Université de Carthage
ISSAT MATEUR

Support de Cours

Intelligence Artificielle

« Niveau de base »

Jamel SLIMI

E-mail : jamel.slimi@gmail.com

Année universitaire 2017 – 2018

Présentation

Objectif :
L'objectif de ce cours est de permettre aux étudiants de se familiariser avec les techniques classiques de l'intelligence artificielle afin de les mettre en œuvre dans des applications concrètes.
Ce support de cours a pour buts d'introduire les notions de base de l'intelligence artificielle sur le plan de la représentation des connaissances et sur le plan du traitement ; de connaître les problèmes qui relèvent de l'intelligence artificielle et les techniques de résolution ; et de connaître le langage PROLOG.

Plan du cours :
- Introduction générale ;
- Les bases logiques ;
- Les systèmes experts ;
- Résolution de problèmes en intelligence artificielle ;
- Annexe pour travaux pratiques : Programmation logique (Prolog).

Bibliographie :

- Jean-Gabriel Ganascia. *L'intelligence artificielle*. Collection Idees Recues, numéro 138, Cavalier Bleu Eds, 2007.
- Stuart Russel, Peter Norvig. *Intelligence artificielle (ou Artificial Intelligence : A Modern Approach, en anglais)*. Pearson Education France, 2ème édition, 2006.
- Hugues Bersini. *De l'intelligence humaine à l'intelligence artificielle*. Ellipses Editions, 1ère édition, 2006.
- Jean-Marc Alliot, Thomas Schiex, Pascal Brisset, Frédérick Garcia. *Intelligence Artificielle et Informatique Théorique*. Cépadues Editions, 2ème édition, 2002.
- Ahmed Hadj Kacem. *Introduction à l'Intelligence Artificielle*. Support de cours, FSEGS, université de Sfax.
- Nicolas Clerbout, *Logique propositionnelle*. Support de cours, université Lille 3.
- Adel Mahfoudhi, *Logique mathématique*. Support de cours, FSS, université de Sfax.
- http://encyclopedie.snyke.com

Plan de cours

Chapitre 1 :

Introduction générale à l'intelligence artificielle

1.1 Quelques définitions

1.1.1 Qu'est ce que Informatique ?

C'est la science du traitement de l'information.

1.1.2 Qu'est ce qu'un Algorithme ?

C'est une suite d'opérations ordonnées, bien définies, exécutables sur un ordinateur actuel, et qui permet d'arriver à la solution en un temps raisonnable (minutes, heures, ou plus, ... mais pas des siècles!).

1.1.3 Qu'est ce qu'un Ordinateur ?

C'est une machine capable de

1. Réaliser une très grande quantité d'opérations arithmétiques sans erreur ;
2. Stocker beaucoup d'information: données, documents, images, sons, etc. ;
3. Simuler des phénomènes très complexes.

1.1.4 Qu'est ce que la Connaissance ?

C'est la compétence qui permet de résoudre des problèmes.

1.1.5 Qu'est ce que le Raisonnement ?

C'est la génération de nouvelles connaissances.

1.1.6 Qu'est ce que l'Artificiel ?

C'est le résultat d'un processus humain plutôt que d'un processus naturel (biologique et évolutionnaire).

1.1.7 Qu'est ce que l'Intelligence ?

a) Qu'est-ce que les tâches suivantes ont en commun ?

- Concevoir un programme capable d'identifier et supprimer le SPAM ;
- Concevoir un programme effectuant de la veille scientifique sur le Web ;

- Concevoir un programme s'adaptant aux connaissances, à la fatigue, à l'émotion ... de son utilisateur ;
- Concevoir un programme résolvant des problèmes de géométrie ;
- Concevoir un robot autonome capable d'évoluer de façon autonome pour sauver des vies en cas de catastrophe . . .

Un tel système doit être intelligent !

b) Définition

Qu'est-ce que signifie être intelligent pour vous? selon New World Dictionary (1988)

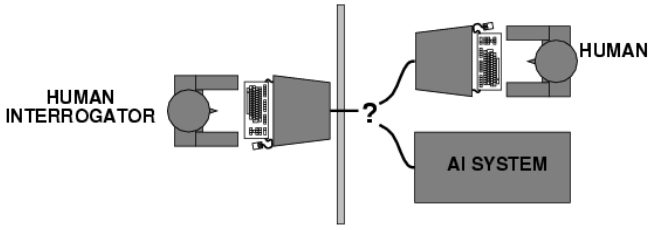
1. La capacité d'apprendre ou de comprendre grâce à l'expérience. La capacité d'acquérir et de retenir les connaissances. La capacité mentale ;
2. La capacité de répondre rapidement et de manière appropriée à une nouvelle situation; L'utilisation de la faculté de raisonnement pour résoudre des problèmes, se comporter en société, etc. de manière effective ;
3. En Psychologie, le succès mesuré de l'utilisation de ces capacités afin de résoudre certaines tâches.

Il y a d'autres réponses générales suivant les individus :

- *Turing* : C'est ce qui rend difficile la distinction entre une tâche réalisée par un être humain ou par une machine.
- *Darwin* : C'est ce qui permet la survie de l'individu le plus apte.
- *Edison* : C'est ce qui fait que cela fonctionne.
- *Lorenz* : C'est collectif et cela émerge du comportement collectif.

c) Comment mesurer l'intelligence ?

- Le test de Turing : si la conversation d'une machine peut pas être différencie de la conversation humaine on peut dire que la machine possède l'intelligence. La durée du test est environ 5min de conversation avec une personne et une machine (ordinateur). Ensuite, il faut deviner qui est la personne et qui est la machine.



- Stratégies d'évaluations pratiques:
    - Échecs ➔ Tournois
    - Quand c'est possible, utiliser les critères d'ingénierie pour analyser la performance : Est-ce que le programme a réussi à accomplir la tâche qui lui a été demandée ? L'a-t-il fait efficacement ? L'a-t-il bien fait ?

d) Exemple de test d'intelligence

C'est un test de quotient intellectuel (QI) pour mesurer l'intelligence des gens. Exemple :

1. Le jour du 14 juillet existe-t-il en Angleterre? Oui/non
2. Combien d'anniversaires a eu une personne qui a vécu 50 ans ? 49/50/51
3. Certains mois ont 31 jours, combien ont un 28ème jour ? 1/2/3/6/9/12
4. Combien de neuf (9) y a-t-il entre 0 et 100 ? 9/10/11/19/20
5. Est-il correct qu'un homme se marie avec la sœur de sa veuve ? oui/non
6. Divise 30 par 1/2 et ajoute 10. Quel est le résultat ? 10/35/50/70/90
7. S'il a 3 pommes et que tu en prends 2, combien en as-tu? 1/2/3
8. Un médecin te donne trois comprimés et te dit d'en prendre un chaque demi-heure. Combien de minutes te durent les comprimés? 20/40/60/90
9. Un fermier a 17 moutons. Tous meurent, sauf 9. Combien lui en reste-t-il ? 1/3/5/8/9
10. Combien de timbres de 2 centimes y-a-t-il dans une douzaine? 1/3/6/9/12

Réponses :

1. Oui, Juste après le 13
2. 50
3. 12, tous ont un 28ème jour
4. 20 ! (9,19,29,..,89,90,91,..99)
5. Non, car il est mort !
6. 70
7. 2
8. 60 - Tu commences en prenant le premier, 30 minutes ensuite tu prends le second, puis 30 minutes plus tard tu prends le dernier...
9. 9 évidemment !
10. 12, il y a 12 timbres dans une douzaine !

1.2 Qu'est ce que l'intelligence artificielle ?

L'Intelligence Artificielle (Artificial Intelligence, en anglais) est une expression forgée par l'informaticien John McCarthy en 1956 au moment où l'informatique en était à ses tout premiers pas. Pour plus de commodité, on l'appelle souvent « IA ». Marvin Lee Minsky, américain co-fondateur avec McCarthy du 1er centre de recherche sur l'IA, la définit à cette époque comme « *la construction de programmes informatiques qui s'adonnent à des tâches qui sont, pour l'instant, accomplies de façon plus satisfaisante par des êtres humains car elles demandent des processus mentaux de haut niveau tels que : l'apprentissage perceptuel, l'organisation de la mémoire et le raisonnement critique* ». Cette définition est débordante d'imagination car elle date d'une époque où l'ordinateur - au sens d'aujourd'hui – n'en était qu'à ses tout débuts, capable seulement d'effectuer les 4 opérations.

Aujourd'hui, soit un demi-siècle plus tard, en dépit des progrès extraordinaires des ordinateurs et de l'informatique, ce projet n'a officiellement pas abouti car les chercheurs eux-mêmes le pensent. Pourtant, l'IA a bien produit ce qu'elle avait projeté. L'organisation de la mémoire des ordinateurs n'est plus un problème depuis longtemps. L'automatisation du « raisonnement critique » est devenue un fait grâce à l'invention – française - du *système expert d'ordre 0* dès les années 1980. L' « apprentissage conceptuel » est entré dans la réalité grâce à l'invention du *réseau de neurones*.

Dans l'ignorance de ces résultats, les chercheurs en IA ne s'accordent même plus sur ce qu'elle est réellement. Ils contribuent même à brouiller l'image de l'Intelligence Artificielle en lui donnant de nouvelles définitions où l'intelligence n'a même plus sa place. Alliot et ses collègues définissent l'IA comme suit : « *l'intelligence artificielle a pour but de faire exécuter par l'ordinateur des tâches pour lesquelles l'homme, dans un contexte donnée, est aujourd'hui meilleur que la machine* ». C'est-à-dire que tout ce qui n'a pas encore été fait en informatique - quand on sait le faire, ce n'est plus de l'IA. C'est une définition évolutive c'est-à-dire que l'***IA d'aujourd'hui n'est ni l'IA d'hier ni celle de demain*.** D'autre part, la réunion des deux mots 'intelligence' et 'artificiel' donne : *l'Intelligence Artificielle est une machine donnant l'impression de penser comme l'homme*. D'où ***l'IA est l'automatisation du raisonnement humain***.

Dans l'atmosphère générale de doute sur la faisabilité de l'IA, certains spécialistes de l'informatique se sont sentis obligés de définir deux types d'IA : une « IA forte », celle qui est définie ci-dessus et qui, dans leur esprit, n'est pas près de voir le jour et une « IA faible », à la portée de tout développeur intelligent.

Cependant, il est extrêmement difficile, aujourd'hui, de donner à l'intelligence artificielle une définition **précise**, tellement les domaines qui recouvrent le terme sont vastes et les avis divergents. Les huit définitions de l'IA les plus connus dans la littérature s'ordonnent selon quatre catégories :

- ***Des systèmes qui pensent rationnellement*** (logicisme, 19e s.) : Ils utilisent les lois de la pensée formulées avec la notation logique ;
- ***Des systèmes qui agissent rationnellement*** (approche de l'agent rationnel, 1990) : un agent rationnel est un agent qui agit (selon ses croyances) de manière à atteindre son objectif avec la meilleure solution ou, dans un environnement incertain, avec la meilleure solution prévisible.
- ***Des systèmes qui agissent comme les humains*** (systèmes passant le test de Turing, 1950) : Ils devraient posséder le traitement de langage naturel, la représentation des connaissances, le raisonnement automatisé, et l'apprentissage. Afin de réussir le test de Turing complet, l'ordinateur doit être doté d'un dispositif de vision informatique pour percevoir des objets, et de capacités robotiques pour manipuler des objets et se déplacer ;
- ***Des systèmes qui pensent comme les humains*** (approche cognitive, 1961) : Il faut pouvoir déterminer comment pensent les êtres humains en pénétrant à l'intérieur des rouages de l'esprit. Deux moyens existent : l'introspection – la tentative de se saisir de ses propres pensées – et les expériences psychologiques;

Concrètement, l'IA se résume dans les 3 phases suivantes :

- ***Rechercher*** (analyser, résoudre des problèmes, trouver des méthodes de résolution)
- ***Représenter des connaissances*** (logique, règles, mémoire, cas, langue naturelle, etc.)
- ***Mettre en application*** les idées 1) et 2) (Systèmes Experts, pilotes automatiques, agents d'interfaces, robots, Data Mining, etc.)

1.3 Historique

L'histoire de l'IA se caractérise par des phases de succès et d'optimisme démesuré, d'une part, et des périodes de pessimisme et de restrictions budgétaires, d'autre part.

Avant 1956 : Gestation (Grossesse) de l'IA

XVIe siècle : A partir des progrès de la médecine (lois de fonctionnement de certains organes), et du perfectionnement des automates, on pense pouvoir créer des mécanismes "intelligents".

XVIIe : Descartes introduit l'idée de l'"animal machine", qui aurait certaines activités humaines, mais pas toute l'intelligence.

1747 : La Mettrie publie "L'homme machine". Il a l'intuition que la distinction homme animal de Descartes ne tient pas, mais ne sait pas comment expliquer le comportement de l'homme.

Fin du XVIIIe : Un anonyme publie une description d'une méthode automatique pour composer des menuets. Il a défini un ensemble de règles. Il n'avait bien sûr pas d'ordinateur, mais a fait des simulations, avec des lancements de dés pour introduire de la variété.

1930 : Gödel, Church, Herbrand, Türing étudient la possibilité d'automatiser le calcul et le raisonnement.

1943 : McCulloch & Pitts : créent le modèle du neurone formel

1943 : Apparition des premiers ordinateurs

1945 : Zuse, un des pères des premiers ordinateurs, programme les règles du jeu d'échecs.

1948 : Création de la cybernétique (science des systèmes) par Norbert Wiener

1950 : Shannon, 1952 Samuel, 1953 Turing : machine pour jouer aux échecs

1956-1966 : Naissance de l'IA : Les années lumières (grands espoirs)

1956 : Réunion de Dartmouth (USA) : le terme "Intelligence Artificielle" est adopté.

Les grands inspirateurs : Minsky (tenant d'une approche par schémas), McCarthy (tenant de la logique), Newell, Simon, Shannon (théorie de l'information), Wiener (cybernétique), Mc Culloch et Pitts (réseaux neuronaux artificiels), Von Neumann (architecture d'un calculateur), Turing (théorisation des fonctions calculables par machine).

- A Dartmouth:
  - Newell et Simon présente le GPS (General Problem Solver) qui démontre la résolution des problèmes par des théorèmes logiques
  - Samuel présente un logiciel capable d'apprendre à jouer aux échecs
  - Grand enthousiasme pour l'automatisation des jeux
  - Mc Carthy présente le langage LISP (langage fonctionnel) pour la manipulation symbolique et propose l'expression: Intelligence Artificielle
  - Minsky critique très virulemment l'approche neuronale dans un livre intitulé le « perceptron »
  - La psychologie redécouvre le « cognitivisme » en alternative au « behaviorisme » avec des auteurs comme William James, Bartlett et Jean Piaget: → Poussée très forte du « symbolisme »

1958 : Newel et Simon pensent qu'avant 1968, un programme sera champion d'échecs et démontrera un important théorème mathématique!

1966 : L'IA découvre la complexité des calculs

1966-1969 : Les années noires

- le remplacement des experts humains par des systèmes experts : échec, raisonnement ok, manque de connaissance, goulot d'étranglement de l'IA
- compréhension du langage naturel : échec, trop difficile, impossible ?

1969-1979 : Les systèmes fondés sur les connaissances

1969 : Premiers développements de systèmes à base de connaissances

1971 : Naissance du langage Prolog

1980-1990 : Une reconnaissance

1980 : Naissance des systèmes experts

1980 : Naissance des micro-ordinateurs

1980 : Le Japon lance le projet cinquième génération des ordinateurs

1985 : Retour des réseaux de neurones

1991-aujourd'hui : Nouvelle IA

Fouille de données, intégration et traitement de données, interaction homme-machine, IA Distribués, systèmes multi-agents, robotique, apprentissage par renforcement, algorithmes génétiques, intelligence collective, émergente, etc.

1.4 Domaines d'application de l'IA

L'IA est cachée presque partout dans plusieurs domaines d'applications, mais souvent trop bien cachée. Parmi ces domaines, on note :

- Reconnaissances et système de parole. Ex : réservation d'hôtel, annuaire téléphonique
- Reconnaissance et système d'image. Ex : effets spéciaux au cinéma, vidéo-surveillance
- Reconnaissance de l'écriture. Ex: reconnaissances chèques, codes postaux
- Reconnaissance des visages. Ex: accès à un bâtiment
- Traitement du langage naturel. Ex: interfaces, text mining, web mining
- Planification
- Aide de la décision. Ex : contrôle de trajectoire du satellite, voyage
- Aide à la programmation. Ex : agent d'interface
- Robotique
- apprentissage/adaptation. Ex : construction de systèmes experts, classification automatique de galaxie, contrôleurs de robots

- Jeux. Ex : Échecs, Checkers, Othello(champion), Backgammon,
- Médecine. Ex : aide à la décision (système expert), prédiction de patients à risques, analyse automatique d'images médicales
- Surveillance du trafic routier

1.5 Conclusion

Nous concluons que l'IA peut être envisagée de différentes manières. Les deux questions essentielles qu'il convient de se poser sont celles-ci : vous intéressez-vous plutôt à la pensée ou au comportement ? Voulez-vous prendre modèle sur les humains ou travailler à partir d'une norme idéale ?

Les avancées de l'IA se sont accélérées au cours de la dernière décennie du fait d'un plus grand usage de la méthode scientifique dans l'expérimentation et de la comparaison des approches.

Lors du 50ème anniversaire de l'IA, Ganascia présente la conclusion suivante : « *L'IA n'est plus seulement une IA, L'IA est une AI. Mais, l'IA est aussi une IA. Et, plus que jamais, l'IA participe d'un AI* » c'est-à-dire, l'Intelligence Artificielle n'est plus seulement une Informatique Autonome, L'Intelligence Artificielle est une Augmentation de l'Intelligence. Mais, l'Intelligence Artificielle est aussi une Informatique Affective. Et, plus que jamais, L'Intelligence Artificielle participe d'un Animisme Informatique (signification de l'Animisme Informatique : L'IA étudie la façon dont nous «leur» attribuons un esprit (pensée) et une intelligence).

Chapitre 2 :

Les bases logiques de l'intelligence artificielle

2.1 Introduction à la logique formelle

2.1.1 Qu'est ce que la logique ?

La logique est *la science du raisonnement* et de son expression. Tout être humain raisonne est donc concerné par la logique. Raisonner c'est produire de l'information à partir d'information préexistante et de certains mécanismes de transformation de l'information.

Par exemple, « Socrate est un homme, les humains sont mortel. Alors, Socrate est mortel ».
prémisse conclusion

Les lois de transformation de l'information (à partir de prémisse, on tire la conclusion) sont supposées guider l'homme dans son raisonnement. La logique s'intéresse à leurs études.

Un langage logique est défini par une *syntaxe,* c'est-à-dire un système de *symboles* et de *règles* pour les combiner sous formes de *formules*. De plus, une *sémantique* est associée au langage. Elle permet de l'interpréter, c'est-à-dire d'attacher à ces formules ainsi qu'aux symboles une signification. Un *système de déduction* permet de *raisonner* en construisant des *démonstrations des formules*.

2.1.2 Utilité de la logique

On peut se demander à quoi sert la logique en tant que sciences, puisqu'elle ne recouvre que des connaissances évidentes. Mais pour l'information en particulier, *l'utilité de la logique est du même ordre que l'arithmétique*. Cette utilité est liée à la *dimension des problèmes traités* et à l'intérêt *d'automatiser la résolution* de ces problèmes.

Par exemple, on produit (6x6)/2=18 sans effort mais il n'en va pas de même pour 34672x16755. Ici, on utilisera une calculatrice ou un ordinateur dont la programmation a demandé la mise en œuvre de règles arithmétiques bien formalisées.

Plus que la taille des problèmes, c'est leur généralisation qui demande l'élaboration d'une science. L'égalité 1+2+…+10=55 présente un intérêt nettement moindre que l'égalité $\sum_{i=1}^{n} i = (n \times (n+1))/2$. Cette égalité ne se déduit pas seulement de tables

arithmétiques ; *une arithmétique est nécessaire*. Il en va de même en logique, où l'on formalise des *règles* générales, tel que la classique règle de récurrence.

Le fait que la logique soit plutôt simple rend possible *l'automatisation du raisonnement*. La logique est donc une clé de l'intelligence artificielle, plus précisément le problème de la représentation des connaissances, et permet en particulier la programmation du système expert.

2.1.3 Validité d'un argument ; schéma d'argument ; constantes logiques

Un *argument* peut être considéré comme une suite de phrases, dont les premières sont appelées *prémisses* et dont la dernière est la *conclusion*. En général, on considère que les prémisses d'un argument fournissent une raison de croire à la vérité de sa conclusion. En ce sens, argumenter c'est transférer la vérité des prémisses vers la conclusion.

Une classe particulière d'arguments intéresse le logicien : les arguments valides. Un argument est *valide* ssi (si, et seulement si) il n'est pas possible que sa conclusion soit fausse alors que ses prémisses sont vraies. En d'autres termes : **si les prémisses d'un argument valide sont vraies, alors sa conclusion doit être vraie également.** Les arguments valides sont ceux qui garantissent la préservation de la vérité. On dit que la conclusion d'un argument valide est une conséquence logique de ses prémisses.

Il est important de noter que cette définition est indifférente au fait que les prémisses soient, en réalité, vraies ou fausses. Voici deux exemples d'arguments. Le premier est valide alors que ses prémisses et la conclusion sont fausses. Le deuxième n'est pas valide alors que les prémisses et la conclusion sont vraies.

Ex 1 :

- Tous les joueurs professionnels de tennis sont des femmes.
- Cristiano Ronaldo est un joueur professionnel de tennis.
- Cristiano Ronaldo est une femme.

Ce qui est important, c'est que si on accepte les prémisses (si on croit à leur vérité), on doit aussi accepter la conclusion de l'argument. Le premier exemple montre donc que la vérité de ses prémisses n'est pas nécessaire à la validité d'un argument.

Ex. 2 :

- Tous les hommes sont des mammifères.
- Quelques mammifères sont bipèdes.
- Tous les hommes sont bipèdes.

Dans l'ex. 2, la vérité des prémisses n'est pas suffisante pour que l'argument soit valide : accepter les prémisses n'oblige pas à accepter la conclusion (même si ici, la conclusion est vraie). On pourrait imaginer une situation où les prémisses de l'ex. 2 sont vraies mais la conclusion fausse (il suffirait que quelques hommes ne soient pas bipèdes).

La vérité ou la fausseté des prémisses n'affecte donc pas la validité d'un argument. Puisque ce n'est pas la vérité des prémisses qui importe, qu'est-ce qui détermine la validité d'un argument ?

L'argument suivant est valide :

Ex. 3 :

- Camille gagne la partie d'échecs ou Benjamin gagne la partie d'échecs.
- Benjamin ne gagne pas la partie d'échecs.
- Camille gagne la partie d'échecs.

On n'a même pas besoin de savoir si Camille et Benjamin jouent ou non aux échecs. La validité de l'argument n'a en fait rien avoir avec Camille et Benjamin. Car on peut remplacer leurs noms par celui d'autres personnes, ainsi que remplacer « gagner la partie d'échec » par autre chose, et conserver la validité de l'argument :

Ex. 4 :

- Jean paye l'addition ou Antoine paye l'addition.
- Antoine ne paye pas l'addition.
- Jean paye l'addition.

Il paraît évident que la validité de l'argument ne dépend finalement que du fait qu'il consiste en la disjonction de deux phrases dans la première prémisse, que la deuxième de ces phrases est niée dans la deuxième prémisse et enfin que la conclusion est la première des phrases. C'est en vertu de cette forme particulière que ces deux arguments sont valides. On peut représenter schématiquement cette forme par :

- A ou B.
- Non B.
- A.

