solutions

Les programmes donnés en annexe sont accessibles en envoyant un courrier (électronique) à l'auteur de ces quelques pages `corsini@u-bordeaux2.fr`.

Commençons par la solution de l'exercice 7.1 page 81 :

1. Écrire la fonction factorielle définie par :

$$\begin{cases} f(0) & = & 1 \\ f(n) & = & n * f(n-1) \quad \forall n > 0 \end{cases}$$

2. Écrire la fonction de fibonacci définie par

$$\begin{cases} f(1) & = & 1 \\ f(2) & = & 1 \\ f(n) & = & f(n-2) + f(n-1) \quad \forall n > 2 \end{cases}$$

Soluce :

```
;; la fonction factorielle non minimale
(define (fact n)
   (cond ( (> 0 n) 'error)
         ( (= 0 n) 1)
         ( (< 0 n) (* n (fact (- n 1))))
        )
)

;; fibonacci peu efficace ...

(define (fib n)
  (cond ((< n 0) 'erreur)
        ((or (= n 0) (= n 1)) 1)
        (else (+ (fib (- n 2)) (fib (- n 1))))))
```

♡

Regardons maintenant comment on peut résoudre le problème posé dans l'exercice 7.2 page 81 : Écrire une fonction (voire plusieurs) qui permettent de :

1. faire la somme des n premiers termes,
2. faire la somme des carrés des nombres pris dans un intervalle,
3. faire la somme des cubes des nombres impairs pris dans un intervalle,
4. faire le développement limité à l'ordre *n* de la fonction cosinus.

Soluce :

```
(define (somme fun var binf bsup next)
  (define (aux res val)
    (if (or (> val bsup) (< val binf))
        res
        (aux (+ res (fun val)) (next val))))
  (aux 0 var))

;; une fonction sigma legerement differente mais en quoi ...
(define (sigma fun a b suivant)
  (define (aux a b resultat)
    (if (>  a b) resultat
        (aux (suivant a) b (+ resultat (fun a)))))
  (aux a b 0))

(define (plus_un x) (+ 1 x))

(define (somme-int n)
  (somme (lambda (x) x) 0 0 n plus_un))

(define (terme-cos x)
  (lambda (n)
    (/
     (* (puiss x (* 2 n))
        (if (even? n) 1 -1))
     (fact (* 2 n)))))
```

```
(define (fact n)
  (define (aux val res)
    (if (= val 1) res
        (aux (- val 1) (* res val))))
  (cond ((< n 0) 'error:FACT)
        ((= n 0) 1)
        (else (aux n 1))))

(define (puiss x n)
  (define (puiss_aux x n res)
    (cond ((= n 1) (* res x))
          ((even? n) (puiss_aux (* x x) (/ n 2) res))
          (else (puiss_aux x (- n 1) (* res x)))))
  (cond ((= n 0) 1)
        ((= n 1) x)
        ((= x 1) 1)
        ((= x 0) 0)
        (else (if (> n 0) (puiss_aux x n 1) 'error:PUISS))))


(define (DLcos x ordre)
  (somme (terme-cos x) 0 0 ordre plus_un))

(define (Cosinus x)
  (DLcos x 20))


;; la suite des nombres pairs
(define (suiv-pair x)
  (if (= 0 (remainder x 2)) (+ x 2) (+ 1 x)))

;; un cube
(define (cube x)
  (* x x x ))

; la somme des cubes
(define (sigma-cube n)
  (sigma cube 1 n next))

; une somme un peu lourde
; a noter la lambda expression
(define (somme_int n)
  (sigma (lambda (x) x) 1 n next))

; une somme rapide
(define (fast-som n)
  (/ (* n (+ 1 n)) 2))

; un successeur
(define (next a)
  (+ 1 a))
```

♡

Pour résoudre l'exercice 7.3 page 82 qui, étant données trois listes

```
l1 = ((a b) a b)
l2 = (a)
l3 = (b)
```

Demande de calculer le `car` et `cdr` pour chaque liste, construire `l1` à partir de `l2` et `l3`, il suffisait d'écrire :

Soluce :

```
(car l1) ⇝ (a b)
(cdr l1) ⇝ (a b)
(car l2) ⇝ a
(cdr l2) ⇝ ()
(car l3) ⇝ b
(cdr l3) ⇝ ()
(cons (car l2) l3) ⇝ (a b)
(cons (cons (car l2) l3) (cons (car l2) l3)) ⇝ l1
```

À noter que s'il on avait voulu écrire directement les listes, il eut fallu les faire précéder par une quote :

```
(car '( (a b) a b)) ⇝ (a b)
```

♡

Voici une solution possible de l'exercice 7.5 page 83.

Soluce :

1. Écrire la fonction `concat`, qui, étant données deux listes `l1` et `l2`, calcule une troisième liste `l3` qui contient, dans l'ordre les éléments de `l1` suivis des éléments de `l2`. Cette opération est connue comme la concaténation de deux listes. À titre d'exemple :

```
(define (concat l1 l2)

 (define (app first second)
  (if (null? first) second
      (cons (car first) (app (cdr first) second))))

 (cond ((null? l1) l2)
       ((null? l2) l1)
       ((not (and (pair? l1) (pair? l2))) 'erreur)
       (else (app l1 l2))))
```

2. Calculer la longueur `longueur` d'une liste i.e. le nombre d'éléments qui la constituent. Si on considère l'exemple précédent, on vérifie que :

```
(define (longueur l)

 (define (long liste acc)
  (if (null? liste) acc
      (long (cdr liste) (+ 1 acc))))

 (cond ((null? l) 0)
       ((pair? l) (long (cdr l) 1))
       (else 'erreur))
)
```

3. Calculer la profondeur `pf` d'une liste définie comme le nombre maximum d'imbrications de parenthèses. Si on reprend l'exemple donné pour la concaténation, on doit vérifier que :

```
(define (pf l)
 (if (not (pair? l)) 0
     (+ 1 (maximum (pf (car l))
                   (-1+ (pf (cdr l)))))))

(define (maximum x y)
```

```
(if (> x y) x y))
```

4. Écrire une fonction `elem?` qui permet de savoir si un objet appartient à une liste. Toujours dans l'exmple pris au début de cet exercice, on vérifiera que :

```
;; cette fonction ne marche que pour la recherche des elements
;; de premier niveau i.e.
;; (elem? 4 '(3 4 (5 6))) ⇝ #t
;; (elem? 5 '(3 4 (5 6))) ⇝ #f
(define (elem? x liste)
 (cond ((not (pair? liste)) #f)
       ((eqv? x (car liste)) #t)
       (else (elem? x (cdr liste)))))
```

Si on veut que la recherche se fasse en profondeur, on peut d'abord aplatir la liste, et ensuite faire la recherche ce qui donne :

```
(define (flatten l)
 (cond ((not (pair? l)) l)
       ((not (pair? (car l))) (cons (car l) (flatten (cdr l))))
       (else (concat (flatten (car l)) (flatten (cdr l))))))

(define (elemf? x liste)
 (elem? x (flatten liste)))
```

L'inconvénient majeur de cette approche étant de parcourir 2 fois une liste, la première pour l'aplatir, la seconde pour regarder s'il y a appartenance. C'est pourquoi on peut écrire directement :

```
(define (element? x l)
 (cond ((not (pair? l)) #f)
       ((pair? (car l))
        (or (element? x (car l))
            (element? x (cdr l))))
       (else
        (or (eqv? x (car l))
            (element? x (cdr l))))
       ))
```

♡

Regardons comment on peut aborder la résolution de l'exercice 7.6 page 7.6.

Soluce :

```
(define (CreerRetraitAmnesie solde)
  (lambda (montant)
    (if (> montant solde) "ERREUR, compte trop faible"
        (- solde montant))))

(define amnesique (CreerRetraitAmnesie 10000))
```

Dans cette première version, qui ne permet que le retrait de valeurs, on s'aperçoit qu'il n'y a pas de mémoire sur les différentes opérations ainsi, l'exécution donnera :

```
> (amnesique 500)
⇝  9500
> (amnesique 500)
⇝  9500
```

Alors que pour la deuxième instruction on eut souhaité récupérer la valeur `9000`. Pour se rapprocher de la solution il faut écrire le bout de programme suivant :

```
;;; VERSION 0

(define (CreerRetrait solde)
  (lambda (montant)
    (if (> montant solde) (error "ERREUR, compte trop faible")
        (begin (set! solde (- solde montant))
               solde))))

(define ComptePierre (CreerRetrait 1000)) ;; meme valeur initiale mais
(define ComptePaul (CreerRetrait 1000))  ;; identite differente

(define CompteJacques (CreerRetrait 2000)) ;; meme valeur initiale
(define CompteJeanne CompteJacques)         ;; meme evolution
```
♡

EXEMPLE C.2.1

Au moment d'utiliser ces fonctions on obtient :

```
> (ComptePierre 50)
⇝ 950
> (ComptePaul 50)
⇝ 950
> (CompteJacques 50)
⇝ 1950
> (CompteJeanne 50)
⇝ 1900
```
◁

Rapprochons nous de la version définitive en intégrant les paramètres `solde, retrait, virement` :

Soluce :

```
(define (cc solde)
  (lambda (operation x)
    (cond ((eqv? operation 'solde)
           solde)
          ((eqv? operation 'retrait)
           (begin
             (set! solde (- solde x))
             solde))
          ((eqv? operation 'virement)
           (begin
             (set! solde (+ solde x))
             solde))
          (else
           (begin
             (display "OPERATION AUTORISEE: solde retrait virement")
             (error "format: <compte> <operation> <valeur>")
             )))))

(define first (cc 10000))
```
♡

L'inconvénient majeur de cette approche est de nécessité l'usage d'un paramètre fictif lorsque l'on cherche le solde du compte courant. Pour circonvenir ce problème, il est possible de passer par la notion de paire pointée, la notation `(a . l)` étant une abreviation commode pour accéder directement au `car` et `cdr` d'une liste. D'où la version finale :

Soluce :

```
(define (CC* solde)
;; in renvoie 0 si arg n'est pas dans l
```

```
;; in renvoie la position de arg dans l sinon
 (define (in arg l)
  (define (position elem liste ou)
    (cond ((null? liste) 0)
          ((eqv? elem (car liste)) ou)
          (else (position elem (cdr liste) (+ 1 ou)))))
  (if (pair? l) (position arg l 1) 0))

 (lambda (arg1 . Reste)
  (let ((local (in arg1 '(solde retrait virement))))
       (if (= local 0) (GestionErreur (cons arg1  Reste))
           (if (= local 1) solde
               (if (not (null? Reste))
                   (begin
                     (set! solde
                           (+ solde
                              (* (cond ((= local 2) -1)
                                       ((= local 3) 1)
                                       (else 0))
                                 (car Reste))))
                     solde)
;; On rajoute un argument pour générer un message spécifique
;; lorsque l'appel est incomplet
                   (GestionErreur (list arg1 'param)))))))

(define GestionErreur
  (lambda (liste)
    (begin
      (newline)
      (display "format: <compte> <operation> <valeur>")
      (newline)
      (display "<operation> : solde retrait virement ")
      (if (and (pair? (cdr liste)) (eqv? (cadr liste) 'param))
          (error  "NOMBRE DE PARAMETRES INSUFFISANTS: " (car liste))
          (error  "OPERATION NON AUTORISEE: " (car liste)))
    )))
```

♡

Ci-dessous un exemple possible de session :
EXEMPLE C.2.2

```
> (define mmc (CC* 10000))
> (mmc 'solde)
↪ 10000
> (mmc 'retrait 5000)
↪ 5000
> (mmc 'virement 500)
↪ 5500
> (mmc 'agio)
↪
format: <compte> <operation> <valeur>
<operation> : solde retrait virement
ERROR: OPERATION NON AUTORISEE:  agio
```

◁

Dans la solution de l'exercice 7.7 ci-dessous on a essayé d'insérer les commentaires nécessaires à une compréhension directe du code fourni.

