>

Algorithmes & Raisonnement

Une introduction à l'IA

Représentation de connaissances

et Techniques de Recherche

2ème Année

2009-20010

ECL-MI

Alexandre SAIDI

Partie 1 : Introduction, Prolog

Table des matières

I- Quelques références Bibliographiques 8

II- Avant Propos : quelques exemples de l'état de l'art en IA....10

III- Ce que l'on appelle IA en 2009-2010....11

IV- Introduction 12

IV.1- Quelques caractéristiques des problèmes d'IA....15

IV.2- Exemples de stratégies de recherche :....16

V- Problématique et Origines de l'Intelligence Artificielle 18

V.1- "D'où tu parles ! " : Un peu d'histoire de l'IA....19

V.2- Exemple de question traitées par le système ANALOGY 22

V.3- Exemple de problème résolu dans le système SHDRLU (monde des blocs) 23

V.4- Importance des bases de connaissances 24

V.5- Évolution : l'IA devient une industrie (1980 - ...)....26

V.6- Les domaines qui contribuent à l'IA....27

V.6.a- Les deux grandes points de vu 29

V.7- Résumé du cadre de l'IA....30

VI- L'expérience MYCIN 31

VII- Outils et problèmes traités......33

VII.1- Différents paradigmes 33

Raisonnement, Perception, Apprentissage, Recherche intelligente, Acquisition de connaissances, etc......33

VII.2- Divers domaines et applications......34

VII.3- Des exemples de réalisation (outre l'état de l'art)......35

VII.4- Black Box : quelques outils 36

VII.4.a- Réseau de Neurones : exemple d'architecture......37

VII.4.b- Réseaux de Neurones & l'apprentissage de la prononciation 38

VII.4.c- Algorithmes Génétiques (un exemple) 39

VII.4.d- Vision artificielle (schéma et exemple)......40

VII.5- Open Box : Système Experts (exemple de règles)......41

VIII- Les langages de représentation de connaissances......42

VIII.1- Logique 42

VIII.2- Langages à base de réseaux......43

VIII.2.a- Un exemple 44

VIII.2.b- Un exemple plus général......45

VIII.2.c- Un autre exemple......46

VIII.2.d- Remarques sur les réseaux sémantiques......47

VIII.3- Objets et Frames......48

VIII.3.a- Un exemple de Frame......49

VIII.4- Discussion : avantage aux frames ! 50

VIII.5- Formalismes Procéduraux 51

IX- Éléments et caractéristiques d'une représentation 52

IX.1- Exemple 53

X- Autres formalismes (plus récents)..54

XI- Recherche : classement de quelques stratégies 55

XII- Exemples en Prolog (LP et CLP)..56

XII.1- Exemples simples en Prolog..56

XII.2- Exemples utilisant les contraintes (CLP)..57

XII.2.a- Puzzle logique (simple)..57

XII.2.b- Factorielle réversible 58

XII.2.c- Un équation simple..58

XII.2.d- Amortissement (domaine des Réels)..59

XII.2.e- Balance..60

XII.2.f- Pierres (exemple moins simple) 61

XII.3- Ingrédients d'un raisonnement en CSP (domaine fini)..63

XII.4- Résolutions dans les environnements CSP actuels..64

XII.5- Techniques de Consistance (Nœud, Arc et Chemin) 65

XII.5.a- Exemple : Planification de tâches..66

XII.6- Les stratégies générales (méta stratégies)..68

XII.6.a- Illustration par les N-reines..70

Qi = le numéro de ligne d'une reine dans la colonne i, 1 <= i <= 4..70

Les contraintes (in extenso pour la clarté) :..71

Méthode Retour arrière :..72

XIII- Résolution : aspects pratiques du Back-tracking....77

XIII.1- Retours arrières vs. Résolution des contraintes....78

XIII.1.a- Comparaison générale 78

XIII.1.b- Exemple de coloration....79

XIII.1.c- Formalisation CSP de la coloration de graphes....81

XIV- Domaines et exemples d'application des contraintes....82

XIV.1- Programmation logique vs. Programmation Impérative....86

XV- Le langage Prolog 87

XVI- Éléments du langage Prolog par exemples....91

XVI.1- Faits et questions élémentaires....91

XVI.2- Exemple d'une base de faits en Prolog 93

XVI.3- Questions plus complexes....94

XVI.4- Les variables....96

XVI.4.a- Variables dans les questions....97

XVI.4.b- Variables dans les faits 99

XVI.4.c- Variables dans les conjonctions....99

XVI.4.d- Variables dans les disjonctions....100

XVI.5- Démarche de résolution 101

XVI.6- Principe de le Résolution en Prolog....103

XVI.7- Unification 104

XVI.7.a- Principe de l'unification 105

XVI.8- Retour sur les règles....106

XVI.8.a- Disjonction dans les règles......*111*

XVI.8.b- La récursivité dans les règles......*112*

XVI.9- Pratique de l'unification......115

XVI.10- La résolution et le retour arrière......117

XVI.11- Hypothèse du monde fermé et la négation......119

XVI.12- Contrôle de la résolution......120

XVI.12.a- L'utilisation courante de cut......*122*

XVI.12.b- Un autre exemple de cut......*129*

XVI.13- Listes......130

XVI.14- Quelques prédicats de manipulation de listes......132

XVI.15- Opérations sur les termes......134

XVI.16- Arithmétique......137

XVI.17- Méta-variables et méta-prédicats......144

XVI.17.a- Méta prédicats toutes-solutions......*145*

XVI.18- Manipulation d'arbres......146

XVII- Retour sur les méthodologies......148

XVII.1- Méthode constructive (Greedy, British Museum)......149

XVII.2- Méthode Retours Arrières avec génération/test......151

XVII.3- Méthode contraindre et générer......153

XVII.3.a- La modification de la règle de calcul de Prolog......*154*

XVII.3.b- Un autre exemple (CSP)......*155*

XVIII- Systèmes d'inférence......158

XIX- La SLD-Résolution : la résolution en Prolog....164

XIX.1- Unification 169

XX- Stratégies de recherche et Programmation Logique....171

XX.1- Règles de choix....173

XX.2- Algorithme de résolution de Prolog....176

XX.3- Principe d'un méta-interpréteur de Prolog....178

XX.3.a- La version Gnu Prolog avec exemple 180

I- Quelques références Bibliographiques

- *Artificial Intelligence*. Elian Rich. Mc Graw Hill.
- *Artificial Intelligence*. P.H. Winston. Addison Wesley1993. 3rd Ed.
- *Heuristiques*. Judea Pearl. Cepadues Ed. 1990.
- *Paradigms of Artificial Intelligence Programming*. Peter Norvig. Morgan Kaufmann. 1992. Case studies in Common Lisp.
- *Principles of Artificial Intelligence*. Nils J. Nilsson. Springer-Verlag 1980.
- *Programmer en Prolog*. Jonathan Elbaz. Ed. Marketing. 1991.
- *Prolog Programming in Depth*. Michael A. Convington & all. Scott, Foresman & Co. 1988.
- *Simply Logical*. Peter Flach. Wiley & sons 1994
- *The Elements of Artificial Intelligence using Common Lisp*. Steven L. Tanimoto. Freeman. 1995.
- *Articicial Intelligence & Soft Computing*, Amit Konar, CRC Press, 2000
- *Crticicial Intelligence : a modern Approach*, S.J. Russel, P. Norvig, Prentice Hall, 1995
- Polycopié Prolog : A. Saidi, ECL 1990-2010

Remarque :

Un polycopié Prolog est séparément disponible et complète ce document.

Objet (sommaire) du cours

- *Une introduction à l'Intelligence Artificielle*
- *Les outils de représentation de connaissances*
- *Les algorithmes de recherche*
- *Techniques de résolution de Problèmes (par raisonnement)*
- *Prise en main du langage Prolog et réalisation de quelques exercices*
- *Programmation avec des contraintes*
- *Formulation et Résolution des problèmes combinatoires*

II- Avant Propos : quelques exemples de l'état de l'art en IA

- *Deep Blue* bat *Kasparov*.
- *Sojourner*, *Spirit*, et *Opportunity* explorent *Mars*.
- L'engin dirigé à distance *Deep Space I* de la *NASA* explore le système Solaire.
- Expérience de conduite *autonome* dans le désert (e.g. Challenge DARPA), le prochain : en ville.
- Les Robots automatiques de *nettoyage* (aspirateurs)
- Les *Robots* japonnais (à l'Image de leur inventeur, d'une hôtesse d'accueil, ..)
- Système automatiques de Parole / langage dans certaines compagnies aériennes
- Filtrage de *Spam* (machine learning).
- Systèmes (TDM) d'extraction de données depuis un corpus d'annonces
- Système automatique de *Q/R* (Q/A : Question Answering) pour des conversations non banales
- Système de traduction automatiques (e.g. Google en bonne voie)
- *OCR* approche 99%
- Classification d'image / émotion, Annotation automatique

  ...
- Différents systèmes de prise de décision

  ➞ Mais aussi des traders automatiques ....

III- Ce que l'on appelle IA en 2009-2010

- Différentes techniques de recherche optimisées (utilisées dans plusieurs application en IA)
  - Stratégies de résolution

- **Machine Learning : apprentissage automatique**
- Raisonnement
  - Raisonnement Logique (Induction, déduction, abduction, etc)
  - Système de raisonnement par règles de production (Systèmes Experts)
  - Formulation avec des contraintes (logique + numérique)

- Extraction de Connaissances et Reconnaissance de Formes (PR)
  - Inférence Grammaticale
  - SPAM
  - Web Sémantique : découverte de connaissances des contenus (texte, image, vidéo, émotion, ...)

- Algorithmes **Génétiques** et Réseaux de neurones
- NLP (statistique ou algorithmique)

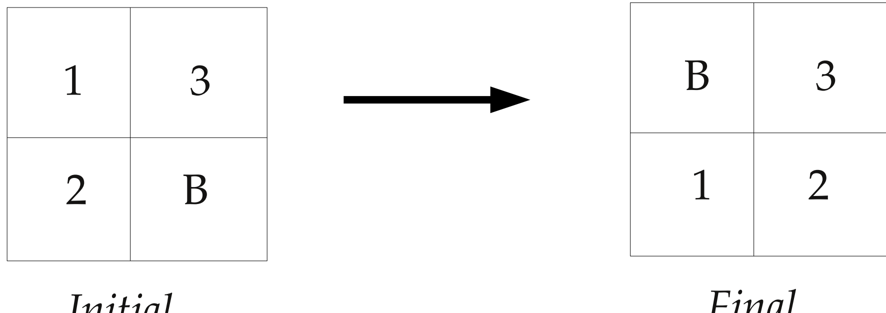
Avec applications dans quasi tous les domaines (industrie, Robotique, KBD, KM, ...)

IV- Introduction

Définition de l'Intelligence Artificielle (IA) :

> ***Simulation de l'intelligence humaine par une machine de telle sorte que la "bonne" connaissance soit utilisée à chaque étape de la résolution de problèmes.***

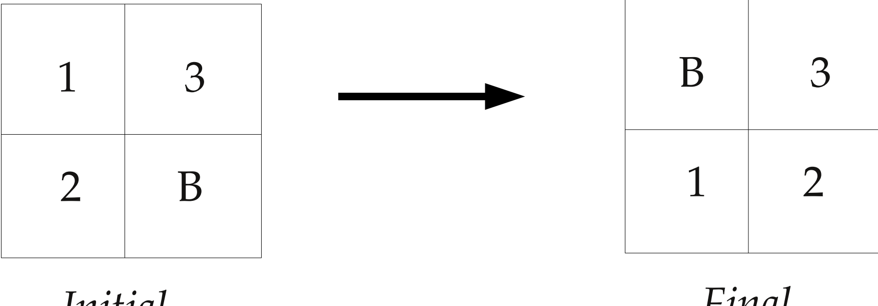
Un exemple simple : le Taquin



Approche générale de résolution de problèmes en IA : une stratégie de recherche de solution

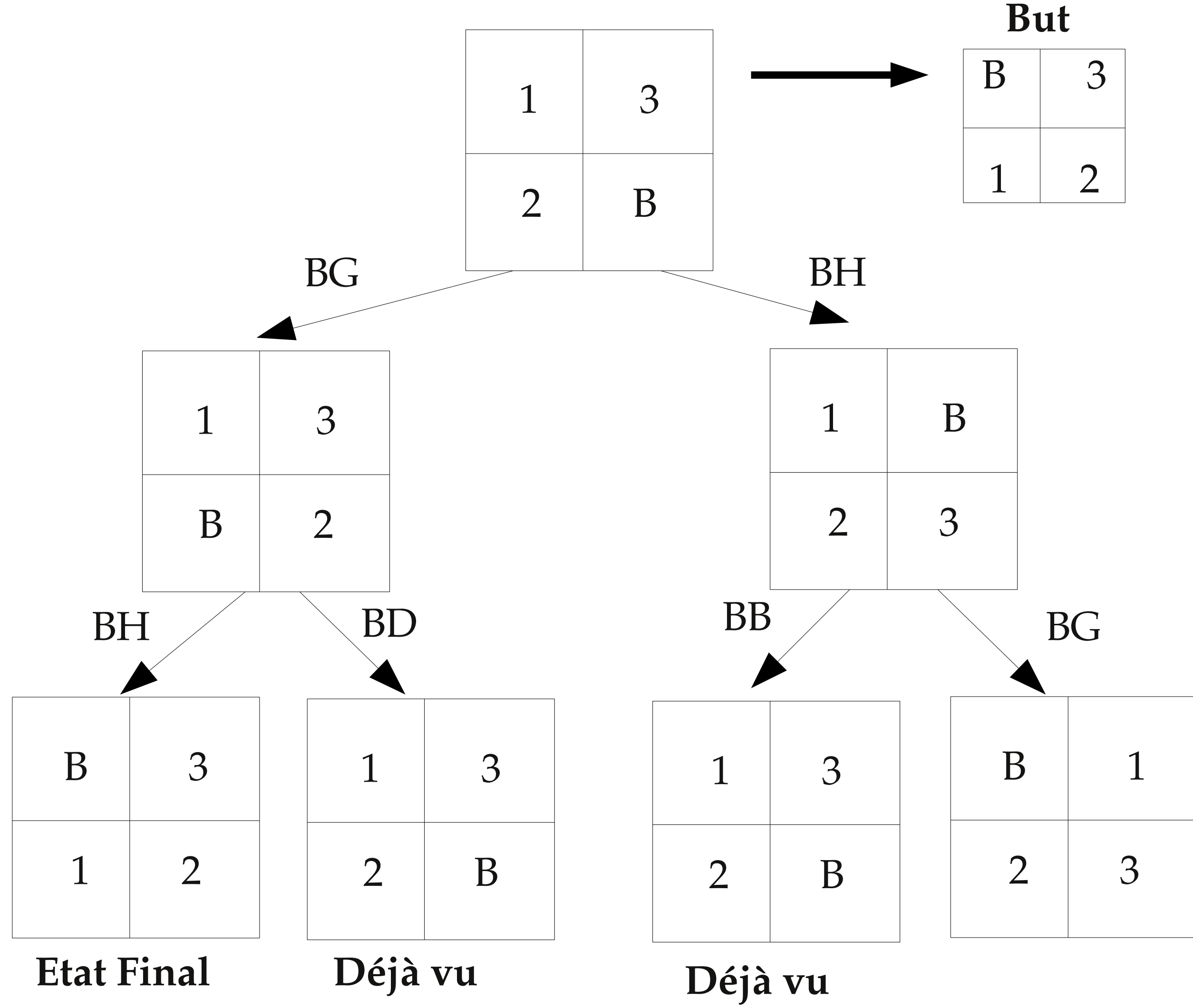
Algorithme général (du développement d'espace d'états) :

```
Etats = Etat_initial;  Deja_Vus = Etats
Tant que Etat_final ∉ Etats
   New_Etats = Appliquer une opération Op ∈ {BG, BD, BH, BB}
                        à chaque élément dans Etats
   Si New_Etats ∩  Deja_Vus ≠ ⊘
   Alors  Etats =  New_Etats - Deja_Vus
          Deja_Vus = New_Etats ∪ Etats
   FinSi
Fin Tant que
```



Initial *Final*

Déroulement :



IV.1- Quelques caractéristiques des problèmes d'IA

- Caractéristique : pas d'algorithme formel pour arriver directement à une solution
  - ➛ Pas de modèle mathématique immuable
  - ➛ on ne sait pas prédire la séquence qui nous mènera à une solution.

- Méthode générale utilisée : ***rechercher – comparer*** (search & match)

Quelques Exemples (jouets) :

- Jauge d'eau (cruches) :
  - ➞ Dans **l'espace d'états**, certaines *transitions* (actions) sont autorisées :
    remplir (ou compléter), *vider* (éventuellement. dans une autre cruche), ...

- **N-reines**, **Cavaliers** : placement (sous contraintes)
- **WGC** (loup/Chèvre/Chou) : charger, traverser, décharger (idem **Cannibales)**
- **TSP** (voyageur de commerce) : transition = aller d’une ville à une autre (sous contrainte)
- **Emploi du temps** (time table),
- Divers problèmes d'**affectation**, ...
- Divers problèmes d'optimisation

IV.2- Exemples de stratégies de recherche :

- **Générer & Tester**

  - Générer des états, les tester (e.g. N-reines), cas extrême !

    Variante : préférence pour les candidats de distance minimum au *But*

- **Retour arrière**

- **Hill Climbing**
  - Idem mais avec un coût de référence (jusqu'au *But*).
    - Si coûts identiques (extremum local), alors choix aléatoire
  - Utilisée (inconsciemment !) par exemple pour démontrer des identités trigonométriques

    ➞ on choisit l'élément qui nous fait avancer <u>localement</u> sans (avoir forcément) une vision <u>globale</u> de la solution complète.

- **Recherche Heuristiques** (e.g. parcours de cavalier)

  Fonction heuristique de choix pour sélectionner le <u>meilleur</u> candidat

- **Analyse "Means and Ends"** (e.g. Robot + pince)
  - Réduction du "gap" entre le **But** et l'état **courant**.
  - Application d'opérateurs qui réduisent ce "gap" (cf. démonstration de théorèmes en Mathématiques)

- **Recuit Simulé (Simulated Annealing : méta-heuristique) :**

  on part d'une solution donnée (S1, pris au hasard dans l'espace des solutions) pour obtenir une seconde (S2) en modifiant S1.

  ➟ Deux cas de figures parmi lesquels on trouve un compromis :
  - S2 améliore le critère que l'on cherche à optimiser (on dit qu'*on a fait baisser l'énergie du système*) :
    accepter S2 tend à chercher l'optimum dans le voisinage de la solution de départ.
  - S2 dégrade S1 : l'acceptation de S2 (" mauvaise ") permet d'explorer une plus grande partie de l'espace de solutions et tend à éviter de s'enfermer trop vite dans la recherche d'un optimum local**.**

    ➟ On accepte S2 selon certaines règles (dite *Metropolis* = un certain seuil de proba franchi)

- **RN et AG :** Solution Black Box

- **Autres (intégrée dans les stratégies ci-dessus) :**
  - Décomposition de problèmes (Divide & Conquer) : ➟ $I = \int (x^2 + 9x + 2)\, d_x$ ➟ $\int (x^2\, d_x) + \int (9x)\, d_x + \int (2)\, d_x$
  - Satisfaction de contraintes : ➟ $\{X1 \geq 2,\ X2 \geq 3,\ X1+X2 \leq 6,\ X1, X2 \in I\}$
  - Programmation Dynamique, B & B (voir plus loin) ...

V- Problématique et Origines de l'Intelligence Artificielle

Origine : Démonstration Automatique de Théorèmes (DAT)

- ✔ Une base d'axiomes + inférence
- ✔ Preuve des théorèmes = solution des problèmes
- ✔ L'idée : Une bonne technique de recherche peut donner une solution à **n'importe quel** problème ?

Mais : cas des problèmes NP-complets ?

➟ Pour résoudre un tel problème, on ne connaît aucun algorithme de complexité polynomiale (= non exponentielle) :

➟ soit aucun algorithme n'est connu, soit il en existe de complexité grande (e.g. exponentielle).

Exemple de problème <u>NP-Complet</u> :

JobShop : *N* tâches pour fabriquer différents produits,

M machines avec un <u>ordre</u> de passage de chaque tâche T_i sur toutes les machines M_j :

➟ Dans quel ordre passer les tâches pour minimiser le temps de fabrication d'une certaine quantité de chaque produit ?

La D.A.T. est inefficace pour les problèmes de taille importante.

➟ Une idée : **Systèmes Experts :** règles adéquates pour diviser les problèmes complexes en sous problèmes plus simples.

Un exemple emblématique : **MYCIN**. ➟ Système Expert de diagnostic des infections sanguines.

V.1- "D'où tu parles ! " : Un peu d'histoire de l'IA

✔ **Pour comprendre où l'on en est**

Une chronologie : 1943 est une date de " début "

◆ **1943** : proposition d'un modèle pour des **neurones artificiels** en partant (Mcculloch & Pitts) :

- des études des fonctions des neurones dans le cerveau
- de l'analyse formelle de la logique propositionnelle (Russel)
- de la théorie de calcul de Turing
- Programme de jeu d'échec (1950)
- 1er réseau de (40) neurones artificiels (SNARC) en 1951
- **GPS** : le 1er programme qui "pensait comme l'homme" (méthode Means-Ends) en 1952

◆ 1er Workshop en IA (proposition du nom IA) ➟ premier D.A.T. en 1956

◆ La langage **Lisp** (McCarthy , MIT) en 1958

◆ Recherches à Stanford, IBM, MIT, CMU (carnegy Melon)

◆ 1963 : **Micro World** : le monde de cubes, le dessin géométrique 2D et la reconnaissance de formes,

- **SIR** (semantics information retrivial) = TLN sur un sous ensemble de l'Anglais,
- **ANALOGY** : calcul & reconnaissance de formes géométriques par analogie (pour planifier le déplacement de Cubes par un Robot)

◆ 1965 : Méthode de résolution de **Robinson** : le 1er D.A.T. de la logique du 1er ordre.

◆ 1967 : **STUDENT** : un solveur Algébrique

◆ **Bloc World** : Huffman 1971 améliore les méthodes

◆ **Vision, propagation de contraintes** : Waltz 1975

◆ Théorie **d'apprentissage** de Winston 1970

◆ **TLN** : Winograd 1972

◆ **Planner** : Fehlman 1974

◆ Divers Programme pour le jeu d'échec

◆ ELIZA : Symbolique Syntactic

◆ Traduction automatique (des papiers scientifiques) Russes en Anglais (avec peu de résultat) .

Constat ➟ ***Les programmes marchent bien avec le Micro World***

(peu de faits) , peu d'états

Mais ➟ ***Cas des problèmes NP-Complets ?***

insuffisance des ressources?!

