Chapitre 10
Introduction à l'intelligence artificielle

La logique classique et les logiques non classiques permettent de représenter la connaissance. Nous introduisons à présent une méthode logique permettant de manipuler la connaissance, c'est-à-dire de mettre en œuvre l'intelligence artificielle, notion qu'il convient de clarifier d'abord. Une première difficulté est que, contrairement à la plupart des domaines de science pure ou appliquée, l'intelligence artificielle n'a pas reçu de définition largement acceptée. Même dans un cas concret, un problème de programmation par exemple, il n'est pas commode de décider si on se trouve ou non dans le cadre de l'intelligence artificielle. Un moyen pragmatique de contourner cet écueil consiste à éviter de définir le concept, en se contentant d'en illustrer divers cas particuliers. Outre la frustration intellectuelle que cette omission peut engendrer, cette solution n'est pas applicable ici. Les exemples classiques reconnus de l'intelligence artificielle sont généralement complexes, et leur exposition sortirait du cadre d'un ouvrage dont l'objet n'est pas l'intelligence artificielle mais plutôt les éléments de logique qui y interviennent. La seconde difficulté que nous rencontrerons sera donc d'illustrer notre propos au moyen d'exemples très simples, voire trop simples, mais permettant quand même de mettre en évidence les spécificités de l'intelligence artificielle par rapport aux autres domaines d'application de la logique, notamment la programmation classique. Traditionnellement, l'intelligence artificielle est subdivisée en de nombreux sous-domaines, dont la plupart sont très spécialisés. Nous n'en évoquerons que quelques uns, qui ne requièrent pas au départ de connaissance extra-logique particulière.

10.1. Une définition naturelle de l'intelligence artificielle

L'intelligence artificielle vise à simuler sur des systèmes artificiels[1]
des comportements "intelligents", typiques des êtres humains.

A priori, cette définition ne semble pas devoir donner lieu à controverse. Divers exemples viennent spontanément à l'esprit; citons notamment le jeu d'échec, l'assistance au diagnostic médical, la traduction d'une langue dans une autre, la vision des appareils robotisés, etc. Néanmoins, cette définition suscite plus de questions qu'elle n'apporte de réponses; nous envisageons certaines de ces questions dans la suite. Signalons déjà que la notion de comportement intelligent est ambiguë; dans une moindre mesure, la notion de simulation peut aussi être interprétée de diverses manières. On peut aussi s'interroger sur le rôle privilégié accordé à l'ordinateur. L'intelligence artificielle est-elle une branche de l'informatique ? Enfin, la définition proposée est peu précise et non formelle, ce qui diminue son utilité concrète.

10.1.1. *Qu'est-ce qu'un comportement intelligent ?*

Nous n'entrerons pas ici dans un débat de psychologues et nous considérerons que les activités et comportements typiques de l'être humain sont intelligents. Par exemple, le fait de se déplacer, celui de parler, celui de regarder, constituent des comportements potentiellement intelligents. L'intelligence ne paraît pas résider dans l'aptitude à se mouvoir, à émettre des sons et à percevoir formes et couleurs, mais plutôt à contrôler et à structurer ces aptitudes. Marcher — intelligemment — implique notamment la possibilité de trouver son chemin, parler sous-entend la capacité de prendre part à une conversation et la vision suppose l'aptitude à interpréter les formes perçues en images signifiantes, en symboles, en lettres, mots et phrases dans le cas de la lecture, etc. Tous ces comportements impliquent une action, une intervention physique, mais c'est surtout le contrôle de cette action qui traduit l'intelligence. Une voiture permet le déplacement mais l'intelligence est dans le chef de son conducteur, qui contrôle ce déplacement. Dans le même ordre d'idée, même si les mains, les pieds et les yeux jouent un rôle important dans la conduite automobile, la partie intelligente de cette activité est le fait du cerveau, qui contrôle les membres et interprète les sensations visuelles transmises par la rétine.

La capacité de contrôle peut exister sans qu'il y ait intelligence. Un transistor contrôle un courant électrique, un thermostat contrôle une température et une écluse contrôle le flux d'une rivière, mais l'absence de complexité, le fait qu'un seul paramètre soit contrôlable, empêche ces objets d'être considérés comme des entités intelligentes. La complexité semble inséparable de l'intelligence, ce qui suggère la "définition" suivante :

[1] principalement des ordinateurs, ou des entités contrôlées par ordinateur.

Un comportement intelligent est une action contrôlée complexe.

On peut naturellement opposer de nombreuses objections à cette affirmation. La plus importante est peut-être son caractère réducteur : il n'est pas réaliste de vouloir traduire en une courte phrase un concept que l'on devine extrêmement sophistiqué. Dans notre contexte, cela n'est pas gênant car l'intelligence artificielle, telle que nous pouvons l'appréhender au travers de ses réalisations les plus marquantes, est fortement réductrice; nous reviendrons plus loin sur ce point. Une objection plus pertinente ici est que la complexité d'un comportement est difficile à apprécier et semble impossible à mesurer. En outre, il faut éviter de confondre la complexité d'un comportement et la complexité du résultat auquel ce comportement conduit. Par exemple, une photographie d'un paysage peut être aussi complexe qu'un tableau représentant ce paysage mais, en général, l'acte de photographier est moins complexe que la réalisation d'un tableau.

Cette dernière assertion mérite quelques justifications, qui permettront aussi d'éclaircir la notion de complexité. Dans la plupart des cas, le photographe pourra en peu de phrases décrire le mode opératoire permettant à une autre personne n'ayant pas vu la photo originale de prendre une autre photo très ressemblante à celle-ci. Il suffit de connaître le lieu, l'heure, les conditions météorologiques, l'angle de prise de vue, le type de pellicule et d'appareil utilisé, le type d'objectif, l'ouverture et la vitesse. Par contre, aussi longues que soient les indications données par le peintre, le copiste ne pourra reproduire fidèlement un tableau qu'il n'a pas vu. La longueur des instructions est une mesure d'une certaine sorte de complexité, celle qui nous intéressera ici. Nous imaginons facilement que la conception et la réalisation d'un robot peintre de tableau serait plus ardue que celle d'un robot photographe.[2] Chaque comportement, même le plus simple, comporte une complexité intrinsèque. Cela s'oppose quelque peu à une intuition selon laquelle peu de comportements seraient réellement "intelligents". Peut-être y a-t-il là une mauvaise appréhension du concept d'intelligence plutôt qu'un défaut dans la notion de complexité. L'intelligence est un concept très anthropomorphe et nous avons tendance à qualifier d'intelligent un comportement non seulement complexe, mais aussi — et surtout — mal compris. Qui pourrait expliquer pourquoi les tableaux de Claude Monet sont "bons" ? On pourrait étudier plus avant, en termes plus techniques, cette dualité intelligence / complexité mais les indications déjà données suffisent à montrer l'existence d'un lien étroit entre les deux notions. Ce lien nous sera utile car il est a priori moins simple d'estimer un niveau d'intelligence que d'estimer, voire même de quantifier un niveau de complexité.

[2] Notre propos n'est pas de prétendre à une supériorité de l'art du peintre sur celui du photographe. Cependant, la créativité de ce dernier consiste surtout à choisir son sujet et à déterminer comment il va le photographier; l'acte même de prendre la photo requiert surtout une bonne compétence technique.

A cet égard, un autre concept intéressant est celui d'apprentissage. On considère souvent qu'un comportement devant faire l'objet d'un apprentissage est intelligent; la longueur et les difficultés de l'apprentissage peuvent dans une certaine mesure se quantifier, ce qui donne une indication utile sur le degré d'intelligence requis par le comportement appris.

10.1.2. *Simuler un comportement intelligent*

Le praticien de l'intelligence artificielle ne vise pas à créer un comportement intelligent, mais seulement à le simuler. Un programme de jeu d'échecs, par exemple, doit produire des coups utiles, susceptibles de conduire à la victoire, mais il n'est pas requis qu'il "réfléchisse" ou qu'il "raisonne". Si une étape de la production du coup à jouer peut se ramener à une recherche systématique et exhaustive dans une table de coups possibles, ce n'en est que mieux; l'informaticien préfère une solution simple à une solution compliquée ... à condition naturellement que la solution simple soit aussi efficace.

Il convient de s'interroger brièvement ici sur la notion de simulation. Un point crucial est qu'une entité simulante doit ressembler à son modèle, mais le degré de similitude requis est extrêmement variable. Bien plus, pousser la ressemblance trop loin est contre-productif. Un exemple célèbre est celui des débuts de l'aviation. La machine volante de Louis Blériot était nettement supérieure à celle de Clément Ader, parce que ce dernier avait cru devoir imiter trop fidèlement le vol biologique.[3] L'évolution ultérieure de l'aviation a confirmé qu'imiter les techniques de vol des insectes, oiseaux et mammifères volants n'était pas indiqué. Observons d'ailleurs que limiter la ressemblance entre un phénomène réel et le modèle physico-mathématique permettant de l'étudier est une technique ancienne. Le physicien peut étudier avec profit les trajectoires des astres en assimilant ceux-ci à des points matériels sans dimension, dont le seul paramètre est la masse. Cela suffit notamment à déduire les lois de Kepler des principes généraux généraux de la mécanique. La ressemblance entre la Terre et le Soleil d'une part, et deux points matériels de paramètres adéquats d'autre part, est pourtant très ténue. A l'opposé, une copie d'un tableau de Monet, même très ressemblante, ne vaudra jamais qu'une petite fraction du prix de l'original. La ressemblance n'est qu'un aspect de l'adéquation entre une entité simulante et l'entité simulée.

Garder ceci à l'esprit permet d'éviter le développement d'une vue étriquée de l'intelligence artificielle. Un programme de jeu d'échecs est une réalisation de l'intelligence artificielle, dont les mérites doivent s'évaluer sur base des performances; le fait que ce programme simule plus ou moins fidèlement les moyens intellectuels mis en œuvre par un joueur humain est non pertinent : seul le résultat compte. Il est d'ailleurs plus prometteur d'essayer de simuler

[3] L'aéroplane de Clément Ader avait des ailes profilées comme celles d'une chauve-souris.

un comportement intelligent quand la qualité de ce comportement peut être évaluée avec une certaine précision. Cela signifie que la plupart du temps on tentera de simuler des comportements très particuliers, dans des contextes restreints. Le célèbre programme "Deep Blue", qui a remporté un match d'échecs contre le champion G. Kasparov, ne ressemble à ce dernier que sur un seul point, la compétence au jeu d'échecs. Même selon ce point de vue restrictif, la ressemblance est superficielle. Les mécanismes de recherche opérationnelle mis en œuvre par Deep Blue sont éloignés des modes de raisonnement humain. En résumé, Deep Blue n'est pas une entité intelligente; c'est une entité qui simule remarquablement bien, mais d'un point de vue extérieur seulement, la démarche du jeu d'échecs. A ce titre, c'est incontestablement un succès de l'intelligence artificielle.

10.1.3. *Intelligence artificielle et informatique*

Un joueur d'échec artificiel est censé jouer convenablement aux échecs mais on n'attend rien d'autre de lui. Il n'est pas nécessaire qu'il sache parler ou marcher, ni même qu'il puisse lui-même mouvoir les pièces et identifier les coups joués par l'adversaire. En fait, le joueur artificiel doit seulement pouvoir communiquer, sous n'importe quelle forme, les coups à jouer; on doit aussi pouvoir lui communiquer les coups joués par l'adversaire. On reconnaît ici le caractère restrictif d'une simulation de comportement intelligent. Admettre cette restriction permet de traiter la plupart des problèmes d'intelligence artificielle par l'informatique. L'entité simulant l'intelligence est simplement un ordinateur convenablement programmé; elle ne fait que traiter de l'information.

Un autre exemple instructif est celui du simulateur de vol. La composante essentielle de cette machine sophistiquée est l'ordinateur, même si les équipements périphériques tels les capteurs et les actuateurs sont indispensables pour permettre un niveau approprié de simulation. Cette simulation reste néanmoins très infidèle, puisque le simulateur est dépourvu de la caractéristique principale de l'avion, à savoir celle de voler. Cette "lacune" n'est pas gênante puisque le simulateur de vol ne teste pas l'habileté du candidat pilote à se mouvoir rapidement en altitude, mais bien celle de réagir rapidement et correctement à un environnement de vol et aux divers incidents qui peuvent se produire. Ces exemples, et d'autres que nous verrons plus loin, nous incitent à considérer que l'intelligence artificielle est principalement une subdivision de l'informatique.

10.1.4. *Le point de vue anthropomorphe*

Le comportement intelligent est pour nous une action contrôlée complexe mais nous considérons seulement sa partie centrale, qui consiste en un traitement d'informations. Par exemple, conduire une voiture requiert une bonne perception de l'environnement, essentiellement visuelle. Les sensations visuelles

sont interprétées en informations diverses, dont la partie utile est "filtrée"[4] et transmise à la partie décisionnelle du cerveau, qui déterminera les commandes motrices transmises aux muscles, qui eux-mêmes agiront sur le volant, l'accélérateur, etc. Les aspects sensoriels et moteurs ne sont habituellement pas considérés comme faisant partie du processus intelligent au sens strict, au contraire de l'étape centrale intermédiaire. De même, l'intelligence artificielle au sens strict n'étudie pas les capteurs, analogues de nos sens, ni les actuateurs, analogues de nos muscles, mais seulement le processus par lequel les données issues des capteurs sont transformées en commandes pour les actuateurs. Tant les données que les commandes sont de l'information, ce qui nous amène à admettre que le succédané artificiel du cerveau ne peut être que l'ordinateur convenablement programmé. On peut objecter à cela que, même sous l'angle structurel, l'ordinateur et le cerveau ont peu de ressemblance mais, comme nous l'avons déjà mentionné, vouloir améliorer cette ressemblance risque d'être contre-productif.[5] D'ailleurs, le lien entre le cerveau et l'intelligence est controversé et on peut difficilement confondre le cerveau proprement dit ("brain") et l'esprit ("mind"), que l'on définit précisément comme le siège de l'intelligence. Certains spécialistes estiment que notre cerveau n'est qu'une machine plutôt stupide, qui doit être remplie d'une grande quantité d'information avant de devenir intelligente; l'analogie avec l'ordinateur qui est inutile tant qu'il n'est pas pourvu d'un système d'exploitation et de programmes d'application est assez naturelle ici. On hésite à aller plus loin et à prétendre que l'être humain ne serait pas intelligent de naissance, mais qu'il le deviendrait au fur et à mesure que son cerveau se "remplit".

Sans adopter un point de vue aussi radical, on peut admettre que l'intelligence artificielle n'est pas le fait de l'ordinateur lui-même, mais des programmes et des données que sa mémoire comporte. Notons ici que la mémoire fait partie de l'intelligence (artificielle ou non).[6] En intelligence artificielle, on s'intéressera plus aux données et aux programmes qu'aux ordinateurs eux-mêmes. Dans les chapitres suivants, on étudiera notamment PROLOG, que l'on peut voir comme un programme intelligent générique; ce programme peut être spécialisé de manière à le rendre apte à résoudre des problèmes particuliers. On étudiera aussi des techniques de démonstration automatique.

[4] Si on envisage de dépasser un autre véhicule, on pourra noter plus ou moins inconsciemment sa marque, sa couleur, le numéro de la plaque, etc., mais l'information utile est son mouvement, c'est-à-dire sa position et sa vitesse; on déterminera aussi si l'environnement routier se prête à un dépassement, mais on ignorera sans doute une foule de détails inutiles.

[5] Les réseaux neuronaux artificiels imitent mieux la structure cérébrale que les ordinateurs classiques mais, pour de nombreuses tâches, sont moins performants que ces derniers.

[6] Dans la vie quotidienne, on note que les gens sont plus disposés à reconnaître des défaillances de mémoire plutôt qu'un défaut d'intelligence. En fait, la mémoire ne peut être séparée de l'intelligence; les gens intelligents se plaignant de leur mémoire déplorent le plus souvent d'oublier des *faits* mais ils gardent en mémoire des structures plus élaborées comme des règles de conduite, des algorithmes, des techniques, etc.

10.1.5. *Intelligence et traitement de l'information*

L'exemple de la vision, naturelle ou artificielle, illustre bien les multiples aspects d'un comportement intelligent, c'est-à-dire d'une action contrôlée complexe. On peut considérer la vision comme un processus en deux étapes. La première est le fait de l'oeil et conduit à une image rétinienne, c'est-à-dire à une très grande matrice de points lumineux, souvent appelés pixels. Vu le nombre énorme de pixels et le fait que chacun d'eux est susceptible de prendre plusieurs états, une image représente une très grande quantité d'information. Cependant, dans la plupart des cas, cette information sera "filtrée" et résumée en une faible quantité d'information utile, exprimable en une courte phrase, telle "voilà Jacques" ou "il fait beau ce matin". L'aptitude à isoler et à structurer en quelques concepts l'information pertinente noyée au sein d'une vaste quantité d'information brute, visuelle ou autre, est une composante essentielle de l'intelligence humaine, qu'il sera important de simuler dans un programme d'intelligence artificielle. Cette aptitude est d'ailleurs systématiquement mesurée par les tests d'intelligence, comme le montre le dessin de la figure 10.1. L'information brute comprise dans ces dessins est abondante; on s'en rend compte si l'on essaie de décrire méthodiquement et exhaustivement chacun de ces dessins. Par contre, si on lui demande ce qui distingue les quatre cases de gauche des quatre cases de droite, l'être humain devra "nettoyer" cette information pléthorique et abstraire de ces dessins que la partie gauche illustre le concept "3" tandis que la partie droite illustre le concept "4".

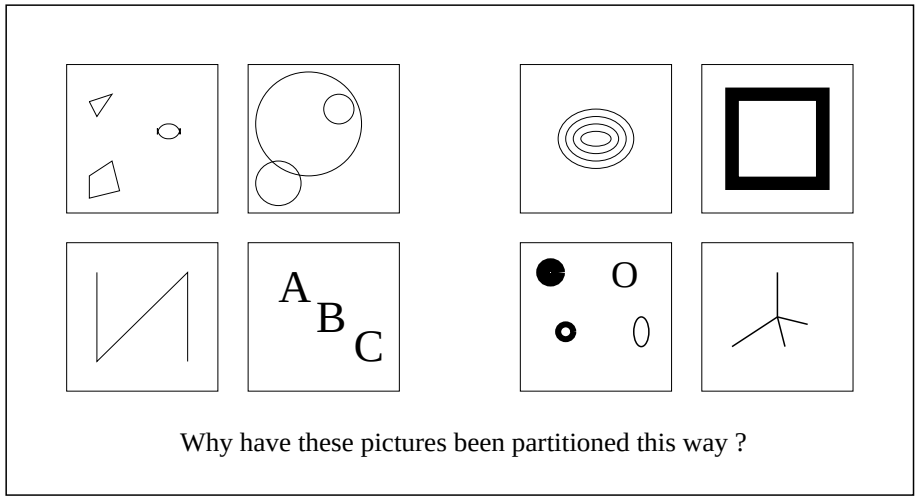


Figure 10.1 *Extraire l'information visuelle et conceptuelle pertinente*

L'esprit humain détermine facilement et rapidement que les huit dessins représentent chacun un nombre mais écrire un programme capable de détecter cela et, plus généralement, de passer avec succès un test d'intelligence de ce type est délicat.[7] Une des nombreuses aptitudes visuelles de l'esprit humain consiste en la reconnaissance des formes, en dépit de leur diversité. On peut créer un programme de reconnaissance de formes et le "nourrir" avec, d'une part, une série de photos d'arbres et, d'autre part, une série de photos de chats, et escompter que par la suite, ce programme soit capable de classer des photos (autres que celles qu'il connaît) selon qu'elles représentent un arbre ou un chat. Un premier module du programme fera de la détection de contour; il convertira chaque image (tableau de pixels) en un nombre relativement faible de segments de droite ou de courbe. Cette étape préliminaire n'est pas vraiment du domaine de l'intelligence artificielle, quoique l'on y reconnaisse un aspect "intelligent" qui est la compression, l'abstraction de l'information. Un deuxième module fera du "nettoyage", c'est-à-dire qu'il décidera, pour chaque segment, de le maintenir, de le supprimer ou de le modifier, par exemple pour le connecter à ses voisins. Un troisième module comparera le contour nettoyé à divers contours représentant des "standards" d'arbre ou de chat et décidera de classer l'image analysée dans l'une ou l'autre catégorie (ou dans aucune). Ce module utilisera diverses techniques (unification de l'image analysée et d'une image de référence par des transformations géométriques telles que les translations, rotations et homothéties, recherche de caractéristiques physiques discriminantes, etc.). L'arbre schématisé à la figure 10.2 montre une fois de plus qu'un modèle n'a pas besoin d'être fidèle pour être utile. Même si la silhouette d'arbre est très sommaire, une fraction de seconde suffit à la distinguer des trois silhouettes de chat représentées à droite de la figure. L'une des silhouettes est très différente des deux autres, parce que le chat est assis; cette différence ne trouble pas l'observateur humain, mais on conçoit aisément qu'une intelligence visuelle non négligeable soit nécessaire pour discerner les similitudes entre ces silhouettes et en tirer les conclusions adéquates.

Une autre aptitude liée à l'intelligence est la synthèse d'une information utile à partir de plusieurs informations fragmentaires et apparemment disparates. Considérons par exemple la phrase

Vincent est le fils du mari
de la fille unique de ma belle-mère.

On peut "décoder" cette phrase en

Vincent est mon fils.

[7] Ecrire un programme capable de compter des objets sur une image est relativement simple; la difficulté ne consiste pas à mettre en œuvre un concept particulier, connu à l'avance, mais à manier différents concepts et même à les créer. L'image de la figure 10.1 fait partie d'une série dont chaque élément concerne des concepts visuels distincts (objets ouverts ou fermés, groupés ou disséminés, de tailles voisines ou de tailles très différentes, etc.).

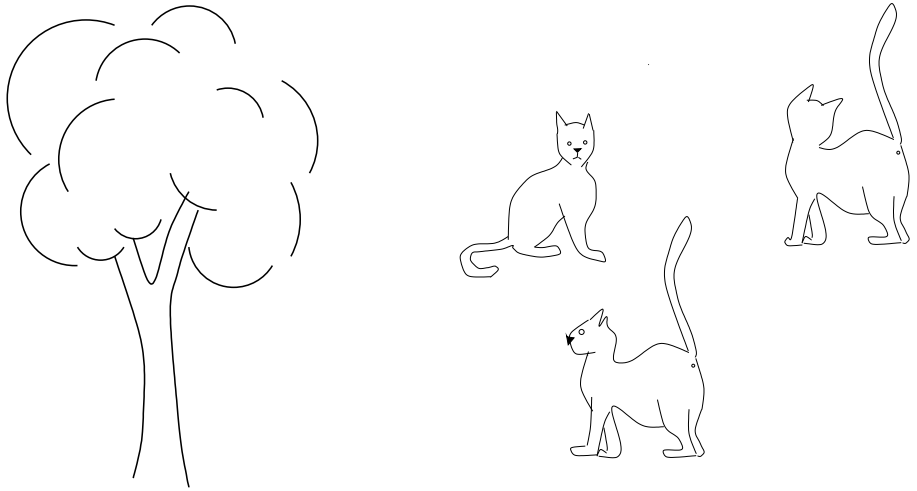


Figure 10.2 *Représentation sommaire de l'Arbre et du Chat*

Beaucoup d'énigmes peuvent se résoudre par un décodage de ce type. Par exemple, analysons le texte qui suit, se rapportant à un concours auquel trois étudiants de différentes nationalités ont pris part; chacun d'eux a son sport favori.

Michel joue au football et est mieux classé que l'Américain.
Simon est Belge et a surclassé le joueur de tennis.
Le nageur s'est classé premier.

Les questions sont :

Qui est le Français ?
Quel sport pratique Richard ?

Il existe une technique simple pour résoudre une énigme de ce type. Elle consiste à énumérer toutes les possibilités puis à les "filtrer" en fonction des indices donnés. Cette méthode, si simple qu'elle soit dans son principe, est une des composantes majeures de notre intelligence et probablement la technique la plus importante en intelligence artificielle. On voit facilement ici qu'il y a a priori $(3!)^3 = 216$ possibilités d'apparier les quatre triplets

$\{$ *Michel, Simon, Richard* $\}$,
$\{$ *Football, Tennis, Natation* $\}$,
$\{$ *Americain, Belge, Français* $\}$,
$\{$ *Premier, Deuxième, Troisième* $\}$.

Une seule de ces possibilités vérifie les trois indices :

Premier : le Belge Simon, nageur;

Deuxième : Michel, qui est Français et joue au football;
Troisième : Richard, l'Americain, qui est joueur de tennis.

De telles énigmes sont élémentaires parce qu'elles se ramènent à la construction et à l'exploration systématique d'un espace structuré et fini de solutions potentielles. Cela n'était pas vrai pour le test visuel de QI évoqué plus haut (Figure 10.1), notamment parce qu'aucune liste exhaustive de concepts susceptibles d'être visuellement illustrés n'est disponible. En fait, l'ensemble de ces concepts n'est qu'assez vaguement défini. Cet aspect n'est d'ailleurs pas spécifique à la vision. Au contraire, dans bien des cas, la "règle du jeu" n'est pas fournie, et la démarche intelligente requise consiste à découvrir cette règle plutôt qu'à la mettre en œuvre. Que signifie par exemple la séquence :

Which Atom Issue Art Tiny Finance Cigarette
All Inside Tear Llama Ignorant Enable Ceylon

composée de mots sans suite logique apparente ?[8]

10.2. Une définition orientée de l'intelligence artificielle

Le but de l'intelligence artificielle est de simuler, au sens large, le travail de contrôle effectué par le cerveau. "Au sens large" signifie que seul le résultat de la simulation compte; les processus mis en œuvre en intelligence artificielle ne sont pas censés ressembler aux processus physiologiques dont le cerveau humain est le siège.[9] L'intelligence artificielle est donc une branche de l'étude des programmes. Etudier un programme signifie s'intéresser à la fonction de contrôle que possède l'ordinateur exécutant ce programme, mais n'implique pas que l'on se préoccupe des processus physiques, électroniques, mis en œuvre lors de l'exécution. Nous considérerons donc les ordinateurs comme des machines abstraites capables d'interpréter les programmes qui leur sont soumis, sans nous soucier de leur fonctionnement interne.

Concrètement, l'intelligence artificielle a donc pour objet l'élaboration de programmes munis d'une fonction de contrôle ou, plus généralement, d'un aspect intelligent. Nous allons préciser les notions de programme et de programme intelligent.

10.2.1. *Qu'est-ce qu'un programme ?*

Tout programme a pour rôle de contrôler le fonctionnement de l'ordinateur qui l'exécute. Un ordinateur est une machine à états et le programme induit

[8] "WhAt Is ArTiFiCiAl InTeLlIgEnCe ?"

[9] On peut aussi s'intéresser à la simulation sur machine des processus de l'intelligence naturelle, mais ces derniers ne sont que très imparfaitement connus. Ce type de simulation donne rarement des résultats techniquement intéressants. Une exception notable est constituée par les réseaux de neurones artificiels, qui ont de multiples applications. Nous ne les évoquerons pas ici, car leur fonctionnement n'est pas basé sur la logique formelle.

une exécution, c'est-à-dire une suite d'états. L'exécution correspond donc à une trajectoire dans l'espace des états et le programme contrôle cette trajectoire. Cet aspect de contrôle ne suffit pas à faire de tout ordinateur exécutant un programme quelconque une machine intelligente. Pour mieux appréhender l'aspect "intelligent" éventuellement présent dans un programme, nous considérons quelques exemples, en commençant par un contre-exemple.

L_0 : $y := 1;$
L_1 : while $x > 0$ do
L_2 : $\quad y := y * x;$
L_3 : $\quad x := x - 1;$
L_4 : $F := y.$
L_5 :

$Ctrl$	x	y	F
L_0	3	$\perp$	$\perp$
L_1	3	1	$\perp$
L_2	3	1	$\perp$
L_3	3	3	$\perp$
L_1	2	3	$\perp$
...		...	
L_1	0	6	$\perp$
L_4	0	6	$\perp$
L_5	0	6	6

Figure 10.3 *Un programme impératif et un exemple d'exécution*

Le programme classique de calcul de la factorielle d'un nombre naturel est représenté à la figure 10.3, avec l'une de ses exécutions. D'un point de vue opérationnel, ce programme induit une séquence d'opérations arithmétiques élémentaires qui conduit au résultat, rangé dans la variable F. Pour justifier que le programme fonctionne bien, c'est-à-dire qu'il fournit le résultat exact, on utilise la méthode des invariants (voir § II.3). Un invariant approprié est :

$$
\begin{aligned}
& at\ L_0 \Rightarrow x = x_0 \\
\wedge\ & at\ L_1 \Rightarrow (0 \leq x \leq x_0 \wedge\ y * x! = x_0!) \\
\wedge\ & at\ L_2 \Rightarrow (0 < x \leq x_0 \wedge\ y * x! = x_0!) \\
\wedge\ & at\ L_3 \Rightarrow (0 < x \leq x_0 \wedge\ y * (x-1)! = x_0!) \\
\wedge\ & at\ L_4 \Rightarrow (x = 0 \wedge\ y = x_0!) \\
\wedge\ & at\ L_5 \Rightarrow (x = 0 \wedge\ y = x_0! \wedge\ F = x_0!)\,,
\end{aligned}
$$

où x_0 désigne la valeur initiale de x, c'est-à-dire un entier naturel quelconque. La vérification elle-même est un processus en deux temps. On s'assure d'abord que la formule ci-dessus est bien un invariant, et ensuite que cet invariant implique le bon fonctionnement du programme. La première étape est élémentaire mais requiert une connaissance suffisante de la fonction factorielle et des opérations arithmétiques qui permettent de la définir. Par exemple, pour vérifier la validité de la transition conduisant du point de contrôle L_2 au point

de contrôle L_3, il faut disposer du théorème suivant :

> Pour tous entiers x_0, x et y,
> si $0 < x \leq x_0 \wedge y * x! = x_0!$ est vrai,
> alors $0 < x \leq x_0 \wedge (y * x) * (x-1)! = x_0!$ est vrai.

Ce théorème, quoique très simple, découle d'un certain nombre de vérités arithmétiques, notamment celles-ci :

$$\forall x \, \forall y \, \forall z \, [x * (y * z) = (x * y) * z] \,,$$
$$\forall x \, [x > 0 \Rightarrow x! = x * (x-1)!] \,.$$

La seconde étape consiste à montrer que cet invariant (ou plutôt le programme auquel il se rapporte) induit un moyen de calcul fiable de la factorielle d'un nombre naturel quelconque x_0. Cette étape est ici très simple : l'invariant indique explicitement que tout calcul démarrant au point L_0 (en un état vérifiant $x = x_0$) et atteignant le point L_5 est tel que l'état final vérifie $F = x_0!$. Il est en outre facile de déterminer que ce calcul requiert l'exécution de $3x_0 + 3$ transitions.

Le procédé complet de calcul de la factorielle d'un entier naturel peut être structuré en trois phases comme suit.

- Les mathématiciens fournissent une définition (ou plusieurs définitions équivalentes) de la fonction factorielle reposant sur des fonctions et concepts mathématiques antérieurement définis (comme les opérations élémentaires) et sur des propriétés déjà démontrées de ces fonctions et concepts (associativité de la multiplication, par exemple).
- Le concepteur de programme sélectionne une définition se prêtant à un calcul effectif et la transforme en un programme exécutable. Le rôle de l'invariant est de convaincre le programmeur de la qualité de son travail, en reliant le programme à la connaissance mathématique qu'il met en œuvre.
- L'ordinateur, muni du logiciel adéquat (un système d'exploitation, un interpréteur ou un compilateur, ...), exécute le code; cela signifie que l'ordinateur passe, en un certain nombre d'étapes, d'un état initial comportant les données à un état final comportant les résultats.

Ce procédé requiert clairement de l'intelligence, de la part du mathématicien et aussi du programmeur. On peut défendre l'idée que le programme et l'ordinateur eux-mêmes présentent une certaine forme d'intelligence, mais elle est ici très limitée. La machine programmée ne fait qu'exécuter des instructions très détaillées et très spécifiques, dans un but bien précis. Il est néanmoins clair que l'intelligence réside essentiellement dans les deux premières phases, et très peu, ou pas du tout, dans la troisième.

Du point de vue de l'informaticien, il n'est pas nécessaire d'approfondir ce point et de décider si l'ordinateur, dont l'intervention est limitée à la troisième phase, est ou non pourvu d'une capacité ressemblant à l'intelligence; la question

intéressante est plutôt de savoir si le rôle de l'ordinateur (et du logiciel) peut être accru, réduisant d'autant le rôle des humains, impliqués dans les deux premières phases. On convertirait ainsi un ordinateur en un système (relativement) intelligent.

10.2.2. *Qu'est-ce qu'un système intelligent ?*

L'exemple très élémentaire que nous venons d'évoquer suggère une réponse, naturellement partielle, à cette question. Un système relativement intelligent serait capable, par exemple, de déduire la factorielle d'un nombre naturel directement de la connaissance mathématique concernant cette fonction, sans passer par l'intermédiaire d'un texte opérationnel, écrit par un concepteur de programme. Cette "définition" n'est pas vraiment satisfaisante puisqu'il n'y a pas une frontière claire entre la connaissance mathématique et la connaissance opérationnelle. En particulier, il n'est pas absurde de considérer que le texte

(10.1) $\quad (x, y) := (n, 1)$; while $x > 0$ do $(x, y) := (x - 1, y * x)$; return y ,

quoique typiquement opérationnel, constitue une définition mathématiquement acceptable de $n!$, même si le mathématicien préférera naturellement une définition telle que

(10.2) $\quad 0! = 1$; pour tout $n > 0$, $n! = n * (n-1)!$

ou encore que

$$(10.3) \quad n! = \int_0^{+\infty} x^n e^{-x} \, dx \,.$$

La définition (10.2) semble plus déclarative que la définition (10.1), ce qui la rend plus attrayante pour le mathématicien, mais la différence entre les deux n'est pas très importante : la plupart des langages de programmation admettent la récursion, ce qui rend la seconde définition aussi opérationnelle que la première. La définition (10.3) est intéressante notamment parce qu'elle permet d'étendre la fonction factorielle à l'ensemble des nombres complexes, entiers négatifs exclus. D'un point de vue opérationnel, par contre, cette définition est moins intéressante parce qu'elle recourt à des concepts comme l'exponentielle et l'intégration, moins élémentaires que les opérations arithmétiques et plus difficiles à mettre en œuvre.