On appelle une telle schématisation un *schéma d'argument*. Les lettres A et B peuvent être remplacées par n'importe quelle phrase, on obtiendra un argument valide. Par contre, on ne peut remplacer « ou » et « non » par n'importe quelle expression et être sûr de sauvegarder la validité du schéma d'argument.

Ce ne sont donc que certains types d'expressions qui sont importants quand on veut déterminer la validité d'un argument. Par exemple, la signification de « ou » tient une part

importante dans la validité du schéma ci-dessus. En explorant la validité des schémas d'argument où cette expression apparaît, on explore du même coup la signification de ce mot. La même chose vaut évidemment pour toutes les expressions qui peuvent jouer un rôle déterminant dans la validité de schémas d'argument.

Pour être plus précis : la *signification* des phrases A et B n'est pas déterminante pour la validité de l'argument1, mais celles de « A ou B » et de « Non A » le sont. Or il se trouve que pour connaître la signification de ces phrases, on s'appuie sur la signification de la disjonction pour la première et sur la signification de la négation pour la deuxième.

Dans le cadre qui nous intéresse ici, les expressions dont la signification est importante pour la validité d'argument sont certaines conjonctions grammaticales (ou, et, si ...alors) et la négation.

Dans les *schémas d'arguments* tels qu'on les a présentés, il y a un petit nombre de mots dont la signification est essentielle à la validité de l'argument. Ces termes sont les constantes logiques. Lorsqu'un argument est valide, on peut altérer la signification de tous les termes qui le composent, si on ne touche pas aux constantes logiques (et que l'on respecte la substitution uniforme), alors le nouvel argument est encore valide.

2.1.4 Raisonnement et calcul

Le raisonnement est proche du calcul, qui lui aussi transforme l'information. Étant donné un triangle dont la base et la hauteur sont de 6cm, on sait que l'air du triangle est de 18cm^2 (nouvelle information). le mécanisme de production est la règle classique s=(bxh)/2.

La logique la plus simple est celle des propositions. Il s'agit bien d'un calcul, dans lequel les objets ne sont pas les nombres et les expressions numériques mais les valeurs de vérité (vrai et faux) et les propositions sont des formules susceptibles d'être vraies ou fausses.

Exemple : l'information préexistante comporte deux énoncés :

P1 : Pour sortir sous la pluie, je prends mon parapluie.

P2 : Je suis dehors sans parapluie.

Le mécanisme de calcul, ou plutôt la règle de déduction est la suivante (Modus Tollens):

$(A \Rightarrow B, \neg B) \rightarrow \neg A$ « Si A implique B est vrai, et si B est faux, alors A est faux ».

On a instancié A en « il pleut » et B en « je sors muni d'un parapluie »

P1 : $A \Rightarrow B$

P2 : $\neg B$

La nouvelle information que l'on peut obtenir ici est $\neg A$: « il ne pleut pas ».

2.1.5 Programmer en logique

Puisque la logique est, dans une certaine mesure, un calcul, elle peut donner naissance à un langage de programmation logique. Le langage PROLOG (PROgrammer en LOGique) est le plus utilisé des langages basés sur la logique. Il se prête bien à la résolution d'une vaste classe de problèmes.

2.1.6 Logiques existantes

La principale différence entre les différentes logiques existantes réside dans la position ontologique, autrement dit ce qui est supposé relativement à la nature de la réalité. Une logique peut aussi être caractérisée par ses positions épistémologiques, autrement dit les états possibles des connaissances qu'elle autorise compte tenu de chaque fait. Le tableau suivant synthétise cinq logiques différentes.

Tableau : les langages formels et leurs positions ontologiques et épistémologiques

Langage	Position ontologique (ce qui existe dans le monde)	Position épistémologique (attitude d'un agent à l'égard des faits)
Logique propositionnelle	Faits	Vrai/faux/inconnu
Logique du premier ordre	Faits, objets, relations	Vrai/faux/inconnu
Logique temporelle	Faits, objets, relations, temps	Vrai/faux/inconnu
Théorie des probabilités	Faits	Degré de croyance $\in[0,1]$
Logique floue	Faits avec degré de vérité $\in$ [0,1]	Valeur dont l'intervalle est connu

Nous présenterons dans la suite de ce chapitre deux types de logique : la logique propositionnelle (calcul des propositions) et la logique du premier ordre (calcul des prédicats).

2.2 La logique propositionnelle : logique d'ordre 0

2.2.1 Introduction

La logique propositionnelle est une théorie logique qui définit les lois formelles du raisonnement. Elle s'intéresse à des énoncés (les ***propositions***) qui peuvent être soit vrais soit faux, ainsi qu'aux rapports entre ces énoncés.

Exemple :

Soit la phrase P : « il fait beau et je suis en vacance ». Cette phrase sera vraie si les deux propositions q : « il fait beau » et r : « je suis en vacance » sont vraies toutes les deux. Si l'une des deux est fausse, alors la phrase P sera fausse.

Exercice :

Les énoncés suivant sont ils des propositions ?

1)	la lune tourne autour de la terre	Proposition
2)	la lune tourne autour de la galaxie	Proposition
3)	il fait chaud	n'est pas une proposition
4)	je suis en train de mentir	n'est pas une proposition
5)	L'IUT est à Talence ou à Gradignan	Proposition

2.2.2 Syntaxe

Un langage L pour la logique propositionnelle a comme symboles primitifs :

- un ensemble dénombrable de ***variables propositionnelles*** *(ou atomes*) qui ne changent pas tout au long du calcul propositionnel, dénotées par des lettres minuscules de l'alphabet éventuellement indexées : {p, q, r, p_1,..., p_n, q_1, ...} ;
- les ***connecteurs*** : $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$, $\neg$: correspondant respectivement à la conjonction (et), la disjonction (ou), le conditionnel (l'implication, si… alors), l'équivalence et la négation (non) ; $\neg$ est un connecteur à une place (unaire) et $\wedge$, $\vee$, $\rightarrow$ et $\leftrightarrow$ des connecteurs à deux places (binaires) ;
- les ***parenthèses*** *(délimiteur)* : (et).

A partir de ce vocabulaire de base, on peut former des ***expressions***, c'est-à-dire des chaînes de symboles primitifs du langage.

Exemples :

- p
- $\neg$ ($\neg$p)
- $\neg$)$\vee$pq$\rightarrow$ ($\wedge$

Mais seules les *expressions* qui sont capables de dénoter une *proposition* nous intéressent, c'est-à-dire seulement l'expression dont la structure est identique à celle d'une proposition. Ces expressions « correctes » sont désignées sous le terme ***expressions bien formées*** (e.b.f.) ou formules.

2.2.3 Définition : Expression bien formé (ebf)

Une expression bien formé (ou formule bien formée fbf ; well formed formula wff) est défini récursivement par les conditions suivantes :

(i) Les atomes sont des e.b.f.

(ii) Si A est une e.b.f., alors $\neg$A est une e.b.f.

(iii) Si A et B sont des e.b.f., alors (A$\wedge$B), (A$\vee$B), (A$\rightarrow$B) et (A$\leftrightarrow$B) sont des e.b.f.

(iv) Rien d'autre n'est une e.b.f. .

Dans cette définition, on utilise ***les lettres capitales*** de l'alphabet (A,B,C…) comme des variables désignant une expression quelconque. Mais en général, ces capitales désignent des formules quelconques (c'est-à-dire des e.b.f.).

(ii) et (iii) sont les clauses inductives de la définition : partant d'un cas connu (A,B) elle permettent de conclure à un nouveau cas. Une clause telle que (iv) est parfois appelée la clause de clôture d'une définition : elle permet de fermer l'ensemble des ebf : ***une expression ne peut être une formule qu'en vertu de l'une des clauses (i) à (iii)***.

2.2.4 Règles de suppression des parenthèses

Une ebf s'écrit en utilisant des parenthèses. Ces parenthèses peuvent être supprimées en respectant les règles suivantes :

- Priorité décroissante des connecteurs dans l'ordre $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$;
- On omet les parenthèses les plus externes ;
- Quand il y a un seul connecteur, l'association de fait de gauche à droite.

Exemple : ((p$\rightarrow$q) $\leftrightarrow$($\neg$r)) se réécrit p$\rightarrow$q $\leftrightarrow\neg$r

2.2.5 Sémantique des propositions

En général, la sémantique étudie les rapports entre les signes d'un langage et les objets (interprétation). Du point de vue extensionnel de la logique mathématique, la sémantique renvoie à l'étude de la façon dont les formules dénotent des valeurs de vérité. Les valeurs de vérité sont des objets qui n'appartiennent pas à notre langage logique L, mais dont on se sert pour parler de L. Dans la logique classique que nous étudions ici, il n'y a que deux valeurs de vérité, le vrai et le faux, auxquels réfèrent respectivement les signes 1 et 0.

Interpréter une proposition (ebf) consiste à lui attribuer une valeur logique : vrai ou faux. Pour pouvoir interpréter une proposition, il est nécessaire d'avoir une interprétation de chaque atome qui lui compose. Une interprétation d'un ensemble d'atomes $\{A_1, A_2, \ldots, A_n\}$ est une fonction de valuation I de l'ensemble $\{A_1, A_2, \ldots, A_n\}$ dans {V, F} tel que $I(A_i)$ est une valeur

de vérité attribué par la fonction I à A_i. Si l'ensemble comporte n atomes, il y a 2^n interprétations. Pour les examiner de façon exhaustive, on utilise des ***tables de vérité***.

Une fois qu'on a établit une interprétation à une expression, sa valeur de vérité peut être déterminée par l'application répétée des règles sémantiques sur des portions de plus en plus grande jusqu'à ce qu'on détermine une seule valeur de vérité pour l'expression.

Les règles sémantiques sont :

- $I(A \wedge B)=V$ si $I(A)=I(B)=V$; =F sinon
- $I(A \vee B)=F$ si $I(A)=I(B)=F$; =V sinon
- $I(\neg A)=V$ si $I(A)=F$; =F sinon
- $I(A \rightarrow B)=F$ si $I(A)=V$ et $I(B)=F$; =V sinon
- $I(A \leftrightarrow B)=V$ si $I(A)=I(B)$; =F sinon

Remarque : Seules les formules sont susceptibles de « recevoir » une valeur de vérité. Les expressions qui ne sont pas bien formées n'en reçoivent pas.

Exemple 1 : soit l'expression $E=(A \wedge \neg B)$

- Une interprétation I_1 qui assigne les valeurs de vérité vrai à A et faux à B → $I_1(A)=V$, $I_1(B)=F$. L'expression E est vraie dans cette interprétation $I_1(E)=V$.
- Une interprétation I_2 qui assigne les valeurs de vérité vrai à A et vrai à B → $I_2(A)=V$, $I_2(B)=V$. L'expression E est fausse dans cette interprétation $I_2(E)=F$.

Exemple 2 : soit ebf G : $A \rightarrow (B \vee \neg C)$

- $I_1(A)=V$, $I_1(B)=F$, $I_1(C)=V$ → $I_1(G)=F$.
- $I_2(A)=F$, $I_2(B)=V$, $I_2(C)=F$ → $I_2(G)=V$.

A	B	C	$\neg C$	$B \vee \neg C$	$A \rightarrow (B \vee \neg C)$
V	F	V	F	F	F
F	V	F	-	-	V

Exercice : Donner l'interprétation selon des formules suivantes :

- $F_1 : p \wedge q$
- $F_2 : p \wedge (\neg r \vee q)$
- $F_3 : p \vee \neg q$

1) Soit l'interprétation I_1 définit comme: I_1: \ $I_1(p)=V$, $I_1(q)=F$, $I_1(r)=V$.

$I_1(F_1)=F$, $I_1(F_2)=F$, $I_1(F_3)=V$

2) Soit l'interprétation I_2 définit comme: I_2: \ $I_2(p)=F$, $I_2(q)=V$, $I_2(r)=F$.

$I_2(F_1)=F$, $I_2(F_2)=F$, $I_2(F_3)=V$

3) Pouvez-vous trouver une interprétation qui rende vrai les 3 formules ?

p	**q**	**r**	**p∧q**	**p∧ (¬r∨q)**	**p∨¬q**
V	V	V	V	V	V
V	V	F	V	V	V
V	F	V	F	F	V
V	F	F	F	V	V
F	V	V	F	F	V
F	V	F	F	F	V
F	F	V	F	F	V
F	F	F	F	F	V

Donc :

I_3: \ $I_3(p)=V$, $I_3(q)=V$, $I_3(r)=V$

I_4: \ $I_4(p)=V$, $I_4(q)=V$, $I_4(r)=F$

2.2.6 Modèle

Si l'on considère qu'une interprétation est à monde possible, il est intéressant de se demander pour une formule donnée s'il existe un monde dans lequel cette formule est vraie. Un tel état est appelé Modèle. Un modèle d'une formule f est une interprétation I tel que I(f)=V. Il peut exister 0 ou 1 ou plusieurs modèles pour une formule donnée. Un modèle d'un ensemble de formules $F=\{f_1, f_2, \ldots, f_n\}$ est une interprétation qui rend vraie chaque formule de F. on peut spécifier les propriétés des expressions bien formées suivantes :

- Une ebf est une ***formule valide*** (ou ***Tautologie***) si l'expression a la valeur **vrai** dans *toute interprétation*. Les tautologies sont des vérités universelles qui ne dépendent pas de l'état du monde. Exemple : $p\vee\neg p$, $((p\rightarrow q)\wedge p)\rightarrow q$
- Une ebf est ***inconsistante*** (antilogie, contradiction, insatisfaisable) si l'expression a la valeur **faux** dans *toute interprétation* (elle n'a aucun modèle). Exemple : $p\wedge\neg p$
- Une ebf est ***satisfaisable*** (consistante, réalisable) si elle a au moins un modèle, c'est-à-dire il existe au moins une interprétation dans laquelle l'expression a la valeur **vrai**.
- Une ebf est dite ***contingente*** (Ce qui peut arriver ou ne pas arriver) lorsque la valeur de vérité dépend d'une décision extra logique prenant en compte le contenu de la proposition (on se réfère donc à ce qu'elles signifient).

Valide	Non valide	
Tautologie	Consistant Réalisable	Inconsistant Antilogie

Exemples :

- La phrase « elle est venue » est vraie ou fausse en fonction du fait qu'elle est ou pas venue. C'est une proposition *contingente*.
- La phrase « elle est venue ou elle n'est pas venue » ne donne pas d'importance au contenu. Le contenu est immatériel, c'est une *tautologie* !
- La phrase « Elle est venue et elle n'est pas venue » est fausse qu'elle soit venue ou qu'elle ne le soit pas. Il s'agit donc d'une *antilogie*.

Exercice : Soit l'ensemble F : {p∧q, p∨q}

1) Vérifier que l'ensemble F est satisfaisable.

Il existe un modèle pour F qui est l'interprétation I tel que I(p)=I(q)=V. Donc l'ensemble F est satisfaisable.

2) Combien de modèles différents existe-t-il ? un seul modèle.

p	q	p∧q	p∨q
V	V	V	V
V	F	F	V
F	V	F	V
F	F	F	F

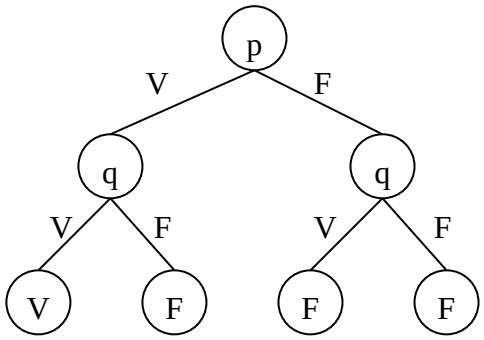
3) Que peut-on dire de F si on lui ajoute ¬p ? F n'a aucun modèle donc F est inconsistante.

p	q	p∧q	p∨q	¬p
V	V	V	V	F
V	F	F	V	F
F	V	F	V	V
F	F	F	F	V

2.2.7 Arbre sémantique

Pour vérifier si une formule est une tautologie, on peut construire un arbre sémantique. L'arbre a autant de niveau qu'il a de propositions atomiques dans la formule. Chaque niveau ajoute 2 branches qui portent les étiquettes vrai ou faux au nœud inférieur de l'arbre représente une interprétation des propositions atomiques. L'arbre représente toutes les interprétations possibles.

Exemple :



p∧q	V	F	F	F
(p∧q) ∨p	V	V	F	F
¬(¬p∨¬q)	V	F	F	F
(p∧q) →q	V	V	V	V

On constate que les propositions (p∧q) et (¬(¬p∨¬q)) prennent les mêmes valeurs de vérité quelque soit l'interprétation. On voit également que ((p∧q) →q) est une tautologie.

2.2.8 Équivalence

Deux formules sont équivalentes si pour toute interprétation, elles prennent les mêmes valeurs de vérité, c'est-à-dire elles ont les mêmes modèles. Voici quelques équivalences utiles :

- *Double négation* : ¬(¬A)=A
- *Associativité* : (A∧B) ∧C=A∧(B∧C) ; (A∨B) ∨C=A∨ (B∨C)
- *Commutativité* : A∧B=B∧A ; A∨B=B∨A
- *Distributivité* : A∨ (B∧C)=(A∨B) ∧ (A∨C) ; A∧ (B∨C)=(A∧B) ∨ (A∧C)
- *Loi de Morgan* : ¬ (A∧B)= ¬A∨¬B ; ¬ (A∨B)= ¬A∧¬B
- *Implication* : A→B=¬A∨B
- *Équivalence* : A↔B=(A→B) ∧(B→A)
- *Loi d'absorption* : A∨ (A∧B)=A ; A∧(A∨B)=A
- *Loi de simplification* : A∨F=A ; A∨V=V ; A∧V=A ; A∧F=F ;
- *Complémentarité* : A∧¬A=F ; A∨¬A=V
- *Identité* : A∧A=A ; A∨A=A

2.2.9 Conséquences logiques

La notion de conséquence logique est destinée à exprimer le fait qu'une proposition donnée est nécessairement ***vraie*** lorsqu'un ensemble d'autre proposition le sont.

Par exemple : si on sait que A est vrai et que B est vrai alors on déduit que (A∧B) est vrai. De même, si on sait que (A∨B) est vrai et (¬B) est vrai, on en déduit que A est vrai.

$$\{A, B\} \models A \wedge B$$

$$\{A \vee B, \neg B\} \models A$$

On formalise cette notion de la manière suivante : *P est une conséquence logique de l'ensemble {P_1, P_2, …, P_n}*. On note $\{P_1, P_2, \ldots, P_n\} \models P$. Tout modèle de $\{P_1, P_2, \ldots, P_n\}$ est forcément un modèle de P.

Exemple : $\{p \rightarrow q, \neg q\} \models \neg p$ (Règle de Modus Tollens)

p	q	$p \rightarrow q$	$\neg q$	$\neg p$
V	V	V	F	F
V	F	F	V	F
F	V	V	F	V
F	F	V	V	V

2.2.10 Manipulation syntaxiques

La définition de la notion de la conséquence logique ne nous donne pas de moyen très efficace pour vérifier qu'une proposition p est conséquence logique d'un ensemble de proposition R (E |= p). En effet, il faut :

- Trouver tous les modèles de E ;
- Pour chaque modèle I de E, vérifier que I(p)=V.

Si on utilise n propositions atomiques, le nombre de modèles peut s'évaluer à 2^n, ce qui peut entrainer un temps de calcul très important (avec 30 atomes, on aura presque un milliard de modèles à tester). D'autres part, la définition ne nous aide pas à trouver quelle sont les conséquences logiques de E.

Pour cela, on a recourt à des ***techniques purement syntaxiques*** qui vont nous permettre de déterminer des conséquences logiques.

2.2.11 Règles d'inférence

Une règle d'inférence est *une opération purement syntaxique* que ne dépend que de l'écriture des propositions qui produit une nouvelle proposition à partir d'une ou plusieurs autres propositions :

- **Modus Ponens** : la règle s'écrit comme suit : $\{A \rightarrow B, A\} \models B$; et se lit : à partir de $A \rightarrow B$ et de A, on en déduit B. A et B peuvent être n'importent quelle proposition atomique ou composée.
- **Modus Tollens** : la règle s'écrit comme suit : $\{A \rightarrow B, \neg B\} \models \neg A$

Exemple : $\{p, \neg r, p\rightarrow(q\rightarrow r)\}$

$\{p\rightarrow(q\rightarrow r), p\} \models q\rightarrow r$ (MP)
$\{p, \neg r, p\rightarrow(q\rightarrow r), q\rightarrow r\}$
$\{q\rightarrow r, \neg r\} \models \neg q$ (MT)
$\{p, \neg r, p\rightarrow(q\rightarrow r), q\rightarrow r, \neg q\}$

2.2.12 Formes normales

Les formes normales furent découvertes dans leurs principes par le logicien Herbrand. Les formes normales conjonctives et disjonctives utilisent la négation, la conjonction et la disjonction.

- La ***forme normale conjonctive***, FNC, est inventée par le philosophe logicien Pears. C'est la transformation d'une proposition complexe en une *conjonction ($\wedge$) de disjonction ($\vee$)* avec seulement des littéraux. Exemple : $((p\vee\neg q\vee r) \wedge (p\vee r))$
- La ***forme normale disjonctive***, FND, est inventée par le mathématicien logicien Boel. C'est la transformation de proposition complexe en *disjonction ($\vee$) de conjonction ($\wedge$)* ne comportant que des littéraux. Exemple : $((p\wedge\neg q\wedge r) \vee (p\wedge r))$

Avec un ***littérale*** est un atome ou la négation d'un atome ; la disjonction des littéraux est appelée ***clause*** ; La conjonction de clauses est appelée ***ensemble de clauses***.

Toute formule peut être écrite indifféremment (en FND ou FNC). Cette propriété est basée sur les équivalences et les propriétés usuelles. Grâce à ces propriétés, il est possible de simplifier les expressions et de trouver ainsi des valeurs de vérité plus facilement. Ces simplifications sont aussi utilisées lorsque les formules sont mises sous formes clausale afin de préparer leur résolution par la méthode de Robinson par exemple.

2.2.13 Algorithme de normalisation

Il existe 2 méthodes pour mettre une formule quelconque sous forme normale :

- En utilisant une ***suite d'équivalence*** : Elle peut se résumer par les règles suivantes :
  - La suppression des équivalences ;
  - La suppression des implications ;
  - Déplacement de la négation à l'intérieur de la formule ;
  - La distributivité ($\wedge$) et ($\vee$).
- En déterminant la ***fonction booléenne associée***, puis à construire une formule sous forme normale. Les étapes peuvent se résumer de la manière suivante :
  - Déterminer les interprétations I \ I(F)=V ;

- À chaque interprétation I_k tel que $I_k(F)=V$ est associée une formule F_k qui est une conjonction de littéraux ;
- La formule F est finalement obtenue par la disjonction des F_k.