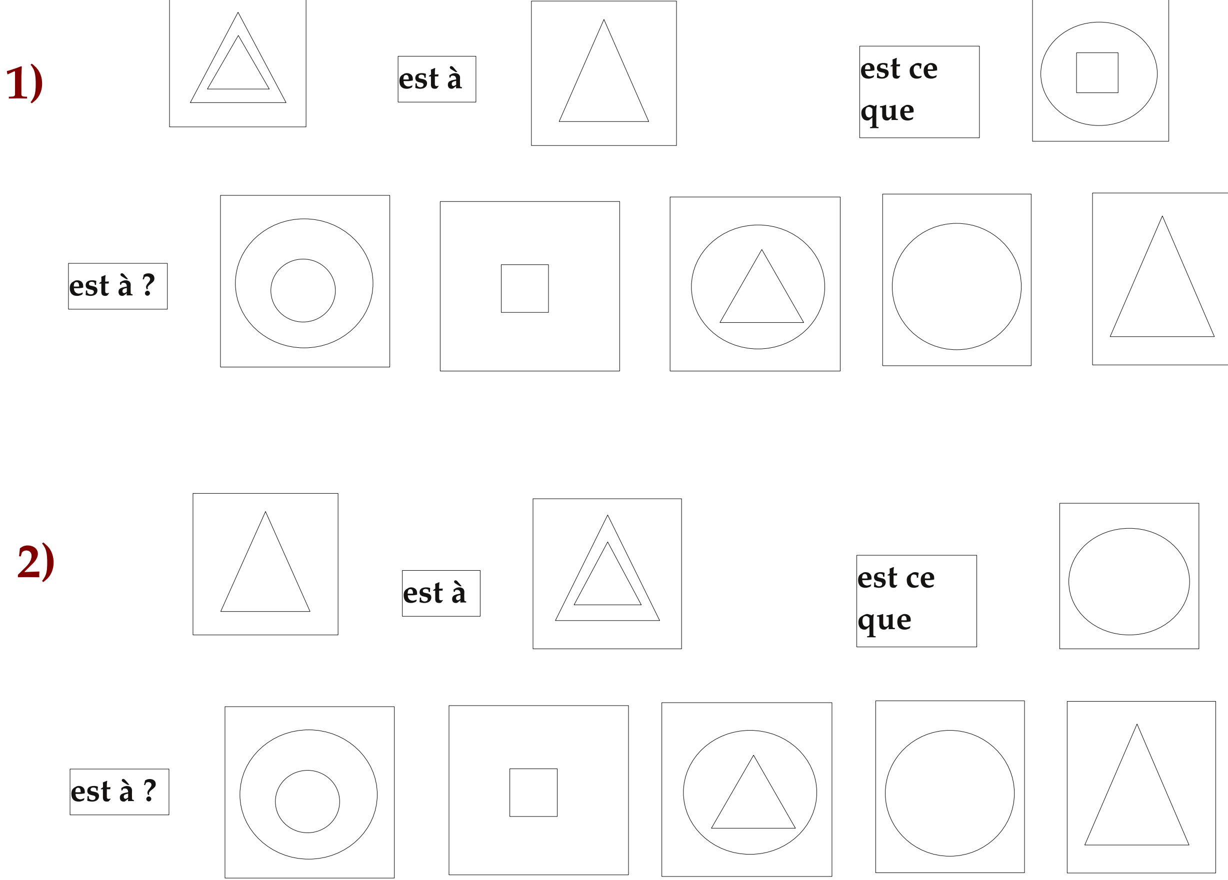
Constat d'échec (anecdotique) : dans la traduction automatique de phrases.

➟ La phrase **"l'esprit est fort mais la chaire est faible"**
a été traduite littéralement en : **"la Vodka est bonne mais la viande est daubée" !**

➟ la difficulté de contexte et tout ce dont il faut tenir compte.
L'avènement des "Algorithmes Génétiques" (Friedberg) et les R.N. (l'algorithme "back-propagation " existait depuis 1969 par Bryson) n'ont pas amélioré les résultats.

➟ Le **gouvernement US** annule les budgets des projets de traduction automatique (TLN).
➟ Le **gouvernement UK** a aussi abandonné le soutien aux projets IA en 1973.
➟ Quelques années de traversée du désert pour les travaux en IA ...

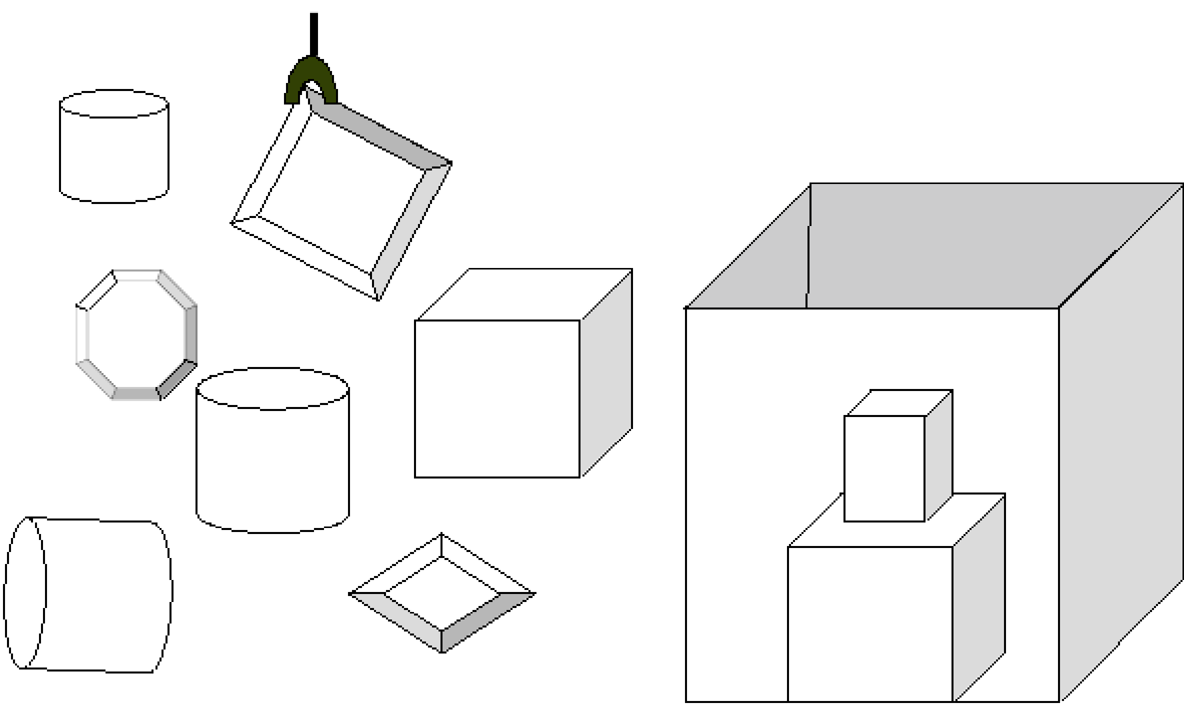
V.2- Exemple de question traitées par le système ANALOGY



V.3- Exemple de problème résolu dans le système SHDRLU (monde des blocs)

La commande (réalisée) :

Trouver un bloc plus grand que celui tenu (par la pince) et le mettre dans la grande boîte.



V.4- Importance des bases de connaissances

On ne peut pas traiter **tous problèmes** par une même méthode (DAT)

➟ Il faut de grosses bases de **connaissances** + des **Heuristiques**

➟ Confirmation par le projet **DENDRAL** (Un des premiers systèmes experts) **:**

Développé à Stanford par Feigenbaum en 1970s, traite le problème d'inférence de structures moléculaires à partir des informations fournies par un spectromètre de masse.

➟ Projet **HPP** : "Heuristique Programming Project" (Feigenbaum)

➟ Création de **MYCIN** pour le diagnostic des infections sanguines.

MYCIN: 450 règles, fonctionnait comme un expert (mieux que les docteurs fraîchement diplômés).

MYCIN avait deux différences majeures avec DENDRAL :

1- Contrairement à DENDRAL, MYCIN n'avait pas de modèle théorique général pour décrire des règles.

➟ Ses règles devaient être produites par des vrais experts.

2- Les règles devaient refléter *l'incertitude* associée aux connaissances médicales.

Leçon : Importance des connaissances et de leurs représentations *(vs. les méthodes générales)*

***Leçon : Importance des connaissances et de leurs représentations** (vs. les méthodes générales)*

Quelques expériences (mettant en oevre ce principe) :

- **PROSPECTOR** : Exploration de sites géologiques, basée sur un raisonnement probabiliste.
- **SHDRLU** (dans le domaine de TLN) a utilisé ces mêmes conclusions (Winograd).
- **LUNAR** : (Wood, 1973) : un système de question / réponse en géologie, en particulier les rochers de la Lune (mission APOLLO).

La forte demande en représentation de connaissances a donné lieu à :

- Prolog en Europe
- PLANNER en US et
- FRAMES (Minsky-75) :

  permettant de regrouper des connaissances sur les objets (du domaine du problème) ainsi des types d'événements.

V.5- Évolution : l'IA devient une industrie (1980 - ...)

● Le premier système expert (**R1)** de DEC pour configurer les ordinateurs, leur conception et montage a permis d'économiser 40 millions $ /an à DEC.

● 1981 : les Japonais ont lancé le projet **5e génération**.

Un projet sur 10 ans à base de Prolog.

Idées : des machines d'inférence (1000.000 inférences/Sec).

Un des buts du projet était le TLN.

➟ Ce projet a relancé les budgets aux US et UK (pour contrer les Japonais ☺).

➟ A démarré beaucoup de travaux sur la *Vision*, *Robotique*, machine Lisp, Chip dédié, IHM, ...

➟ En parallèle, le chiffre d'affaires de l'industrie est passé de quelques $millions en 1980 à 2 billions en 1988.

V.6- Les domaines qui contribuent à l'IA

- La **Philosophie** (et les sciences cognitives) a permis de considérer l'idée que le cerveau est comme une machine qui fonctionne avec des informations encodées dans un langage interne.

- La **Psychologie** : idée que l'homme et les animaux sont comme des machines de traitement d'information (*Information Processing Machines)*.

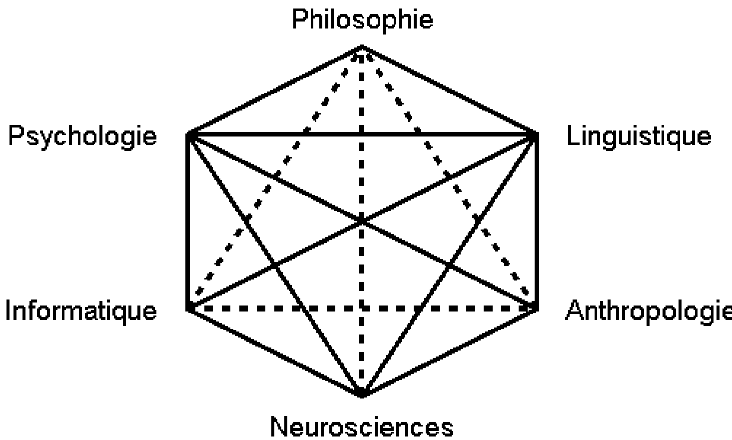
- Les linguistes ont montré que l'usage de la langue rentrait dans ce modèle.

- Les **Mathématiques** ont apporté des outils : logique (faits certains, incertains), probabilité, logique modale, ...

Les **informaticiens** ont donné les moyens de mettre l'IA en pratique.

☞ Une remarque : l'homme = fonctionnel (cerveau primitif, réflexes), mental (décision), esprituel (& conscience)

La complexité de l'intelligence humaine : l'hexagramme cognitiviste :



Cerveau : 20 milliards de neurones (= l'objet le plus complexe de l'Univers).

Les 3 cerveaux (évolution du cerveau) :

3 types de cerveaux ("cerveau reptilien" ➟ le "cerveau mammalien" ➟ "cerveau humain") :

- Le cerveau reptilien : tronc cérébral comprend le bulbe rachidien et le mésencéphale (comme les reptiles). Structures primitives, responsable des instincts et des réflexes innés.
- Le cerveau mammalien (système limbique). Siège des émotions (comme la plupart des mammifères).
- Le cerveau humain (hémisphères cérébraux ou néo-cortex), particulièrement développés chez l'Homme et les Primates. Regroupe les trois-quarts des neurones de l'organisme, le néo-cortex est le siège de l'intellect.

➟ Efforts de l'IA sur le cerveau reptilien : cf. Robots.

V.6.a- Les deux grandes points de vu

☞ 2 approches : *Thinking & Behavior* ➟ modèle **humain** ou modèle **idéal (rationnel)**

1. des systèmes qui **pensent** comme les humains (comportement *rationnel* minimisé, spirituel maximisé)
2. des systèmes qui **pensent** rationnellement

Penser comme les Humains	**Penser Rationnellement**
- Concevoir des ordinateurs dotés d'un **esprit** au sens le plus littéral. - Capable (comme les humains) d'activités telles que : la prise de **décision**, **apprentissage**, **résolution de problèmes,** etc.	- Etude des *facultés mentales* grâce à des modèles Informatiques - Etude des moyens informatiques qui rendent possibles la **perception**, le **raisonnement** et **l'action**.
Agir comme des humains	**Agir rationnellement**
- Des fonctions qui exigent de **l'intelligence** - Accomplissement des choses pour lesquelles on a <u>encore</u> recours à l'humain : que les robots ne peuvent faire.	- Etude et conception d'**agents intelligents**. - **Comportement " intelligent "** dans des artefacts

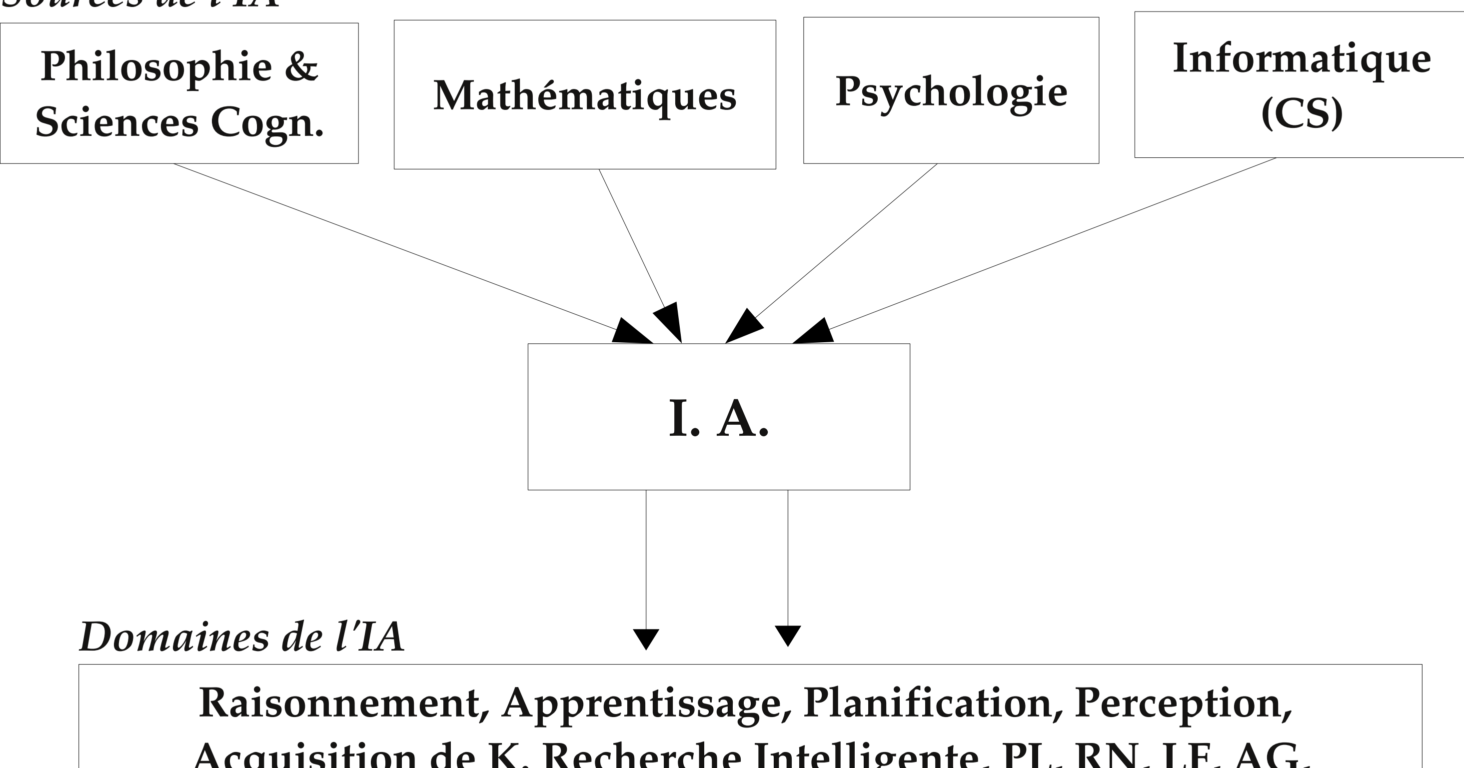
Peut-on faire un parallèle en médecine ? <u>patient</u> **ou** <u>machine.</u>

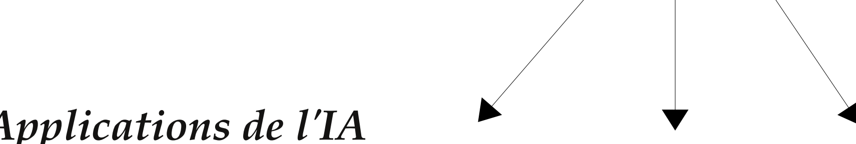
➡ Comment doit-on nous annoncer un cancer ? Comme si l'on a coulé <u>une bielle</u>.

Question (méta-physique) :

le degré de l'intelligence de l'homme peut-il lui permettre de comprendre sa propre intelligence ?

V.7- Résumé du cadre de l'IA





Jeux, DAT, TLN, Parole, Compréhension d'Images, Vision, Robotique & Navigation (mobile), S.E., Scheduling, etc...

VI- L'expérience MYCIN

Un tournant (les leçons des systèmes experts) : 1991

Système expert en diagnostic médical. L'expert soumet les symptômes à un système expert en médecine et demande un dia gnostic.

Étonné de la réponse du système, l'expert demande pourquoi ?

➡ Le système lui répond en traçant sa décision. L'expert accepte ou non

Mycin Utilise règles et **méta règles** (manipulation des connaissances) :

➡ **Règle** : ***si a>b & b>c alors a>c.***

➡ **Méta règle** : ***si R transitive & si (a R b) & (b R c) alors (a R c)***

Une méta règle peut aussi donner des indications sur l'application d'autres règles.

MYCIN est une illustration de l'importance de la bonne connaissance et de sa représentation par rapport aux mécanismes d'inférence.

Les leçons tirées de cette expérience :

➟ Peu importe le mécanisme d'inférence; l'essentiel est de savoir représenter les informations (mêmes complexes) .

➟ **But** : Présenter

- Une sémantique claire des données ;
- Des algorithmes de manipulation de ces connaissances;
- Des compromis entre la capacité d'expression et l'efficacité;
- Trouver un langage efficace pour donner des réponses rapides pour la majorité des questions ;

✔ Le "cas général" est plus important que les cas limites ;

✔On ne peut pas résoudre les cas limites "incalculables" (NP-hard, NP-complet) mais ces cas là se présentent rarement.

- La tendance actuelle en IA : la représentation de connaissances.

➟ **La représentation des connaissances et raisonnement**

VII- Outils et problèmes traités

De nos jours, il existe plusieurs cadres (méthodes, paradigmes) pour résoudre les problèmes complexes :

- Modélisation Mathématique et statistique (quand c'est possible)
- Systèmes experts (présence d'expert)
- Apprentissage et DM <=> méthodes statistiques / algorithmiques + systèmes de règles

VII.1- Différents paradigmes

✓ Modèle **probabiliste**, **Belief Network et** représentation de faits incertains.

➟ Systèmes experts qui fonctionnent rationnellement à base de la "théorie de la décision" (en n'essayant pas d'imiter les humains)

✓ **Data Mining, Ontologie, Web Sémantique, ...**

En :

Raisonnement, Perception, Apprentissage, Recherche intelligente, Acquisition de connaissances, etc.

VII.2- Divers domaines et applications

✓ **Speech &Hand Written Characters Recognition** :

à base de "Hidden Markov Model" pour la re-formulation Mathématique des règles / faits dans les systèmes.

✓ **Planification** : Planning dans les entreprises, usines, missions spatiales.

✓ **Robotique, Vision, Apprentissage, Représentation de connaissances.**

➟ Système **SOAR** : agents intelligents + capteurs, ... en robotique et en Vision (sur les chaînes de montage)

✓ **Jeux, Médecine**

Prévision et Prédiction diverses (finances, météo, bourse,...), Planification,

Génétique, Image et Imagerie médicale, Optimisations, Reconnaissance Vocale, TLN, ...

Navigation, Génétique , etc....

VII.3- Des exemples de réalisation (outre l'état de l'art)

1- *"Je veux aller de Strasbourg à Bézier"* dit l'utilisateur dans le microphone.

Le système **PEGASUS** demande la date du voyage.

L'utilisateur dit qu'il veut partir le 20 octobre, non-stop, le moins cher pour revenir samedi.

PEGASUS lui réserve sa place et lui fait gagner 194€ par rapport au tarif normal.

Même si la reconnaissance vocale a un taux de 1/10 mots d'erreur, il se corrige en s'appuyant sur le contexte.

2- Caméra de **surveillance de trafic** (1994)

Signal un accident, téléphone à la Police et aux dépanneurs. ...

3- Divers dispositifs de sécurité (reconnaissance, ...)

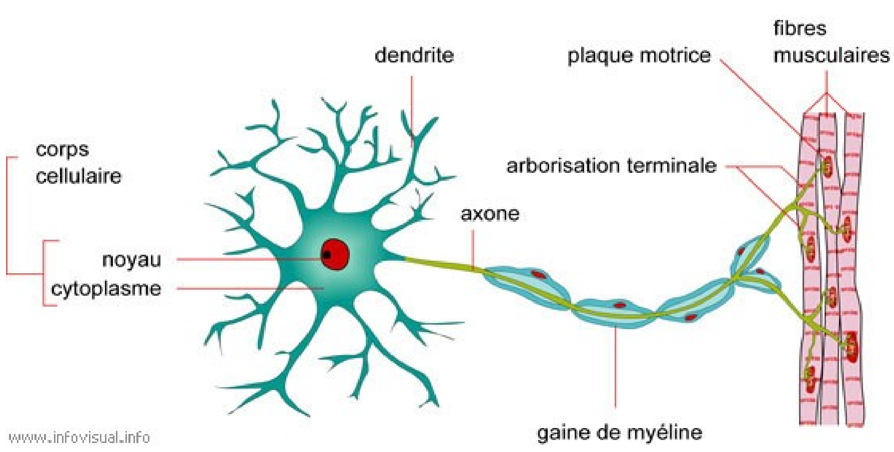
4- Systèmes de **décision financières, Robots intelligents**, **reconnaissance faciale,** système **d'extraction de connaissances** sophistiqués, systèmes **géologiques**, système de **poursuite de mouvement**, etc.

➟ 2008 : Robot bipède Japonnais capable de courir, se tenir accroupi,

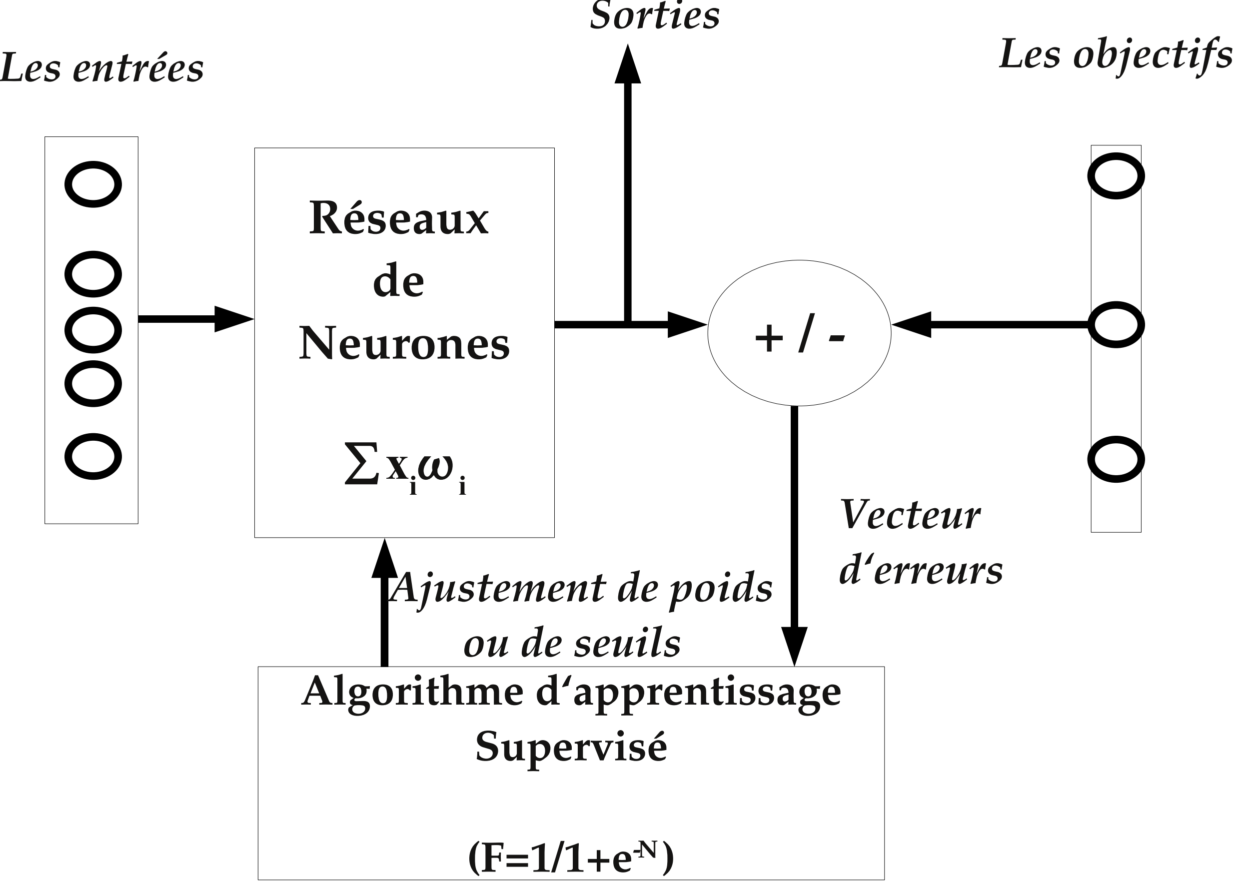
VII.4- Black Box : quelques outils

Application massive des **Réseaux de Neurones (1986) & DM**

- Application de l'algorithme de *Back Propagation* aux projets sur l'apprentissage.
- Relance de la "Distributed Prallèle Processing" pour les R.N. ➡ résultats encourageants.