D'une manière générale, le mathématicien privilégie un langage spécifique, comportant les égalités, les équations, les relations, etc., alors que les langages de programmation sont basés sur des notions assez différentes, comme les itérations, les séquences, les affectations, les procédures. Les deux types de langages admettent des recouvrements partiels; par exemple, la notion mathématique de fonction est proche de la notion informatique de procédure. Un premier pas, relativement modeste, vers les systèmes intelligents, consiste à doter la machine de l'aptitude à interpréter opérationnellement des fragments

de connaissance mathématique écrits en un langage mathématique, déclaratif. Notons que le langage mathématique formel est, le plus souvent, une logique; ceci est une première explication du lien étroit existant entre la logique et l'intelligence artificielle.

On peut estimer que, dans le cas du programme de calcul de la factorielle donné plus haut, le travail de construction est réduit et qu'il n'est donc pas très utile de vouloir s'en dispenser. Remarquons néanmoins que ce travail est plus important qu'il n'y paraît, puisque s'assurer de la correction du programme requiert la construction de l'invariant —une formule relativement longue— et la validation de celui-ci. Le fait que le processus de calcul associé au programme soit le même que le processus de calcul employé par un être humain pour calculer une factorielle n'incite pas non plus à chercher une solution plus "intelligente", qui ne serait sans doute pas meilleure.

Nous pouvons admettre que tout moyen permettant de réduire le travail de conversion de la connaissance mathématique déclarative en connaissance informatique, c'est-à-dire en un programme exécutable, est un pas vers l'intelligence artificielle. En effet, la machine prend alors en charge, en partie du moins, un travail qui, accompli par l'être humain, paraît requérir une démarche intellectuelle.

Des langages fonctionnels comme Lisp, Scheme et ML permettent la mise en œuvre opérationnelle immédiate de connaissances "semi-déclaratives" d'un type particulier, à savoir les définitions récursives telle que (10.2). Des systèmes de calcul symbolique (MACSYMA et REDUCE par exemple) permettent d'exploiter de la connaissance mathématique déclarative de types plus variés, et notamment la définition intégrale (10.3). Le mécanisme d'exploitation opérationnelle de la récursion est, en principe du moins, extrêmement simple; il ne suffit donc pas à faire de Lisp un système d'intelligence artificielle dans l'acception courante du terme. En ce qui concerne REDUCE et MACSYMA, les mécanismes sont plus nombreux et plus sophistiqués.

10.2.3. *Les Tours de Hanoï*

Un autre exemple bien connu, le problème des Tours de Hanoï, peut aider à mieux appréhender le concept d'intelligence artificielle. On dispose de trois piquets, nommés A, B et C, et de N disques de tailles échelonnées, percés en leur centre de manière à pouvoir être empilés sur les piquets. Un disque plus grand ne peut jamais se trouver au-dessus d'un disque plus petit. Au départ, tous les disques se trouvent empilés sur le piquet A. Le problème consiste à construire une suite de mouvements élémentaires, impliquant chacun le déplacement d'un seul disque, conduisant à la situation où tous les disques sont empilés sur le piquet B.

La manière classique d'aborder ce problème consiste à observer d'abord qu'une solution immédiate existe dans le cas où $N = 1$. En outre, et ceci est moins évident, on observe que, pour déplacer N disques du piquet A vers le piquet B, il faudra procéder nécessairement en trois étapes. La première consiste à transférer $N-1$ disques du piquet A au piquet C, la deuxième à transférer un disque (le plus grand) du piquet A au piquet B et la troisième à transférer $N-1$ disques du piquet C au piquet B. Cette tactique typiquement récursive est illustrée à la figure 10.4, dans le cas $N = 5$.

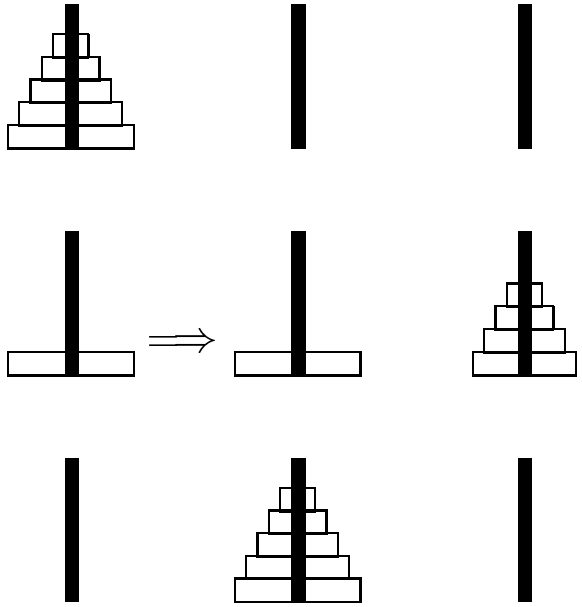


Figure 10.4 *Tours de Hanoï : approche classique*

Cette approche donne lieu au programme récursif ci-dessous :

$$\begin{aligned}\text{Towers}(1, X, Y, Z) &= [X \to Y]\\ \text{Towers}(N+1, X, Y, Z) &= \text{Towers}(N, X, Z, Y)\\ &\quad * \text{Towers}(1, X, Y, Z)\\ &\quad * \text{Towers}(N, Z, Y, X)\end{aligned}$$

(Le symbole $*$ désigne ici l'opérateur de concaténation de listes.)

Cette manière de résoudre le problème des Tours de Hanoï n'est pas considérée habituellement comme un exemple d'intelligence artificielle parce que la machine, ou plutôt l'interpréteur du langage fonctionnel utilisé, n'a déchargé l'humain que d'une tâche modeste (en principe), à savoir l'exploitation de la récursivité. Remarquons aussi que la séquence de mouvements induits

par ce programme est optimale, c'est-à-dire aussi courte que possible. Si l'on note $\lambda(N)$ cette longueur, on a :

$$\lambda(1) = 1\,,$$
$$\lambda(N+1) = 1 + 2\,\lambda(N)\,,$$

d'où on déduit $\lambda(N) = 2^N - 1$.

Pour voir que la tâche de "dérécursivation" est modeste dans son principe, il suffit de la simuler. On considère la relation récursive comme une règle de réécriture et on convertit la liste initiale, comportant un seul mouvement composé :

$$\text{Towers}(N, X, Y, Z)$$

en la liste :

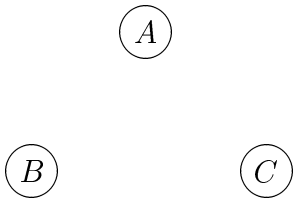
$$\text{Towers}(N-1, X, Z, Y)$$
$$\text{Towers}(1, X, Y, Z)$$
$$\text{Towers}(N-1, Z, Y, X)$$

comportant trois mouvements dont le premier et le dernier sont encore composés, sauf si $N = 2$. On réapplique le même procédé de réécriture pour obtenir

$$\text{Towers}(N-2, X, Y, Z)$$
$$\text{Towers}(1, X, Z, Y)$$
$$\text{Towers}(N-2, Y, Z, X)$$
$$\text{Towers}(1, X, Y, Z)$$
$$\text{Towers}(N-2, Z, X, Y)$$
$$\text{Towers}(1, Z, Y, X)$$
$$\text{Towers}(N-2, X, Y, Z)$$

et ainsi de suite. Après $N-1$ répétitions du procédé, on obtient la liste optimale des mouvements élémentaires.

Il est intéressant d'observer qu'une autre approche est possible, dont la mise en œuvre ne requiert pas la maîtrise du mécanisme de la récursion. Supposons que les trois piquets soient disposés en triangle, comme ci-dessous :



Le programme suivant convient, quand N est pair :

Répéter

- déplacer le petit disque dans le sens des aiguilles d'une montre;
- s'arrêter si la situation finale est atteinte;
- déplacer un autre disque, de la seule manière possible.

(Si N est impair, le sens de rotation du plus petit disque est inversé.)

La tâche de la machine est ici de simple exécution; cette approche a requis plus d'intelligence de la part du concepteur, et donc moins de la part de la machine. On peut s'en rendre compte facilement : se convaincre de l'exactitude du second programme est moins simple que pour le premier. On peut aussi considérer que la tactique récursive suggérée dans la première approche est plus facile à imaginer et (légèrement) plus difficile à mettre en œuvre que la tactique itérative de la seconde approche.

Le but ultime de l'intelligence artificielle serait de dispenser l'être humain de fournir une tactique; le système intelligent disposerait à l'avance de toutes les tactiques nécessaires ou, encore mieux, de l'aptitude lui permettant de les "concevoir" au fur et à mesure qu'elles se révéleraient nécessaires. Cette ambition est naturellement irréaliste mais, en délimitant le type de problèmes que la machine intelligente est censée résoudre, on peut arriver à des résultats intéressants. Les deux tactiques fournies pour résoudre notre exemple des Tours de Hanoï sont spécifiques à ce problème mais on peut concevoir une tactique plus générale, qui s'appliquerait à une vaste classe de problèmes.

Cette tactique générale provient d'une constatation très simple. A chaque étape, le nombre de mouvements possibles est limité. On peut donc représenter le problème des Tours de Hanoï par un arbre finitaire; la racine correspond à la situation initiale et tout nœud correspondant à une situation donnée admet pour successeurs directs les situations que l'on peut atteindre en déplaçant un seul disque. Chaque branche représente donc une suite licite de mouvements. Il suffit donc de construire l'arbre, "en largeur d'abord", pour obtenir la meilleure solution au problème. Cette tactique de construction d'arbre s'applique à une très large classe de problèmes, mais son intérêt pratique semble limité par un phénomène évident : le processus de construction de la solution est désespérément lent. Dans le cas qui nous occupe, la plupart des situations admettent trois successeurs directs, deux correspondant à un mouvement du petit disque, et un correspondant au mouvement d'un autre disque. Cela signifie que pour obtenir à coup sûr la solution optimale dans le cas de N disques, il faudrait construire un arbre ternaire de profondeur $2^N - 1$, comportant jusqu'à 3^{2^N-1} feuilles (et autant de nœuds internes). C'est naturellement impossible sauf pour les plus petites valeurs de N. On peut cependant observer que dans le cas présent, ainsi d'ailleurs que pour la plupart des problèmes similaires, l'arbre peut être sérieusement élagué. Tout d'abord, l'un des trois successeurs d'un nœud est nécessairement le prédécesseur de ce nœud et peut, de ce fait, être omis; il est en effet inutile de repasser par une situation déjà rencontrée. On peut encore réduire l'arbre

en excluant des successeurs non seulement le prédécesseur mais aussi toute situation déjà rencontrée plus haut dans l'arbre, même dans une autre branche. Ces optimisations élémentaires suffisent à rendre plus acceptable la taille de l'arbre, qui est représenté à la figure 10.5 dans le cas de trois disques. On pourrait encore réduire l'arbre en tenant compte des symétries; si on connaît la suite $S(3, A, B)$ de mouvements nécessaires pour faire passer, par exemple, trois disques du piquet A vers le piquet B, on obtiendra $S(3, C, A)$ en appliquant à $S(3, A, B)$ la substitution $(A, B, C) := (C, A, B)$.

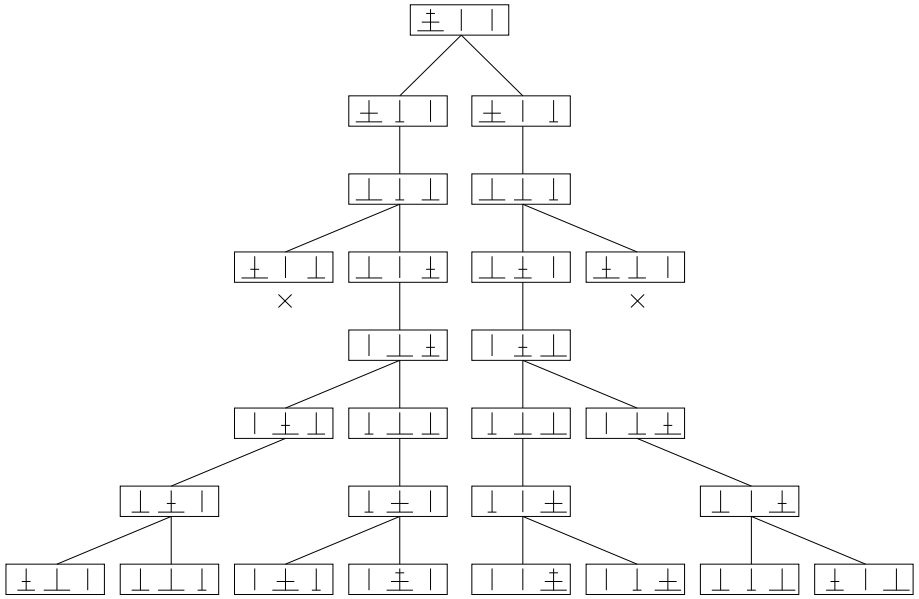


Figure 10.5 *Tours de Hanoï: recherche systématique*

Il est peu intéressant d'appliquer la recherche systématique dans le cas du problème des Tours de Hanoï parce qu'une tactique plus spécifique et aussi plus efficace est connue ... c'est-à-dire parce que la partie intelligente du problème a déjà été effectuée. En fait, on pourrait dire que ce problème *n'appartient plus* à l'intelligence artificielle. Les exemples qui suivent montreront que l'appartenance d'un problème à l'intelligence artificielle dépend aussi de la connaissance que nous avons de ce problème. En fait, une situation parfaitement connue requiert moins le recours à l'intelligence (artificielle ou non) qu'une situation mal connue. De toute manière, la frontière entre les situations requérant de l'intelligence et les autres sera naturellement floue. Un autre point important, bien illustré par le problème des Tours de Hanoï, est que, si on a le choix entre l'approche classique et l'approche intelligence artificielle, il faut choisir la première. Ceci n'est paradoxal qu'en apparence. Dans une situation où l'être humain peut réagir par réflexe, il aurait généralement tort de prendre le temps de vérifier le bien-fondé de l'action réflexe avant de l'accomplir.[10]

10.2.4. *Nim*

Le jeu de Nim met en compétition deux joueurs. Il nous intéresse parce que l'on peut y jouer en utilisant les ressources de l'intelligence, naturelle ou non, avec un certain intérêt; cependant, une tactique infaillible, ne requérant pas de raisonnement mais un simple calcul, permet de jouer de manière infaillible. Un certain nombre de tas d'allumettes sont disponibles au départ. (La configuration initiale habituelle est de quatre tas, comportant respectivement sept, cinq, trois et une allumette, respectivement.) A tour de rôle, chaque joueur enlève une ou plusieurs allumettes, prises dans un seul tas, au choix. Le gagnant est le joueur prenant la dernière allumette.

La démarche du joueur humain ne connaissant pas la tactique spécifique du jeu sera d'anticiper les réactions possibles de l'adversaire, en prévoyant une certain nombre de coups à l'avance. Cette stratégie est optimale ... à condition de pouvoir la mettre en œuvre complètement. Le nombre de situations possibles croît très vite en fonction du nombre de coups considérés. Un programme d'intelligence artificielle procédera de même, en construisant un arbre de jeu analogue à celui déjà évoqué pour le problème des Tours de Hanoï. Les coups joués par le programme viseront à explorer une branche de cet arbre, gagnante pour le programme. En pratique, si le nombre total d'allumettes disponibles au départ n'est pas très élevé, cette stratégie fonctionnera très bien; elle devient cependant impraticable s'il y a par exemple 200 allumettes réparties en dix tas.

La tactique spécifique repose sur les quelques notions simples suivantes. On représente une situation de jeu par la liste des effectifs des tas, écrits en numération binaire. Par exemple, la configuration initiale habituelle est notée $(111, 101, 11, 1)$, soit l'équivalent binaire de $(7, 5, 3, 1)$. Une situation est *paire* si, pour tout n, le nombre de bits de rang n égaux à 1 est pair. C'est le cas pour la situation ci-dessus, qui comporte deux bits de rang 2 égaux à 1, deux bits de rang 1 égaux à 1 et quatre bits de rang zéro égaux à 1; ceci se remarque immédiatement si les effectifs sont disposés en une colonne:

$$\begin{array}{rr} 7 & 111 \\ 5 & 101 \\ 3 & 11 \\ 1 & 1 \end{array}$$

[10] Si on entre en contact avec un objet très chaud, le recul réflexe permettra d'éviter une brûlure; par contre, un recul conscient et motivé interviendrait trop tard.

Chaque colonne de chiffres comporte un nombre pair de 1.

Une situation qui n'est pas paire est dite *impaire*. On démontre facilement deux propriétés intéressantes :

- Tout coup licite joué au départ d'une situation paire conduit à une situation impaire.
- Au départ d'une situation impaire, il existe au moins un coup licite conduisant à une situation paire.

Il est donc possible au joueur confronté à une situation impaire d'adopter une stratégie telle que, jusqu'à la fin de la partie, son adversaire soit toujours confronté à une situation paire, dont la dernière sera inévitablement la situation vide, perdante.

Il est donc facile, pour un humain comme pour un programme, de jouer à Nim de manière optimale, même si le nombre d'allumettes au départ est élevé; c'est seulement si le joueur ne connaît pas la tactique qu'il recourra à une méthode d'intelligence artificielle, telle la recherche systématique dans un arbre modélisant le jeu. L'intelligence artificielle constitue donc le second choix et n'est adoptée que pour les problèmes où une tactique *ad hoc* n'est pas connue.

10.2.5. *Echecs et Hexapion*

Le célèbre jeu d'Echecs est intéressant parce qu'aucune tactique complète n'est connue, même si des centaines de stratégies de jeu, plus ou moins partielles, ont été proposées; les joueurs sont donc obligés de réfléchir. En début de partie, les joueurs utilisent beaucoup leur culture; il existe des "bibliothèques" d'ouvertures dont plusieurs siècles d'expérience ont prouvé l'efficacité. En fin de partie, il est parfois possible d'utiliser une stratégie algorithmique permettant, par exemple, à un Roi et une Tour de vaincre à coup sûr un Roi isolé. C'est sans doute en milieu de partie que le joueur aura le plus tendance à considérer, explicitement ou non, un arbre de jeu, nécessairement très partiel; en fait, le joueur examine les conséquences, à plus ou moins court terme, de quelques coups "prometteurs". Les Echecs constituent en théorie un bon exemple d'investigation des méthodes de l'intelligence artificielle mais la complexité même du jeu le rend peu approprié à cette brève introduction.

La difficulté quantitative disparaît si l'on considère une version simplifiée à l'extrême du jeu d'Echecs, à savoir l'Hexapion. Ce jeu se pratique sur un "échiquier" de trois cases sur trois. Chaque joueur dispose initialement de trois pions, posés sur la rangée la plus proche de lui. Comme aux Echecs, le pion avance en ligne droite (d'une case) et prend en diagonale. Pour éviter le risque de nullité, on convient qu'un joueur peut gagner de deux manières différentes :

- En menant un pion "à dame", c'est-à-dire sur la rangée la plus proche de l'adversaire.
- En mettant l'adversaire dans l'impossibilité de jouer.

Quatre parties d'Hexapion (pas nécessairement optimales) sont représentées à la figure 10.6. Notons que, comme aux Echecs, ce sont toujours les Blancs qui jouent les premiers.

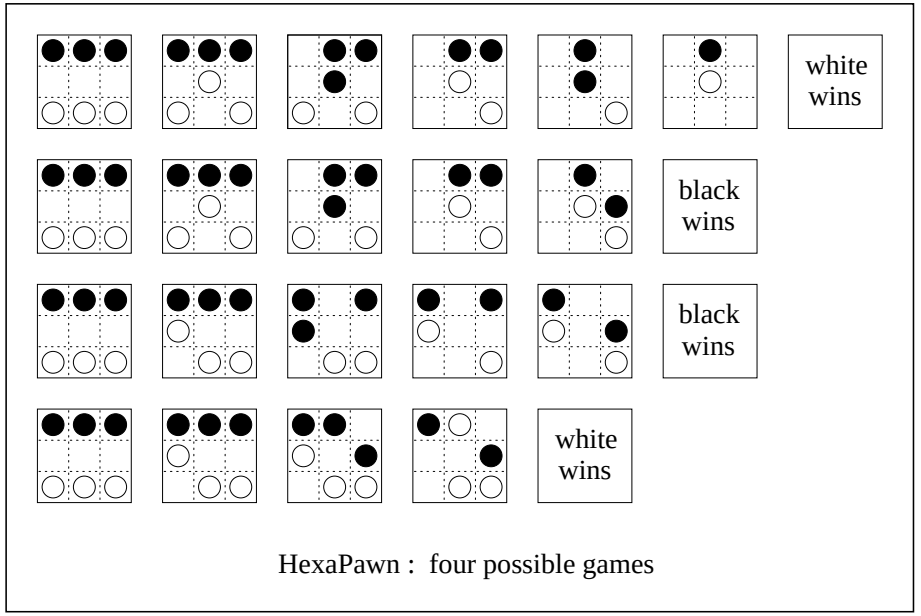


Figure 10.6 *Quatre parties d'Hexapion*

Il y a seulement quelques centaines de parties possibles à ce jeu; on pourrait donc concevoir un programme jouant de manière optimale par simple exploration d'une base de données comportant toutes les parties possibles. Une approche plus intéressante consiste à structurer cette base de données en un arbre de jeu. La figure 10.7 représente un fragment important de cet arbre; chaque nœud représente une configuration possible et chaque arc représente un coup possible.

L'arbre partiel de la figure 10.7 permet déjà d'apprendre beaucoup à propos d'Hexapion, et notamment qu'il existe une stratégie gagnante pour les Noirs.[11] Au départ, les Blancs ont deux possibilités, dont nous allons explorer les conséquences : avancer un pion latéral, ou avancer le pion central.

[11] En fait, la plupart des nœuds omis dans cet arbre correspondent à des configurations symétriques de celles associées à des nœuds présents dans l'arbre, ou encore à des nœuds frères d'une configuration gagnante, et donc inutile si on suppose qu'un joueur détecte systématiquement une victoire atteignable en un seul coup.

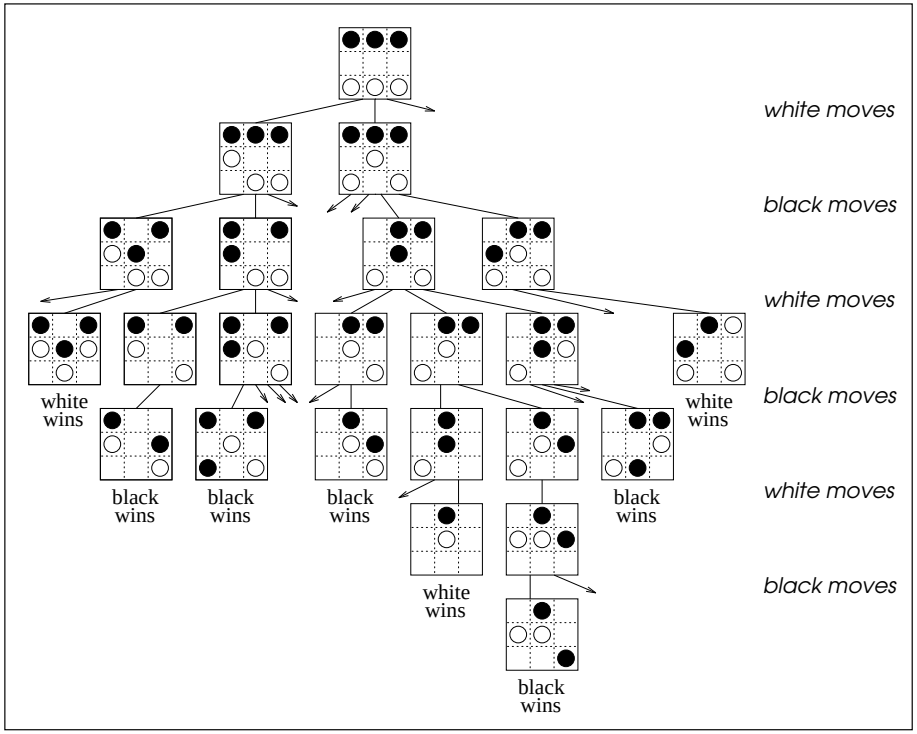


Figure 10.7 *Arbre partiel de jeu pour Hexapion*

- Supposons que les Blancs avancent le pion de gauche. Si les Noirs prennent immédiatement, il reste aux Blancs deux possibilités, dont aucune ne permet d'empêcher à coup sûr la victoire des Noirs :
  - Les Blancs prennent le pion avancé des Noirs, mais ces derniers peuvent gagner immédiatement en bloquant les Blancs.
  - Les Blancs avancent un de leurs pions (deux possibilités existent mais une seule est illustrée). Les Noirs répliquent en conduisant leur pion avancé à dame.
- Si les Blancs avancent le pion central, les Noirs ont à nouveau intérêt à prendre immédiatement. Après cela, les Blancs ont deux possibilités, dont aucune ne leur assure le gain :
  - Les Blancs prennent le pion avancé (deux possibilités), mais les Noirs peuvent provoquer un blocage (si une colonne est vide) ou avancer le pion latéral qui leur reste, et le mener à dame au coup suivant (voir figure).
  - Les Blancs avancent un de leurs pions, mais les Noirs peuvent alors immédiatement mener à dame l'un des leurs.

Cette analyse a mis en évidence une stratégie gagnante pour les Noirs : tous les mouvements possibles sont pris en compte pour les Blancs, seul le meilleur est considéré pour les Noirs. On pourrait de même étudier les stratégies possibles pour les Blancs, mais ceux-ci ne peuvent gagner que si les Noirs commettent une erreur.

Ces considérations stratégiques ne nous intéressent pas pour elles-mêmes; le point de vue de l'intelligence artificielle est qu'une stratégie gagnante peut non seulement être mise en œuvre par un programme, mais aussi, de manière plus ou moins explicite, inférée par un programme. Nous montrerons ceci dans la suite de ce chapitre; nous verrons aussi comment un programme peut apprendre au contact de son adversaire et devenir optimal après avoir perdu un certain nombre de parties.

10.3. Pour en savoir plus

Nous avons étudié brièvement dans ce chapitre introductif quelques aspects de l'intelligence artificielle et observé que ce concept était difficile à cerner. Dans les chapitres suivants nous abordons des points plus concrets, dans lesquels la logique joue un rôle important. Nous introduisons la programmation logique, qui permet de modéliser un univers particulier en une base de connaissances, et de déterminer ce qui peut être déduit de cette base de connaissances. La programmation logique permet de résoudre les problèmes les plus variés; nous l'utilisons notamment pour résoudre des énigmes amusantes mais non évidentes, et aussi pour écrire un programme simulant l'apprentissage du jeu d'Hexapion. Dans un autre registre, nous abordons le problème de la déduction automatique, que nous illustrons en vérifiant des propriétés de programmes. L'intelligence artificielle comporte beaucoup d'autres domaines, comme les systèmes experts, la planification, l'analyse et la génération de discours en langage naturel, l'apprentissage par réseaux neuronaux ou par arbres de décision, la vision artificielle, etc. De nombreux ouvrages ont été consacrés à l'intelligence artificielle; citons notamment [Dean et al. 95, Ginsberg 93, Luger and Stubblefield 98, Winston 92]. Mentionnons en français quelques classiques [Farreny et Ghallab 87, Haton et al. 91, Laurière 87]; de nouvreux ouvrages paraissent régulièrement, entre autres dans la présente collection "Langue, Raisonnement, Calcul".

Chapitre 11
Principes de la programmation logique

Nous avons observé que le raisonnement implique habituellement la définition d'un espace structuré d'objets et l'exploration de cet espace; les exemples d'Hexapion, de l'énigme des Trois Etudiants et du problème des Tours de Hanoï sont typiques à cet égard. Il est dès lors nécessaire de pouvoir représenter des objets et déterminer l'espace qu'ils forment, qui est l'ensemble de tous les objets; il faut également pouvoir spécifier la structure de cet espace et les techniques permettant de l'explorer. La logique formelle est le langage généralement le mieux adapté à la représentation de la connaissance et la programmation logique est un moyen de manipuler cette connaissance. Le système PROLOG est une réalisation imparfaite des techniques de la programmation logique, mais il apporte une solution simple et économique à la plupart des problèmes de représentation et de manipulation de la connaissance. A ce titre, il est un outil privilégié de l'intelligence artificielle.

11.1. Concepts de base

11.1.1. *Les termes*

Le premier concept en Prolog et en intelligence artificielle est celui d'objet. N'importe quoi est un objet : nombres, mots, personnes, ensembles, configurations de jeu d'échecs, etc. Cependant, le calcul et le raisonnement (automatisé) ne manipulent pas directement des objets mais leurs représentations. Pour des objets "abstraits" tels les nombres, la distinction ne semble pas très importante et il est fréquent de confondre, par exemple, le nombre 13 et sa représentation sous forme des deux caractères 1 et 3. On sait cependant que certaines

représentations (numérations binaire et décimale) sont plus commodes que d'autres (écriture en toutes lettres, chiffres romains). Pour des entités plus "concrètes", comme les personnes par exemple, la différence entre l'objet et sa représentation est plus importante même si elle n'est pas toujours explicitement soulignée. Les personnes peuvent être désignées par des mots (noms et prénoms) ou des nombres (numéros d'ordre, matricules), etc.

En Prolog comme dans beaucoup de langages de programmation, les objets sont représentés par des *termes*. Les *termes de base* ou *termes atomiques* ou *atomes* sont de deux sortes; un objet est désigné par une *constante* si la liaison entre l'objet et sa représentation présente un certain caractère de permanence; sinon, il est désigné par une *variable*. Plus précisément, Prolog *identifie* les constantes aux objets qu'elles représentent; en fait, Prolog ignore les objets et ne connaît que les représentations. Les variables, par contre, sont liées à des objets, ou plutôt aux constantes qui représentent ces objets, mais cette liaison peut évoluer en cours d'exécution d'un programme. Une même variable (un "mot" commençant par une majuscule ou par le caractère spécial `_`) peut selon les moments et les occurrences, représenter des objets divers; de même, un objet unique peut être lié à des variables distinctes.[1] Les constantes de Prolog sont essentiellement les nombres, tels `0`, `-3` et `2.567`, et les symboles, comme `abc`, `+` et `rs232`.

Un *foncteur* est un symbole que l'on utilise pour construire des *termes composés*. Chaque foncteur a une *arité* qui est un nombre naturel. Etant donné un foncteur `f` d'arité n et n termes $\tau_1, \ldots, \tau_n$ (atomiques ou composés), l'expression `f(`τ_1`, ... ,`τ_n`)` est un terme composé. Les expressions `m(a,b,c)`, `m(f(a,b),a(c,g(2,d)))` et `+(3,*(4,5))` sont des exemples de termes composés *fondamentaux*, c'est-à-dire ne comportant pas de variable. Comme les constantes,[2] les termes fondamentaux sont, en Prolog, identifiés aux objets qu'ils représentent. Du point de vue de Prolog, les termes fondamentaux *sont* les objets. C'est sous sa seule responsabilité que l'utilisateur Prolog associe à un terme fondamental un autre objet; il est assez naturel d'associer aux termes `-3` et `+(3,*(4,5))` le nombre -3 et l'expression arithmétique $3 + (4 * 5)$ mais ces associations sémantiques ne sont pas obligatoires. Les constantes lexicales (non numériques) ne représentent qu'elles-mêmes. Les termes composés sont des arbres dont les feuilles sont des atomes et les nœuds internes des foncteurs. Par exemple, `m(a,b,c)` est un arbre ternaire dont la racine est (étiquetée par) `m` et dont les feuilles sont (étiquetées par), de gauche à droite, `a`, `b` et `c`. A une même constante symbolique `f` peuvent correspondre plusieurs foncteurs, que l'on distingue par leurs arités respectives.

Supposons que dans une base de données, une personne soit représentée par son nom, son prénom et son âge. Une représentation typique en Prolog

[1] La notion de variable, ici assez floue, sera approfondie plus loin.

[2] Les constantes sont en fait les termes fondamentaux atomiques.

serait, par exemple, `p(gates,bill,43)`. Ici, `p` pourrait rappeler à l'utilisateur qu'un terme de ce type est associé à une personne, mais tout autre symbole pourrait convenir. Dans la vie courante comme en mathématique et dans d'autres sciences, une structure générale et importante est celle de *liste*. On pourrait par exemple représenter des personnes par des "structures anonymes" telles que `[gates,bill,43]`. Prolog permet cette notation commode mais l'interprète comme une variante syntaxique d'un terme composé classique, à savoir `.(gates,.(bill,.(43,[])))`. La constante symbolique `[]` désigne la liste vide. La constante symbolique "`.`" est un foncteur binaire (d'arité 2) employé pour apparier deux termes. Une liste non vide est, par définition, une paire dont la *tête* (premier composant) représente le premier élément de la liste et dont le *reste* (second composant) est la liste des éléments restants. En Prolog, la paire `.(a,b)` peut aussi s'écrire `[a|b]`. On en déduit que, par exemple, le terme `[gates|[bill|[]]]` est aussi une variante syntaxique du terme `.(gates,.(bill,[]))`, ou encore du terme `[gates,bill]`. Soulignons que, les termes composés étant des arbres, les foncteurs (et en particulier le foncteur "`.`") ne sont pas associatifs. Les termes `.(gates,.(bill,[]))` et `.(.(gates,bill),[])` sont donc bien distincts, de même que les termes `f(a,f(b,c))` et `f(f(a,b),c)`.

L'ensemble des termes fondamentaux, c'est-à-dire, pour Prolog, l'ensemble de tous les objets, semble très rudimentaire. Le concept de terme est néanmoins suffisamment flexible pour permettre de représenter commodément n'importe quel objet de la vie courante ou d'un domaine particulier de la connaissance. Par exemple, une configuration du jeu d'Hexapion peut être représentée par le terme :

```
hp(c(1,1,v),c(1,2,n),c(1,3,n),
   c(2,1,n),c(2,2,b),c(2,3,v),
   c(3,1,b),c(3,2,v),c(3,3,b))
```

dans lequel chaque `c`-triplet représente une case du tableau de jeu; les deux premières composantes identifient la position de la case (ligne et colonne) et la troisième identifie le contenu (`b` pour pion blanc, `n` pour pion noir, `v` pour case vide). D'autres représentations sont possibles et tout aussi commodes.