Exemple : Soit $F = \neg(p \leftrightarrow (q \rightarrow r))$

- 1ère méthode :

$$\neg[(p \rightarrow (q \rightarrow r)) \wedge ((q \rightarrow r) \rightarrow p)] \equiv \neg[(\neg p \vee (\neg q \vee r)) \wedge (\neg (\neg q \vee r) \vee p)]$$

$$\equiv [\neg (\neg p \vee (\neg q \vee r)) \vee \neg (\neg (\neg q \vee r) \vee p)] \equiv [(p \wedge q \wedge \neg r) \vee ((\neg q \vee r) \wedge \neg p)]$$

$$(p \wedge q \wedge \neg r) \vee (\neg q \wedge \neg p) \vee (r \wedge \neg p) : \text{FND}$$

- 2ème méthode :

p	q	r	q→r	¬(p↔(q→r))	
V	V	V	V	F	
V	V	F	F	V	$F_1=p \wedge q \wedge \neg r$
V	F	V	V	F	
V	F	F	V	F	
F	V	V	V	V	$F_2= \neg p \wedge q \wedge r$
F	V	F	F	F	
F	F	V	V	V	$F_3= \neg p \wedge \neg q \wedge r$
F	F	F	V	V	$F_4= \neg p \wedge \neg q \wedge \neg r$

$$F= (p \wedge q \wedge \neg r) \vee (\neg p \wedge \underbrace{q \wedge r) \vee (\neg} p \wedge \neg \underbrace{q \wedge r) \vee (\neg} p \wedge \neg q \wedge \neg r)$$

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

2.2.14 Conclusion

Malheureusement, la logique propositionnelle est un langage trop faible pour représenter précisément les connaissances relatives à des environnements complexes. Par exemple, le Traitement Automatique des Langues Naturelles (TALN) qui consiste à extraire le contenu sémantique de données textuelles. Dans la section suivante, nous étudions la logique du premier ordre, également appelée calcul des prédicats du premier ordre (First Order Logic or First Order Predicate Calculus). C'est une version étendu de la logique propositionnelle en ajoutant les quantificateurs $\forall$ et $\exists$.

2.3 La logique des prédicats : logique d'ordre 1

2.3.1 Qu'est ce qu'un prédicat

Le prédicat fait partie de la proposition :

- ***Énoncé*** : les phrases du langage courant.
- ***Proposition*** : énoncé auquel on attribue une valeur de vérité soit vraie (1) soit fausse (0)
- ***Prédicat*** : un énoncé qui peut contenir plusieurs variables et qui devient une proposition chaque fois que les variables sont fixées dans leurs ensembles respectifs.

Si je dis : « Le chien mange. » « Le chien » est le sujet. « Mange » n'est pas un prédicat en grammaire classique. Il faut donc transformer cette phrase en « Le sujet EST mangeant. » On obtient ainsi l'équivalent de la forme « S est P » où P est l'attribut du sujet. En logique des prédicats, la formule de cette phrase est plus complexe : « S est mangeant. » Le calcul se fait donc sur une partie de la proposition simple.

Il existe différents types de prédicats :

- sous forme **verbale**
- **Intentionnel**. Par exemple, « le chien est beau » en admettant qu'il y ait une définition de la beauté.
- **Extensionnel** implique une opération de collation des individus pour constituer un tout, un ensemble ou une classe qui correspondra au prédicat. Ainsi, avec le projet extensionaliste (Cf. logique des propositions), si on peut construire un ensemble « beau », on aurait le « chien » dans cet ensemble, le « chien » appartient à cet ensemble.

Un prédicat est avant tout un concept défini en extension. Par exemple, « rouge » est un concept intentionnel. Cependant, si je dis que « avoir telle longueur d'onde qui donne la sensation du rouge », je fixe un concept extensionnel.

Exemple :

- Les entiers x et x+2 sont premiers ;
- Il existe un entier x tel que x et x+2 sont premiers ;
- Il existe une infinité de nombres premiers x tel que x+2 est aussi premier.

En fait, même les raisonnements courants impliquent des propositions paramétriques comme le montre l'exemple classique suivant :

- Tous les hommes sont mortels
- Socrate est un homme
- Socrate est mortel

On voit que la 3ème proposition est conséquence logique des deux premières, mais une analyse purement propositionnelle ne rendre pas compte de ce fait. Nous avons donc besoin d'un langage formel plus riche que le calcul des propositions qui permettra d'exprimer des propriétés vraies pour certains individus pris dans un ensemble.

Exercice :

Les énoncés suivants sont ils des prédicats ?

1)	Pour $x \in Q$, $x^2 \neq 2$	Prédicat
2)	$X \in E$, $X>10$	Prédicat
3)	P est un nombre premier	n'est pas un prédicat
4)	Pour f : R=>R, f est paire	Prédicat
5)	x est impair	n'est pas un prédicat (x n'est pas définit)
6)	Sin(x) est impair, $x \in R$	n'est pas un prédicat

2.3.2 Syntaxe

Un langage L pour la logique des prédicats est défini par l'alphabet de symboles suivant :

1. un ensemble de symboles de constantes *Cons* = {a, b, c, c_1, c_2, c_3, …}
2. un ensemble de symboles de prédicats *Pred* = {P, Q, R, P_1, P_2, P_3, …}
3. un ensemble de symboles de fonctions *Fonc* = {f, g, h, f_1, f_2, f_3, …}
4. un ensemble de variables *Var* = {x, y, z, x_1, x_2, x_3, …}
5. des quantificateurs {∀, ∃}. ∃ (il existe) est le quantificateur existentiel ; ∀ (quel que soit) est le quantificateur universel.
6. l'ensemble de connecteurs de la logique propositionnelle : {¬, ∧, ∨, →, ↔}
7. une parenthèse ouvrante et d'une parenthèse fermante {), (}

Remarques :

- Les ensembles 1 – 3 sont les ***symboles non logiques*** : des langages de premier ordre différents ont des symboles de constantes, de prédicats et de fonctions différentes.
- Les ensembles 4 – 7 sont les ***symboles logiques*** : ils sont communs à tous les langages du premier ordre.
- ***L'arité*** d'un prédicat est le *nombre d'argument du prédicat*. C'est un nombre positif. Si le prédicat est d'arité 0, il correspond à la notion de *proposition* de la logique des propositions.
- ***L'arité*** d'une fonction est le *nombre d'argument de la fonction*. C'est un nombre positif. Si la fonction est d'arité 0, elle correspond à la notion de *constante*.

2.3.3 Quelques définitions

a. Les termes

Par définition, tout terme est engendré par application des deux lois suivantes :

- Les constantes et les variables sont des termes
- si f est un symbole de fonction d'arité n (n>=1) et si $t_1..t_n$ sont des termes alors $f(t_1..t_n)$ est un terme

Exemple :

successeur(X) est un terme ; poids(b) est un terme

successeur(poids(b)) est un terme ; P(X, bleu) n'est pas un terme

poids(P(X)) n'est pas un terme

b. Les atomes

Par définition, tout atome est engendré par application des deux lois suivantes

- Les propositions sont des atomes
- si P est un prédicat d'arité n (n >=1) et si $t_1..t_n$ sont des termes alors $P(t_1..t_n)$ est un atome

Exemple :

P(X, bleu) est un atome ; VIDE est un atome

ENTRE(table,X, appui(fenetre)) est un atome ; successeur(X) n'est pas un atome

appui(fenetre) n'est pas un atome

c. Les formules bien formées fbf

Le langage est constitué de l'ensemble des Formules Bien Formées (appelées aussi : FBFs ou Well Formed-Formula WFF) ou expressions bien formées défini comme suit :

- Les atomes sont des fbfs
- si F et G sont des fbfs alors ($\neg$G), (F$\wedge$G), (F$\vee$G), (F$\rightarrow$G) et (F$\leftrightarrow$G) sont des fbfs
- si G est une fbf et X une variable alors ($\exists$X)G et ($\forall$X)G sont des fbfs.
- toutes les fbfs sont obtenues par application des 3 règles ci-dessus.

Exemples :

- ($\exists$X) ($\forall$Y) (P(X, Y) $\vee$ Q(X, Y) $\rightarrow$ R(X)) et ((Ø,(P(a) $\rightarrow$ P(b))) $\rightarrow$ Ø,P(b)) sont des fbfs
- (Ø,(f(a))) et f(P(a)) ne sont pas des fbfs

d. Variable libre et liée

Une occurrence de X est ***liée*** dans une fbf si elle est dans le champ d'un quantificateur $\forall$ ou $\exists$ qui l'utilise ou si elle le suit. Sinon cette occurrence (variable) est dite ***libre***.

Exemples :

$A = \forall X\ (\exists Y\ P(X, Y) \wedge Q(X, Z)) \wedge R(X)$
liée liée liée liée liée libre libre

$B = \forall X\ ((\exists Y\ Q(X, Y)) \wedge P(X, Y, Z))$
liée liée liée liée liée libre libre

Une variable est ***libre*** (resp. liée) si au moins une de ses occurrences est ***libre*** (resp. liée).

Exemples :

Variables libres de A = {Z, X}

Variables liées de A = {X, Y}

Variables libres de B = {Z, Y}

Variables liées de B = {X, Y}

Une formule dont toutes les variables sont liées est dite ***fermée***. Si ce n'est pas le cas, la formule est dite ***ouverte***.

Exemple :

$\forall X\ \exists Y\ (P(X, Y) \wedge \forall Z\ R(X, Y, Z))$
liée liée liée liée liée

e. Substitution et instanciation

L'opération de ***substitution*** consiste à remplacer certaines variables libres d'une formule F par des termes. Soit F une formule où x_1, …, x_n sont des variables libres et σ la substitution $(t_1/x_1,.., t_n/x_n)$, donc $F\sigma$ est la formule F où toutes les occurrences des x_i sont remplacées par t_i.

Exemple : Soit $F=Q(x_1, y_1) \leftrightarrow \forall x_2(R(x_2, z_1) \vee s(x_2, y_1))$ et $\sigma=(z_8/x_1, g(a, b)/y_1)$

Donc $F\sigma=Q(x_8, g(a,b)) \leftrightarrow \forall x_2(R(x_2, z_1) \vee s(x_2, g(a,b)))$

On dira qu'une substitution ***instancie*** x si elle remplace x par un terme où n'apparaît aucune variable.

2.3.4 Sémantique du calcul des prédicats

Le sens d'une formule sera l'une des valeurs vraie ou fausse de l'anneau booléen. Il faut donner une valeur aux variables mais aussi

- Une valeur à chaque constante
- Un moyen d'interpréter les atomes p(terme$_1$, …, terme $_n$)
- Un moyen d'interpréter les fonctions f(terme$_1$, …, terme $_m$)
- Une règle d'interprétation des formules quantifiées

a. Interprétation

Pour interpréter une formule, on commence par interpréter chaque symbole.

- un ensemble D appelé ***domaine*** de l'interprétation

- pour chaque constante c, un élément c_I de D
- pour chaque symbole de fonction n-aire f, une fonction $f_I : D^n \rightarrow D$
- pour chaque symbole de prédicat n-aire p une relation n-aire p_I sur D

$$\{<e_{1,1}, \ldots, e_{1,n}>, <e_{2,1}, \ldots, e_{2,n}>, \ldots, <e_{m,1}, \ldots, e_{m,n}>\}$$

Interprétation des variables : Comme pour la logique des propositions, on assigne une valeur à chaque variable

- Cette valeur n'est pas booléenne mais un élément de D.
- On a une *fonction d'assignation* de l'ensemble des variables dans D.

Exemple:

$a_0=[x\rightarrow 25, y\rightarrow 33, z\rightarrow -1]$

Extension d'une assignation : $a[x\rightarrow v]$: l'assignation à laquelle on a ajouté $x\rightarrow v$

Ex. $a_0[u\rightarrow 314] =[x\rightarrow 25, y\rightarrow 33, z\rightarrow -1, u\rightarrow 314]$

Interprétation des termes : soit I une interprétation du vocabulaires (const., préd., fonct.), a une assignation des variables, on définit l'interprétation des termes de la manière suivante :

- terme constant c : $I_a(c)=c_I$
- terme variable x : $I_a(x)=a(x)$
- terme avec fonction $t=f(t_1, \ldots, t_p)$: $I_a(t)=f_I(I_a(t_1), \ldots, I_a(t_p))$

Exemple :

W=constantes : a, b, c, d, fonctions : h

Interprétation I

D={1, 2, 3, 4, rouge, bleu, vert}

$a_I=4$, $b_I=7$, c_I=rouge, d_I=bleu

h_I={<1, 1>$\rightarrow$rouge, <1, 2>$\rightarrow$bleu, <3, 4>$\rightarrow$bleu, …}

I(b)=7

$I_{[z\rightarrow 3,\ x\rightarrow 1]}(z)=3$

$I_{[z\rightarrow 3]}(h(z, a)) = h_I(I_{[z\rightarrow 3]}z, I_{[z\rightarrow 3]}a) = h_I(3, 4) =$ bleu

Interprétation des atomes : Formule atomique $\alpha=p(t_1, \ldots, t_n)$

$$I_a(\alpha)= \begin{cases} \text{Vrai si } <I_a(t_1), \ldots, I_a(t_n)> \in p_I \\ \text{Faux sinon} \end{cases}$$

L'interprétation d'une formule composée de connecteurs, de négations et d'atomes se calcule

- à partir de la valeur des atomes
- selon les règles de la logique des propositions.

Exemple :

W = *constantes* : a, b, c, d ; *prédicat*: q ; Interprétation I ; D = {1, 2, 3, 4, rouge, bleu, vert}

$a_I = 4$, $b_I = 7$, c_I = rouge, d_I = bleu

q_I= {<rouge, rouge, 1> , <rouge, vert, 2>, <rouge, bleu, 4>}

$I(q(c, d, a)) = <I(c), I(d), I(a)> \in q_I = <rouge, bleu, 4> \in q_I = v$

$I(q(c, c, a)) = f$

$I_{[x \to vert,\ y \to 2]}(q(c,x,y) \vee q(c,c,a) = v$

Interprétation des formules quantifiées : La valeur d'une formule quantifiée $A = \forall x.B$ est :

vrai si pour tout $d \in D$, $I_{a[x \to d]}(B) = v$

faux sinon.

La valeur d'une formule quantifiée $A = \exists x.B$ est :

vrai s'il existe au moins un élément $d \in D$ tel que $I(B)_{a[x \to d]} = v$

faux sinon.

Exemple :

Soit W = {paul, albert, milan, tokyo, madrid (constantes), x, y (variables), ville(1-aire), personne(1-aire), habite(2-aire), }

Interprétation I:

D = {Paul, Albert, Milan, Tokyo, Madrid}

$paul_I$= Paul, $albert_I$= Albert, $milan_I$= Milan, $tokyo_I$= Tokyo, $madrid_I$= Madrid

$ville_I$= {<Milan>, <Tokyo>, <Madrid>}

$personne_I$= {<Paul>, <Albert>}

$habite_I$= {<Paul, Milan>, <Albert, Tokyo>}.

Interprétations de formules selon I:

1. I(albert) = Albert
2. $I_{[y \to Milan]}(y)$ = Milan
3. $I(ville(tokyo)) = ville_I(<I(tokyo)>) = ville_I(<Tokyo>) = v$
4. I(ville(paul)) = f
5. $I(habite(paul, tokyo)) = habite_I(<I(paul), I(tokyo)>) = habite_I(<Paul, Tokyo>) = f$

Formules quantifiées:

1. $I(\exists x.habite(x, tokyo)) = v$ car $I_{[x \rightarrow Albert]}(habite(x, tokyo)) = v$.

2. $I(\forall x.\ \exists y.habite(x, y)) = f$ car $I_{[x \rightarrow Tokyo]}(\ \exists y.habite(x, y)) = f$, entre autres.

3. $I(\forall x.personne(x) \rightarrow \exists y.habite(x, y)) = v$ car

$I_{[x \rightarrow Paul]}(personne(x) \rightarrow \exists y.habite(x, y)) = v$ (prendre $y \rightarrow$ Milan)

$I_{[x \rightarrow Albert]}(personne(x) \rightarrow \exists y.habite(x, y)) = v$ (prendre $y \rightarrow$ Tolyo)

$I_{[x \rightarrow Milan]}(personne(x) \rightarrow \exists y.habite(x, y)) = v$ (car personne(x) = f)

etc.

Attention au domaine !

D = {Paul, Albert, Milan, Tokyo, Madrid}

$ville_I$= {<Milan>, <Tokyo>, <Madrid>}

$personne_I$= {<Paul>, <Albert>}

$I(\forall x.(ville(x) \vee personne(x))) = v$

D = {Paul, Albert, Milan, Tokyo, Madrid, Amsterdam}

$ville_I$= {<Milan>, <Tokyo>, <Madrid>}

$personne_I$= {<Paul>, <Albert>}

$I(\forall x.(ville(x) \vee personne(x))) = f$

Attention aux cas d'identité des variables!

D = {Milan, Tokyo} $\cup$ N

$ville_I$= {<Milan>, <Tokyo>}

$distance_I$= {<Milan, Tokyo, 12000>, <Tokyo, Milan, 12000>}

1. $I(\forall x.\ \forall y.\ \exists z.distance(x, y, z)) = f$! (à cause du domaine)

2. $I(\forall x.\ \forall y.((ville(x) \wedge ville(y)) \rightarrow \exists z.distance(x, y, z)) = f$! (si x = y …)

Pour obtenir un modèle, ajouter : <Milan, Milan, 0>, <Tokyo, Tokyo, 0>

b. Formules satisfaisables

A est ***vraie*** pour l'interprétation I si I(A) = v. Dans ce cas I est un ***modèle*** de A

Notations : $I \models A$, $I \not\models A$ si I(A) = f.

A est ***satisfaisable*** s'il existe une interprétation I telle que $I \models A$

A est ***valide*** si pour toute interprétation I on a $I \models A$

Notation : $\models A$

A et B sont ***équivalentes*** si pour toute interprétation I on a I(A) = I(B).

Notation : $A \approx B$ ou $A \leftrightarrow B$

Exemple : $\forall x.Personne(x) \approx \neg\ \exists x.(\neg\ Personne(x))$

c. Ensemble de formules satisfaisables

F = {w_1, …, w_n} un ensemble de formules fermées, un ***modèle*** de F est une interprétation I telle que I(w_i) = v pour i =1, …, n.

F est ***satisfaisable*** s'il existe au moins un modèle de F. Sinon F est ***inconsistant***.

Exemple :

F = {p(a, b), ¬ ∃y.p(a, y)} est inconsistant

pour avoir I |= p(a, b), il faut que <a_I, b_I> appartienne à p_I mais alors I |≠ ¬ ∃y.p(a, y)

d. Conséquences logiques

w est une ***conséquence logique*** de F si pour tout modèle I de F, I |= w.

Notation: F |= w,

Si F |= f (la formule qui est toujours fausse), alors F est inconsistant.

Principe de déduction : Utile pour prouver que w est une conséquence logique de {w_1, …, w_n}.

{w_1, …, w_n}|= w si et seulement si{w_1, …, w_n, ¬ w} |= f

2.3.5 Normalisation des formules

a. Mise en forme prenex (les quantificateurs devant)

Une formule est dite en ***forme Prenex*** si tous les quantificateurs apparaissent au début. Il est toujours possible de transformer une formule en une formule équivalente sous forme Prenex.

Méthode: on applique autant de fois que nécessaire les équivalences :

¬(∀x.w1) = ∃x.(¬w1),

¬(∃x.w1) = ∀x.(¬w1),

(∀x.w1∧ w2) = ∀x.(w1∧ w2), (si x n'apparait pas dans w_2)

(∀x.w1∨ w2) = ∀x.(w1∨ w2), (si x n'apparait pas dans w_2)

Exemple de mise en forme Prenex :

∀x. ∀y.(enseigne(x, y) → ∃z.diplomé(x, z)) = ∀x. ∀y.(¬ enseigne(x, y) ∨ ∃z.diplomé(x, z))

= ∀x. ∀y. ∃z.(¬ enseigne(x, y) ∨ diplomé(x, z))

= ∀x. ∀y. ∃z.(enseigne(x, y) → diplomé(x, z)).

b. Skolemisation (suppression des ∃)

Le but est de supprimer le quantificateur ∃.

Idée: quand on a ∀x. ∃y.p(x, y) cela signifie que pour chaque x on peut trouver un y tel que p(x, y). On pourrait donc définir une fonction f qui nous fournit une bonne valeur de y à partir d'une valeur de x. On récrit : ∀x.p(x, f(x))

La fonction dépend de toutes les variables qui viennent avant y dans la formule.

Exemple :

$A = \forall x.\ \forall y.\ \exists z.(\text{enseigne}(x, y) \rightarrow \text{diplomé}(x, z))$.

$\text{skolem}(A) = \forall x.\ \forall y.(\text{enseigne}(x, y) \rightarrow \text{diplomé}(x, f(x, y)))$.

$A = \forall x.\ \exists u.\ \exists y.\ \exists z.(p(x, u) \wedge (q(u, y) \rightarrow r(y, z)))$.

$\text{skolem}(A) = \forall x.\ \forall y.(p(x, f(x)) \wedge (q(f(x), y) \rightarrow r(y, g(x, y))))$.

Propriété de la Skolemisation : Skolem(A) n'est en général pas équivalente à A !

Théorème : A est satisfaisable si et seulement si skolem(A) l'est

c. Mise en forme normale conjonctive

Une ***ou-clause*** est une formule composée de disjonctions de littéraux positifs ou négatifs.

Exemple : $p(x, y) \vee q(x, z, u) \vee \neg r(t, x)$.

Une formule est sous ***forme normale conjonctive*** (FNC) si c'est la conjonction de ou-clauses

Exemple : $(p(x, y) \vee q(x, z, u) \vee \neg r(t, x)) \wedge (\neg p(a, y) \vee t(u)) \wedge (d(x))$

Si w est une formule sans quantificateurs, il existe un algorithme qui fournit une formule équivalente w′ qui est en FNC.

Exemple de normalisation :

$\forall x.\ \forall y.((\text{enseigne}(x, y) \vee \text{agréé}(x)) \rightarrow \exists z.\text{diplomé}(x, z))$

$= \forall x.\ \forall y.(\neg\ (\text{enseigne}(x, y) \vee \text{agréé}(x)) \vee \exists z.\text{diplomé}(x, z))$

$= \forall x.\ \forall y.\ \exists z.(\neg\ (\text{enseigne}(x, y) \vee \text{agréé}(x)) \vee \text{diplomé}(x, z))$

Skolemisation :

$\forall x.\ \forall y.(\neg\ (\text{enseigne}(x, y) \vee \text{agréé}(x)) \vee \text{diplomé}(x, f(x, y)))$

FNC :

$= \forall x.\ \forall y.((\neg\text{enseigne}(x, y) \wedge \neg\text{agréé}(x)) \vee \text{diplomé}(x, f(x, y)))$

$= \forall x.\ \forall y.((\neg\text{enseigne}(x, y) \vee \text{diplomé}(x, f(x, y))) \wedge (\neg\text{agréé}(x) \vee \text{diplomé}(x, f(x, y)))$

Forme clausale (suppression des $\forall$):

$\neg\text{enseigne}(x, y) \vee \text{diplomé}(x, f(x, y)),\ \neg\text{agréé}(x) \vee \text{diplomé}(x, f(x, y))$

Chapitre 3 :

Les systèmes experts

3.1 Introduction

L'objectif de l'intelligence artificielle est à long terme de faire effectuer par un système informatique, tout ce que peut faire l'homme en matière de raisonnement. Le problème central rencontré dans tous les domaines de l'intelligence artificielle est celui de la ***modélisation de la connaissance***. En d'autres termes :

- L'IA cherche à reproduire l'intelligence **humaine**
- L'intelligence humaine est basée sur la **connaissance**
- Comment représenter la connaissance en **informatique** pour le développement de **systèmes intelligent** ?

On remarque que la représentation des connaissances représente la tâche de base pour le développement d'un système intelligent « performant ». Ces connaissances sont représentées dans la plupart des cas par une base de règles et une base de faits.

3.2 Les connaissances

3.2.1 Donnée, Information et Connaissance

- L'***information*** représente une communication au public en matière d'actualités, des renseignements sur quelques choses ou de la documentation.
- Une ***donnée*** est vu comme étant une information représentée sous une forme conventionnelle (ou codé), afin de pouvoir être traitée automatiquement. Elle représente un élément fondamental sur lequel se bâtit un raisonnement, une recherche, une étude ou une œuvre.
- La ***connaissance*** est la maîtrise intellectuelle acquise par l'apprentissage, la recherche ou l'expérience. Elle représente la capacité à mobiliser des informations pour agir. Le passage de INFORMATION à CONNAISSANCE est lié à l'expérience de l'action (pas de frontière parfaitement définie).