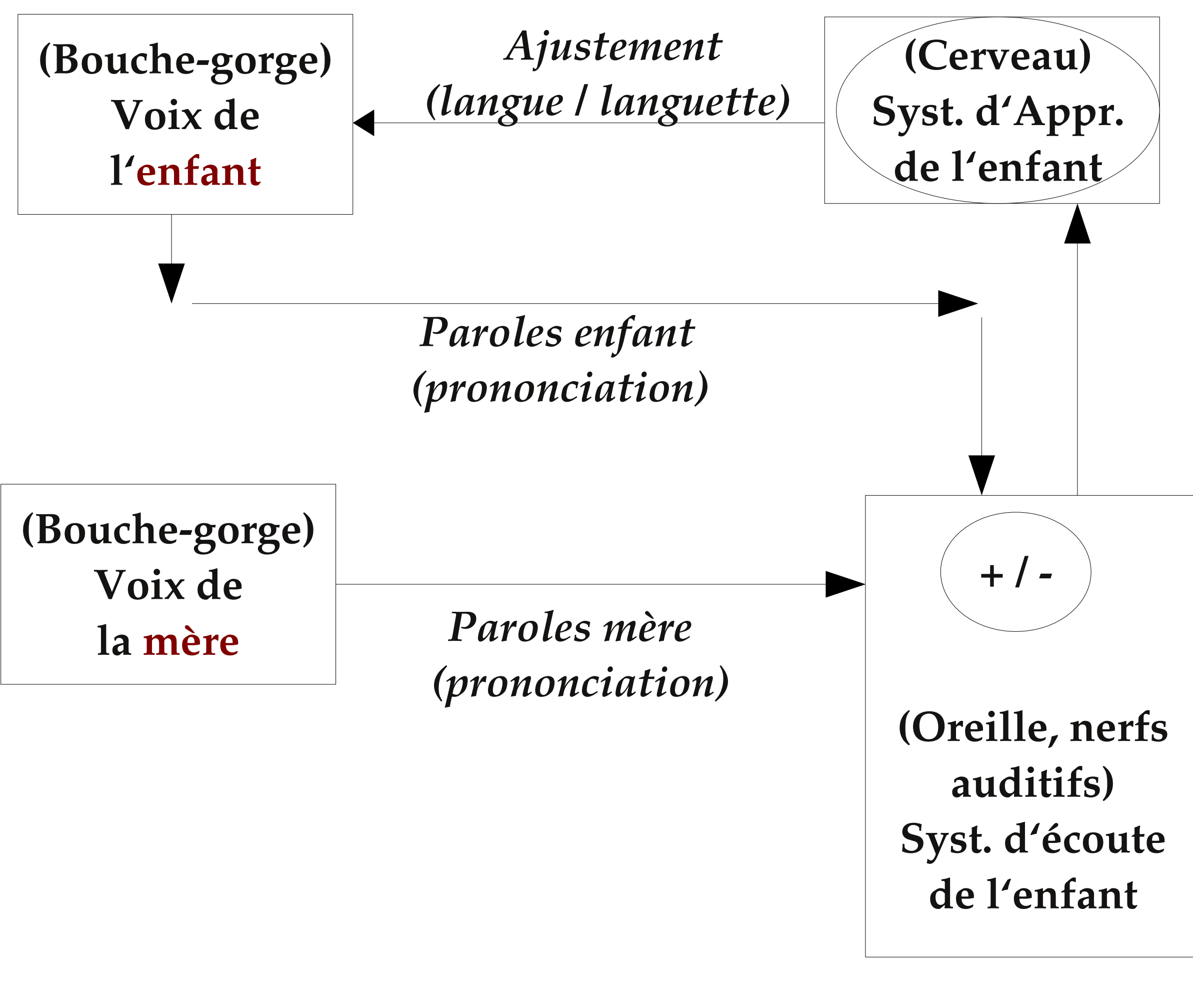


VII.4.a- Réseau de Neurones : exemple d'architecture



VII.4.b- Réseaux de Neurones & l'apprentissage de la prononciation

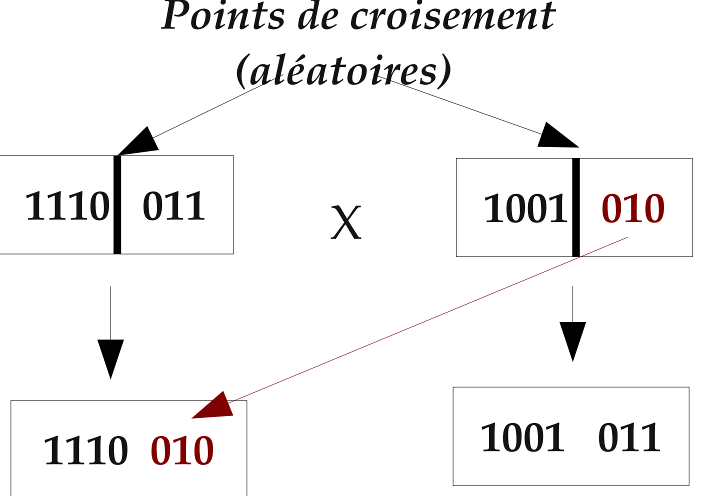
Apprentissage de la prononciation chez l'enfant (utilisé en Robotique)



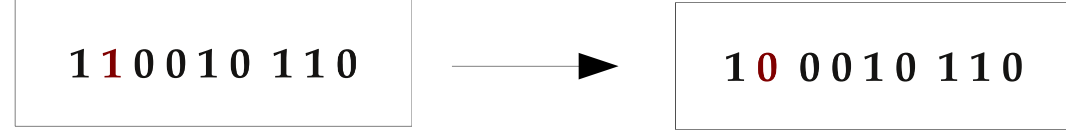
VII.4.c- Algorithmes Génétiques (un exemple)

Deux opérateurs : Croisement & mutation (outre la *sélection*)

- Chromosomes parents et les chromosomes obtenus par croisement



- Chromosomes obtenus par mutation aléatoire :



➡ Le 2e bit à gauche (aléatoirement) choisi a été complémenté.

VII.4.d- Vision artificielle (schéma et exemple)

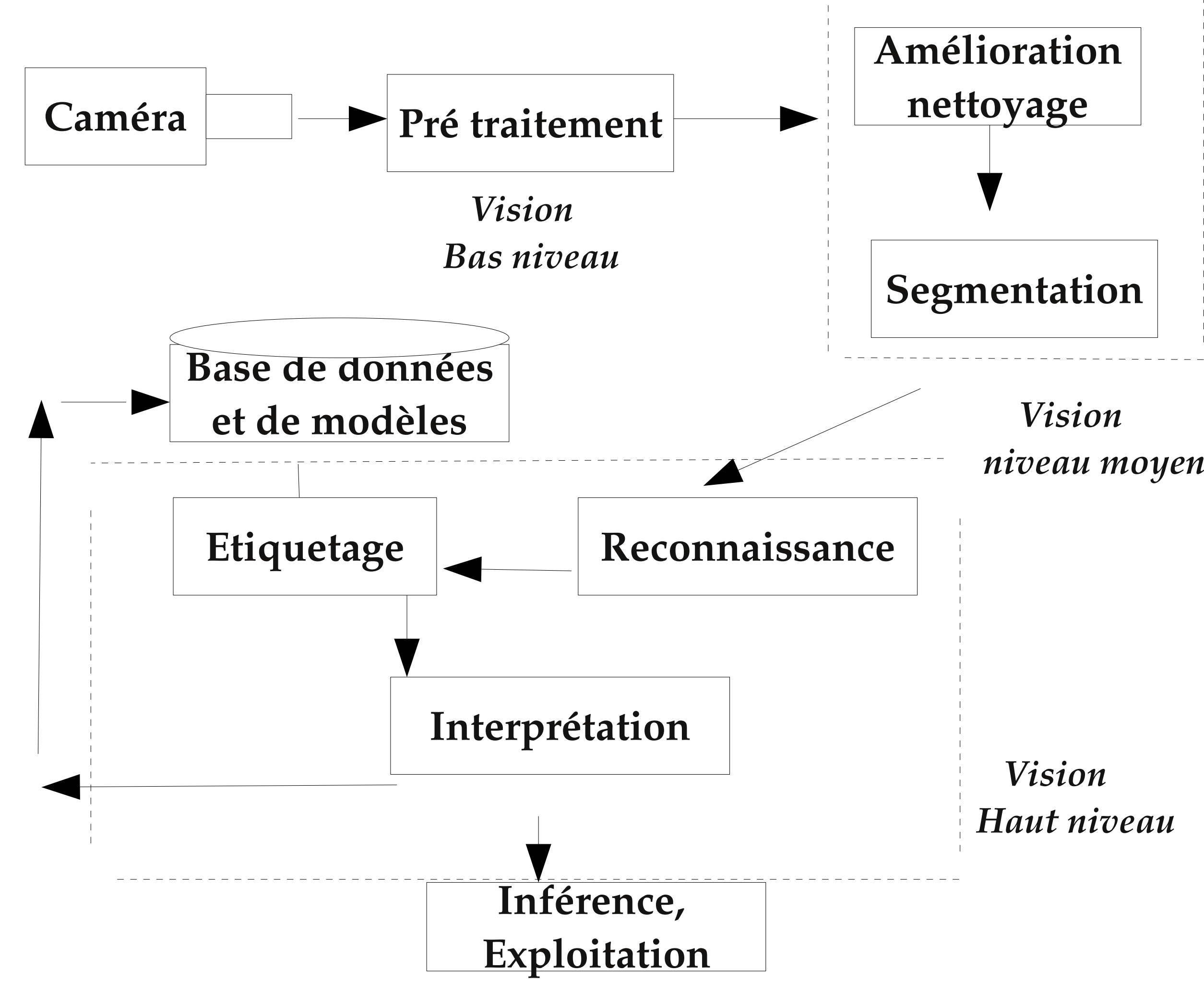


Schéma d'analyse et de reconnaissance applicable en Reconnaissance de Formes (RF) en général..

VII.5- Open Box : Système Experts (exemple de règles)

Base de connaissance :

(1) Si temps chaud (chaleur)

& nuages & précipitation

Alors il pleuvra.

(2) Si il pleut

Alors éboulements sur routes (de pays chaud)

Base de Données :

nuages.

Inférence :

éboulement ....

Données contextuelles : on est dans un pays chaud (donne lieu à une méta règle)

VIII- Les langages de représentation de connaissances

- Formules logiques (Prolog, ...)
- Réseaux (réseaux sémantiques, graphes conceptuels)
- Objets (scripts, frames)
- Procédures (systèmes de production, Lisp, ..)

VIII.1- Logique

- A base de relations : Rel(A,B), ...
- Exemple : *Hélène donne un livre à jean*

∃p,q,r : personne(p) ∧ nom(p, Jean) ∧ personne(q) ∧ nom(q, Hélène) ∧ livre(r) ∧ donne(q,r,p).

- **Prolog**

VIII.2- Langages à base de réseaux

- Pour définir le sens des concepts par des relations aux autres concepts.
- Une variante des langages logiques.
- Un lien *Rel* entre deux nœuds A et B est un autre moyen de spécifier *Rel(A,B)*.

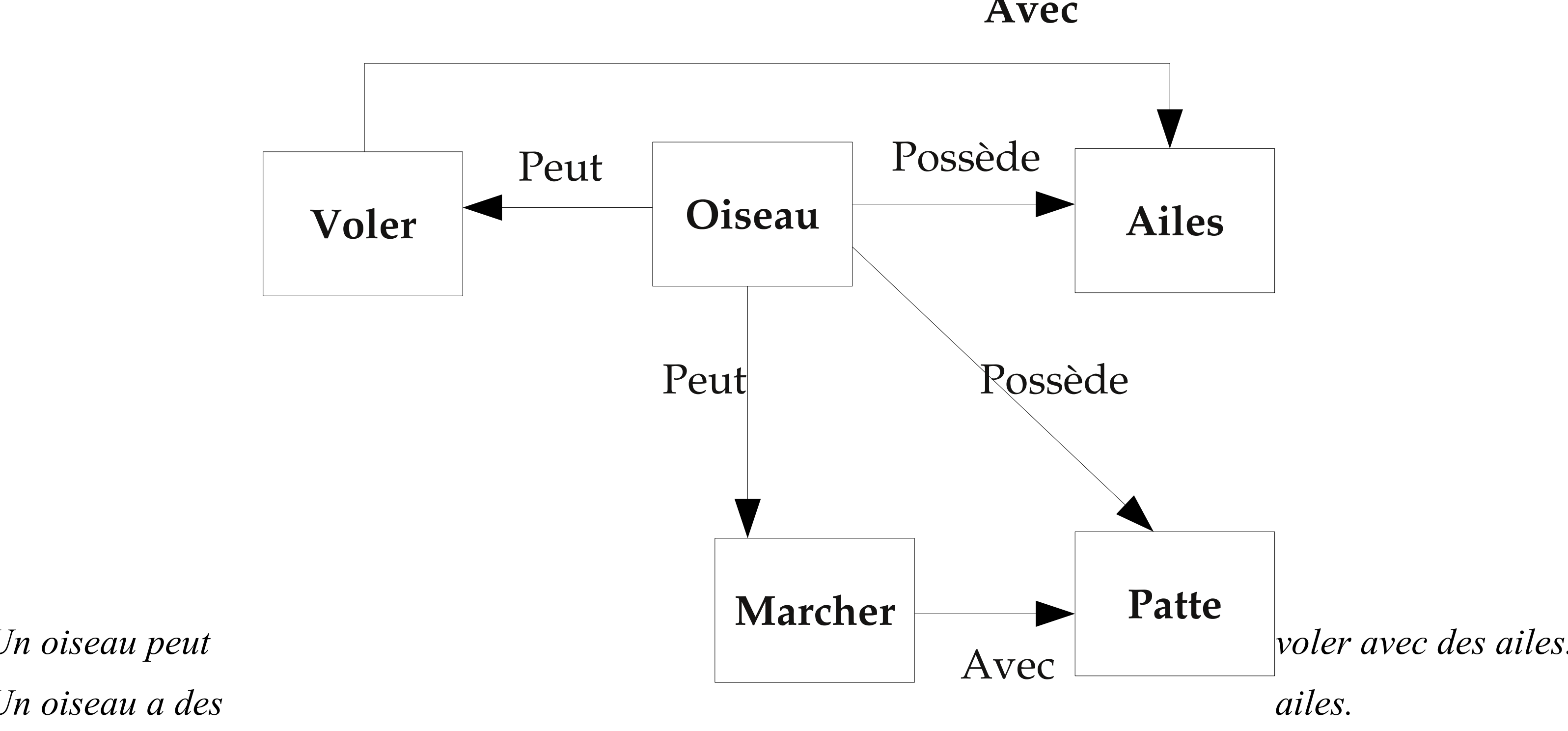
- Les liens jouent un rôle plus opérationnel :

  l'inférence est effectuée en traversant ces liens.

  ✔ *Rel(A,B)* est vraie et en plus il expliquera comment la base de connaissances doit être recherchée (attachements procédu raux).

VIII.2.a- Un exemple

décrit la connaissance dans une approche structurée **:**



Un oiseau peut voler avec des ailes.

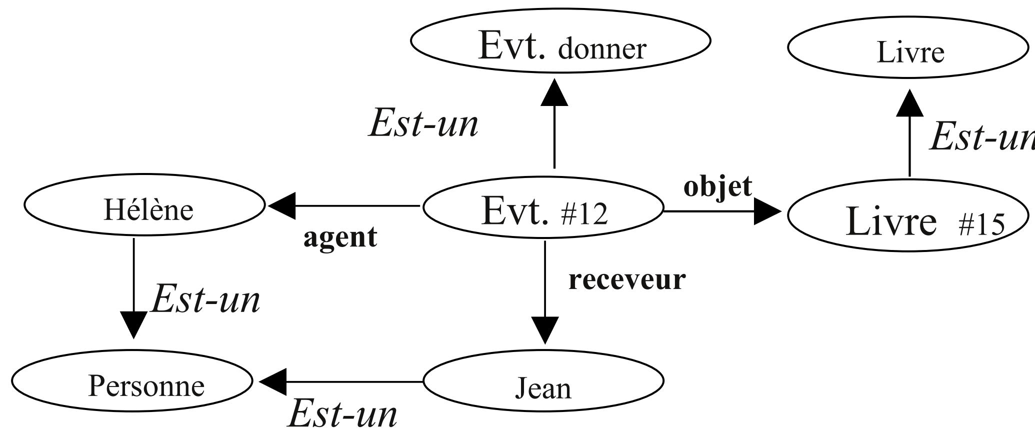
Un oiseau a des ailes.

Un oiseau a des pattes.

Un oiseau peut marcher avec ses pattes.

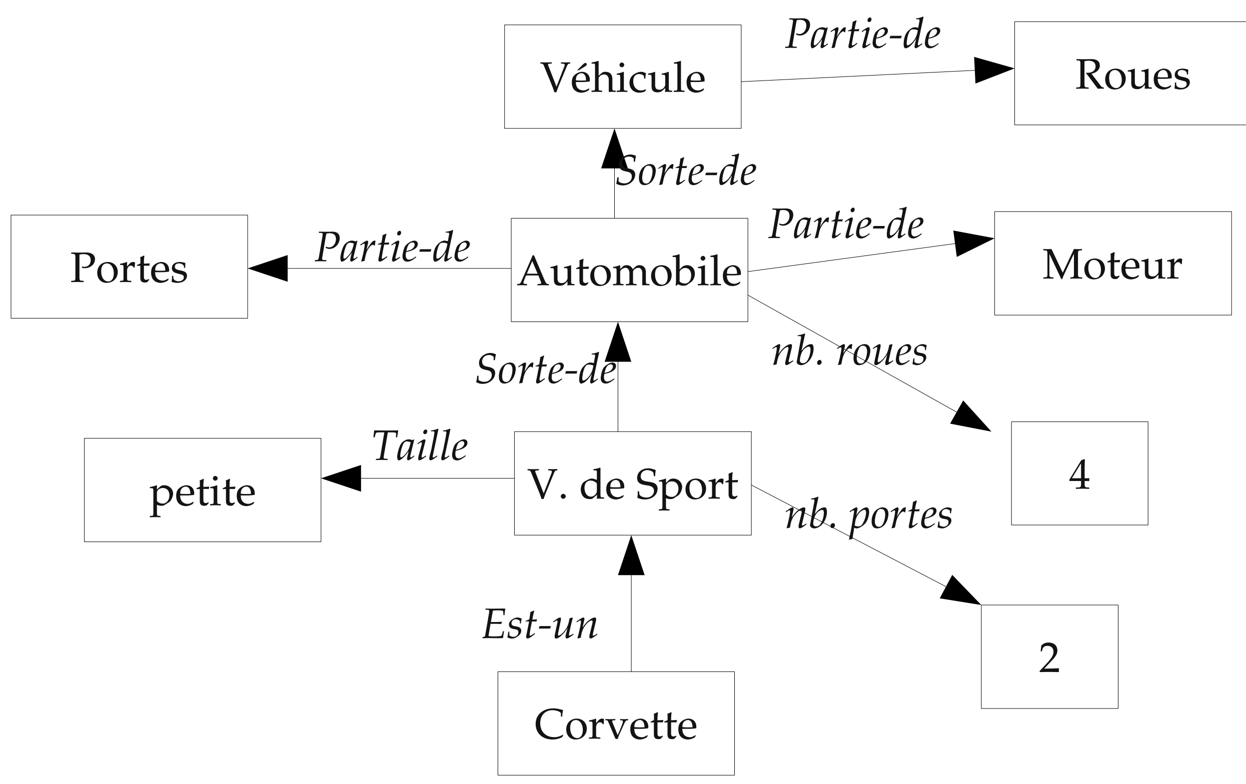
VIII.2.b- Un exemple plus général

Hélène donne un livre à jean



On peut utiliser les relations : *partie-de, sorte-de, instance (est-un), composé-de, ..*

VIII.2.c- Un autre exemple



VIII.2.d- Remarques sur les réseaux sémantiques

- Dans un réseau sémantique, on utilise la technique appelée "*spreading activation*" pour trouver comment deux concepts (ou nœuds) sont reliés.

- Exemple de système : **KnU** d'IBM (knowledge Utility)
  Utilisé sur le site d'IBM (www.ibm,aqui,com)

Remarque :

- Il existe des langage de représentation de connaissances (et de leur manipulation / conversion) :
  KIF : Knowledge Interchange Format (Gensereth & Filces 1992)

VIII.3- Objets et Frames

- Une variante des calculs des prédicats.
- Un exemple typique dans les langages "slot-filler-frame" :

(une personne

(nom=jean)

(age = 25))

➟ équivalent à : $\exists$ ***p*** *: personne(p)* $\wedge$ *nom(p, jean)* $\wedge$ *age(p,25)*

Remarques :

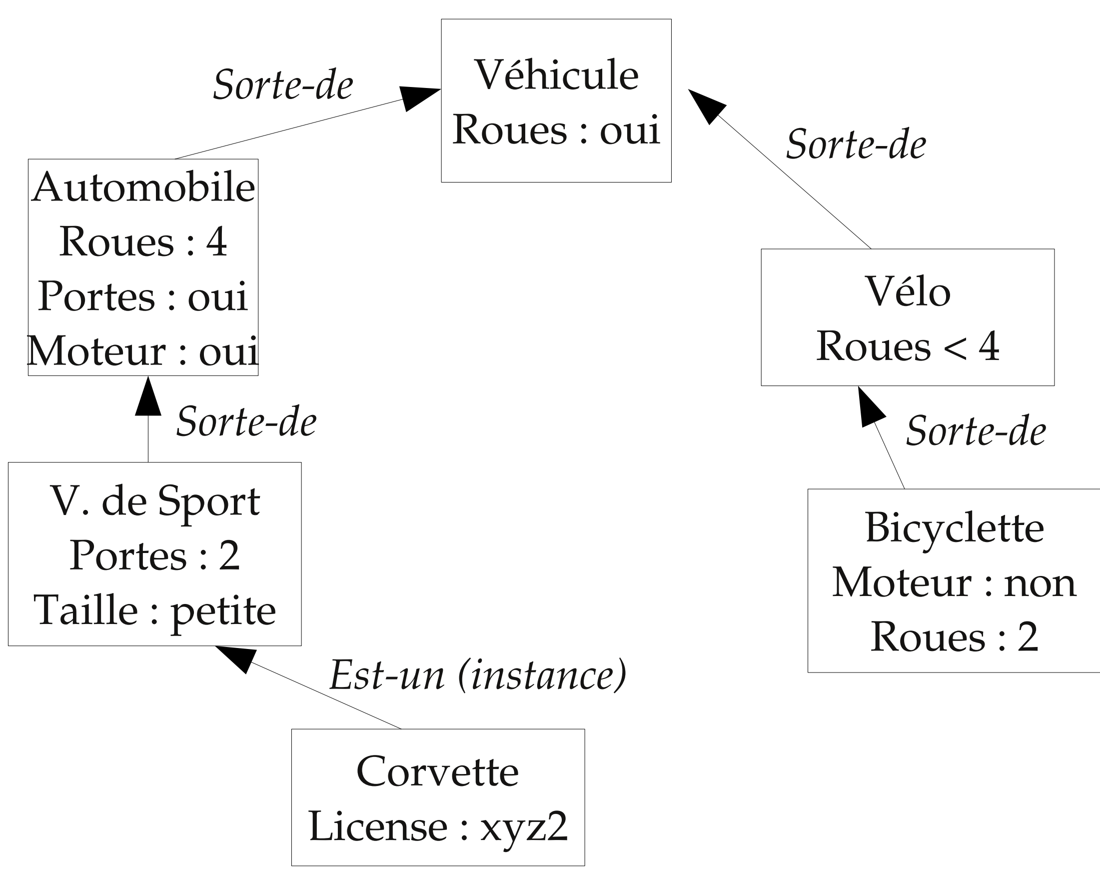
- Les *Frames* et les *Réseaux sémantiques* sont **très proches** de la logique des prédicats.