Soluce :

```
;; debut de derivation formelle
;; DeriveSimple retourne une version simplifiée de la dérivée
(define (DeriveSimple exp var)
  (simplify (derive (simplify exp) var)))

;; on ne derive que les fonctions arithmétiques de base
(define (derive exp var)
  (cond
   ((constante? exp) 0)
   ((variable? exp) (if (eq? exp var) 1 exp))
   (else
    (let
        ((exp1 (terme1 exp))
         (exp2 (terme2 exp)))
      (cond
       ((addition? exp)  (list '+ (derive exp1 var)
                               (derive exp2 var) ))
       ((multiplication? exp) (list '+
                                    (list '*
                                          (derive exp1 var) exp2)
                                    (list '* exp1 (derive exp2 var))
                                    ))
       )) ; fin du let local
    )))

;; une série de petits utilitaires
;; terme1 récupère la première expression
(define (terme1 exp)
  (cadr exp))

;; terme2 récupère le deuxième terme de l'expression
;; à dériver
(define (terme2 exp)
  (caddr exp))

;; un test pas si utile
(define (multiplication? exp)
  (eq? (car exp) '*))

;; cf plus haut
(define (addition? exp)
  (eq? (car exp) '+))

;; on exprime ici qu'une constante est
;; soit un nombre
;; soit une lettre qui n'est pas une variable
(define (constante? exp)
  (or (number? exp)
      (member? exp '(a b c d e f g h i j k l m n o p q r s t u v w))
      ))

;; les seules variables sont x, y et z
(define (variable? v)
  (member? v '(x y z)))

(define (member? v l)
  (if (eq? l '()) #f
      (or
       (eq? v (car l))
```

```
       (member? v (cdr l))
       )))

(define (simplify exp)
  (if (pair? exp)
      (cond
       ((addition? exp) (SimplAdd exp))
       ((multiplication? exp) (SimplMult exp))
       )
      exp))

(define (SimplAdd exp)
  (let
      ((exp1 (simplify (terme1 exp)))
       (exp2 (simplify (terme2 exp))))
    (cond
     ((eq? exp1 0) exp2)
     ((eq? exp2 0) exp1)

;; essayons d'evaluer les expressions numeriques simples

     ((and (number? exp1) (number? exp2)) (+ exp1 exp2))
     (else (list '+ exp1 exp2))
     )))

(define (SimplMult exp)
  (let
      ((exp1 (simplify (terme1 exp)))
       (exp2 (simplify (terme2 exp))))
    (cond
     ((eq? exp1 0) 0)
     ((eq? exp2 0) 0)
     ((eq? exp1 1) exp2)
     ((eq? exp2 1) exp1)

;; essayons d'evaluer les expressions numeriques simples

     ((and (number? exp1) (number? exp2)) (* exp1 exp2))
     (else (list '* exp1 exp2))
     )))
```

♡

EXEMPLE C.2.3

```
> (derive '(+ (* x a) (* b (+ (* 5 x) x))) 'x)
↪ (+ (+ (* 1 a) (* x 0)) (+ (* 0 (+ (* 5 x) x)) (* b (+ (+ (* 0 x) (* 5 1)) 1))))
> (define a 5)
> (define b 2)
> (define x 1)
> (eval (derive '(+ (* x a) (* b (+ (* 5 x) x))) 'x))
↪ 17
> (DeriveSimple '(+ (* x a) (* b (+ (* 5 x) x))) 'x)
↪ (+ a (* b 6))
> (eval (DeriveSimple '(+ (* x a) (* b (+ (* 5 x) x))) 'x))
↪ 17
```

◁

Le dernier exercice du chapitre 7, page 85 a pour objectif de gérer via des S-expressions des Arbres Binaires de Recherche. On considère que la relation d'ordre est "cablée" dans le programme, et ce afin d'alléger l'écriture des différentes fonctions nécessaires à la résolution du problème.

Soluce :

```
;; Gestion d'arbre binaire de recherche
;; L'ordre cablé est >
;; Si une valeur est > a la racine elle va dans le sous arbre droit

;; Un arbre Binaire est
;; soit l'arbre vide, noté ()
;; soit un triplet (racine fg fd)
;; ou fg et fd sont des arbres
;; une feuille sera représentée par (rac) ou par (rac () ())

;; savoir si un arbre est une feuille
(define (feuille? tree)
 (or (null? (cdr tree))
     (and (VideTree? (fg tree)) (VideTree? (fd tree)))))

;; acces au sous arbre gauche
(define (fg tree)
 (cadr tree))

;; acces au sous arbre droit
(define (fd tree)
 (caddr tree))

;; acces a la racine
(define (root tree)
  (car tree))

;; savoir si l'arbre est vide ou non
(define (VideTree? tree)
 (null? tree))

;; hauteur d'un arbre
(define (hauteur tree)
  (cond ((VideTree? tree) 0)
        ((feuille? tree) 1)
        (else   (+ 1 (maximum (hauteur (fg tree))
                              (hauteur (fd tree)))))
        )
  )

(define (maximum x y) (if (>= x y) x y))

;; LCE
(define (LCE tree)
  (define (f h arbre)
    (cond ((VideTree? arbre) 0)
          ((feuille? arbre) h)
          (else (+ (f (+ 1 h) (fg arbre))
                   (f (+ 1 h) (fd arbre))))
          ))
  (f 1 tree))

;; LCI
(define (LCI tree)
  (define (f h arbre)
    (cond ((VideTree? arbre) 0)
          ((feuille? arbre) 0)
```

```
          (else (+ h
                   (f (+ 1 h) (fg arbre))
                   (f (+ 1 h) (fd arbre))))
          ))
  (f 1 tree))

;; LC = LCE + LCI
(define (LC tree)
  (define (f h arbre)
    (cond ((VideTree? arbre) 0)
          ((feuille? arbre) h)
          (else (+ h
                   (f (+ 1 h) (fg arbre))
                   (f (+ 1 h) (fd arbre))))
          ))
  (f 1 tree))

;; parcours infixe
(define (infixe tree)
  (cond
   ((VideTree? tree) '())
   ((feuille? tree) (list (root tree)))
   (else (append
          (infixe (fg tree))
          (cons
           (root tree)
           (infixe (fd tree)))
          ))))


;; creer un ABR
(define CreerTree
 (lambda () '()))

;; ajouter un sommet dans un ABR
(define (AjoutTree sommet tree)
  (cond ((VideTree? tree) (list sommet))
        ((feuille? tree)
         (if (<= sommet (root tree)) (list (root tree) (list sommet) '())
             (list (root tree) '() (list sommet))))
        (else
         (if (<= sommet (root tree)) (list (root tree)
                                           (AjoutTree sommet (fg tree))
                                           (fd tree))
             (list (root tree)
                   (fg tree)
                   (AjoutTree sommet (fd tree)))))
        )
  )




;; Un element est-il présent ou non dans un arbre donné
(define (InTree? sommet tree)
  (cond ((VideTree? tree) #f)
        ((feuille? tree) (eqv? sommet (root tree)))
        (else
         (cond ((eqv? sommet (root tree)) #t)
```

```
                ((< sommet (root tree)) (InTree? sommet (fg tree)))
                (else (InTree? sommet (fd tree)))))
          ))


;; Supprimer un élément dans un ABR et garder la propriété d'ABR
(define (LeftMost tree)
  (cond ((VideTree? tree) tree)
        ((feuille? tree) (root tree))
        ((VideTree? (fg tree)) (root tree))
        (else (LeftMost (fg tree)))))

(define (RightMost tree)
  (cond ((VideTree? tree) tree)
        ((feuille? tree) (root tree))
        ((VideTree? (fd tree)) (root tree))
        (else (RightMost (fd tree)))))

(define (OteTree fun elem tree)
  (cond
   ((VideTree? tree) tree)
   ((eqv? elem (root tree))
    (cond
     ((feuille? tree) '())
     ((VideTree? (fg tree)) (fd tree))
     ((VideTree? (fd tree)) (fg tree))
     (else
      (cond
       ((eqv? fun 'LeftMost)
        (let ((value ((eval fun) (fd tree))))
          (list value (fg tree) (OteTree fun value (fd tree)))))
       ((eqv? fun 'RightMost)
        (let ((value ((eval fun) (fg tree))))
          (list value (OteTree fun value (fg tree)) (fd tree))))
       (else (error "ERREUR, fonction indéfinie")))))))
   ((> elem (root tree))
    (list (root tree)
          (fg tree)
          (OteTree fun elem (fd tree))))
   (else
    (list (root tree)
          (OteTree fun elem (fg tree))
          (fd tree)))))


;;; Petite Initialisation
(define arbre
  (AjoutTree
   4
   (AjoutTree
    6
    (AjoutTree 2 (AjoutTree 5 (CreerTree))))))
```

♡

Et voici un exemple de session possible :

EXEMPLE C.2.4

```
12:26:00 hobbes-mmc-bash:  scm tree.scm
```

```
> arbre
↪ (5 (2 () (4)) (6))
> (infixe arbre)
↪ (2 4 5 6)
> (infixe (OteTree 'LeftMost 5 arbre))
↪ (2 4 6)
> (infixe (OteTree 'RightMost 5 arbre))
↪ (2 4 6)
> (hauteur (OteTree 'RightMost 5 arbre))
↪ 2
> (LC (OteTree 'RightMost 5 arbre))
↪ 5
> (LCE (OteTree 'RightMost 5 arbre))
↪ 4
> (LCI  (OteTree 'RightMost 5 arbre))
↪ 1
> (exit)
12:29:48 hobbes-mmc-bash
```

◁

Annexe D

Plus d'information sur Prolog

Dans cette partie on trouvera les solutions aux exercices posés dans la section 8.1, ainsi que quelques compléments de syntaxe.