Par exemple : Je jette une balle de 100 grammes à une hauteur de 10 mètres. A quelle vitesse percutera-t-elle le sol? Les données (masse, hauteur, accélération terrestre) doivent être tirées de l'information (l'énoncé du problème) et de tes connaissances (la loi de Newton, l'accélération terrestre, etc.)

3.2.2 Représentation des connaissances

Pour comprendre une scène ou une histoire, il faut trouver une correspondance entre les éléments perçus et les choses connus. Représenter les connaissances en IA consiste donc à établir un lien entre le concept du monde réel et le modèle théorique donnant accès au raisonnement. La représentation de connaissance exige une définition de structures de données, générale et flexible, qui enregistrent l'information. Les connaissances peuvent être représentées en utilisant la logique.

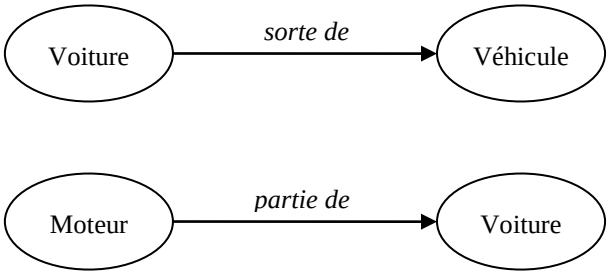
Par exemple, avec la logique du premier ordre, on utilise les prédicats ou les objets pour représenter les catégories mentales (les concepts) :

- Un objet est un membre d'une catégorie : $BF \in BallonDeFootballs$
- Une catégorie est une sous-classe d'une autre : BallonDeFootballs $\subset$ Ballons
- Tous les membres d'une catégorie ont certaines propriétés : $x \in BallonDeFootballs \Rightarrow Rond(x)$
- Les membres d'une catégorie peuvent être reconnus en identifiant certaines propriétés : $Blanc(x) \wedge Noir(x) \wedge Rond(x) \wedge Diametre(x)=25cm \wedge x \in Ballons \Rightarrow x \in BallonDeFootballs$

Une connaissance doit être représentée afin qu'elle puisse être exploitée le plus efficacement possible et qu'elle occupe un espace mémoire le plus faible possible. Pour faire une bonne représentation de connaissance, il faut prendre ces points :

- Extensibilité : l'utilisation de l'héritage. Par exemple :
  - Toutes les instances de Nourriture sont consommables.
  - Fruit est une sous-classe de Nourriture
  - Pomme est une sous-classe de Fruit
  - → Alors, on sait que toutes les pommes sont consommables.
- Les modèles envisagés doivent être souple pour permettre une modification du système ;
- Simplicité : les concepts retenus ne doivent pas être trop complexe ;
- Connaissance explicite : c'est un facteur important dans la recherche des erreurs et la justification du système.

Il est nécessaire aussi d'exprimer des relations entre les concepts à représenter (les nœuds représentent les concepts et les arcs représentent les relations) :



Les systèmes de représentation existants sont :

- Les réseaux sémantiques, les prototypes, les objets : proche de la pensée humaine (compréhension aisée) ;
- La logique, les règles de production : proche des mécanismes de raisonnement ;
- Les procédures : liées à notre conception actuelle des ordinateurs.

3.2.3 Raisonnement

Le but d'un programme d'IA est de réaliser des tâches cognitives (basée sur connaissance) tels que la manipulation d'objet, la compréhension de texte, l'analyse d'une scène. Le but ne sera atteint que si le programme ***raisonne*** sur des structures mémorisant d'une manière adéquate les informations manipulées. Donc, *raisonner c'est savoir quoi obtenir à partir de ce qui est déjà connu.*

Un programme raisonne lorsqu'il doit retrouver ou vérifier de nouvelles informations non explicitement représentées.

Exemple :	Question :
Tous les oiseaux volent	Un oiseau vole-t-il ?
Titi est un oiseau	Titi est-il un oiseau ?

Les réponses sont contenus dans la structure → ne demande pas de raisonnement. Par contre, la question « vole-t-il ? » demande de la part du système un raisonnement. Pour aboutir à une réponse, le système doit analyser et interpréter le lien hiérarchique entre Titi et les oiseaux.

3.2.4 Bases des connaissances

Une base de connaissance regroupe des connaissances spécifiques à un domaine spécialisé donné, sous une forme exploitable par un ordinateur. Elle sert à rassembler - de manière centralisée - l'expertise d'un domaine généralement formalisée de manière déclarative. Elle est composée d'une ***base de règles*** et d'une ***base de faits***.

Remarque : A ne pas confondre avec :

- une banque de connaissance (comme Wikipedia),
- une base de données (comme Oracle, Access, MySQL),
- une base de faits.

3.3 Les systèmes à base de règles

Parmi les premiers systèmes intelligents développés, on peut citer les systèmes à bases de règles. L'utilisation d'une base de règles est une méthode classique de représentation de connaissances (depuis 1943, voir les règles de production, cours théorie des langages). Chaque unité de connaissance est appelée règle. Un tel système à base de règles s'articule autour de deux éléments principaux : la base de connaissance et le moteur d'inférence.

3.3.1 La base de connaissance

La base de connaissance est constituée principalement par une base de règles et une base de faits. On inclut parfois deux éléments supplémentaires : les méta-règles et les agendas.

a) La base de règles

La base de règles contient l'ensemble des règles de production et des connaissances heuristiques déterminant la résolution du problème. Les règles sont de la forme : SI *Antécédents* ALORS *Conséquents*.

- *Antécédents* représente les prémisses, souvent une conjonction de conditions, parfois des négations ou des disjonctions.
- *Conséquents* représentent des actions à envisager. Souvent l'ajout de nouveaux faits. Toujours sous forme de conjonction.

Un exemple de base de règles basée sur la logique des propositions (un système de conseil en voyages) :

R1 : SI distance<2km ALORS aller_à_pied

R2 : SI ¬ distance<2km ET distance<300km ALORS prendre_le_train

R3 : SI ¬distance<300km ALORS prendre_l'avion

R4 : SI acheter_un_billet_d'avion ET avoir_le_téléphone ALORS téléphoner_à_l'agence

R5 : SI acheter_un_billet_d'avion ET ¬avoir_le_téléphone ALORS aller_à_l'agence

R6 : SI prendre_l'avion ALORS acheter_un_billet_d 'avion

R7 : SI durée>2jours ET être_fonctionnaire ALORS ¬prendre_l'avion

b) La base de faits

La base de faits contient, pour une situation donnée, l'ensemble des faits avérés ou à établir relatifs au problème considéré. C'est là que sont stockées les données symboliques et numériques. Par exemple, en logique propositionnelle, il s'agit des affirmations concernant la valeur de vérité des propositions (p est vrai, q est faux, …).

Exemple : F1 : ¬distance<300km F2 : avoir-le-téléphone

c) Les méta-règles

Il s'agit de règles destinées à guider le moteur d'inférence quant à la stratégie de résolution ou à la maintenance des bases de règles et de faits.

Exemple : Si une règle R a deux prémisses identiques Alors la règle R est anormale

Cette règle est une méta-règle, puisqu'elle parle de règles.

d) Les agendas

Il s'agit d'un moyen de gérer la notion de plans et de planification. L'agenda est une liste de tâches que le système peut effectuer avec les « raisons » qui poussent le système à penser qu'une tâche donnée est « intéressante ». Ainsi le système peut choisir la tâche la plus intéressante pour poursuivre la résolution. En résolution logique, nous pourrions chaque fois que nous générons une nouvelle clause, noter les raisons qui la rendent intéressante à sélectionner.

3.3.2 Le moteur d'inférence

Le moteur d'inférence est la partie créative du système. Il détient le ***contrôle du raisonnement***. À partir de règles et de faits, il génère de nouveaux faits (donc de nouvelles connaissances) afin de réaliser la résolution effective du problème. Le ***raisonnement*** suit la boucle suivante : À partir des faits appliquer les règles, déduire de nouveaux faits, appliquer les règles… jusqu'à ce qu'il n'y ait plus rien à déduire. Ceci représente le cycle du moteur d'inférence.

Exemple de construction d'un moteur d'inférence :

Algorithme MI1 :

```
Boucle sur les règles : soit R une règle
        Si les prémisses de R appartiennent à BF
                Alors ajouter les conclusions de R à BF
        FinSi
FinBoucle
```

Test de l'algorithme MI1 :

R3 et F1 → F3 : prendre_l'avion

R6 et F3 → F4 : acheter_un_billet_d'avion

Problème : ne déclenche pas R4

Algorithme MI2 :

```
Changement ← vrai
Tant que changement
        Changement ← faux
        Boucle sur les règles : soit R une règle
                Si les prémisses de R appartiennent à BF
                        Alors   ajouter les conclusions de R à BF
                                changement ← vrai
                FinSi
        FinBoucle
FinTQ
```

Test de l'algorithme MI2 :

R3 et F1 → F3, R6 et F3 → F4, R3 et F1 → F3

R4 et F4 et F2 → F5 : téléphoner_à_l'agence, R6 et F3 → F4, ...

Problème : les règles s'appliquent plusieurs fois

Il faut donc «!marquer!» les règles déclenchées pour ne plus les considérer dans la boucle

Algorithme MI3 :

```
Changement ← vrai
Tant que changement
        Changement ← faux
        Boucle sur les règles : soit R une règle
                Si R n'est pas marquée et si les prémisses de R appartiennent à BF
                        Alors   ajouter les conclusions de R à BF
                                changement ← vrai
                                marquer R
                FinSi
        FinBoucle
FinTQ
```

Test 1 de l'algorithme MI3 :

R3 et F1 → F3

R6 et F3 → F4

R4 et F4 et F2 → F5

Test 2 de l'algorithme MI3 avec une nouvelle base de faits :

Considérons une nouvelle base de faits :

F1 : ¬distance<300km

F2 : avoir_le_téléphone

F3 : durée>2jours

F4 : être_fonctionnaire

R3 et F1 → F5 : prendre_l'avion

R7 et F3 et F4 → F6 : ¬prendre_l'avion (Contradiction !!)

Algorithme MI4 :

```
Changement ← vrai
Tant que changement
        Changement ← faux
        Boucle sur les règles : soit R une règle
            Si R n'est pas marquée et si les prémisses de R appartiennent à BF
                Alors pour chaque conclusion C de R :
                        Si ¬C appartient à BF
                            Alors message d'erreur
                        Sinon Ajouter C à BF
                        FinSi
                        Changement ← vrai
                        Marquer R
            FinSi
        FinBoucle
FinTQ
```

a) Cycle d'un moteur d'inférence

La majorité des moteurs d'inférence utilise un cycle à trois étapes :

- ***Sélection (ou sélection et filtrage)*** : cette étape consiste à rechercher l'ensemble des règles d'inférence qui peuvent s'appliquer à un instant donné à la base de faits, et à noter les faits à utiliser. Ainsi, en résolution avec des clauses de Horn, si je dois résoudre la question p, le moteur va sélectionner toutes les clauses dont la tête est p.
- ***Résolution de conflit :*** le moteur va choisir parmi toutes les règles qu'il peut utiliser, celle (ou celles) qu'il va exécuter. Un exemple typique consiste à choisir la première règle disponible.
- ***Déclenchement*** : après avoir sélectionné la règle, le moteur d'inférence l'exécute et exécute toutes les actions associées (remise à jour de la base, …).

b) Algorithmes d'un moteur d'inférence

Les caractéristiques principales d'un moteur d'inférence sont liées aux techniques de résolution qu'il implante.

Chaînage :

On distingue souvent trois catégories, basées sur la manière dont les problèmes sont résolus :

Un moteur d'inférence peut fonctionner en chaînage avant ou en chaînage arrière.

- Un moteur d'inférence fonctionnant en ***chaînage avant*** (démarche déductive, Raisonnement progressif, à partir des faits) tente, à partir de la base de faits et des règles, de s'approcher des faits recherchés par le problème et de générer une réponse.

  Les inconvénients de cette méthode sont :

  1. déclenche toutes les règles ;
  2. demande beaucoup de faits initiaux (manque d'interactivité) ;
  3. explosion combinatoire possible (en largeur).

  Algorithme :

  Entrée : une base de faits F, une base de règles R
  Sortie : la base de faits F transformée
  DEBUT
  TANT QUE G n'est pas dans BF ET QU'IL existe dans BR une règle applicable
  FAIRE
  - Filtrage → E ensemble de conflits
  - Décision → R
  - BF ← BF ∪ concl(R)
  - BR ← BR - R

  FIN DU TANT QUE
  Si G appartient à BF ALORS G est établi
  SINON G n'est pas établi
  FIN

Exemple 1 :

REGLE R1

SI animal vole ET animal pond des œufs ALORS animal est un oiseau

REGLE R2

SI animal a des plumes ALORS animal est un oiseau

REGLE R3

SI animal est un oiseau ET animal a un long cou ET animal a de longues pattes

ALORS animal est une autruche

FAIT F1 : animal a des plumes

FAIT F2 : animal a un long cou

FAIT F3 : animal a de longues pattes

Application de l'algorithme :

- CYCLE 1

  R2 seule règle déclenchable. Donc R2 est choisie.

  Déclenchement de R2 : le fait F4 "animal est un oiseau" est ajouté à BF.

  BF = { F1 ; F2 ; F3 ; F4 }

- CYCLE 2

  R3 seule règle déclenchable. Donc R3 est choisie.

  Déclenchement de R3 : le fait F5 "animal est une autruche" est ajouté à BF.

  BF = { F1 ; F2 ; F3 ; F4 ; F5 }

- CYCLE 3

  Arrêt.

Exemple 2 :

R1 B et D et E → F

R2 D et G → A

R3 C et F → A

R4 C → D

R5 D → E

R6 A → H

R7 B → X

R8 X et C → A

Base de faits : B Vrai ; C Vrai ; H ?

Application de l'algorithme :

- Stratégie : choix de la première règle

  B, C ____ B, C, D ____ B, C, D, E ____ B, C, D, E, F ____B, C, D, E, F, A ____ B, C, D, E, F, A, H

- Stratégie : choix de la dernière règle

  B, C ____ B, C, X ____ B, C, X, A ____ B, C, X, A, H

- Un moteur d'inférence à ***chaînage arrière*** (démarche hypothético-déductive, Raisonnement régressif, à partir des buts) tente, par l'intermédiaire des règles, de « remonter » à des faits connus et d'invalider la question en montrant l'inconsistance de la nouvelle base de faits comprenant la négation de la question. Les inconvénients de cette méthode sont : l'algorithme est plus compliqué ; et il y a un risque de bouclage.

Algorithme :

```
entrée : une base de faits F, une base de règles R, un ensemble P de faits à prouver
sortie : la base de faits transformée F
DEBUT
TANT QUE P ≠ Ø
    extraire (en l'effaçant) de P un fait f
    SI le fait f est absent de F alors
        S = ensemble des règles concluant sur f
        SI S ≠ Ø alors
            a = choix d'une règle de S
            prouver tous les faits en prémisse de a
            SI a est applicable alors
                appliquer la règle a
            FinSi
        FinSi
    FinSi
FIN DU TANT QUE (attention au problème du bouclage, donc de non terminaison)
FIN
```

Exemple 1 :

REGLE R1

SI animal vole ET animal pond des œufs ALORS animal est un oiseau

REGLE R2

SI animal a des plumes ALORS animal est un oiseau

REGLE R3

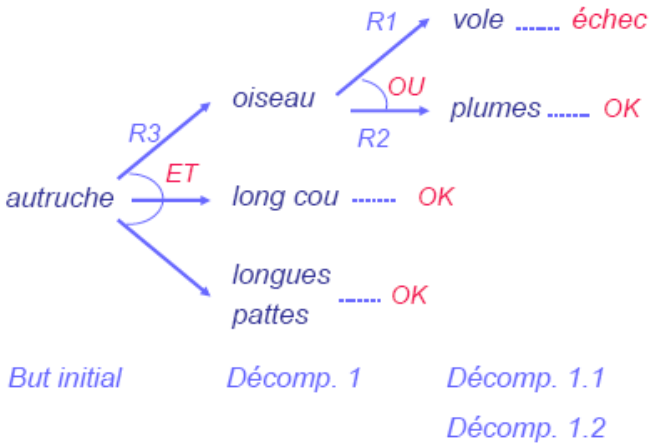
SI animal est un oiseau ET animal a un long cou ET animal a de longues pattes ALORS animal est une autruche

FAIT F1 : animal a des plumes

FAIT F2 : animal a un long cou

FAIT F3 : animal a de longues pattes

→ BUT : animal est une autruche



- Il est également possible d'utiliser le ***chaînage mixte*** qui tente de concilier les deux méthodes. L'utilisation du chaînage mixte demande la mise en place d'heuristiques particulières capables de décider face à une situation donnée quel est le type de stratégie le plus adapté pour poursuivre la résolution :

  - en avant pour *exploiter* les faits connus ou inférés ;
  - en arrière pour *focaliser* sur les buts et poser les bonnes questions.

Exemple de démarche mixte :

- Fonctionner en marche arrière ;
- Dès qu'un fait nouveau est connu, fonctionner en marche avant ;
- Poursuivre la marche arrière.

c) Caractéristiques d'un moteur d'inférence

Régime de contrôle :

Le régime de contrôle peut être ***irrévocable*** ou à ***tentative***. Si le système est à régime de contrôle irrévocable, le moteur d'inférence s'arrêtera dès que l'ensemble des règles sélectionnables sera vide. En revanche, dans un système à tentative, le moteur réalisera un retour-arrière (backtrack), et remettra en cause les règles déclenchées précédemment.

Monotonie :

Le moteur d'inférence peut être ***monotone*** ou ***non-monotone***. En régime monotone, le moteur ne fait qu'ajouter des faits à base de faits. En régime non-monotone, le moteur peut, en cas de retour-arrière par exemple, retrancher de la base de faits qui pourraient se révéler contradictoires.

3.3.3 Types des systèmes à base de règles

Les types des systèmes à base de règles dépendent du formalisme de représentation utilisé :

1. Système d'ordre 0 : valeurs vrais ou faux (logique propositionnelle).

   SI il y a douleur ET patient a plus de 40 ans ET ...

2. Système d'ordre 0+ : valeurs discrètes et continues.

   SI symptôme = douleur ET age >= 40 ET ...

3. Système d'ordre 1 : instanciation de variables (logique du premier ordre).

   SI ?x a une douleur ET ?x à l'âge ?n ET ?n > 40 ET ...

3.3.4 Remarque

Les problèmes rencontrés dans le développement des systèmes à base de règles sont l'acquisition des connaissances et la construction de la base de connaissances. Pour résoudre ce problème, on fait appel à un spécialiste dans le domaine d'application du système appelé ***expert*** (ou ingénieur de connaissance). D'où la naissance des « ***systèmes experts*** ».

Dans un système expert, l'ingénieur de connaissance (appelé aussi **cogniticien**) doit traduire l'expertise informelle en un langage formel adapté au mode du raisonnement du système c'est-à-dire en règles. Plusieurs points doivent être soulevés concernant l'acquisition des connaissances :

1. La compétence humaine n'est pas souvent accessible via la conscience. Avec l'expérience acquise, la compétence et la performance d'un expert s'installe et opère dans l'inconscient. Par conséquence, il est difficile aux experts d'expliciter son savoir-faire.

2. L’expertise humaine prend souvent la forme du savoir comment plus que la forme du savoir quoi.
3. L’expertise humaine représente un modèle individuel ou un modèle de communauté. Ces modèles sont soumis aux conventions et aux procédés sociaux.
4. L’expertise change et peut subir des reformulations radicales.

A cause de la complexité et de l’ambiguïté posées par le problème de l’acquisition de connaissances, l’ingénieur de connaissances doit avoir un modèle conceptuel lui permettant de faire la liaison entre l’expertise humaine et le langage de programmation. Donc un tel système expert doit être capable de dialoguer avec l’expert et l’utilisateur afin d’améliorer l’acquisition et la représentation des connaissances. L’expertise est nécessaire aussi dans le choix de l’algorithme à utiliser dans la machine d’inférence.

3.4 Les systèmes experts

3.4.1 Définition d’un expert

Un expert est quelqu'un qui connaît un domaine et qui est plus ou moins capable de transmettre ce qu'il sait : ce n'est par exemple pas le cas d'un enfant par rapport à sa langue maternelle.

L'expert est :

- au début, *enchanté* : explicitation de sa connaissance (il apprend lui-même de l'analyse de son propre savoir ainsi mis à plat) ;
- ensuite, *inquiet* : la compétence du système expert semble réelle ;
- enfin, *rassuré* : il existe toujours des problèmes que le système expert ne sait pas résoudre, malgré les multiples révisions de la base de connaissance.

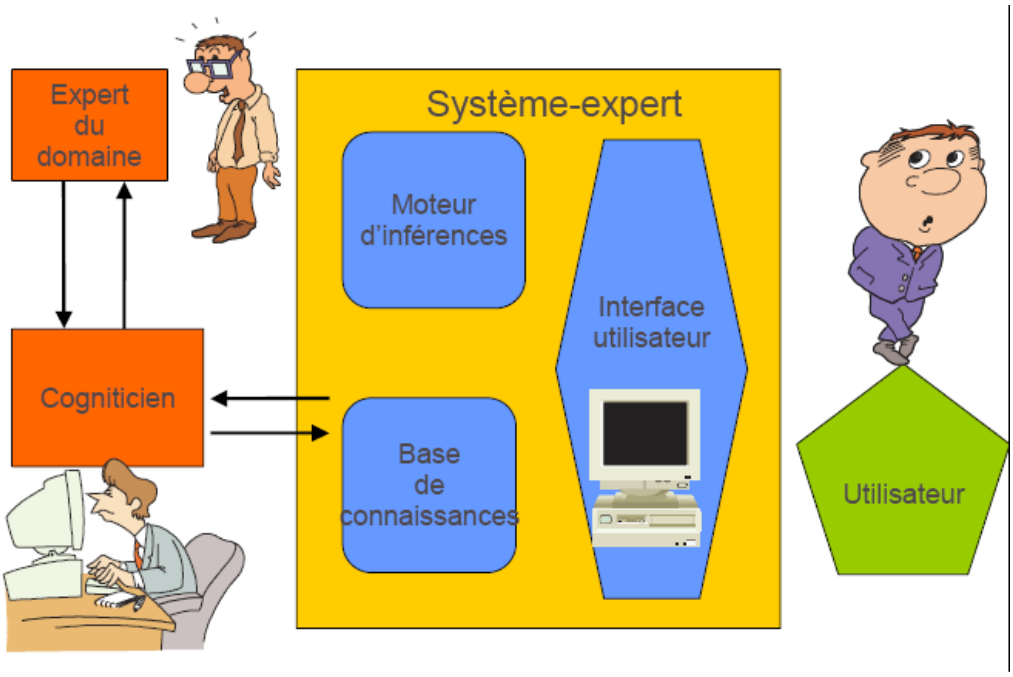
Les experts d'un domaine ont pour caractéristiques :

- d'être rares, donc peu disponibles
- d'être compétents (si possible les meilleurs)
- d'être souvent incapables d'expliquer leur démarche
- d'être mortels

3.4.2 Définition d’un système expert

Le terme « Système Expert (SE) » a fait son apparition au début des années 70. Les systèmes experts sont le résultat de l’évolution de l’IA. Edward Feigenbaum (informaticien américain qui travaille dans le champ de l'IA) définit les systèmes experts comme des « programmes conçus pour raisonner à propos de tâches dont on pense qu’elles requièrent une expertise humaine considérable. »

Les systèmes experts sont des outils de l'intelligence artificielle, c'est-à-dire qu'on ne les utilise que lorsqu'aucune méthode algorithmique exacte n'est disponible ou praticable.
D'une manière générale, un système expert est un outil capable de reproduire les mécanismes cognitifs d'un expert, dans un domaine particulier. Il s'agit de l'une des voies tentant d'aboutir à l'intelligence artificielle. Plus précisément, un système expert est un logiciel capable de répondre à des questions, en effectuant un raisonnement à partir de faits et de règles connus. Un tel système n'est concevable que pour les domaines dans lesquels il existe des experts humains.