Chaque terme fondamental est un objet précis. On peut voir un terme non fondamental comme un objet générique, élément d'un certain ensemble. Cet ensemble sous-jacent est l'ensemble de tous les objets dans le cas d'une variable, mais devient un ensemble plus restreint dans le cas d'un terme composé. Les éléments de cet ensemble sont appelés les *instances* du terme. Tout objet (tout terme fondamental) est une instance de n'importe quelle variable. Les instances du terme `m(f(a,X),f(X,b),Y)` s'obtiennent en remplaçant `X` et `Y` par des termes fondamentaux quelconques. Des exemples d'instances de `m(f(a,X),f(X,b),Y)` sont `m(f(a,a),f(a,b),c)` (on a intancié `X` en `a` et `Y` en `c`) et `m(f(a,g(c)),f(g(c),b),g(c))` (on a intancié `X` et `Y` en `g(c)`); un contre-exemple est `m(f(a,b),f(c,b),d)` (les deux occurrences de `X` ont été

remplacées par deux termes différents).
Remarque. Un foncteur est toujours une constante symbolique; une expression telle que `X(a,b)` n'est donc pas un terme.

On peut résumer comme suit l'association que Prolog fait entre un terme et l'objet qu'il représente:

- Les objets de Prolog sont les termes fondamentaux.
- A tout terme est associé un ensemble d'instances, qui est un ensemble de termes fondamentaux.
- Un terme représente une instance générique, ou une instance particulière, ou l'ensemble des instances associé à ce terme.
- L'ensemble des instances d'un terme fondamental se réduit à ce terme; en conséquence, un terme fondamental représente un seul objet, lui-même.
- L'ensemble des instances d'une variable est l'ensemble de tous les termes fondamentaux.

11.1.2. *Faits et règles*

Un programme Prolog est une collection de *clauses*; une clause est un *fait* ou une *règle*. Syntaxiquement, un fait est simplement un terme suivi d'un point. Sémantiquement, un fait fondamental affirme l'appartenance d'un n-uplet à une relation. Par exemple, le fait

```
r(a,b,c).
```

affirme que le n-uple `(a,b,c)` appartient à la relation ternaire `r`. (Savoir ce que représentent la relation `r` et les éléments `a`, `b` et `c` est une autre question, qui ne concerne pas la sémantique de Prolog.) Un identificateur utilisé comme foncteur d'un fait est un *prédicat*. Comme en logique, un prédicat représente une relation. Les phrases "Le triplet `(a,b,c)` appartient à la relation (représentée par) `r`" et "`r(a,b,c)` est vrai" sont donc équivalentes. L'*arité* du prédicat est son nombre d'argument(s). Deux prédicats d'arités différentes sont différents même s'ils portent le même nom. Les prédicats d'arité 0 sont les *propositions*; on les écrit naturellement sans parenthèses. Par exemple,

```
z.
```

est un fait, affirmant que la proposition `z` est vraie. Tout fait fondamental affirme qu'un prédicat est vrai.[3]

Il est possible de décrire des relations finies en utilisant exclusivement des faits fondamentaux mais cela ne suffit pas pour décrire des relations infinies ni, en pratique, des relations finies de grande taille. Nous le montrons simplement en introduisant une structure de donnée essentielle en intelligence artificielle, le

[3] Au sens strict, `r` est un symbole prédicatif et `r(a,b,c)` est une forme prédicative. Dans l'usage courant, les deux expressions peuvent être qualifiées de prédicats.

graphe, c'est-à-dire la relation binaire. Nous avons vu qu'une relation binaire, et donc un graphe, se représentait par un prédicat binaire. Si on souhaite manipuler simultanément plusieurs graphes — c'est fréquent en intelligence artificielle — on prévoira un argument supplémentaire qui identifiera le graphe. En outre, on souhaite parfois introduire dans le domaine de la relation, c'est-à-dire dans l'ensemble des nœuds du graphe, des éléments qui n'interviennent dans aucun couple de la relation, c'est-à-dire dans aucun arc du graphe.[4] Un graphe (orienté) G dont les nœuds sont $n_1, \ldots, n_6$ et dont les arcs sont (n_1, n_3), (n_1, n_5), (n_2, n_3), (n_2, n_4), (n_4, n_5), (n_4, n_6) et (n_5, n_6) est naturellement décrit par un ensemble de faits fondamentaux:

```
node(g,n1).          arc(g,n1,n3).    arc(g,n1,n5).
node(g,n2).          arc(g,n2,n3).    arc(g,n2,n4).
node(g,n3).
node(g,n4).          arc(g,n4,n5).    arc(g,n4,n6).
node(g,n5).          arc(g,n5,n6).
node(g,n6).
```

Le premier argument `g` n'est nécessaire que si d'autres graphes sont aussi modélisés; rappelons que le nom du graphe doit commencer par une minuscule; tout identificateur commençant par une majuscule est, conventionnellement, une variable. Un *chemin* dans un graphe est une suite de nœuds telle que deux éléments consécutifs sont respectivement l'origine et l'extrémité d'un arc du graphe. Le premier élément d'un chemin est l'origine du chemin. Le dernier élément d'un chemin fini est l'extrémité du chemin. Un *cycle* est un chemin fini dont l'origine et l'extrémité coïncident. Un graphe *acyclique* est un graphe sans cycle. Un graphe acyclique fini, tel celui représenté ci-avant, comporte un nombre fini de chemins. Si nous nous intéressons à cet ensemble, il est possible de le décrire par des faits fondamentaux. L'un de ces faits serait:

```
path(g,[n1,n5,n6]).
```

Il est néanmoins fastidieux et inutile de construire cette description de manière explicite, car elle se déduit aisément de la description du graphe lui-même. L'une des fonctions de base du système Prolog est de dispenser son utilisateur de la construction explicite de telles descriptions; une description implicite, très simple, suffit. Elle est basée sur deux constatations. Tout d'abord, si `X` est un nœud, alors `[X]` est un chemin ("clause de base"). Ensuite, si `(X,Y)` est un arc et si `[Y|Ys]` est un chemin, alors `[X,Y|Ys]` est un chemin ("clause

[4] Il est question ici de graphe orienté. Chaque arc comporte une origine et une extrémité, dont les rôles ne sont pas interchangeables. Dans le cas d'un graphe non orienté, un arc (ou une arête) est une paire de nœuds plutôt qu'un couple de nœuds. On peut toujours assimiler un graphe non orienté à un graphe orienté symétrique : l'arête $\{a, b\}$ est alors modélisée par les deux couples (a, b) et (b, a). Dans la suite, il sera surtout question de graphes orientés.

inductive, ou récursive").[5] Ces deux constatations affirment qu'une certaine liste de nœuds est un chemin à certaines conditions. En Prolog, on formalise les affirmations conditionnelles par des *règles*. La notion de chemin se traduit par les deux règles suivantes :[6]

```
path(G,[X]) :- node(G,X).
path(G,[X,Y|Ys]) :- node(G,X), arc(G,X,Y), path(G,[Y|Ys]).
```

La signification logique d'une clause Prolog, c'est-à-dire sa sémantique déclarative, est très simple. Une règle telle que :

```
A :- B,C,D.
```

exprime que, si `B`, `C` et `D` sont vrais, alors `A` est vrai.[7] Le prédicat `A` est la *tête* de la clause, la suite de prédicats `B,C,D` en est le *corps*. Les variables sont universellement quantifiées. Par exemple, la première règle du programme ci-avant affirme que pour tout graphe `G` et pour tout nœud `X` de ce graphe, la liste `[X]` est un chemin de `G`. Un fait est simplement une clause dont le corps est vide. On écrit "`A.`" au lieu de "`A :- .`".

11.1.3. *Questions*

Un programme Prolog peut être vu comme une base de données relationnelle (descriptions explicites de relations) enrichie de règles (descriptions implicites de relations). On peut appeler cela une *base de connaissances*. Les bases de données et de connaissances peuvent être interrogées. Voici par exemple quelques questions envisageables dans le cas du programme relatif au graphe `g`, enrichi de la notion implicite de chemin.

- Quels sont les nœuds du graphe `g` ?
- Est-ce que `[n1,n5,n6]` est un chemin de `g` ?
- Quels sont les chemins de `g` dont l'origine est `n1` ou `n2` et dont l'extrémité est `n6` ?

La réponse à la première question est immédiate puisque les nœuds sont explicitement mentionnés (sous forme de faits fondamentaux) dans le programme. La seconde question appelle une réponse par oui ou non. Il est clair que l'usage

[5] On pourrait ajouter une "clause de fermeture", à savoir, tout chemin s'obtient par application des clauses de base et récursive. Ceci est implicite en Prolog.

[6] Notons ici une convention habituelle mais non impérative : quand une variable désigne une liste, on peut attirer l'attention sur ce point en utilisant un identificateur se terminant par "`s`". Observons aussi l'usage (impératif) de majuscules : notre description implicite de la notion de chemin dans un graphe est valable pour tout graphe et pour tout chemin.

[7] En logique, une formule de ce type est notée $(B \wedge C \wedge D) \Rightarrow A$ ou $(A \vee \neg B \vee \neg C \vee \neg D)$; on appelle "clause" une formule de ce type, d'où le choix naturel du mot "clause" pour désigner un élément d'une description en Prolog, c'est-à-dire une instruction d'un programme Prolog.

de la seconde règle relative aux chemins est indispensable pour répondre à la question. Un cas particulier de cette règle, ou plutôt une *instance* de cette règle est :

```
path(g,[n1,n5|[n6]]) :-
  node(g,n1), arc(g,n1,n5), path(g,[n5|[n6]]).
```

ou encore, de manière plus lisible :

```
path(g,[n1,n5,n6]) :-
  node(g,n1), arc(g,n1,n5), path(g,[n5,n6]).
```

La question principale se réduit à la suite des trois *sous-questions* `node(g,n1)`, `arc(g,n1,n5)` and `path(g,[n5,n6])`. Nous observons aussi que :

```
node(g,n1).
arc(g,n1,n5).
```

sont des faits fondamentaux présents dans la base de connaissance, donc la question principale est maintenant réduite à la sous-question : `path(g,[n5,n6])`. On utilise à présent une autre instance de la règle récursive relative aux chemins, à savoir :

```
path(g,[n5,n6]) :- node(g,n5), arc(g,n5,n6), path(g,[n6]).
```

A nouveau, on répond affirmativement aux deux premières nouvelles sous-questions `node(g,n5)` et `arc(g,n5,n6)`, qui correspondent à des faits fondamentaux du programme; seule la troisième sous-question :

```
path(g,[n6]).
```

subsiste. On utilise maintenant une instance de la règle de base relative aux chemins, à savoir :

```
path(g,[n6]) :- node(g,n6).
```

Cela conduit à la sous-question :

```
node(g,n6).
```

Cette sous-question correspond à un fait du programme, donc la réponse à cette sous-question, ainsi qu'aux sous-questions antérieures et à la question originale, est `oui`.

La troisième question est un peu moins simple. On peut construire tous les chemins issus de `n1` ou `n2` et ensuite éliminer ceux dont l'extrémité n'est pas `n6`; on peut aussi construire tous les chemins d'extrémité `n6` et ensuite éliminer ceux dont l'origine n'est ni `n1` ni `n2`. Il n'est pas évident a priori de choisir entre ces deux stratégies. On observe aussi un certain "gaspillage" : on construit une collection relativement importante de chemins pour n'en garder en fin de compte qu'une partie. Il convient de souligner que ce phénomène est omniprésent en intelligence artificielle ... et même en intelligence naturelle. Si la résolution d'un problème implique une démarche intelligente, elle implique aussi le risque de se tromper, ce qui, dans notre cadre, correspond à la construction de "solutions" non satisfaisantes, qui seront finalement rejetées.

Les quelques exemples envisagés au chapitre précédent, quoique simples, mettent bien en évidence le phénomène. Des jeux tels que Nim et Hexapion comportent des risques d'erreurs; un joueur, même raisonnablement compétent, est susceptible d'envisager plusieurs coups malheureux avant de penser à un coup favorable. Dans le cas de l'énigme des Trois Etudiants, on peut envisager diverses associations Etudiant-Nationalité-Sport avant de découvrir la bonne. En fait, l'intelligence implique la faculté de reconnaître les erreurs rapidement. Les bons joueurs d'Echecs envisagent relativement peu de coups avant d'en choisir un, mais même un champion ne pourra gagner face à un joueur relativement compétent s'il joue systématiquement le premier coup qui lui vient à l'esprit. Il n'empêche que l'aptitude à éliminer rapidement les "solutions" non satisfaisantes est très précieuse. En Prolog, on veillera à construire le moins possible de solutions insatisfaisantes; on essaiera aussi d'éliminer une solution partielle peu prometteuse avant d'avoir achevé sa construction. D'une manière générale, concevoir un espace de solutions potentielles et l'explorer efficacement est un des buts majeurs de l'intelligence artificielle.

11.2. Sémantique

La programmation logique met "la logique en action"; elle apparie la programmation et la logique. Elle comporte donc une sémantique opérationnelle (comment le système Prolog explore-t-il et exploite-t-il une base de connaissance, un programme logique?) et une sémantique déclarative (quels faits sont tenus et reconnus pour vrais par le système Prolog?). Les deux types de sémantiques sont introduits ici.

11.2.1. *Sémantique opérationnelle*

Comme l'ont déjà suggéré les exemples élémentaires de traitement de graphe, le système Prolog répond aux questions qui lui sont posées en construisant et en explorant une structure arborescente. Une ramification se produit quand plusieurs clauses sont susceptibles d'être utilisées pour la réduction d'une sous-question donnée. Une branche se termine positivement quand une solution est trouvée, après réduction de toutes les sous-questions (branche-succès); une branche se termine négativement lorsqu'aucune clause ne permet de réduire une certaine sous-question (branche-échec). Il se peut aussi qu'une branche ne se termine jamais (branche infinie). La *sémantique opérationnelle* de Prolog est la description précise de la procédure de construction et d'exploitation de cette structure arborescente que l'on nomme *arbre de recherche*. Cette notion a été étudiée sur le plan conceptuel dans le volume précédent (cf. § II.5.6); rappelons qu'elle repose sur deux algorithmes fondamentaux de la logique, l'algorithme de résolution et l'algorithme d'unification, étudiés dans le premier volume (cf. §§ I.4.4, I.6.4 et I.8.3). Nous illustrons ici, de manière plus concrète, la notion essentielle d'arbre de recherche, en abordant un exemple de traitement de listes; ce domaine est spécialement intéressant dans ce contexte parce que les listes constituent probablement la structure de donnée la plus utilisée en programmation symbolique et en intelligence artificielle.

Les mécanismes de base pour la construction de listes sont très simples. D'une part, on dispose d'une constante spéciale, notée `[]`, correspondant à la liste vide; d'autre part, étant donné un objet quelconque `X` et une liste quelconque `Xs`, le terme `.(X,Xs)`, plus communément noté `[X|Xs]`, désigne la liste dont le premier élément est `X` et dont les autres éléments forment la liste `Xs`. Par exemple, si `X` = `a` et `Xs` = `[1,2,3]`, alors `[X|Xs]` = `[a,1,2,3]`. Une construction plus générale est la *concaténation*. La concaténation de `[a,b,c]` et `[1,2]` est `[a,b,c,1,2]`. La concaténation est classiquement définie de manière récursive. D'abord, la concaténation de la liste vide et d'une liste quelconque `Bs` est simplement `Bs`. Ensuite, si la concaténation de `As` et `Bs` (listes quelconques) est `Cs`, alors, pour tout objet `A`, la concaténation de `[A|As]` et `Bs` est `[A|Cs]`. Ces deux postulats suffisent à définir la concaténation. Le programme Prolog correspondant permet de définir un prédicat ternaire, classiquement nommé `append`; ce programme est donné à la figure 11.1. Sur base du prédicat `append`, on peut définir un nouveau prédicat binaire `prefix` au moyen d'un simple fait (non fondamental) interprété comme suit : `Xs` est un préfixe de `Ys` si une liste `Zs` existe telle que la concaténation de `Xs` et `Zs` est `Ys`. Enfin, la figure 11.1 montre aussi l'arbre de recherche engendré par Prolog quand la question `prefix(Xs,[a,b])` lui est soumise.

L'arbre de recherche est construit selon des règles élémentaires mais strictes, qu'il convient d'analyser en détail. Observons d'abord que les nœuds et les arcs de l'arbre sont tous munis d'une étiquette. L'étiquette de la racine est la question soumise au système. Par rapport à cette question, le programme comporte une seule clause *unifiante*, c'est-à-dire une clause dont la tête a pour prédicat celui de la question.[8] En conséquence, la racine aura un seul fils. L'arc unissant la racine à ce fils est étiquetée par l'*unificateur principal* permettant d'égaliser le (premier) prédicat étiquetant la racine, soit `prefix(Xs,[a,b])` et la tête de la clause unifiante, soit `prefix(Xs,Ys)`.[9] Rappelons que les variables

[8] Une question ou une sous-question peut comporter plusieurs prédicats; c'est alors pour le premier de ceux-ci que l'on cherche des clauses unifiantes.

[9] La notion d'unificateur principal a été définie au paragraphe I.6.4.8; le paragraphe I.6.4.10 présente l'algorithme d'unification, qui détermine si deux termes admettent des unificateurs principaux et, si oui, fournit l'un d'eux. En première lecture, il est suffisant de se souvenir de ceci. Une *substitution* sur un ensemble V de variables est une fonction σ qui à toute variable de V associe un terme. Le domaine de la fonction σ s'étend immédiatement à l'ensemble $\mathcal{T}_V$ des termes dont toutes les variables sont dans V; l'image $\sigma(t)$ d'un terme t par σ s'obtient en remplaçant toute variable v de t par le terme $\sigma(v)$. Etant donnés deux termes $t_1, t_2 \in \mathcal{T}_V$ et une substitution σ de domaine V, on dit que σ est un *unificateur* de t_1 et t_2 si $\sigma(t_1) = \sigma(t_2)$. Un unificateur σ de t_1 et t_2 est dit *principal* si pour tout unificateur ρ de t_1 et t_2 il existe une substitution ϕ telle que $\phi \circ \sigma = \rho$.

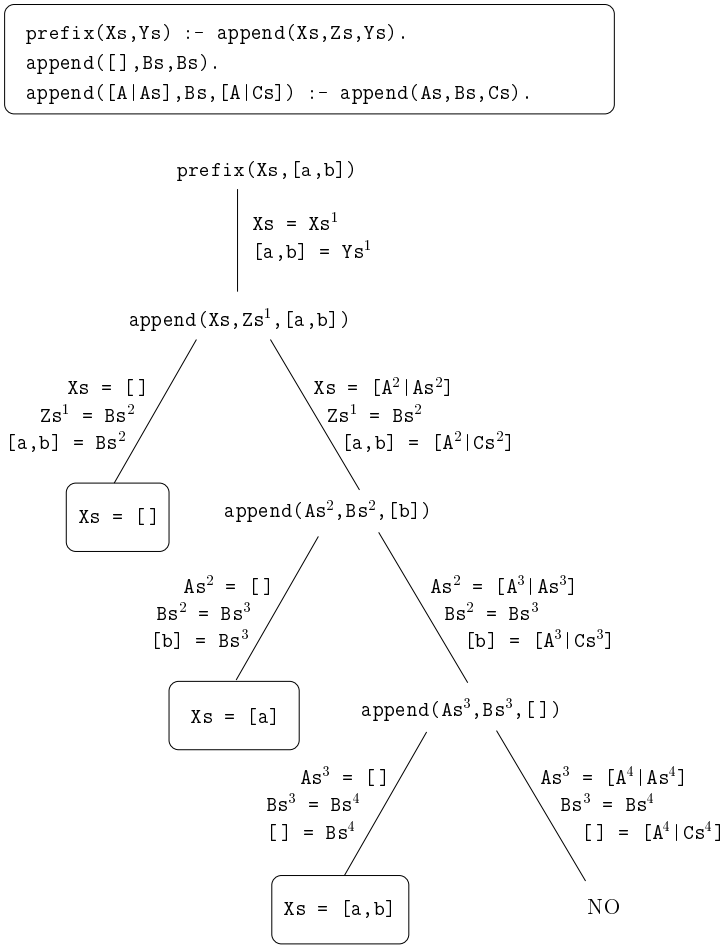


Figure 11.1 *Programme et arbre de recherche*

des clauses sont quantifiées universellement; l'identité de nom de la variable `Xs` de la question et de la variable `Xs` de la tête de la clause est donc accidentelle et nous rebaptisons la seconde variable en $\texttt{Xs}^1$. Plus généralement, les variables des clauses sont affectées d'un exposant correspondant au niveau de l'arbre où elles sont utilisées. L'unificateur principal adéquat doit ici égaliser la question `prefix(Xs,[a,b])` et la tête de la clause unifiante $\texttt{prefix(Xs}^1\texttt{,Ys}^1\texttt{)}$; il se composera donc des deux équations $\texttt{Xs} = \texttt{Xs}^1$ et $\texttt{[a,b]} = \texttt{Ys}^1$. En fait, la clause unifiante signifie que, pour obtenir $\texttt{prefix(Xs}^1\texttt{,Ys}^1\texttt{)}$, il est suffisant d'obtenir $\texttt{append(Xs}^1\texttt{,Zs}^1\texttt{,Ys}^1\texttt{)}$. En tenant compte de l'unificateur, cette clause permet la réduction de la question `prefix(Xs,[a,b])` à la sous-question $\texttt{append(Xs,Zs}^1\texttt{,[a,b])}$; cette sous-question sera l'étiquette du nœud-fils de la racine de l'arbre.[10]

La deuxième étape de la construction de l'arbre de recherche est la réduction de la sous-question $\texttt{append(Xs,Zs}^1\texttt{,[a,b])}$. Il existe deux clauses unifiantes (dont la tête comporte le prédicat `append`), donc le nœud étiqueté par la sous-question à réduire aura deux fils. Le corps de la première clause unifiante (branche de gauche) est vide, ce qui signale que cette branche sera une branche succès. L'unificateur adéquat comporte ici les trois équations car le prédicat `append` est d'arité 3; on a $\texttt{Xs} = \texttt{[]}$, $\texttt{Zs}^1 = \texttt{Bs}^2$ et $\texttt{[a,b]} = \texttt{Bs}^2$. Cet unificateur égalise la sous-question $\texttt{append(Xs,Zs}^1\texttt{,[a,b])}$ et la tête de la clause unifiante $\texttt{append(Xs,Bs}^2\texttt{,Bs}^2\texttt{)}$. La solution correspondant à cette branche-succès est obtenue au moyen des unificateurs étiquetant cette branche; on obtient ainsi $\texttt{Xs} = \texttt{[]}$, ce qui est correct : la liste vide est bien un préfixe de la liste `[a,b]`.

La troisième étape consiste à construire le second fils de la sous-question $\texttt{append(Xs,Zs}^1\texttt{,[a,b])}$. La tête de la clause unifiante s'écrit maintenant $\texttt{append([A}^2\texttt{|As}^2\texttt{],Bs}^2\texttt{,[A}^2\texttt{|Cs}^2\texttt{])}$; l'unificateur comporte les trois équations $\texttt{Xs} = \texttt{[A}^2\texttt{|As}^2\texttt{]}$, $\texttt{Zs}^1 = \texttt{Bs}^2$ et $\texttt{[a,b]} = \texttt{[A}^2\texttt{|Cs}^2\texttt{]}$. Le corps de la clause unifiante est $\texttt{append(As}^2\texttt{,Bs}^2\texttt{,Cs}^2\texttt{)}$ qui, vu l'unificateur, se réduit à la sous-question $\texttt{append(As}^2\texttt{,Bs}^2\texttt{,[b])}$.[11] Ce prédicat étiquètera donc le second fils. Les étapes ultérieures construisent la descendance de ce dernier nœud, ce qui, comme l'indique la figure, produit deux solutions supplémentaires : `Xs = [a]` et `Xs = [a,b]`. On observe aussi que la dernière branche, c'est-à-dire la branche la plus à droite, est une branche-échec. La raison en est que la troisième équation du dernier unificateur est insoluble. En effet, l'égalité $\texttt{[]} = \texttt{[A}^4\texttt{|Cs}^4\texttt{]}$ est impossible puisque le membre de gauche désigne une liste sans élément tandis que le membre de droite désigne une liste comportant au moins un élément.

Remarque. Nous avons mentionné que les variables d'une clause étaient, de manière implicite, quantifiées universellement. Précisons que la portée d'une variable est la clause qui la contient; on peut donc, par exemple, remplacer la clause :

```
prefix(Xs,Ys) :- append(Xs,Zs,Ys).
```

par la clause :

[10] Les variables apparaissant dans une question ne sont jamais renommées, c'est-à-dire affectées d'un exposant; en effet, ces variables ont été choisies par l'utilisateur du système, donc le système les emploiera pour communiquer les résultats du calcul à l'utilisateur. En Prolog, les résultats d'un calcul sont les valeurs des variables (si elles existent) qui permettent de répondre affirmativement à la question de l'utilisateur. Les variables des clauses sont au contraire renommées à chaque utilisation au sein d'une même branche de l'arbre. On peut cependant avoir les mêmes noms dans des branches distinctes car deux branches distinctes n'interfèrent pas.

[11] De $\texttt{[a,b]} = \texttt{[A}^2\texttt{|Cs}^2\texttt{]}$ on déduit $\texttt{a} = \texttt{A}^2$ et $\texttt{[b]} = \texttt{Cs}^2$.

```
prefix(As,Bs) :- append(As,Cs,Bs).
```

qui est équivalente, mais pas par la clause :

```
prefix(As,Bs) :- append(Cs,Ds,Es).
```

qui est trop générale, ni par la clause :

```
prefix(As,Bs) :- append(As,Bs,Bs).
```

qui est trop restrictive. D'autre part, les variables d'une question sont quantifiées existentiellement; dans l'exemple développé plus haut, la question `prefix(Xs,[a,b])` s'interprète en "existe-t-il une liste `Xs` qui soit un préfixe de la liste `[a,b]` ?". Prolog répond de manière affirmative, constructive et exhaustive à cette question, en proposant successivement pour `Xs` les valeurs `[]`, `[a]` et `[a,b]`.

Au premier abord, on constate que la construction de l'arbre de recherche est un processus fastidieux mais élémentaire. On pourrait décrire le noyau d'un système Prolog en spécifiant de manière plus formelle la manière dont les étiquettes des nœuds et des arcs sont produites, ainsi que le mécanisme par lequel les solutions sont extraites et fournies à l'utilisateur. L'ordre dans lequel les manipulations sont faites est également important; une branche de gauche doit être terminée (par un succès ou par un échec) avant que la construction d'une branche plus à droite soit entamée. Cela a pour conséquence qu'une branche infinie provoque la non-terminaison du programme, et la non-production des résultats correspondant aux branches plus à droite éventuelles. Dans la suite du présent chapitre, nous illustrons par des exemples simples la *sémantique logique*, ou *déclarative*, de Prolog, c'est-à-dire le lien logique existant entre les questions posées par un utilisateur et les réponses que Prolog fournit au moyen du mécanisme de construction d'arbre que nous venons de décrire. D'autres exemples plus élaborés seront introduits au chapitre suivant; ils permettront de mieux illustrer l'intérêt de la programmation logique en intelligence artificielle.

11.2.2. *Sémantique déclarative*

La sémantique opérationnelle de Prolog, esquissée au paragraphe précédent, permet de comprendre comment le système produit les réponses aux questions qui lui sont soumises, mais la portée de ces réponses, c'est-à-dire leur lien logique avec la question et le programme, se comprend mieux par le biais de la sémantique déclarative. En effet, d'un point de vue opérationnel, une clause telle que "`A :- B,C.`" traduit l'idée "pour obtenir `A`, on doit obtenir (au préalable) `B` et `C`". Le point de vue logico-déclaratif est "`A` est vrai si `B` et `C` sont vrais". La clause Prolog est donc équivalente à la clause logique notée $(\texttt{B} \wedge \texttt{C}) \Rightarrow \texttt{A}$, ou encore $\texttt{A} \vee \neg\texttt{B} \vee \neg\texttt{C}$.

Du point de vue déclaratif, le programme décrit précédemment apparaît comme une théorie, c'est-à-dire un ensemble de postulats relatifs à une

structure, l'ensemble des listes en l'occurrence, munie d'opérations algébriques, ici la concaténation et le préfixe. Chaque branche-succès de l'arbre de recherche peut être vue comme une preuve d'un théorème correspondant à la solution associée à cette branche. Par exemple, la branche se terminant sur la solution `Xs = [a]` peut être lue, de bas en haut, comme une démonstration du fait que la liste `[a]` est un préfixe de la liste `[a,b]`. Nous reproduisons cette preuve sous une forme plus classique.

1. `append([],[Bs`^{3}`],[Bs`^{3}`])` ; (clause de base – `append`)
2. `append([],[b],[b])` ; (instantiation, ligne 1)
3. `append([A`^{2}`|As`^{2}`],Bs`^{2}`,[A`^{2}`|Cs`^{2}`])` si `append(As`^{2}`,Bs`^{2}`,Cs`^{2}`)` ; (clause inductive – `append`)
4. `append([a],[b],[a,b])` if `append([],[b],[b])` ; (instant., ligne 3)
5. `append([a],[b],[a,b])` ; (déduction, lignes 2 et 4)
6. `prefix(Xs`^{1}`,Ys`^{1}`)` if `append(Xs`^{1}`,Zs`^{1}`,Ys`^{1}`)` ; (clause – `prefix`)
7. `prefix([a],[a,b])` if `append([a],[b],[a,b])` ; (instant., ligne 6)
8. `prefix([a],[a,b])`. (déduction, lignes 5 et 7)

Pour l'arbre de la figure 11.1, on constate aisément que, d'une part, il possède trois branches succès et que, d'autre part, les étiquettes des trois feuilles correspondantes sont précisément les réponses à la question `prefix(Xs,[a,b])` étiquetant la racine de l'arbre. Il n'est pas évident que cette coïncidence entre l'approche opérationnelle et l'approche déclarative se produise aussi dans des cas plus complexes; cette question a donné lieu à toute une théorie dont nous donnons ici quelques éléments.

11.3. Approche théorique de la programmation logique

La programmation logique est un concept extrêmement ambitieux puisque l'idée sous-jacente consiste en un algorithme dont les données sont un ensemble de formules (le "programme" ou les "hypothèses") et une formule (la "question" ou la "conclusion"); cet algorithme détermine si la conclusion est conséquence logique des hypothèses et, si c'est le cas, construit une preuve de ce fait. Il y a bien certaines restrictions : les hypothèses sont des clauses de Horn (ou plutôt, des fermetures universelles de clauses de Horn) et la conclusion est un cube positif (ou plutôt, la fermeture existentielle d'un cube positif)[12] mais on conçoit, vu le théorème de Church affirmant l'indécidabilité du calcul des prédicats

[12] Rappelons qu'un atome est une forme prédicative, c'est-à-dire l'application d'un prédicat d'arité n à n arguments qui sont des termes; une formule est un atome si et seulement si elle ne contient ni connecteur ni quantificateur. Un littéral est un atome (littéral positif) ou la négation d'un atome (littéral négatif). Une clause est une disjonction de littéraux; elle est unitaire si elle comporte un seul littéral. Une clause de Horn est une clause ne comportant aucun littéral positif (clause de Horn négative) ou exactement un littéral positif (clause de Horn définie). Un cube est une conjonction de littéraux; un cube positif est une conjonction de littéraux positifs, c'est-à-dire d'atomes.

(cf. § I.8.6.2), que des limitations sont inévitables; en particulier, l'algorithme peut ne pas se terminer si la conclusion n'est pas conséquence logique des hypothèses.[13] Par ailleurs, la programmation logique fournit une preuve constructive, en ce sens que, si la fermeture existentielle du cube-conclusion est reconnue comme conséquence logique des hypothèses, alors l'algorithme doit aussi fournir des instances appropriées des variables intervenant dans ce cube (en fait, ce sont ces instances qui intéressent l'utilisateur et qui constituent les résultats fournis par l'exécution d'un programme logique).

Les quelques exemples de programmes introduits jusqu'ici, quoique très simples, suffisent à montrer que Prolog n'est pas une réalisation entièrement satisfaisante du concept de programmation logique. Nous avons vu, notamment, qu'un programme Prolog est une liste de clauses et non un ensemble; l'ordre des clauses a un impact parfois important sur le comportement du programme, de même que l'ordre des littéraux au sein des clauses. De même, nous avons vu que certaines exécutions de programmes Prolog pouvaient ne pas se terminer. Ces phénomènes et d'autres du même genre justifieraient une étude générale, théorique, des liens entre une programmation logique idéale d'une part, et les systèmes Prolog réels d'autre part. Cependant, une telle étude sortirait largement du cadre de cet ouvrage (cf. [Lloyd 87] ou [Doets 94]. Dans la suite de ce paragraphe, nous donnons quelques points saillants de la théorie de la programmation logique et de Prolog. Le cas propositionnel étant nettement plus simple que le cas général prédicatif, nous le développerons davantage.

11.3.1. *La programmation logique propositionnelle*

Le problème de la programmation logique propositionnelle consiste à déterminer si une proposition est ou n'est pas conséquence logique d'un ensemble de clauses de Horn définies. Une variante consiste à déterminer l'ensemble des propositions conséquences logiques d'un ensemble de clauses de Horn définies. Rappelons d'abord brièvement la solution à ces problèmes introduite au volume précédent (cf. § II.5.6), qui repose sur l'algorithme de résolution unitaire (positive) représenté à la figure 11.2.

Cet algorithme permet de tester assez efficacement l'inconsistance d'un ensemble S_0 de clauses de Horn propositionnelles.[14] L'algorithme comporte un composant non déterministe de choix. Si plusieurs couples (p, c) peuvent être sélectionnés à un moment donné, un couple arbitraire acceptable est effectivement choisi. Remarquons aussi qu'une exécution de cet algorithme peut s'arrêter *normalement*, quand la condition $\mathbf{F} \in S$ devient vraie, ou

[13] En pratique, d'autres limitations sont à accepter si l'on souhaite pour l'algorithme une vitesse d'exécution "raisonnable"; cela justifie notamment le fait que l'on se restreigne le plus souvent aux clauses de Horn.