  Russel & Norvig (1995) ont proposé un algorithme pour les transformer en logique du 1er ordre.

- **La différence principale entre ces deux formalismes = la syntaxe.**

VIII.3.a- Un exemple de Frame

La notion de classe et d'instance plus forte que dans les réseaux sémantiques



VIII.4- Discussion : avantage aux frames !

- Formalisme très équivalent aux réseaux sémantiques
  - ✔ **Un réseau sémantique ≃ une collection de frames**
  - ✔Aussi, les **frames** ≃ les réseaux sémantiques généralisés où la notion de classe et d'instance est plus forte.
- Les frames contiennent des *slots* (cases) et des valeurs de slots
- Peuvent décrire des instances et des classes (et autres relations);
- Peuvent avoir des procédures d'accès, **démons , réflexes , ...**

✔ Démons de maintien des contraintes (attachement procédural) activé avant, après, pendant une modification de valeur d'un slot : ***si-crée, si_détruit, si_modifié*, ...**

- Sont plus simples à comprendre mais sont moins expressifs.

✔ Parfois insuffisant : on ne peut pas dire :

- "le nom de la personne est Jean ou Jacques" ou
- "l'age n'est pas 30".

- Un frame est en général relié à un autre par des relations *partie-de, Est-un, etc*.

VIII.5- Formalismes Procéduraux

Calculent des réponses sans une représentation explicite et indépendante des connaissances.

Exemple : Système de production, ... avec des règles de la forme

Si conditions Alors Actions

• Langages de représentation de connaissances **hybrides** :

- KL-ONE : logique+objets
- LIFE : relation, fonction, objets, héritage.
- L&O (Logic & Object),
-- .....

• Des langages de frame utilisent des attachement procéduraux pour Compléter et calculer des expressions non exprimées par les frames.

IX- Éléments et caractéristiques d'une représentation

Caractéristiques d'une (bonne) représentation de connaissances :

• Objets et relations explicites ;

• Possibilité d'exprimer les contraintes naturelles ;

• Suppression des détails inutiles ;

• Représentation transparente , complète et concise ;

• La manipulation est rapide;

• La représentation est **calculable** ;

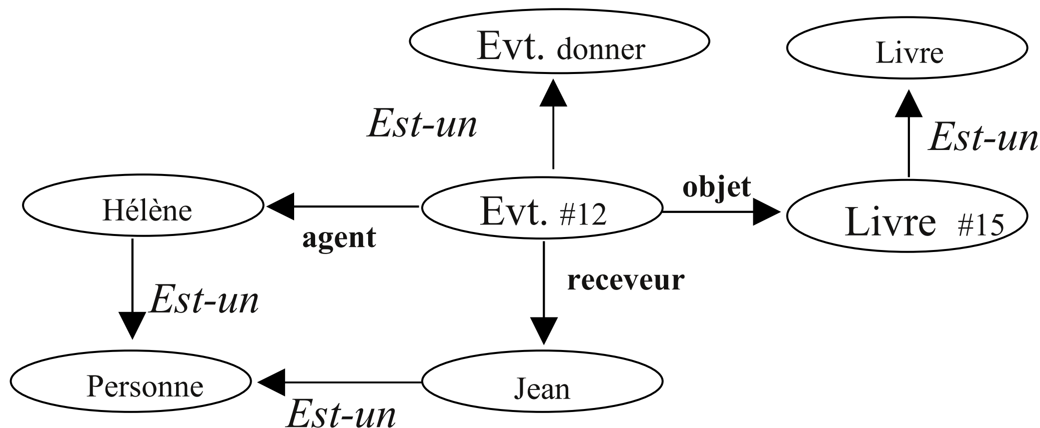
Eléments fondamentaux (constituants) d'une représentation

- **lexicale** : symboles, vocabulaire
- **structure** : les règles de construction et d'arrangement
- **procédurale** : créations, modifications, question-réponses, calcul.
- L'héritage et les démons forment une sémantique (signification, sens) procédurale.

IX.1- Exemple

Pour l'exemple du réseau *Hélène donne un livre à jean*

- <u>lexicale</u> : nœuds, liens et liens spécifiques
- <u>structure</u> : un lien reliant deux nœuds
- <u>procédurale</u> : création de nœud, lien reliant deux nœuds, nœuds d'un lien, lien de deux nœuds, liens d'un nœud, démons, ...
- <u>sémantique</u> : les nœuds et le liens spécifient les entités de l'application.



X- Autres formalismes (plus récents)

Ontologie (et Ingénierie Ontologique) :

➔ Une représentation Hiérarchique du monde

Exemple : vache / mammifère / animal

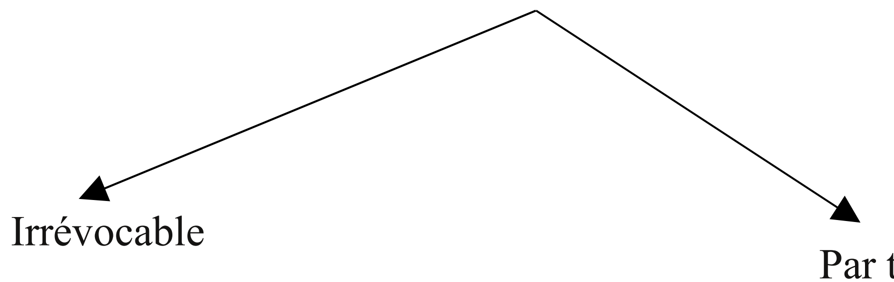
- Englobe les
  - Objets : ensembles, Nombres, Objets Descriptifs (Mesure, énoncé, etc)
  - Évènements : Intervalle, Lieux, Objets Descriptifs (Entité, Matière, etc), Processus

- Autre exemple :

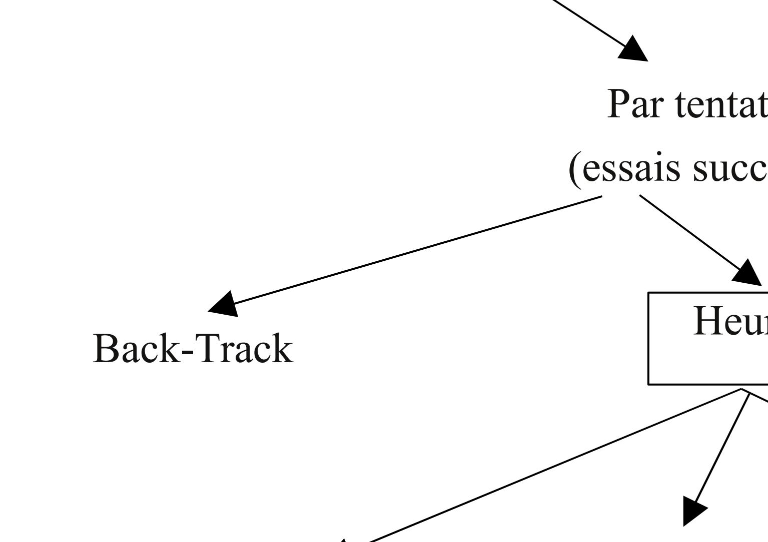
  un **Humain** est de la catégorie "Agent & Animaux", sous catégorie d' "Entité", sous "Objets-Descriptifs"

XI- Recherche : classement de quelques stratégies

Parcours de graphes



Par tentatives
(essais successifs)



Heuristique

BF-BS-HC

B & B
A- A*

Théorie
des Jeux

Min-Max

Alpha-
Beta

BF : Best First

BS : Beam Search

HC : Hill Climbing

B & B : Branch and Bound

Remarque :

dans ce polycopié, nous détaillerons la plupart de ces stratégies.

➡Le langage utilisé : **Prolog**

XII- Exemples en Prolog (LP et CLP)

XII.1- Exemples simples en Prolog

```
entier(0).
entier(succ(N)) :- entier(N).
```

```
homme(jean).        homme(pierre).        homme(paul).
mortel(X) :- homme(X).
```

```
pere(jean, pierre).        pere(jean, marie).
pere(pierre, paul).        pere(jecques, jean).
mere(anne, jean).          mere(kate, marie).        mere(helene, paul).

frere_ou_soeur(X,Y) :- pere(Z,X), pere(Z,Y), X \= Y . % voir la relation @>
parents(X,Y) :- pere(X,Y) ; mere(X,Y).
ancetre(A, P) :- parents(A, P).
ancetre(A, P) :- parents(A, Par), ancetre(Par, P).
```

XII.2- Exemples utilisant les contraintes (CLP)

XII.2.a- Puzzle logique (simple)

TWO *+*

TWO

=   FOUR   *(Les lettres I= [T,W,O,F,U,R] sont des chiffres tous différents.)*

Principe de la solution :

- Soit L = [T,W,O,F,U,R]   la liste des variables dont le **domaine** est 0..9
- On pose : tous les éléments de L deux à deux différents.
- On pose ensuite : **O+O=R+10*X1, X1+W+W=U+10*X2, X2+T+T=O+10*X3, X3=F**

Ou   **2*(100*T+10*W+O) = 1000*F+100*O+10*U+R**

- Finalement, si le système a plus d'une solution unique (proposé directement), on énumère des valeurs.
- Exemples de solution : {l = <1,3,2,0,6,4>}, et {l = <7,3,4,1,6,8>}, etc.

XII.2.b- Factorielle réversible

Factorielle (avec des contraintes) réversible.

```
fact(0, 1).

fact(N, M) :-  N #> 0   , N1 #= N-1   , fact(N1, M1)  , M #= N * M1
```

Exemples de questions :

?- fact(3, X). => X=6

?- fact(Y, 24), ! . => X=4 % le cut (" ! ") à voir plus loin.

XII.2.c- Un équation simple

```
nbr_tetes_pattes(Lapins, Pigeons,Tetes, Pattes) :-

   Tetes  #= Lapins + Pigeons

   , Pattes #= 2 * Pigeons + 4 * Lapins.
```

?- nbr_tetes_pattes(Lapins, Pigeons, 6 , 10). ==> no.

?- nbr_tetes_pattes(Lapins, Pigeons,4 ,10). ==> Lapins = 1, Pigeons = 3

?- nbr_tetes_pattes(Lapins, Pigeons,Tetes, 10). ==>

Lapins = _#3(0..2), Pigeons = _#22(1..5) , Tetes = _#41(1..7)

XII.2.d- Amortissement (domaine des Réels)

Calcul de mensualité d'un prêt (réversible). Nécessite des calculs sur les réels.

Sous GProlog-RH

```
versement(Capital, Mois, Taux, Due, Mensualite) :-
   { Mois = 1,    Due = Capital + (Capital *Taux  - _Mensualite)}.


versement(Capital,  Mois,  Taux,  Due,  Mensualite) :-
     { Mois >= 1,   Mois1 = Mois -1 ,    Capital1 =  Capital*(1 + Taux) - _Mensualite},
      versement(Capital1, Mois1,  Taux,  Due,  Mensualite).
```

Des questions :

1) Pour 1000 Euros, remboursé en 1 mois, à 10% : <u>combien doit-on payer à la fin du mois ?</u>

```
versement(1000, 1, 0.10, 0,  M).          ➡    M = 1100.0
```

2) Pour 1000 Euros, remboursé en 12 mois, à 10% : <u>quelle mensualités ?</u>

```
versement(1000, 12,  0. 10, 0, M ).       ➡    M = 146.7633151
```

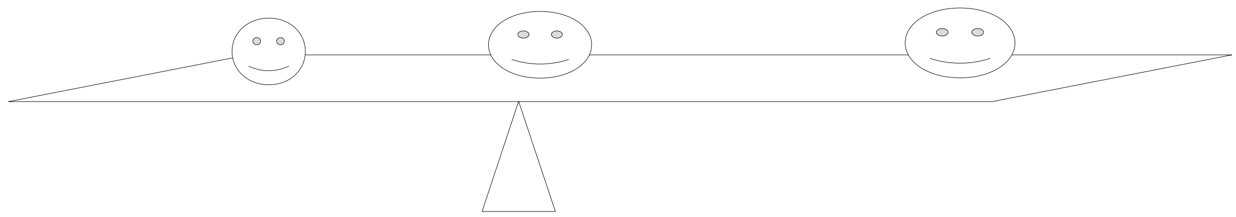
3) Remboursé en 12 mois, à 10%, 150 Euros par mois : <u>quelle était la capital de départ ?</u>

```
versement(X, 12, 0.10, 0, 150).           ➡    X = 1022.0537734
```

XII.2.e- Balance

- 3 enfants (Jean, Pierre et Paul) sont assis sur une planche de 10 mètres (avec des repères sur l'intervalle **-5 .. 5**).
- Jean pèse 36 kg, Pierre 32 et Paul 16.

Comment s'asseoir tel que la distance minimum entre chaque couple soit **supérieure à 2 mètres** et la planche s'équilibre?



- Variables (emplacements X,Y,Z) et Domaines : **-5 .. 5**
- Contraintes (autres) : **X*36 +Y* 32 + Z * 16 = 0**

|X - Y| > 2, |X – Z| > 2, |Y - Z| > 2

➡ {-4, 2, 5}, {-4, -4, 1} {-4, 5 -1} ...

NB : on accélère les choses si l'on impose X < 0 (casse les symétries)

XII.2.f- Pierres (exemple moins simple)

Une pierre de 40 kg se brise en 4 morceaux. Avec ces morceaux, on peut peser tous les objets de 1 a 40 kg.

Quel est le poids de ces morceaux ?

En Bprolog :

```
pierre(L, N) :-
    L = [P1, P2, P3, P4],                % La liste des 4 pierres : P1, P2, P3, P4
    L :: 1..40,                          % Pi à valeur dans 1..40
    P1+P2+P3+P4 #= 40,                   % Leur somme = 40
    P1 #=< P2,  P2 #=< P3, P3 #=< P4,    % La relation entre les 4 pierres (pour accélerer)
    tous_les_kilos_pesables(1,40, L),    % Que l'on puisse peser de 1..40 kg avec les 4 pierres
    labeling(L).
```

Tous les poids entiers entre A et B doivent être pesables.

```
tous_les_kilos_pesables(A, A, _) :- !.     % Tout est vérifié.
tous_les_kilos_pesables(A, B, L) :-
    peser(A, L),                           % peser A kg avec les 4 pierres
    A1 is A +1 ,                           % vérifier que A+1 kg est pesable.
    tous_les_kilos_pesables(A1, B, L).
```

Chaque poids peut etre d'un côté ou de l'autre de la balance (ou inutilise)

Envisager une disposition des 4 morceaux de pierres sur la balance de telle sorte que *A* kg soit "pesable " :

➞ Comment peser *A kg* à l'aide des 4 pierres P1,P2,P3,P4 ?

P_i : i = 1..4

- peut être sur le plateau gauche (B_i = -1 : sa valeur est soustraite du plateau droit),
- peut être sur le plateau gauche (B_i = +1 : sa valeur est ajoutée au plateau droit),
- peut être absente (non utilisée) (B_i = 0 : sa valeur =0).

```
peser(A, [P1, P2, P3, P4]) :-                   % Pour peser un poids donné A :
    L = [B1,B2,B3,B4],                          % Choisir les Bi tels que les 4 pierres disposés de part
    A #= B1*P1 + B2*P2 + B3*P3 + B4*P4,         % et d'autre de la balance fassent A kg.
    L :: [-1,0,1].                              % BI prend sa valeur dans -1 (à gche), 0 (absence) ou 1
```

Test :

```
pierre(L,40).
==> L = [1,3,9,27].
```

XII.3- Ingrédients d'un raisonnement en CSP (domaine fini)

Pour résoudre un problème *P(X1, ..., Xn)* :

1) Déclarer les domaines finis des variables du problème :

[X, Y, Z] ∈ 1..10,

2) Exprimer les contraintes **2*X + 3*Y + 2 < Z,**

3) Rechercher des solutions (énumération si nécessaire, *indomain/1* ou *fd_labeling/1*) :

indomain(X), indomain(Y), indomain(Z).

Sous Gprolog :

```
?- [X,Y,Z] : 1.. 10,    2*X + 3*Y + 2 #< Z,    enum([X,Y,Z]).
```

X = 1, Y=1, Z=8

X = 1, Y = 1, Z = 9

X = 1, Y = 1, Z = 10

X = 2, Y = 1, Z = 10

XII.4- Résolutions dans les environnements CSP actuels

- Traduction en contraintes plus basiques : 2*X + 3*Y + 2 #< Z, est traduit en contraintes plus simples
  - ➡ A base de ***X in R*** **:** *étant donné les domaines de* ***X,Y et Z,*** *procéder à un eréduction si* ***Z = X+Y.***
  - ➡ Les 3 tests de **consistances** + moteur **BT** (voir plus loin)
  - ➡ ....

- Résolution directe (sans simplification), par exemple à l'aide du Simplex particulier (cf. PIII et dif)

- Résultion à base d'intervalles (Réels)

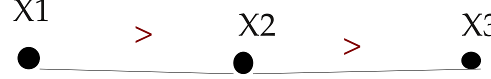
- Cas des booléen, réels, ensembles, ...

- Autres contraintes globales : ***au plus*** *K éléments d'une séquences doivent vérifier une relation R.*

XII.5- Techniques de Consistance (Nœud, Arc et Chemin)

Il est possible de construire un solveur (complet) avec ces consistances + BT.

Soit X = {X1,X2,X3}, D = {D1, D2, D3} = {1,2,3}, Z = {X1 > X2 > X3}



- **Node conscience** :

aucune simplification, les variables gardent leurs domaines.

Les contraintes des domaines sont vérifiées.

Exemple : **X in 1..3, X>3** n'est pas valide.

- **Arc conscience :**

Entre X1 et X2 => X1={2,3}, X2={1,2}. Puis indépendamment pour X2 et X3.

En Gprolog, on notera #> pour une consistance d'arc partiel et #>\# pour un arc consistance totale.

- **Path Consistance :**

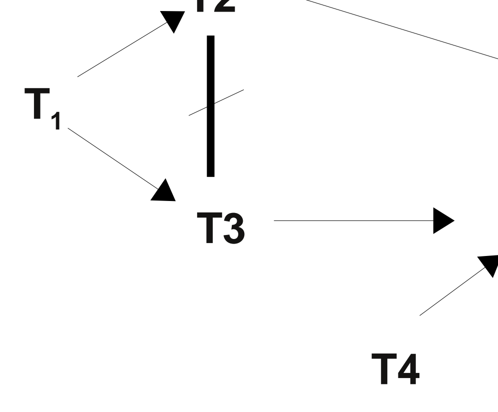
on peut simplifier en : X1=3, X2=2, X3=1

(Gprolog n'implante pas ce mécanisme couteux).

XII.5.a-

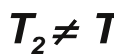
XII.5.b- Exemple : Planification de tâches

Exemple de résolution de contraintes (*Arc Consistency*)



$T_1 < T_2$

$T_1 < T_3$



$T_2 \neq T_3$

$T_2 < T_6$

$T_3 < T_5$

$T_4 < T_5$

$T_5 < T_6$

Le domaine (des tâches) : **{1, 2, 3, 4, 5}**

La propagation des contraintes :

Si l'on propage les contraintes (directement puis par transitivité = *Arc-consistancy* puis *Path Consistancy*), les variables conservent les possibilités suivantes :

T1 = {1,2}, T2 = {2,3,4}, T3={2,3}, T4={1,2,3}, T5={3,4} T6={4,5}

Si par exemple on pose **T1 = 2 :**

on obtiendra T1=2, T2=4, T3=3, T4={1,2,3}, T5=4, T6=5.

Consistance et simplification des contraintes :

Sous Gprolog :

```
fd_domain([T1,T2,T3,T4,T5,T6], 1,5),
T1 #<# T2, T1 #<# T3,   T2 #\=# T3,    T2 #<#T6,    T3 #<# T5,    T4 #<# T5,    T5 #<# T6.
```

T1 :: 1..2	T2 :: 2..4
T3 :: 2..3	T4 :: 1..3
T5 :: 3..4	T6 :: 4..5

Attention : ce ne sont pas les dates au plus tôt / tard

On pose T2 = 2 : cela déclenche d'autres simplifications

```
fd_domain([T1,T2,T3,T4,T5,T6], 1,5),    T1=2,
T1 #< T2, T1 #< T3,T2 #\= T3, T2 #<T6, T3 #< T5, T4 #< T5, T5 #< T6.
```

T1 = 2

T2 = 4

T3 = 3

T4 :: 1..3

T5 = 4

T6 = 5

XII.6- Les stratégies générales (méta stratégies)

a) Les méthodes et techniques qui ***<u>regardent en arrière</u>*** :

1. **Générer-Tester** (le plus **inefficace**) :

   Ne considère pas les contraintes lors d'affectation de valeurs individuelles aux variables.

   Choisit $X_n = d_n \in D_n$ tel que $\{X_1, ..., X_n\}$ satisfasse les contraintes

   ➡ le test est fait <u>trop tard</u> !

2. **Retour arrière** (BackTrack) : une sorte de ***node consistancy***

   Restreint le choix de $d_{k+1} \in D_{k+1}$ pour la variable X_{k+1} (suivant les contraintes)

   ➡ On essaie une valeur pour X_{k+1} en vérifiant les contraintes avec les valeurs (actuelles) de X_1 .. X_k.

   ⇨ Il y a quelques variantes de Retour arrière (intelligent, etc.)

   - Dans sa forme basique, BT utilise générer-tester. La différence est dans le moment des vérifications (tests).
   - Les stratégies telles que la *Programmation Dynamiques, B&B*, etc sont des formes plus évoluées de BT (avec des tests effectués à divers moments).

b) Les techniques qui *regardent en avant* :

1. **Forward Checking (BT + Arc consistancy)**

   (En plus de BT), le choix de d_{k+1} pour X_{k+1} laisse une chance à X_{k+2} .. X_n (par node consistancy)

   ➡ Dans une forme partielle, on anticipe seulement sur X_{k+2}.

   Les vérifications sont faites entre la **dernière** variable instanciée et les **restantes**.

2. **Look Ahead (BT + Path consistancy)**

   En plus de (FC), on vérifie la **satisfiabilité** d'une solution possible pour les autres variables (deux à deux).

   C-à-d :

   on vérifie qu'il y aura non seulement une chance pour chaque variable X_{k+1}... X_n sachant les valeurs de X_1..X_k,

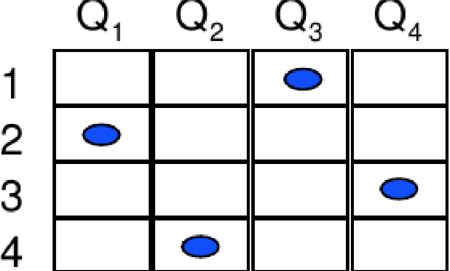
   mais aussi qu'en plus, X_{k+1}..X_n se laissent **deux à deux** une chance possible et satisfaisante.

   ➡ Arc consistance pour X_{k+1}... X_n

Remarque : la solution complète = path consistancy + BT.