3.4.3 Objectif

Un système expert a le but de faciliter le transfert d'expertise :

- de l'expert vers l'entreprise, dans le cas de départ
- de l'expert vers un utilisateur averti, dans le cas d'un expert indisponible
- de l'expert vers d'autres experts, dans le cas d'une coopération.

3.4.4 Domaines d'applications

Les systèmes experts sont généralement conçus pour résoudre des problèmes de classification ou de décision (outil d'aide à la décision) dans des domaines comme la médecine, le droit, la chimie, l'éducation etc.

Les catégories de problèmes abordés par les systèmes experts sont :

- L'interprétation ou la construction d'une description abstraite à partir de données.
- La prédiction des conséquences à partir de situations données.
- Le diagnostic d'une défaillance à partir d'un ensemble d'observations.
- La conception d'une configuration de composants à partir d'un ensemble de contraintes.
- La planification d'une séquence d'actions pour l'accomplissement d'un ensemble de buts à partir de certaines conditions de départ et en présence de certaines contraintes.
- La réparation d'un dysfonctionnement.
- le contrôle du comportement d'un environnement complexe.

3.4.5 Problèmes adaptés aux systèmes experts

Les chercheurs ont défini un ensemble informel de critères pour déterminer si un problème est adapté ou non à être résolu par la technologie système expert :

1. Le besoin d'une solution doit justifier le coût et l'effort de la construction d'un système expert.
2. L'expertise humaine n'est pas valable dans toutes les situations dont on a besoin.
3. Le problème peut être résolu en utilisant une technique de raisonnement symbolique.
4. Le domaine est bien structuré.
5. Le problème ne peut pas être résolu en utilisant des méthodes traditionnelles de calcul.
6. La coopération entre experts de domaine existe.
7. Le problème est de taille considérable.

3.4.6 Recommandations

Pour écrire un « bon » système expert :

- il faut séparer la base de connaissances du moteur d'inférences ;
- il faut utiliser des représentations les plus uniforme possible ;
- il faut garder le moteur d'inférences le plus simple possible ;
- il faut utiliser la redondance de l'information.

3.4.7 Processus d'ingénierie de connaissance

Les personnes concernées par le développement d'un système expert sont :

- l'ingénieur de connaissance (cogniticien) qui est un expert en langage IA. Son rôle est de trouver les outils et les logiciels nécessaire pour l'accomplissement du projet, d'aider l'expert du domaine à expliciter sa connaissance et d'implanter cette connaissance dans la base de connaissances.

- l'expert du domaine qui fournit les connaissances nécessaires liées au problème.
- l'utilisateur final dont le rôle est de spécifier l'application et de déterminer les contraintes de la conception.

En général, le travail commence par une interview entre l'ingénieur de connaissance et l'expert du domaine. L'ingénieur essaie de comprendre le domaine, d'observer l'expert pendant son travail. Une fois l'expert ait obtenu des informations complètes et précises sur le domaine ainsi que sur la résolution du problème, il pourrait entamer la tâche de la conception du système.

Il choisit la façon de la représentation des connaissances, Il détermine le type du raisonnement et la stratégie utilisée (chaînage avant ou arrière, en profondeur ou en largeur). Il conçoit de même l'interface utilisateur. Le prototype obtenu doit être capable de résoudre correctement un problème typique. Ce prototype doit être testé et affiné par l'ingénieur et l'expert du domaine en même temps.

Le prototype peut être complété au fur et à mesure en ajoutant des nouveaux éléments dans la base de connaissance. Souvent, à la fin de ce travail progressif, l'ingénieur serait amené à refaire une version plus propre qui réécrit la connaissance d'une façon plus sommaire.

3.4.8 Historique

Les années 70 et 80 virent un véritable engouement pour les systèmes experts. Le premier système expert est DENDRAL apparu en 1970. Il permettait d'identifier une structure chimique à partir de résultats de mesures chimiques, physiques et spectrométriques. Il contient plusieurs milliers de connaissances élémentaires, données sous forme de règles granulaires. Le principe de ce système est de déduire à partir des résultats de mesures toutes les informations possibles ; interroger l'usager si des informations sont manquantes ; faire une synthèse et conclure. Ce système a des performances remarquables, et est vendu avec le spectrographe par le fabricant. Les inconvénients sont : le système est programmé de manière classique, d'où il est difficilement maintenable ; et il n'y a aucune possibilité d'explication.

Un deuxième SE appelé MYCIN a été développé en 1974, dans le domaine de médecine, afin d'aider des médecins à diagnostiquer et soigner des maladies infectieuses du sang. Il aide à la détermination de la meilleure thérapie lors d'une infection bactérienne en 4 étapes : déterminer l'importance de l'infection ; déterminer l'organisme responsable ; identifier des médicaments potentiellement utilisables ; choisir le meilleur traitement pour le cas considéré.

La base de connaissance contient 200 règles sur les maladies infectieuses du sang et 300 règles sur la méningite.

Voici un exemple de règle :

SI le site de la culture est le sang
ET l'identité de l'organisme n'est pas connu avec certitude
ET la coloration de l'organisme est GRAM négatif
ET la morphologie de l'organisme est batonnet
ET le patient a été sérieusement brûlé
ALORS il y a des chances (0.4) que l'organisme soit pseudomonas

3.5 Critique des systèmes experts

L'étude des problèmes de la représentation de l'expertise humaine amène, d'après Randall Davis[1], les enseignements suivants :

- le savoir est la base de l'ensemble du système ;
- le savoir est souvent inexact, voire incomplet ;
- le savoir est souvent mal spécifié ;
- un amateur devient un expert progressivement.

Donc, il est clair que les systèmes experts doivent être flexibles et transparents, car une grande partie de la vie d'un système expert sera consacrée à de permanents changements et de fréquences remises à jour et améliorations. Davis estime qu'un système expert connaît un cycle de développement en sept étapes et que sa croissance se poursuivra pendant chacune de ses étapes :

1. conception du système ;
2. développement du système ;
3. évaluation formelle de performances ;
4. évaluation formelle d'acceptabilité ;
5. utilisation dans un environnement-prototype ;
6. développement de plans de maintenance ;
7. mise sur le marché

Davis remarque que rares sont les SE qui dépassent l'étape (2) de leur évolution. La plupart des SE sont conçus pour tester des idées sur la représentation des connaissances, le contrôle, l'acquisition, etc. En quand les concepteurs disent que le système fonctionne, ils veulent généralement dire l'idée fonctionne.

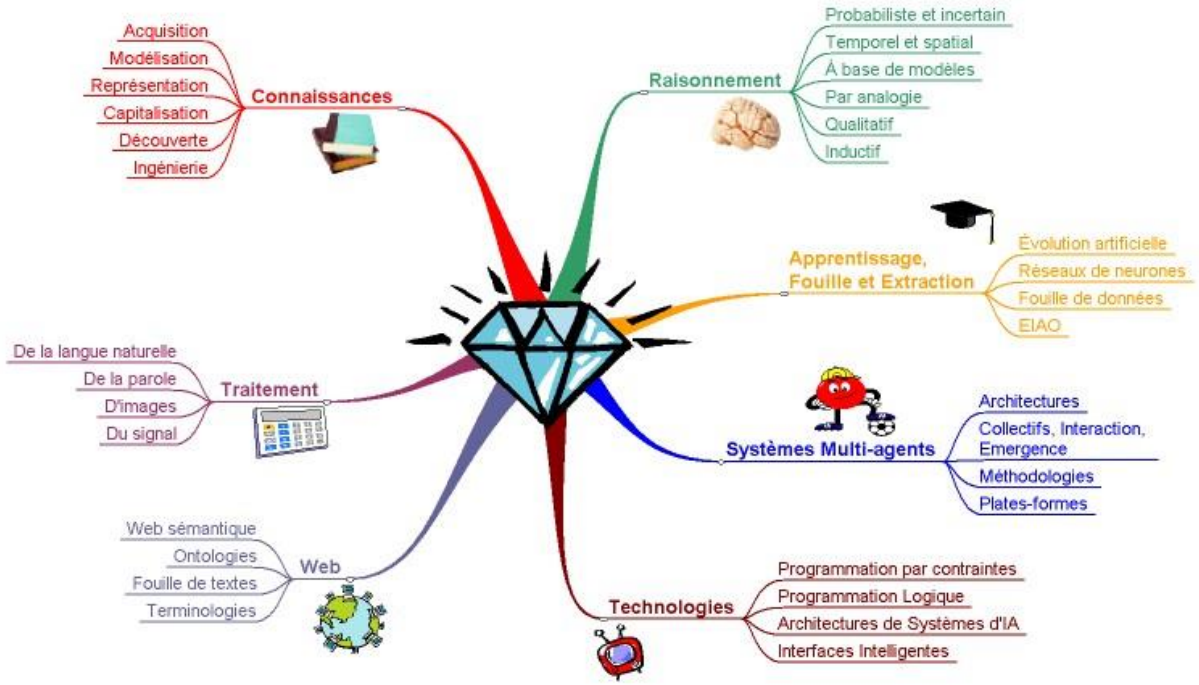
[1] Davids R. *Expert Systems: Where are we? And where do we go from here?* The AI magazine, Printemps, 1982.

3.6 Conclusion : Naissance des systèmes à base de connaissance

Il est probable que la meilleure utilisation des systèmes experts est de remplacer ou assister l'homme pour effectuer des tâches peu complexes et répétitives. Malheureusement, Vu que les être humains experts sont rares ou peu disponible, le développement d'un SE nécessite un coût énorme, même pour des domaines peu utiles. Ce développement devient très difficile en cas où l'expertise humaine n'est pas accessible. De plus, si le monde évolue un peu, tout le système doit changer. La solution était d'automatiser le processus d'acquisition et d'extraction de connaissances ; et d'utiliser un algorithme d'apprentissage pour apprendre de nouvelles connaissances.

Ceci a comme résultat la naissance des systèmes à base de connaissances (Knowledge base system) dont le principal objectif est comment arriver à des systèmes experts sans l'aide d'un être humain expert. Ce système doit avoir la possibilité d'extraire les connaissances automatiquement, sans l'intervention d'un expert, à partir des données et d'apprendre de nouvelles connaissances grâce aux dialogues entre le système et l'utilisateur.

La figure suivante les domaines actuels de recherche en intelligence artificielle (d'après le site de l'Association Française d'Intelligence Française (AFIA)).



Chapitre 4 :

Résolution de problèmes en IA

4.1 Introduction

Un ***problème*** est défini par la situation devant laquelle on ne voit pas directement les étapes permettant de dépasser la situation et satisfaire un besoin.

Résoudre un problème revient à chercher un chemin qui mène d'une situation initiale à une autre finale pour atteindre un but.

Parmi les problèmes que l'on tente de faire résoudre par une machine sont :

- les jeux ;
- vision ;
- planification ;
- la preuve de théorèmes ;

Au départ, nous avons pensé que les ordinateurs sont capables de résoudre ces problèmes plus vite que l'homme vue leurs capacités de mémorisation et la rapidité de traitement de l'information. Mais puisque le nombre de chemins à explorer pour arriver à une solution est gigantesque, la structure matérielle de la machine reste insuffisante. Dans la plupart des cas, le joueur se base sur des règles et des capacités stratégiques et non sur la puissance de calcul. Il s'agit de doter la machine de telles capacités.

→ Donc, *la résolution de problème nécessite des techniques de l'IA fondées sur des démarches de nature intelligentes*.

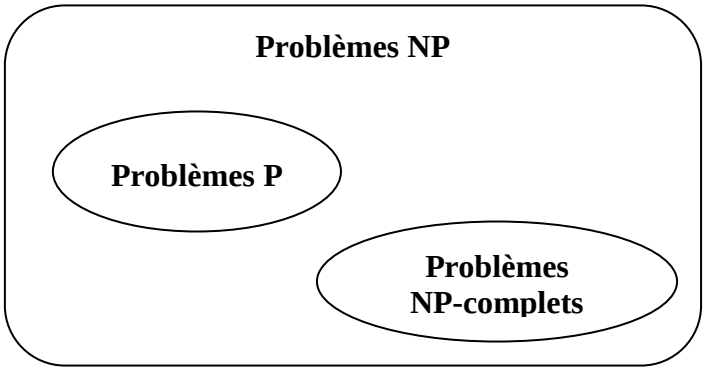
La démarche d'IA consiste que l'algorithme doit être "neutre" sur le domaine concerné. De plus, tout ce qui concerne les connaissances de description du problème et de sa résolution doit être clairement séparé de l'algorithme.

4.2 Analyse de la complexité et catégories de problèmes

L'analyse des algorithmes et la notation *O()* permettent de parler de l'efficacité d'un algorithme spécifique. En revanche, on ne peut pas savoir s'il existe un algorithme qui serait meilleur pour un problème donné. L'analyse de la complexité traite des problèmes plutôt que

des algorithmes. Indépendamment de l'algorithme employé, on peut diviser les problèmes entre ceux qui peuvent être résolus en un temps polynomial et ceux qui ne peuvent pas l'être.

- La classe des problèmes **P**olynomiaux – ceux qui peuvent être résolus en un temps en $O(n^k)$ pour un k donné – est appelée **classe P**. Ces problèmes sont dits « faciles » si le temps d'exécution est de l'ordre de $O(\log n)$ ou $O(n)$. Mais elle contient également des problèmes dont les temps d'exécution sont en $O(n^{1000})$.
- La classe des problèmes **N**on déterministes **P**olynomiaux est appelée classe **NP** (différent de déterministe non-polynomiaux). Ces problèmes ne peuvent pas être résolus dans un temps d'exécution raisonnable. Un problème appartient à cette classe s'il existe un algorithme pour *proposer* une solution puis *vérifier* si elle est correcte en un temps polynomial. Un problème NP devient un problème P si l'on dispose d'un nombre très important de processus.
- La classe des problèmes **NP-complets**. Le terme « complet » doit être pris dans le sens de « plus extrême » et fait donc référence aux problèmes les plus difficiles de la classe NP.



4.3 Exemples de problèmes

4.3.1 Problème de passage de rivière

Un fermier, un loup, une chèvre et un chou se trouvent sur la rive gauche d'une rivière. Un bateau, mené seulement par le fermier, peut le transporter, lui et éventuellement l'un des trois autres protagonistes, d'une rive à l'autre. En l'absence du fermier, le loup mangera la chèvre et la chèvre mangera le chou. Comment faire passer tout le monde sans encombre sur la rive droite de la rivière ?

4.3.2 Problème de 12 billets

On possède 12 billes. Une de ces 12 billes est plus légère ou plus lourde. Comment trouver ce billet si on dispose d'une balance mais on n'a droit qu'à 3 mesures seulement ?

4.3.3 Problème de taquin

Le taquin est un jeu solitaire en forme de damier. Comment à partir d'une configuration initiale atteindre une autre finale ?

1	5	2
6	4	7
3	□	8

4.3.4 Problème de voyageur de commerce

Un voyageur de commerce part d'une ville et veut se rendre dans un certain nombre de villes sans passer deux fois par la même ville et doit revenir à la ville de départ en fin de voyage. On connaît les distances entre les villes. Quelle est le trajet à faire en minimisant la distance parcourue ?

4.3.5 Problème des tours de Hanoï

Le problème consiste à déplacer des disques de diamètres différents d'une tour de « départ » à une tour d'« arrivée » en passant par une tour « intermédiaire » et ceci en un minimum de coups, tout en respectant les règles suivantes :

- on ne peut déplacer plus d'un disque à la fois ;
- on ne peut placer un disque que sur un autre disque plus grand que lui ou sur un emplacement vide.

4.3.6 Problème des récipients

Nous disposons de deux récipients sans aucune marque de graduation : un de 4 litres (X) et un de 3 litres (Y). Comment remplir le récipient de 4 litres avec exactement 2 litres d'eau ?

4.4 Méthodes de résolution de problèmes en IA

Il existe différents méthodes de résolution de problèmes en IA. Les techniques utilisées dépendent de certains facteurs :

- **Optimisation** : trouver une solution facile et rapide sans se poser des questions sur son optimisation (voyageur de commerce) ;
- **Réversibilité** : c'est le fait d'annuler une action décidée en cours de recherche de la solution une fois qu'elle a été exécuté (taquin) ;

- **Prévision** : c'est prévoir exactement l'effet d'une action sur l'état courant du problème ;
- **Décomposition** : c'est décomposer le problème en sous problèmes plus facile à résoudre ;
- **Connaissances** : c'est toutes les connaissances minimales requises pour résoudre le problème.

4.5 Démarche de recherche d'une solution

La résolution de nombreux problèmes peut en effet être décrite comme la séquence d'états permettant de mener de l'état initial (ce que l'on sait au début du problème) jusqu'à l'état final (le but à atteindre). Dans ces conditions, à partir de l'état initial, il "suffit" d'explorer de manière plus ou moins astucieuse les états possibles à atteindre depuis cet état initial. Si la solution est dans l'espace de ces états alors on a trouvé.

Pour modéliser un problème de cette façon, il faut donc :

- décrire un état de départ
- définir les opérateurs permettant de passer d'un état à d'autres (fonction permettant de retourner l'ensemble des états atteignables par une action particulière)
- pouvoir construire l'espace des états (l'ensemble des états atteignables depuis l'état de départ)
- disposer d'un test permettant de savoir que la solution est trouvée (si on a trouvé l'état but final)
- construire un chemin de l'état de départ à l'état final (une séquence d'états dans l'espace des états)
- disposer d'une fonction de coût sur le chemin : cette fonction associe un coût au chemin, coût calculé comme la somme des coûts individuels des actions le long du chemin.

Dans ces conditions, qu'est-ce que faire une recherche de solution ?

La recherche de solution consiste à faire un examen systématique des états à trouver en partant de l'état de départ jusqu'à trouver l'état but. L'ensemble des états possibles connectés par les opérateurs permettant de passer des uns aux autres représente l'espace d'états.

Le résultat de l'algorithme de recherche est donc la solution, c'est-à-dire un chemin de l'état initial à l'état qui satisfait les conditions du test d'atteinte du but.

4.6 Graphe de représentation pour la résolution d'un problème

L'espace des états successifs d'un problème à résoudre peut être réduit sous forme d'un graphe (ou dans d'autres cas par un arbre).

Représentation de graphes en sous problèmes

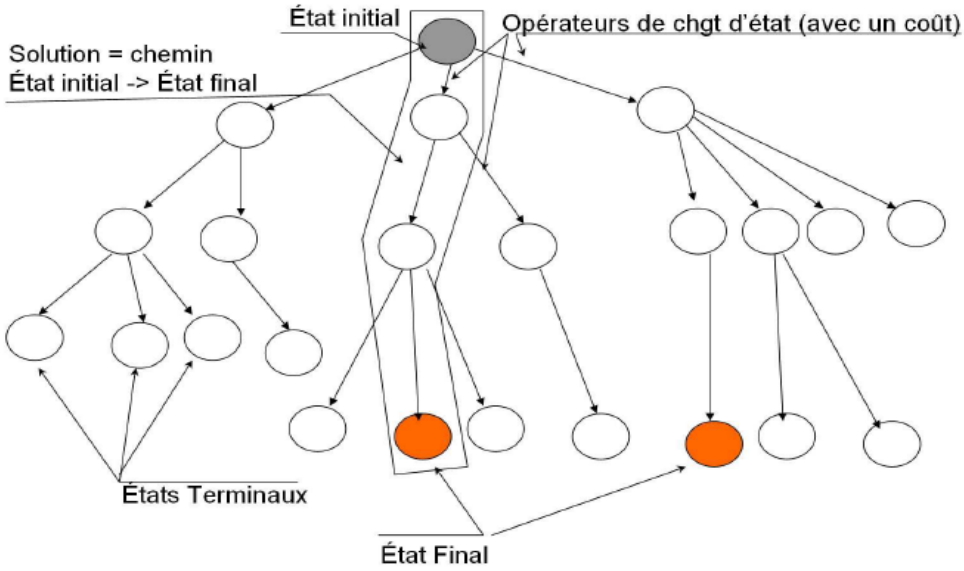
- Un état est un problème (ou un sous-problème à résoudre).
- L'état initial est le problème non décomposé
- Les états terminaux sont des sous-problèmes triviaux (solution immédiate)
- L'opérateur est celui de décomposition de problème
- Il y a deux types de nœuds :
- les nœuds "OU" associés aux problèmes
- les nœuds "ET" associés aux règles de décomposition

Certains problèmes peuvent être représentés par la technique de réduction de problème, c'est-à-dire que le problème peut être considéré comme une conjonction de plusieurs sous-problèmes indépendants les uns des autres, que l'on peut résoudre séparément.

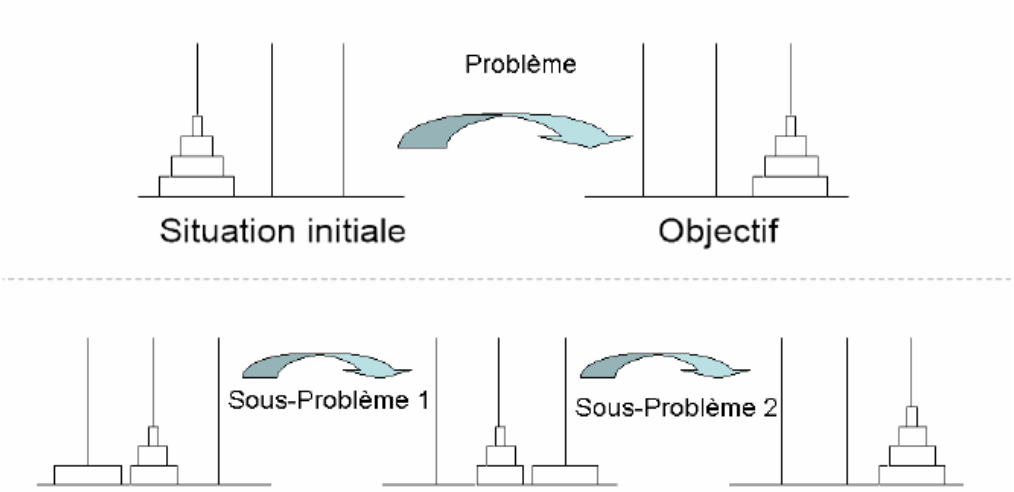
D'autre part, il peut se présenter dans la résolution d'un problème un ensemble de possibilités indépendantes les unes des autres (donc exclusives) telles que la résolution d'une seule de ces possibilités suffit à résoudre le problème.

On peut représenter ces deux possibilités par des arcs différents dans le graphe représentant l'espace de recherche :

- Les arcs qui représentent différentes possibilités dans la résolution d'un problème associés au nœud dont ils découlent sont des arcs OU ;
- Les arcs qui représentent différents sous-problèmes composant la résolution d'un problème associés au nœud dont ils découlent sont des arcs ET.