[14] Toute clause est ici assimilée à l'ensemble des littéraux dont elle est la disjonction. La clause vide (ensemble vide de littéraux) est naturellement notée $\mathbf{F}$.

$\{S := S_0\}$
Tant que $\mathbf{F} \notin S$ faire
    choisir p et c tels que
        – p est une clause unitaire positive de S,
        – c est une clause de S contenant $\neg p$;
    $r := c \setminus \{\neg p\}$;
    $S := (S \setminus \{c\}) \cup \{r\}$.

Figure 11.2 *Résolution unitaire positive*

s'arrêter *anormalement*, si aucun choix n'est possible pour p et c. On appelle *dérivation* une exécution de l'algorithme de résolution et, comme nous l'avons vu au paragraphe I.4.4, il est commode de représenter les dérivations par des arbres. Une dérivation se terminant par la production de la clause vide est une *réfutation*. Un exemple de réfutation est donné à la figure 11.3; cet exemple prouve l'inconsistance de l'ensemble

$$S = \{t \vee \neg p \vee \neg r\,,\ p \vee \neg r \vee \neg t\,,\ r\,,\ t \vee \neg q\,,\ q\,,\ \neg p \vee \neg q \vee \neg r\}\,.$$

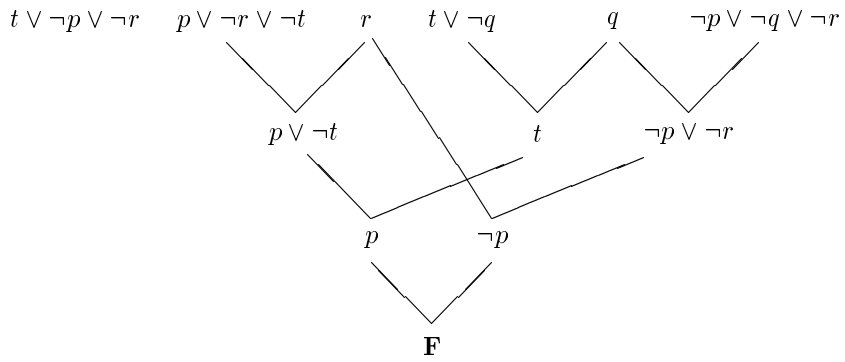


Figure 11.3 *Arbre de réfutation unitaire pour l'ensemble S*

Au paragraphe II.5.6, nous avions démontré que, pour tester si une proposition p est conséquence logique d'un programme logique L (ensemble de clauses de Horn définies), il suffit d'appliquer l'algorithme à l'ensemble $S_0 =_{def} L \cup \{\neg p\}$. L'exécution se termine toujours, normalement si $L \models p$ et anormalement sinon; le temps d'exécution est au pire fonction quadratique du nombre de littéraux intervenant dans S_0. Nous avions aussi prouvé que, si l'algorithme est appliqué à l'ensemble L (toujours consistant), l'exécution se termine toujours anormalement; l'ensemble des propositions conséquences logiques de L sont exactement les propositions qui interviennent comme clauses unitaires dans la valeur finale de l'ensemble S. Ces propositions déterminent

le modèle minimal, ou modèle canonique, du programme logique L.

Nous avions enfin noté que l'algorithme de résolution unitaire n'accorde aucun rôle particulier à la proposition p à tester; sa négation est simplement adjointe au programme logique L. L'algorithme de résolution d'entrée, qui est la version propositionnelle abstraite de l'algorithme Prolog, permet d'exploiter le programme logique de manière plus focalisée, sans jamais perdre la question de vue (figure 11.4). Cet algorithme utilise une variable G, dont la valeur est toujours une clause de Horn négative; initialement, on a $G = G_0 = \{\neg g_1, \ldots, \neg g_n\}$, où $g_1, \ldots, g_n$ est l'ensemble conjonctif des questions.

$\{G = G_0\}$
Tant que $G \neq \mathbf{F}$ faire
choisir p et c tels que
$-\ \neg p \in G$,
$-\ c \in L$ et $p \in c$;
$G := (G \setminus \{\neg p\}) \cup (c \setminus \{p\})$.

Figure 11.4 *Résolution d'entrée*

Naturellement, on ne peut envisager de remplacer la résolution unitaire par la résolution d'entrée que si la seconde jouit des mêmes propriétés que la première. Avant d'étudier cette question, il convient d'observer que les exécutions de l'algorithme de résolution d'entrée, appelées aussi dérivations ou réfutations, peuvent aussi se représenter de manière arborescente. A titre d'exemple, nous montrons à la figure 11.5 que la proposition p est bien conséquence logique du programme logique

$$L = \{\, t \vee \neg p \vee \neg r\,,\ p \vee \neg r \vee \neg t\,,\ r\,,\ t \vee \neg q\,,\ q\,\}\,.$$

On voit que, dans un arbre de réfutation d'entrée, il existe une branche principale unissant le but initial (ici, $\neg p$) à la clause vide. Les branches auxiliaires sont de longueur 1 et unissent une clause d'entrée (d'où le nom de l'algorithme) à un but intermédiaire.

Le résultat suivant garantit que l'algorithme de résolution d'entrée hérite des propriétés importantes de l'algorithme de résolution unitaire. En particulier, il est adéquat et complet : on peut obtenir une réfutation d'entrée de $\neg p$ par rapport au programme logique L si et seulement si p est conséquence logique de L.

Théorème 11.1. L'ensemble $L \cup \{\neg p\}$ admet une réfutation unitaire si et seulement s'il admet une réfutation d'entrée basée sur $\neg p$.

Remarque préliminaire. On montrera comment transformer une réfutation unitaire en réfutation d'entrée; la transformation inverse est laissée au lecteur. La preuve se fait par induction sur le lexique, c'est-à-dire sur l'ensemble des propositions contenues dans le programme logique. On peut aussi se limiter aux ensembles $L \cup \{\neg p\}$ minimaux, c'est-à-dire tels que tout sous-ensemble

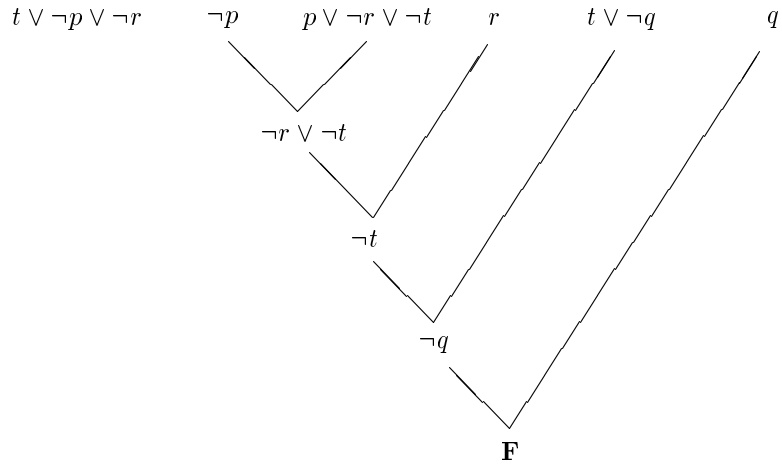


Figure 11.5 *Arbre de réfutation d'entrée pour L et p*

propre est consistant (et donc ne donne jamais lieu à une réfutation). En effet, tout ensemble admet un sous-ensemble minimal, et une réfutation sur le sous-ensemble minimal est aussi une réfutation sur l'ensemble original.

Preuve. Le cas de base est celui où le lexique se réduit à $\Pi = \{p\}$. L'ensemble minimal $L \cup \{\neg p\}$ est nécessairement $\{p, \neg p\}$; il admet une seule réfutation, qui est à la fois unitaire et d'entrée. Supposons maintenant que l'ensemble $L \cup \{\neg p\}$ soit minimal, que le lexique ne se réduise pas à la proposition p et qu'une réfutation unitaire existe. Cela signifie que L comporte au moins une clause unitaire positive q distincte de p; toute autre occurrence de q dans L est nécessairement sous la forme du littéral négatif $\neg q$. On forme alors l'ensemble L' obtenu à partir de L en supprimant la clause q et, dans toutes les clauses qui le contiennent, le littéral $\neg q$. Soit $L'' \subset L'$ tel que $L'' \cup \{\neg p\}$ soit minimal. La réfutation unitaire relative à $L \cup \{\neg p\}$ se transforme aisément en une réfutation unitaire de $L'' \cup \{\neg p\}$, par simple omission des étapes utilisant la clause q et par suppression de toutes les occurrences du littéral $\neg q$. Le lexique de L'' est un sous-ensemble propre du lexique de L d'où, par hypothèse d'induction, il existe une réfutation d'entrée de $L'' \cup \{\neg p\}$ basée sur $\neg p$. En rajoutant le littéral $\neg q$ aux endroits adéquats de cette réfutation, on obtient une dérivation d'entrée basée sur $\neg p$ dont la racine est $\neg q$ et dont les feuilles appartiennent à $L \cup \{\neg p\}$. Comme q est une clause de L, on complète cette dérivation par une dernière étape utilisant cette clause, ce qui conduit à une réfutation d'entrée de $L \cup \{\neg p\}$ basée sur $\neg p$.

11.3.2. *Prolog propositionnel*

L'algorithme de Prolog est une version concrète de l'algorithme de résolution d'entrée. Les clauses sont représentées par des listes de littéraux et le programme logique L est une liste de clauses. Les choix de p et c sont imposés par une stratégie très simple; p est nécessairement la proposition correspondant au premier littéral du but courant et c est la première clause de L convenable qui n'a pas encore été utilisée. En effet, le système Prolog essaiera, dans l'ordre où elles se présentent dans la liste L, toutes les clauses dont la tête est p.

11.4. Programmer en Prolog

Avant de considérer plus avant l'intérêt de la logique et de Prolog en intelligence artificielle, il est intéressant d'écrire quelques petits programmes; ils suffisent à mettre en évidence certains principes et surtout la concision et le grand pouvoir d'expression de Prolog: des problèmes d'apparence complexe peuvent être résolus simplement. On voit aussi que la programmation en Prolog, simple dans son principe, requiert une grande attention; deux programmes équivalents en apparence peuvent se comporter très différemment.

11.4.1. *Listes, préfixes, suffixes, sous-listes et sous-ensembles*

Nous continuons ici l'étude des listes et des divers concepts associés. Au lieu de dériver le prédicat `prefix` du prédicat `append`, on peut le définir directement, par une clause de base et une clause récursive. D'une part, la liste vide est un préfixe de toute liste et, d'autre part, `[X|Xs]` est un préfixe de `[X|Ys]` si `Xs` est un préfixe de `Ys`. La notion de suffixe se définit de manière analogue. Enfin, on définit une sous-liste de `Xs` comme une liste d'éléments consécutifs de `Xs`. On voit immédiatement qu'une sous-liste est le suffixe d'un préfixe de la liste de départ, ce qui permet une programmation très simple. A première vue, les deux variantes `sublist1` et `sublist2` de la figure 11.6 sont équivalentes.

Nous utilisons d'abord `sublist2` pour construire les sous-listes de la liste `[a,b]`. L'arbre de recherche correspondant est représenté à la figure 11.7. Nous observons que toutes les sous-listes sont obtenues mais la sous-liste vide est construite trois fois. Prolog fournit les résultats à l'utilisateur dès que ceux-ci sont obtenus; cela signifie que, durant la construction de gauche à droite de l'arbre de recherche, l'utilisateur reçoit successivement les solutions suivantes:

```
[], [a], [], [a,b], [b], [].
```

Si l'utilisateur réclame une solution supplémentaire, le système affiche "`NO`" et l'exécution s'arrête. Du point de vue de la logique, ce comportement est parfaitement correct. Par analogie avec le vocabulaire employé pour les systèmes axiomatiques, on peut dire que le programme est adéquat (seules


```
suffix(Xs,Xs).
suffix(Xs,[Y|Ys]) :- suffix(Xs,Ys).

prefix([],Xs).
prefix([X|Xs],[X|Ys]) :- prefix(Xs,Ys).

sublist1(Xs,Ys) :- suffix(Xs,Zs), prefix(Zs,Ys).

sublist2(Xs,Ys) :- prefix(Zs,Ys), suffix(Xs,Zs).
```

Figure 11.6 *Deux variantes du prédicat* `sublist`

des réponses correctes sont fournies) et complet (toutes les réponses correctes sont fournies). Néanmoins, d'un point de vue pragmatique, on préférerait que chaque résultat soit fourni une seule fois.[15]

Nous utilisons maintenant la variante `sublist1` pour vérifier si `[a]` est une sous-liste de `[b]`. Nous observons que toutes les branches finies de l'arbre de recherche (figure 11.8) se terminent par "`NO`", ce qui est normal. Par contre, on observe la présence d'une branche infinie. Ce phénomène est gênant: Prolog ne pourra répondre à la question de l'utilisateur, puisqu'une réponse négative n'est logiquement et opérationnellement possible que quand la construction de l'arbre de recherche est achevée et n'a pas produit de réponse positive. En fait, la variante `sublist1` est inadéquate dans ce contexte même si, à proprement parler, elle n'est pas incorrecte. Un autre problème survient si `sublist1` est utilisé pour construire toutes les sous-listes d'une liste donnée. Prolog donne bien toutes les réponses (pas dans le même ordre qu'avec le programme `sublist2`),[16] mais, et cela est gênant, Prolog n'arrête pas son exécution; une branche infinie et inutile se trouve à la droite de l'arbre de recherche. Curieusement, `sublist2` fonctionne mieux, au moins pour vérifier que `[a]` n'est pas une sous-liste de `[b]`, ainsi que le montre l'arbre de recherche

[15] On peut écrire une variante remplissant cette exigence. Les codes pour `prefix` et `suffix` ne sont pas modifiés mais on définit `sublist3` par les deux clauses suivantes:

```
sublist3([],Xs).
sublist3([X|Xs],Ys) :- prefix(Zs,Ys), suffix([X|Xs],Zs).
```

Les solutions sont produites et fournies à l'utilisateur dans un ordre précis; par exemple, les solutions de:

```
sublist3(Xs,[a,b,c,d]).
```

sont `[]`, `[a]`, `[a,b]`, `[b]`, `[a,b,c]`, `[b,c]`, `[c]`, `[a,b,c,d]`, `[b,c,d]`, `[c,d]`, `[d]`.

[16] Les sous-listes de `[a,b,c]` sont fournies par `sublist1` dans l'ordre:
`[]`, `[a]`, `[a,b]`, `[a,b,c]`, `[]`, `[b]`, `[b,c]`, `[]`, `[c]`, `[]`;
alors que, par `sublist2`, elles sont fournies dans l'ordre:
`[]`, `[a]`, `[]`, `[a,b]`, `[b]`, `[]`, `[a,b,c]`, `[b,c]`, `[c]`, `[]`.

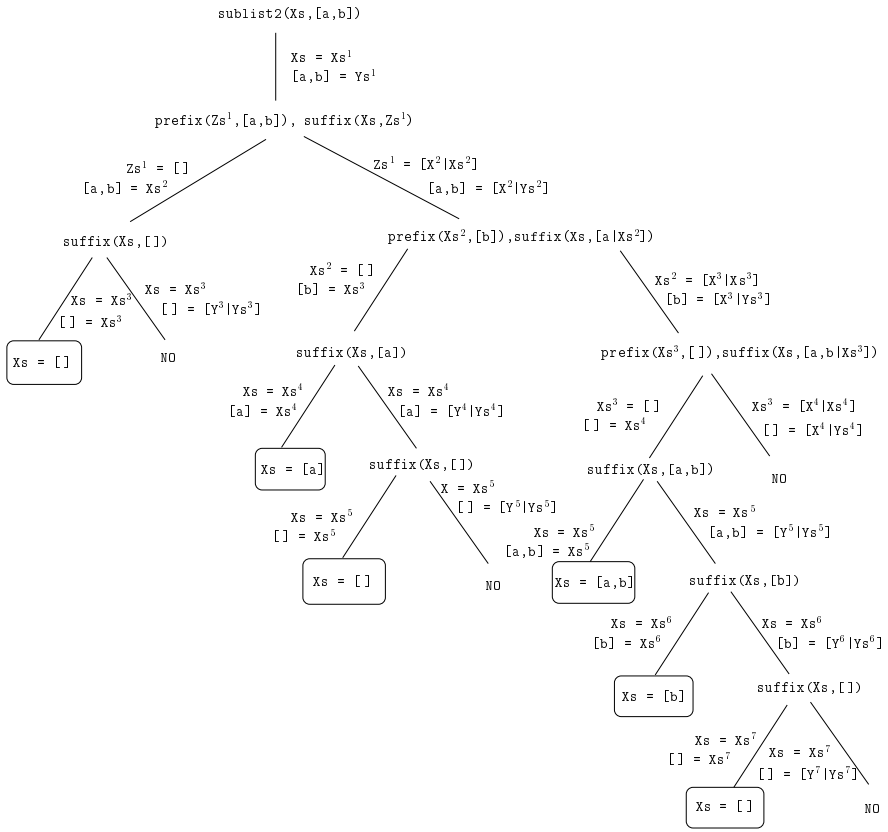


Figure 11.7 *Un arbre de recherche fini pour un calcul de sous-listes*

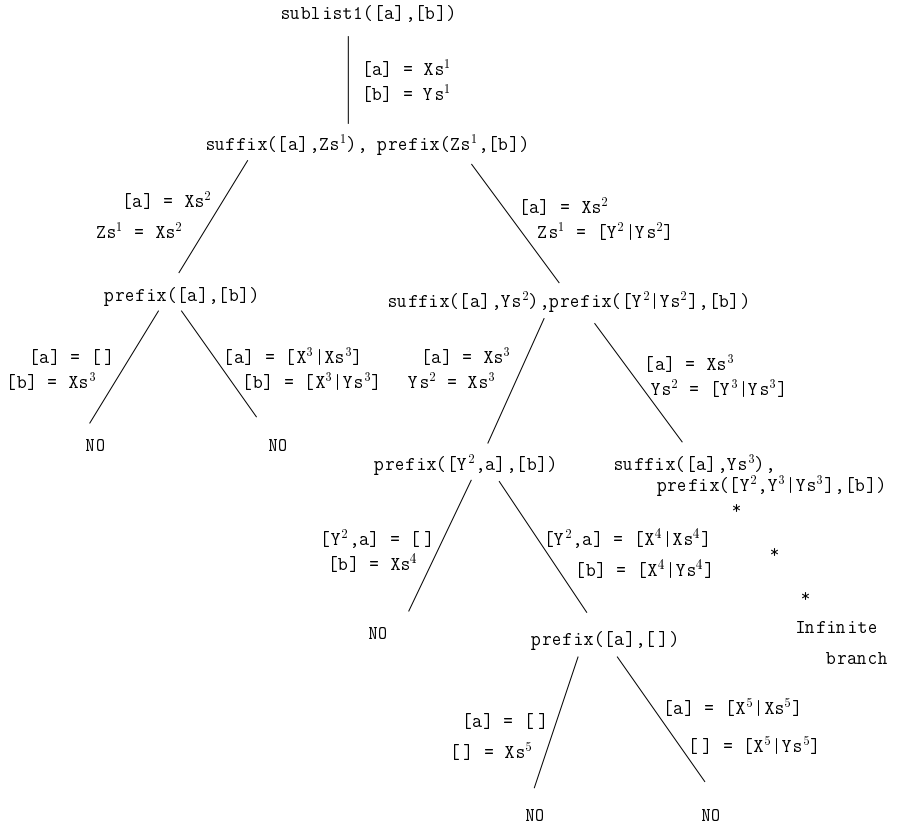


Figure 11.8 *Un arbre de recherche infini pour un calcul de sous-listes*

de la figure 11.9. Cet arbre est très différent du précédent, et fini. Une première conclusion est que l'ordre des éléments dans le corps d'une clause Prolog peut être très important.

Le fait que le comportement de Prolog dépende de l'ordre des prédicats au sein des corps des clauses pourrait paraître surprenant; en effet, d'après la sémantique déclarative, il ne devrait pas y avoir de différence entre les deux clauses "`A :- B,C.`" et "`A :- C,B.`". L'explication est que Prolog tient pour vraies seulement les propositions démontrables au moyen d'une stratégie spécifique, et cette stratégie n'est pas nécessairement complète. Notons aussi que, du point de vue déclaratif, l'ordre des clauses elles-mêmes dans le programme devrait être indifférent. En fait, l'ordre des clauses détermine celui des branches des arbres de recherche et influe donc sur l'ordre dans lequel Prolog produit et fournit ses réponses. Cela peut devenir gênant si l'arbre comporte une ou plusieurs branches infinies; seule la partie de l'arbre se trouvant à gauche de la première branche infinie sera explorée, ce qui peut induire la perte de certaines, voire de toutes les réponses déclarativement attendues. Différents exemples de ce phénomène seront rencontrés dans la suite.

On utilise souvent des listes pour représenter d'autres structures de données. Par exemple, il est fréquent de modéliser un ensemble par une liste sans répétition, ou encore par une liste triée sans répétition.[17] Il est facile d'écrire un prédicat `subset_1` qui reconnaît et/ou construit les sous-ensembles d'un ensemble :

```
subset_1([],[]).
```

[17] Le tri alourdit la construction de la structure mais peut alléger son exploration; de plus, il permet d'obtenir une propriété intéressante : chaque ensemble est représenté par une et une seule liste triée sans répétition.

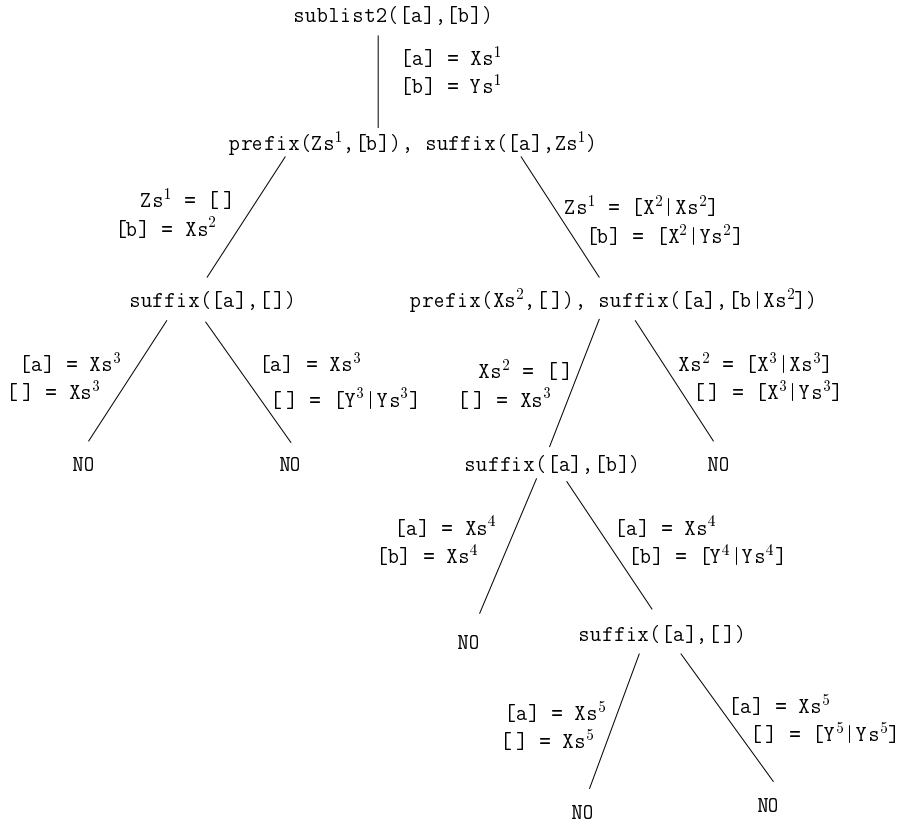


Figure 11.9 *Un arbre de recherche fini infructueux pour un calcul de sous-listes*

```
subset_1([X|Xs],[X|Zs]) :- subset_1(Xs,Zs).
subset_1(Xs,[X|Zs]) :- subset_1(Xs,Zs).
```

Ce prédicat construit correctement tous les sous-ensembles; par exemple, on a :

```
?- subset_1(Xs,[a,b,c]).
Xs = [a, b, c] ;
Xs = [a, b] ;
Xs = [a, c] ;
Xs = [a] ;
Xs = [b, c] ;
Xs = [b] ;
Xs = [c] ;
Xs = [] ;
No
```


Il peut aussi être utilisé pour reconnaître les sous-ensembles mais il ne reconnaîtra pas les listes avec répétitions et/ou inversions, comme le montre la session suivante :

```
?- subset_1([1,3],[1,2,3,4]).
Yes
?- subset_1([1,1,3],[1,2,3,4]).
No
?- subset_1([3,1],[1,2,3,4]).
No
```

Si on souhaite que les répétitions et inversions soient reconnues, on pourra utiliser

```
member(X,[X|Xs]).
member(X,[Y|Xs]) :- member(X,Xs).

subset_2([],Xs).
subset_2([Y|Ys],Xs):- member(Y,Xs), subset_2(Ys,Xs).
```

On a alors :

```
?- subset_2([1,1,3],[1,2,3,4]).
Yes

?- subset_2([3,1],[1,2,3,4]).
Yes
```

Par contre, `subset_2` ne peut pas être utilisé pour la génération des sous-ensembles :

```
?- subset_2(Xs,[1,2,3,4]).
Xs = [] ;
Xs = [1] ;
Xs = [1, 1] ;
Xs = [1, 1, 1] ;
...
```

Remarque. Le prédicat auxiliaire `member` est très utile; il sera abondamment employé au chapitre suivant.

Le prédicat `subset_1` construit les sous-ensembles un à un mais on programme aisément le prédicat `subset_list` qui permet la construction de la liste complète des sous-ensembles d'un ensemble donné :

```
subset_list([],[[]]).
subset_list([X|Xs],Zss) :- subset_list(Xs,Uss),
                           put_in_all(X,Uss,Vss),
                           append(Vss,Uss,Zss).
put_in_all(A,[],[]).
```

```
put_in_all(A,[Bs|Bss],[[A|Bs]|Css]) :- put_in_all(A,Bss,Css).
```

L'exemple suivant montre comment ce prédicat et le prédicat auxiliaire `put_in_all` fonctionnent :

```
?- put_in_all(x,[[a,b],[],[c,d,e]],Xss).
Xss = [[x, a, b], [x], [x, c, d, e]]
?- subset_list([a,b,c],Xss).
Xss = [[a, b, c], [a, b], [a, c], [a], [b, c], [b], [c], []]
```

11.4.2. *Machines abstraites et fonctions récursives*

Les mécanismes Prolog que nous venons de décrire et d'illustrer peuvent sembler rudimentaires mais cette apparence est trompeuse; ils permettent en fait de programmer tout ce qui est programmable, bien souvent de manière simple et concise (mais pas nécessairement très efficace). Nous démontrons ceci en recréant en Prolog les machines abstraites, y compris la machine de Turing.[18] Nous montrons aussi comment les fonctions récursives introduites par Gödel (cf. § I.2.3.4) peuvent être simulées en Prolog.

Remarque. Nous ne voulons pas introduire ou rappeler ici la théorie des machines abstraites, qui est une branche relativement distincte de la logique. Toutefois, les exemples donnés dans cette section ne requièrent pas de connaissance préalable de cette théorie.

11.4.2.1. *Automates*

Les automates finis ou machines à états finis[19] constituent un mécanisme de calcul très élémentaire, ce qui ne les empêchent pas d'avoir de nombreuses applications, tant en informatique théorique qu'en programmation réelle. L'automate fini représenté à la figure 11.10 comporte les deux états q_0 (l'état initial, repéré par une grande pointe de flèche) et l'état q_1. (L'état q_0 est aussi final, ce qu'indique le double cercle.) Cet automate reconnaît certains mots construits au moyen de l'alphabet $\{a, b\}$.[20] Lire a dans l'état q_0 conduit en l'état q_1 et lire b en l'état q_1 conduit en l'état q_0. On en déduit immédiatement que le langage reconnu par l'automate, c'est-à-dire l'ensemble des mots acceptés par cet automate, est :

$$(ab)^* =_{def} \{\varepsilon, ab, abab, ababab, \dots\},$$

où ε désigne le mot vide.

[18] Elles sont analogues aux machines de Post décrites au paragraphe I.8.5.2.

[19] Plus précisément : les machines dont l'espace des états est un ensemble fini.

[20] Les mots sont simplement des suites de lettres de l'alphabet. Un langage est un ensemble de mots. Il existe une infinité non dénombrable de langages sur l'alphabet $\{a, b\}$; les automates et autres machines abstraites permettent de reconnaître et de construire les langages les plus intéressants.

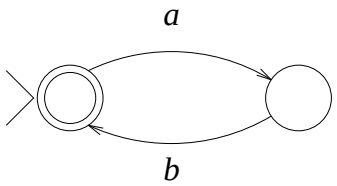


Figure 11.10 *Un automate fini acceptant le langage* $(ab)^*$

Cet automate se représente facilement en Prolog par quatre clauses qui sont des faits fondamentaux. De plus, le comportement de cet automate (ou de n'importe quel automate fini) est simulé par un interpréteur Prolog composé de trois clauses élémentaires (voir figure 11.11).

```
/* Version Prolog d'un automate acceptant le langage (ab)* */
init(q0).
fin(q0).
delta(q0,a,q1).
delta(q1,b,q0).
/* Interprétation d'un automate représenté en Prolog */
acc(Xs) :- init(Q), acc(Q,Xs).
acc(Q,[]) :- fin(Q).
acc(Q,[X|Xs]) :- delta(Q,X,Q1), acc(Q1,Xs).
```

Figure 11.11 *Automate fini, représentation et simulation en Prolog*

Les automates finis, quoique très simples, permettent de résoudre des problèmes intéressants et de reconnaître une vaste classe (dénombrable) de langages, dits réguliers. Il existe toutefois des langages non réguliers très importants, comme le langage des palindromes, c'est-à-dire des mots identiques à leur "miroir", tels ε, b, bb, aba, $bbabab$, etc). Le langage des palindromes appartient à la famille des langages hors-contexte, qui est un sur-ensemble de la famille des langages réguliers. Les langages hors-contexte sont exactement ceux reconnus par les automates à pile non déterministes. Ces automates sont obtenus en ajoutant aux automates finis une mémoire non bornée, structurée comme une pile ("premier entré, dernier sorti"). Chaque transition de l'automate à pile peut provoquer l'ajout d'une lettre sur la pile, ou le retrait d'une lettre. Un équivalent Prolog d'un automate à pile reconnaisseur du langage des palindromes sur l'alphabet $\{a, b\}$ est illustré à la figure 11.12.[21] On observe que l'interpréteur Prolog pour les automates à pile est très semblable

[21] La lecture de la partie gauche du palindrome correspond à une succession de transitions $q_0 \to q_0$ (première clause); toute lettre lue est aussi empilée. L'automate doit détecter le milieu du palindrome (choix non déterministe), ce qui le conduit en l'état q_1, (deuxième δ-clause pour un palindrome pair, troisième pour un palindrome impair). L'automate

à celui utilisé plus haut pour les automates finis.

```
/* Automate à pile pour le langage des palindromes sur {a,b} */
init(q0).
fin(q1).
delta(q0,X,S,q0,[X|S]).
delta(q0,X,S,q1,[X|S]).
delta(q0,X,S,q1,S).
delta(q1,X,[X|S],q1,S).
/* Interprétation */
acc(Xs) :- init(Q), acc(Q,Xs,[]).
acc(Q,[],[]) :- fin(Q).
acc(Q,[X|Xs],S) :- delta(Q,X,S,Q1,S1), acc(Q1,Xs,S1).
```

Figure 11.12 *Automate à pile, représentation et simulation*

Les automates à pile sont plus expressifs que les automates finis, surtout si, comme ici, le non-déterminisme est autorisé. Néanmoins, certains langages restent hors de portée des automates à pile, ce qui signifie qu'aucun automate à pile ne peut reconnaître exactement ces langages-là. Par exemple, il est facile de concevoir un automate à pile acceptant le langage :

$$\bigcup_{n\in\mathbb{N}} a^n b^n \;=\; \{\varepsilon, ab, aabb, aaabbb, \dots\}\,,$$

mais aucun automate à pile ne peut reconnaître le langage :

$$\bigcup_{n\in\mathbb{N}} a^n b^n c^n \;=\; \{\varepsilon, abc, aabbcc, aaabbbccc, \dots\}\,,$$

défini sur l'alphabet $\{a, b, c\}$. Ceci est dû à la restriction qui pèse sur le mode d'accès aux éléments d'une pile : on doit nécessairement retirer le dernier élément placé sur la pile avant d'accéder aux autres éléments. Cette restriction disparaît si l'on dispose d'une seconde pile, où les éléments retirés de la première pile peuvent être entreposés,[22] et les automates à deux piles ont la même puissance d'expression que les machines de Turing ou de Post (cf. I.8.5.2). Une machine de Turing est un automate fini augmenté d'une mémoire non bornée organisée en un ruban. Le mode d'accès aux éléments d'un ruban est séquentiel, mais il n'est pas nécessaire de retirer un élément avant d'accéder au suivant. La figure 11.14 représente une machine de Turing reconnaissant le langage :

consomme ensuite la partie droite du palindrome, en vidant la pile et en vérifiant l'égalité de la lettre lue avec la lettre dépilée (quatrième δ-clause).

[22] A titre d'exercice, le lecteur peut concevoir un automate à une pile pour le langage $\{a^n b^n : n \in \mathbb{N}\}$ et un automate à deux piles pour le langage $\{a^n b^n c^n : n \in \mathbb{N}\}$.

$$\bigcup_{n\in\mathbb{N}} a^n b^n \;=\; \{\varepsilon, ab, aabb, aaabbb, \dots\}\,,$$

ainsi qu'un interpréteur Prolog pour ce type de machines abstraites. Le terme `config(Q,Ds,Head,Fs)` représente la configuration dont l'état est `Q`, la partie gauche du ruban est la liste `Ds` inversée, la cellule-mémoire repérée par la tête de lecture est `Head` et la partie droite du ruban est `Fs`. La machine utilise des symboles auxiliaires pendant le calcul.