XII.6.a- Illustration par les N-reines

Placer 4 reines tel qu'elles ne s'attaquent pas (sur une ligne, colonne et diagonale)



une solution à 4-rienes

Q_i = le numéro de ligne d'une reine dans la colonne i, 1 <= i <= 4

Les contraintes (*in extenso* pour la clarté) :

$Q_1, Q_2, Q_3, Q_4 \in \{1, 2, 3, 4\}$
$Q_1 \neq Q_2$, $Q_1 \neq Q_3$, $Q_1 \neq Q_4$,
$Q_2 \neq Q_3$, $Q_2 \neq Q_4$,
$Q_3 \neq Q_4$,

$Q_1 \neq Q_2-1$, $Q_1 \neq Q_2+1$, $Q_1 \neq Q_3-2$, $Q_1 \neq Q_3+2$,
$Q_1 \neq Q_4-3$, $Q_1 \neq Q_4+3$,
$Q_2 \neq Q_3-1$, $Q_2 \neq Q_3+1$, $Q_2 \neq Q_4-2$, $Q_2 \neq Q_4+2$,
$Q_3 \neq Q_4-1$, $Q_3 \neq Q_4+1$

	Q_1	Q_2	Q_3	Q_4
1				
2				
3				
4				

Méthode Générer-tester :

Au total, 256 évaluations (= 4^4):

64 échecs avec **Q1=1** (4 x 4 x 4 = 64 possibilités pour Q2, Q3 et Q4)

➥ Echecs constatés (avec Q1=1)

48 échecs avec Q1=2, 1 =< Q2 =< 3, ... (Q2=4 sera un bon placement)

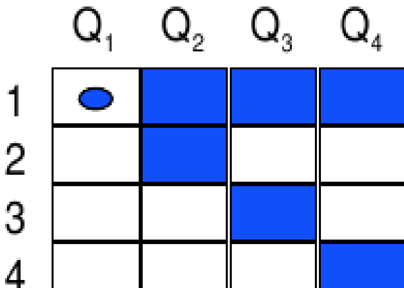
3 avec Q1=2, Q2=4, Q3=1

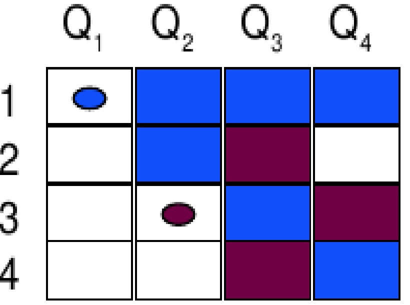
un total de **115 évaluations** pour trouver la première solution.

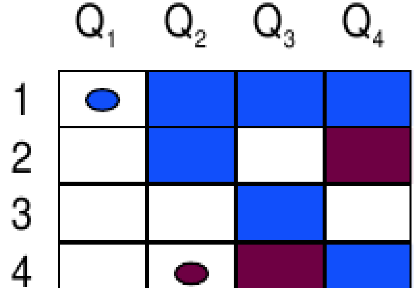
Méthode Retour arrière :

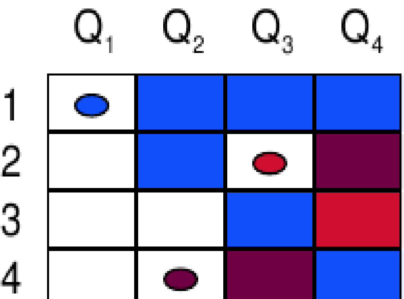
	Q_1	Q_2	Q_3	Q_4
1				
2				
3				
4				

Méthode Forward Checking (FC) : une couleur par Qi.









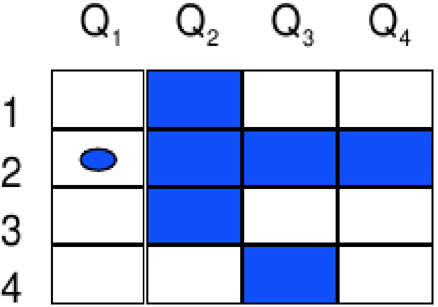
De gauche à droite et d haut vers le bas : Q1=1 permet d'éliminer les cases bleues (et laisse {3, 4} à Q2)

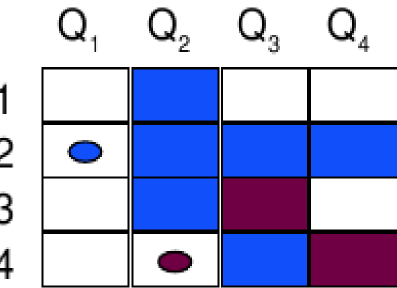
puis Q2=3 élimine les cases bordeaux (ne laisse rien à Q3); on défait Q2=3

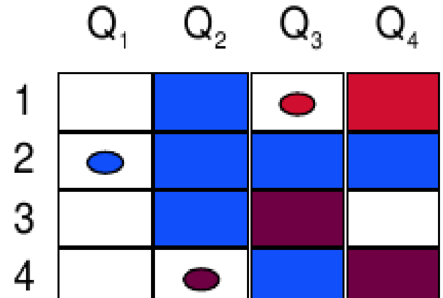
puis Q2=4 et Q3=2 ne laisse pas de chance à Q4. On défait Q1=1

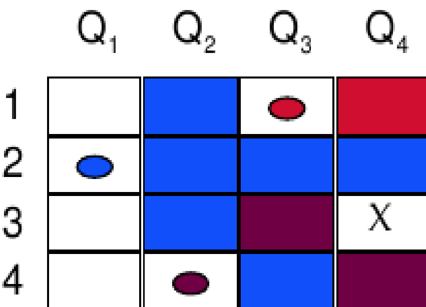
.....

Après le placement de Q2, un examen entre Q3 et Q4 (anticipation)



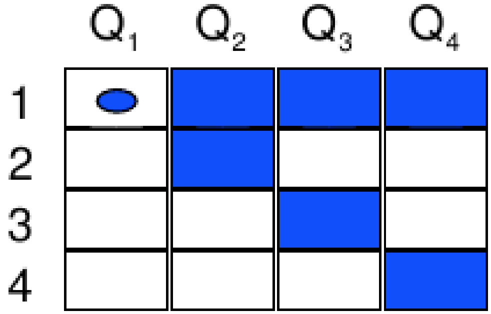


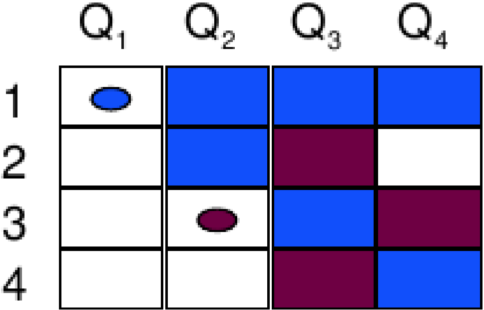


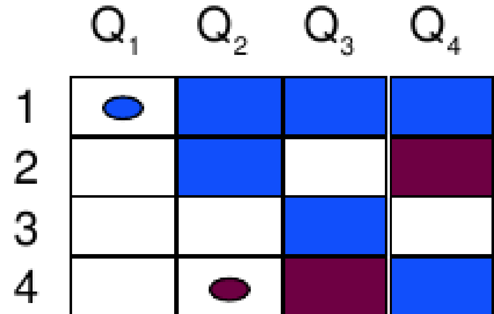


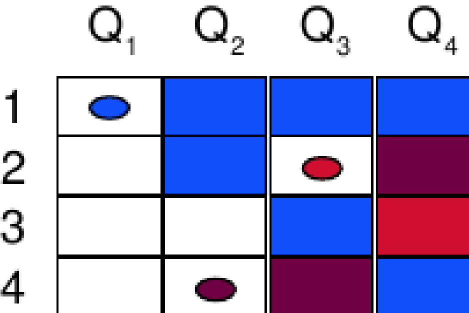
Méthode Look Ahead (LA): les couleurs impriment l'étendu des Qi.

Q1 =1 n'aboutira pas : on le sais sans être allé jusqu'au bout.



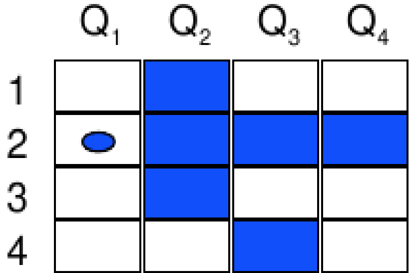


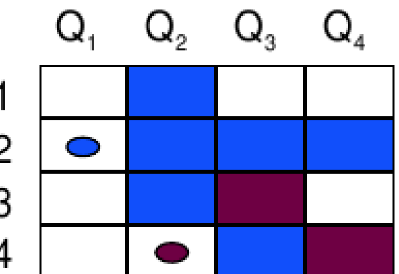


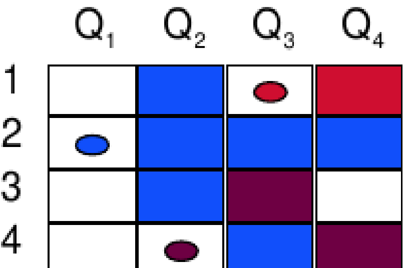


Echec dès le placement de Q2 : pas de compatibilité entre Q3 et Q4.

On passe à la 2e possibilité de Q1 :







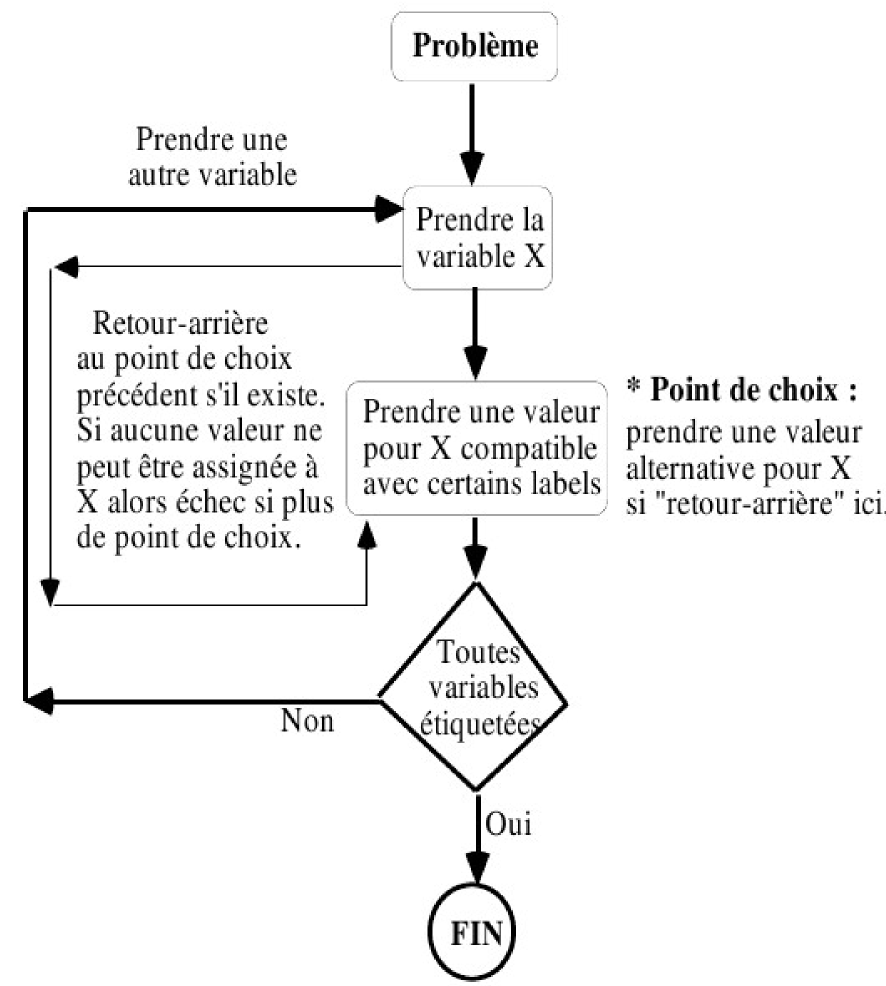
Autres stratégies appliquées en optimisation (voir plus loin) :

Programmation dynamique, Branch & Bound, A*, …

XIII- Résolution : aspects pratiques du Back-tracking

- L'opération de base dans la résolution :
  - prendre une variable à la fois
  - lui donner une valeur à la fois
  - s'assurer que le label ainsi constitué est compatible avec tous les autres labels construits jusqu'à ce stade.

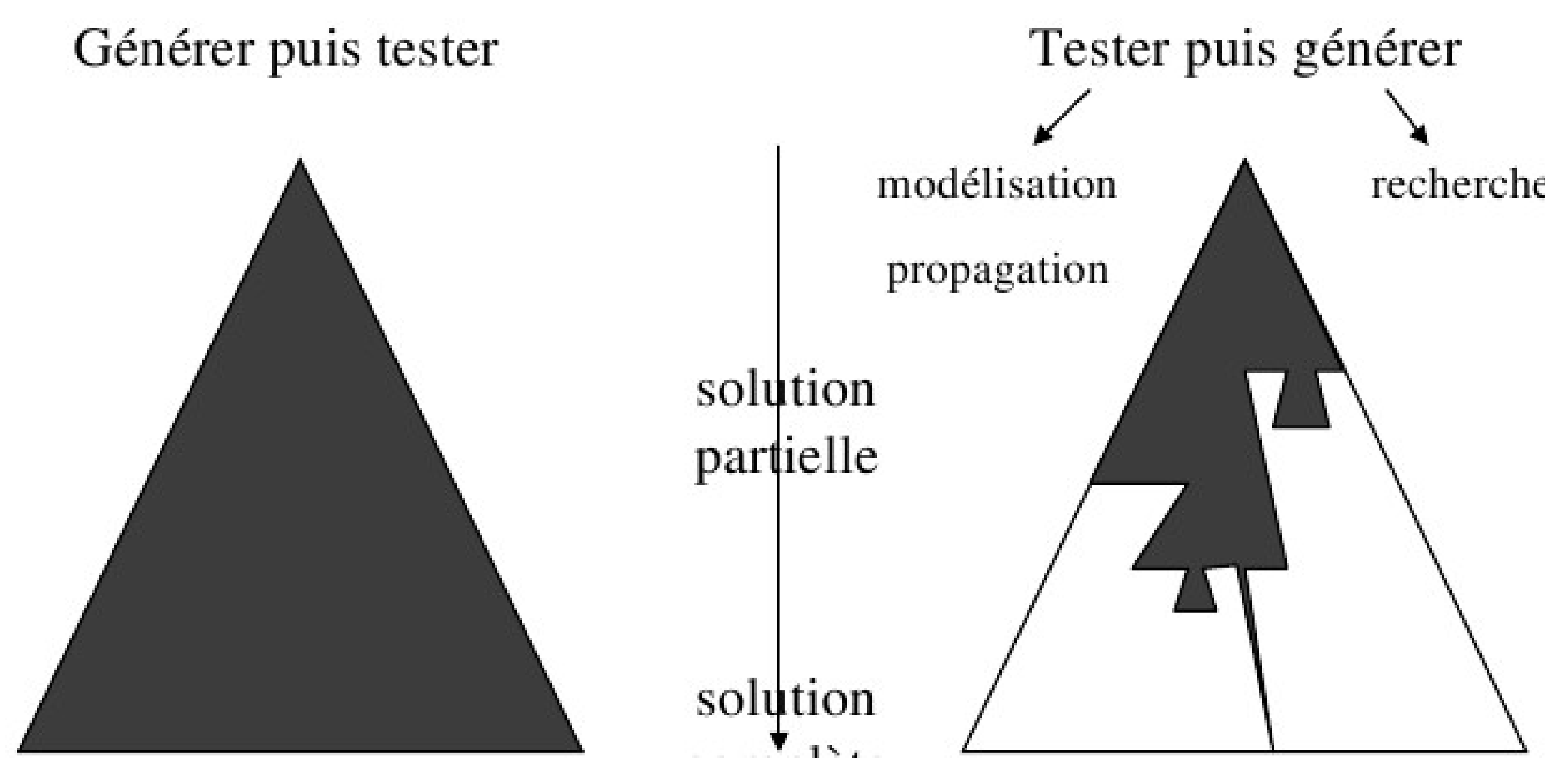
- L'algorithme "retour-arrière chronologique" (chronological back-tracking **BT**) est une stratégie générale de recherche largement utilisée dans la résolution des problèmes.



Contrôle de l'algorithme de retour-arrière chronologique

XIII.1- Retours arrières vs. Résolution des contraintes

XIII.1.a- Comparaison générale



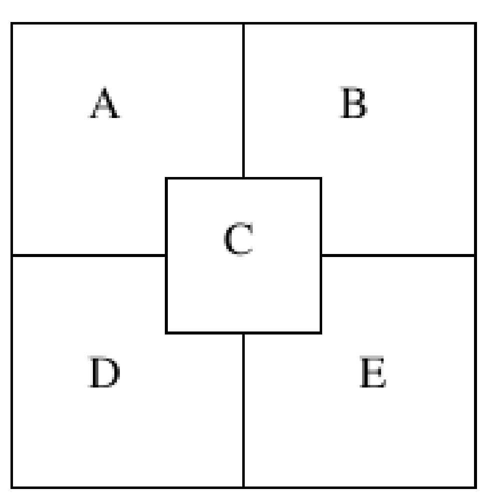
XIII.1.b- Exemple de coloration

version générer tester + BT

```
coloriage(A,B,C,D,E):-
    % on énumère
    member(A,[1,2,3]),
    member(B,[1,2,3]),
    member(C,[1,2,3]),
    member(D,[1,2,3]),
    member(E,[1,2,3]),
    % on teste
    A=\=B, A=\=C, A=\=D,
    B=\=C,B=\=E,
    C=\=D,C=\=E,
    D=\=E.
```

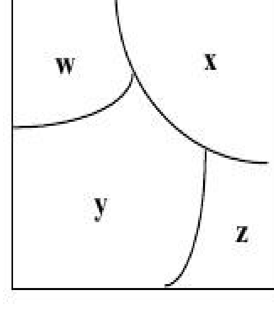
Version avec contraintes :

```
coloriageCE(A,B,C,D,E):-
    % On impose que les variables A,B,C,D,E valent 1, 2 ou 3
    fd_domain([A,B,C,D,E],1,3),
    % 2 régions voisines sont de couleurs différentes
    A#\=B, A#\=C, A#\=D,
    B#\=C,B#\=E,
    C#\=D,C#\=E,
    D#\=E,
    % énumèrer les valeurs des variables dans l'ordre A,B,C,D,E
    fd_labeling([A,B,C,D,E]).
```

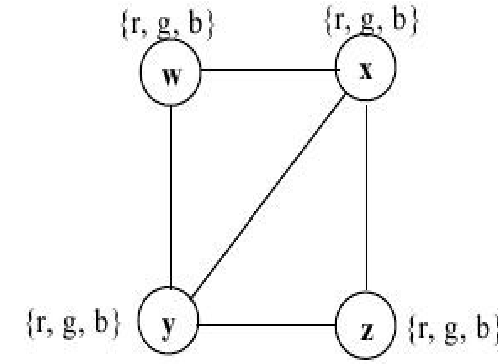


XIII.1.c- Formalisation CSP de la coloration de graphes

- Problème souvent utilisé pour illustrer diverses stratégies.
- Le problème de coloration de cartes en est une instance car les régions peuvent être considérées comme des nœuds d'un graphes et une arête relie deux régions voisines.
- Chaque arête représente une contrainte



La carte



Le graphe des contraintes

La formalisation du problème :

- Variables *{X,Y,Z,W}*
- Domaines (identiques) = *{r,v,b}*
- Les contraintes C = $\{C_{w,x}, C_{w,y}, C_{x,y}, C_{x,z}, C_{y,z}\}$.
- $C_{A,B}$: Compatible(A,B) :- $A \neq B$.
- Colorer(X,Y,Z,W) :-

  [X,Y,Z,W] :: {r,v,b},

  installer_contraintes(C),

  instancier([X,Y,Z,W]).

  .....

XIV- Domaines et exemples d'application des contraintes

Conception de matériel informatique

vérification de circuits

connexion des couches de circuits

moins efficace que le code dédiés mais plus flexible

utilisateurs : Dassault, Siemens

Placement d’objets

placement de containers

remplissage de containers

utilisateurs : Michelin

Problèmes de découpage

minimisation de pertes lors de la découpe de matériaux

(papier, verre, bois, m´taux, · · ·)

utilisateur : Dassault pour les pièces d'avion

➞ performances dépendent du contexte

papier : facile, programmation lin´aire

métaux : plus difficile, programmation par contraintes

Allocation d'espace

portes pour les avions

quais pour les trains et les bateaux

utilisateur : Aéroport CDG

Allocation de fréquences

Trouver des fréquences radio pour les :

- Téléphones portables
- communications radio
- armée

Ordonnancement de la production

- planifier les tâches sur des machines dans une usine
- plus important succès de la PPC
- meilleures performances que la RO
- plusieurs installations commerciales
- librairies dédiées à l'ordonnancement (ILOG scheduler)

Emploi du temps

- emploi du temps
- construction d'horaires de personnel :
  - santé, commerce, usine, ...
- planification des équipages sur les avions, les trains
- tournée de véhicules
- ...

Aperçu du
Langage de programmation

Prolog

NB : Le contenu de ce chapitre est traité en détails dans le polycopié Prolog disponible à l'ECL.

XIV.1- Programmation logique vs. Programmation Impérative

programmation logique	programmation impérative
règle	procédure
ensemble de règles	programme
question (but)	appel de procédure
preuve	exécution
substitution, unification	passage de paramètres

XV- Le langage Prolog

- Origines : Début des 70 , Marseille

- Permet de décrire des propriétés/relations sur les objets du problème par
  - la description des connaissances simples via des faits avérés;
  - la description des connaissances déductibles par des règles .

  On pose ensuite des questions relatives à ces descriptions et obtient des réponses calculées selon un algorithme fixe pré défini (moteur Prolog)

Domaines d'applications de Prolog :

- Intelligence Artificielle, Démonstration Automatique
- Traitement de langues naturelles, Base de données déductives, ...

Exemples :

Relation	*Interprétation possible*
pere(jean, paul).	*Jean est le père de Paul*
fils(X, Y) :- pere(Y, X).	*X est le fils de Y si Y est le père de X*
origine((X,Y)) :-	Le *couple (X,Y) est l'origine si* ***X=Y=0***
X=0 , Y =0.	
dans_cercle(X,Y, R) :-	Le *point <X,Y> est dans le cercle de rayon R si* $X^2 + Y^2 < R^2$
$X^2 + Y^2 < R^2$.	*(tout triplet <X,Y,R> tel que X2 + Y2 < R2 décrivent un cercle)*
entier(0).	*0 est un entier*
entier(succ(N)) :-	*Si N est un entier*
entier(N).	*alors son successeur l'est également*
	(l'entier N génère son entier successeur)

⇨ Ce qui rend Prolog différent des autres langages :

Il y a **deux lecture**s à tout programme Prolog : une *déclarative* et une *procédurale*.

➥ Une qui définit une relation et
l'autre qui permet de calculer des instances de cette relation.

Prolog : langage déclaratif avec une interprétation procédurale

```
derivee_symbolique(X+Y, Z, Dx + Dy) :-
    derivee_symbolique(X, Z, Dx) ,
    derivee_symbolique(Y, Z, Dy) .
```

Deux lectures possibles :

- **Lecture déclarative :** définition de la dérivée d'une somme

  La dérivée de la somme X+Y est Dx + Dy, la somme des dérivées de chacun des composants X et Y.

- **Lecture procédurale :** démarche de calcul de la dérivée d'une somme

  Pour calculer la dérivée de X+Y, calculer Dx, la dérivée de X puis calculer Dy, la dérivée de Y et construire le terme Dx+Dy représentant la somme des deux.

De même :

```
entier(succ(N)) :- entier(N).
```

- *Si N est un entier alors son successeur est un entier*
- *Pour calculer un entier, calculer l'entier N, son prédécesseur.*

  (ou si l'on a l'entier N, on calcule un autre entier -- son successeur -- par le symbole 'succ')

Le **quoi** et le **comment** : en Prolog

- l'aspect procédural est confié au dispositif de résolution de Prolog.
- le programmeur se concentre sur l'aspect déclaratif des descriptions.

  - Le quoi : *Chainage Avant* : sémantique déclarative (plus petit modèle)/ dénotationnelle (point fixe)
  - Le Comment : *Chainage Arrière* : sémantique Procédurale (SLD résolution)

```
arc(a, b).                 arc(b, c).            ...
chemin(X, Y) :- arc(X, Y).
chemin(X, Y) :- arc(X, Z) , chemin (Z, Y).
but : ?- chemin(a, c).
```

XVI- Éléments du langage Prolog par exemples

Un programme Prolog manipule des objets appelés *termes*.