Exemple de description du problème des tours de Hanoi en sous-problèmes :



4.7 Stratégies de recherche de solution

Deux types de stratégies ont été développés :

- ***L'exploration non informée*** : ne dispose pas d'information sur les états. Elle peut générer des successeurs et distinguer un état final d'un état non final.
- ***L'exploration informée*** : elle est capable de déterminer si un état non final est plus prometteur qu'un autre.

Les stratégies d'exploration se distinguent par l'ordre choisi pour développer les nœuds. Les critères de choix d'une stratégie de recherche sont essentiellement :

- La **complétude** : lorsque la méthode garantit de trouver une solution si elle existe ;
- L'**optimalité** : ce critère concerne l'optimalité de la solution trouvée – par rapport à celles qui existent – en se basant sur une fonction de coût. Est-ce que la stratégie trouve la solution optimale ;
- La **complexité** : Elle est exprimée en utilisant les quantités suivantes:
    - *b* : le facteur de branchement : le nombre maximum de successeurs à un nœud ;
    - *d* : la profondeur du nœud but le moins profond (profondeur de la solution) ;
    - *m* : la longueur maximale d'un chemin dans l'espace d'états (profondeur maximum de l'arbre de recherche).
    - La **complexité en temps** : le nombre de nœuds générés pendant la recherche ;
    - La **complexité en espace** : le nombre maximum de nœud en mémoire ;

4.7.1 Stratégies d'exploration non-informé (Aveugle)

d) Recherche en largeur d'abord (breadth-first search)

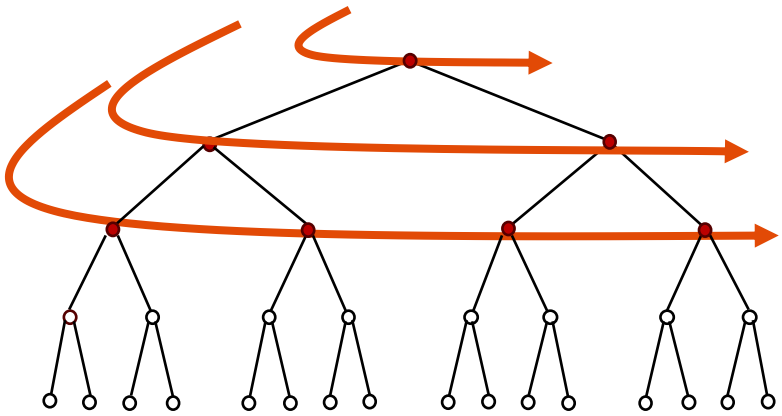
Cette stratégie donne une plus grande priorité aux sommets les moins profonds du graphe de recherche en explorant les sommets de même profondeur. En d'autres termes, il faut développer tous les nœuds au niveau i avant de développer les nœuds au niveau i+1. L'ensemble OUVERT aura une structure de file d'attente.

Avantages :

- Trouver une solution si elle existe ;
- Si le graphe possède un nombre fini de branche alors la recherche est dite algorithmique.

Inconvénients :

- Gourmande en espace mémoire ;
- Nombre de nœuds à mémoriser par niveau crois exponentiellement ;
- Très peu adapter à une solution admettant plusieurs chemins de solution et la solution se trouve très loin.



L'algorithme de recherche peut être décrit comme suit :

OUVERT : liste des nœuds apparus ou mis en évidence

FERME : liste des nœuds développés (pour lesquels nous avons mis en évidence les descendants éventuels)

Début
Mettre le nœud initial dans OUVERT (initialement vide)
Tant que OUVERT ≠ ∅ et BUT non atteint faire
Début
    Déterminer si la tête de OUVERT est le but

2.a. Si oui, rien faire

2.b Sinon, enlever la tête de OUVERT

La mettre dans FERME

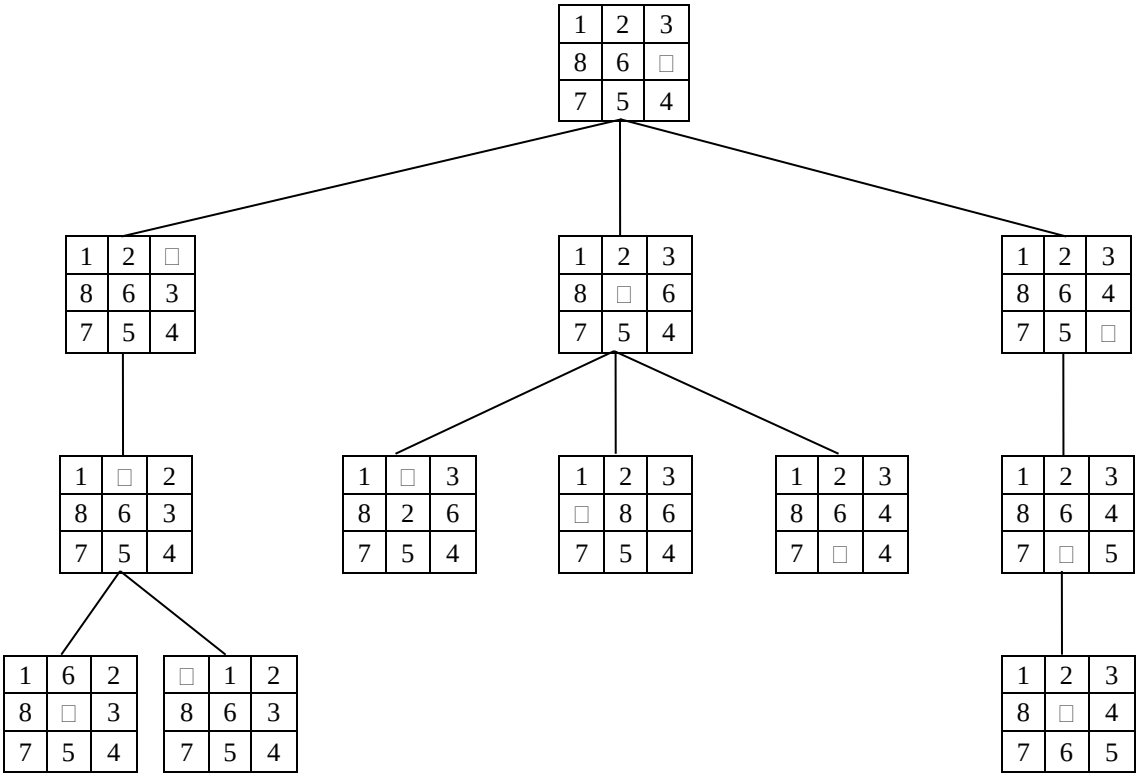
Ajouter ses descendants en QUEUE de OUVERT

Fin Tant que

Si le BUT est atteint alors succès sinon échec

Fin

Exemple : Jeu de taquin



e) La recherche coût uniforme (uniform-cost search)

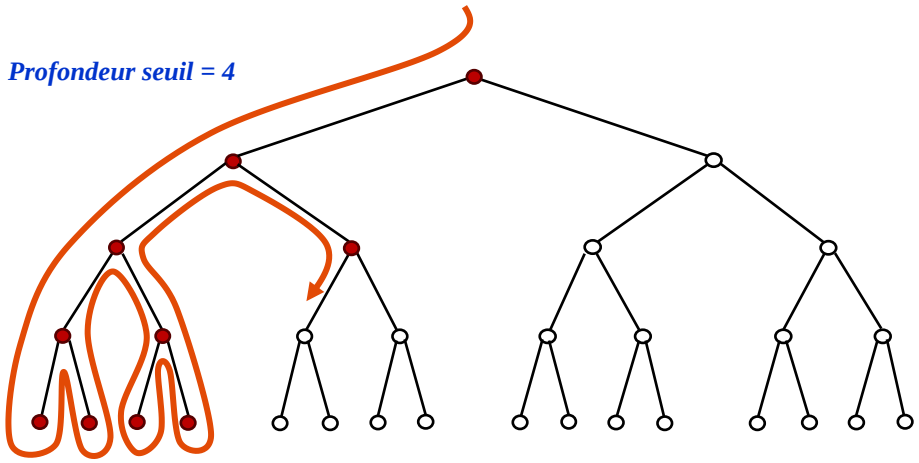
Dans la recherche avec coût uniforme, la priorité est donnée aux nœuds ayant le coût le plus bas. La structure de données utilisée est une file triée selon le coût.

Cette recherche est équivalente à la méthode largeur d'abord si le coût des actions est toujours le même.

f) La recherche en profondeur d'abord (depth-first search)

Dans la recherche en profondeur, la priorité est donnée aux sommets des niveaux les plus profonds du graphe de recherche. Chaque sommet choisi pour être exploré voit tous ses successeurs générés avant qu'un autre sommet ne soit exploré. Dans la liste OUVERTS le

sommet le plus profond est celui le plus récemment généré. Cet ensemble aura donc une structure de pile.



L'algorithme peut s'écrire de la manière suivante :

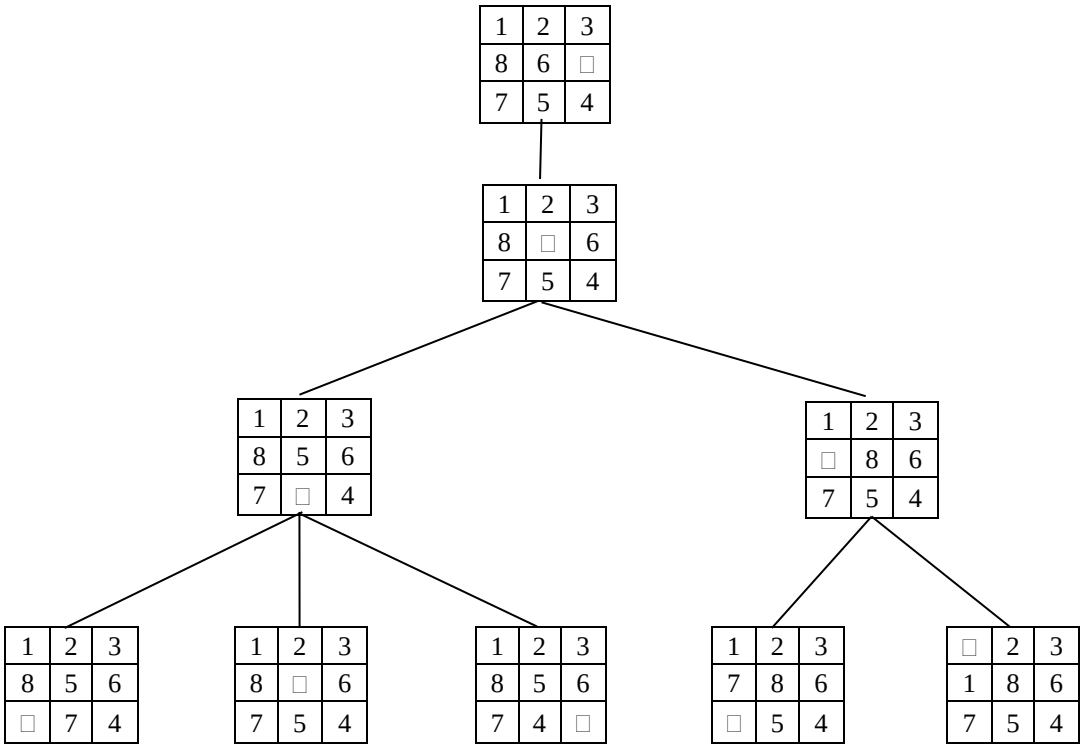
Début
Mettre dans OUVERT le nœud de départ
Tant que OUVERT ≠ ∅ et le nœud but non atteint faire
Début
    Déterminer si le premier nœud est un nœud but
    2.a. Si oui, rien faire
    2.b Sinon, enlever le premier nœud de ouvert et le mettre dans FERME
Ajouter en tête de OUVERT les descendants du nœud qu'on vient d'enlever
Fin Tant que
Si le BUT est atteint alors succès sinon échec
Fin

g) La recherche en profondeur limitée (depth-limited search)

Les graphes de recherche des problèmes qu'on cherche à résoudre peuvent être de très grande taille (voire infinie). Une recherche en profondeur peut se révéler dangereuse. L'algorithme doit tenir compte de deux possibilités pour le retour arrière :

- la limite de profondeur est dépassée ;
- un sommet est reconnu comme une impasse.

Exemple : Jeu de taquin avec une profondeur limitée = 3



h) La recherche en profondeur itérative (Iterative deepening search)

Le problème avec la recherche en profondeur limitée est comment fixer la bonne valeur de L. Cet algorithme consiste à répéter la recherche en profondeur pour toute les valeurs possibles de L (0, 1, 2, …) jusqu'à trouver une solution. Donc, la méthode combine les avantages de la recherche en largeur et en profondeur. C'est la meilleure méthode si l'espace de recherche est grand et si la profondeur de la solution est inconnue.

4.7.2 Comparaison des algorithmes de recherches aveugles

Critères	Largeur d'abord	Coût uniforme	Profondeur d'abord	Profondeur limitée	Profondeur itérative
Complétude	oui (si b est fini)	oui	- non si la profondeur est infinie, s'il y a des cycles. - oui si espaces finis acycliques.	oui, si $l \geq d$	Oui
Temps	$1 + b + b2 + b3 + \ldots + bd + (b^{d+1} - b) = O(b^{d+1})$	$O(b^d)$	$O(b^m)$ - Très mauvais si m est plus grand que d	$O(b^l)$	$O(b^d)$
Espace	$O(b^{d+1})$ (Garde tous les nœuds en mémoire)	$O(b^d)$	O(bm) linéaire	O(b*l) linéaire	O(b*d)
Optimal	- non en général - oui si le coût des actions est le même pour toutes les actions	oui	Non	Non	presque, si le coût de chaque action est de 1

4.7.3 Stratégie d'exploration informé (Heuristique)

a) Principe

Les algorithmes de recherche aveugle n'exploitent aucune information concernant la structure de l'arbre de recherche ou la présence potentielle de nœuds-solution pour optimiser la recherche. C'est une recherche "simple" à travers l'espace jusqu'à trouver une solution. D'où, la plupart des problèmes réels sont susceptibles de provoquer une explosion combinatoire du nombre d'états possibles.

Les méthodes heuristiques viennent pour surmonter ces problèmes qui sont liés aux :

- ne tenir pas compte des spécificités des problèmes ;
- loin de la méthode de recherche pratiquée chez l'homme ;
- la solution trouvée n'est pas forcément optimale.

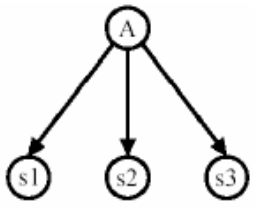
L'heuristique introduit des stratégies de contrôle de la recherche. Une heuristique est un critère ou une méthode permettant de déterminer parmi plusieurs lignes de conduite celle qui promet d'être la plus efficace pour atteindre un but. Une information heuristique est une règle ou une méthode qui presque toujours améliore le processus de recherche.

Il s'agit d'utiliser une fonction d'évaluation (fonction heuristique) pour chiffrer la validité d'un nœud par rapport à l'autre. À chaque étape, les nœuds sont réordonnés au lieu d'être mis dans une pile ou une file. On parle dans ce cas d'une recherche ordonnée qui revient à choisir de développer le nœud ayant les meilleures chances de mener au but.

Une fonction heuristique $H : E \rightarrow R$ fait correspondre à un état $s \in E$ (espace d'états) un nombre $h(s) \in R$ qui est généralement une estimation du rapport coût/bénéfice qu'il y a à étendre le chemin courant en passant par s.

Exemple :

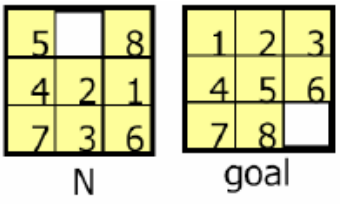
- h(solution)=0
- le nœud A a 3 successeurs pou lesquels h(s1)=0.8 ; h(s2)=2.0 ; h(s3)=1.6



Donc, la poursuite de la recherche par s1 est heuristiquement la meilleure.

<u>Exemples de fonctions heuristiques :</u>

- h(N) = nombre de plaquettes mal placées = 6
- h(N) = somme des distances (Manhattan) de chaque plaquette à sa position finale
  = 3+1+3+0+2+1+0+3 = 13



b) Recherche meilleur-d'abord

Ce type de recherche est une combinaison entre recherche en profondeur (Av. solution trouvée sans avoir besoin de calculer tous les nœuds) et en largeur (Av. ne risque pas de rester pris dans une « impasse »). L'algorithme recherche meilleur-d'abord permet d'explorer les nœuds dans l'ordre (croissant/décroissant) de leurs valeurs heuristiques. Deux cas particuliers existent : l'algorithme de recherche avare et l'algorithme A*.

➢ *Recherche gloutonne (Greedy Search)*

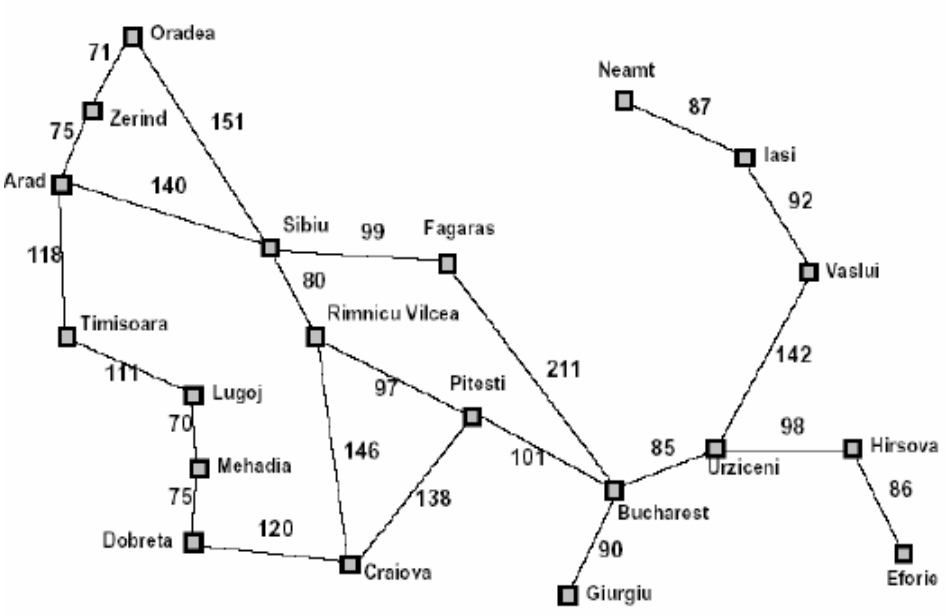
C'est la stratégie la plus simple pour la recherche meilleur-d'abord. La fonction heuristique utilisée est h(n)= estimation du coût du nœud n au but. L'objectif est donc de minimiser le coût estimé pour atteindre le but. Le nœud qui semble être le plus proche du but sera étendu en priorité. Les Fonctions heuristiques classiques utilisées sont :

- distance à vol d'oiseau ;
- distance "Manhattan" : déplacements limités aux directions verticales et horizontales.

Quelques méthodes de sélection des règles :

- La première (numérique)
- La plus fréquente
- Les plus récentes
- La plus prometteuse (selon le contexte)
- La plus fiable
- La moins coûteuse
- Celles avec des faits importants.

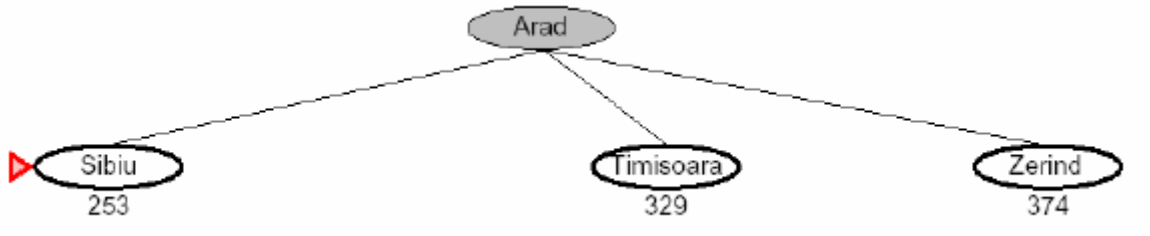
Exemple : voyage en Roumanie (livre de Russel et Norvig)



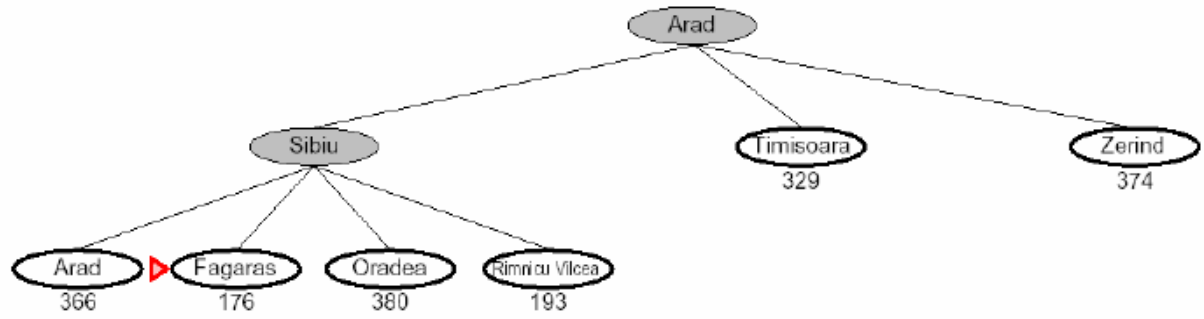
Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Les nœuds sont étiquetés avec leurs valeurs heuristiques. h(N) = distance à vol d'oiseau.
1er étape :

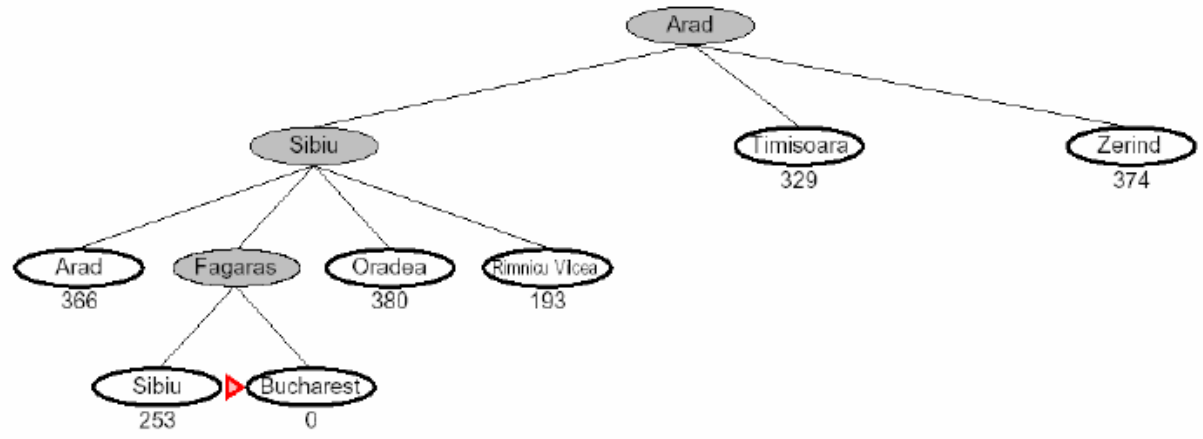
2ème étape :



3ème étape :



4ème étape :



<u>Propriètés :</u>

- Complétude : Non, peut rester pris dans une boucle. Oui, complet, si espace de recherche fini et si vérification d'absence de boucle ;
- Complexité en temps : $O(b^m)$ exponentielle ;
- Complexité en espace : $O(b^m)$ garde tous les nœuds en mémoire ;
- Optimal : Non

Remarque : la performance de la recherche avare est fonction de la précision de h(), avec une bonne fonction heuristique, les complexités en temps et en espace peuvent être fortement réduites.