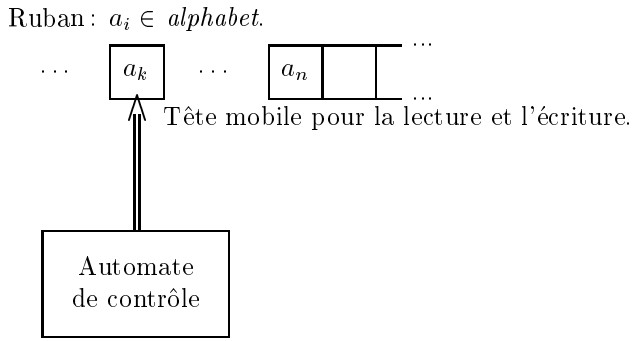


Figure 11.13 *Architecture d'une machine de Turing*

11.4.2.2. *Fonctions récursives*

Les machines de Turing constituent un outil théorique précieux mais ne conviennent guère pour effectuer des calculs réels. Les fonctions récursives, introduites par Gödel pour des raisons également théoriques, sont en apparence plus proches du monde numérique. Nous rappelons ici leur définition et montrons que, une fois de plus, la simulation de ces fonctions en Prolog est immédiate. Il existe trois familles de fonctions récursives de base, à savoir les fonctions nulles, la fonction successeur et les projections. Elles sont résumées dans le tableau suivant.

$(x_1, \dots, x_n) \mapsto 0$: fonction zéro d'arité n;
$x \mapsto x + 1$: fonction successeur;
$(x_1, \dots, x_n) \mapsto x_i$: $i$ème projection d'arité n.

On obtient des fonctions récursives plus complexes en utilisant trois mécanismes de combinaison, qui sont la composition, la récursion primitive et la minimisation. La composition permet de définir une fonction f en termes de fonctions récursives $g_1, \dots, g_m$ et h définies au préalable, selon le schéma suivant :

$$f(x_1, \dots, x_n) \;=_{def}\; h(g_1(x_1, \dots, x_n), \dots, g_m(x_1, \dots, x_n))\,.$$

```
/* Exemple de machine de Turing codée en Prolog */
initial(q0).            final(q4).

delta(q0,a,q1,x,right).  delta(q0,y,q3,y,right).
delta(q1,a,q1,a,right).  delta(q1,b,q2,y,left).
delta(q1,y,q1,y,right).  delta(q2,a,q2,a,left).
delta(q2,x,q0,x,right).  delta(q2,y,q2,y,left).
delta(q3,y,q3,y,right).  delta(q3,blanc,q4,blanc,right).

/* Interpréteur Prolog pour machines de Turing */
accept_MT([]) :- initial(Q), accept(config(Q,[],blanc,[])).
accept_MT([A|As]) :- initial(Q), accept(config(Q,[],A,As)).

accept(config(Q,Ds,Head,Fs)) :- final(Q).
accept(config(Q,Ds,Head,Fs)) :-
   next_config(config(Q,Ds,Head,Fs), config(Qn,Dsn,Headn,Fsn)),
   accept(config(Qn,Dsn,Headn,Fsn)).

next_config(config(Q,Ds,Head,Fs), config(Qn,Dsn,Headn,Fsn)) :-
   delta(Q,Head,Qn,Symb,Act),
   modif(config(Q,Ds,Head,Fs), Symb, Act, config(Qn,Dsn,Headn,Fsn)).

modif(config(Q,Ds,Head,[]),Symb,right,config(Qn,[Symb|Ds],blanc,[])).
modif(config(Q,Ds,Head,[F|Fs]),Symb,right,config(Qn,[Symb|Ds],F,Fs)).
modif(config(Q,[D|Ds],Head,Fs),Symb,left,config(Qn,Ds,D,[Symb|Fs])).
```

Figure 11.14 *Machine de Turing et interpréteur*

La récursion primitive permet de définir une fonction f en termes de fonctions récursives g et h définies au préalable, selon le schéma suivant :

$$f(x_1,\ldots,x_n,0) = h(x_1,\ldots,x_n)\,,$$
$$f(x_1,\ldots,x_n,y+1) = g(x_1,\ldots,x_n,y,f(x_1,\ldots,x_n,y))\,.$$

La minimisation permet de définir une fonction f en termes de la fonction récursive g définie au préalable, selon le schéma suivant :

$$f(x_1,\ldots,x_n) = \mu y.[g(x_1,\ldots,x_n,y) = 0]\,,$$

où $\mu y.p(y)$ désigne le plus petit entier naturel y, s'il existe, tel que $p(y)$ soit vrai. Le schéma de minimisation permet la définition de fonctions partielles, contrairement aux deux autres schémas. Les fonctions obtenues sans l'aide du schéma de minimisation sont les fonctions primitives récursives; elles sont toujours totales.

Le langage Prolog pur (au contraire des systèmes Prolog réels) ne possède pas de primitives arithmétiques, puisque l'unique type de donnée est le terme. On peut néanmoins représenter les nombres par des termes particuliers et convenir, par exemple, que la suite des entiers naturels $0, 1, 2, 3, \ldots$ est représentée par la suite des termes `o`, `s(o)`, `s(s(o))`, `s(s(s(o)))`, ... Nous convenons aussi de représenter une fonction f d'arité n par un prédicat `f_` d'arité n+1, le fait `f_(X1, ... ,Xn,Y)` représentant l'égalité $f(x_1,\ldots,x_n) = y$. A la fonction nulle d'arité n est donc associé le prédicat `zero_`, défini par le fait :

```
zero_(X1,...,Xn,o).
```

On définit de même le prédicat associé à la $j$ième projection d'arité n :

```
pr_j(X1,...,Xn,Xj).
```

A la fonction successeur correspond un prédicat binaire :

```
succ_(X,s(X)).
```

La clause suivante modélise le mécanisme de composition :

```
f_(X1,...,Xn,Y) :-
      g_1(X1,...,Xn,Y1),...,g_m(X1,...,Xn,Ym),
      h_(Y1,...,Ym,Y).
```

Deux clauses permettent de simuler la récursion primitive :

```
f_(X1,...,Xn,o,Z) :- h_(X1,...,Xn,Z).
f_(X1,...,Xn,s(Y),Z) :- f_(X1,...,Xn,Y,U),
                        g_(X1,...,Xn,Y,U,Z).
```

Enfin, voici le programme de trois clauses réalisant la minimisation :

```
f_(X1,...,Xn,Y) :- g_(X1,...,Xn,o,U),r_(X1,...,Xn,o,U,Y).
r_(X1,...,Xn,Y,o,Y).
r_(X1,...,Xn,Y,s(V),Z) :- g_(X1,...,Xn,s(Y),U),
                          r_(X1,...,Xn,s(Y),U,Z).
```

Le prédicat `r_` à (n+3) arguments est associé à une fonction auxiliaire r d'arité (n+2) qui peut être spécifiée comme suit :

Si $g(x_1,\ldots,x_n,\xi) > 0$ *pour tout* $\xi = 0, 1, \ldots, y-1$
et si $g(x_1,\ldots,x_n,y) = u$,
alors $r(x_1,\ldots,x_n,y,u) = \inf\{\zeta : \zeta \geq y \wedge g(x_1,\ldots,x_n,\zeta) = 0\}$.

On voit que la valeur $r(x_1,\ldots,x_n,y,u)$ n'est pas définie si $u > 0$ et $g(x_1,\ldots,x_n,\zeta) > 0$ pour tout $\zeta > y$. Quand c'est le cas, l'évaluation de la question `r_(X1,...,Xn,Y,U,Z)` ne se termine pas.

Il est facile de programmer dans ce cadre les fonctions arithmétiques habituelles; on a notamment :

```
add(X,o,X).
add(X,s(Y),s(Z)) :- add(X,Y,Z).

mult(X,o,o).
mult(X,s(Y),U) :- mult(X,Y,Z), add(Z,X,U).
```

Ces programmes ne constituent naturellement pas un moyen très efficace et très commode de calculer numériquement, mais il est intéressant d'observer que les mécanismes de base de Prolog, en dépit de leur simplicité, permettent de calculer tout ce qui est calculable.

11.4.3. *Calcul symbolique*

Prolog n'est pas vraiment adapté au calcul numérique mais convient très bien pour le calcul symbolique. Pour illustrer ce fait, nous considérons une application classique, la transformation d'expressions arithmétiques ordinaires en expressions préfixées ou postfixées ("notation polonaise", directe ou inverse). Pour simplifier le problème, nous supposons que les opérations arithmétiques sont binaires et que les parenthèses ne sont jamais omises. Une expression arithmétique, telle que :

$$((2 * (9 - (3 + 4))) + 5),$$

est en fait une notation linéaire désignant un arbre dont chaque nœud interne est un opérateur; les deux fils d'un opérateur sont naturellement les opérandes. L'arbre correspondant à l'expression ci-dessus est représenté à la figure 11.15. Comme nous n'avons pas introduit les primitives Prolog

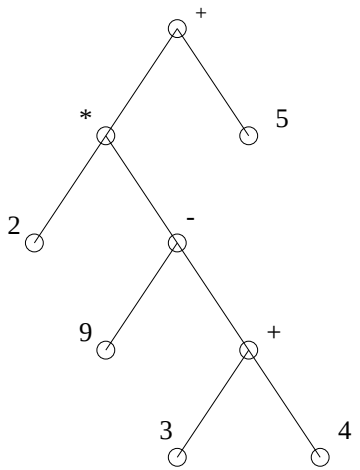


Figure 11.15 *Une expression arithmétique*

permettant la manipulation de termes quelconques, nous supposons qu'une expression arithmétique est représentée par un $\mathbb{N}$-arbre. Un $\mathbb{N}$-arbre atomique est simplement un nombre.[23] Un $\mathbb{N}$-arbre composé est une liste à trois éléments `[X,Y,Z]` où `X` est un opérateur arithmétique[24] et `Y` et `Z` sont les $\mathbb{N}$-arbres représentant les opérandes de gauche et de droite, respectivement. Le $\mathbb{N}$-arbre associé à l'expression arithmétique de la figure 11.15 est donc :

[23] En Prolog pur, les nombres sont des termes atomiques; ils acquièrent une signification spécifiques en Prolog réel, grâce aux opérateurs arithmétiques.

[24] toujours sans signification spécifique en Prolog pur.


```
[+,[*,2,[-,9,[+,3,4]]],5].
```

La liste des symboles correspondant à la forme usuelle, infixée de cette expression est :

```
[(,(,2,*,(,9,-,(,3,+,4,),),),+,5,)].
```

Cette forme semble commode mais la notation infixée a l'inconvénient de requérir des parenthèses, ce que permettent d'éviter les notations préfixée et postfixée. En notation préfixée, l'opérateur apparaît avant ses opérandes, tandis qu'il apparaît après en notation postfixée.[25] Dans notre exemple, la liste des symboles de la notation préfixée est :

```
[+,*,2,-,9,+,3,4,5].
```

Pour la notation postfixée, on a :

```
[2,9,3,4,+,-,*,5,+].
```

Un problème se posant naturellement est celui de la conversion d'un $\mathbb{N}$-arbre quelconque en la liste des symboles correspondant à l'une des trois notations. Pour la forme infixée, nous aurons des parenthèses; comme il s'agit de caractères spéciaux en Prolog, nous les remplacerons par des lettres, `l` pour une parenthèse gauche et `r` pour une parenthèse droite. La figure 11.16 présente une première solution à ce problème. Cette solution est correcte mais pas très efficace; la

```
tree_to_pref(T,[T]) :- number(T).
tree_to_pref([X,Y,Z],[X|Xs]) :-
    tree_to_pref(Y,Ys), tree_to_pref(Z,Zs), append(Ys,Zs,Xs).

tree_to_postf(T,[T]) :- number(T).
tree_to_postf([X,Y,Z],Us) :-
    tree_to_postf(Y,Ys), tree_to_postf(Z,Zs),
    append(Ys,Zs,Xs), append(Xs,[X],Us).

tree_to_inf(T,[T]) :- number(T).
tree_to_inf([X,Y,Z],[l|Xs]) :-
    tree_to_inf(Y,Ys), tree_to_inf(Z,Zs),
    append([X|Zs],[r],Us), append(Ys,Us,Xs).
```

Figure 11.16 *Analyse de $\mathbb{N}$-arbre, solution naïve*

conversion d'une expression prendra un temps proportionnel au carré de la longueur de l'expression. Cela est dû au recours systématique au prédicat `append`. En outre, les prédicats proposés sont "à sens unique" : ils peuvent être utilisés pour transformer un $\mathbb{N}$-arbre en une liste de symboles mais ils ne peuvent opérer la conversion inverse, pourtant tout aussi utile. Il est facile de remédier à ce dernier défaut; les programmes de conversion inverse sont

[25] La notation postfixée est utilisée dans les calculateurs de poche Hewlett-Packard.

représentés à la figure 11.17. Ces programmes sont encore moins efficaces que

```
pref_to_tree([T],T) :- number(T).
pref_to_tree([X|Xs],[X,Y,Z]) :-
     append(Ys,Zs,Xs), pref_to_tree(Ys,Y), pref_to_tree(Zs,Z).
postf_to_tree([T],T) :- number(T).
postf_to_tree(Us,[X,Y,Z]) :-
    append(Xs,[X],Us), append(Ys,Zs,Xs),
    postf_to_tree(Ys,Y), postf_to_tree(Zs,Z).
inf_to_tree([T],T) :- number(T).
inf_to_tree([l|Xs],[X,Y,Z]) :-
    append(Ys,Us,Xs), append([X|Zs],[r],Us),
    inf_to_tree(Ys,Y), inf_to_tree(Zs,Z).
```

Figure 11.17 *Synthèse de* $\mathbb{N}$*-arbre, solution naïve*

les précédents parce que la première étape de la conversion d'une liste en un $\mathbb{N}$-arbre est la coupure de la liste au moyen du prédicat `append`. Aucun "indice" n'est fourni au système concernant l'endroit de la liste où cette coupure doit se faire, donc le système procède par essais successifs. (On peut s'en rendre compte en construisant un arbre de recherche.) Néanmoins, si l'expression à convertir n'est pas trop longue, la bonne réponse est fournie à l'utilisateur sans délai perceptible, et surtout sans que les effets du "tâtonnement" se fassent sentir. On peut donc considérer que le système Prolog simule un comportement intelligent puisque, du point de vue de l'utilisateur, le système opère le bon choix. La gestion du non-déterminisme est un mécanisme fondamentalement simple mais qui contribue largement à l'utilité de Prolog en intelligence artificielle. Nous en verrons quelques exemples dans la suite.

11.5. Pour en savoir plus

De nombreux livres ont été consacrés aux principes et à la pratique de la programmation classique et de Prolog, ainsi qu'à l'utilisation de Prolog en intelligence artificielle. Citons notamment, en français, [Delahaye 86] et [Thayse et al. 88]. Les aspects théoriques de la programmation logique sont étudiés dans [Doets 94] et [Lloyd 87]. Parmi les ouvrages d'initiation à Prolog et d'applications à l'intelligence artificielle, on notera [Bratko 90], [Shoam 95] et [Sterling et Shapiro 94].

Chapitre 12
Le langage Prolog en intelligence artificielle

Nous avons déjà montré que la gestion du non-déterminisme permettait d'organiser simplement une recherche dans un espace d'états. Cette recherche consiste souvent en la construction d'un arbre ou d'un graphe dont les nœuds sont étiquetés par des états. Deux exemples significatifs sont la recherche systématique des solutions au problème des Tours de Hanoï (cf. Fig. 10.5) et celle d'une tactique optimale pour le jeu d'Hexapion (cf. Fig. 10.7).

Nous avons vu aussi que le langage Prolog permettait une gestion simple (quoique pas toujours efficace) du non-déterminisme. L'exemple le plus typique, et peut-être aussi celui dont l'usage est le plus fréquent, est constitué par le prédicat `append`. En effet, celui-ci permet non seulement de concaténer deux listes, mais aussi de réaliser l'opération inverse, c'est-à-dire de décomposer une liste en un préfixe et un suffixe, de toutes les manières possibles.

Le système Prolog permet de résoudre simplement des problèmes admettant plusieurs solutions. En intelligence artificielle, on peut fréquemment présenter un problème dans ce cadre. Par exemple, un programme de jeu d'échec doit pouvoir sélectionner, parmi tous les mouvements de pièces et pions possibles, ceux qui sont conformes aux règles du jeu; il doit ensuite "filtrer" l'ensemble des coups possibles de manière à maximiser les chances de victoire. Cette technique de filtration est utile dans des contextes variés. Par exemple, le prédicat `pref_to_tree` (Fig. 11.17) convertit une expression arithmétique préfixée, représentée par sa liste de symboles, en un arbre syntaxique; pour ce faire, il "devine" d'abord comment la liste doit être décomposée en un opérateur et deux opérandes. Le programme ne comporte aucune indication sur la manière de procéder; en fait, le système Prolog essaie toutes

les décompositions possibles jusqu'à en trouver une adéquate. Cette tactique un peu simpliste est correcte car, si l'expression préfixée d'entrée est bien construite, elle ne peut être décomposée valablement que d'une seule manière. On pourrait objecter qu'il serait plus efficace, plus "intelligent", d'étudier d'abord le problème plus en profondeur, de manière à découvrir un moyen de déterminer la bonne décomposition a priori, sans essais préalables. Un tel moyen existe et peut facilement être programmé mais, pour de nombreux autres problèmes, ce n'est pas le cas; pour ces problèmes, la technique "essais et erreurs" prend tout son sens, ainsi que l'illustrent les exemples développés dans la suite de ce chapitre.[1]

Dans la mesure où la recherche non déterministe dans un espace d'états est une des techniques primordiales de l'intelligence artificielle, il est naturel de penser que le langage PROLOG se prêtera bien à la résolution de problèmes d'intelligence artificielle. Dans ce chapitre, nous allons essayer de montrer que c'est effectivement le cas.

Cette tâche est plutôt malaisée pour plusieurs raisons. D'une part, l'intelligence artificielle est un ensemble hétéroclite de techniques nombreuses, dont certaines n'ont que peu de rapport avec la logique; d'autres sont trop complexes pour être introduites ici; nous nous limiterons donc à ce qui concerne la recherche dans un espace d'états. D'autre part, les applications d'intelligence artificielle typiques sont longues et compliquées; elles nécessitent souvent le recours simultané à plusieurs techniques et se basent généralement sur un domaine de connaissance étendu et pointu.[2] Pour éviter ces écueils, nous choisissons de traiter des problèmes "jouets", dont l'utilité n'est guère apparente. Néanmoins, ces problèmes, par ailleurs amusants, reflètent plusieurs démarches typiques de problèmes plus sérieux et plus réalistes; ils constituent donc une illustration adéquate de notre propos.

12.1. Enigmes attribuables à Lewis Carroll

L'auteur de *Alice in the Wonderland* était un excellent logicien. Il est à l'origine de nombreuses énigmes, puzzles et problèmes logiques, dont certains sont des classiques des tests de quotient intellectuel. Lewis Carroll a eu dans ce domaine — et a toujours — de nombreux émules. Les recueils d'énigmes se sont multipliés lors de ces dernières décennies; de nouvelles sont en permanence

[1] Soulignons aussi le fait que, pour l'utilisateur, le prédicat `pref_to_tree` convient parfaitement. Le fait que l'algorithme sous-jacent ne soit pas "intelligent" a une conséquence positive : le système PROLOG fonctionne (dans ce cas) sans que le programmeur ait eu à lui fournir une tactique intelligente; cela signifie que PROLOG peut pallier l'absence de cette tactique en simulant un comportement intelligent. Nous reviendrons sur ce point au paragraphe 12.1.5.

[2] Un système expert de diagnostic médical, par exemple, comporte notamment une vaste base de données de faits médicaux.

soumises à la sagacité des lecteurs de journaux et de revues, sans qu'il soit généralement possible d'attribuer la paternité de telle énigme à telle personne. Nous pensons que Lewis Carroll n'aurait pas désavoué les quelques problèmes que nous présentons dans ce chapitre, et dont l'origine exacte nous est inconnue.

12.1.1. *Un problème d'intelligence artificielle ?*

Voici l'énoncé d'une énigme classique.

Cinq maisons alignées,
Cinq nationalités,
Cinq couleurs,
Cinq boissons favorites,
Cinq marques de tabac,
Cinq animaux familiers,
et de plus ...

1. *Les numéros des maisons sont 1, 2, 3, 4, 5.*
2. *L'Anglais habite la maison verte.*
3. *L'Espagnol possède un chien.*
4. *On boit du café dans la maison rouge.*
5. *On boit du thé chez l'Ukrainien.*
6. *La maison rouge suit la maison blanche.*
7. *Le fumeur de Old Gold élève des escargots.*
8. *On fume des Gauloises dans la maison jaune.*
9. *On boit du lait au numéro 3.*
10. *Le Norvégien habite au numéro 1.*
11. *Le fumeur de Chesterfield et le propriétaire du renard sont voisins.*
12. *Le fumeur de Gauloises habite à côté du propriétaire du cheval.*
13. *Le fumeur de Lucky Strike boit du jus d'orange.*
14. *Le Japonais fume des Gitanes.*
15. *La maison bleue jouxte celle du Norvégien.*

Qui possède le zèbre ... et qui boit de l'eau ?

On peut discuter la question de savoir si cette énigme est bien un problème d'intelligence artificielle. Cela dépend en fait du point de vue auquel on se place. Chaque maison se décrit par la nationalité de son propriétaire, la couleur de sa façade, un animal familier, une boisson et une marque de tabac. D'un point de vue opérationnel, nous devons sélectionner cinq permutations (indépendantes) de listes de cinq éléments; il existe $(5!)^5 = 120^5 = 24\,883\,200\,000$ manières d'opérer cette sélection. Résoudre ce problème ne requiert en principe guère d'intelligence : il suffit de construire toutes les solutions puis de les tester une à une jusqu'à trouver celle qui vérifie tous les indices qui nous sont donnés. Une meilleure méthode, plus efficace mais moins simple à mettre en œuvre, consiste à construire des solutions partielles et à les rejeter ou les modifier dès que l'un

des indices n'est pas vérifié. C'est en fait la manière dont Prolog procède, et aussi celle que l'être humain essaiera de mettre en pratique, lors d'un test d'intelligence par exemple. Cela nous incite à considérer qu'il s'agit bien d'un problème d'intelligence artificielle.

12.1.2. *Un programme de résolution en* PROLOG

L'ensemble des définitions auxiliaires codées en Prolog se trouve à la figure 12.1; on définit les notions d'appartenance, de précédence (une maison précède une autre si le numéro de la première est plus petit que celui de la seconde), de première maison, de maison du milieu et de maisons voisines. Le programme Prolog permettant de résoudre l'énigme est présenté à la figure 12.2. On observe immédiatement que ce programme n'est rien d'autre qu'un codage de l'énoncé de l'énigme.

```
prc(A,B,[A,B,C,D,E]).  prc(A,C,[A,B,C,D,E]).  prc(A,D,[A,B,C,D,E]).
prc(A,E,[A,B,C,D,E]).  prc(B,C,[A,B,C,D,E]).  prc(B,D,[A,B,C,D,E]).
prc(B,E,[A,B,C,D,E]).  prc(C,D,[A,B,C,D,E]).  prc(C,E,[A,B,C,D,E]).
prc(D,E,[A,B,C,D,E]).

one(A,[A,B,C,D,E]).
three(C,[A,B,C,D,E]).

neighbor(A,B,[A,B,C,D,E]).  neighbor(B,C,[A,B,C,D,E]).
neighbor(C,D,[A,B,C,D,E]).  neighbor(D,E,[A,B,C,D,E]).
neighbor(B,A,[A,B,C,D,E]).  neighbor(C,B,[A,B,C,D,E]).
neighbor(D,C,[A,B,C,D,E]).  neighbor(E,D,[A,B,C,D,E]).

nation(h(N,C,A,B,T),N).
color(h(N,C,A,B,T),C).
animal(h(N,C,A,B,T),A).
drink(h(N,C,A,B,T),B).
tobacco(h(N,C,A,B,T),T).
```

Figure 12.1 *Résoudre l'énigme du Zèbre : les définitions*

La clause principale définit le prédicat `go`. Les deux premières lignes correspondent à l'indice 1; les lignes suivantes correspondent aux quatorze autres indices, sauf les deux dernières lignes qui correspondent aux deux questions. L'exécution de ce programme est représentée à la figure 12.3.

Cet exemple de l'énigme du zèbre peut être vu comme une parfaite illustration de la puissance de la logique : pour résoudre le problème, il a suffi de le traduire en logique formelle. Le même exemple illustre aussi l'intérêt de PROLOG et, plus généralement, l'intérêt des aspects opératoires de la logique (méthodes de résolution et d'unification). Cet intérêt est spécialement marqué dans le domaine de l'intelligence artificielle, puisque l'utilisateur de PROLOG n'a eu qu'à fournir au système l'énoncé de son problème pour en obtenir la solution.


```
go(X,Y) :- St = [h(N1,C1,A1,B1,T1),h(N2,C2,A2,B2,T2),
                 h(N3,C3,A3,B3,T3),h(N4,C4,A4,B4,T4),h(N5,C5,A5,B5,T5)],
  member(X2,St),nation(X2,english),color(X2,green),
  member(X3,St),nation(X3,spanish),animal(X3,dog),
  member(X4,St),color(X4,red),drink(X4,coffee),
  member(X5,St),nation(X5,ukrainian),drink(X5,tea),
  neighbor(X6a,X6b,St),prc(X6b,X6a,St),color(X6a,red),color(X6b,white),
  member(X7,St),tobacco(X7,oldgold),animal(X7,snails),
  member(X8,St),color(X8,yellow),tobacco(X8,gauloises),
  three(X9,St),drink(X9,milk),
  one(X10,St),nation(X10,norwegian),
  neighbor(X11a,X11b,St),tobacco(X11a,chesterfield),animal(X11b,fox),
  neighbor(X12a,X12b,St),tobacco(X12a,gauloises),animal(X12b,horse),
  member(X13,St),tobacco(X13,luckystrikes),drink(X13,orangejuice),
  member(X14,St),nation(X14,japanese),tobacco(X14,gitanes),
  neighbor(X15a,X15b,St),nation(X15a,norwegian),color(X15b,blue),
  member(Q,St),animal(Q,zebra),nation(Q,X),
  member(R,St),drink(R,water),nation(R,Y).
```

Figure 12.2 *Résoudre l'énigme du Zèbre : le programme*

```
?- go(ZebraOwner,WaterDrinker).
ZebraOwner = japanese
WaterDrinker = norwegian ? ;
no
```

Figure 12.3 *Résoudre l'énigme du Zèbre : la question et les réponses*

Nous avons déjà mentionné que ce problème était "intelligent", puisqu'il est difficile à résoudre par l'être humain; la difficulté vient de l'absence de méthode, et une demi-heure d'essais et d'erreurs, avec crayon et papier, est en général nécessaire pour le résoudre (sans l'aide de PROLOG). On peut donc dire qu'en l'occurrence, la logique formelle et le système PROLOG qui la met en œuvre ont conjointement simulé un comportement intelligent.

Cette argumentation n'est pas fondamentalement incorrecte, mais elle doit être nuancée. L'objection majeure est que, contrairement à ce que l'on pourrait croire, une méthode systématique de résolution de ce type d'énigmes existe : l'être humain peut lui-même simuler PROLOG et résoudre l'énigme en construisant l'arbre de recherche; nous avons vu que cette construction était élémentaire : il suffit de connaître les algorithmes de résolution et d'unification. Cela nous rappelle que le système PROLOG résout cette énigme comme un calculateur électronique évalue des expressions arithmétiques complexes, à savoir en appliquant des méthodes le plus souvent élémentaires, mais à une vitesse inaccessible pour le cerveau humain.

Cette contre-argumentation est elle aussi excessive. Tout d'abord, on ne demande pas à un système informatique d'être intelligent, mais de traiter l'information correctement. Il y a intelligence artificielle dès que les résultats produits par le système requièrent, pour être produit par l'être humain, une démarche intelligente. Le fait est que, pour l'être humain typique, la résolution de l'énigme du zèbre sera vue comme un défi intellectuel, sans qu'il soit question d'appliquer point par point une méthode systématique (ce qui prendrait d'ailleurs un temps très considérable). Le système PROLOG applique une méthode non intelligente (c'est-à-dire, une méthode systématique) mais, ce faisant, il accomplit le même travail que l'être humain appliquant une méthode intelligente (non systématique, faite d'essais et d'erreurs plus ou moins nombreux). Notre conclusion est qu'il y a bien intelligence artificielle, mais l'intelligence artificielle simule l'intelligence naturelle dans ses résultats concrets et non dans les processus calculatoires conduisant à ces résultats.

12.1.3. *La classe de mathématiques*

Cette énigme ressemble à la précédente mais met en évidence quelques limitations de Prolog.

> *Le professeur de la classe de mathématiques a perdu le classement de ses onze élèves et ne dispose, pour le reconstituer, que des renseignements que veut bien lui donner l'élève Lolo :*
>
> 1. *Le premier et le dernier de la liste portent des noms ayant la même initiale.*
> 2. *Aucun élève n'a la même initiale que son ou ses voisins de classement.*
> 3. *William et Wilfrid sont après Washington qui, au classement, n'est pas voisin de Ludwig.*
> 4. *Herbert, par contre, est au classement, voisin de Ludwig.*
> 5. *Ludovic, voisin de Wolfgang au classement, occupe l'une des trois dernières places.*
> 6. *Le nombre de places séparant Hubert de Ludwig est égal au nombre de places séparant Hubert de William, lequel est à moins de trois places d'Honoré.*
> 7. *Il n'y a pas d'ex-æquo et Wilfrid est avant moi.*
> 8. *Un seul d'entre nous occupe la place qu'il aurait dans l'ordre alphabétique.*
> 9. *Harold occupe une place impaire et est mieux classé que William.*

A priori, ce problème est plus facile que le précédent car le nombre de classements possibles pour onze personnes est "seulement" 11!, c'est-à-dire 39 916 800, alors que l'énigme précédente comportait plus de 24 milliards de possibilités. En outre, le codage des indices reste très simple, puisque ceux-ci ne concernent que les différentes manières de ranger onze éléments dans une liste.

Néanmoins, on observe quelques difficultés inattendues. Le premier type d'indice concerne directement les positions des élèves dans le classement. Deux élèves peuvent être voisins dans la liste (`neighbour`) ou non (`not_neighbour`: au moins un troisième élève se situe entre les deux premiers); un élève peut précéder un autre dans le classement (`before`), occuper une place impaire (`odd_rank`) ou être éloigné de moins de trois places d'un autre (`less_than_three`). Tous ces prédicats s'écrivent sans difficulté; plusieurs programmes sont possibles mais un choix raisonnable est le suivant :

```
neighbour(X,Y,Zs) :- append(Xs,[X,Y|Ys],Zs).
neighbour(Y,X,Zs) :- append(Xs,[X,Y|Ys],Zs).

not_neighbour(X,Y,Zs) :- append(Xs,[A|Ys],Zs),
                         member(X,Xs), member(Y,Ys).
not_neighbour(Y,X,Zs) :- append(Xs,[A|Ys],Zs),
                         member(X,Xs), member(Y,Ys).

before(X,Y,Zs) :- append(Xs,[Y|Ys],Zs), member(X,Xs).

odd_rank(X,[X|Xs]).
odd_rank(X,[A,B|Xs]) :- odd_rank(X,Xs).

less_than_three(X,Y,St) :- append(Xs,[X,Y|Ys],St).
less_than_three(Y,X,St) :- append(Xs,[X,Y|Ys],St).
less_than_three(X,Y,St) :- append(Xs,[X,A,Y|Ys],St).
less_than_three(Y,X,St) :- append(Xs,[X,A,Y|Ys],St).
```

Le prédicat `up_mid` permet de déterminer si trois élèves `A`, `M` et `B` se succèdent dans cet ordre au sein du classement `St`, avec la restriction additionnelle que l'écart entre `A` et `M` est le même que celui entre `M` et `B`. Le prédicat `middle` permet de détecter la situation analogue dans le même ordre ou dans l'ordre inverse.

```
up_mid(A,M,B,St) :- append(Xs,[A,M,B|Ys],St).
up_mid(A,M,B,St) :- append(Xs,[A,A1,M,B1,B|Ys],St).
up_mid(A,M,B,St) :- append(Xs,[A,A1,A2,M,B2,B1,B|Ys],St).
up_mid(A,M,B,St) :- append(Xs,[A,A1,A2,A3,M,B3,B2,B1,B|Ys],St).
up_mid(A,M,B,[A,A1,A2,A3,A4,M,B4,B3,B2,B1,B]).

middle(A,M,B,St) :- up_mid(A,M,B,St).
middle(B,M,A,St) :- up_mid(A,M,B,St).
```

On notera dans ces prédicats l'usage intensif de `append`. Remarquons aussi que, dans la mesure où la longueur du classement (la structure `St`) est fixée à 11, on aurait pu programmer `odd_rank` comme suit :

```
odd_rank(X1,[X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11]).
odd_rank(X3,[X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11]).
odd_rank(X5,[X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11]).
odd_rank(X7,[X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11]).
odd_rank(X9,[X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11]).
odd_rank(X11,[X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11]).
```

C'est peu concis, mais simple et efficace. Notons aussi à cette occasion qu'il n'est pas utile de nommer une variable qui n'intervient qu'une seule fois dans une clause; on aurait pu écrire, par exemple,

```
...
odd_rank(X3,[_,_,X3,_,_,_,_,_,_,_,_]).
...
```

où le caractère "`_`" est un nom générique pour une variable "anonyme".