D.1 L'unification PROLOG

Voici l'algorithme d'unification utilisé par PROLOG, il est écrit en pseudo-Pascal. On appelle variable libre une variable non unifiée, une variable libre peut être vue comme un pointeur à NIL. Si une variable n'est pas libre on dit qu'elle est liée, déréférencer une variable voudra dire que l'on parcourt la chaîne des unifications jusqu'à obtenir la valeur de la variable.

```
procedure unifie(t1,t2);

(* t1 et t2 sont deux termes quelconques *)

debut
t3:=dereferencer(t1);t4:=dereferencer(t2);

(* dereferencer est une fonction qui n'agit que sur des vars liees
   elle vaut l'identite sinon *)

si t3 est une variable alors t3 pointe sur t4 (*succes*)
sinon si t4 est une variable alors t4 pointe sur t3 (*succes*)
      sinon si t3 est un atome et t4 un atome alors
                  si t3=t4 alors succes
                  sinon echec
            sinon
                on pose t3=f(t31,..,t3n) et t4=g(t41,..,t4m)
                si f=g alors (* n=m *)
                            pour i:=1 a n faire unifie(t3i,t4i)
                            (* s'il existe un i tq unifie(t3i,t4i)
                               echoue alors unifie(t1,t2) echoue *)
                sinon (* f<>g *)  echec
fin (*unifie*)
```

D.2 Langage

Il arrive parfois que deux clauses dans la définition d'un prédicat ne diffèrent que très peu, il est alors désagréable (et surtout très inefficace de devoir recalculer un certain nombre de littéraux).

EXEMPLE D.2.1

```
...
p :- p1, ..., pj, q1, ... qm.
p :- p1, ..., pj, qq1, ... qql.
...
```

ici les deux définitions ne diffèrent qu'à partir du $j+1$ ème littéral, il serait très agréable de pouvoir factoriser les deux clauses. Ce mécanisme est disponible en PROLOG *grâce à la présence de deux opérateurs -> et ;. Grosso-modo l'opérateur -> peut être vu comme un si-alors sinon, tandis que l'opérateur ; est interprété comme un $\vee$ logique. Pour reprendre le morceau de code précédant on pourra le simplifier en :*

```
...
p :- p1, ..., pj, (condition -> q ; qq).
...

q :- q1, ... qm.
qq :- qq1, ... qql.
```

Où `condition` *est à déterminer en fonction de l'application, voir à titre d'exemple la solution au problème des missionnaires et des cannibales 192.* ◁

D.3 Exercices

Correction de l'exercice 8.2 page 100. Cet exercice a pour but de trouver s'il existe un chemin élémentaire entre deux sommets (puisque l'on veut éviter les problèmes de bouclages, pour cela, nous devons mémoriser les sommets par lesquels nous passons, et vérifier, lorsque nous avons un arc utilisable, si le sommet a déjà été visité :

```
/* a -> b <-> b -> c -> d, a */

arc(a,b).
arc(b,b).
arc(b,c).
arc(c,d).
arc(c,a).

/* version sans mémoire, renvoie vrai ou faux suivant la requête */

path(X,Y) :- arc(X,Y).
path(X,Y) :- arc(X,Z), path(Z,Y).

/* version mémorisant les sommets intermédiaires, insuffisant
 pour éviter les bouclages */

path(X,Y,[]) :- arc(X,Y).
path(X,Y,[Z|L]) :- arc(X,Z), path(Z,Y,L).

/* définition de l'appartenance à une liste */

elem(X,[X|_]).
elem(X,[_|L]) :- elem(X,L).

/* définition de l'inversion avec accumulateur */

rev(L1,L2) :- reverse(L1,[],L2).
reverse([],L,L).
reverse([A|Faire],Fait,Res) :- reverse(Faire,[A|Fait],Res).
```

```
/* définition de la négation */

non(X) :- call(X), !, fail.
non(_).

/* une solution */
path2(X,Y,L) :- path3(X,Y,L,[X]).

path3(X,Y,L,Vu):- arc(X,Y), rev([_|Vu],L).
path3(X,Y,L,Vu):- arc(X,Z), non(elem(Z,Vu)), path3(Z,Y,L,[Z|Vu]).
```

Attaquons nous au problème des missionnaires tel qu'il est défini dans l'exercice 8.4 page 101. Pour mémoire :
Trois missionnaires et trois cannibales se trouvent sur le bord d'une rivière et disposent d'une barque à deux places. Il faut transporter tout le monde sur l'autre rive. La condition à respecter est : « si le nombre de cannibales excède le nombre de missionnaires sur une rive, alors les missionnaires présents sur cette rive sont dévorés ».

Un état sera défini par un triplet (concrétisé par une liste de 3 éléments) indiquant respectivement le nombre de missionnaires, le nombre de cannibales qu'il y a sur la rive où se trouve la barque.
`ga` sera l'abréviation pour « rive gauche »et `dr` correspondra à « rive droite ». Ainsi `[2,2,ga]` signifie que deux missionaires et deux cannibales sont sur la rive gauche avec la barque.

Une étape de calcul liera deux états consécutifs viables, cette relation sera définie en fonction du nombre d'individus qu'il y a sur la rive considérée et du nombre de cannibales et de missionnaires transférés d'une rive à l'autre.
`traverse` : est la relation qui, à partir du nombre de cannibales et du nombre de missionnaires, va fournir le nombre et la catégorie des personnes transférées.
`barque` : est la contrainte sur la capacité de transfert de la barque.
`viable` : permet de savoir si pour une rive la situation est viable ou non.
`essai` : va chercher à rajouter une nouvelle étape si on n'est pas dans l'état final caractérisé par `[3,3,ga]`.
`canib_missio` : est le prédicat initial par lequel on accède au programme.

Une première version du programme pourrait être :

```
canib_missio(L):-X=[3,3,dr],essai(X,[X],L0),inv(L0,L).

essai([3,3,ga],L,L):-!.
essai(X,Li,Lf):-etape(X,Y),non(member(Y,Li)),
                essai(Y,[Y|Li],Lf).

etape([Mi,Ci,ga],[Mf,Cf,dr]):-traverse(Mi,Ci,M,C),Mf is 3-(Mi-M),
                              Cf is 3-(Ci-C),viable(Mf,Cf).
etape([Mi,Ci,dr],[Mf,Cf,ga]):-traverse(Mi,Ci,M,C),Mf is 3-(Mi-M),
                              Cf is 3-(Ci-C),viable(Mf,Cf).

traverse(M1,C1,M2,C2):-barque(M2,C2),M1>=M2,C1>=C2,M is M1-M2,C is C1-C2,
                       viable(M,C).
barque(2,0).
barque(1,0).
barque(1,1).
barque(0,1).
barque(0,2).

viable(0,C):-!.
viable(M,C):-C=<M.

member(X,[X|_]).
```

```
member(X,[_|L]):-member(X,L).

inv(L0,L1):-inv(L0,[],L1).
inv([],L,L).
inv([X|L],L1,L2):-inv(L,[X|L1],L2).
```

Version équivalente faisant appel au prédicat ->, qui correspond à un *si-alors-sinon*.

```
init(L):- X=[3,3,dr],essai(X,[X],L0),inv(L0,L).

essai(X,Li,Lf):- etape(X,Y),non(member(Y,Li)),
                 (Y=[3,3,ga]->Lf=[Y|Li];essai(Y,[Y|Li],Lf)).

etape([Mi,Ci,D],[Mf,Cf,A]):- (D=dr->A=ga;A=dr),
                             traverse(Mi,Ci,M,C),Mf is 3-(Mi-M),
                             Cf is 3-(Ci-C),viable(Mf,Cf).

traverse(M1,C1,M2,C2):- barque(M2,C2),M1>=M2,C1>=C2,M is M1-M2,C is C1-C2,
                        viable(M,C).
barque(2,0).
barque(1,0).
barque(1,1).
barque(0,1).
barque(0,2).

viable(0,C):-!.
viable(M,C):-C=<M.

member(X,[X|_]).
member(X,[_|L]):-member(X,L).

inv(L0,L1):-inv(L0,[],L1).
inv([],L,L).
inv([X|L],L1,L2):-inv(L,[X|L1],L2).
```

Annexe E

Algorithmes génétiques

Dans cette partie nous donnons le listing complet de deux utilisations d'algorithmes génétiques. Le premier écrit en NetLogo, décrit une tentative de faire évoluer une population d'animats dans un environnement plus ou moins agressif. Le lecteur pourra se reporter à la section 13.5 du chapitre 13 pour des commentaires sur certaines parties de l'implémentation, l'implémentation de NetLogo que nous avons utilisée est la 1.0 beta 9, le site de référence est à l'adresse suivante : `ccl.northwestern.edu/netlogo/`. Le second, écrit en SCHEME tourne sous `guile` est cherche à résoudre la position optimale de 32 interrupteurs pour la luminosité maximale d'une ampoule. Là encore, nous renvoyons le lecteur à la section 13.6 du même chapitre.