Les connaissances sur le problème décrit sont exprimées sous forme de **faits** et de **règles**.

➥ On définit des **propriétés** et des **relations**

XVI.1- Faits et questions élémentaires

Un **fait** est une **assertion** (une connaissance prouvée / avérée) :

Exemples de faits:

- “jean est le père de pierre" ➟ *pere(jean, pierre)* .
- “le successeur de 0 est un entier" ➟ *entier(succ(0))*

Un **fait** est une affirmation simple et sans condition préalable. Un fait est toujours vrai.

☞ On appelle **terme fonctionnel** un terme de la forme p(x,y, ...).

Questions simples :

Une **question** simple est précédé de "?-" ou de ":-".

Par exemple, avec le programme

```
entier(0).
entier(succ(X)) :- entier(X).
```

On pose la question :

?- entier(0).

Le sens de la question : “est-ce que 0 est un entier ?

XVI.2- Exemple d'une base de faits en Prolog

Soit la base de faits :

entier(0).

pere(jean, paul).	**pere(jean, helene).**	**pere(jean, pierre).**
pere(pierre, vincent).	**pere(jacques, marie).**	**pere(olivier, mark).**

epouse(sylvie, jean).

homme(jean).

patron(emile, henri). **patron(bordet, jean).**

eleve(durand, x25). **il_pleut.**

Questions simples (question dont la réponse est <u>oui</u> ou <u>non</u>) :

?- pere(jean,pierre). ➟ succès

?- pere(jean,marie). ➟ échec

?- il_pleut(maintenant) . ➟ échec

XVI.3- Questions plus complexes

Conjonction et Disjonction de questions

- Une conjonction est vraie si tous ses composants le sont.

  ?- pere(jean,paul) , pere(jean,pierre). ➟ succès

☞ On appelle **littéral** un élément d'une conjonction.

On sépare ces littéraux avec une virgule.

- Une disjonction sera vraie si l'un de ses composants est vrai.

?- pere(jean,paul) ; homme(jean). ➟ succès

⇑ ⇑

vrai vrai

☞ On utilise un point virgule pour dire OU.

?- pere(jean, carlos) ; homme(jean). ➟ succès

?- pere(jean, carlos) ; homme(marie). ➠ échec

faux faux

XVI.4- Les variables

• Une variable est une inconnue algébrique **dont on cherche une valeur.**

• Toute variable commence par une lettre majuscule ou par '_'.

• La variable réduite à _ est une variable dite *anonyme*.

Elle occupe la place d'un argument (pour avoir le bon compte des paramètres / arguments)

• La valeur d'une variable anonyme n'est pas accessible.

Exemples de variables : **X** **Y1** **_toto** **_12056** **_**

• Une variable Prolog :

- Représente un terme.
- Si elle possède une valeur ➡ elle est dite instanciée ➡ On ne peut pas **changer** sa valeur.
- Si elle ne possède pas de valeur ➡ elle est dite **libre**.

- La valeur d'une variable **ne peut pas changer** mais elle peut être **défaite**.

➡ Un exemple : Backrack dans le problème N-reines en C/C++ et en Prolog (et en fonctionnel)

XVI.4.a- Variables dans les questions

?- pere(jean, X).

Le sens de la question : *Existe-t-il une valeur C pour X telle que pere(jean, C) soit dans la base ?*

➟ X= paul (première réponse)

➟ X= helene

➟ X= pierre

Le sens de la question : *Peut on trouver une valeur pour X telle que la formule résultante soit dans la base ?*

Le moteur Prolog essaie de trouver toutes les valeurs C pour X telles que *pere(jean, C)* soit dans la base.

C'est comme si l'on reposait la même question, tant qu'il y a un succès.

➥ Une réponse est donnée (X est instanciée), puis la valeur de X est défaite et on recommence....

Utilisation des variables anonymes

?- pere(_, vincent) ➡ succès

Les sens de la question :

- *Est-ce que vincent a un père ?*
- *Existe-t-il un individu C (dont le nom ne nous intéresse pas) tel que pere(C, vincent) soit dans la base ?*

XVI.4.b- Variables dans les faits

pere(adam, X). % Pour tout X, adam est le père de X.

patron(bordet, _).

Un fait avec variable représente d'un ensemble (**infini**) de faits.

XVI.4.c- Variables dans les conjonctions

?- pere(X, pierre) , pere(X, helene). ➟ X= jean

↑ ↑

X représente le même individu

• *Existe-t-il une valeur C pour X telle que pere(C, pierre) et pere(C, helene) soient dans la base ?*

• *Est-ce que pierre et hélène sont frère et soeur ?*

• <u>Les variables identiques dans une conjonction prennent toujours la même valeur.</u>

XVI.4.d- Variables dans les disjonctions

?- pere(X, Y) ; patron(X, _).

➟ Les deux variables X sont <u>différentes</u>

➟ On peut reformuler la même question par :

?- pere(X, Y) ; patron(Z,_).

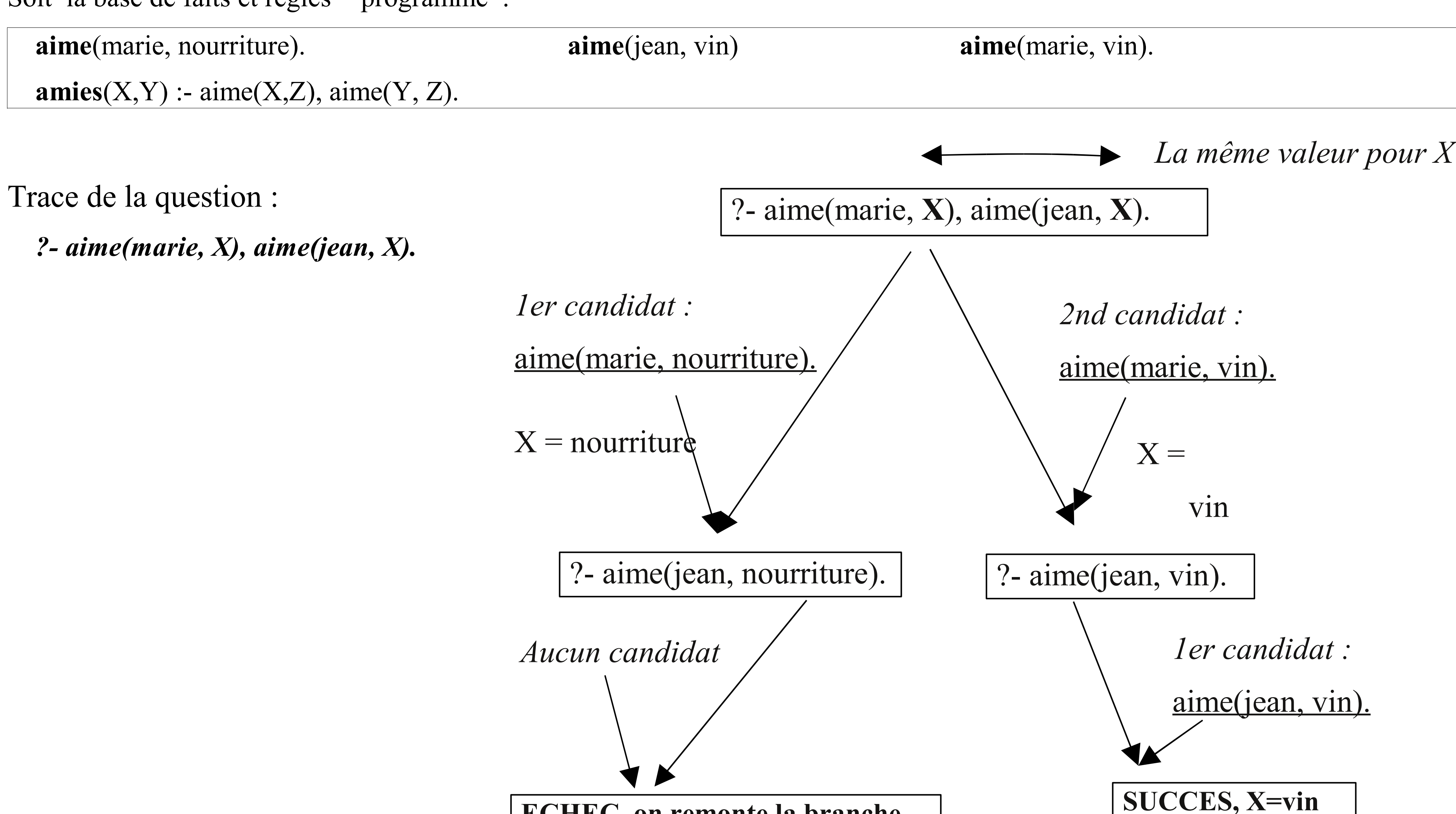
XVI.5- Démarche de résolution

Soit la base de faits et règles = programme :

aime(marie, nourriture). **aime**(jean, vin) **aime**(marie, vin).
amies(X,Y) :- aime(X,Z), aime(Y, Z).

Trace de la question :

?- aime(marie, X), aime(jean, X).



Démarche de résolution :

Une (sous) question + un candidat

Produisent des valeurs pour les variables + nouvelle question

- Si plus aucune question alors **succès.**
- Si pas/plus de candidat alors **échec**;
- En cas d'échec, remonter à l'étape précédente, défaire les valeurs des variables qui ont été instanciées et essayer d'autres candidats.
- En cas de succès, défaire les variables et essayer d'autres solutions (à reculons = backtrack).

XVI.6- Principe de le Résolution en Prolog

Soit la base de données familiale des pages précédentes.

Pour répondre à la question ?-*pere(X,Y)* :

- Recherche d'un fait *pere*(α, β) .
- X est **unifié** avec α (X est **instancié** à α); Y avec β
- Constitution de la **substitution** Θ={X/jean, Y/paul}.
- Extraction des réponses de Θ en cas de succès.

  pere(jean, paul) est une **instance** de *pere(X,Y)*.

  pere(jean, paul) = (pere(X,Y)) = (pere(X,Y)) Θ

• Si d'autres réponses demandées,

➟ Θ est remis à {} et on retourne en arrière (**Back-Track**)

On reprend à partir du fait suivant celui qui a donné un succès.

XVI.7- Unification

L'unification à Prolog est comme l'affectation dans les langages impératifs.

Unifier deux termes t1 et t2 :

trouver des valeurs pour les variables (de ces termes) qui rendent t1 et t2 **identiques**.

XVI.7.a- Principe de l'unification

(sans les termes composés qui seront traités plus loin)

- Deux termes identiques s'unifient ;
- Une variable s'unifie avec n'importe quel autre terme ;
- Deux constantes différentes ne s'unifient pas.

➡ ***Pour unifier*** *deux termes* ***t1*** *et* ***t2*** *en Prolog,*
on écrit ***t1 = t2.***

Exemples :

12=? 15 ➡ échec

X =? eleve ➡ succès, Θ={X/eleve}

X =? Y ➡ succès, Θ={X/Y}

X =? X ➡ succès, Θ={}

Ecole =? "ecole" ➡ succès, Θ={Ecole/"ecole"}

XVI.8- Retour sur les règles

- Comment exprimer que "Jean aime tous les animaux" ?

➟ Une solution : fournir une suite potentiellement infinie d'assertions (faits) dans la base de connaissances :

```
aime(jean, chien).
aime(jean, chat).
aime(jean, poisson). ....
```

➟ Une autre solution : utiliser la règle

```
aime(jean, X) :- animal(X).
```

- On utilise une règle quand on veut dire qu'un fait dépend d'autres faits.

- Les faits simples permettent de représenter les connaissances de base du monde que l'on décrit;

✔ Les **règles** Prolog permettent de représenter de nouveaux faits déductibles à partir des précédents..

- Une règle est sous la forme :

f(a1, a2, ..., an) :- g(b1, b2, ...,bm), ... , h(p1, p2, ..., pl).

↑ ↑ ↑________↑________↑

tête de la règle *Si* *corps de la règle*

La tête de la règle dépend du corps de la règle.

Lecture : **Conclusions Si Conditions**

- Lorsque le corps de la règle sera démontré vrai, alors la tête de la règle sera vraie.

- Les éléments du corps peuvent être *vrais* ou *faux*.

☞ On utilise **clause** pour désigner un fait ou une règle.

- Les règles sont aussi utilisées pour exprimer des définitions telles que :

Tout individu X est grand-père paternel de l'individu Y si :
Il existe un individu Z tel que X est père de Z et Z est pere de Y.

```
gd_pere(X, Y)      :- pere(X, Z) , pere(Z, Y).
```

Cette règle (qui utilise 3 littéraux) permet de retrouver les couples <X,Y> tels que "X est le grand père de Y" à partir de la relation "pere".

Remarques :

- Dans une règle, une variable qui apparaît plusieurs fois représente toujours le même objet.
- Quand une variable X est instanciée par un objet, tous les X sont instanciées dans la règle (= la limite de la portée de X).

Exemples :

- *X est un oiseau si : X est un animal*
  et X a des plumes.

  ➠ oiseau(X) :- animal(X) , possede(X, plumes).

- *X est la soeur de Y si : X est une femme*
  et X et Y ont les mêmes parents (père).

  ➠ soeur(X, Y) :- femme(X) , pere(Z, X) , pere(Z, Y).

- Une règle est une déclaration d'ordre général sur des objets et les relations qui les relient.

  Jean aime toute personne aimant le vin.

  ➠ *Jean aime X si X aime le vin.*

  ➠ aime(jean, X) :- aime(X, vin).

• Exemple de description des individus considérés comme des *personnes* :

```
personne(X) :- eleve(X) ; enseignant(X).
personne(Y) :- travailleur(Y).
personne(Z) :- pere(Z, _).    % un père  est une personne
personne(Z) :- pere(_, Z).    % un enfant  est une personne
```

☞ On appelle **prédicat** un paquet de règles et faits du même nom (même symbole fonctionnel à gauche des parenthèses).
Dans un programme, l'ordre entre les prédicats n'a pas d'importance.

XVI.8.a- Disjonction dans les règles

Soit la relation parent/2 :

```
(1)    parent(X, Y) :- pere(X, Y) .
(2)    parent(A, B) :- mere(A, B) .
```

On peut regrouper ces deux règles en :

```
(3)  parent(X, Y) :- pere(X, Y) ; mere(X, Y) .
```

Les règles d'instanciation des variables sont les mêmes dans les deux cas.

Dans la règle (1), X et Y sont liées; dans (2) aussi. Mais il n'y a aucun lien entre les variables de (1) et celles de (2).

Dans (3), les variables X et Y de la partie droite de la règle sont liés à celles de la partie gauche, mais elles ne sont pas liées entre elles dans la disjonction.

La disjonction facilite **seulement** l'écriture des règles. On peut toujours s'en passer !

➟ Certains Prolog interdisent l'utilisation de la disjonction.

Attention dans certains cas (e.g. 'cut', voir plus loin).

XVI.8.b- La récursivité dans les règles

Exemple-1 : définition de la fonction factorielle :

- *La factorielle de 0 est 1*
- *La factorielle de n > 0 est n fois la factorielle de n-1.*

Ce qui donne :

```
fact(0, 1).
fact(N, M) :-
    N > 0, soustraire(N, 1, N1), fact(N1, M1),     multiplier(M1, N, M).
```

N.B. : au lieu de *soustraire(N, 1, K)*, on utilise *K is N – 1.* **is** permet d'évaluer une opération (voir plus bas).
au lieu de *multiplier(L, N, M)*, on utilise *M is L * N.*

☞ Prolog n'est pas un langage fonctionnel (cf. CAML ou LISP), Il faut donc passer par *N1 is N-1, fact(N1,M1)* au lieu de *fact(N-1, M1)*, c-à-d., faire *-1* sur *N* lors du passage des paramètres.

Exemple-2 : définition des entiers (axiomes de Peano) :

On se sert de la constante *0* et de l'opérateur *succ* pour générer les termes représentant les entiers : 0, succ(0), succ(succ(0)) ...

- *0 est un entier*
- *succ(N) est un entier si N est un entier.*

Ce qui donne :

```
entier(0).
entier(succ(N)) :- entier(N).
```

Exemple-3 : la généalogie

Définition de la relation "ancêtre" à partir de la relation "parent".
(les éléments de la base sont énumérés par la relation "parent")

- *X est un ancêtre de Y si X est parent de Y (père ou mère).*
- *X est un ancêtre de Y si X est le parent d'un ancêtre de Y.*

On obtient le prédicat *ancètre/2* :

```
ancetre(X, Y) :- parent(X, Y) .
ancetre(X, Y) :- parent(X, Z) , ancetre(Z, Y).
```

NB : voir le polycopié Prolog pour plus de détails.

XVI.9- Pratique de l'unification

- Soit deux termes T1 et T2 à unifier (noté ?- T1 = T2) :
  - Si T1 et T2 s'unifient, cet algorithme produit une substitution Θ telle que Θ(T1) = Θ(T2).
  - Θ initial ={}.

- L'unification des termes "simples" est déjà étudiée
- L'unification de deux arbres (termes composés) peut réussir si

  - les racines des 2 arbres sont identiques;

  - les fils respectifs s'unifient.

Exemple : **personne(jean, Age) =? personne(Y, 25)**

➟ Θ= {Age/25, Y/jean}

Exemple : ***on veut unifiet T1 et T2***

T1 = f(a, g(X, b)) ET **T2** = f(Y, g(b, X))

étape 1- **Y =? a** (Y est-il unifiable avec a ?)

➟ Le résultat est Θ={Y/a}. On applique Θ à g(X, b) et à g(b, X) :

- Θ(g(X, b)) = g(X, b) car X n'est pas lié dans Θ
- Θ(g(b, X)) = g(b, X) idem

A ce stade, avec Θ={Y/a} , on a (égaliés des vertes) f(a, g(X, b)) $=_1$ f(a, g(b, X))
➟ On continue sur les restes

étape 2- **g(X, b) =? g(b, X)** c'est à dire :

2.1- **X =? b** ➡ Θ={Y/a, X/b}

On a : Θ(X) = b, Θ(b) = b.

A ce stade, avec Θ={Y/a, X/b} , on a (égaliés des vertes) f(a, g(X, b)) $=_{2.1}$ f(Y, g(b, X))
➟ On continue sur les restes

2.2- **b =? b** ➡ Oui

Cette étape ne produit rien de plus pour Θ;

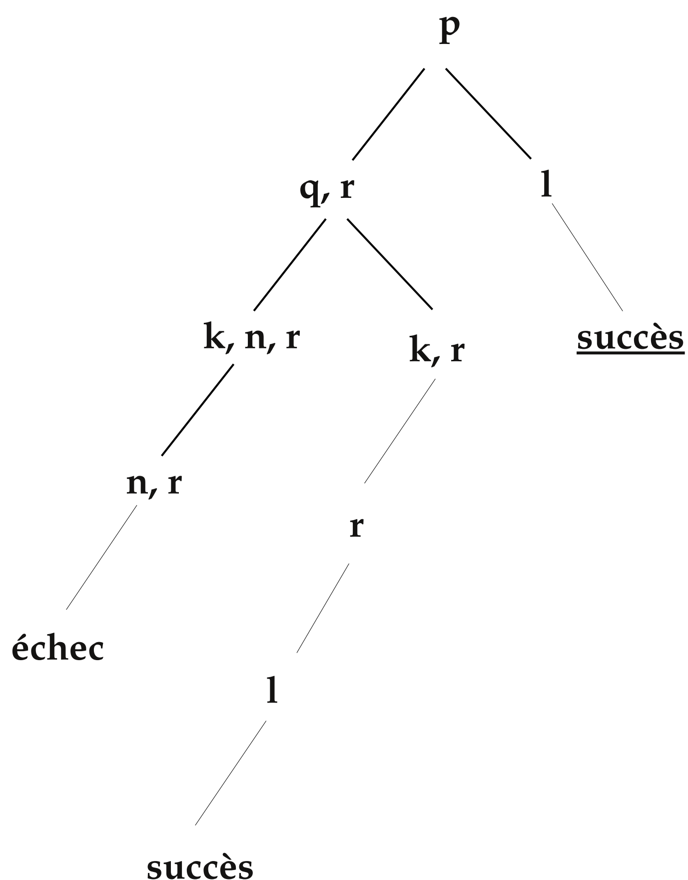
Le résultat de l'unification (ce résultat est appelé un substitution) : Θ=**{Y/a, X/b}**

XVI.10- La résolution et le retour arrière

Lors de la résolution, le retour arrière est provoqué par :

1 - un échec de démonstration

2 - un échec explicite demandé par l'utilisateur



Exemple 1 : programme sans variable

(1) p :- q , r.

(2) p :- l .

(3) q :- k , n.

(4) q :- k.

(5) k.

(6) r :- l.

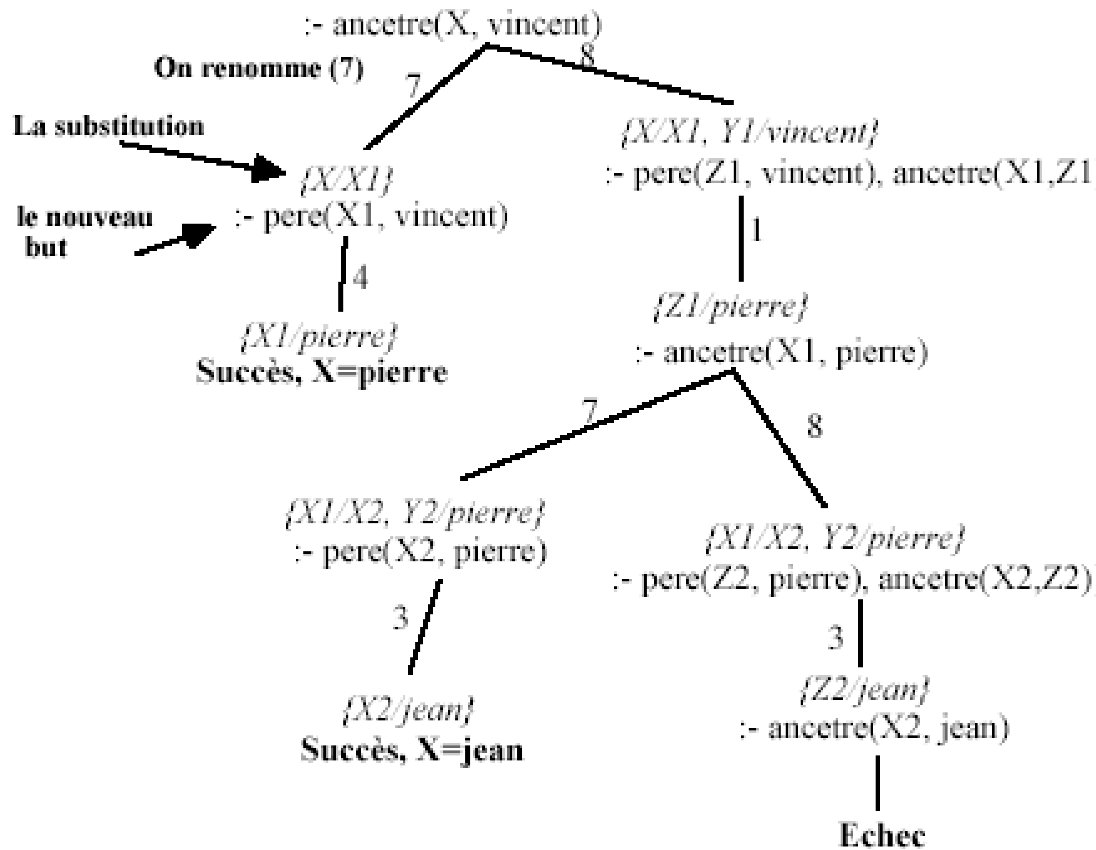
(7) l.

Et la question **p.**

Exemple 2 : programme avec variable

(1) pere(jean, jacques).
(2) pere(jean,helene).
(3) pere(jean,pierre).
(4) pere(pierre,vincent).
(5) pere(jacques,marie).
(6) pere(jacques,michel).
(7) ancetre(X,Y) :-
pere(X,Y).
(8) ancetre(X,Y) :-
pere(Z,Y) , ancetre(X,Z).

La question : **?- ancetre(X,vincent).**



XVI.11- Hypothèse du monde fermé et la négation

N'est vrai que ce que l'on peut prouver

non P est vrai si **P** est faux

non P est faux si **P** est vrai

Exemple (\+ veut dire 'not') : *femme(X)* :- \+ *homme(X)*.