D'autres prédicats auxiliaires se réfèrent aux noms des élèves, à l'ordre alphabétique et à l'initiale des noms. On écrit sans peine

```
alpha([harold,herbert,honore,hubert,
       lolo,ludovic,ludwig,
       washington,wilfrid,william,wolfgang]).

init(X,h) :- member(X,[harold,herbert,honore,hubert]).
init(X,l) :- member(X,[lolo,ludovic,ludwig]).
init(X,w) :- member(X,[washington,wilfrid,william,wolfgang]).
```

Ces prédicats permettent de tester qu'un seul élève occupe dans le classement la même place que dans l'ordre alphabétique, et que deux voisins au classement portent toujours des noms dont les initiales sont différentes :

```
exactly_one([X|Xs],[X|Ys]) :- exactly_none(Xs,Ys).
exactly_one([X|Xs],[Y|Ys]) :- X \= Y, exactly_one(Xs,Ys).

exactly_none([],[]).
exactly_none([X|Xs],[Y|Ys]) :- X \= Y, exactly_none(Xs,Ys).

distinct(h,l). distinct(h,w).
distinct(l,h). distinct(l,w).
distinct(w,h). distinct(w,l).

distinct_init_neighbour([X]).
distinct_init_neighbour([X,Y|Ys]) :-
          init(X,I), init(Y,J), I \= J,
          distinct_init_neighbour([Y|Ys]).
```


On peut à présent écrire le prédicat correspondant à la question. Sur base de ce qui précède, on pourra écrire

```
go(St) :-
  St = [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11],
  init(X1,I), init(X11,I),             /* indice 1  */
  distinct_init_neighbour(St),         /* indice 2  */
  before(washington,william,St),       /* indice 3a */
  before(washington,wilfrid,St),       /* indice 3b */
  not_neighbour(washington,ludwig,St), /* indice 3c */
  neighbour(herbert,ludwig,St),        /* indice 4  */
  neighbour(ludovic,wolfgang,St),      /* indice 5a */
  member(ludovic,[X9,X10,X11]),        /* indice 5b */
  middle(ludwig,hubert,william,St),    /* indice 6a */
  less_than_three(william,honore,St),  /* indice 6b */
  before(wilfrid,lolo,St),             /* indice 7  */
  alpha(L), exactly_one(St,L),         /* indice 8  */
  odd_rank(harold,St),                 /* indice 9a */
  before(harold,william,St).           /* indice 9b */
```

Malheureusement, cette solution a priori correcte ne fonctionne pas en pratique, parce que les indices sont soumis au système dans un ordre inadéquat. En conséquence, lorsque le système PROLOG répond à la question `go(St)`, il génère quantité d'essais infructueux; par exemple, le "classement" :

```
[harold, lolo, harold, ludwig, washington, ludwig,
 william, ludwig, herbert, wilfrid, harold]
```

quoique grossièrement impossible (certains élèves n'apparaissent pas, d'autres apparaissent deux fois ou plus), respecte quand même les quatre premiers indices. Par tâtonnements[3] on crée un ordonnancement plus adéquat pour les indices :

```
go(St) :-
  St = [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11],
  member(ludovic,[X9,X10,X11]),
  odd_rank(harold,St),
  before(harold,william,St),
  neighbour(ludovic,wolfgang,St),
  neighbour(herbert,ludwig,St),
  before(wilfrid,lolo,St),
  before(washington,william,St),
  not_neighbour(washington,ludwig,St),
```

[3] Et en utilisant quelques règles de bonne pratique non présentées ici.

```
less_than_three(william,honore,St),
before(washington,wilfrid,St),
middle(ludwig,hubert,william,St),
alpha(L), exactly_one(St,L),
init(X1,I), init(X11,I),
distinct_init_neighbour(St).
```

Le système fournit alors l'unique solution au problème; le classement correct est :

```
[washington, herbert, ludwig, wilfrid, harold, lolo,
 hubert, wolfgang, ludovic, honore, william] ;
```

Néanmoins, le temps nécessaire à la résolution du problème est de l'ordre d'une minute sur un ordinateur puissant, alors qu'il n'était que d'une seconde pour le problème précédent. La raison est que, dans l'écriture des prédicats, nous avons privilégié la clarté aux dépens de l'efficacité. Par exemple, le prédicat `not_neighbour` est une source de ralentissement; plutôt que d'en donner une définition explicite, il vaudrait mieux indiquer au système que ce prédicat est vrai si et seulement si le prédicat `neighbour` est faux. PROLOG permet cela; il suffit de remplacer ci-avant la ligne :

```
not_neighbour(washington,ludwig,St),
```

par la ligne :

```
not neighbour(washington,ludwig,St),
```

où le `not` a précisément l'effet souhaité. Ce changement en apparence mineur accélère l'exécution d'un facteur 10 ! Sans entrer dans les détails des aspects "extra-logiques" de PROLOG, signalons que ceux-ci rendent l'emploi du système délicat dans les problèmes complexes; PROLOG n'est donc pas la panacée dans le domaine de la logique appliquée à l'intelligence artificielle, même s'il constitue clairement un des fleurons de ce domaine.

12.1.4. *Les aléas de l'exploration*

Ce nouvel exemple montre comment l'intelligence naturelle et l'intelligence artificielle peuvent utilement collaborer.

> *Un explorateur galactique raconte son arrivée sur une petite planète couverte de forêts et habitée par deux peuples amis, les Véridiques (ils disent toujours la vérité) et les Contradiques (ils ne la disent jamais). Les habitants ne répondent aux questions qu'on leur pose que par "whamm" et "galagala", et n'utilisent jamais un autre moyen d'expression. Certains jours, whamm veut dire "oui" et galagala "non" mais d'autres jours, c'est le contraire ! De plus, un étranger n'a pas le droit de poser plus d'une question à un autochtone.*

> *Egaré dans la forêt, l'explorateur rencontre un groupe d'habitants au carrefour de deux pistes et veut savoir quelle piste conduit à sa base. Il rapporte le dialogue suivant :*
>
> 1. *Explorateur : Whamm est-il affirmatif aujourd'hui ?*
>   *Grocongo : whamm !*
> 2. *Explorateur : Etes-vous tous du même peuple ?*
>   *Minibola : whamm !*
> 3. *Explorateur : Dodo, Enigmo, Sitopri et toi, êtes-vous du même peuple ?*
>   *Biribiri : galagala !*
> 4. *Explorateur : Crocro et Minibola sont-ils du même peuple ?*
>   *Limacela : galagala !*
> 5. *Explorateur : Grocongo, Crocro et toi, êtes-vous du même peuple ?*
>   *Lasercho : whamm !*
> 6. *Explorateur : Crocro, Limacela et toi, êtes-vous du même peuple ?*
>   *Albarama : galagala !*
> 7. *Explorateur : Pikaglass et Bonifacio sont-ils du même peuple ?*
>   *Dada : galagala !*
> 8. *Explorateur : Es-tu du même peuple que Minibola ?*
>   *Crocro : galagala !*
> 9. *Explorateur : Es-tu du même peuple que Biribiri ?*
>   *Dodo : whamm !*
> 10. *Explorateur : Sitopri, Dada et toi, êtes-vous du même peuple ?*
>   *Enigmo : whamm !*
> 11. *Explorateur : Dodo et Enigmo sont-ils du même peuple ?*
>   *Sitopri : whamm !*
> 12. *Explorateur : Enigmo, Dodo et toi, êtes-vous du même peuple ?*
>   *Gagagogo : whamm !*
>
> *A quel peuple appartient Gagagogo ? Quelle question l'explorateur a-t-il posée à Bonifacio, le dernier membre du groupe, pour connaître la direction de sa base ?*

On constate qu'à l'exception du premier, tous les indices sont de même nature; ils concernent l'appartenance de certains individus à l'un des deux peuples. La première question est également de ce type, alors que la seconde est de nature différente et semble a priori réclamer un peu de créativité. Ceci suggère de réserver au système Prolog les onze derniers indices ainsi que la première question, tandis que notre intelligence humaine se chargera du premier indice et de la seconde question.

Il est clair que le premier indice ne permet pas de déterminer si "whamm" veut dire oui ou non, il permet seulement de déterminer que Grocongo est un Véridique. Les onze autres indices requièrent l'écriture d'un prédicat `meme_peuple` vérifiant si tous les éléments d'une liste `L` appartiennent au même peuple. La valeur retournée par ce prédicat dépendra non seulement de `L` mais aussi :

- du peuple auquel appartient le locuteur;
- du fait que "oui" soit "whamm" ou "galagala";
- de la réponse fournie par le locuteur.

Le prédicat à écrire est donc du type `meme_peuple(Ploc,List,Oui,Rep)`. Il y a quatre possibilités, selon que la réponse est affirmative ou négative et que le locuteur est un Véridique ou un Contradique, ce qui conduit immédiatement aux quatre clauses suivantes :

```
meme_peuple(vr,L,Oui,Rep) :- yes(Oui,Rep), meme(L).
meme_peuple(vr,L,Oui,Rep) :- no(Oui,Rep), diff(L).
meme_peuple(ct,L,Oui,Rep) :- yes(Oui,Rep), diff(L).
meme_peuple(ct,L,Oui,Rep) :- no(Oui,Rep), meme(L).
```

Les prédicats auxiliaires sont extrêmement simples :

```
meme([X]).
meme([X,X|Xs]) :- meme([X|Xs]).

some_verid([vr|Xs]).
some_verid([ct|Xs]) :- some_verid(Xs).

some_contra([ct|Xs]).
some_contra([vr|Xs]) :- some_contra(Xs).

diff([vr|Xs]) :- some_contra(Xs).
diff([ct|Xs]) :- some_verid(Xs).

yes(w,w).
yes(g,g).
no(w,g).
no(g,w).
```

Comme précédemment, un prédicat spécial `go` permettra de répondre à la (première) question. Ce prédicat `go` prendra comme premier argument `Gag`, qui s'instanciera en le peuple auquel appartient Gagagogo ("`vr`" pour Véridique, "`ct`" pour Contradique). Même si cela n'est pas demandé explicitement, nous essayerons également de savoir quel mot veut dire oui et à quel peuple appartient chacun des protagonistes; cela sera utile pour répondre à la seconde question. On obtient ainsi le prédicat suivant, dans lequel `Struc` (pour "structure") représente la liste des nationalités des autochtones, dans l'ordre où ils sont évoqués dans l'énoncé :

```
go(Gag,Oui,Struc) :-
  Struc =
   [Gro,Min,Bir,Lim,Las,Alb,Dad,Cro,Dod,Eni,Sit,Gag,Pik,Bon],
  Gro = veri,
  meme_peuple(Min,Struc,Oui,w),
  meme_peuple(Bir,[Dod,Eni,Sit,Bir],Oui,g),
  meme_peuple(Lim,[Cro,Min],Oui,g),
  meme_peuple(Las,[Gro,Cro,Las],Oui,w),
  meme_peuple(Alb,[Cro,Lim,Alb],Oui,g),
  meme_peuple(Dad,[Pik,Bon],Oui,g),
  meme_peuple(Cro,[Min,Cro],Oui,g),
  meme_peuple(Dod,[Bir,Dod],Oui,w),
  meme_peuple(Eni,[Sit,Dad,Eni],Oui,w),
  meme_peuple(Sit,[Dod,Eni],Oui,w),
  meme_peuple(Gag,[Eni,Dod,Gag],Oui,w).
```

Le programme s'exécute comme indiqué à la figure 12.4.[4]

On constate que Gagagogo est nécessairement un Contradique et que, ce jour-là, "whamm" est le mot affirmatif. Prolog fournit quatre réponses parce que, d'une part, la nationalité d'Alberama n'est pas établie et, d'autre part, Pikaglass et Bonifacio sont de nationalités différentes mais on ne peut préciser qui est le Véridique et qui est le Contradique. Ce dernier point est gênant puisque la seconde question ne peut être résolue aisément que si la nationalité de Bonaficio est connue. Toutefois, il est certain que, de Pikaglass et Bonifacio, un seul est Contradique. L'explorateur peut donc poser à Bonifacio la question suivante : "si je demandais à Pikaglass 'le chemin de droite conduit-il à ma base?', que me répondrait-il?". Si Bonifacio répond "whamm", le chemin à prendre est celui de gauche; s'il répond "galagala", c'est celui de droite. On notera sans surprise que le système PROLOG n'invente pas la question adéquate; il fournit seulement les informations permettant à l'utilisateur de découvrir lui-même cette question.

12.1.5. *Une enquête de Sherlock Holmes*

L'énigme présentée ici illustre un type plus élaboré de collaboration entre l'intelligence naturelle et l'intelligence artificielle; elle permet aussi de montrer le lien étroit existant entre la logique, la programmation logique et la démonstration automatique de théorème.

[4] Pour plus de lisibilité, on a inséré une première ligne spécifiant l'identité des autochtones.

```
%%%%%%%%[Gro,Min,Bir,Lim,Las,Alb,Dad,Cro,Dod,Eni,Sit,Gag,Pik,Bon]

?- go(Gag,Oui,Struc).

Gag = ct
Oui = w
Struc = [vr, ct, vr, ct, ct, vr, vr, ct, vr, ct, ct, ct, vr, ct];

Gag = ct
Oui = w
Struc = [vr, ct, vr, ct, ct, vr, vr, ct, vr, ct, ct, ct, ct, vr];

Gag = ct
Oui = w
Struc = [vr, ct, vr, ct, ct, ct, vr, ct, vr, ct, ct, ct, vr, ct];

Gag = ct
Oui = w
Struc = [vr, ct, vr, ct, ct, ct, vr, ct, vr, ct, ct, ct, ct, vr];

No
```

Figure 12.4 *Résolution du problème de l'explorateur*

12.1.6. *L'énoncé*

Un jour, Sherlock Holmes reçoit la visite de son ami Watson que l'on avait chargé d'enquêter sur un assassinat mystérieux datant de plus de dix ans. A l'époque, le duc de Densmore avait été tué par l'explosion d'une bombe, qui avait également détruit le château de Densmore où il s'était retiré. Les journaux d'alors relataient que le testament, détruit lui aussi par l'explosion, avait tout pour déplaire à l'une de ses sept ex-femmes. Or, avant de mourir, le duc les avait toutes invitées à passer quelques jours dans sa retraite écossaise.

H. *Je me souviens de l'affaire. Ce qui est étrange, c'est que la bombe avait été fabriquée spécialement pour être cachée dans l'armure de la chambre à coucher, ce qui montre que l'assassin a nécessairement effectué plus d'une visite au château.*

W. *Certes, et pour cette raison, j'ai interrogé chacune des femmes : Ann, Betty, Charlotte, Edith, Felicia, Georgia, Helen. Elles ont toutes juré qu'elles n'avaient été au château de Densmore qu'une seule fois dans leur vie.*

H. *Hum ! Leur avez-vous demandé à quelle période ont eu lieu leurs séjours respectifs ?*

W. *Hélas ! Aucune ne se rappelait les dates exactes, après plus de dix ans ! Néanmoins, je leur ai demandé qui elles y avaient rencontré :*

- *Ann a rencontré Betty, Charlotte, Felicia et Georgia.*
- *Betty a rencontré Ann, Charlotte, Edith, Felicia et Helen.*
- *Charlotte a rencontré Ann, Betty et Edith.*
- *Edith a rencontré Betty, Charlotte et Felicia.*
- *Félicia a rencontré Ann, Betty, Edith et Helen.*
- *Georgia a rencontré Ann et Helen.*
- *Helen a rencontré Betty, Felicia et Georgia.*

Vous voyez, mon cher Holmes, ces réponses sont concordantes ...

H. *Cela vous étonne ? Ces femmes se méfient ... mais aussi, elles se haïssent ! Aucune n'aurait accepté de s'associer avec une autre, ou de couvrir son crime. Non, il y a une seule coupable ...*

QUI ?

12.1.6.1. *Une solution directe*

Nous savons qui chacune des suspectes a rencontré et aussi que toutes prétendent n'avoir effectué qu'un seul séjour au château. On peut donc essayer, sur base de ces indices, de reconstituer l'ordre des départs et des arrivées dans le temps. Naturellement, comme l'une des suspectes a menti, il ne sera pas possible de mener à bien cette reconstitution, sauf si on ne tient pas compte de la coupable. La démarche consiste donc à déterminer qui doit être éliminée pour qu'un ordonnancement existe, compatible avec le tableau des rencontres. Trois prédicats, `ex_wives`, `same_time` et `with`, modélisent la liste des suspectes et le tableau des rencontres :

```
ex_wives([ann,betty,charlotte,edith,felicia,georgia,helen]).

same_time(ann,[betty,charlotte,felicia,georgia]).
same_time(betty,[ann,charlotte,edith,felicia,helen]).
same_time(charlotte,[ann,betty,edith]).
same_time(edith,[betty,charlotte,felicia]).
same_time(felicia,[ann,betty,edith,helen]).
same_time(georgia,[ann,helen]).
same_time(helen,[betty,felicia,georgia]).

with(A,B) :- same_time(A,L), member(B,L).
```

Le prédicat `history(M,H)` devra produire un ordonnancement des départs et des arrivées `H` compatible avec la partie du tableau des rencontres relative aux suspectes appartenant à la liste `M`. Par exemple, les réponses à la question :

```
?- histoire([ann,betty,edith],H).
```

seront :

```
H = [[ann|in],[betty|in],[ann|out],[edith|in],[betty|out],[edith|out]]
H = [[betty|in],[ann|in],[ann|out],[edith|in],[betty|out],[edith|out]]
H = [[ann|in],[betty|in],[ann|out],[edith|in],[edith|out],[betty|out]]
H = [[betty|in],[ann|in],[ann|out],[edith|in],[edith|out],[betty|out]]
H = [[betty|in],[edith|in],[edith|out],[ann|in],[ann|out],[betty|out]]
H = [[betty|in],[edith|in],[edith|out],[ann|in],[betty|out],[ann|out]]
H = [[edith|in],[betty|in],[edith|out],[ann|in],[ann|out],[betty|out]]
H = [[edith|in],[betty|in],[edith|out],[ann|in],[betty|out],[ann|out]]
```

puisque ces ordonnancements sont compatibles avec les faits : Ann et Betty se sont rencontrées, Betty et Edith se sont rencontrées, Ann et Edith ne se sont pas rencontrées. On construira les ordonnancements par la méthode habituelle : une clause concerne la liste vide, l'autre l'adjonction d'un nouvel élément dans la liste. On observe aussi que, si `A` arrive avant `B`, alors `A` rencontre `B` si et seulement si `B` arrive avant le départ de `A`. On utilise le prédicat `insert_in_order`, tel que la réponse à la question :

```
?- insert_in_order(x,y,L0,L1).
```

où `L0` est une liste et `L1` une variable, soit l'instantiation de `L1` en une liste résultant de l'insertion de `x` et de `y` (dans l'ordre) dans la liste `L1`. Par exemple, l'une des réponses si `L0` est `[a,b,c]` sera `[a,x,b,y,c]`. On obtient aisément les prédicats suivants :

```
history([],[]).
history([A|M],H) :- history(M,H1), h_ins(A,H1,H), ok(A,M,H).

h_ins(A,H1,H) :- insert_in_order([A|in],[A|out],H1,H).

insert_in_order(X,Y,H1,H) :-
     append(Ha,Hx,H1), append(Hb,Hc,Hx),
     append([X|Hb],[Y|Hc],Hbc), append(Ha,Hbc,H).

ok(A,[],H).
ok(A,[B|M],H) :- with(A,B), ok_y(A,B,H), ok(A,M,H).
ok(A,[B|M],H) :- not with(A,B), ok_n(A,B,H), ok(A,M,H).

ok_y(A,B,H) :- before([A|in],[B|in],H), before([B|in],[A|out],H).
ok_y(A,B,H) :- before([B|in],[A|in],H), before([A|in],[B|out],H).

ok_n(A,B,H) :- before([A|out],[B|in],H).
ok_n(A,B,H) :- before([B|out],[A|in],H).

before(X,Y,[X|U]) :- member(Y,U).
```



```
before(X,Y,[_|U]) :- before(X,Y,U).
```

Le prédicat `ok_y(A,B,H)` est vrai si l'ordonnancement `H` est compatible avec le fait que `A` et `B` se sont rencontrés; le prédicat `ok_n(A,B,H)` est vrai si l'ordonnancement `H` est compatible avec le fait que `A` et `B` ne se sont pas rencontrés. La réponse à la question :

```
?- ex_wives(L), history(L,_).
```

est `no` : il n'existe pas d'ordonnancement compatible avec le tableau complet des rencontres. Si on admet l'hypothèse de Sherlock Holmes, il suffit de retirer la coupable du tableau pour qu'un ordonnancement (au moins) soit possible. Cela se fait au moyen des prédicats suivants :

```
guilty(X) :- ex_wives(L), select(X,L,M), history(M,_).

select(X,[X|M],M).
select(X,[Y|M],[Y|N]) :- select(X,M,N).
```

Le prédicat `select` permet d'obtenir une liste `M` en extrayant un élément `X` de la liste `L`. La seule réponse à la question `?- guilty(X)` est `X = ann`.

12.1.6.2. *Une solution plus rapide*

La technique consistant à construire les ordonnancements est simple, naturelle et, bien sûr, correcte, mais elle n'est pas très efficace. On apprend en théorie des graphes et des ordonnancements que l'ordonnancement est impossible s'il existe un "carré contradictoire", c'est-à-dire un groupe de quatre personnes A, B, C et D tel que A et D ont rencontré B et C sans que A ait rencontré D ni que B ait rencontré C. Le prédicat `contra_square` permet de détecter les carrés contradictoires dans le tableau des rencontres :

```
contra_square([A,B,C,D]) :- with(A,B), with(D,B),
                            with(A,C), with(D,C),
                            not with(A,D), not with(B,C).
```

On notera cependant que, si `[A,B,C,D]` est un carré contradictoire, alors `[A,C,B,D]`, `[D,B,C,A]`, `[D,C,B,A]`, `[B,A,D,C]`, `[B,D,A,C]`, `[C,A,D,B]` et `[C,D,A,B]` le sont aussi. Pour éviter ces redondances, on utilisera le prédicat `@<`, correspondant à l'ordre alphabétique. Signalons aussi le prédicat spécial, prédéfini, permettant de regrouper en une liste toutes les solutions à une question donnée; `findall(X,pred(X),Xs)` instancie `Xs` en la liste des solutions à la question `pred(X)`.

```
contra_square([A,B,C,D]) :- with(A,B), A @< B, with(D,B), A @< D,
                            with(A,C), B @< C, with(D,C),
                            not with(A,D), not with(B,C).

contra_list(SQs) :- findall(SQ, contra_square(SQ), SQs).
```

Il reste alors à isoler la coupable, qui intervient dans tous les carrés contradictoires; cela se fait au moyen des prédicats `guilty` et `member_all`:

```
guilty(X) :- contra_list(SQs), member_all(X,SQs).

member_all(X,[]).
member_all(X,[SQ|SQs]):- member(X,SQ), member_all(X,SQs).
```

Cette technique permet de résoudre le problème beaucoup plus rapidement que la précédente. Le temps et les ressources informatiques supplémentaires requises par la première solution représentent le "prix de l'ignorance": le système PROLOG a simulé par une recherche intensive l'absence d'une règle "intelligente" pour résoudre le problème qui lui était soumis. Pour l'utilisateur, les deux solutions se valent dans la mesure où chacune d'elle fournit le résultat correct dans un délai raisonnable. Néanmoins, si la taille du problème avait été plus considérable (par exemple, cinquante suspects au lieu de sept), la première technique aurait été impraticable; cela montre que le recours à l'intelligence artificielle reste un pis-aller, à n'utiliser que si aucune technique plus classique n'est connue.

12.1.6.3. PROLOG *comme outil de preuve*

Il est facile de démontrer au moyen de PROLOG que la présence d'un carré contradictoire empêche la possibilité d'un ordonnancement où chacun des protagonistes n'effectuerait qu'un seul séjour. On peut utiliser le tableau suivant:

```
with(a,d).  with(d,a).  with(b,c).  with(c,b).
```

et d'observer que la question `?- history([a,b,c,d],H)` amène la réponse `no`.

12.2. Une limitation importante

Les énigmes décrites au début de ce chapitre constituent une application intéressante de la logique, de l'intelligence artificielle et surtout de PROLOG. Le point crucial est le fait qu'il suffit (surtout dans le cas de l'énigme du Zèbre) de donner l'énoncé du problème pour que PROLOG fournisse la solution. Les énigmes suivantes nous ont amenés à nuancer ce succès; ne tenir compte que des aspects logiques, déclaratifs, de PROLOG, en négligeant les aspects opérationnels, conduirait à faire de PROLOG un usage naïf, inefficace et le plus souvent inapproprié. Plus généralement, c'est l'ensemble des techniques de l'intelligence artificielle qu'il convient d'utiliser avec circonspection, et seulement dans le cas où les techniques de programmation plus classiques ne s'appliquent pas. Dans ce paragraphe, nous donnons un (contre-)exemple célèbre, montrant la supériorité de l'approche classique sur l'approche "intelligente". On montre aussi, cependant, que le passage de l'une à l'autre n'est pas une rupture radicale, mais peut se faire méthodiquement.

12.2.1. *Un tri "orienté IA"*

Le problème du tri est l'un des plus étudiés de l'algorithmique classique et donc, a priori, l'un des moins appropriés pour une approche orientée vers l'intelligence artificielle. Il n'est cependant pas difficile, en principe, de résoudre le problème de tri en PROLOG comme s'il s'agissait d'une énigme. En effet, la version triée d'une liste est, par définition, la *permutation ordonnée* de cette liste, ce qui suggère immédiatement le programme suivant:

```
intelsort(Xs,Ys) :- permutation(Xs,Ys), ordered(Ys).
```

Ce programme est acceptable, pour peu que l'on définisse les deux prédicats auxiliaires `permutation` et `ordered`. Ces deux prédicats sont élémentaires et on peut écrire immédiatement

```
permutation([],[]).
permutation([X|Xs],[U|Us]) :-  selection(U,[X|Xs],Zs),
                               permutation(Zs,Us).

selection(X,[X|Xs],Xs).
selection(X,[Y|Ys],[Y|Zs]) :- selection(X,Ys,Zs).

ordered([]).
ordered([X]).
ordered([X,Y|Zs]) :- X =< Y, ordered([Y|Ys]).
```

Cet ensemble de prédicats fonctionne correctement; on a par exemple

```
?- intel_sort([3,2,1,4],Xs).
Xs = [1,2,3,4].
```

Cependant, l'extrême lenteur de l'exécution rend ce programme sans intérêt pratique; il ne permet pas de trier une liste de 20 éléments en un temps raisonnable. Une variante du prédicat `permutation` permet d'améliorer légèrement les choses:

```
permutation([],[]).
permutation([X|Xs],Zs) :- permutation(Xs,Us), selection(X,Zs,Us).
```

Dans la version initiale, le prédicat `selection` était utilisé pour extraire un élément d'une liste; dans la nouvelle version, il est utilisé pour insérer un élément dans une liste. Le problème est toutefois ailleurs; on observe que la méthode de tri consiste à générer des permutations de la liste à trier et à les tester, jusqu'à ce que la bonne permutation soit découverte. Le fait que le nombre $n!$ de permutations augmente très vite en fonction de la longueur n de la liste à trier explique les mauvaises performances de l'algorithme. Plus concrètement, l'approche "intelligente" est ici directement responsable de l'échec. En effet, on demande au système PROLOG de trier une liste sur base d'un "indice" qui permet d'isoler la permutation triée parmi les $n!$ permutations de la liste fournie. Cet indice se révèle insuffisant en pratique.

12.2.2. *Vers un tri raisonnablement efficace*

On peut observer qu'un indice crucial manque à PROLOG: il n'est pas nécessaire de construire entièrement une permutation si la partie déjà construite comporte deux éléments se succédant dans l'ordre décroissant. (On a supposé que le tri se faisait dans l'ordre croissant.) Un premier essai d'intégration de cet indice s'obtient à partir de la première version du prédicat `permutation`:

```
sorted_permutation([],[]).
sorted_permutation([X|Xs],[U|Us]) :-  selection(U,[X|Xs],Zs),
                                      sorted_permutation(Zs,Us),
                                      ordered([U|Us]).
```

Cet essai est clairement infructueux parce que le test `ordered` intervient trop tard. La variante du prédicat `permutation` se prête mieux à l'intégration de l'indice.

```
sorted_permutation([],[]).
sorted_permutation([X|Xs],Zs) :- sorted_permutation(Xs,Us),
                                 selection(X,Zs,Us),
                                 ordered(Zs).
```

Le test `ordered` intervient ici suffisamment tôt pour accélérer radicalement l'exécution du programme. On observe néanmoins que, `Zs` devant être trié, il est indispensable que `Us` le soit aussi, ce qui nous conduit à la variante

```
sorted_permutation([],[]).
sorted_permutation([X|Xs],Zs) :- sorted_permutation(Xs,Us),
                                 ordered(Us),
                                 selection(X,Zs,Us),
                                 ordered(Zs).
```

Le gain d'efficacité est toutefois compromis par un gaspillage; la liste `Zs` résulte de l'insertion dans la liste triée `Us` de l'élément `X`; il ne serait pas nécessaire de tester l'ordonnancement de `Zs` si `X` était inséré à la bonne place. Cela suggère l'amélioration suivante:

```
sorted_permutation([],[]).
sorted_permutation([X|Xs],Zs) :- sorted_permutation(Xs,Us),
                                 ordered(Us),
                                 sorted_select(X,Zs,Us).


sorted_select(X,[X],[]).
sorted_select(X,[X,Y|Ys],[Y|Ys]) :- X =< Y.
sorted_select(X,[Y|Zs],[Y|Ys]) :- X > Y, sorted_select(X,Zs,Ys).
```

Enfin, on montre facilement par récurrence sur la longueur de la liste à trier que, dans le code de `sorted_permutation`, le prédicat `ordered(Us)` est inutile; en effet, chaque insertion se fait maintenant à sa place.

12.2.3. *Approche directe du problème de tri*

On peut aborder le problème du tri de manière entièrement différente. En effet, le tri est une activité banale, à laquelle on se livre fréquemment; il suffira donc de formaliser cette activité pour obtenir un programme de tri. Considérons par exemple le cas d'un joueur de cartes, qui souhaite trier sa main (par exemple dans l'ordre ♡, ♠, ♢, ♣ et, au sein d'une même couleur, dans l'ordre `A`, `R`, `D`, `V`, `10`, `9`, `8`, `7`, `6`, `5`, `4`, `3`, `2`). Une technique habituelle consiste à ramasser les cartes une à une et à les insérer directement à leur place, au fur et à mesure. Quand le joueur ramasse la treizième carte, il a déjà trié les douze premières; il lui reste à parcourir visuellement son jeu, par exemple de gauche à droite, pour ranger la dernière carte à l'endroit approprié. Cela suggère le code suivant:

```
insert_sort([],[]).
insert_sort([X|Xs],Zs) :- insert_sort(Xs,Us),
                          insert(X,Us,Zs).
```

Insérer une carte dans une main vide est immédiat; si par contre la main comporte au moins une carte, on compare la carte à insérer à la première carte de la main; on obtient aisément le code suivant:

```
insert(X,[],[X]).
insert(X,[Y|Ys],[X,Y|Ys]) :- X =< Y.
insert(X,[Y|Ys],[Y|Zs]) :- X > Y, insert(X,Ys,Zs).
```

On a obtenu un algorithme de tri classique, appelé "tri par insertion". Il est intéressant de noter que, en rebaptisant les prédicats et en inversant les deuxième et troisième arguments du prédicat auxiliaire, on retombe sur la dernière version de l'algorithme de tri développé au paragraphe précédent (prédicat principal `sorted_permutation`, prédicat auxiliaire `sorted_select`).

Une autre technique de tri familière consiste à considérer simultanément tous les objets à trier et à sélectionner successivement le premier, le second (le premier des objets qui restent), etc. Cette idée donne lieu immédiatement au code suivant ("tri par sélection"):

```
sel_sort([],[]).
sel_sort([X|Xs],[Y|Ys]) :- first([X|Xs],Y),
                           selection(Y,[X|Xs],Zs),
                           sel_sort(Zs,Ys).


first([X],X).
first([Y,X|Xs],Z) :- first([X|Xs],U), smallest(Y,U,Z).

smallest(Y,U,Y) :- Y =< U.
smallest(Y,U,U) :- Y > U.
```

12.2.4. *Le tri rapide*

Le "tri des joueurs de cartes", c'est-à-dire le tri par insertion présenté au paragraphe 12.2.3, est qualifié de raisonnablement efficace parce que le temps mis à trier n données est proportionnel au carré de n. En effet, pour trier n données on effectue n insertions dans des listes déjà triées, mais les longueurs respectives de ces listes sont $0, 1, 2, \ldots, n-1$, et le temps nécessaire à réaliser une insertion est naturellement proportionnel à la longueur de la liste dans laquelle on insère.

La méthode dite de tri rapide ("quicksort") procède différemment. Pour trier n objets, on exécute l'algorithme récursif suivant :

- Si $n = 0$ ou $n = 1$, c'est déjà trié;
- Si $n > 1$, on procède comme suit :
  - on choisit un objet, appelé "pivot";
  - on classe les autres objets en "petits" et "grands" par rapport au pivot;
  - les deux parties sont triées séparément ...
  - puis concaténées (sans oublier le pivot) en une liste triée.

Le prédicat de tri rapide se définit immédiatement :

```
quick([], []).
quick([X|Xs], Ys) :- part(X, Xs, Low, High),
                     quick(Low, L), quick(High, H),
                     append(L, [X|H], Ys).
```

Si `part(X, Xs, Low, High)` est vrai, alors `Low` doit contenir les éléments de `Xs` inférieur à `X` tandis que `High` doit contenir les autres. On a

```
part(A, [], [], []).
part(A, [X|Xs], [X|Ys], Zs) :- before(X, A),
                               part(A, Xs, Ys, Zs).
part(A, [X|Xs], Ys, [X|Zs]) :- not before(X, A),
                               part(A, Xs, Ys, Zs).
```

On suppose que le prédicat `before(X,Y)` est vrai si `X` doit précéder `Y` dans le classement, c'est-à-dire si `X` est inférieur à `Y`.

On peut montrer que cet algorithme devient nettement plus efficace que le tri par insertion si le nombre d'objets à trier est relativement élevé et si le pivot est en général bien choisi (les "petits" éléments sont à peu près aussi nombreux que les "grands").

12.2.5. *Le problème des partitions*

L'avantage immédiat du système PROLOG par rapport à d'autres langages de programmation est sa gestion automatique du non-déterminisme. Plus concrètement, PROLOG peut répondre correctement et exhaustivement aux questions admettant plusieurs réponses. Nous avons vu que cet avantage est décisif dans certains problèmes d'intelligence artificielle mais il est important de souligner son grand intérêt pour des problèmes plus classiques; nous illustrons ce fait en traitant le problème des partitions d'abord classiquement, puis en exploitant le non-déterminisme du prédicat `append`.