E.1 Evolution d'un animat en *Logo

La version ci-après n'est pas forcément la plus efficace pour implémenter un algorithme génétique. Elle n'est fournie qu'à titre d'exemple, ayant été développée en 48h ell est grandement modifiable.

```
; add model procedures here

turtles-own [genome age lgoals speed vision energy score
             high dist lmines speed-clock]
patches-own [food smell aGoal aMine clock]
breeds [dead alive]
globals [tick nbactions nbstate pool-genome pool-fitness
         min-fitness max-fitness moy-fitness med-fitness
         list-actions score-pool
         c-turtle c-goal c-mine c-food c-smell]

to setup
ca
init-main-variables
;; variables automates
set nbstate 64
set nbactions 16
;; initialisation
init-pool-genome
;; les courbes
start-plot
end

to init-main-variables
;; variables genetiques
set min-fitness 0
set max-fitness 0
set moy-fitness 0
```

```
set med-fitness 0
set score-pool  0
;; horloge
set tick 0
; le pool genetique
set pool-genome []
;; customization des couleurs
set c-mine   red
set c-goal   sky
set c-turtle white
set c-food   green
set c-smell  lime
end

; nbstate 0 dans une liste
to-report raz-la
locals [l]
set l []
repeat nbstate [set l lput 0 l]
report l
end

to start-plot
set-current-plot "Evolution"
auto-plot-on
set-plot-x-range  0 100
set-plot-y-range  0 iter

set-plot-pen "min"
set-plot-pen-color black
set-plot-pen "max"
set-plot-pen-color red
set-plot-pen "moy"
set-plot-pen-color green
set-plot-pen "next"
set-plot-pen-color sky

set-current-plot "MineGoal"
auto-plot-on
set-plot-x-range 0 100
set-plot-y-range 0 1

set-plot-pen "mine"
set-plot-pen-color red
set-plot-pen "maxm"
set-plot-pen-color red
set-plot-pen "goal"
set-plot-pen-color sky
set-plot-pen "maxg"
set-plot-pen-color sky

set-current-plot "AgeFood"
auto-plot-on
set-plot-x-range 0 100
set-plot-y-range 0 iter

set-plot-pen "age"
set-plot-pen-color violet
set-plot-pen "food"
```

```
set-plot-pen-color green
end

to do-plot [x]
set-current-plot x
if x = "Evolution" [
set-plot-pen "min"
plot (min-fitness / 10)
set-plot-pen "max"
plot (max-fitness / 10)
set-plot-pen "moy"
plot (moy-fitness / 10)
set-plot-pen "next"
plot (score-pool / 10)
]
if x = "MineGoal" [
set-plot-pen "mine"
plot stat-mines turtles
set-plot-pen "maxm"
plot mine-avoider
set-plot-pen "goal"
plot stat-goals turtles
set-plot-pen "maxg"
plot goal-catcher
]
if x = "AgeFood" [
set-plot-pen "age"
plot stat-age turtles
set-plot-pen "food"
plot stat-food
]
end

to set-world
ct ; kill old ladies
create-turtles population ; les nouvelles
cp ; clear patches
init-patches
init-turtles
paint-patches
; reset timer
set tick 0
set list-actions raz-la
end

; le programme principal
to go
set-world
run-one-generation
ifelse count dead mod 2 = 0
 [show "S-A" select-A][show "S-B" select-B]
do-plot "Evolution"
do-plot "MineGoal"
do-plot "AgeFood"
end

to run-one-generation
locals [i]
set i 1
```

```
while [i <= iter and count alive > 0][
   set tick tick + 1 ; l'age reel du systeme
   update-patches
   repeat 4 [ diffuse smell 0.8 ]
   paint-patches
   ask alive [set age age + 1
              move ]
   ask alive [ifelse hidden?
                  [set breed dead]
                  [age-effect]
         shapes-looking
             ]
   set i i + 1
]
ask alive [set score fitness-value]
make-pool-fitness
; show list-actions
show compute-list-actions-used
export-output "sortie.txt"
end

to update-patches
ask patches [
   ifelse aMine > 0
    [set smell 0]
    [set smell sqrt smell]
   if (clock = update and food > 0) [set food food - 1 set clock -1]
   if food > 0 [set clock clock + 1 set smell 10 * food]
]
if tick mod update = 0 [ add-food ]
end

; init-turtles
to init-turtles
ask turtles [
   set breed alive
   set energy maxenergy
   set age 0
   set speed-clock 0
   set speed floor (maxspeed / 2)
   set dist 0
   set lmines []
   set lgoals []
   set high 0
   set vision 1
   set genome item who pool-genome
   set color c-turtle
   set heading (random 360)
   setxy (random screen-size-x) (random screen-size-y)
   while [aMine-of patch-at 0 0 > 0 or
          aGoal-of patch-at 0 0 > 0 ][fd 1 rt (random 45)]
]
end

to shapes-looking
locals [sz]
set lgoals (remove-duplicates lgoals)
set sz length lgoals
if sz = 1 [set shape "circle" stop]
```

```
if sz = 2 [set shape "ant"    stop]
if sz = 3 [set shape "turtle" stop]
if sz = 4 [set shape "bee"    stop]
if sz > 4 [set shape "person" stop]
end

; init-patches
to init-patches
locals [i]
ask patches [
   set aGoal 0
   set aMine 0
   set clock 0
   set food 0
   set smell 0
]

; set goals
set i 0
while [i < buts] [
   ask (random-one-of patches) [
      if aGoal = 0 [set i i + 1] ; found one
      set aGoal 1 ]
]

; set mines
set i 0
while [i < mines] [
   ask (random-one-of patches) [
      if not (aGoal > 0 or aMine > 0) [
         set i i + 1 ; found one
         set aMine 1]
      ]
]

; set food
add-food
end

to add-food
locals [i]

set i 0
while [i < amount][
   ask (random-one-of patches)[
      if aMine = 0 [
         set i i + 1 ; found one
         set food food + 1
         set clock 0
         set smell 10 * food
      ]
   ]
]
end

to paint-patches
ask patches [
   ifelse aGoal > 0
     [set pcolor c-goal]
```

```
     [ifelse aMine > 0
         [set pcolor c-mine]
         [ifelse food > 0
              [set pcolor c-food]
              [scale-pcolor c-smell smell 0.1 10]
         ]
     ]
  ]
end

;; specific algorithme genetique

to init-pool-genome
repeat population [
 set pool-genome lput create-random-genome pool-genome
 ]
end

to-report create-random-genome
locals [g o]
set g []
repeat nbstate [
   set o []
   repeat (1 + random actions) [ set o lput (random nbactions) o ]
   set g lput o g
]
report g
end

; on privilegie le nombre de buts reperes
; la longevite
; la distance parcourue
; le niveau energetique eleve
; on deprecie la rencontre de mines
to-report fitness-value
locals [s lg lm]
set lg remove-duplicates lgoals ; supprime les doubles
set lm remove-duplicates lmines ; supprime les doubles
set s 1 + (length lg)
set s  (s * age + sqrt dist  + high) / (1 + length lm + speed-clock)
report s
end

to make-pool-fitness
locals [i l x]
set pool-fitness []
set i 0
set min-fitness score-of (turtle i)
set max-fitness score-of (turtle i)
while [i < population][
 set x score-of (turtle i)
 set pool-fitness lput x pool-fitness
 if min-fitness < x [set min-fitness x]
 if max-fitness > x [set max-fitness x]
 set i i + 1
]
; recadrage des scores
; on assure un min a 1
set min-fitness min-fitness + 1
```

```
set i 0
while [i < population][
 set x score-of (turtle i) - min-fitness
 set x x * x
 set pool-fitness replace-item i pool-fitness x
 set i i + 1
]
set max-fitness max pool-fitness
set min-fitness min pool-fitness
set moy-fitness mean pool-fitness
set med-fitness median pool-fitness
end

to select-A
locals [x sigma lst y toKeep laliste]
set x 0
set sigma sum pool-fitness
set toKeep []
set score-pool 0
repeat (population / 2)[
 set x (random sigma) ; the winner
 if min-fitness < 0 [set x x + min-fitness]
 set lst -1 ; the winner index
 set y sigma ; partial sum
 while [y > x][
   set y y - (item (lst + 1) pool-fitness)
   set lst lst + 1
 ]
 ; when we leave lst is the true winner
 set toKeep lput lst toKeep
 set score-pool score-pool + (item lst pool-fitness)
]
set laliste []
set x 0
set y population / 2
while [x < y] [set laliste lput x laliste
               set x x + 1]
set score-pool score-pool / y
set pool-genome (mutation (generate-pool toKeep laliste []))
end

; on regarde chaque item
; on calcule la partie entiere de f_i / f
; on a le nombre de representants dans le pool
to select-B
locals [sigma best best-i i j k toKeep toTreat sz x laliste]
set best-i -1
set best   -1
set sigma sum pool-fitness
set toKeep []
set sz 0
set toTreat []
set i 0
set score-pool 0
while [i < population and sz < (population / 2)][
 set x (item i pool-fitness)
 if x > best [set best-i i set best x]
 set j (x / sigma)
 ifelse x >= 0 [set k floor j][set k 0]
```

```
 repeat j [set toKeep lput i toKeep
           set score-pool score-pool + x
           set sz sz + 1 ]
 set x (random 1.0)
 if k != 0 [set k j - k]
 if (j < 1 or x <= k) [set toTreat lput i toTreat]
 set i i + 1
]
set j (population / 2 ) - (length toKeep)
repeat j [ set i (first toTreat)
       set toKeep lput i toKeep
           set score-pool score-pool + (item i pool-fitness)
           set toTreat (bf toTreat) ]

show word best word " " (length remove-duplicates lgoals-of (turtle best-i))