Un individu I pour lequel la propriété *homme(I)* n'est pas démontrable est considéré comme femme.

Par exemple, étant donné les individus {jean, pierre, jacques} et les faits :

homme(jean).

homme(pierre).

La question **?- femme(jacques).** réussit car le fait *homme(jacques)* est absent de la base.

Un littéral négatif (*not p*) figure dans le corps d'une règle ou dans une question.

<u>Remarque sur la syntaxe</u> : On écrit *not p* par \+**p**.

XVI.12- Contrôle de la résolution

- Constat : Prolog trouve toutes les solutions.

  Parfois, une seule solution (succès ou échec) nous suffit.

  cut permet d'exprimer ce besoin.

- **cut** (noté !) a deux sens :

  - Echec explicite (*fail*)
  - Coupe-choix ou cut (*!*)

Coupe-choix(cut noté **'!')**

- Prolog donne toutes les solutions à une question (but).
- L'interprète Prolog mémorise les **points de choix**.

  a :- b , c , d , f.

- Lors Prolog rencontre un **cut** dans un but, il "oublie" tous les choix possibles précédant ce cut.

  a :- b , c , ! , d , f.

  a :- ....

  b :- ....

- **cut** réduit l'espace de recherche par l'élagage de l'arbre de résolution
- **cut** réussit toujours (est toujours effacé)
- Lorsqu'il est franchi, il:
  - Supprime les points de choix sur les prédicats figurant à sa gauche (sur “b, c" de l’exemple ci-dessus)
  - Supprime les points de choix sur la tête de la règle qui le contient (sur “a" de l’exemple ci-dessus)
- N'a pas d'effet sur sa droite (sur "d, f" de l’exemple ci-dessus) :

```
a :- b , c , ! , d , f.
a :- ....
b :- ....
```

Classification des cuts

- Cut **vert**
- Cut **rouge**

XVI.12.a- L'utilisation courante de cut

- **Cut** peut être utilisé en deux types de situations.

- Il peut être utilisé pour "dire" à Prolog :

1 ➟ "En arrivant là (sur le cut), on a utilisé la règle qu'il fallait pour satisfaire le but" (=> ne plus examiner les autres possibilités).

x_present_dans_liste(X, Liste) :-
decompose(Liste, Tete, Reste),
X=Tete, !. % Ici, on estime avoir la réponse à notre requête. On n'examine pas la clause suivante.

% On vient ici si la clause précédente n'a pas réussit.
x_present_dans_liste (X, Liste) :-
decompose(Liste, Tete, Reste), x_present_dans_liste (X, Reste).

2 ➟ "En arrivant là (sur le cut), il faut arrêter de satisfaire ce but"
(dans ce cas, on fait suivre le **cut** par **fail,** cf. *not*).

```
est_en_bonne_santé(Personne) :-
   symptôme_présent (Personne, _maladie),
   ! , fail.    % échec ➟ est malade !
est_en_bonne_santé(Personne) :-
   true.
```

D'une manière équivalente à (1) :
➟ "En arrivant là (sur le cut), on a trouvé la seule solution au problème et il est inutile de continuer à chercher d'autres solutions.

```
max(X,Y,Z) :- X > Y , ! , Z = X .
max(X,Y,Z) :- Z=Y.
```

N.B. : A propos de la mauvaise version de max !
N.B. : le 'cut' supprime souvent le caractère réversible des prédicats.

Exemple d'utilisation de cut:

```
age(vieux).
age(jeune).
taille(grand).
taille(petit).

choix1(X, Y) :- age(X) , taille(Y).     % pas de cut
choix2(X, Y) :- ! , age(X) , taille(Y).       % cut inutile (vert)
choix3(X, Y) :- age(X) , ! , taille(Y).       % cut rouge
choix4(X, Y) :- age(X) , taille(Y) , !.       % cut rouge
```

Questions :

:- choix1(X,Y) .

- ➟ X=vieux , Y = grand
- ➟ X=vieux , Y = petit
- ➟ X=jeune , Y = grand
- ➟ X=jeune , Y = petit

../..

:- choix2(X, Y) . ➟ les mêmes réponses

:- choix3(X, Y) . ➟ X=vieux , Y = grand
➟ X=vieux , Y = petit

:- choix4(X, Y) . ➟ X=vieux , Y = grand

- Le "!" dans choix2 est *Vert*.
- Le "!" dans choix3 (et dans choix4) est *Rouge*.

Un autre exemple de cut :

Expression conditionnelle : si alors sinon :

```
début
   si X mod 2 = 0 alors
      écrire (" pair ")
   sinon
      écrire (" impair ")
   finsi
fin
```

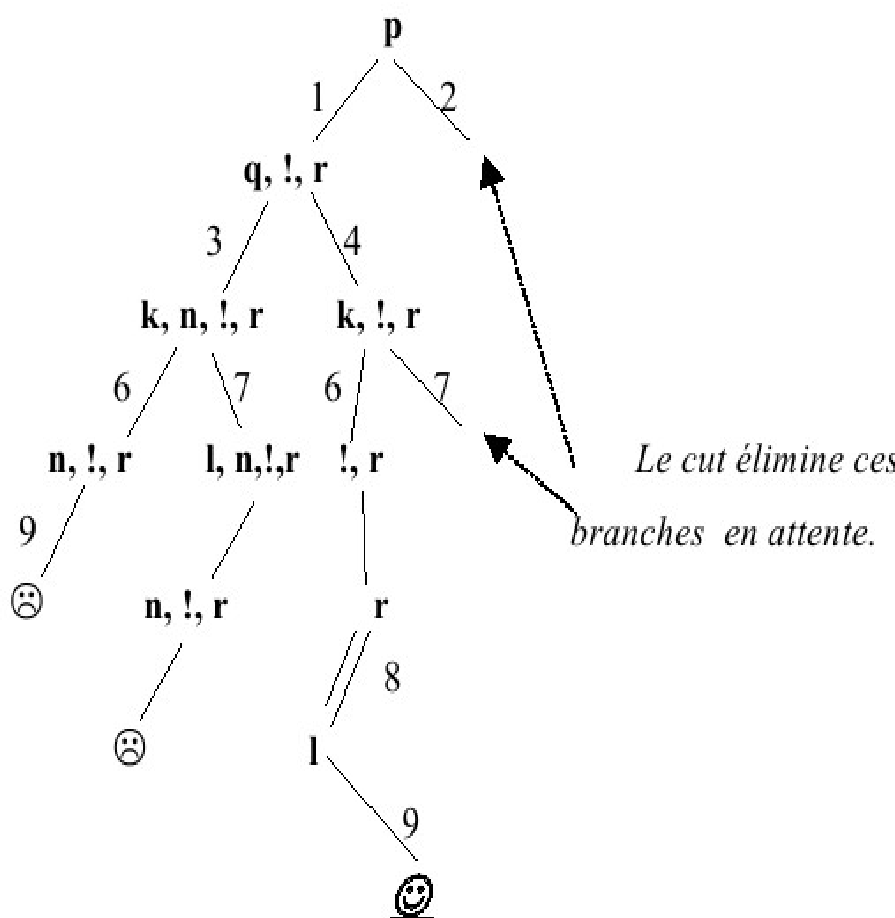
```
parite(X) :- Y is X mod 2, Y is 0, !, write('pair').
parite(X) :- write('impair').
```

Effets de cut dans l'arbre de résolution

Exemple:

(1)	**p :- q , ! , r.**	% le 1er succès de "q" suffit
(2)	**p :- l .**	% si le "!" de (1) est franchi, on
(3)	q :- k , n.	% n'essaye pas la règle (2)
(4)	q :- k.	
(5)	q.	
(6)	k.	
(7)	k :- l**.**	
(8)	r :- l.	
(9)	l.	

Et la question **:- p.**



Exemple :

trouver le maximum de 3 entiers

```
max_des_3(X, Y, Z, Max) :-
    max_des_2(X,Y, Max1),
    max_des_2(Max1, Z, Max).

max_des_2(X, Y, Max)  :- X > Y , !, Max=X.
max_des_2(X, Y, Max)  :- Max = Y.
```

Questions :

| ?- max_des_3(1,6,2,M). ==> M = 6

| ?- max_des_3(1,6,8,M). ==> M = 8

| ?- max_des_3(10,6,8,M). ==> M = 10

| ?- max_des_3(10,10,8,M). ==> M = 10

XVI.12.b- Un autre exemple de cut

<table>
<tr>
<td>forme1(X,Y) :-
member(X, [1,2]), member(Y, [1,2]), X \= Y.
forme1(0,0).</td>
<td>forme2(X,Y):-
!,
member(X, [1,2]), member(Y, [1,2]), X \= Y.
forme2(0,0).</td>
<td>forme3(X,Y) :-
member(X, [1,2]),
!,
member(Y, [1,2]), X \= Y.
forme3(0,0).</td>
</tr>
<tr>
<td>forme4(X,Y) :-
member(X, [1,2]), member(Y, [1,2]),
!,
X \= Y.
forme4(0,0).</td>
<td>forme5(X,Y) :-
member(X, [1,2]), member(Y, [1,2]),
X \= Y,
!.
forme5(0,0).</td>
<td>Questions avec cut.</td>
</tr>
</table>

<table>
<tr>
<td>| ?- forme1(X,Y).
X = 1, Y = 2
X = 2 Y = 1
X = 0 Y = 0</td>
<td>| ?- forme2(X,Y).
X = 1 Y = 2
X = 2 Y = 1</td>
<td>| ?- forme3(X,Y).
X = 1 Y = 2</td>
</tr>
<tr>
<td>| ?- forme4(X,Y).
no</td>
<td>| ?- forme5(X,Y).
X = 1 Y = 2</td>
<td>Par exemple : | ?- forme1(X,Y), !.
X = 1, Y = 2</td>
</tr>
</table>

XVI.13- Listes

- Une liste est un terme composé.
- En prolog, on place les éléments d'une listes entre [et].

  Le premier argument est appelé **tête**; le reste est appelé **queue**.

  La tête est un terme, la queue est une liste.

  On sépare les éléments de la liste par une virgule.

  Exemple: *[a, f(g), c]*

  ➟ tête = a

  ➟ queue = la liste *[f(g), c]*

- La liste vide est notée *[]*.
- On peut également séparer les termes d'une liste par '|'.

  '|' sépare la (ou les éléments de) tête de la queue.

  Le symbole '|' ne peut figurer qu'**une seule fois** dans une liste.

  Exemple :

  [a, 1, toto, f(X)] = [a | [1, toto, f(X)]] = [a, 1 | [toto, f(X)]]

Représentation canonique de [a , 124 , X]

N.B. : Tout terme peut être représenté par un arbre de la forme **terme (args)**

➟ la racine "terme" est une constante non numérique

Exemples de listes :

• [X , Y] :	tête= X ,	queue = [Y]
• [a,b,c]:	tête= a ,	queue = [b,c]
• [f(X), g([a])] :	tête= f(X),	queue = [g([a])]
• [[a,b(12)],c] :	tête = [a,b(12)] ,	queue = [c]
• [X | Y] :	tête= X ,	queue = Y
• [a , b] = [a, b | []] :	tête= a ,	queue = [b]
• [a, b, []] :	tête = a,	queue = [b, []]

Exemples d'unification

?- [a , b] = [X|Y]. ➟ X = a , Y = [b]

?- [X, a | Y] = [b, a, c, d] ➟ X = b, Y = [c, d]

Définition récursive d'une liste en Prolog :

```
liste([]).
liste([Tete | Queue]) :- liste(Queue).
```

XVI.14- Quelques prédicats de manipulation de listes

Exemple-1:

membre(Ele, Liste) vrai si Ele appartient à Liste.

```
membre(X, [X|Y]).
membre(X, [Y|Z]) :- membre(X, Z).
```

Exemples d'interrogation :

:-membre(a , [b,a,c]) ➟ succès

:-membre(a , [b,a,c,a]) ➟ 2 succès

:-membre(X , [b,a,c]) ➟ X = b ; X = a ; X= c

Exemple-2 : concaténation de deux listes

concat(L1, L2, L3) vrai si L3 est le résultat de la concaténation de L1 et de L2.

```
concat([], L, L).
concat([X|Y], L1, [X|L2]) :- concat(Y, L1, L2).
```

:- concat([a,b], [c,d], L). ➡ L= [a,b,c,d]

:- concat(X, Y, [a,b,c]). ➡ X = [] , Y = [a,b,c] ;
X = [a] , Y = [b,c] ;
X = [a,b] , Y = [c] ;
X = [a,b,c] , Y = []

XVI.15- Opérations sur les termes

Les variables sont d'abord remplacées par leur valeur éventuelle.

- **Unification :**
  - **X = Y** X et Y s'unifient et produisent une substitution .
  - **X \= Y** (ou ***not(X=Y)***)X et Y ne s'unifient pas (=NULL)

- Comparaison et Coïncidence:
  - **X == Y** X et Y littéralement identiques
  - **X \== Y** X et Y littéralement différents

• **Comparaison avec @**

Suivant l'ordre défini **croissant** sur les termes

- **X @< Y** réussit si le terme X est inférieur à Y
- **X @=< Y** réussit si X est inférieur ou égal à Y
- **X @> Y** réussit si X est supérieur à Y
- **X @>= Y** réussit si X est supérieur ou égal à Y

- **Compare(Opérateur, Terme1, Terme2)**

  Exemple :- compare(X, a, b). ➟ X= ‘<‘

Passage entre Listes et Arbres

Un_arbre **=..** Une_liste

Exemples :

f(a,b) =.. Liste. ➟ Liste = [f , a , b]

Arbre =.. [pere, jean, X] ➟ Arbre = pere(jean, X).

Remarques :

- Le terme fonctionnel (e.g. *pere(jean, X)) p*eut être appelé comme un prédicat (à l'aide de *call*) :

  Exemple : *Arbre =.. [pere, jean, X], call(Arbre).*

- Le symbole fonctionnel (*pere* ou *f* ci-dessus) ne doit être ni une variable ni un nombre au moment de l'exécution de **=..** .

- Une terme fonctionnel (de la forme p(....)) est un arbre.

XVI.16- Arithmétique

- opérateurs :

+	: addition des nombres
-	: soustraction (et le - unaire préfixé)
*	: multiplication
/	: division
mod	: reste de la division
abs	: valeur absolue (unaire)

- Evaluation puis comparaison avec les opérateurs :

<	inférieur
=<	inférieur ou égal
>	supérieur
>=	supérieur ou égal

NB : ‘=‘ et ' \=‘ ne provoquent pas d’évaluation de leur paramètre mais les deux opérateurs suivants le font :

(... suite ...)

=:=	égalité
=\=	différence

NB : ces deux dernière évaluent d'abord les deux expressions.

NB : ne pas se servir de **=:=** pour faire une affectation.

Exemple :

X=20, 3*4-2 =:= X/2 ➟ succès

is affectation / test

Dans "X is Expr", l'expression arithmétique Expr est évaluée

puis "affectée" à X (si X est variable) ou confrontée à la valeur de X.

N.B. : la partie gauche est une variable (libre ou instanciée à une valeur numérique), ou une constante.

Exemples :

X is 2 , Y is X+1. ➟ X=2, Y=3

2 is 5-3 ➟ succès

Mais (cette fois avec '=:=') :

4+3 =:= 10-3 ➟ succès

Pour toutes les formes d'expressions en Gnu Prolog, voir le polycopié Prolog.

Exemples d'utilisation :

- PGCD de deux nombres :

```
pgcd(X,X,X).
pgcd(X,Y,D) :- X < Y , Y1 is Y - X, pgcd(X,Y1,D).
pgcd(X,Y,D) :-   Y < X , pgcd(Y,X,D).
```

?-pgcd(13,78,J). ➟ J = 13.

- Version "Fast :

```
pgcd(X,X,X).
pgcd(X,Y,Z) :-
Rem is X mod Y,
 (Rem=0  -> Z = Y ; pgcd(Y,Rem,Z)).
```

- Valeur absolue :

```
vabs(X, X) :- X > 0 , !.
vabs(X, Y) :- Y is -X.
```

NB : *vabs(X,Y)* est équivalent à *Y is abs(X)*.

- minimum(X,Y, Z) : Z est min(X,Y)

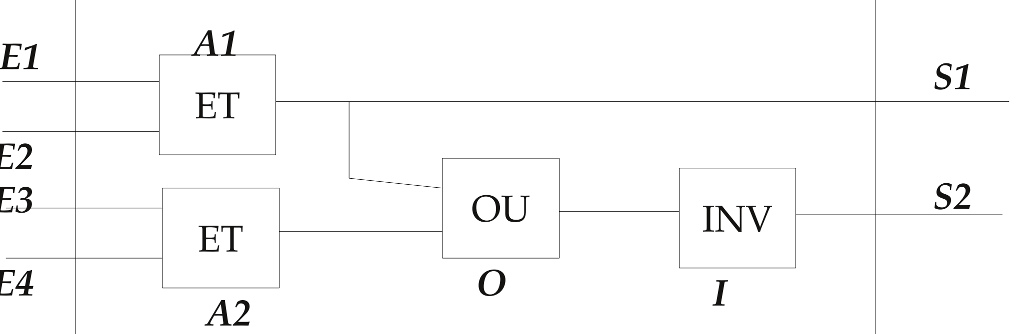
```
minimum(X, Y, Z) :- X =< Y, Z = X.
minimum(X, Y, Z) :- X > Y, Z=Y.
```

Exemple de description de circuits à l'aide de Prolog

```
circuit('C', [E1,E2,E3,E4] , [S1,S2]) :-      % 'C' est le nom du circuit
      and('A1',[E1,E2], [S1]) ,               % il est mis entre '' pour être
      and('A2',[E3,E4], [S3]) ,               % différencié d'une variable
      or('O',[S1,S3], [S4]) ,
      inv('I', [S4],[S2]).
```

Les portes élémentaires (tables de vérité de *and* (Et), *or*(Ou) et *inv* (Non)) :

and(_, [1 , 1], [1]).
and(_, [0 , 1], [0]).
and(_, [1 , 0], [0]).
and(_, [0 , 0], [0]).



or(_, [1 , 1], [1]).
or(_, [0 , 1], [1]).
or(_, [1 , 0], [1]).
or(_, [0 , 0], [0]).

inv(_, [1], [0]). inv(_, [0], [1]).

Exploitation:

:- circuit('C',[1 , 1 , 0 , 1], [0 , 1]).

➟ échec

:- circuit(X, [1 , 1 , 0 , 1], [S1, S2]).

➟ X = 'C' , S1 = 1 , S2 = 0

:- circuit(X,[1 , 1 , 0 , 1], S).

➟ X = 'C' , S = [1, 0]

:- circuit(X,[E1 , E2 , E3 , E4], [0 , 1]).

➟ 9 réponses

:- circuit('C',[E1 , E2 , E1 , E4], [0 , 1]).

➟ 5 réponses

:- circuit('C',[E1 , 0 , E1 , E4], [0 , 1]).

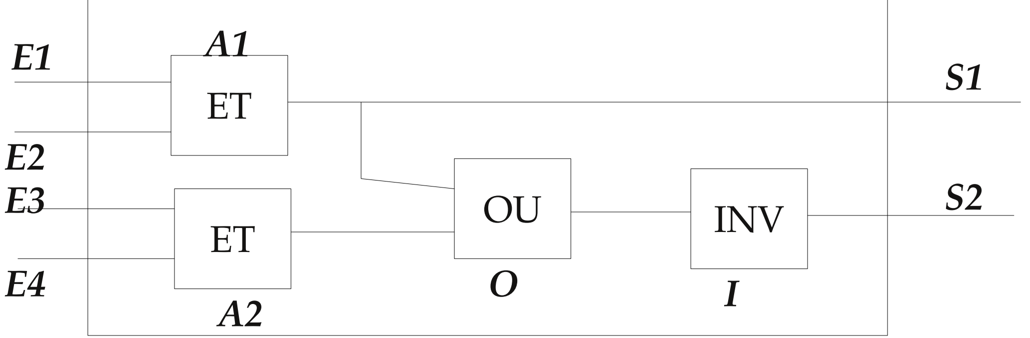
➟ 3 réponses

:- circuit('C',E, [0 , 1]).

➟ 9 réponses

:- X = [0 , 1] , circuit('C', E , X).

➟ 9 réponses



:- circuit('C', E, S). ....

XVI.17- Méta-variables et méta-prédicats

Donnée <=> Programme

Définition : un méta-prédicat est un prédicat de manipulation de prédicat.

Exemple : le schéma "For I in A .. B"

```
for(X, X, B) :- X =< B.
for(X, A,B) :- A =< B, A1 is A+1, for(X, A1, B).
```

?- for(X, 1,4), write(X).

1 X = 1

2 X = 2

3 X = 3

4 X = 4

XVI.17.a- Méta prédicats toutes-solutions

findall(Terme, But$_X$, Liste_Resultat). **%** multiple réponses (liste)

=> réponse vide possible (toujours succès)

bagof(Terme, But$_X$, Liste_Resultat). **%** multiple réponses (liste)

setof(Terme, But$_X$, Ens_Resultat). **%** multiple réponses (ensemble)

=> échec possible si liste vide.

setof = bagof ordonné ensembliste.

• Schéma générateur / testeur

pourtout(Q,P) :- \+ **(Q ,** \+ **P).**

NB : voir le polycopié pour plus de détails.

XVI.18- Manipulation d'arbres

Exemple : parcours d'arbres binaires

Recherche et insertion dans un arbre binaire ordonné horizontalement

<u>Convention:</u>

- L'arbre vide est noté []
- Il n'y a pas de doublon dans l'arbre
- La relation d'ordre sur chaque noeud (ABOH) :

  max(sag) < info(racine) < min(sad).
- La représentation de différentes configurations:

f([], []) ➟ **f** (une feuille) ➟ **f**

f(G, D) ➟

```
      f
     / \
    G   D
```

```
cherche(Ele, Ele) .
cherche(Ele, Arbre) :- Arbre =.. [Ele, _, _] .
cherche(Ele, Arbre) :-
    Arbre =.. [Racine, Gauche , _ ] ,
    plus_petit(Ele , Racine) , cherche(Ele, Gauche).
cherche(Ele, Arbre) :-
    Arbre =.. [Racine , _ ,Droite ],
    plus_petit(Racine, Ele) , cherche(Ele, Droite).
```

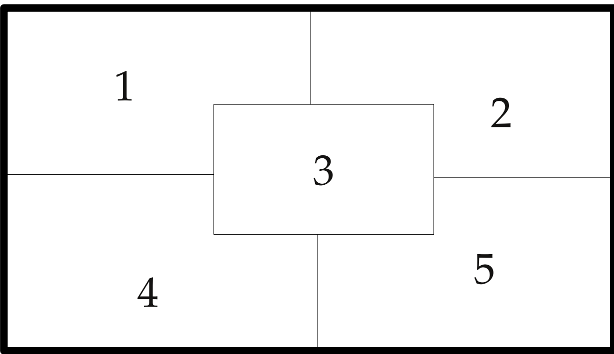
Exemples de questions:

:- N= i(b(a, f),z) , cherche(z,N) ➟ succès

:- N= i(b(a, f),z) , cherche(b,N) ➟ succès

XVII- Retour sur les méthodologies

Coloration de cartes



Enoncé:

Colorier la carte ci-dessus par les trois couleurs R, B et V en respectant la contrainte : deux régions voisines ne doivent pas avoir la même couleur.

XVII.1- Méthode constructive (Greedy, British Museum)

Stratégie générer tester :

Génération suivi de test d'une séquence <S_0, S_1, ... ,S_n> de configuration partielles.

La configuration S_{i+1} est le successeur de la configuration S_i.

Arrêt sur une configuration finale S_n . Celle-ci est déjà vérifiée.