La solution (*Arad* → *Sibiu* → *Fagaras* → *Bucarest*) n'est pas optimale; elle est de 32 km plus longue que (*Arad* → *Sibiu* → *Rimnicu* → *Pitesti* → *Bucarest*) car l'algorithme ne considère pas la distance parcourue.

Stratégie : toujours enlever le plus grand morceau du coût restant pour atteindre le but c'est-à-dire minimiser le coût estimé pour atteindre la solution (relativement efficace, quoique pas toujours optimal).

Susceptible de faux départ : exemple: pour aller de *Iasi* à *Fagaras*, la recherche avare considère *Neamt* avant *Vaului*, même si c'est une impasse.

Exemple de navigation robotique : h(n) = distance de Manhattan

8	7	6	5	4	3	2	3	4	5	6
7		5	4	3						5
6			3	2	1	0	1	2		4
7	6									5
8	7	6	5	4	3	2	3	4	5	6

- ***Algorithme A****

L'idée de cet algorithme est de combiner les deux méthodes de recherche avare et en coût uniforme. En effet, la recherche avare minimise le coût estimé h(n) du nœud n au but réduisant ainsi beaucoup le coût de la recherche, mais il *n'est pas optimal et pas complet*. De plus, l'algorithme de recherche en coût uniforme minimise le coût g(n) depuis l'état initial au noeud n, il est optimal et complet, mais pas très efficace.

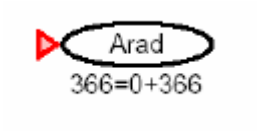
Donc l'objectif de A* est de minimiser le coût total f(n) du chemin passant par le nœud n avec f(n) = g(n) + h(n).

- f(n) : Fonction d'évaluation : **Coût estimé** d'un chemin de la racine à un objectif passant par n
- g(n) : **Coût** du chemin le plus court connu actuellement pour aller de la racine au nœud n : fonction dynamique
- h(n) : **Estimation** du coût optimal pour atteindre un objectif à partir du nœud n : fonction statique

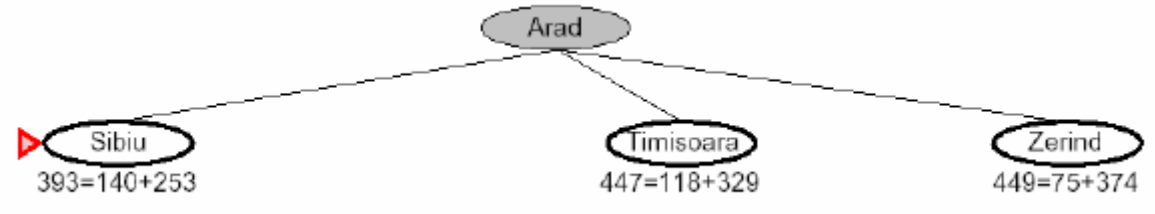
A* est optimal ssi A* utilise une fonction *heuristique admissible*, c'est-à-dire qui ne surestime jamais le coût réel : $\forall n, 0 \leq h(n) \leq h^*(n)$ avec $h^*(n)$ est le coût réel de n au but. Par exemple, $h_{\text{vol d'oiseau}}(n)$ de surestime jamais la distance réelle.

Exemple : voyage en Roumanie (exemple précédent)

1er étape :

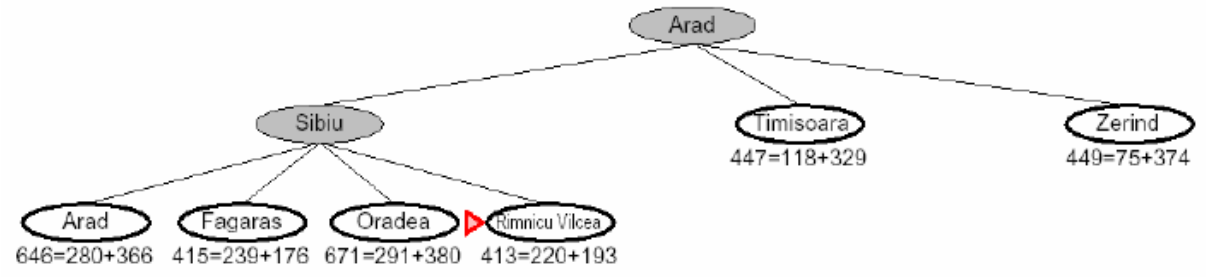


2ème étape :



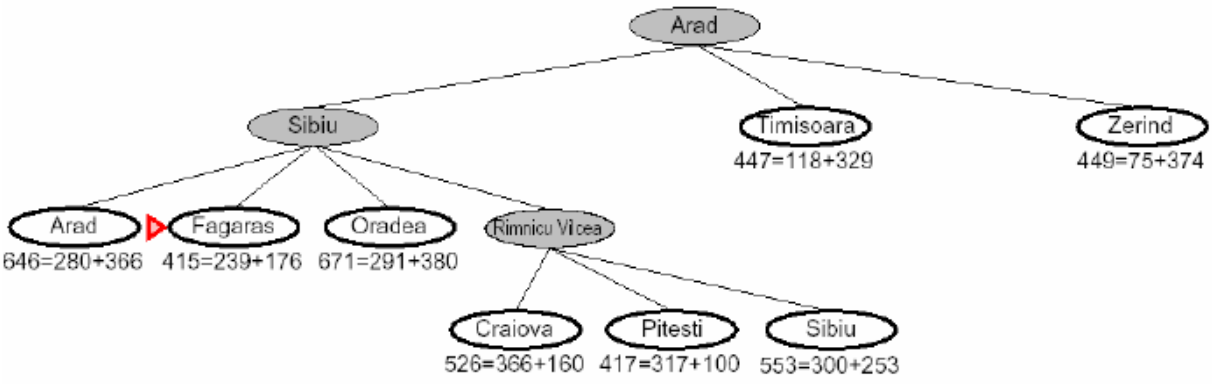
Min $f_i(n)=393$

3ème étape :



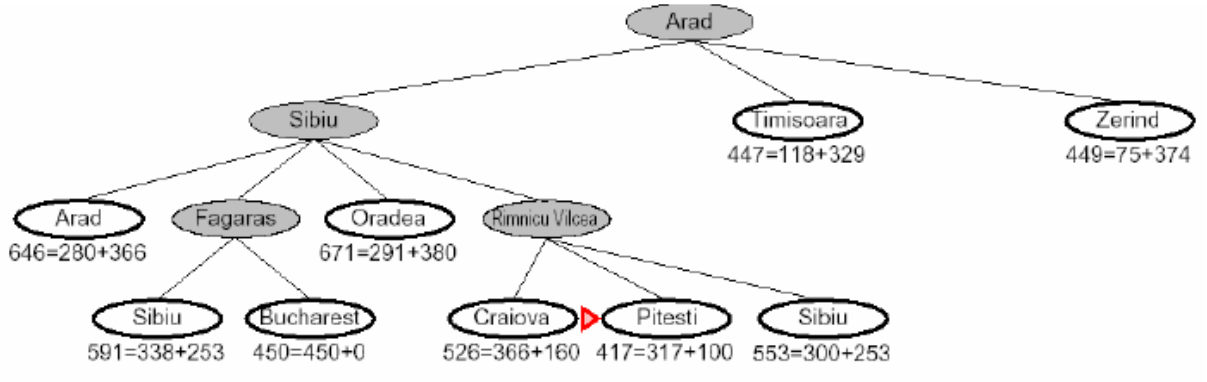
Min $f_i(n)=413$

4ème étape :



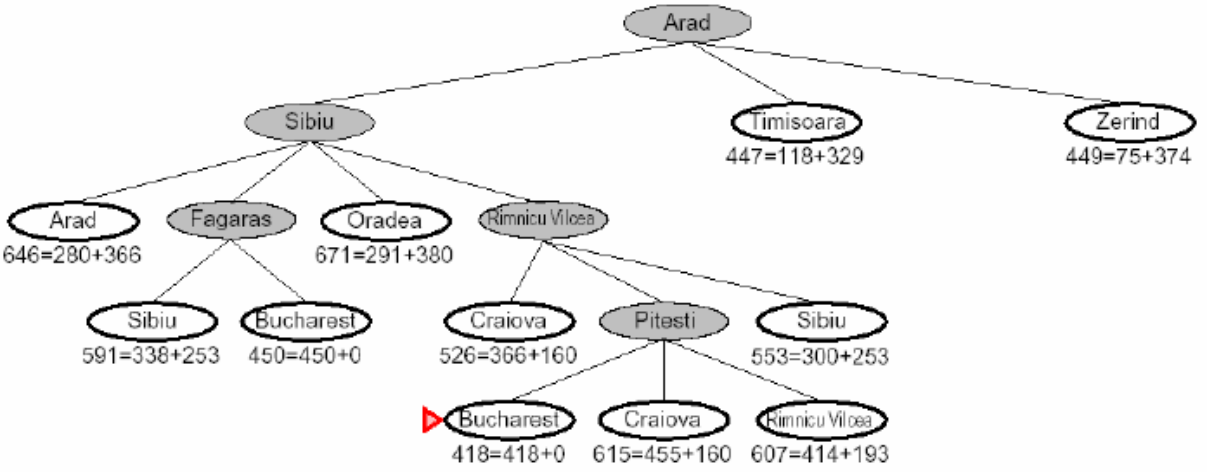
Min $f_i(n)=415$

5ème étape :



Min $f_i(n)=417$

6ème étape :



Propriètés :

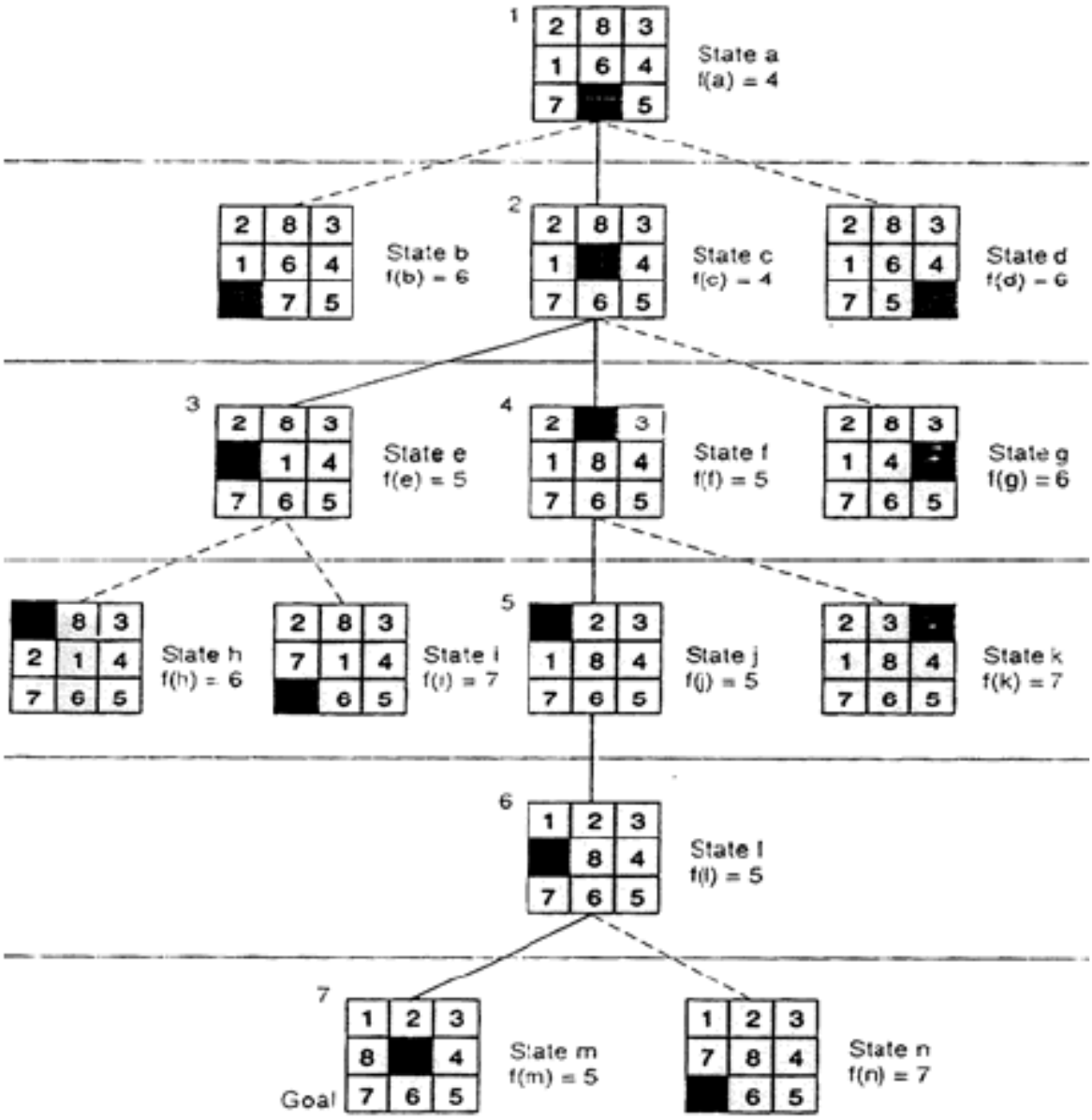
- Complétude : Oui, sauf si nombre infini de nœuds ;
- Complexité en temps : exponentielle ;
- Complexité en espace : garde tous les nœuds en mémoire ;
- Optimal : Oui

Exemple de navigation robotique : h(n) = distance de Manhattan et g(n) = distance de l'état de départ à l'état n.

8+3	7+4	6+3	5+6	4+7	3+8	2+9	3+10	4	5	6
7+2		5+6	4+7	3+8						5
6+1			3	2+9	1+10	0+11	1	2		4
7+0	6+1									5
8+1	7+2	6+3	5+4	4+5	3+6	2+7	3+8	4	5	6

Exemple 2 : Jeu de taquin

Appliquons A* pour le jeu de taquin avec g étant le nombre de jetons déplacés (distance de l'état de départ à l'état courante) et h le nombre de jeton mal placés.



4.8 Exemple de résolution : Problème des récipients

Nous disposons de deux récipients sans aucune marque de graduation : un de 4 litres (X) et un de 3 litres (Y). Comment remplir le récipient de 4 litres avec exactement 2 litres d'eau ?

Espace d'états : C'est l'espace de tous les états, de toutes les solutions du problème posé. Dans notre problème, l'état du système à tout moment est déterminé par la quantité d'eau dans chacune des récipients.

État : (x, y)	où	x : 0, 1, 2, 3, 4	# litres dans X
		y : 0, 1, 2, 3	# litres dans Y

- État initial : (0, 0)

- État final : (2, y)

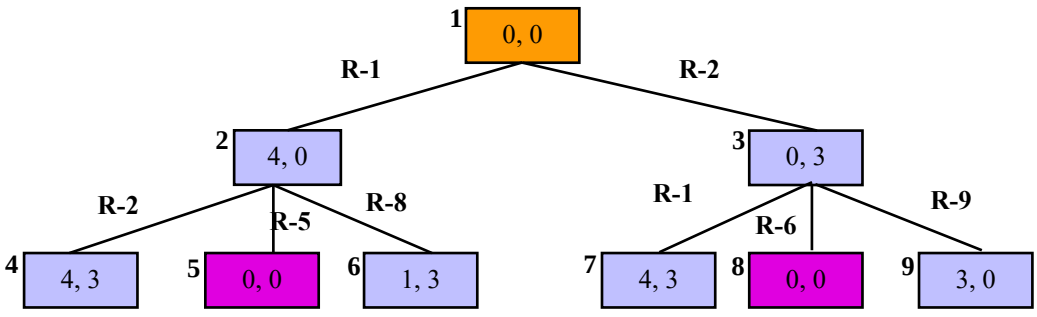
Base de connaissances :

- Connaissances explicites : Pas de moyen de mesure
- Connaissances implicites : Vider par terre le contenu d'un récipient ; Transvider d'un récipient à l'autre ; Vider un récipient d'une quantité indéterminée d'eau
- Connaissances à priori : Transvider les 2 litres du récipient Y à récipient X ; Vider le récipient X si Y contient 2 litres.

Règles de production :

R-1	Si x < 4 Alors (4, y)	Remplir le récipient X
R-2	Si y < 3 Alors (x, 3)	Remplir le récipient Y
R-3	Si x > 0 Alors (x-d, y)	Vider d'une quantité d le récipient X
R-4	Si y > 0 Alors (x, y-d)	Vider d'une quantité d le récipient Y
R-5	Si x >0 Alors (0, y)	Vider le récipient X
R-6	Si y > 0 Alors (x, 0)	Vider le récipient Y
R-7	Si x+y ≥ 4 ET y > 0 Alors (4, y-(4-x))	Transvider le récipient Y dans X jusqu'à la remplir
R-8	Si x+y ≥ 3 ET x > 0 Alors (x-(3-x), 3)	Transvider le récipient X dans Y jusqu'à la remplir
R-9	Si x+y ≤ 4 ET y > 0 Alors (x+y, 0)	Transvider le récipient Y dans X
R-10	Si x+y ≤ 3 ET x > 0 Alors (0, x+y)	Transvider le récipient X dans Y
R-11	Si x = 0 ET y = 2 Alors (2, 0)	Transvider les 2 litres recherchés de le récipient Y à X
R-12	Si y = 2 ET x > 0 Alors (0, 2)	Vider le récipient X si le récipient Y en contient 2

Stratégies de recherche: largeur d'abord

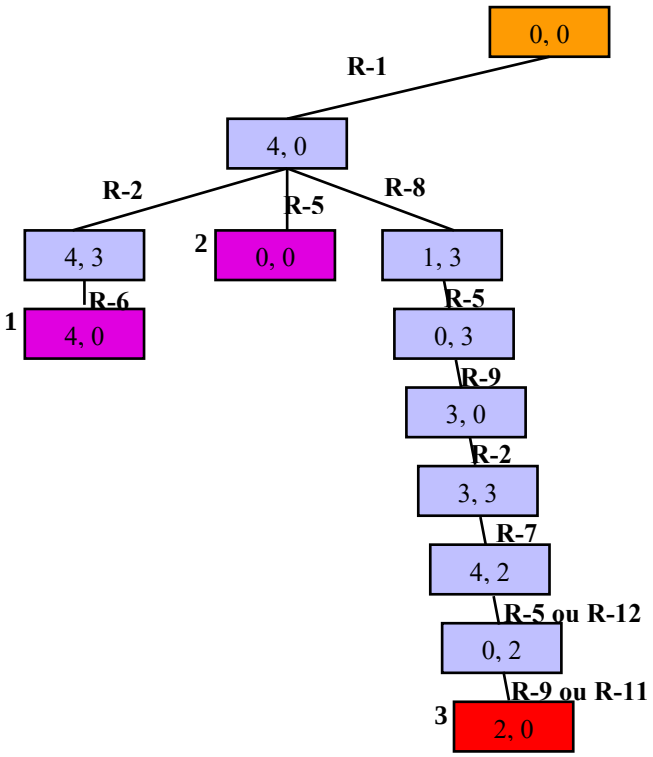


Stratégies de recherche: profondeur d'abord

Sélection des règles: choix par tentatives ; la première applicable

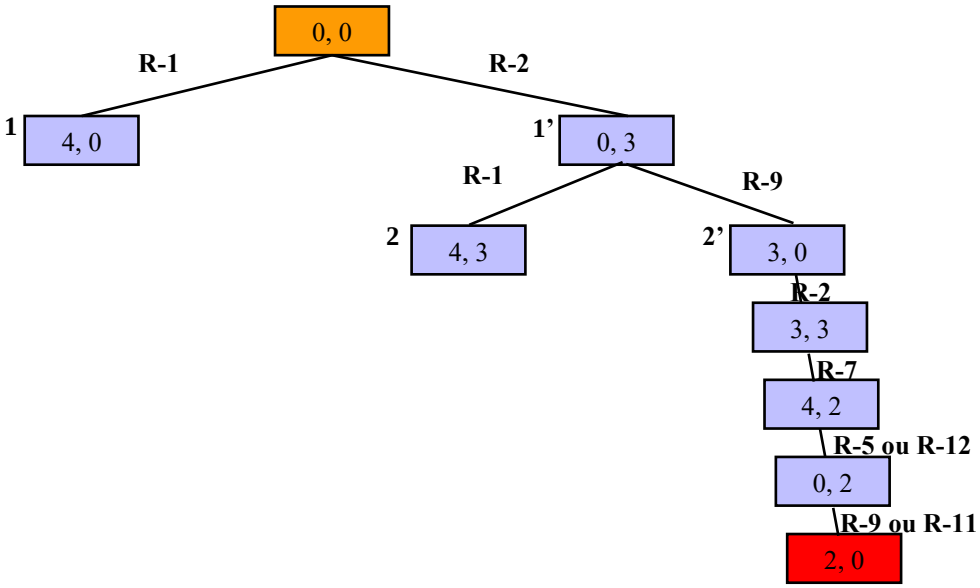
Irréversible : un seul chemin exploré

Tentatives : retour arrière si but non-atteint

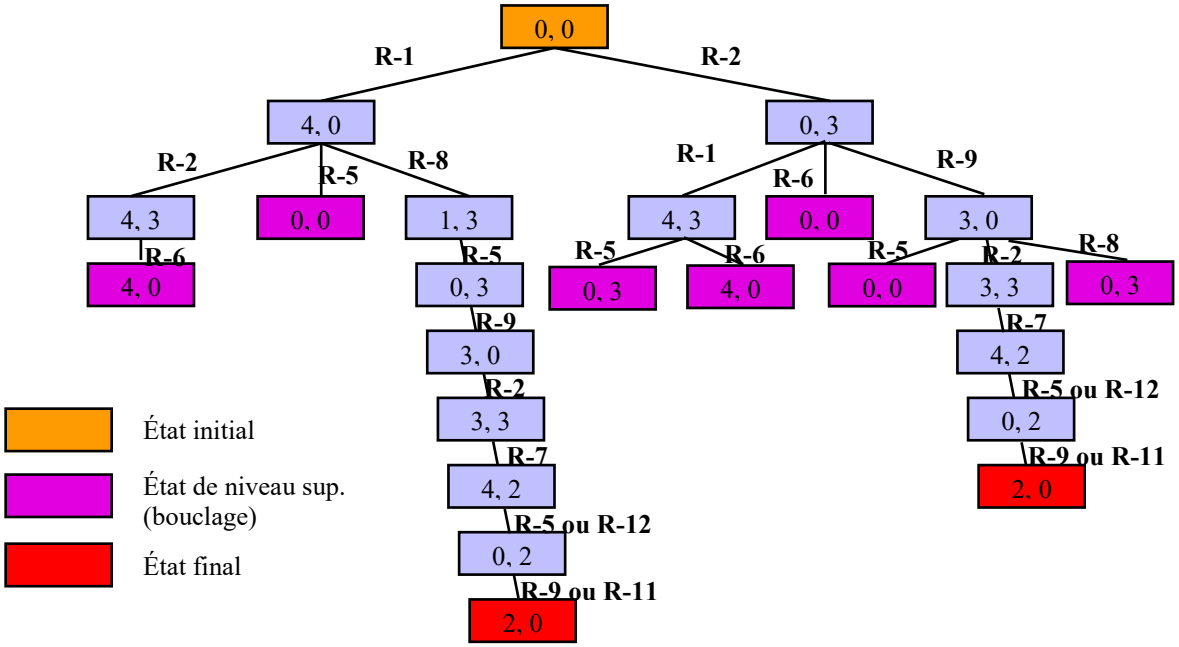


Stratégies de recherche: heuristique

Règle heuristique: choisir x+y le plus petit



Stratégies de recherche: résumé



Annexe :

Programmation logique

Présentation du langage PROLOG

Buts :

- Initiation à la programmation en logique : application de certaines notions vues en cours de logique
- Découverte de la programmation en Prolog : utilisation d'exemples liés à l'intelligence artificielle

A.1 Introduction à la programmation logique

La programmation dans le domaine informatique est l'ensemble des activités qui permettent l'écriture des programmes informatiques. Les techniques de programmation existantes sont :

- Programmation impérative
    - Programmation orientée objet
    - Programmation par contrat
- Programmation déclarative
    - Programmation fonctionnelle
    - Programmation logique
    - Programmation par contraintes
- Programmation à base de composants
- Programmation orientée aspect
- Programmation concurrente

La programmation logique est une forme de programmation dont le principe est de définir des *règles de logique mathématique* au lieu de fournir une succession d'instruction que l'ordinateur exécuterait. **Prolog** est l'un des principaux langages de programmation logique. Le nom Prolog est un acronyme de *PROgrammation LOGique*. Il a été créé par Alain Colmerauer et Philippe Roussel vers **1972**. Le but était de faire un langage de programmation qui permettait d'utiliser l'expressivité de la logique au lieu de définir pas à pas la succession d'instruction que doit exécuter un ordinateur.