show genome-of (turtle best-i)
set laliste []
set i 0
set j population / 2
while [i < j] [set laliste lput i laliste
               set i i + 1]
set score-pool score-pool / j
set pool-genome (mutation (generate-pool toKeep laliste []))
end

; ilist list of turtles index
; llist index of not used turtles
; sol the answer
; generate-pool is recursive terminal
to-report generate-pool [ilist llist sol]
locals [i xi j xj]
ifelse not empty? llist [
 set i (item (random (length llist)) llist)
 set xi (item i  ilist)
 set llist remove i llist
 set j (item (random (length llist)) llist)
 set xj (item j ilist)
 set llist remove j llist
; show word i word " " word j word " " (length llist)
 report generate-pool ilist llist (cross-over xi xj sol)
 ]
 [ report sol ]
end

to-report cross-over [i j pool]
locals [g1 g2 g12 g21 cut k]

set g1 (item i pool-genome)
set g2 (item j pool-genome)

set cut (1 + random (nbstate - 1))
set k 0
set g12 []
set g21 []
while [k < cut][
 set g12 lput (item k g1) g12
 set g21 lput (item k g2) g21
  set k k + 1
]
```

```
while [k < nbstate][
 set g12 lput (item k g2) g12
 set g21 lput (item k g1) g21
 set k k + 1
]
report (fput g1 (fput g2 (fput g12 (fput g21 pool))))
end


to-report mutation [pool]
locals [g s x i j k]
repeat mutations [
  set i (random population) ; population genomes
  set g (item i pool-genome)
  ; show word "avant : " g
  set j (random nbstate) ; nbstate possibilities
  set s (item j g)
  ; type word " avant " s
  set k random (length s) ; at most length s actions possibles
  set x (item k s + 1 + random (nbactions - 1)) mod nbactions
  set s replace-item k s x
  ; show word " apres " s
  set g replace-item j g s
  ; show word "apres : " g
  set pool-genome replace-item i pool-genome g
] ; fin mutations
report pool
end

;; Savoir si de la nourriture est accessible
to-report food-accessible
locals [fp]
set fp 0
ask patches in-radius speed [
   if food > 0 [set fp fp + 1]
]
report fp
end

to-report food-visible
locals [fp]
ask patches in-radius vision [
  if food > 0 [set fp fp + 1]
]
report fp
end

to-report isFAV?
 report ((food-accessible > 0) and (food-visible > 0))
end

;; Savoir si un but est dans un certain rayon
to-report goal-accessible
 locals [gp]
 set gp 0
 ask patches in-radius speed [
    if aGoal > 0 [set gp gp + 1]
 ]
; if aGoal-of patch-at 0 0 > 0 [set gp gp - 1]
```

```
 report gp
end

to-report goal-visible
 locals [gp]
 set gp 0
 ask patches  in-radius vision [
    if aGoal > 0 [set gp gp + 1]
 ]
 if aGoal-of patch-at 0 0 > 0 [set gp gp - 1]
 report gp
end

to-report isGV?
 report (goal-visible > 0)
end

to-report isGA?
 ifelse isGV?
  [report (goal-accessible > 0)]
  [report false]

end

;; presence d'une mine
to-report mine-accessible
 locals [mp]
 set mp 0
 ask patches in-radius speed [
    if aMine > 0 [set mp mp + 1]
 ]
 if aMine-of patch-at 0 0 > 0 [set mp mp - 1]
 report mp
end

to eval-prox
ask patches in-radius speed [set pcolor magenta]
end

to-report mine-visible
 locals [mp]
 set mp 0
 ask patches in-radius vision [
    if aMine > 0 [set mp mp + 1]
 ]
; if aMine-of patch-at 0 0 > 0 [set mp mp - 1]
 report mp
end

to-report isMV?
 report (mine-visible > 0)
end

to-report isMA?
 ifelse isMV?
  [report (mine-accessible > 0)]
  [report false]
end
```

```
;; the actions for the turtles

to move
locals [x y]
 set x findState
 set y (item x list-actions)
 set list-actions (replace-item x list-actions (y + 1))
 performActionSet (item x genome)
 repeat speed [
    ifelse (not hidden?) [eat bouge]
    [stop]
 ]
 ifelse speed = 0
  [ ; on n'a pas fait bouge
    set speed-clock speed-clock + 1
    eat
    evalue-sante
  ]
  [set speed-clock 0]
end

to bouge
; les mines et goal ont un effet sur l'age
; il y a veillissement tous les update
; donc on calcule age mod update
; on l'enleve a l'age et on rajoute update pour
; goal update - 1 pour mine

locals [year]
set year (age mod update)
if aMine-of (patch-at dx dy) > 0
   [ set lmines fput (patch-at dx dy) lmines
     set age age - year - 1 ]
if aGoal-of (patch-at dx dy) > 0
    [set lgoals fput (patch-at dx dy) lgoals
     set age age + update - year ]

; deplacement
fd 1
set dist dist + 1
lost-energy
evalue-sante
end

to evalue-sante
; if speed-clock > age / 2 [ age-effect ]
if energy < vieillir
   [ht set score fitness-value]
if energy > maxenergy
   [set energy maxenergy]
end

to eat
locals [f]
set f food-of patch-at 0 0
if f > 0 [
  ask patch-at 0 0 [ set food f - 1
                     set smell 10 * food ]
```

```
  set high high + 1
  gain-energy ]
end

; l'effet de l'age sur l'energie
to age-effect
if age mod update = 0
 [set energy energy * (1 - vieillir)]
end

to gain-energy
 set energy energy + gain
end

to lost-energy
 set energy energy * (1 - lost)
end

to-report findState
locals [s]
set s 0
 if isFAV?                      [set s s + 1 ]
 if energy > level              [set s s + 2 ]
 if aGoal-of patch-at 0 0 > 0   [set s s + 4 ]
 if isGV?                       [set s s + 8 ]
 if aMine-of patch-at 0 0 > 0   [set s s + 16]
 if isMV?                       [set s s + 32]
report s
end

to-report compute-list-actions-used
locals [l i]
set l []
set i 0
while [ i < nbstate ][
 if (item i list-actions) != 0 [
  set l lput i l
 ]
 set i i + 1
]
report l
end

to performActionSet [vlist]
 if empty? vlist [stop]
 performOneAction (first vlist)
 performActionSet (bf vlist)
end

to performOneAction [val]
if val = 0 [nope stop]
if val = 1 [speedup stop]
if val = 2 [slowdown stop]
if val = 3 [resetspeed stop]
if val = 4 [head-food stop]
if val = 5 [random-heading stop]
if val = 6 [half-turn stop]
if val = 7 [head-towards-friends stop]
if val = 8 [head-towards-mine stop]
```

```
if val = 9 [head-towards-smell stop]
if val = 10 [head-towards-goal stop]
if val = 11 [increase-vision stop]
if val = 12 [decrease-vision stop]
if val = 13 [run-away-goal stop]
if val = 14 [run-away-mine stop]
if val = 15 [run-away-friends stop]
end

;; les actions possibles

; 0
to nope
end

; 1
to speedup
if speed < maxspeed [set speed speed + 1]
end

; 2
to slowdown
if speed > 0 [set speed speed - 1]
end

; 3
to resetspeed
set speed 0
end

; 4
to head-food
 set heading towards max-one-of (patches in-radius vision) [food]
end

; 5
to random-heading
 set heading (random 360)
end

; 6
to half-turn
if speed <= 1
 [set heading heading + 180]
end

; 7
to head-towards-friends
 set heading towards max-one-of (patches in-radius vision) [count alive-here]
end

; 8
to head-towards-mine
set heading towards max-one-of (patches in-radius vision) [aMine]
end

; 9
to head-towards-smell
 set heading uphill smell
```

```
end

; 10
to head-towards-goal
set heading towards max-one-of (patches in-radius vision) [aGoal]
end

; 11
to increase-vision
if vision < visibilite [set vision vision + 1]
end

; 12
to decrease-vision
if vision > 0 [set vision vision - 1]
end

; 13
to run-away-goal
head-towards-goal
half-turn
end

; 14
to run-away-mine
head-towards-mine
half-turn
end

; 15
to run-away-friends
head-towards-friends
half-turn
end

;; un peu de stats
to-report stat-goals [b]
 report mean values-from b [length (remove-duplicates lgoals)]
end

to-report stat-mines [b]
 report mean values-from b [length (remove-duplicates lmines)]
end

to-report stat-age [b]
 report mean values-from b [age]
end

to-report stat-food
 report max values-from turtles [high]
end

to-report goal-catcher
 report max values-from turtles [length (remove-duplicates lgoals)]
end

to-report mine-avoider
 report max values-from turtles [length (remove-duplicates lmines)]
end
```

```
to-report stat-vision [b]
 report mean values-from b [vision]
end

to-report stat-speed [b]
 report mean values-from b [speed]
end

@#$#@#$#@
CC-WINDOW
321
326
636
446
Command Center

MONITOR
10
410
60
459
aTurt
stat-age turtles
1
1

MONITOR
70
410
120
459
vAlive
stat-vision alive
2
1

MONITOR
130
410
180
459
sAlive
stat-speed alive
2
1

MONITOR
10
360
60
409
gAlive
stat-goals alive
2
1

MONITOR
70
```

```
360
120
409
gDead
stat-goals dead
2
1

MONITOR
130
360
180
409
aDead
stat-age dead
2
1

MONITOR
190
360
240
409
Food
stat-food
2
1

SLIDER
10
20
130
53
population
population
0
400
100
4
1

SLIDER
10
60
130
93
maxspeed
maxspeed
0
10
5
1
1

SLIDER
140
60
240
93
```

```
vieillir
vieillir
0
1
0.30000000000000004
0.05
1

SLIDER
10
100
130
133
maxenergy
maxenergy
0
10
10
1
1

SLIDER
140
100
240
133
level
level
0
10
1
0.1
1

SLIDER
10
150
130
183
mines
mines
1
100
60
1
1

SLIDER
140
150
240
183
iter
iter
0
100
50
1
1
```

```
SLIDER
10
200
130
233
buts
buts