Le schéma général de la méthode constructive :

p(...) :-
générer puis vérifier une succession de configuration partielle jusqu'à la configuration finale

Cette stratégie est caractérisée par l'absence d'emploi de retour arrière :

En Prolog, l'effet BT du moteur est annulé / contrôlé.

```
coloration(C1, C2, C3, C4, C5) :-
    voisin(C1, C2) ,  voisin(C1, C3) , voisin(C1, C4) , voisin(C2, C3) ,
    voisin(C2, C5) , voisin(C3, C4) , voisin(C3, C5) , voisin(C4, C5),
    verifier_voisinage(C1,C2, C3, C4, C5) .


couleur(r).      couleur(b).      couleur(v).
voisin(X,Y) :-  couleur(X) ,  couleur(Y) .
verifier_voisinage(C1,C2, C3, C4, C5) :-     % expression des contraintes de voisinage.
    C1 \== C2, C1 \== C3, C1 \== C4, ....
```

Questions (Gnu Prolog) : **| ?- coloration(X1,X2,X3,X4,X5).**

=> X1 = r, X2 = b, X3 = v, X4 = b, X5 = r

=> X1 = r, X2 = v, X3 = b, X4 = v, X5 = r

=> X1 = b, X2 = r, X3 = v, X4 = r, X5 = b

=> X1 = b, X2 = v, X3 = r, X4 = v, X5 = b

=> X1 = v, X2 = r, X3 = b, X4 = r, X5 = v

=> X1 = v, X2 = b, X3 = r, X4 = b, X5 = v

- Une trace sur cette exécution montre que toutes les combinaisons de couleurs sont testées.
- Cependant, **dès que** deux régions ont choisi leur couleur, le test (\==) vérifie la contrainte.

XVII.2- Méthode Retours Arrières avec génération/test

Le schéma général de cette méthode est :

p(...) :-

Génération des configurations totales

Vérification des Contraintes.

```
coloration(C1,C2,C3,C4,C5) :-
    cinq_couleurs(C1, C2, C3, C4, C5) ,      % génération
    different(C1,C2) , different(C1,C3) , different(C1,C4),
    different(C2,C3) , different(C2,C5),
    different(C3,C4) , different(C3,C5) ,
    different(C4,C5) .
couleur(r).      couleur(b).      couleur(v).    % cf. ci-dessus
different(X,Y) :- X \== Y.
cinq_couleurs(C1,C2,C3,C4,C5) :-
    couleur(C1) , couleur(C2) , couleur(C3) ,
    couleur(C4) , couleur(C5).
```

Questions (Gnu Prolog) :

| ?- coloration(X1,X2,X3,X4,X5).

=> X1 = r, X2 = b, X3 = v, X4 = b, X5 = r

=> X1 = r, X2 = v, X3 = b, X4 = v, X5 = r

=> X1 = b, X2 = r, X3 = v, X4 = r, X5 = b

=> X1 = b,X2 = v, X3 = r, X4 = v, X5 = b

=> X1 = v, X2 = r, X3 = b, X4 = r, X5 = v

=> X1 = v, X2 = b, X3 = r, X4 = b, X5 = v

XVII.3- Méthode contraindre et générer

On pose les contraintes puis on génère les combinaisons.

Les contraintes posées ont un effet global sur l'environnement.

Les combinaisons invalides seront rejetées immédiatement.

On évite de procéder à la génération dès que les contraintes sont jugées inconsistantes.

XVII.3.a- La modification de la règle de calcul de Prolog

Mécanisme de retardement (C.f. PrologII/III/IV, EcLipSe, BNR Prolog, Clp(R), Chip...).

Cette possibilité n'est pas offerte par un Prolog standard.

Le schéma général de cette méthode est

p(...) :-
Contraintes à vérifier
Génération des configurations.

Exemple en (par exemple) PrologII avec la contrainte *dif*.

```
color(C1,C2,C3,C4,C5) :-
   [C1, C2, C3, C4, C5] :: {r,v,b},
   dif(C1,C2) , dif(C1,C3) , dif(C1,C4) ,
   dif(C2,C3) , dif(C2, C5) ,
   dif(C3,C4) , dif(C3,C5) , dif(C4,C5),
   couleur(C1) , couleur(C2) ,couleur(C3) ,
   couleur(C4) , couleur(C5).
   couleur(r).      couleur(b).      couleur(v).
```

=> Mêmes réponses mais beaucoup plus rapides.

XVII.3.b- Un autre exemple (CSP)

qui montre la méthode Contraindre-Générer

- Une tâche placée à droite s'effectue après celles à sa gauche.
- Durée de chaque tâche = 1 heure
- Début de chaque tâche ∈ {1,2,3,4,5}
- Les contraintes :

```
avant(T1, T2).      % T1 < T2
avant(T1, T3).
avant(T2, T6).
avant(T3, T5).
avant(T4, T5).
avant(T5, T6).
dif(T2, T3).        % T2 et T3 disjonctives
```

Techniques de **consistance** et de **propagation**

- Avant tout calcul, les propagations successives réduisent les domaines et donnent la solution générale :

  T1 ∈ {1,2}, T2∈ {2,3,4}, T3∈ {2,3}, T4 ∈ {1,2,3},

  T5∈ {3,4}, T6∈ {4,5}

- On procède ensuite par énumération (les domaines sont finis)

Par exemple : T1 = 2 ➟ T2=4, T3=3, T4∈ {1,2,3}, T5=4, T6=5.

- Si l'on est contraint à terminer toutes les tâches en 4 heures :

➟ Max(Ti) <=4

➟ T6∈ {4,5} ➟ T6=4

➟ T5<T6, T5∈ {3,4} ➟ T5=3

➟ T4 < T5 , T4 ∈ {1,2,3}, T5=3 ➟ T4 ∈ {1,2}

➟ T3 < T5 , T3∈ {2,3} , T5=3 ➟ T3=2

➟ T1 < T3 , T1∈ {1,2} , T3=2 ➟ T1=1

➟ T2∈ {2,3,4}, T2 < T6, T6=4 ➟ T2 ∈ {2,3}

CHAPITRE 2

Systèmes d'inférence

et

Stratégies de Recherche

XVIII- Systèmes d'inférence

- Méthodes de preuve par Confirmation / Contradiction
- Exemple :

aime(jean, jean) <- aime(jean, logique). (1)

aime(jean, marie)<- aime(marie, logique). (2)

aime(jean, logique) <- aime(marie, logique). (3)

aime(marie, logique). (4)

- Pour affirmer la proposition

aime(jean, marie)

par **confirmation**, on applique la règle ***modus ponens*** :

{B, (A <- B)} |- *A.*

Cette règle déduit <u>toutes</u> les conséquences du programme ci-dessus et permet de confirmer, par la règle

$$\mathbf{En = E_{n-1}\ U\ \{les\ conséquence\ de\ E_{n-1}\}}$$

les faits suivants :

- E0={}
- E1={aime(marie, logique)}

Puis par (3) et (2) en utilisant E :

- E2={aime(marie, logique), aime(jean, logique), ime(jean, marie)}

Puis par (1)

- E3={aime(marie, logique), aime(jean, logique),
  aime(jean, marie), aime(jean, jean)}

=> E4 = E3, le processus s'arrête.

- Pour démontrer *aime(jean, marie)* par **contradiction**, on utilise la règle ***modus tolens*** :

  {~A, (A <- B)} |- *~B.*

  ~A et A s'éliminent.

  Du même programme, on obtient, par l'application de *modus tolens* (étant donné *~aime(jean, marie)*):

  { ~aime(jean, marie) &
  (aime(jean, marie) <- aime(marie, logique))}

  |- *~aime(marie, logique)*

  ~aime(jean, marie) et *aime(jean, marie)* se sont éliminés.

- Prolog utilise la réfutation : une méthode de preuve par contradiction en appliquant la règle modus tolens. *{~A, (A <- B)}* |- *~B.*

  En particulier, pour B=vrai, on a : *{~A, A}* |- *faux* (noté ◊).

  Le symbole ◊ veut dire : contradiction (faux, incohérent, inconsistant).

Remarque : en Prolog, Une question = la négation de ce que l'on veut démontrer.

En posant la question **q**, on demande à Prolog de démontrer que la négation de **q** (~q) conduit à une contradiction => donc, **q** est vrai.

N.B. : la question

?- q .

faux ← q

qui est équivalent à **~ q ∨ faux**.

Si **~q** mène à une contradiction alors **q** est vrai; sinon, c'est **faux**.

- Apporter la preuve par contradiction de la proposition A dans un programme P, c'est de prouver :

P ∪ {~A} |= ◊

NB : |= est le symbole de déduction par calcul

Donc, au lieu de dériver A de P, on peut dériver une contradiction de (P & ~A).

N.B. : P |- A = P & ~A |- ◊

car en ajoutant ~A des deux côtés, on aura :

P |- A = P & ~A |- A & ~A (◊)

- Pour la dérivation de :

{~aime(marie, logique) & aime(marie, logique)} |- ◊

=> On traite uniquement la contradiction d'une proposition particulière.

C'est souvent le cas en programmation logique où l'on est plutôt intéressé par la démonstration d'une conséquence que par celle de <u>toutes</u> les conséquences d'un programme.

- Une preuve terminée par la contradiction "◊" est appelée une **réfutation**.

➟ Au lieu d'affirmer la proposition *aime(jean, marie)* comme cela se fait avec <u>modus ponens</u> dans une preuve par confirmation; Prolog utilise le programme ci-dessus pour *réfuter* la proposition

~aime(jean, marie)

- La réfutation est le mécanisme de base de la majorité des langage de programmation logique.

Autre règle remarquable : **Abduction**

{A, (A <- B)} |- B

- Exemple1 :

"Les belges aiment les frites; Napoléon aime les frites

➟ Napoléon est belge !".

- Exemple2 :

"Il n'y a pas de fumée sans feu ; il y a fumée ➞ Il y a feu ! "

La règle d'abduction est utilisée en diagnostic (de pannes) basé sur les symptômes.

XIX- La SLD-Résolution : la résolution en Prolog

- Pour avancer dans chaque étape de la résolution et dériver la clause vide (◊), on choisit une règle, un "sous but" et on développe un **résolvant** :

A ⇨ B1, B2, .. , Bn ⇨ **C1 ,.., Cm**, B2, ... , Bn ⇨ ... ⇨ ◊

Les littéraux soulignés sont ceux choisi (pour développement)à l'aide des règles suivantes :

A ← B1, B2, ..., Bn

B1 ← C1, C2, ..., Cm

- Ils existent plusieurs façons de dériver la clause vide "◊" par la résolution (c-à-d : plusieurs manières de développer une preuve) :
  - Il y plusieurs manières de choisir une clause parent
  - Il y plusieurs manières de choisir un littéral dans le résolvant.

- Le but est de rechercher une stratégie à combiner avec la résolution pour rendre la résolution efficace sans sacrifier la complétude (c-à-d. trouver toutes les réponses).

- Une des méthodes de résolutions appliquée aux clauses *définies* est la **SLD-Résolution**.

  N.B. : une clause **définie** (definite clause) est une clause sans négation avec un seul littéral à gache.

Dans la **SLD Résolution** :

- la résolution utilise le résolvant le plus récent (le dernier) et utilise une règle (clause) du programme.
- une règle particulière appelée la **règle de calcul** choisit un littéral dans le dernier résolvant.
- En Prolog, le littéral choisi est celui le plus à gauche.

- Malgré ces choix, la SLD-Résolution donne toutes les réponses et produit toutes les dérivations, quelque soit la règle de sélection.

  Cette propriété est appelée l'**indépendance de la règle de calcul** et constitue un élément important du formalisme.

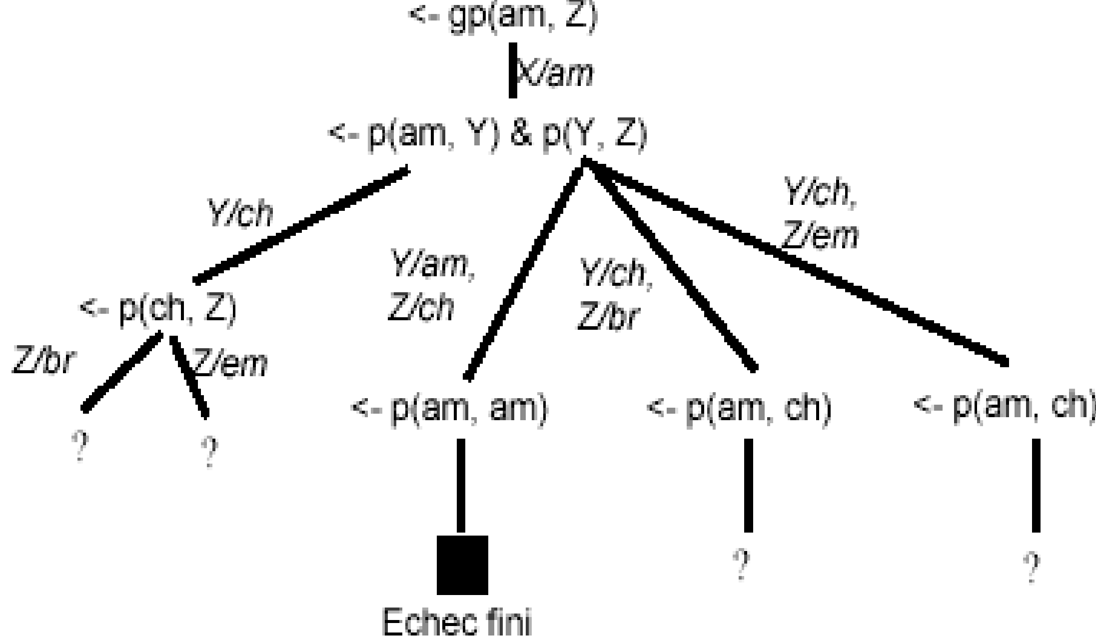
Exemple : soit le programme et la question suivante :

C1 : gp(X,Z) :- p(X, Y) & p(Y, Z).	% gp : grand parent
C2 : p(amenda, christophe).	% p : parent
C3 : p(christophe, brigitte).	
C4 : p(christophe, emile).	
Q1 : <- gp(amenda, Z).	

- Un premier arbre de recherche s'obtient si l'on choisit de résoudre tout littéral du résolvant.

  Cette règle conduit à des calculs redondants :

- Une deuxième règle de calcul peut traiter les littéraux du résolvant de gauche à droite (comme en Prolog). On obtient comme



arbre la branche la plus à gauche de l'arbre ci-dessus.

- Finalement, en traitant les littéraux du résolvant de droite à gauche, on obtient un autre arbre qui est le sous-arbre droit de l'arbre ci-dessus.

- Chacun de ces arbres est appelé un arbre SLD. Dans tous les trois cas ci-dessus et, en accord avec le principe de l'indépendance de la règle de calcul, l'ensemble de réponses obtenu est :

  {gp(amenda, brigitte), gp(amenda, emile)}

• Le choix de la règle (ou le choix de littéral dans le résolvant) peut affecter l'efficacité de la procédure de résolution.

➟ La second règle (i.e. à la Prolog) est plus efficace (dans cet exemple) que les autres.

• Connaissant la règle de calcul du système d'inférence, le programmeur peut mieux organiser (la partie droite des clauses de) son programme.

• Notons que dans les trois résolutions, les clauses C1-C4 ont été choisies dans leur ordre textuel. Ce choix est d'une importance capitale.

• Pour formaliser la résolution :

$\mathbf{Q : \leftarrow A_1 \ \& ... \& \ A_i \ \& ... \& \ A_n = \sim A_1 \mid ... \mid \sim A_n}$ le résolvant

$\mathbf{C : A \leftarrow B_1 \ \& ... \& \ B_m}$ une clause du programme

On renomme les variables de C. Si A s'unifie avec A_i et produit □, le nouveau résolvant est :

Q' : <- (A_1 &...& A_{i-1} & **B_1 &...& B_m** & A_{i+1} &...& A_n**)** □

On a :

- {Q, C} |= Q'

- Si Q' peut être résolue (réfutée) alors Q le peut.

- Si n=1 et m=0, alors Q'=◊.

- Le cœur de la procédure de résolution est la procédure d'**unification**.

XIX.1- Unification

L'unification de deux termes t1 et t2 consiste à trouver un unificateur le plus général □tel que t1□= t2□.

L'algorithme suivant (algorithme de Robinson) produit une substitution □ (≠ NULL) si l'unification réussit ou bien produit NULL dans le cas d'échec.

Fonction unifier(T1,T2 : termes; Θ : substitution) : substitution =

Cas

- ***T1 est identique à T2 :*** ***retourne(Θ)***
- ***T1 est une variable :*** ***retourne (Θ ∪ {T1/T2})***
- ***T2 est une variable :*** ***retourne (Θ ∪ {T2/T1})***
- ***T1 ou T2 est une constante :***

retourne(NULL); ***% échec***

../..

```
• T1 et T2 sont des termes composés :
  Soit : T1 = f(t1, t2,..., tn)   et    T2 = f(t'1, t'2,..., t'n).
          i<- 0;
          Répéter
                i ← i+1;
                appliquer Θ à ti et à t'i
                test <- unifier(ti, t'i, Θ)
          Jusqu'à (i>n) ou (test = NULL)
          retourne(test)
• Autre : retourne(NULL)
  Fin cas
Fin unifier;
```

XX- Stratégies de recherche et Programmation Logique

- Concerne le développement de l'arbre de recherche.

- Les interpréteurs proposent en général une règle de calcul figée d'avance.

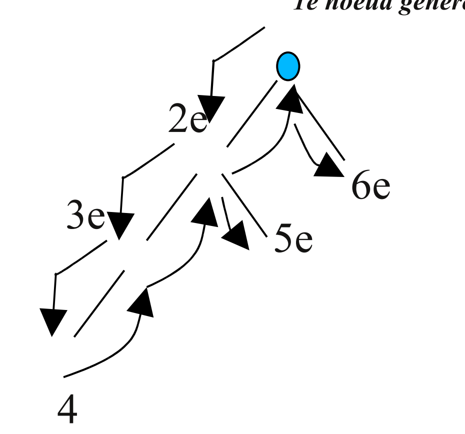
- On peut implanter notre propre règle en écrivant un *méta-programme* s'interposant entre l'interpréteur et notre programme.

- La recherche peut être déterministe (avec une seule réponse = fonctionnel) ou non-déterministe (avec plusieurs réponses = relationnel).

- Habituellement, les programmes logiques nécessitent une recherche non-déterministe; ce qui implique l'emploi d'une stratégie de recherche.

- Il existe un certain nombre de stratégies de recherche. La stratégie classique des interpréteurs logique tournant sur un seul processeur est :

    - Séquentielle
    - Descendante
        - En profondeur d'abord
        - Avec retour arrière



- La stratégie standard développe un arbre de recherche SLD selon le schéma ci-dessus. Cette stratégie est exhaustive et excessive; elle ne garanti pas la terminaison. Elle termine si l'arbre est de profondeur finie.

- Lorsque le nombre de clauses du programme est fini, on peut envisager un développement "en largeur" permettant de trouver les réponses; ce qui n'empêche pas un calcul infini (si l'arbre contient des branches infinie, le calcul "boucle" après avoir donné toutes les réponses).

- La stratégie "en profondeur d'abord" est plus optimale et utilise le minimum de mémoire toute en <u>préservant la complétude</u> de la SLD-résolution.

XX.1- Règles de choix

- Il y a deux choix possibles dans la SLD-résolution : le choix d'un littéral dans le résolvant (appelé *la règle de calcul*) et le choix d'une clause du programme (appelé *la règle de recherche*)

- Pour un programme et une question, le choix de la règle de calcul détermine l'arbre SLD correspondant.

- La règle de recherche détermine l'ordre dans lequel les noeuds (les calculs) sont générés dans l'arbre SLD. Elle concerne le choix des clauses dans le programme.

- Les exemples suivants (problème classique de la recherche d'un chemin dans un graphe) montrent les différentes réponses que l'on peut obtenir d'un même programmes en variant les deux règles ci-dessus.

1 - Règle de calcul : choix du littéral le plus à gauche

Règle de recherche : choix des clauses dans l'ordre d'entrée

Programme :

```
p(X,Z) <- a(X,Z)
p(X,Z) <- a(X,Y) & p(Y,Z)
a(a,b)
a(b,c)
<- p(a,Z) .
```

Réponses : on obtient un calcul fini avec Z=b, Z=c puis deux échecs.

2 - Règle de calcul : choix du littéral le plus à droite

Règle de recherche : choix des clauses dans l'ordre d'entrée

Programme :

```
p(X,Z) <- a(X,Z)
p(X,Z) <- a(X,Y) & p(Y,Z)
a(a,b)
a(b,c)
<- p(a,Z)
```

Réponses : on obtient un calcul infini avec Z=b, échec, Z=c puis une branche infinie.

3 - Règle de calcul : choix du littéral le plus à gauche

Règle de recherche : choix des clauses dans l'ordre d'entrée

Programme (ordre des clauses modifié) :

p(X,Z) <- a(X,Y) & p(Y,Z)

p(X,Z) <- a(X,Z)

a(a,b)

a(b,c)

<- p(a,Z)

Réponses : on obtient un calcul fini avec deux échecs, puis Z=c, Z=b (réponses inversées).

4 - Règle de calcul : choix du littéral le plus à droite

Règle de recherche : choix des clauses dans l'ordre d'entrée

Programme : celui du cas 3

Réponses : boucle immédiate et infinie sans aucune réponse.

XX.2- Algorithme de résolution de Prolog

- Soit le but **B= b1, b2 , ... , bn** et un programme **P**
- Appel : Θ <- resoudre (B , {});
- Retour : Si Θ = NULL alors **échec**

Sinon Θ contient les valeurs des variables de la question B.

Fonction resoudre (B : but ; Θ : substitution) : substitution =

Si B = {} alors retourne (Θ); Fin si;

(1) Considérer b1 dans B; b1 de la forme **f(a1,..., am)**

LC = l'ensemble des règles du programme P de la forme

f(a'1,..., a'm) :- A1, A2, ..., Ak.

Tant que LC ≠ {} faire

(2) Choisir C le premier élément de LC; LC = LC - {C};

Θ <- **unifier(**Θ(ai), Θ(a'i), Θ**)** pour i=1..m

Si Θ ≠ NULL Alors

B' = **A1, A2, ..., Ak** , b2 , ... , bn

Θ <- resoudre(Θ(B'), Θ)

```
(3)                       Si Θ ≠ NULL Alors  retourne (Θ) ; Fin si;
        Fin si;
  Fin Tant que;
  Retourne (NULL);
Fin resoudre;
```

Remarques : règle de calcul en (1); règle de recherche en (2).

Cette version est déterministe en (3).

Exemple : reprendre l'exemple *grand_pere/2* ci-dessus et appliquer l'algorithme ci-dessus pour la question

?- gp(amenda, Z).

XX.3- Principe d'un méta-interpréteur de Prolog

Remarque :

Le prédicat prédéfini **Clause(Tête, Corps)** permet d'extraire (une par une avec retours arrières) les clauses du programme dont la tête s'unifie avec *Tête*.

Le principe d'une solution :

```
prouver((B ,BS)) :-           % Le but est composé
    prouver_un_but(B) ,
    prouver(BS).
prouver(B) :-
    B \= (_, _) ,             % Le but est simple
    prouver_un_but(B).


prouver_un_but(true).         % ‘true’ est vrai !
prouver_un_but(B) :-          % pour tout autre (que ‘true’)
    clause(B, Corps),         % trouver un candidat
    prouver(Corps).
```

Question :

En Prolog : **?- la_question.**

Ici : **?- prouver(la_question).**

Remarques :

- On peut ajouter le cas de disjonction.
- On ne tient pas compte de **cut** ici.
- Les boucles éventuelles du programmes ne sont pas détectées.
    - Une version opérationnelle en Gnu Prolog est ci-dessous.

XX.3.a- La version Gnu Prolog avec exemple

```
:-public(test/1).      % pour l'accès par 'clause'


prouver((B ,BS)) :-        prouver_un_but(B) , prouver(BS).

prouver(B) :-          B \= (_, _), prouver_un_but(B).


prouver_un_but(true).

prouver_un_but(B) :-

   extraire_partie_dte(B,Corps),             prouver(Corps).


% vérifier s'il s'agit d'un prédéfini ?

extraire_partie_dte(T, true) :-

   T =.. [H|R],   length(R,N),

   predicate_property((H)/N, built_in),        % un prédéfini ?

   !,   call(T).

../..
```

```
% autre qu'un prédéfini


extraire_partie_dte(T, Q) :-
        clause(T, Q).     % extraction de la partie droite


% Exemple d'un programme simple à tester :


test(0).
test(X) :- nonvar(X), X>0, X1 is X-1, test(X1).
```

Questions :

| ?- prouver(test(5)). => true

| ?- prouver(test(A)). => A = 0 (pas d'autre réponse)