Prolog est utilisé dans de nombreux programmes d'**intelligence artificielle** et dans le traitement de la linguistique par ordinateur (surtout ceux concernant les langages naturels). Sa syntaxe et sémantique sont considérées comme très simples et claires. Prolog est basé sur le calcul des prédicats du premier ordre ; cependant il est restreint à n'accepter que les clauses de Horn. L'exécution d'un programme Prolog est effectivement une application du théorème prouvant par résolution du premier ordre. Les concepts fondamentaux sont l'*unification*, la *récursivité* et le *retour sur trace*.

Il existe différents outils pour vous permettre de programmer en Prolog :

- Turbo Prolog par Borland
- SWI Prolog (http://www.swi-prolog.org)
- Visual Prolog (http://www.visual-prolog.com)
- GNU Prolog (http://www.gprolog.org/)

A.2 Prolog et les systèmes experts

Prolog est parfaitement adapté pour formaliser des systèmes experts. En effet, un système expert est un programme informatique simulant l'intelligence humaine dans un champ particulier de la connaissance ou relativement à une problématique déterminée. Un système expert a trois composantes essentielles :

- Une base de connaissances, formée des énoncés relatifs aux faits de tous ordres constitutifs du domaine ;
- Un ensemble de règles de décision, consignant les méthodes, procédures et schémas de raisonnement utilisés dans le domaine ;
- Un moteur d'inférence, sous-système qui permet d'appliquer les règles de décision à la base de connaissances.

Ces trois points sont extrêmement simples à implémenter avec Prolog :

- La base de connaissances est constituée par les faits et quelques règles pour éviter l'énumération exhaustive de tous les faits ;
- Les règles de décision sont des règles (au sens de Prolog) ;
- Le moteur d'interface est l'interpréteur Prolog lui-même.

L'**univers** PROLOG est une base de **connaissances** décrivant l'état du monde à l'aide de relations (**prédicats**) portant sur des entités (**termes**)

A.3 Structure d'un programme en Prolog

Un programme en Prolog est une description du problème en trois parties :

- Nom et structure des objets mis en jeu,
- Nom des relations que nous établissons entre ces objets,
- Faits et règles exprimant ces relations.

D'où la structure suivante d'un programme en Turbo Prolog :

Domains
/* déclaration des types */
Predicates
/* déclaration des prédicats */
Clauses
/* déclaration des clauses (règles et faits) */
Goal
/* définition d'un but */

Remarque : Le but est une requête de l'utilisateur. Si cette requête est insérée dans la section goal alors c'est un *but interne* qui fournit une seule solution. Si le goal est un *but externe* il fournit plusieurs solutions.

Exemple 1 :

```
domains
 qui,quoi = symbol

predicates
 aime_bien(qui,quoi)

clauses
 aime_bien(nour,football). /* nour aime bien le football*/
 aime_bien(nour, tennis).
 aime_bien(noura,tennis).

 aime_bien(nour,X) :- aime_bien(noura,X). /*Si noura aime bienX alors nour aime bienX*/
```

Exemple 2 : Considérons l'énoncé suivant :

Socrate est un homme.

Tout homme est mortel.

Socrate est-il mortel ?

Calcul des prédicats	**Prolog**
$\exists x$, homme(x) $\forall x$, homme(x) $\rightarrow$ mortel(x)	homme (socrate). mortel(X) :- homme(X). ?- mortel (socrate).

Exemple 3 : la famille :

```
masculin(tom).    % tom est de sexe masculin
masculin(tim).
masculin(bob).
masculin(jim).     % «paquet» de clauses

feminin(pam).
feminin(liz).
feminin(ann).
feminin(pat).

enfant(bob,pam).
enfant(bob,tom).
enfant(liz,tom).
enfant(ann,bob).
enfant(pat,bob).
enfant(tim,liz).
enfant(jim,pat).
```

Est-ce que pat est un enfant de bob ?

```
?- enfant(pat,bob).
Yes
```

Quels sont les enfants de tom ?

```
?- enfant(X,tom).
X = bob
X = liz
```

A.4 Syntaxe et terminologie Prolog

Un programme Prolog est constitué d'un ensemble de clauses ; une clause est une affirmation portant sur des atomes logiques ; un atome logique exprime une relation entre des termes ; les termes sont les objets de l'univers.

a. Les termes

Les objets manipulés par un programme Prolog (les données du programme) sont appelés des termes. On distingue trois sortes de termes :

1. Les variables représentant des objets inconnus de l'univers. Syntaxiquement, une variable est une chaîne alphanumérique commençant par une majuscule (exp : Var, X, Var_prog) ou par un souligné (exp : _objet, _21). La variable anonyme est notée « _ » et représente un objet dont on ne souhaite pas connaître la valeur.

2. Les termes élémentaires (ou termes atomiques) représentent les objets simples connus de l'univers. On distingue trois sortes de termes élémentaires :
    - Les nombres : entiers ou flottants,
    - Les identificateurs : un identificateur est une chaîne alphanumérique commençant par une minuscule (exp : toto, aX, jean_Paul),
    - Les chaînes de caractères entre guillemets (exp : "Toto","123")

3. Les termes composés représentent les objets composés (structurés) connus de l'univers. Syntaxiquement, un terme composé est de la forme :

    $foncteur(t_1,\ldots,t_n)$

    Un foncteur est une chaîne alphanumérique commençant par une minuscule, et t_1,..,t_n : sont des termes (variables, termes élémentaires ou termes composés). Le nombre d'arguments n est appelé arité du terme.

    Par exemple, adresse (18, "rue de Gafsa", Ville) est un terme composé de foncteur adresse et d'arité 3 dont les deux premiers arguments sont les termes élémentaires 18, "rue de Gafsa", et le troisième argument est la variable Ville.

    De même, cons(a,cons(X,nil)) est un terme composé de foncteur cons et d'arité 2, dont le premier argument est le terme élémentaire a et le deuxième argument le terme composé cons(X,nil).

b. Les relations, ou atomes logiques

Un *atome logique* exprime une *relation* entre des termes ; cette relation peut être vraie ou fausse. Syntaxiquement, un atome logique est de la forme:

symbole-de-prédicat(t_1, ..., t_n)

ou symbole-de-prédicat est une chaîne alpha-numérique commençant par une minuscule, et t_1, ..., t_n sont des termes. Le nombre d'arguments n est appelé arité de l'atome logique.

Exemple : pere(toto,paul) est une relation d'arité 2 entre les termes élémentaires toto et paul pouvant être interprétée par ``toto est le père de paul".

De même, habite(X,adresse(12,"rue r",Gafsa)) est une relation d'arité 2 entre la variable X et le terme composé adr(12,"rue r", Gafsa) pouvant être interprétée par ``une personne inconnue X habite à l'adresse (12,"rue r", Gafsa)".

c. Les clauses

Une clause est une affirmation inconditionnelle (un fait) ou conditionnelle (une règle).

1. Un *fait* est de la forme : A. où A est un atome logique (relation entre les termes) et signifie que la relation définie par A est vraie (sans condition).

   Par exemple, le fait père (toto, paul) indique que la relation « toto est le père de paul » est vraie.

   Une variable dans un fait est quantifiée universellement. Par exemple, le fait : egal(X,X) indique que la relation « X est egale à X » pour toute valeur (tout terme) que peut prendre X.

2. Une *règle* est de la forme : A_0 :- A_1,…A_n. où A_0,… , A_n sont des atomes logiques. Une telle règle signifie que la relation A_0 est vraie si les relations A_1 et... et A_n sont vraies. A_0 est appelé tête de clause et A_1,…,A_n est appelé corps de clause.

   Une variable apparaissant dans la tête d'une règle (et éventuellement dans son corps) est *quantifiée universellement*. Une variable apparaissant dans le corps d'une clause mais pas dans sa tête est *quantifiée existentiellement*. Par exemple, la clause

   meme_pere(X,Y) :- pere(P,X), pere(P,Y).

   se lit: « pour tout X et pour tout Y, meme_pere(X,Y) est vrai s'il existe un P tel que pere(P,X) et pere(P,Y) soient vrais ».

d. Les programmes Prolog

Un programme Prolog est constitué d'une suite de clauses regroupées en paquets. L'ordre dans lequel les paquets sont définis n'est pas significatif. Chaque paquet définit un prédicat et est constitué d'un ensemble de clauses dont l'atome de tête a le même symbole de prédicat et la même arité. L'ordre dans lequel les clauses sont définies est significatif. Intuitivement, deux clauses d'un même paquet sont liées par un ou logique. Par exemple, le prédicat personne défini par les deux clauses:

personne(X) :- homme(X).

personne(X) :- femme(X).

se lit « pour tout X, personne(X) est vrai si homme(X) est vrai ou femme(X) est vrai ».

e. Exécution de programmes Prolog

« Exécuter » un programme Prolog consiste à poser une question à l'interprète PROLOG. Une question (ou but ou activant) est une suite d'atomes logiques séparés par des virgules. La réponse de Prolog est « yes » si la question est une conséquence logique du programme, ou « no » si la question n'est pas une conséquence logique du programme. Une question peut

comporter des variables, quantifiées existentiellement. La réponse de Prolog est alors l'ensemble des valeurs des variables pour lesquelles la question est une conséquence logique du programme.

Par exemple, la question

?- pere(toto,X), pere(X,Y).

se lit « est-ce qu'il existe un X et un Y tels que pere(toto,X) et pere(X,Y) soient vrais ». La réponse de Prolog est l'ensemble des valeurs de X et Y qui vérifient cette relation. Autrement dit, la réponse de Prolog à cette question devrait être l'ensemble des enfants et petits-enfants de toto... si toto est effectivement grand-père.

A.5 Signification (sémantique) d'un programme Prolog

a. Définitions préliminaires

Substitution : Une substitution (notée s) est une fonction de l'ensemble des variables dans l'ensemble des termes. Par exemple, s = { X <- Y, Z <- f(a,Y) } est la substitution qui « remplace » X par Y, Z par f(a,Y), et laisse inchangée toute autre variable que X et Z. Par extension, une substitution peut être appliquée à un atome logique.

Par exemple, s(p(X,f(Y,Z))) = p(s(X),f(s(Y),s(Z))) = p(Y,f(Y,f(a,Y)))

Instance : Une instance d'un atome logique A est le résultat s(A) de l'application d'une substitution s sur A.

Par exemple, pere(toto,paul) est une instance de pere(toto,X).

Unificateur : Un unificateur de deux atomes logiques A1 et A2 est une substitution s telle que s(A1) = s(A2).

Par exemple, soit A1 = p(X,f(X,Y)), et soit A2 = p(a,Z) ; s = { X <- a, Z <- f(a,Y) } est un unificateur de A1 et A2 car s(A1) = s(A2) = p(a,f(a,Y)).

Unificateur plus général : Un unificateur s de deux atomes logiques A1 et A2 est le plus général (upg) si pour tout autre unificateur s' de A1 et A2, il existe une autre substitution s" telle que s' = s"(s).

Par exemple, s = { X <- Y } est un upg de p(X,b)et p(Y,b), tandis que s' = { X <- a, Y <- a } n'est pas un upg de p(X,b)et p(Y,b).

L'algorithme de Robinson calcule un upg de deux termes, ou rend "echec" si les deux termes ne sont pas unifiables :

```
fonction upg( t1, t2: termes) rend l'upg de t1 et t2 ou echec
   si t1 = t2 alors  rendre(substitution vide) finsi
   si t2 est une variable alors permuter t1 et t2 finsi
   si t1 est une variable alors
      si t2 contient t1 alors rendre(echec)  /* test d'occurrence */
      sinon rendre({ t1 <-  t2 }) finsi
   sinon
      si t1 et t2 sont 2 structures de meme symbole et de meme arité
      alors
         soit t1 = symb(u_1, ..., u_n) et t2 = symb(v_1, ..., v_n)
         s <- substitution vide
         pour i de 1 a n faire
            s' <-  upg( s(u_i), s(v_i) )
            si s' = echec alors rendre(echec) finsi
            s <-  s'(s)
         finfaire
      sinon rendre(echec) finsi
   finsi
```

b. Dénotation d'un programme Prolog

La dénotation d'un programme Prolog P est l'ensemble des atomes logiques qui sont des conséquences logiques de P. Ainsi, la réponse de Prolog à une question est l'ensemble des instances de cette question qui font partie de la dénotation. Cet ensemble peut être "calculé" par une approche ascendante, dite en chaînage avant: on part des faits (autrement dit des relations qui sont vraies sans condition), et on applique itérativement toutes les règles conditionnelles pour déduire de nouvelles relations ... jusqu'à ce qu'on ait tout déduit. Considérons par exemple le programme Prolog suivant:

```
parent(paul,jean).
parent(jean,anne).
parent(anne,marie).

homme(paul).
homme(jean).

pere(X,Y) :- parent(X,Y), homme(X).

grand_pere(X,Y) :- pere(X,Z), parent(Z,Y).
```

L'ensemble des relations vraies sans condition dans P est E_0 :

E_0 = {parent(paul,jean), parent(jean,anne), parent(anne,marie), homme(paul), homme(jean)}

À partir de E_0 et P, on déduit l'ensemble des nouvelles relations vraies E_1:

E_1 = { pere(paul,jean), pere(jean,anne) }

À partir de E_0, E_1 et P, on déduit l'ensemble des nouvelles relations vraies E_2:

E_2 = { grand_pere(paul,anne), grand_pere(jean,marie) }

À partir de E_0, E_1, E_2 et P, on ne peut plus rien déduire de nouveau. L'union de E_0 , E_1 et E_2 constitue la dénotation (l'ensemble des conséquences logiques) de P.

Malheureusement, la dénotation d'un programme est souvent un ensemble infini et n'est donc pas calculable de façon finie. Considérons par exemple le programme P suivant:

```
plus(0,X,X).
plus(succ(X),Y,succ(Z)) :- plus(X,Y,Z).
```

L'ensemble des atomes logiques vrais sans condition dans P est

E_0 = { plus(0,X,X) }

à partir de E_0 et P, on déduit

E_1 = { plus(succ(0),X,succ(X)) }

à partir de E_0, E_1 et P, on déduit

E_2 = { plus(succ(succ(0)),X,succ(succ(X))) }

à partir de E_0, E_1, E_2 et P, on déduit

E_3 = { plus(succ(succ(succ(0))),X,succ(succ(succ(X)))) }

etc ...

c. Signification opérationnelle

D'une façon générale, on ne peut pas calculer l'ensemble des conséquences logiques d'un programme par l'approche ascendante: ce calcul serait trop couteux, voire infini. En revanche, on peut démontrer qu'un but (composé d'une suite d'atomes logiques) est une conséquence logique du programme, en utilisant une approche descendante, dite en chaînage arrière:

Pour prouver un but composé d'une suite d'atomes logiques (par exemple, But = [A_1, A_2, .., A_n]), l'interprète Prolog commence par prouver le premier de ces atomes logiques (A_1). Pour cela, il cherche une clause dans le programme dont l'atome de tête s'unifie avec le premier atome logique à prouver (par exemple, la clause A'_0 :- A'_1, A'_2, ..,A'_r telle que upg(A_1,A'_0) = s).

Puis l'interprète Prolog remplace le premier atome logique à prouver (A_1) dans le but par les atomes logiques du corps de la clause, en leur appliquant la substitution (s). Le nouveau but à prouver devient :

But = [s(A'_1), s(A'_2), .., s(A'_r), s(A_2), .., s(A_n)]

L'interprète Prolog recommence alors ce processus, jusqu'à ce que le but à prouver soit vide, c'est à dire jusqu'à ce qu'il n'y ait plus rien à prouver. A ce moment, l'interprète Prolog a prouvé le but initial ; si le but initial comportait des variables, il affiche la valeur de ces variables obtenue en leur appliquant les substitutions successivement utilisées pour la preuve.
Il existe généralement plusieurs clauses dans le programme Prolog dont l'atome de tête s'unifie avec le premier atome logique à prouver. Ainsi, l'interprète Prolog va successivement répéter ce processus de preuve pour chacune des clauses candidates. Par conséquent, l'interprète Prolog peut trouver plusieurs réponses à un but.
Ce processus de preuve en chaînage arrière est résumé par la fonction prouver(But) suivante. Cette fonction affiche l'ensemble des instances de But qui font partie de la dénotation du programme:

```
procedure prouver(But: liste d'atomes logiques )
   si But = [] alors
      /* le but initial est prouvé */
      /* afficher les valeurs des variables du but initial */
   sinon soit But = [A_1, A_2, .., A_n]
      pour toute clause (A'_0 :- A'_1, A'_2, ..,A'_r) du programme:
         (ou les variables ont été renommées)
         s <- upg(A_1 ,A'_0)
         si s != echec alors
            prouver([s(A'_1), s(A'_2), .. s(A'_r), s(A_2), .. s(A_n)], s(But-init}))
         finsi
      finpour
   finsi
fin prouver
```

Quand on pose une question à l'interprète Prolog, celui-ci exécute dynamiquement l'algorithme précédent. L'arbre constitué de l'ensemble des appels récursifs à la procédure prouver_bis est appelé ***arbre de recherche***.

Remarques :

- La stratégie de recherche n'est pas complète, dans la mesure ou l'on peut avoir une suite infinie d'appels récursifs,
- La stratégie de recherche dépend d'une part de l'ordre de définition des clauses dans un paquet (si plusieurs clauses peuvent être utilisées pour prouver un atome logique, on considère les clauses selon leur ordre d'apparition dans le paquet), et d'autre part de l'ordre des atomes logiques dans le corps d'une clause (on prouve les atomes logiques selon leur ordre d'apparition dans la clause).

A.6 Les listes

La liste est un terme composé particulier de symbole de fonction « . » et d'arité 2 : le premier argument est l'élément de tête de la liste, et le deuxième argument est la queue de la liste. La liste vide est notée « [] ».

Notations:

la liste .(X,L) est également notée [X|L] ,

la liste .(X1, .(X2, L)) est également notée [X1, X2|L],

la liste .(X1, .(X2, ..., .(Xn, L) ...)) est également notée [X1, X2, ..., Xn|L],

la liste [X1, X2, X3, ..., Xn|[]] est également notée [X1, X2, X3, ..., Xn].

Par exemple, la liste [a,b,c] correspond à la liste .(a,.(b,.(c,[]))) et contient les 3 éléments a , b et c. La liste [a,b|L] correspond à la liste .(a,.(b,L)) et contient les 2 éléments a et b , suivis de la liste (inconnue) L.

Une liste est une structure récursive: la liste Liste = [X|L] est composée d'un élément de tête X et d'une queue de liste L qui est elle-même une liste. Par conséquent, les relations Prolog qui ``manipulent" les listes seront généralement définies par :

- Une ou plusieurs clauses récursives, définissant la relation sur la liste [X|L] en fonction de la relation sur la queue de liste L,
- Une ou plusieurs clauses non récursives assurant la terminaison de la manipulation, et définissant la relation pour une liste particulière (par exemple, la liste vide, ou la liste dont l'élément de tête vérifie une certaine condition...).

A.7 La coupure

a. Signification opérationnelle de la coupure

La coupure est un prédicat sans signification logique (la coupure n'est ni vraie ni fausse), utilisée pour "couper" des branches de l'arbre de recherche. La coupure, aussi appelée "cut", est notée !.

La coupure est toujours « prouvée » avec succès dans la procédure prouver décrite plus haut. La « preuve » de la coupure a pour effet de bord de modifier l'arbre de recherche: elle coupe l'ensemble des branches en attente créées depuis l'appel de la clause qui a introduit la coupure.

Considérons par exemple le programme Prolog suivant:

```
p(X,Y) :- q(X), r(X,Y).
p(c,c1).

q(a).
q(b).
```

```
r(a,a1).
r(a,a2).

r(b,b1).
r(b,b2).
r(b,b3).
```

L'arbre de recherche construit par Prolog pour le but p(Z,T) est:

```
prouver([p(Z,T)])
    b1 : s = {Z <- X, T <- Y}
    ---- prouver([q(X),r(X,Y)])
        b11 : s = {X <- a}
        ----- prouver([r(a,Y)])
            b111 : s = {Y <- a1}
            ------ prouver([],[p(a,a1)] --> solution1 = {Z=a, T=a1}
            b112 : s = {Y <- a2}
            ------ prouver([]) --> solution2 = {Z=a, T=a2}
        b12 : s = {X <- b}
        ----- prouver([r(b,Y)])
            b121 : s = {Y <- b1}
            ------ prouver([]) --> solution3 = {Z=b, T=b1}
            b122 : s = {Y <- b2}
            ------ prouver([]) --> solution4 = {Z=b, T=b2}
            b123 : s = {Y <- b3}
            ------ prouver([]) --> solution5 = {Z=b, T=b3}
    b2 : s = {Z <- c, T <- c1}
    ---- prouver([]) --> solution6 = {Z=c, T=c1}
```

En fonction de l'endroit ou l'on place une coupure dans la définition de p, cet arbre de recherche est plus ou moins élagué, et certaines solutions supprimées :

- Si on définit p par

  ```
  p(X,Y) :- q(X), r(X,Y), !.

  p(c,c1).
  ```

  Prolog donne la solution 1 puis coupe toutes les branches en attente (b112, b12 et b2).

- Si on définit p par

  ```
  p(X,Y) :- q(X), !, r(X,Y).

  p(c,c1).
  ```

  Prolog donne les solutions 1 et 2 puis coupe les branches en attente (b12 et b2).

- Si on définit p par

  ```
  p(X,Y) :- !, q(X), r(X,Y).

  p(c,c1).
  ```

  Prolog donne les solutions 1, 2, 3, 4 et 5 puis coupe la branche en attente (b2).

Exercice : que répond Prolog aux questions suivantes ?

?- element(L,[[a,b],[c,d,e]]), element(X,L).

?- element(L,[[a,b],[c,d,e]]), element(X,L), !.

?- element(L,[[a,b],[c,d,e]]), !, element(X,L).

?- !, element(L,[[a,b],[c,d,e]]), element(X,L).

b. Les applications de la coupure

Recherche de la première solution

Exprimer le déterminisme

La négation

Le "si-alors-sinon"

A.8 Bibliographie :

- Christine Solnon, *Introduction à PROLOG*, Université Lyon 1, 1997.
- J. W. Lloyd, *Fondements de la programmation en logique*, Eyrolles, 1988.
- F. Giannesini, H. Kanoui, R. Pasero et M. Van Caneghem, *Prolog*, InterEditions, 1985.
- Bratko, *Programmation en Prolog pour l'intelligence artificielle*, InterEditions, 1988.