1
20
10
1
1

SLIDER
140
200
240
233
actions
actions
0
10
3
1
1

SLIDER
10
250
130
283
gain
gain
0
2
1.0
0.1
1

SLIDER
140
250
240
283
lost
lost
0
2
0.2
0.1
1

SLIDER
10
300
130
333
update
```

```
update
0
10
5
1
1

SLIDER
140
300
240
333
amount
amount
0
200
60
1
1

SLIDER
660
340
760
373
mutations
mutations
0
5
3
1
1

BUTTON
162
20
217
53
start
setup
NIL
1
T
OBSERVER

BUTTON
249
20
304
53
run
repeat 1000 [go]
NIL
1
T
OBSERVER

MONITOR
660
```

```
30
730
79
lowest
min-fitness
2
1

MONITOR
740
30
800
79
best
max-fitness / 10
2
1

MONITOR
660
100
730
149
moyen
moy-fitness / 10
2
1

MONITOR
740
100
800
149
median
med-fitness / 10
2
1

MONITOR
770
400
830
449
pool
score-pool / 10
2
1

SLIDER
660
400
760
433
visibilite
visibilite
0
10
5
1
```

```
1

MONITOR
660
170
730
219
goals
goal-catcher
0
1

MONITOR
740
170
800
219
mines
mine-avoider
0
1

PLOT
10
470
360
600
Evolution
temps
score
0.0
1235.0
0.0
2362.0

PLOT
370
470
660
600
MineGoal
temps
score
0.0
1235.0
0.0
7.0

PLOT
670
470
960
600
AgeFood
temps
score
0.0
1235.0
0.0
```

```
50.0

GRAPHICS-WINDOW
321
10
636
325
17
17
9.0
1

@#$#@#$#@
add model documentation here
@#$#@#$#@
default
true
0
Polygon -7566196 true true 150 5 40 250 150 205 260 250

arrow
true
0
Polygon -7566196 true true 150 0 0 150 105 150 105 293 195 293 195 150 300 150

box
true
0
Polygon -7566196 true true 45 255 255 255 255 45 45 45

spacecraft
true
0
Polygon -7566196 true true 150 0 180 135 255 255 225 240 150 180 75 240 45 255 120 135

thin-arrow
true
0
Polygon -7566196 true true 150 0 0 150 120 150 120 293 180 293 180 150 300 150

turtle
true
0
Polygon -7566196 true true 138 75 162 75 165 105 225 105 225 142 195 135 195 187 225 195 225 225 195 217 195

person
false
0
Circle -7566196 true true 155 20 63
Rectangle -7566196 true true 158 79 217 164
Polygon -7566196 true true 158 81 110 129 131 143 158 109 165 110
Polygon -7566196 true true 216 83 267 123 248 143 215 107
Polygon -7566196 true true 167 163 145 234 183 234 183 163
Polygon -7566196 true true 195 163 195 233 227 233 206 159

truck-down
false
0
Polygon -7566196 true true 225 30 225 270 120 270 105 210 60 180 45 30 105 60 105 30
```

```
Polygon -8716033 true false 195 75 195 120 240 120 240 75
Polygon -8716033 true false 195 225 195 180 240 180 240 225

truck-right
false
0
Polygon -7566196 true true 180 135 75 135 75 210 225 210 225 165 195 165
Polygon -8716033 true false 210 210 195 225 180 210
Polygon -8716033 true false 120 210 105 225 90 210

truck-left
false
0
Polygon -7566196 true true 120 135 225 135 225 210 75 210 75 165 105 165
Polygon -8716033 true false 90 210 105 225 120 210
Polygon -8716033 true false 180 210 195 225 210 210

circle
false
0
Circle -7566196 true true 34 34 230

butterfly1
true
0
Polygon -16777216 true false 151 76 138 91 138 284 150 296 162 286 162 91
Polygon -7566196 true true 164 106 184 79 205 61 236 48 259 53 279 86 287 119 289 158 278 177 256 182 164 181
Polygon -7566196 true true 136 110 119 82 110 71 85 61 59 48 36 56 17 88 6 115 2 147 15 178 134 178
Polygon -7566196 true true 46 181 28 227 50 255 77 273 112 283 135 274 135 180
Polygon -7566196 true true 165 185 254 184 272 224 255 251 236 267 191 283 164 276
Line -7566196 true 167 47 159 82
Line -7566196 true 136 47 145 81
Circle -7566196 true true 165 45 8
Circle -7566196 true true 134 45 6
Circle -7566196 true true 133 44 7
Circle -7566196 true true 133 43 8

boat1
false
0
Polygon -1 true false 63 162 90 207 223 207 290 162
Rectangle -6524078 true false 150 32 157 162
Polygon -16776961 true false 150 34 131 49 145 47 147 48 149 49
Polygon -7566196 true true 158 33 230 157 182 150 169 151 157 156
Polygon -7566196 true true 149 55 88 143 103 139 111 136 117 139 126 145 130 147 139 147 146 146 149 55

boat2
false
0
Polygon -1 true false 63 162 90 207 223 207 290 162
Rectangle -6524078 true false 150 32 157 162
Polygon -16776961 true false 150 34 131 49 145 47 147 48 149 49
Polygon -7566196 true true 157 54 175 79 174 96 185 102 178 112 194 124 196 131 190 139 192 146 211 151 216 1
Polygon -7566196 true true 150 74 146 91 139 99 143 114 141 123 137 126 131 129 132 139 142 136 126 142 119 1

boat3
false
0
Polygon -1 true false 63 162 90 207 223 207 290 162
```

```
Rectangle -6524078 true false 150 32 157 162
Polygon -16776961 true false 150 34 131 49 145 47 147 48 149 49
Polygon -7566196 true true 158 37 172 45 188 59 202 79 217 109 220 130 218 147 204 156 158 156 161 142 170 12
Polygon -7566196 true true 149 66 142 78 139 96 141 111 146 139 148 147 110 147 113 131 118 106 126 71

bee
true
0
Polygon -256 true false 151 152 137 77 105 67 89 67 66 74 48 85 36 100 24 116 14 134 0 151 15 167 22 182 40 2
Polygon -16777216 true false 151 150 149 128 149 114 155 98 178 80 197 80 217 81 233 95 242 117 246 141 247 1
Polygon -7566196 true true 246 151 241 119 240 96 250 81 261 78 275 87 282 103 277 115 287 121 299 150 286 18
Polygon -16777216 true false 115 70 129 74 128 223 114 224
Polygon -16777216 true false 89 67 74 71 74 224 89 225 89 67
Polygon -16777216 true false 43 91 31 106 31 195 45 211
Line -1 false 200 144 213 70
Line -1 false 213 70 213 45
Line -1 false 214 45 203 26
Line -1 false 204 26 185 22
Line -1 false 185 22 170 25
Line -1 false 169 26 159 37
Line -1 false 159 37 156 55
Line -1 false 157 55 199 143
Line -1 false 200 141 162 227
Line -1 false 162 227 163 241
Line -1 false 163 241 171 249
Line -1 false 171 249 190 254
Line -1 false 192 253 203 248
Line -1 false 205 249 218 235
Line -1 false 218 235 200 144

wolf-left
false
3
Polygon -6524078 true true 117 97 91 74 66 74 60 85 36 85 38 92 44 97 62 97 81 117 84 134 92 147 109 152 136
Polygon -6524078 true true 87 80 79 55 76 79
Polygon -6524078 true true 81 75 70 58 73 82
Polygon -6524078 true true 99 131 76 152 76 163 96 182 104 182 109 173 102 167 99 173 87 159 104 140
Polygon -6524078 true true 107 138 107 186 98 190 99 196 112 196 115 190
Polygon -6524078 true true 116 140 114 189 105 137
Rectangle -6524078 true true 109 150 114 192
Rectangle -6524078 true true 111 143 116 191
Polygon -6524078 true true 168 106 184 98 205 98 218 115 218 137 186 164 196 176 195 194 178 195 178 183 188
Polygon -6524078 true true 207 140 200 163 206 175 207 192 193 189 192 177 198 176 185 150
Polygon -6524078 true true 214 134 203 168 192 148
Polygon -6524078 true true 204 151 203 176 193 148
Polygon -6524078 true true 207 103 221 98 236 101 243 115 243 128 256 142 239 143 233 133 225 115 214 114

wolf-right
false
3
Polygon -6524078 true true 170 127 200 93 231 93 237 103 262 103 261 113 253 119 231 119 215 143 213 160 208
Polygon -6524078 true true 201 99 214 69 215 99
Polygon -6524078 true true 207 98 223 71 220 101
Polygon -6524078 true true 184 172 189 234 203 238 203 246 187 247 180 239 171 180
Polygon -6524078 true true 197 174 204 220 218 224 219 234 201 232 195 225 179 179
Polygon -6524078 true true 78 167 95 187 95 208 79 220 92 234 98 235 100 249 81 246 76 241 61 212 65 195 52 1
Polygon -6524078 true true 48 143 58 141
Polygon -6524078 true true 46 136 68 137
Polygon -6524078 true true 46 130
```

```
Polygon -6524078 true true 45 129 35 142 37 159 53 192 47 210 62 238 80 237
Line -16777216 false 74 237 59 213
Line -16777216 false 59 213 59 212
Line -16777216 false 58 211 67 192
Polygon -6524078 true true 38 138 66 149
Polygon -6524078 true true 46 128 33 120 21 118 11 123 3 138 5 160 13 178 9 192 0 199 20 196 25 179 24 161 25
Polygon -6524078 true true 67 122 96 126 63 144

ant
true
0
Polygon -7566196 true true 136 61 129 46 144 30 119 45 124 60 114 82 97 37 132 10 93 36 111 84 127 105 172 10
Polygon -7566196 true true 150 95 135 103 139 117 125 149 137 180 135 196 150 204 166 195 161 180 174 150 158
Polygon -7566196 true true 149 186 128 197 114 232 134 270 149 282 166 270 185 232 171 195 149 186 149 186
Polygon -7566196 true true 225 66 230 107 159 122 161 127 234 111 236 106
Polygon -7566196 true true 78 58 99 116 139 123 137 128 95 119
Polygon -7566196 true true 48 103 90 147 129 147 130 151 86 151
Polygon -7566196 true true 65 224 92 171 134 160 135 164 95 175
Polygon -7566196 true true 235 222 210 170 163 162 161 166 208 174
Polygon -7566196 true true 249 107 211 147 168 147 168 150 213 150
Polygon -7566196 true true 270 14
Polygon -7566196 true true 276 21

bird1
false
0
Polygon -7566196 true true 2 6 2 39 270 298 297 298 299 271 187 160 279 75 276 22 100 67 31 0

bird2
false
0
Polygon -7566196 true true 2 4 33 4 298 270 298 298 272 298 155 184 117 289 61 295 61 105 0 43

@#$#@#$#@
NetLogo 1.0 Beta 9
@#$#@#$#@
@#$#@#$#@
@#$#@#$#@
```