Une partition d'un ensemble E est une famille de parties non vides de E telle que chaque élément de E appartienne à exactement un membre de la famille. Le problème consiste à construire toutes les partitions d'un ensemble donné. A titre d'exemple, les partitions de l'ensemble $\{a, b, c\}$ sont au nombre de cinq :

$$\{\{a, b, c\}\},$$
$$\{\{a\}, \{b, c\}\},$$
$$\{\{a, b\}, \{c\}\},$$
$$\{\{a, c\}, \{b\}\},$$
$$\{\{a\}, \{b\}, \{c\}\}.$$

Ce problème, comme la plupart des problèmes d'analyse combinatoire, se résout par récurrence : comment construit-on une partition d'un ensemble à $n+1$ éléments au départ d'une partition d'un ensemble à n éléments ? Il existe deux procédés : on ajoute une partie comportant uniquement le nouvel élément, ou on insère ce nouvel élément dans une partie existante. On résout aussi explicitement le cas de base : un ensemble à 0 élément est l'ensemble vide. Il admet une seule partition, l'ensemble vide lui-même.

L'approche classique pour obtenir les partitions de l'ensemble $\{x\} \cup E$ consiste à construire (en deux exemplaires) la liste des partitions de l'ensemble E puis à transformer ces listes, l'une par le premier procédé, l'autre par le second, pour obtenir les partitions de l'ensemble $\{x\} \cup E$. On représente un ensemble tel que $\{a, b, c\}$ par la liste `[a,b,c]` et une partition telle que $\{\{a, c\}, \{b\}\}$ par la liste `[[a,c],[b]]`. Le programme principal s'obtient immédiatement :

```
c_partitions([],[[]]).
c_partitions([X|Xs],Ps) :- c_partitions(Xs,Qs),
                           proc_1(X,Qs,P1s), proc_2(X,Qs,P2s),
                           append(P1s,P2s,Ps).
```

Appliquer le premier procédé avec l'élément a et la liste des partitions de $\{b, c\}$ consiste à insérer la partie $\{a\}$ dans chacune des partitions, ce qui se réalise facilement :

```
proc_1(X,Qs,Rs) :- insert_in_all([X],Qs,Rs).

insert_in_all(U,[],[]).
insert_in_all(U,[Us|Uss],[[U|Us]|Vss]) :-
            insert_in_all(U,Uss,Vss).
```

Appliquer le second procédé avec l'élément a et la liste des partitions de

$\{b, c\}$ consiste à "éclater" chaque partition en une liste de partitions; par exemple, la partition $\{\{b\}, \{c\}\}$ devient la liste des deux partitions $\{\{a, b\}, \{c\}\}$ et $\{\{b\}, \{a, c\}\}$. Ces listes de partitions sont concaténées en une liste globale. On a :

```
proc_2(X,[],[]).
proc_2(X,[Q|Qs],Ps) :- split(X,Q,R1s), proc_2(X,Qs,R2s),
                       append(R1s,R2s,Ps).

split(X,[],[]).
split(X,[Xs|Xss],[[[X|Xs]|Xss]|Ysss]) :-
     split(X,Xss,Xsss), insert_in_all(Xs,Xsss,Ysss).
```

Tout ceci est raisonnablement simple et on n'a utilisé `append` que dans le sens direct, pour concaténer des listes. Si maintenant on s'autorise à utiliser la réversibilité de `append`, on peut apporter au problème des partitions une solution nettement plus simple :

```
partition([],[]).

partition([X|Xs],[[X]|P]) :- partition(Xs,P).

partition([X|Xs],Q) :- partition(Xs,P),
                       append(R1,[R|R2],P),
                       append(R1,[[X|R]|R2],Q).
```

Le prédicat `partition` construit une à une, via son second argument, les partitions de l'ensemble (la liste) spécifié par son premier argument. Soulignons que, dans la dernière clause, la première occurrence de `append` décompose une liste `P` tandis que la seconde occurrence construit une liste `Q`. On utilise le prédicat spécial `findall` pour regrouper (si on le souhaite) toutes les partitions en une liste :

```
partitions(Xs,Ps) :- findall(P,partition(Xs,P),Ps).
```

("Trouver tous les `P` tels que `partition(Xs,P)` soit vrai et les regrouper dans une liste `Ps`.")

Voici deux exemples d'utilisation :

```
?- c_partitions([a,b,c],Ps).

?- partitions([a,b,c],Ps).
```

La réponse est la même dans les deux cas :

```
?- c_partitions([a,b,c],Ps).
Ps = [[[a], [b], [c]],
      [[a], [b,c]],
      [[a,b], [c]],
      [[b], [a,c]],
      [[a,b,c]]]
```


12.2.6. *Conclusion*

L'expérience confirme qu'un problème pour lequel des mécanismes algorithmiques sont connus n'est pas, ou n'est plus, du ressort de l'intelligence artificielle. Cela n'est pas vraiment étonnant; appliquer une recette met moins de temps, en règle générale, que découvrir cette recette ou apprendre à s'en passer. Par contre, il apparaît que le paradigme de programmation logique est utile tant dans le domaine de l'intelligence artificielle que dans celui de la programmation classique; de plus, la frontière entre ces deux domaines est assez floue.

L'expérimentation menée à propos du problème de tri explique l'influence très importante que peuvent avoir sur la vitesse d'exécution certains "détails" de programmation, comme l'ordre des clauses d'un programme et, plus encore, l'ordre des prédicats au sein d'une clause. L'explication a posteriori et, mieux encore, la prévision de ces variations, ne sont pas du ressort de la logique. Soulignons quand même un principe semblant se dégager de ces expériences; il ne faut pas attendre qu'un objet soit entièrement construit pour le tester et éventuellement le rejeter car cela permet d'épargner le temps consacré à finir la construction d'un objet inutile. Plus simplement, il faut "échouer aussi tôt que possible". Ce principe s'interprète facilement. Résoudre un problème intelligent implique nécessairement le risque de se tromper (si ce risque était absent, le problème ne serait pas qualifié d'intelligent). La clef du succès n'est donc pas d'éviter systématiquement les échecs provisoires mais de détecter ceux-ci au plus tôt, de manière à pouvoir réaliser rapidement d'autres essais.

12.3. Un exemple d'apprentissage

12.3.1. *Introduction*

12.3.1.1. *Hexapion*

Le jeu d'Hexapion introduit au paragraphe 10.2.5 permet d'illustrer simplement une autre potentialité de la programmation logique en intelligence artificielle. On a vu que ce jeu, comme beaucoup de jeux de stratégie, pouvait être décrit, au moins en principe, par un arbre de jeu représentant l'ensemble des parties possibles. Une vue partielle de l'arbre de jeu relatif à l'Hexapion est représenté à la figure 10.7. On conçoit aisément que cet arbre puisse être inclus dans un programme PROLOG, qui permettrait d'obtenir un joueur optimal pour l'Hexapion. Il ne serait pas nécessaire, pour cela, d'avoir développé l'arbre "à la main" : un autre programme PROLOG pourrait construire cet arbre, d'une manière qui le rende directement exploitable par le premier programme.

12.3.1.2. *Principe simple d'apprentissage*

Une autre approche, a priori plus ambitieuse mais aussi plus conforme à l'esprit de l'intelligence artificielle, consiste à construire un programme connaissant les règles du jeu mais pas la stratégie, qu'il doit acquérir en jouant contre un adversaire humain. Il s'agit donc de réaliser un apprentissage artificiel. Quoique la faculté d'apprentissage soit un aspect particulièrement important de l'intelligence, simuler cette faculté est relativement aisé. Il faut cependant observer que le principe même de l'apprentissage suppose que l'entité apprenante (ici, un programme PROLOG) évolue au cours du temps.

Nous supposons que le programme devra jouer systématiquement avec les pions blancs, ce qui, comme indiqué au paragraphe 10.2.5,[5] est un désavantage: si les Noirs ne commettent aucune erreur, les Blancs (qui commencent toujours la partie) perdent toujours. Le principe du programme est qu'il mémorise l'historique des parties qu'il joue, et qu'il ne commette pas deux fois la même erreur. Il peut donc, au bout d'un certain nombre de parties, ne plus jouer du tout, puisqu'une stratégie gagnante existe pour l'adversaire.

A priori, le meilleur choix pour un joueur se trouvant dans cette situation d'infériorité consiste à mettre à profit les erreurs de l'adversaire, qui peut ne pas connaître la stratégie gagnante, et à jouer des coups d'attente, plus ou moins au hasard, quand l'adversaire ne commet pas d'erreur. Dans le cas présent, pour mettre en évidence la faculté d'apprentissage, nous faisons un choix extrême: le programme refuse de jouer et reconnaît sa défaite quand il atteint une situation à partir de laquelle chaque coup possible pour lui s'est déjà avéré perdant lors d'une partie précédente. Le programme considère donc que, tout comme lui, son adversaire apprend et tire parti de ses erreurs passées. Cela signifie que si l'adversaire humain est raisonnablement compétent, le programme finira par refuser de jouer. Dans le cas contraire, il pourra jouer indéfiniment ... ce qui traduit le fait que la qualité de l'apprentissage dépend de la qualité de l'enseignement.

12.3.2. *Le programme de jeu*

12.3.2.1. *Quelques mécanismes particuliers*

Pour permettre l'évolution de sa base de connaissance, le système PROLOG dispose d'instructions spéciales `assert` et `retract` qui lui permettent respectivement d'ajouter à sa base de connaissance une clause (donnée en argument) en cours d'exécution, ou d'en supprimer une.

Le système utilise aussi les instructions `read` et `write`; la première lui permet d'appréhender les coups joués par l'adversaire et la seconde permet

[5] Le lecteur est invité à relire les règles du jeu d'Hexapion dans ce paragraphe.

de communiquer à celui-ci ses propres coups et de récapituler à son intention la situation actuelle du jeu. L'instruction `nl` ("new line") permet d'aller à la ligne.

Un troisième mécanisme, appelé "coupure" et représenté par un point d'exclamation, permet de programmer efficacement certaines situations. Sans entrer dans des détails opératoires qui dépasseraient le cadre de cet exposé introductif, signalons que le fragment de programme:

```
A :- B, !, C.
A :- D.
```

représente souvent:

```
A :- si B alors C sinon D.
```

ce qui dans certains cas pourrait aussi s'écrire:

```
A :- B, C.
A :- not B, D.
```

12.3.2.2. *Le principe du jeu*

Au départ, le programme doit entamer la partie, sauf s'il a déjà déterminé qu'une stratégie gagnante existe pour l'adversaire. Ceci est modélisé par les clauses suivantes:

```
init([p(black, 1, 1), p(black, 2, 1), p(black, 3, 1),
      p(white, 1, 3), p(white, 2, 3), p(white, 3, 3)]).

hexa :- init(S), not bad(S), learn_maybe(S), play(white, S).

hexa :- init(S), bad(S), nl, write('I cannot win.'), nl.
```

Il est avantageux ici d'utiliser la coupure, ce qui donne:

```
hexa :- init(S), not bad(S), !, learn_maybe(S), play(white, S).

hexa :- nl, write('I cannot win.'), nl.
```

La variable `S` s'instancie en la situation initiale; le prédicat `bad` s'applique à une situation et est vrai si la situation a été reconnue comme perdante. Le prédicat `learn_maybe` sert à l'apprentissage, nous y reviendrons plus loin. Le prédicat `play` sert à générer les coups. On a:

```
play(C, S) :- chessboard(S), ch_color(C, C1), win(C1, S), !,
              write('The winner is : '), write(C1),
              nl, winner(C1), abort.
play(black, S) :- human(black, S, S_aft), play(white, S_aft).
play(white, S) :- computer(white, S, S_aft), play(black, S_aft).
```

La première clause définissant `play` correspond à la fin de la partie, les deux autres à un coup du programme et à un coup de l'adversaire humain. Quelques prédicats auxiliaires sont utilisés ici :

```
win(C, S) :- ch_color(C, C1), not legal(C1, M, S).
win(black, S) :- member(p(black, X, 3), S).
win(white, S) :- member(p(white, X, 1), S).

human(C,S1,S2) :- learn_maybe(S1), ask_human(C,M,S1),
                  move(M,S1,S2).

computer(C,S1,S2) :- possibility(C, S1, Ms), !,
                     choose_move(S1, Ms, M), move(M, S1, S2).
computer(C, S1, S2) :- learn(C, S1), end_msg, abort.

winner(black) :- !, maybe(S), forget_maybe(S), learn_bad(S).
winner(white).


end_msg :- nl, write('I know that you are the winner').
```

Le prédicat `win` permet de détecter la fin de la partie. Le prédicat `winner` permet au système, en cas de gain de l'adversaire, de faire passer la situation `S` de la qualification "maybe" (jouable) à la qualification "bad" (perdante). Rappelons que le système refuse de jouer si cela le conduit nécessairement dans une situation perdante.

12.3.2.3. *Les coups possibles*

Le système maintient une liste de coups possibles, qui sont licites et ne conduisent pas à une situation reconnue comme perdante; il donne la priorité à un coup conduisant immédiatement à la victoire, s'il en existe un.

```
choose_move(S, [M|Ms], M1) :- choose_win(S, [M|Ms], M1), !.
choose_move(S, [M|Ms], M).

choose_win(S,[M|Ms],M) :- move(M,S,S1), win(white,S1), !.
choose_win(S,[M|Ms],M1) :- choose_win(S,Ms,M1).

possibility(C, S, [M1|M1s]) :- findall(M, legal(C, M, S), Ms),
        sub_bad(S, Ms, [M1|M1s]), maybe(S1), forget_maybe(S1).

legal(black, m(p(black, X, Y), p(black, X, Y1), forward), S) :-
        member(p(black, X, Y), S), incr(Y, Y1), free(X, Y1, S).
legal(white, m(p(white, X, Y), p(white, X, Y1), forward), S) :-
        member(p(white, X, Y), S), decr(Y, Y1), free(X, Y1, S).
```



```
legal(C, m(p(C, X, Y), p(C, X1, Y1), eat), S) :-
        member(p(C, X, Y), S), diag(C, X, Y, X1, Y1),
        ch_color(C, C1), member(p(C1, X1, Y1), S).

move(m(P1, P2, forward), S_bef, S_aft) :-
     sub(P1, P2, S_bef, S), sort(S, S_aft).
move(m(P1, P2, eat), S_bef, S_aft) :- P2 = p(C2, X, Y),
     ch_color(C2, C), remove(p(C, X, Y), S_bef, S1),
     sub(P1, P2, S1, S), sort(S, S_aft).
```

Les mouvements effectués impliquent une gestion de la configuration des pions sur l'échiquier; nous ne rentrons pas ici dans les détails. Le prédicat `sort` permet de trier la configuration, vue comme la liste des positions des pions. Nous donnons sans commentaires les clauses servant à manipuler les configurations :

```
is_range(X)  :- X>0, X=<3.

incr(X, X1) :- X<3, X1 is X+1.
decr(X, X1) :- X>1, X1 is X-1.

diag(black, X, Y, X1, Y1) :- incr(Y, Y1), incr(X, X1).
diag(black, X, Y, X1, Y1) :- incr(Y, Y1), decr(X, X1).
diag(white, X, Y, X1, Y1) :- decr(Y, Y1), incr(X, X1).
diag(white, X, Y, X1, Y1) :- decr(Y, Y1), decr(X, X1).

ch_color(white, black).        ch_color(black, white).

free(X, Y, S) :- not member(p(C,X,Y), S).
```

12.3.3. *Le dialogue avec l'adversaire*

Nous donnons aussi sans commentaire les clauses qui régissent le dialogue avec l'adversaire humain :

```
ask_human(C, m(P1,P2,Ty), S) :- can_move_it(C, P1, S),
                                where(m(P1,P2,Ty), S).

read_x(X) :- write('X: '), read(X), is_range(X), !.
read_x(X) :- read_x(X).

read_y(Y) :- write('Y: '), read(Y), is_range(Y), !.
read_y(Y) :- read_y(Y).

coord(X, Y) :- write('Enter the coordinate: '),
               nl, read_x(X), read_y(Y).
```

```
which_pawn(C, p(C, X, Y), S) :- nl, write('Which '), write(C),
   write(' pawn do you want to move ?'),
   nl, coord(X, Y), member(p(C, X, Y), S), !.
which_pawn(C, P, S) :-
   write('There is not a '), write(C), write(' pawn there !'),
   nl, which_pawn(C, P, S).

can_move_it(C, P, S) :-
   which_pawn(C, P, S), legal(C, m(P, P1, Type), S), !.
can_move_it(C, P, S) :-
   write('There is not a legal move for it!'),
    nl, can_move_it(C, P, S).

where(m(p(C, X1, Y1), p(C, X2, Y2), Type), S) :- nl,
   write('Where do you want to move it ?'), nl, coord(X2, Y2),
   legal(C, m(p(C, X1, Y1), p(C, X2, Y2), Type), S), !.
where(M, S) :- write('This is not a legal move !'),
               nl, where(M, S).
```

12.3.3.1. *Gestion de l'apprentissage*

Le prédicat `show` permet l'affichage, à tout moment de la partie, des situations considérées comme mauvaises par le système. Les prédicats `learn` et `forget` permettent d'actualiser les listes de situations reconnues comme jouables ou comme mauvaises. On a :

```
show :- findall(S, bad(S), Ss), show_sit(Ss).

learn(C, S) :- findall(M, legal(C, M, S), Ms),
     forget(S, Ms), maybe(S1), forget_maybe(S1), learn_bad(S1).

forget(S, [M|Ms]) :- move(M, S, S1),
                     forget_bad(S1), forget(S, Ms).
forget(S, []).

sub_bad(S1, [M|M1s], M2s) :- move(M, S1, S2), bad(S2), !,
                             sub_bad(S1, M1s, M2s).
sub_bad(S1, [M|M1s], [M|M2s]) :- sub_bad(S1, M1s, M2s).
sub_bad(S1, [], []).

show_sit([[P|Ps]|Ss]) :- chessboard([P|Ps]), show_sit(Ss).
show_sit([[]|Ss]) :- show_sit(Ss).
show_sit([]).
```



```
learn_bad([P|Ps]) :- asserta((bad([P|Ps]))).
learn_bad([]).

forget_bad([P|Ps]) :- retract((bad([P|Ps]))).
forget_bad([]).

learn_maybe([P|Ps]) :- asserta((maybe([P|Ps]))).
learn_maybe([]).

forget_maybe([P|Ps]) :- retract((maybe([P|Ps]))).
forget_maybe([]).

bad([]).

maybe([]).
```

12.3.3.2. *Prédicats auxiliaires supplémentaires*

Nous donnons ici la liste des prédicats auxiliaires permettant notamment de dessiner l'échiquier à l'écran et de trier une situation.

```
chessboard(S) :- nl, nl, draw_coord_col, draw_border,
                 draw_board(1, S), draw_border, nl.

draw_board(Y,S) :- draw_line(Y,S), incr(Y,Y1), !,
                   draw_board(Y1,S).
draw_board(Y,S).

draw_line(Y,S) :- write(' '),write(Y),write(' |'),
                  draw_line(1,Y,S),write('|'),nl.
draw_line(X,Y,S) :- draw_cell(X,Y,S),incr(X,X1),!,
                    draw_line(X1,Y,S).
draw_line(X, Y, S).

draw_cell(X, Y, S) :- member(p(C, X, Y), S), !, draw_pawn(C).
draw_cell(X, Y, S) :- write(' ').

draw_pawn(white) :- write('W').
draw_pawn(black) :- write('B').

draw_coord_col :- Xd is 3//10, write('    '), draw_ten(1, Xd),
                  nl, write('    '), draw_num(3), nl.

draw_ten(N,M) :- N=<M,!,write('         '),write(N),
```

```
                N1 is N+1, draw_ten(N1,M).
draw_ten(N, M).

draw_num(N) :- N>9,number(9,Xs),draw(Xs),write('0'),
               N1 is N-10, draw_num(N1).
draw_num(N) :- N=<9, N>0, number(N, Xs), draw(Xs).
draw_num(0).

draw_border :- make_list(3,'-',Bord), write('   +'),
               draw(Bord), write('+'), nl.

remove(X, [X|Xs], Xs).
remove(X, [Y|Ys], [Y|Zs]) :- X\==Y, remove(X, Ys, Zs).

sub(X, Y, [X|Xs], [Y|Xs]).
sub(X, Y, [Z|Xs], [Z|Ys]) :- sub(X, Y, Xs, Ys).

make_list(N, X, [X|Xs]):- N>0, N1 is N-1, make_list(N1, X, Xs).
make_list(0, X, []).

draw([X|Xs]) :- write(X), draw(Xs).
draw([]).

number(N, Xs) :- number(1, N, Xs).
number(N, N, [N]).
number(S, N, [S|Xs]) :- S<N, S1 is S+1, number(S1, N, Xs).

before(p(black, X1, Y1), p(white, X2, Y2)) :- !.
before(p(C, X1, Y1), p(C, X2, Y2)) :- X1 < X2, !.
before(p(C, X1, Y1), p(C, X2, Y2)) :- Y1 < Y2.

sort  /* procedure de tri, voir section 3.2 */
```

12.3.4. *Exemples d'exécution*

Nous avons donné le texte complet du programme de jeu mais le lecteur non informaticien préférera appréhender les principes de ce programme par le biais de quelques exécutions commentées.

12.3.4.1. *Première partie*

Le système joue sans tactique mais en respectant les règles. Il joue avec les pions `W` (white); l'adversaire humain répond avec les pions `B` (black).


```
?- hexa.

    123     123
   +---+   +---+
 1 |BBB| 1 |BBB|
 2 |   | 2 |W  |
 3 |WWW| 3 | WW|
   +---+   +---+

Which black pawn do you want to move ?
Enter the coordinate:
X: 2.
Y: 1.

Where do you want to move it ?
Enter the coordinate:
X: 1.
Y: 2.

    123     123
   +---+   +---+
 1 |B B| 1 |B B|
 2 |B  | 2 |BW |
 3 | WW| 3 |  W|
   +---+   +---+
```

Le haut de cette figure représente la situation initiale et la situation après le premier coup du système. Le milieu représente le dialogue avec l'adversaire humain, qui déclare vouloir prendre le pion blanc avancé au moyen du pion noir médian; celui-ci passe donc de sa position initiale `(2,1)` à la position `(1,2)`, comme indiqué en bas à gauche de la figure. Le système réagit en avançant son propre pion médian, ce qui conduit à la situation représentée en bas à droite de la figure.

En présence d'un adversaire humain jouant convenablement, la partie s'achève immédiatement :

```
    123     123
   +---+   +---+
 1 |B B| 1 |B B|
 2 |BW | 2 | W |
 3 |  W| 3 |B W|
   +---+   +---+

The winner is : black
Execution Aborted
```

Les Noirs ont gagné parce qu'un de leur pion a atteint la ligne opposée. Le système apprend à chaque défaite; il mémorise le fait que sa dernière position était perdante, ce que l'on met en évidence en appelant le prédicat sans argument `show` :

```
?- show.

   123
  +---+
1 |B B|
2 |BW |
3 |  W|
  +---+
```

12.3.4.2. *Deuxième partie*

Nous donnons simplement la liste des situations successives. Rappelons que le système joue toujours le premier, avec les Blancs (`W`).

```
   123     123     123     123     123
  +---+   +---+   +---+   +---+   +---+
1 |BBB| 1 |BBB| 1 |B B| 1 |B B| 1 |B B|
2 |   | 2 |W  | 2 |B  | 2 |B W| 2 |  W|
3 |WWW| 3 | WW| 3 | WW| 3 | W | 3 |BW |
  +---+   +---+   +---+   +---+   +---+
```

```
The winner is : black
Execution Aborted
```

Le système a de nouveau perdu, il mémorise une nouvelle situation perdante :

```
   123    123
  +---+   +---+
1 |B B| 1 |B B|
2 |B W| 2 |BW |
3 | W | 3 |  W|
  +---+   +---+
```

12.3.4.3. *Progression de l'apprentissage*

Supposons que l'adversaire humain commette une erreur; le système peut éventuellement en profiter et gagner la partie mais, dans ce cas, il n'apprend rien et sa liste de mauvaises situations reste inchangée.

A l'issue d'une troisième partie perdante, la liste des mauvaises situations sera :


```
   123     123     123
  +---+   +---+   +---+
1 |B B| 1 |B B| 1 |B B|
2 |W  | 2 |B W| 2 |BW |
3 |  W| 3 | W | 3 |  W|
  +---+   +---+   +---+
```

Si l'adversaire humain ne commet pas d'erreur, la partie suivante pourrait être celle-ci :

```
   123     123     123
  +---+   +---+   +---+
1 |BBB| 1 |BBB| 1 |B B|
2 |   | 2 |W  | 2 |B  |
3 |WWW| 3 | WW| 3 | WW|
  +---+   +---+   +---+
```

```
I know that you are the winner
```

Le système renonce à continuer la partie parce qu'il n'a que trois possibilités, chacune d'elle conduisant à une situation interdite :

- Prendre le pion noir avancé avec le pion blanc médian.
- Avancer le pion blanc latéral.
- Avancer le pion blanc médian.

Cette nouvelle défaite du système améliore l'apprentissage; le système sait maintenant que la situation résultant d'une entame latérale est perdante, ce que l'on met en évidence en invoquant le prédicat `show` :

```
?- show.

   123
  +---+
1 |BBB|
2 |W  |
3 | WW|
  +---+
```

Lors de la partie suivante, le système effectuera une entame centrale :

```
?- hexa.

   123     123     123     123     123
  +---+   +---+   +---+   +---+   +---+
1 |BBB| 1 |BBB| 1 | BB| 1 | BB| 1 | BB|
2 |   | 2 | W | 2 | B | 2 |WB | 2 |W  |
3 |WWW| 3 |W W| 3 |W W| 3 |  W| 3 | BW|
  +---+   +---+   +---+   +---+   +---+
```

```
The winner is : black
Execution Aborted
```

Après avoir perdu une vingtaine de parties, et ceci indépendamment du nombre de parties qu'il peut avoir gagnées, l'ensemble des mauvaises situations est le suivant :

```
   123     123     123
  +---+   +---+   +---+
1 |BBB| 1 |BBB| 1 |BBB|
2 |W  | 2 | W | 2 |  W|
3 | WW| 3 |W W| 3 |WW |
  +---+   +---+   +---+
```

Il n'y a donc plus d'entame jouable; la partie suivante sera :

```
   123
  +---+
1 |BBB|
2 |   |
3 |WWW|
  +---+

I know that you are the winner
Execution Aborted
```

Le système renonce immédiatement; il apprend de ce fait qu'il a perdu avant de commencer, du moins si l'adversaire humain ne commet pas d'erreur; il a donc appris qu'il existait une stratégie gagnante pour son adversaire :

```
   123
  +---+
1 |BBB|
2 |   |
3 |WWW|
  +---+

I cannot win.
```

12.3.4.4. *Conclusion*

Le programme que nous avons présenté ici met bien en évidence la manière dont l'apprentissage peut s'intégrer à la programmation logique. On pourrait sans difficulté perfectionner le programme pour qu'il tienne indifféremment le rôle des Blancs ou celui des Noirs, et aussi pour que, en situation perdante, il joue un coup d'attente plutôt que de renoncer. Le système se comporterait alors comme un joueur en progrès, nul au début, puis graduellement optimal au fur et à mesure qu'il tire les leçons des parties perdues.

Il ne faudrait évidemment pas en déduire que quelques lignes de programme suffisent à doter un système d'une capacité d'apprentissage importante. En effet, si l'approche présentée ici convient parfaitement à un jeu tel que Hexapion, c'est parce qu'une partie d'Hexapion ne peut durer que six coups; le nombre de parties distinctes est donc relativement limité. Sur le plan théorique, la même technique pourrait être utilisée pour l'apprentissage des Echecs mais, une partie d'Echecs pouvant comporter une centaine de coups, l'acquisition d'une stratégie gagnante (s'il en existe une) prendrait un temps énorme; quels que soient les progrès futurs de la technologie des ordinateurs, il est impensable d'obtenir un programme de jeu performant par une technique d'apprentissage aussi simple.

12.4. Pour en savoir plus

Nous avons déjà mentionné à la fin du chapitre précédent diverses références concernant PROLOG et l'intelligence artificielle. La rubrique tenue par Martin Gardner dans la revue *Scientific American* a influencé le choix des applications que nous avons traitées ici. Tant pour les énigmes que pour l'intelligence artificielle elle-même, nous renvoyons le lecteur aux nombreux livres et recueils regroupant ces rubriques [Gardner]. C'est dans l'une d'elles, dix ans avant l'invention de PROLOG, qu'a été introduit le programme d'apprentissage d'Hexapion développé ici.

Bibliographie

[Ackermann 56] W. Ackermann, Begründung einer Strengen Implikation, *Journal of Symbolic Logic*, vol. 21, p. 113-128, 1956.

[Alimi et al. 99] J. Alimi, G. Dowek et L. Rolland, in N. Farouki (éditeur), *Quand la Science a dit...c'est impossible*, Le Pommier-Fayart, Paris 1999.

[Anderson and Belnap 62] A. Anderson and N. Belnap, The pure calculus of entailment, *The Journal of Symbolic Logic*, vol. 27, p. 19-52, 1962.

[Anderson and Belnap 68] A. Anderson and N. Belnap, Entailment, in G. Iseminger (éditeur), *Logic and Philosophy*, p. 76-110, Appelton-Century-Crofts, New York, 1968.

[Anderson and Belnap 75] A. Anderson and N. Belnap, *Entailment. The Logic of Relevance and Necessity*, vol. 1, Princeton University Press, Princeton, 1975.

[Anderson and Johnstone 62] A. Anderson and H. Johnstone, *Natural Deduction*, Wadsworth, Belmont, 1962.

[Aristote 59] Aristote, *Organon I, Catégories II, De l'Interprétation*, traduction de J. Tricot, Vrin, Paris, 1959.

[Aristote 47] Aristote, *Organon III, Les Premiers Analytiques*, traduction de J. Tricot, Vrin, Paris, 1947.

[Aristote 62] Aristote, *Organon IV, Les Seconds Analytiques*, traduction de J. Tricot, Vrin, Paris, 1962.

[Bailhache 91] P. Bailhache, *Essai de Logique Déontique*, Vrin, Paris, 1991.

[Barcan 46] R. Barcan, A functional calculus of first order based on strict

implication, *The Journal of Symbolic Logic*, vol. 11, no 1, March, p. 1-16, 1946.

[Barendregt 80] H. Barendregt, *The Lambda Calculus, its Syntax and Semantics*, North-Holland, Amsterdam, 1980.

[Barreau 84] H. Barreau, Diodore Cronos, in D. Huisman (éditeur), *Dictionnaire des Philosophes*, Presses Universitaires de France, p. 754-759, 1984.

[Basin et al. 97] D. Basin, S. Matthews and L. Vigano, Labelled quantified modal logic, in G. Brewka et al. (éditeurs), *KI-97 : Advances in Artificial Intelligence*, p. 171-182, Springer-Verlag, Berlin, 1997.

[Basin et al. 98] D. Basin, S. Matthews and L. Vigano, Labelled modal logics : quantifiers, *Journal of Logic, Language and Information*, vol. 7, p. 237-263, 1998.

[Batens 95] D. Batens, Blocks. The clue to dynamic aspects of logic, *Logique et Analyse*, juin-septembre, no 150-152, décembre, p. 285-327, 1995.

[Bayart 58] A. Bayart, La correction de la logique modale du premier et du second ordre $S5$, *Logique et Analyse I*, no 1, janvier, p. 28-45, 1958.

[Bayart 59] A. Bayart, Quasi-adéquation de la logique modale du second ordre $S5$ et adéquation de la logique modale du premier ordre $S5$, *Logique et Analyse II*, no 6-7, avril, p. 28-45, 1959.

[Bell and Slomson 69] J. Bell and A. Slomson, *Models and Ultraproducts*, North Holland, Amsterdam, 1969.

[Ben Ari 93] M. Ben Ari, *Mathematical Logic for Computer Sciences*, Prentice Hall, New York, 1993.

[Bencivenga 86] E. Bencivenga, Free logic, in D. Gabbay and F. Guenthner (éditeurs), *Handbook of Philosophical Logic, vol. 3, Alternatives in Classical Logic*, p. 373-426, 1986.

[Binot 90] J.-L. Binot, Traitement de la langue naturelle, logique et programmation, in A. Thayse (éditeur), *Approche Logique de l'Intelligence Artificielle, tome 3 : Du Traitement de la Langue à la Logique des Systèmes Experts*, Dunod, Paris, 1990.

[Blackburn et al.] P. Blackburn, M. de Rijke and I. Venema, *Modal Logic*, à paraître.

[Bläsius und Bürkhert 92] K. Bläsius und H. Bürkhert, *Automatisierung des Logischen Denkens*, 2ième édition, Oldenburg, Munich, 1992.

[Bloesch 93a] A. Bloesch, *Signed Tableaux. A Basis for Automated Proving in non Classical Logics*, Ph. D. Thesis, University of Queensland, Australia, 1993.

[Bloesch 93b] A. Bloesch, A tableau style proof system for two paraconsistent logics, *Notre Dame Journal of Formal Logic*, vol. 34, p. 295-301, 1993.

[Bratko 90] I. Bratko, *Prolog – Programming for Artificial Intelligence*, Addison-Wesley, Reading, Mass., 2ième édition, 1990.

[Carnap 36] R. Carnap, *The Logical Syntax of Language*, Routledge and Kegan, Londres, 1936.

[Carnap 46] R. Carnap, Modalities and quantification, *The Journal of Symbolic Logic*, vol. 11, no 2, June, p. 33-64, 1946.

[Carnap 47] R. Carnap, *Meaning and Necessity : A Study in Semantics and Modal Logic*, The University of Chicago Press, Chicago, 1947.

[Carnap 97] R. Carnap, *Signification et Nécessité*, traduction et présentation de F. Rivenc et Ph. de Rouilhan, Gallimard, Paris 1997.

[Castilho et al. 97] M. Castilho, L. Fariñas del Cerro, O. Gasquet and A. Herzig, Modal tableaux with propagation rules and structural rules, *Fundamenta Informaticae*, vol. 32, p. 291-297, 1997.

[Ceri et al. 90] S. Ceri, G. Gottlob and L. Tanca, *Logic Programming and Databases*, Springer-Verlag, Berlin, 1990.