E.2 Un problème d'interrupteurs en SCHEME

Cet exemple permet de se faire une idée d'une implémentation possible en SCHEME. Elle a été développée avec une visée pédagogique (enfin j'espère), et n'est donc pas exempte de simplifications ou d'optimisations.

```
;;;;;;; Description du probleme ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                  ;;
;; On dispose de 32 interrupteurs (on/off) qui influencent          ;;
;; une ampoule (une sorte de variateur), on cherche la config       ;;
;; des interrupteurs fournissant le maximum de luminosite           ;;
;; la luminosite est un nombre compris entre 0 et 100               ;;
;; une seule configuration donne la valeur 100                      ;;
;; a toute configuration on a une reponse                           ;;
;; une reponse peut etre obtenue par plusieurs configurations       ;;
;; Il y a 2^32 configurations possibles soit 4 294 967 296          ;;
;;                                                                  ;;
;; Une configuration est un vecteur de 32 valeurs                   ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;; Les variables globales sont declarees
;; l'initialisation est faite par setup (fin du fichier)
(define laSolution -1)
(define secret -1)
(define chromoSz -1)

;; La fonction fitness a un vecteur renvoie sa valeur
;; il s'agit du
;; nombre de bits communs a 0 + 2 * nombre commun a 1
;; exple (0 1 0 1) vs (1 1 1 1) = 4
;; exple (0 1 0 1) vs (1 0 0 1) = 3

(define (true-fitness v)

  (define (aux i score)
    (if (< i chromoSz)
(let* ((a  (vector-ref v i))
       (b  (vector-ref laSolution i))
       (x (if (= a b) 1 0)))
(aux (+ i 1) (+ score x (* a b)) )) score))

  (aux 0 0)
)

;; on normalise sur 100
;; secret = chromoSz + # 1
;; exple la soluce est (0 1 0 1) -> secret = 4 + 2
(define (eval-fitness v)
  (/ (* (true-fitness v) 100) secret))

;; creation aleatoire d'un chromosome
(define (build-chromosome-alea)

  (define (aux i soluce)
    (if (< i chromoSz)
(aux (+ i 1) (cons (random 2) soluce))
(list->vector soluce))
    )
 (aux 0 '())
)

;; renvoie la version mutee du vecteur
(define (mutation v)

  (let ( (p (random (length (vector->list v)))) )
    (vector-set! v p (modulo (+ 1 (vector-ref v p)) 2))
    v
    ))

;; renvoie le resultat du croisement de deux vecteurs
;; on suppose que les deux vecteurs sont de la taille fournie (sz)
(define (simple-cross-over v1 v2 sz)
 (let ( (p (+ 1 (random (- sz 1)))) )
   (do (( w1 (make-vector sz))
( w2 (make-vector sz))
( i 0 (+ i 1))) ; iterateur
       ((= i sz) (list w1 w2) ) ; condition d'arret
     (if (< i p)
```

```
 (begin (vector-set! w1 i (vector-ref v1 i))
(vector-set! w2 i (vector-ref v2 i)) )
 (begin (vector-set! w1 i (vector-ref v2 i))
(vector-set! w2 i (vector-ref v1 i)) )
 )
     )
   )
)

;; renvoie le resultat du croisement de deux vecteurs
;; on suppose que les deux vecteurs sont de la taille fournie (sz)
(define (double-cross-over v1 v2 sz)
 (let ( (p (+ 1 (random (- (floor (/ sz 2)) 1))))
(q (- sz  (random (- (floor (/ sz 2)) 1)))) )
   (do (( w1 (make-vector sz))
( w2 (make-vector sz))
( i 0 (+ i 1))) ; iterateur
       ((= i sz) (list w1 w2) ) ; condition d'arret
     (if (or (< i p) (> i q))
 (begin (vector-set! w1 i (vector-ref v1 i))
(vector-set! w2 i (vector-ref v2 i)) )
 (begin (vector-set! w1 i (vector-ref v2 i))
(vector-set! w2 i (vector-ref v1 i)) )
 )
     )
   )
)

;; vi (ch #(s s))
;; @return (w1 w2)
(define (cross-over type v1 v2)

  (let* ( (x (if (= type 1)
 (simple-cross-over (car v1)
    (car v2) chromoSz)
 (double-cross-over (car v1)
    (car v2) chromoSz)
 ) )
  (v (car x)) (w (cadr x))
  (sv (eval-fitness v)) (sw (eval-fitness w)) )
    (list  (cons v (vector sv sv)) (cons w (vector sw sv)) ) )
)

;; type (1 ou 2) pour le type de cross-over
;; l est une liste de longueur paire
(define (cross-list type l)


  (define (crossing done todo)

    (if (null? todo) done
(let* ( (v (car  todo))
(w (cadr todo))
(out (cross-over type v w))
(x   (car out))
(y   (cadr out))
)
  (crossing (cons v (cons w (cons x (cons y done))))
    (cddr todo)))))
```

```
  (crossing '() l)
)

;; on donne la taille de la population
;; on donne le nombre de generation
;; on donne le type de croisement (1 ou 2)
;; on donne la proba de mutation (entre 0 et 10)
;; on indique si on veut optimiser (0 1)
;; on indique combien d'affichage on veut
(define (ag szP nG crossing pmut op nbaff)

(define (aux population reply psz iter type-cr pm o best nbrun)

  ; on sort lorsque l'on a fini les iterations
  ; on sort lorsque le score du meilleur = 100
  ; on sort lorsque le score du meilleur = 0
  (if (or (= iter 0)
  (= (vector-ref (cdr best) 0) 100)
  (= (vector-ref (cdr best) 0)   0))
      (affiche-resultat reply nbaff)
      (let* ( (p (one-run population psz type-cr pm o best) )
      (x (make-stat p))
      (newbest (car x)) )
(aux p (cons  (cons nbrun x) reply)
     psz (- iter 1) type-cr pm o newbest (+ nbrun 1)) ) )
 )


 ; on verifie la qualite des parametres
(let* (( mypopsz (cond ((< szP 20) 20)
       ((> szP 1000) 1000)
       ((= (modulo szP 4) 0) szP)
       (else (- szP (remainder szP 4))) ))
       ( mynbg   (cond ((< nG 1) 1)
       ((> nG 1000) 1000)
       (else nG)))
       ( mycr    (if (or (< crossing 1) (> crossing 2)) 1 crossing))
       ( mypm    (if (or (< pmut 0) (> pmut 10)) 10 (- 10 pmut)))
       ( myoptim (if (= op 1) op 0))
       ( init-p  (init-population mypopsz) )
       ( best-i  (make-stat init-p) )
       )
  (aux init-p (list (cons 0 best-i))
       mypopsz mynbg mycr mypm myoptim (car best-i) 1)
 )
)

;; affichage minimaliste
; @p1 : la liste des resultats
; @p2 : le nombre d'elements a afficher
(define (affiche-resultat reply iter)
  (if (or (null? reply) (= iter 0))
     (begin (display "-----------------------") (newline))
     (begin (affiche-last (car reply))
    (affiche-resultat (cdr reply) (- iter 1))))
)

(define (affiche-last soluce)
```

```
  (let* ( (v (list->vector soluce))
  (index  (vector-ref v 0))
  (chromo (vector-ref v 1))
  (min-f  (vector-ref v 2))
  (moy-f  (vector-ref v 3))
  (max-f  (vector-ref v 4)) )
       (begin
 (display "-----------------------") (newline)
 (display chromo) (newline)
 (display "iterate : ") (display index) (newline)
 (display "minimum : ") (display (floor min-f)) (newline)
 (display "moyenne : ") (display (floor moy-f)) (newline)
 (display "maximum : ") (display (floor max-f)) (newline)
 )))

;; initialisation de la population
;; on recupere une liste  (sigma-fitness (c1 #score) ... (cn #score) )
(define (init-population size)

  (define (aux sz soluce sigma)
    (if (= sz 0) (cons (vector sigma sigma) soluce)
(let* ((x (build-chromosome-alea))
       (y (eval-fitness x)) )

  (aux (- sz 1) (cons (cons x (vector y y)) soluce) (+ sigma y)))))

  (aux size '() 0)
)

;; prend en entree une population
;; renvoie un quadruplet
;; le meilleur chromosome
;; le score mini, moyen et maxi
(define (make-stat population)

  (define (aux todo best max-fit min-fit moy-fit)
    (if (null? todo)
(list best min-fit moy-fit max-fit)
(let* ((chromo (car todo))
       (score-chro  (vector-ref (cdr chromo) 0))
       (score-best (vector-ref (cdr best) 0))
       (newbest (if (> score-chro score-best) chromo best)))
  (aux (cdr todo) newbest
       (max max-fit score-chro)
       (min min-fit score-chro)
       moy-fit))))

  (let* ((sigma (vector-ref (car population) 0) )
 (vlist (cdr population))
 (moy-fitness (/ sigma (length vlist))) )
    (aux (cdr vlist) (car vlist)
 (vector-ref (cdar vlist) 0)
 (vector-ref (cdar vlist) 0) moy-fitness) )
)

;; choisi sz elements dans la population
;; npop est soit vide, soit contient deja un element

(define (select population sz sigma npop item)
```

```
  (define (aux newpop iter)

    (if (= iter sz) newpop
(let* ((val (random sigma))
       (got (find-one val population)) )
  (aux (cons got newpop) (+ iter 1)) )))

  (define (find-one x tovisit)
    (let* ((z (car tovisit))
   (y (vector-ref (cdr z) item)))
      (if (<= x y) z (find-one (- x y) (cdr tovisit)))))

  (aux npop 0) )

; one-run
; selection
; crossover
; mutation
; @return nouvelle population
(define (one-run pop szp cross pm opt current-best)

  (let* ((sigma (vector-ref (car pop) opt) )
 (mypop (cdr pop)) ; list des chromosomes
 (szz   (if (= opt 0)
    (quotient szp 2)
    (- (quotient szp 2) 1) ) )
 (nextg (if (= opt 0) '() (list current-best)))
 (pool  (select mypop szz sigma nextg opt))
 (thepool (cross-list cross  pool)) )
    (build-next-population thepool pm opt) ) )

;; @p0 = la somme fitness actuelle
;; @p1 = une liste de paires chromosome score
;; @p2 = la proba d'une mutation
;; @p3 = 0 ou 1 pour l'optimisation des scores
;; @return une paire (sigma population)
(define (build-next-population laliste pm o)

  (define (finalize sigma0 sigma done todo minimum)

    (if (null? todo) (cons (vector sigma0 sigma) done)
        (let* ((first (car todo))
       (v (car first))
       (s (- (vector-ref (cdr first) 0) minimum))
       (t (* s s)) )
  (finalize  sigma0 (+ sigma t)
     (cons (cons v (vector (vector-ref (cdr first) 0)
   t))
   done)
     (cdr todo) minimum)))
    )

  (define (aux sol todo sigma mini)

    (if (null? todo)
(if (= o 0)
    (cons (vector sigma 0) sol)
    (finalize sigma 0 '() sol (+ 1 mini)))
```

```
(let* ((x (car todo))
       (y (if (< (random 10) pm)
      (mutation (car x)) (car x) ))
       (z (if (equal? y (car x))
      (cdr x) (vector (eval-fitness y) 0) ))
       (minima (min mini (vector-ref z 0)) ))
  (aux (cons  (cons y z) sol)
       (cdr todo)
       (+ sigma (vector-ref z 0)) minima)
  )
))

  (aux '() laliste 0 101)
)

;; la configuration mysterieuse
(define setup
 (begin
   (set! chromoSz 32)
   (set! laSolution (build-chromosome-alea))
   (set! secret (true-fitness laSolution))
   (display 'done) (newline)
   )
)
```

Bibliographie

[1] H. Abelson, N. Adams IV, D. Bartley, and M. others. Revised[4] report on the algorithmic language scheme, Nov. 1991.

[2] H. Abelson, G. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1985. traduit en 89 chez InterEditions.

[3] J.-M. Alliot, T. Schiex, P. Brisset, and F. Garcia. *Intelligence Artificielle & Informatique Théorique*. Cépaduès Éditions, Toulouse, France, 2nd edition, 2002. ISBN 2-85428-578-6.

[4] K. R. Apt. Introduction to logic programming. Report CS-R8741, CWI, Sept. 1987.

[5] A. Arnold and I. Guessarian. *Mathématique pour l'informatique*. Number 10 in LMI. Masson, 1993. ISBN 2-225-84011-3.

[6] I. Asimov. *Les robots*, volume 453. J'AI LU, 1967.

[7] E. Audureau, P. Enjalbert, and L. Fariñas del Cerro. *Logique Temporelle : Sémantique et Validation de programmes parallèles*. Études et Recherches en Informatique (ERI). Masson, Paris, France, 1990. ISBN 2-225-81967-X.

[8] D. Beasley, D. Bull, and R. Martin. An overview of genetic algorithmes : Part 1, fundamentals. *University Computing*, 15(2) :58–69, 1993.

[9] J.-M. Besnier. *Les théories de la connaissance*, volume 105 of *Dominos*. Flammarion, 1996. ISBN 2-08-035458-2.

[10] Boizumault. *Prolog : l'implantation*. Collection études et recherches en informatique. Masson, 1989.

[11] A. Bonnet. *L'intelligence artificielle : promesses et réalités*. InterÉditions, 1984. ISBN 2-7296-0077-9.

[12] B. Bouchon-Meunier. *La Logique Floue*, volume 2702 of *Que sais-je ?* PUF, 2eme edition, 1994. ISBN 2-13-045007-5.

[13] J.-P. Braquelaire and M. Desainte-Catherine. Modèles et Techniques de Programmation. Technical report, Université de Bordeaux I, Oct. 1992. Polycopié de LMI.

[14] I. Bratko. *Programmation en PROLOG pour l'intelligence artificielle*. InterEditions, 1988.

[15] R. Bryant. Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. *ACM Computing Surveys*, 24 :293–318, Sept. 1992.

[16] S. Byrne. The gnu smalltalk user's guide, 1990.

[17] J. Chailloux. *Le_Lisp de l'INRIA, manuel de référence*, Nov. 1983. draft.

[18] J.-P. Changeux and A. Connes. *Matière à pensée*. Number OJ22 in Odile Jacob. Éditions du Seuil, Feb. 1992. ISBN 2-02-014676-2.

[19] H. Chaudet and L. Pellegrin. *Intelligence Artificielle et Psychologie Cognitive*. Dunod, 1998. ISBN 2-10-002989-4.

[20] A. C. Clarke. *2001, l'odyssée de l'espace*, volume 349. J'AI LU, 1968.

[21] Clocksin and C. S. Mellish. *Programming in Prolog*. Springer Verlag, 1980. traduit en 1986 chez Eyrolles.

[22] C. Codognet. *Backtracking Intelligent en Programmation Logique : un cadre général*. PhD thesis, Université de Paris VII, Jan. 1989.

[23] P. Codognet. *Backtracking Intelligent en Programmation Logique : de la théorie à la pratique et à l'application au parallélisme*. PhD thesis, Université de Bordeaux I, Jan. 1989.

[24] O. commune. *Jeux et Stratégie*. Sciences et Avenir, 1980–1988.

[25] O. commune. *LA RECHERCHE en intelligence artificielle*. Number S52 in Sciences. Éditions du Seuil, 1987. ISBN 2-02-009451-7.

[26] M.-M. Corsini. Algorithmique et structures de données (brouillon). Notes de cours, 65 pages, Juillet 2002.

[27] M.-M. Corsini. Introduction aux réseaux de neurones (brouillon). Notes de cours, 91 pages, Juillet 2002.

[28] M.-M. Corsini. Recherche opérationnelle (brouillon). Notes de cours, 28 pages, Juillet 2002.

[29] M.-M. Corsini and A. Rauzy. Toupie : un langage de programmation par contraintes pour l'analyse formelle de programmes concurrents. *Technique et Science Informatiques*, 14(6) : 753–782, June 1995.

[30] D. Crevier. *à la recherche de l'I.A.* NBS. Flammarion, 1997. ISBN 2-08-211217-9.