[Chagrov and Zakharyaschev 97] A. Chagrov and M. Zakharyaschev, *Modal Logic*, Clarendon Press, Oxford, 1997.

[Chang and Keisler 73] C. Chang and H. Keisler, *Model Theory*, North-Holland, Amsterdam, 1973.

[Chellas 80] B. Chellas, *Modal Logic*, Cambridge University Press, Cambridge, 1980.

[Church 51] A. Church, The weak theory of implication. Kontrolliertes Denken, *Festgabe zum 60 Geburtstag van Prof. W. Britzelmayr*, Munich 1951.

[Clifford 90] J. Clifford, *Formal Semantics and Pragmatics for Natural Language Querying*, Cambridge Tracts in Theoretical Computer Science, vol. 8, 1990.

[Cocchiarella 66] N. Cocchiarella, *Tense Logic : A Study of Temporal Reference*, Ph. D. Thesis, U.C.L.A., 1966.

[Cocchiarella 84] N. Cocchiarella, Philosophical perspectives on quantification in tense and modal logic, in D. Gabbay and F. Guenthner (éditeurs), *Handbook of Philosophical Logic, vol. 2, Extensions in Classical Logic*, Reidel, Dordrecht, p. 309-353, 1984.

[Codd 70] E. Codd, A relational model of data for large shared data banks, *Communications of the ACM*, vol. 13, p. 377-387, 1970.

[Chomsky 59] N. Chomsky, On certain formal properties of grammars, *Information and Control*, vol. 2, p. 137-167, 1959.

[Colmerauer 78] A. Colmerauer, Metamorphosis grammars, in L. Bole (éditeur), *Natural Language Communication with Computers,* p. 139-169, *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, vol. 63, 1978.

[Colmerauer et al. 83] A. Colmerauer, H. Kanoui et M. Van Caneghem, Prolog, bases théoriques et développements actuels, *T.S.I.*, vol. 2, no 4, 1983.

[Copeland 96] J. Copeland (éditeur), *Logic and Reality, Essays on the Legacy of Arthur Prior*, Clarendon Press, Oxford, p. 1-40, 1996.

[Crabbé 86] M. Crabbé, Le calcul lambda, *Cahiers du Centre de Logique*, no 6 : *Logique et Informatique*, Cabay, Louvain-la-Neuve, p. 63-86, 1986.

[Crocco 96] G. Crocco, *Fondements Logiques du Raisonnement Contextuel. Une Etude sur les Logiques des Conditionnels*, Unipress, Padova, 1996.

[D'Agostino and Mondadori 94] M. D'Agostino and M. Mondadori, The taming of the cut. Classical refutations with analytic cut, *J. Logic Computat.*, vol. 4, p. 285-319, 1994.

[Davio et al. 78] M. Davio, J.-P. Deschamps and A. Thayse, *Discrete and Switching Functions*, McGraw-Hill, New York 1978.

[Dean et al. 95] T. Dean, J. Allen et Y. Aloimonos, *Artificial Intelligence, Theory and Practice*, Benjamin-Cummings, Redwood, California, 1995.

[Dekker 89] P. Dekker, Dynamic interpretation, negation and flexibility, *First European Summer School on Natural Language Processing, Logic and Knowledge Representation*, Groningen, June 5-17, 1989.

[Dekker 90] P. Dekker, Dynamic interpretation, flexibility and monotonicity, in M. Stokhof and L. Torenvliet (éditeurs), *Proceedings of the Seventh Amsterdam Colloquium, 1989*, Amsterdam, 1990.

[Delahaye 86] J. Delahaye, *Outils Logiques pour l'Intelligence Artificielle*, Eyrolles, Paris, 1986.

[de Rijke 93] M. de Rijke, *Extending Modal Logic*, Dissertation, Institute for Logic, Language and Computation, University of Amsterdam, 1993.

[De Swart 94] H. De Swart, *Mathematics, Language, Computer Science*, Peter Lang, Bern, 1994.

[Doets 94] K. Doets, *From Logic to Logic Programming*, MIT Press, Cambridge, Mass., 1994.

[Došen 92] D. Došen, The first axiomatization of relevant logic, *Journal of Philosophical Logic*, vol. 21, p. 339-356, 1992.

[Dowty et al. 81] D. Dowty, R. Wall and S. Peters, *Introduction to Montague Semantics*, Reidel, Dordrecht 1981.

[Enderton 72] H. Enderton, *A Mathematical Introduction to Logic*, Academic Press, New York and London, 1972.

[Engel] P. Engel, Modalities (Logique), *Encyclopedia Universalis*, Paris, p. 509-511.

[Fagin et al. 95] R. Fagin, J. Halpern, Y. Moses, M. Vardi, *Reasoning about Knowledge*, The MIT Press, Cambridge, Mass., 1995.

[Farreny et Ghallab 87] H. Farreny et M. Ghallab, *Eléments d'Intelligence Artificielle*, Hermès, Paris, 1987.

[Feys 50] R. Feys, Les systèmes formalisés des modalités aristotéliciennes, *Revue Philosophique de Louvain*, p. 478-509, 1950.

[Fine and Schurz 96] K. Fine and G. Schurz, Transfer theorems for multimodal logics, in J. Copeland (éditeur), *Logic and Reality*, Clarendon Press, Oxford, 1996.

[Fitch 52] F. Fitch, *Symbolic Logic. An Introduction*, The Ronald Press, New York, 1952.

[Fitting 83] M. Fitting, *Proof Methods for Modal and Intuitionistic Logics*, Reidel, Dordrecht, 1983.

[Fitting 93] M. Fitting, Basic modal logic, in D. Gabbay et al. (éditeurs), *Handbook of Logic in Artificial Intelligence and Logic Programming, vol. 1 : Elements of Classical Logic*, Clarendon Press, Oxford, p. 318-448, 1993.

[Fitting and Mendelsohn 98] M. Fitting and R. Mendelsohn, *First–Order Modal Logic*, Kluwer, Dordrecht, 1998.

[Forbes 94] G. Forbes, *Modern Logic. A Text in Elementary Symbolic Logic*, Oxford University Press, Oxford, 1994.

[Frege 52, 71] G. Frege, Über Sinn und Bedentung, *Zeitschrift für Philosophie und Philosophische Kritik*, p. 25-50, vol. 100, 1893; traduit dans P. Geach and M. Black (éditeurs), *Translation from the Philosophical Writings of Gottlob Frege*, p. 56-78, Basil Blackwell, Oxford, 1952. Traduction française et introduction par C. Imbert, *Ecrits Logiques et Philosophiques*, p. 102-126, Le Seuil, Paris, 1971.

[Gabbay 96] D. Gabbay, *Labelled Deductive Systems*, Clarendon Press, Oxford, 1996.

[Gabbay and Guenthner 83] D. Gabbay and F. Guenthner (éditeurs), *Handbook of Philosophical Logic; vol. 1 : Elements of Classical Logic*, Reidel, Dordrecht, 1983.

[Gabbay and Guenthner 84] D. Gabbay and F. Guenthner (éditeurs), *Handbook of Philosophical Logic; vol. 2 : Extensions of Classical Logic*, Reidel, Dordrecht, 1984.

[Gabbay and Guenthner 86] D. Gabbay and F. Guenthner (éditeurs), *Handbook of Philosophical Logic; vol. 3 : Alternatives to Classical Logic*, Reidel, Dordrecht, 1986.

[Gabbay and Guenthner 89] D. Gabbay and F. Guenthner (éditeurs), *Handbook of Philosophical Logic; vol. 4 : Topics in the Philosophy of Language*, Reidel, Dordrecht, 1989.

[Gabbay et al. 95] D. Gabbay, C. Hogger and J. Robinson (éditeurs), *Handbook of Logic in Artificial Intelligence and Logic Programming; vol. 4 : Epistemic*

and Temporal Reasoning, Oxford Science Publications, 1995.

[Gallier 86] J. Gallier, *Logic for Computer Science*, Harper and Row, New York, 1986.

[Galton 95] A. Galton, *Time and Change for AI*; in D. Gabbay, C. Hogger and J. Robinson (éditeurs), *Handbook of Logic in Artificial Intelligence and Logic Programming; vol. 4 : Epistemic and Temporal Reasoning*, Oxford Science Publications, 1995.

[Gardies 75] J. Gardies, *La Logique du Temps*, Paris Presses Universitaires de France, 1975.

[Gardies 79] J. Gardies, *Essai sur la Logique des Modalités*, Presses Universitaires de France, Paris, 1979.

[Gardner] M. Gardner, *Martin Gardner's column in Scientific American*, voir `http://www.scientificamerican.com/` et `http://www.amazon.com/`

[Garson 84] J. Garson, *Quantification in Modal Logic*; in D. Gabbay and F. Guenthner (éditeurs), *Handbook of Philosophical Logic; vol. 2: Extensions of Classical Logic*, Reidel, Dordrecht, 1984.

[Genesereth and Nilsson 87] M. Genesereth and N. Nilsson, *Logical Foundations of Artificial Intelligence*, Morgan Kaufman, Los Altos, 1987.

[Ginsberg 93] M. Ginsberg, *Essentials of Artificial Intelligence*, Morgan Kaufmann, 1993.

[Gochet 75] P. Gochet, La logique du sens, in *Les Langages, le Sens et l'Histoire*, Presses Universitaires de Lille, p. 225-240, Lille, 1975.

[Gochet 82a] P. Gochet, La Sémantique récursive de Davidson et de Montague, in M. Loi (éditeur), *Penser les Mathématiques*, p. 73-87, Le Seuil, Paris, 1982.

[Gochet 82b] P. Gochet, L'originalité de la sémantique de Montague, *Etudes Philosophiques*, p. 149-175, 1982.

[Gochet 83] P. Gochet, La sémantique des situations, in F. Nef (éditeur), *Histoire, Epistémologie, Langage, tome 5, fascicule 2 : La Sémantique Logique*, pp. 195-212, 1983.

[Gochet 86] P. Gochet, *Ascent to Truth*, Philosophia Verlag, Berlin, 1986.

[Gochet 95] P. Gochet, Problèmes de logique de la connaissance, in J. Dubucs et F. Lepage, *Méthodes pour les Sciences Cognitives*, p. 281-294, Hermès, Paris 1995.

[Gochet 97] P. Gochet, Le syllogisme pratique d'Aristote, in J. Denooz et A. Motte (éditeurs), *Mélanges en l'honneur de C. Rutten*, LASLA, pp. 117-129, Liège.

[Gochet 01] P. Gochet, Logic, existence and ontology, in D. Jacquette (éditeur), *A Companion to Philosophical Logic*, Blackwell, Oxford, à paraître, 2001.

[Gochet et al. 95] P. Gochet, P. Gribomont and D. Rossetto, Algorithms for relevant logic, *Logique et Analyse*, juin-septembre-décembre, p. 150-152, 329-346, 1995.

[Gochet and Gillet 99] P. Gochet and E. Gillet, Quantified modal logic. Dynamic semantics and $S5$, *Dialectica*, p. 243-250, 1999.

[Gochet et Gribomont 91] P. Gochet et P. Gribomont, *Logique, volume 1*, Hermès, Paris, 1991.

[Gochet et Gribomont 94] P. Gochet et P. Gribomont, *Logique, volume 2*, Hermès, Paris, 1994.

[Goldblatt 87] R. Goldblatt, *Logics of Time and Computation*, Lecture Notes no 7, Center for the Study of Language and Information, 1987.

[Goldblatt 92] R. Goldblatt, *Logics of Time and Computation*, Lecture Notes no 7, Center for the Study of Language and Information, revised edition, 1992.

[Goré 92] R. Goré, *Cut-free Sequent and Tableau Systems for Propositional Normal Modal Logics*, Technical Report 257, University of Cambridge, Cambridge, 1992.

[Goré 95] R. Goré, *Tableau Methods for Modal and Temporal Logics*, Technical Report TR-ARP-15-95, R.S.S.S., Australian National University, 1995.

[Gribomont 85] P. Gribomont, *Synthesis of Parallel Programs Invariants*, *Lecture Notes in Computer Science*, vol. 186, p. 325-338, Springer-Verlag, Berlin, 1985.

[Gribomont 00] P. Gribomont, Simplification of Boolean verification conditions, à paraître, *Theoretical Computer Science*, 2000.

[Granger 95] G. Granger, *Le probable, le possible et le virtuel*, Editions Odile Jacob, Paris 1995.

[Groenendijk and M. Stokhof 89] J. Groenendijk and M. Stokhof, Dynamic Predicate Logic; towards a compositional, non-representational semantics of discourse, *First International Summer School on Natural Language Processing, Logic and Knowledge Representation*, Groningen, 1989; aussi dans *Linguistics and Philosophy*, vol. 14, p. 39–100, 1991.

[Groenendijk and Stokhof 90] J. Groenendijk and M. Stokhof, Dynamic Montague grammar, *Proceedings of the Second Symposium on Logic and Language*, Hajduszboszlo, Hongrie, 1990.

[Guenthner and Guenthner-Reutter 78] F. Guenthner and M. Guenthner-Reutter, *Meaning and Translation (Philosophical and Linguistic Approaches)*, New York University Press, New York, 1978.

[Guillaume 58a, b, c] M. Guillaume, Rapports entre calculs propositionnels modaux et topologie impliqués par certaines extensions de la méthode des tableaux, Systèmes de Feys-von Wright, *Comptes rendus de l'Académie des*

Sciences, vol. 246, p. 1140-1142; Système $S4$ de Lewis, *ibidem*, p. 2207-2210; Système $S5$ de Lewis, *ibidem*, vol. 247, p. 1282-1284; Paris, 1958.

[Haack 74, 96] S. Haack, *Deviant Logic*, Cambridge University Press; Cambridge 1974, revised edition, Chicago University Press, Chicago, 1996.

[Halpern 95] J. Halpern, Reasoning about Knowledge : A survey; in D. Gabbay, C. Hogger and J. Robinson (éditeurs), *Handbook of Logic in Artificial Intelligence and Logic Programming; vol. 4 : Epistemic and Temporal Reasoning*, Oxford Science Publications, 1995.

[Halpern et al. 95] J. Halpern, Y. Moses and M. Vardi, *Reasoning About Knowledge*, The MIT Press, Cambridge Mass., 1995.

[Harel 84] D. Harel, Dynamic Logic, in D. Gabbay and F. Guenthner (éditeurs), *Handbook of Philosophical Logic, vol. 2, Extensions of Classical Logic*, vol. 2, p. 497-604, Reidel, Dordrecht, 1984.

[Haton et al. 91] J.-P. Haton et al., *Le Raisonnement en Intelligence Artificielle*, InterEditions, Paris, 1991.

[Hendriks 89] H. Hendriks, Flexible Montague grammar, *First European Summer School on Natural Language Processing, Logic and Knowledge Representation*, Groningen, 1989.

[Hilpinen 81] R. Hilpinen (éditeur), *New Studies in Deontic Logic, Norms, Actions and the Foundations of Ethics*, Reidel, Dordrecht, 1981.

[Hintikka 57] J. Hintikka, Quantifiers in deontic logic, *Societas Scientiarum Fennica Commentationes Humanorum Litterarum*, vol. 23, 1957.

[Hintikka 62] J. Hintikka, *Knowledge and Belief*, Cornell Univ. Press, Ithaca 1962.

[Hintikka 69] J. Hintikka, *Models for Modalities*, Reidel, Dordrecht, 1969.

[Hintikka 70] J. Hintikka, Existential and uniqueness presuppositions, in Lambert, (éditeur), *Philosophical Problems in Logic*, Reidel, Amsterdam, 1970.

[Hintikka 73] J. Hintikka, Grammar and logic, some borderline problems, in J. Hintikka, J. Moravcsik and P. Suppes (éditeurs), *Approaches to Natural Languages*, Reidel, Dordrecht, p. 197-214, 1973.

[Hintikka 81] J. Hintikka, Standard vs. Nonstandard logics : higher-order, modal and first-order logics, in E. Agazzi, *Modern Logics; a Survey*, p. 283-296, Reidel, Dordrecht, 1981.

[Hintikka 89] J. Hintikka, *L'Intentionnalité et les Mondes Possibles*, traduction et présentation de N. Lavand, Presses universitaires de Lille, 1989.

[Hintikka and Sandu 95] J. Hintikka and G. Sandu, The fallacies of the new theory of reference, *Synthese*, p. 245-283, 1995.

[Hopcroft and Ullman 79] J. Hopcroft and J. Ullman, *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, Reading, Mass., 1979.

[Hughes and Cresswell 84] G. Hughes and M. Cresswell, *A Companion to Modal Logic*, Methuen, London, 1984.

[Hughes and Cresswell 96] G. Hughes and M. Cresswell, *A New Introduction to Modal Logic*, Methuen, London, 1996.

[Husserl 59] E. Husserl, *Recherches Logiques; Tome Premier*, traduction H. Elie, Presses Universitaires de France, Paris, 1959.

[Indrzejczak 99] A. Indrzejczak, A survey of natural deduction systems for modal logics, *Logica Trianguli*, vol. 3, p. 55-83, 1999.

[Iseminger] G. Iseminger (éditeur), *Logic and Philosophy*, Appelton-Century-Crofts, New York.

[Jackson et al. 89] P. Jackson, H Reichgelt and F. van Harmelen, *Logic–Based Knowledge Representation*, MIT Press Series in Logic Programming, Cambridge, Mass., 1989.

[Jansana 90] R. Jansana, *Una introducción a la lógica modal*, tecnos, Madrid, 1990.

[Jansana 95] R. Jansana, *Introduction to Modal Logic*, ESSLLI 7th., Barcelona, 1995.

[Janssen 86] T. Janssen, *Foundations and Applications of Montague Grammar, Part I : Philosophy, Framework, Computer Science*, CWI Tract 19, Amsterdam, Center for Mathematics and Computer Science, 1986.

[Jayez 88] J. Jayez, *L'Inférence en Langue Naturelle*, Hermès, Paris, 1988.

[Kalinowski 96] G. Kalinowski, *La Logique Déductive, Essai de Présentation aux Juristes*, Presses Universitaires de France, Paris, 1996.

[Kamp 84] J. Kamp, A Theory of truth and semantic representation, in Groenendijk et al., (éditeurs), *Truth, Interpretation and Information*, p. 1-41, Foris Publications, Dordrecht, 1984.

[Kanger 57] S. Kanger, *Provability in Logic*, Almqvist and Wiksell, Stockholm, 1957.

[Keenan and Faltz 85] E. Keenan and L. Faltz, *Boolean Semantics for Natural Language*, Reidel, Dordrecht, 1985.

[Kleene 71] S. Kleene, *Logique Mathématique*, (trad. de *Mathematical Logic* 1967), Armand Colin, Paris, 1971.

[Kneale and Kneale 62] W. Kneale and M. Kneale, *The Development of Logic*, Clarendon Press, Oxford 1962.

[Knuuttila 81] S. Knuuttila, The Emergence of Deontic Logic in the Fourteenth Century, in R. Hilpinen (éditeur), *New Studies in Deontic Logic, Norms, Actions and the Foundations of Ethics*, p. 225-248, Reidel, Dordrecht, 1981.

[Konolige 86] K. Konolige, *A Deduction Model of Belief*, Research Notes in Artificial Intelligence, Pitman, London, 1986.

[Kripke 59] S. Kripke, A completeness theorem in modal logic, *The Journal of Symbolic Logic*, vol. 24, no 1, p. 1-14, 1959.

[Kripke 63] S. Kripke, Semantic analysis of modal logic : normal propositional calculi, *Zeit. Math. Logik Grund. Math.*, vol. 9, p. 67-96, 1963.

[Kripke 71] S. Kripke, Identity and necessity, in M. K. Munitz (éditeur), *Identity and Individuation*, New York University Press, New York, p. 135-164, 1971.

[Kripke 72] S. Kripke, *La Logique des Noms Propres*; trad. franç. Paris, Editions de Minuit, 1982.

[Kuiper] R. Kuiper and W. Penczek, Verification by hand using linear time temporal logic, in H.C.M. De Swart (éditeur), *Mathematics, Language, Computer Science and Philosophy, vol. 2, Logic and Computer Science*, p. 229-251,

[Lambert 70] K. Lambert, (éditeur), *Philosophical Problems in Logic*, Reidel, Amsterdam, 1970.

[Laurière 87] J. Laurière, *Intelligence Artificielle, Résolution de Problèmes par l'Homme et la Machine*, Eyrolles, Paris, 1987.

[Lavand 89] N. Lavand, Introduction, voir J. Hintikka, *L'Intentionnalité et les Mondes Possibles*, p. 10-34, 1989.

[Leibniz 03] G. Leibniz, Vérités nécessaires et contingentes, in L. Couturat (éditeur), *Opuscules et Fragments Inédits de Leibniz*, p. 16-24, Alcan, Paris, 1903.

[Lemmon 57] E. Lemmon, Is there only one correct system of modal logic, *The Aristotelian Society*, Supplementary volume, p. 23-40, Harrison, London, 1957.

[Lewis 12] C. Lewis, Implication and the algebra of logic, *Mind*, vol. 21, 1912.

[Lewis 18] C. Lewis, *A Survey of Symbolic Logic*, University of California Press, Berkeley, 1918.

[Lewis 73] C. Lewis, *Counterfactuals*, Harvard University Press, Cambridge, 1973.

[Lewis and Langford 32, 59] C. Lewis and C. Langford, *Symbolic Logic*, 1932; Dover Publications (2ième édition), 1959.

[Lewis 18] D. Lewis, *A Survey of Symbolic Logic*, University of California, Berkeley, 1918.

[Linski 72] L. Linski (éditeur), *Reference and Modality*, p. 63-72, Oxford University Press, 1972.

[Lloyd 87] J.W. Lloyd, *Foundations of Logic Programming*, Springer-Verlag (2ième édition), Berlin, 1987.

[Lucas 85] T. Lucas, La logique modale, *Revue Philosophique de Louvain*, no 60, p. 595-598, 1985.

[Luger and Stubblefield 98] G. Luger and W. Stubblefield, *Artificial Intelligence*, Addison-Wesley, (3ième édition), Reading, Mass., 1998.

[Łukasiewicz 30] W. Łukasiewicz, Many-valued systems of propositional logic, in S. McCall (éditeur), *Polish Logic*, Oxford University Press, Oxford, 1967.

[Łukasiewicz 53] W. Łukasiewicz, A system of modal logic, *Journal Computing Systems*, vol. 1, p. 111-149, 1953.

[Łukasiewicz 70] W. Łukasiewicz, *Selected Works*, L. Borkowski (éditeur), North-Holland, Amsterdam, 1970.

[McCarthy 80] J. McCarthy, Circumscription : a form of non-monotonic reasoning, *Artificial Intelligence*, vol. 13, no 1-2, p. 27-39, 1980.

[McCarthy 86] J. McCarthy, Applications of circumscription to formalizing common-sense knowledge, *Artificial Intelligence*, vol. 28, p. 89-116, 1986.

[Mendelson 64] E. Mendelson, *Introduction to Mathemetical Logic*, Van Norstrand, Princeton, 1964.

[Meyer and Wieringa 93] J. Meyer and R. Wieringa, *Deontic Logic in Computer Science*, John Wiley, Chichester, 1993.

[Meyer and van der Hoek 95] J. Meyer and W. van der Hoek, *Epistemic Logic for A.I. and Computer Science*, Cambridge University press, Cambridge, 1995.

[Moisil 72] G. Moisil, *Essai sur les Logiques non Chrysippiennes*, Editions de l'Académie de la République Socialiste de Roumanie, Bucarest, 1972.

[Montague 73] R. Montague, The proper treatment of quantification in ordinary English, in Hintikka et al. (éditeurs), *Approaches to Natural Language*, p. 221-242, Reidel, Dordrecht, 1973. Egalement traduction de F. Guenthner, in F. Nef (éditeur), *L'analyse logique des langues naturelles (1968–1978)*, éditions du C.N.R.S., p. 135-159, 1984.

[Montague 74a] R. Montague, On the nature of certain philosophical entities, *The Monist*, vol. 53, p. 159-194, 1960; repris dans R. Thomason (éditeur), *Formal Philosophy : Selected Papers of Richard Montague*, pp. 148–187, Yale University Press, 1974.

[Montague 74b] R. Montague, Pragmatics, in R. Klibansky (éditeur), *Contemporary Philosophy : A Survey*, p. 102-122, La Nuova Italia Editrice, 1968; repris dans : R. Thomason (éditeur), *Formal Philosophy : Selected Papers of Richard Montague*, p. 95-118, Yale University Press, 1974.

[Montague 74c] R. Montague, English as a formal language, B. Visentini et al. (éditeurs), *Linguaggi nella e nella Tecnica*, p. 189-224, Edizioni di Commutà, 1970; repris dans : R. Thomason (éditeur), *Formal Philosophy : Selected Papers of Richard Montague*, p. 108-221, Yale University Press, 1974.

[Montague 74d] R. Montague, Universal grammars, Theoria, vol. 36, p. 373-398, repris dans : R. Thomason (éditeur), *Formal Philosophy : Selected Papers of Richard Montague*, p. 222-246, Yale University Press, 1974.

[Muskens 88] R. Muskens, Going partial into Montague grammar, ITLI Prepublication series, 1988; également dans R. Bartsch et al. (éditeurs), *Proceedings of the Sixth Amsterdam Colloquium*, Foris Publications, Dordrecht, 1989.

[Mutombo 95] M. Mutombo, Remarques sur l'argument du slingshot, *Revue Philosophique de Kinshasa*, vol. 15-16, p. 27-43, 1995.

[Nef 84] F. Nef (éditeur), *L'analyse Logique des Langues Naturelles*, Editions du Centre National de la Recherche Scientifique, Paris 1984.

[Nef 91] F. Nef, *Logique, Langage et Réalité*, Editions Universitaires, Paris, 1991.

[Nef 98] F. Nef, *L'objet Quelconque. Recherches sur l'Ontologie de l'Objet*, Vrin, Paris, 1998.

[Nerode and Shore 93] A. Nerode and R. Shore, *Logic for Applications*, Springer-Verlag, Berlin, 1993.

[Ntambue 98] R. Ntambue, *Eléments de Logique Trivalente. Du Calcul Logique Trivalent à sa Modalité Ontique*, Bruylant-Academia, Louvain-la-Neuve, 1998.

[Øhrstrøm and Hasle 95] P. Øhrstrøm and P. Hasle, *Temporal Logic. From Ancient Ideas to Artificial Intelligence*, Kluwer Academic Press, 1995.

[Parikh 78] R. Parikh, The completeness of propositional dynamic logic, in *Lecture Notes in Computer Science*, vol. 64, p. 403-415, Springer-Verlag, Berlin, 1978.

[Pereira and Warren 80] F. Pereira and D. Warren, Definite clause grammar for language analysis - A survey of the formalism and a comparison with ATN, *Artificial Intelligence,* vol. 13, p. 231-278, 1980.

[Popkorn 94] S. Popkorn, *First Steps in Modal Logic*, Cambridge University Press, Cambridge, 1994.

[Priest and Sylvan 92] G. Priest and R. Sylvan, Simplified semantics for basic relevant logic, *Journal of Philosophical Logic*, vol. 21, p. 217-232, 1992.

[Prior 57] A. Prior, *Time and Modality*, Oxford University Press, Oxford 1957.

[Prior 62] A. Prior, *Formal Logic*, Clarendon Press, 1962.

[Prior 67] A. Prior, *Past, Present and Future*, Clarendon Press, Oxford 1967.

[Quine 40, 81] W. Quine, *Mathematical Logic*, Revised Edition, Harvard University Press, Harvard 1981.

[Quine 72a] W. Quine, *Logique élémentaire*, traduction de J. Largeault et B. Saint-Sernin, Colin, Paris, 1972.

[Quine 72b] W. Quine, Methodological reflections on current linguistic theory, in D. Davidson and G. Harman (éditeurs), *Semantics of Natural Language*, p. 442, 454, Reidel, Dordrecht, 1972.

[Quine 73] W. Quine, *Méthodes de Logique*, traduction M. Clavelin, 3ième édition, Armand Colin, Paris, 1973.

[Quine 76, 81] W. Quine, Worlds Away, in *Journal of Philosophy*, 1976; repr. in *Theories of Things*, The Belknap Press, Harvard, p. 124-128, 1981.

[Quine 93] W. Quine, *La Poursuite de la Vérité*, traduction de M. Clavelin, Editions du Seuil, Paris, 1993.

[Ramsay 88] A. Ramsay, *Formal Methods in Artificial Intelligence*, Cambridge University Press, 1988.

[Rautenberg 79] W. Rautenberg, *Klassische und Nichtklassische Aussagenlogic*, Vieweg, Braunschweig, 1979.

[Rautenberg 83] W. Rautenberg, Modal tableau calculi, *Journal of Philosophical Logic*, p. 403-423, 1983.

[Read 93] S. Read, The slingshot argument, in *Logique et Analyse*, p. 195-218, 1993.

[Rescher 69] N. Rescher, *Many-Valued Logic*, McGraw Hill, New York, 1969.

[Rescher and Urquhart 71] N. Rescher and A. Urquhart, *Temporal Logic*, Springer-Verlag, Berlin 1971.

[Reyes 94] M. Reyes, Referential structure of fictional texts, in J. MacNamara and G. Reys (éditeurs), *The Logical Foundations of Cognition*, p. 309-324, Oxford University Press, Oxford, 1994.

[Riche 91] J. Riche, *Decidability, Complexity and Automated Reasoning in Relevant Logic*, Dissertation doctorale, Australian National University, Dept. of Philosophy, ARP, RSSS 1991.

[Rivenc et de Rouilhan 97] F. Rivenc et de Rouilhan, Traduction et introduction à R. Carnap; voir R. Carnap, *Signification et nécessité*, 1997.

[Sahlqvist 75] H. Sahlqvist, Completeness and correspondence in the first and second order semantics for modal logic, in S. Kanger (éditeur), *Proceedings of the Third Scandinavian Logic Symposium*, pp. 110-143, North-Holland Publishing Company, Amsterdam, 1975.

[Shoam 95] Y. Shoam, *Artificial Intelligence Techniques in Prolog*, 1995.

[Sifakis 82] J. Sifakis, A unified approach for studying the properties of transition systems, *Theoret. Comput. Sci.*, vol. 18, p. 227-259, 1982.

[Smullyan 68] R. Smullyan, *First–Order Logic*, Springer-Verlag, Berlin, 1968.

[Stalnaker 68] R. Stalnaker, A theory of conditionals, in N. Rescher (éditeur), *Studies in Logical Theory, American Philosophical Quarterly*, Blackwell, Oxford, 1968.

[Stenning 76] N. Stenning, A data transfer protocol, *Computer Networks*, vol. 1, p. 99-110, 1976.

[Sterling et Shapiro 94] L. Sterling et E. Shapiro *The Art of Prolog*, MIT Press, Cambridge, Mass., 1994.

[Tarski 44] A. Tarski, The semantic conception of truth, *Philosophy and Phenomenological Research*, vol. 4, p. 341-375, 1944; repris dans : L. Linsky (éditeur), *Semantics and the Philosophy of Language*, p. 13-47, University of Illinois Press, Urbana, 1972. Repris aussi dans A. Tarski, *Logique, Sémantique, Métamathématique, 1923–1944, tome 2*, traduit sous la direction de G. Granger, Armand Colin, p. 265-305, Paris, 1974.

[Thomason 70] R. Thomason, Some completeness results for modal predicate calculi, dans Lambert, 1970.

[Thayse et al. 88] A. Thayse, *Approche Logique de l'Intelligence Artificielle, tome 1 : De la Logique Classique à la Programmation Logique*, Dunod, Paris, 1988.

[Thayse et al. 89] A. Thayse, *Approche Logique de l'Intelligence Artificielle, tome 2 : De la Logique Modale à la Logique des Bases de Données*, Dunod, Paris, 1989.

[Thayse et al. 90] A. Thayse, *Approche Logique de l'Intelligence Artificielle, tome 3 : Du Traitement de la Langue à la Logique des Systèmes Experts*, Dunod, Paris, 1990.

[Thayse et al. 91] A. Thayse, *Approche Logique de l'Intelligence Artificielle, tome 4 : De l'Apprentissage Artificiel aux Frontières de l'IA*, Dunod, Paris, 1991.

[Turner 84] R. Turner, *Logics for Artificial Intelligence*, Ellis Horwood Limited, Chichester 1984. Traduit par P. Besnard, *Logiques pour l'Intelligence Artificielle*, Masson, Paris.

[Ullman 86] J. Ullman, Implementation of logical query languages for databases, *ACM Transactions on Database Systems*, vol. 10, no 3, p. 289–321, 1986.

[Ullman 88] J. Ullman, *Principles of Databases and Knowledge-Base Systems*, Computer Science Press, 1988.

[van Benthem 76] J. van Benthem, *Modal correspondence Theory*, Doctoral Dissertation, Mathematical Institute, University of Amsterdam, 1976.

[van Benthem 91] J. van Benthem, *Language in Action, Categories, Lambdas and Dynamic Logic*, North-Holland, Amsterdam, 1991.

[van Benthem 95] J. van Benthem, Temporal logic; in D. Gabbay, C. Hogger and J. Robinson (éditeurs), *Handbook of Logic in Artificial Intelligence and Logic Programming; vol. 4 : Epistemic and Temporal Reasoning*, Oxford Science Publications, 1995.

[van Benthem 96] J. van Benthem, *Exploring Logical Dynamics*, CSLI Publications and FoLLI, Stanford, 1996.

[van Heijenoort 67] J. van Heijenoort, *From Frege to Gödel. A Source Book in Mathematical Logic*, p. 1879-1931, Harvard University Press, Cambridge Mass., 1967.

[von Wright 51, 57] von Wright, Deontic Logic, *Mind* 1951, repris dans *Logical Studies*, p. 58, 74, Routledge and Kegan Paul, Londres 1957.

[Vuillemin 84] J. Vuillemin, *Nécessité ou contingence. L'Aporie de Diodore et les Systèmes Philosophiques*, Les éditions de Minuit, Paris 1984.

[Winston 92] P. Winston, *Artificial Intelligence*, Addison-Wesley, Reading (Mass.), 1992 (3ième édition).

[Wolper 83] P. Wolper, Temporal logic can be more expressive, *Information and Control* vol. 56, p. 72-99, 1983.

[Wolper 85] P. Wolper, The tableau method for temporal logic : an overview, *Logique et Analyse*, p. 119-136, 1985.