[31] D. Diaz. The GNU prolog web site, 2002.

[32] R. Dubarle. *Initiation à la logique*. Number Série A in Collection de la logique mathématique. Gauthier-Villars, 1957.

[33] EC2, editor. *11ème Journées internationales. Le traitement du langage naturel et ses applications*, volume Volume 8 of *Les systèmes experts et leurs applications*, Avignon, 27-23 Mai 1991.

[34] J.-L. Ermine. *Système Expert : Théorie et Pratique*. Technique et Documentation, 1989.

[35] B. S. Everitt. *The analysis of contingency table*. Chapman and Hall, London, 1977.

[36] D. Friedman, M. Wand, and C. Haynes. *Essentials of Programming Languages*. MIT Press, 1992. ISBN 0-262-56067-4.

[37] T. Frühwirth, A. Herold, V. Küchenhoff, T. Le Provost, P. Lim, E. Monfroy, and M. Wallace. Constraint logic programming : An informal introduction. In G. Comyn, N. E. Fuchs, and M. J. Ratcliffe, editors, *Logic Programming in Action*, LNCS 636, pages 3–35. Springer Verlag, 1992. (Also available as Technical Report ECRC-93-5).

[38] L. Gacôgne. *Eléments de logique floue*. Hermes, 1997. ISBN 2-86601-618-1.

[39] J.-G. Ganascia. *L'intelligence artificielle*, volume 4 of *Dominos*. Flammarion, 1993. ISBN 2-08-035141-9.

[40] A. Goldberg and D. Robson. *Smalltalk-80 the Language and its Implementation*. Addison Wesley, 1983. connu aussi sous le nom de :*The Blue Book*.

[41] D. E. Goldberg. *Algorithmes Génétiques*. Vie artificielle. Addison-Wesley, June 1994. ISBN 2-87908-054-1.

[42] R. Griswold and M. Griswold. *The Icon Programming Language*. Prentice Hall, 1985.

[43] M. Guillen. *Invitation aux mathématiques : des ponts vers l'infini*. Number S104 in Sciences. Éditions du Seuil, Jan. 1995. ISBN 2-02-022674-X.

[44] C. Hanson. The scheme programming language, 2002.

[45] J.-P. Haton and co authors. *Le Raisonnement en intelligence artificielle*. IIA. InterÉditions, 1991. ISBN 2-7296-0335-2.

[46] D. Hofstadter. *Gödel, Escher and Bach : an Eternal Golden Braid*. New York : Basic Books, 1979. traduit en 1985 chez InterEditions.

[47] J. H. Holland. *Adaptation in Natural and Artificial Systems : an Introductory Analysis with Applications to Biology, Control and AI*. Ann Arbor, The University of Michigan Press, 1975.

[48] C. Jacquemin. *Logique et mathématiques pour l'informatique et l'I.A. 109 exercices corrigés*. MEMO-GUIDES. Masson, 1994. ISBN 2-225-84642-1.

[49] J. Jaffar and M. J. Maher. Constraint logic programming : A survey. *Journal of Logic Programming*, 19/20 :503–581, 1994.

[50] B. Jarrosson. *Invitation à la philosophie des sciences*. Number S74 in Sciences. Éditions du Seuil, Jan. 1992. ISBN 2-02-013315-6.

[51] M. Kahana. Foundations of human memory : Class notes for npsy 137, 29 Aout 2002.

[52] B. Kosko and S. Isaka. La logique floue. *Pour la Science*, 191 :62–68, Sept. 1993.

[53] M. Koutchouk. *Cobol : Perfectionnement et Pratique*. Dunod, Jan. 1989. ISBN 2-225-81532-1.

[54] J. Largeault. *La Logique*, volume 225 of *Que Sais-je ?* PUF, Nov. 1993. ISBN 2-13-046088-7.

[55] R. Lassaigne and M. de Rougemont. *Logique et fondements de l'informatique*. Hermes, 1993. ISBN 2-86601-380-8.

[56] K. Laumer. *L'ordinateur désordonné*, volume 93 of *Présence du Futur*. Eds. Denoel, 1963.

[57] J.-L. Laurière. *Intelligence Artificielle, résolution de problème par l'homme et la machine*. Eyrolles, 1987.

[58] J.-F. Le Ny, editor. *Intelligence naturelle et intelligence artificielle*. Psychologie d'aujourd'hui. PUF, 1ere edition, July 1993. ISBN 2-13-045271-X.

[59] R. Lefébure and G. Venturi. *Le Data Mining*. Informatiques. Eyrolles, 1998. ISBN 2-212-08981-3.

[60] J. Leroux. *Introduction à la logique*. Bibliothèque des sciences. Diderot Ed., 1998. ISBN 2-84352-084-3.

[61] P. Lignelet. *Fortran 77, langage Fortran V*. Masson, 1982.

[62] R. Loaëc. Nikon f 90, le "matriciel flash" est enfin une réalité. *Chasseur d'images*, 146 : 40–48, Oct. 1992.

[63] P. Lévy. *La machine univers. Création, cognition et culture informatique*. Number S79 in Sciences. Éditions du Seuil, Apr. 1992. ISBN 2-02-013091-2.

[64] L. Martin and G. Baillargeon. *Statistique appliquée à la psychologie*. SMG, 2nde edition, 1990.

[65] G. Masini and co authors. *Les langages à objets*. IIA. InterÉditions, 1989.

[66] L. Matyska. Constraint logic programming : An overview. In *SOFSEM'93*, Hrdoňov, 1993. VÚSEI AR Bratislava.

[67] J. Mc. Carthy. Recursive functions of symbolic expression and their computation by machine. *Communications of the ACM*, 3(4) :184–195, 1960.

[68] A. Michiels. *Traitement du langage naturel et Prolog. Une introduction*. Hermes, 1991.

[69] E. Nagel, J. R. Newman, K. Gödel, and J.-Y. Girard. *Le théorème de Gödel*. Sources du Savoir. Éditions du Seuil, 1989. ISBN 2-02-010652-3.

[70] œuvre collective. *Le petit Larousse Illustré*. Larousse, 1989.

[71] œuvre collective. Bibliothèque pour la science. pour la science, diffusion belin, Sept. 1991.

[72] P. Oléron. *Le raisonnement*, volume 1671 of *Que Sais-je ?* PUF, 5eme edition, July 1996. ISBN 2-13-047031-9.

[73] A. Pelissier and A. Tête. *Sciences Cognitives : textes fondateurs (1943–1950)*. Psychologie et Sciences de la pensée. PUF, 1995. ISBN 2-13-46695-8.

[74] R. Penrose. *L'esprit, l'ordinateur et les lois de la physique*. InterEditions, 1992. ISBN 2-7296-0367-0.

[75] S. Pinson. Représentation des connaissances dans les systèmes experts. *R.A.I.R.O Informatique/Computer Science*, 15(4) :343–367, 1981.

[76] J. Pitrat. *Textes, Ordinateurs et Compréhension*. Eyrolles, 1985.

[77] C. Queinnec. *Langage d'un autre type : LISP*. Eyrolles, 1980.

[78] A. Rauzy. *L'évaluation sémantique en calcul propositionnel*. PhD thesis, Université Aix-Marseille II, Luminy, Jan. 1989. (in french).

[79] M. Resnick. *Turtles, Termites and Traffic Jams : Explorations in Massively Parallel Microworlds*. MIT Press, sixth edition, 2000. ISBN 0-262-68093-9.

[80] F. Rivenc. *Introduction à la logique*. Number P14 in Petite Bibliothèque. Payot, 1957. ISBN 2-228-88204-6.

[81] B. Ruyer. *Logique*. PUF, 3eme edition, 1998. ISBN 2-13-049287-8.

[82] G. Sabah. *L'intelligence artificielle et le langage*, volume T1. Hermes, 1988. Représentation des connaissances.

[83] J. Smith. *An Introduction to Scheme*. Prentice All, 1988.

[84] R. Smullyan. *Le livre qui rend fou*. Dunod, 1984. ISBN 2-040-15556-2.

[85] F. F. Software. Free foundation software, 2002. last visited : Nov 2002.

[86] F. F. Software. Guile : Project GNU's extension language, 2002.

[87] F. F. Software. Swi prolog, 2002. last visited : Nov 2002.

[88] M. C. South, G. B. Wetherill, and M. T. Tham. Hitch-hiker's guide to genetic algorithmes. *Journal of Applied Statistics*, 20(1) :153–175, 1993.

[89] G. Springer and D. Friedman. *Scheme and the Art of Programming*. MIT Press, 1989. ISBN 0-262-69136-1.

[90] K. Swingler. *Applying Neural Networks : A practical Guide*. Academic Press, 1996. ISBN 0-12-679170-8.

[91] G. Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38 (3), Mar. 1995.

[92] A. Thayse and co authors. *Approche logique de l'intelligence artificielle*, volume De la logique classique à la programmation logique. DUNOD informatique, 1989. ISBN 2-04-019860-1.

[93] A. Thayse and co authors. *Approche logique de l'intelligence artificielle*, volume De la logique modale à la logique des bases de données. DUNOD informatique, 1989. ISBN 2-04-018757-X.

[94] P. Thiry. *Notions de logique*. Méthodes en Sciences Humaines. De Boeck Université, 3eme edition, 1998. ISBN 2-8041-2965-9.

[95] J.-R. Tong-Tong. *La logique floue*. Hermes, 1995. ISBN 2-86601-485-5.

[96] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. Logic Programming Series. MIT Press, 1989.

[97] P. Van Hentenryck. Constraint logic programming. Technical Report CS-91-05, Department of Computer Science, Brown University, Jan. 1991.

[98] G. Vignaux. *Les Sciences Cognitives : une introduction*, volume 4193 of *essais*. Le Livre de Poche, 1994. ISBN 2-253-94193-X.

[99] D. Whitley. A genetic algorithm tutorial. Technical Report CS-93-103, Department of Computer Science, Colorado State University, march, 10 1993. URL `www.cs.colostate.edu/~whitley/`. voir aussi Q.10.5 de la faq de comp.ai.genetic.

[100] J. Wijsen. Data mining et data warehousing. 5 avril 2000.

[101] L. Wittgenstein. *Tractatus logico-philosophicus*. Number 109 in TEL. Éditions Gallimard, 1988. ISBN 2-07-070773-3. suivi de Investigations philosophiques.

[102] P. Wolper. *Introduction à la calculabilité*. InterÉditions, 1991. ISBN 2-7296-0372-7.

[103] X. Yao. A review of evolutionary artificial neural networks. *International Journal of Intelligent Systems*, 8 :539–567, 1993. URL `www.cs.bham.ac.uk/~xin/`.

[104] X. Yao. Evolving artificial neural networks. *PIEEE : Proceedings of the IEEE*, 87(9) : 1423–1447, 1999. URL `www.cs.bham.ac.uk/~xin/`.

[105] D. Zighed and R. Rakotomalala. *Graphes d'induction : Apprentissage et Data Mining*. Hermes, 2000. ISBN 2-7462-0072-4.