Introduction à

l'Intelligence Artificielle

Algorithmes

&

Raisonnement

Représentation de connaissances

et

Techniques de Recherche

2ème Année ECL-MI

1998-2003

A. S. SAIDI

Table des matières

Remarque :

Un polycopié Prolog est séparément disponible et complète ce document.

Objet du cours

- Les outils de représentation de connaissances
- Les algorithmes de recherche

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.

Introduction

Origine : Démonstration Automatique de Théorèmes (ATP)

- ➘ 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.

Un problème est NP-Complet : un algorithme de complexité polynomial n'est pas connu pour résoudre ce problème (soit aucun algorithme n'est connu, soit il en existe de complexité grande, par exemple exponentielle).

L'ATP est inefficace pour les problèmes de taille importante.

➘ Systèmes Experts :

Règles adéquates pour diviser les problèmes complexes en sous problèmes plus simples.

• L'expérience de MYCIN :

Importance de la bonne connaissance par rapport aux mécanismes d'inférence.

Peu importe le mécanisme d'inférence; l'essentiel est de savoir représenter les informations :

"*pseudomonas est de souche gram-négative et il peut infecter un patient dont le système immunitaire est déficient !*"

• Les SE ont eu quelques succès.

• Tendance actuelle de la recherche :

L'accent sur la représentation connaissances.

But :

• Présenter une sémantique claire des données ;

• Algorithmes de manipulation de ces connaissances;

• 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 peut pas résoudre les cas limites "incalculables" (NP-hard, i.e. NP-complet) mais ces cas là se présentent rarement.

La représentation des connaissances et raisonnement

Les langages de représentation de connaissances

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

• Logique :

A base de relations : *Rel(A,B), ...*

• Exemple : *Hélène donne un livre à jean*

$\exists$p,q,r : personne(p) $\wedge$ nom(p, Jean) $\wedge$ personne(q)

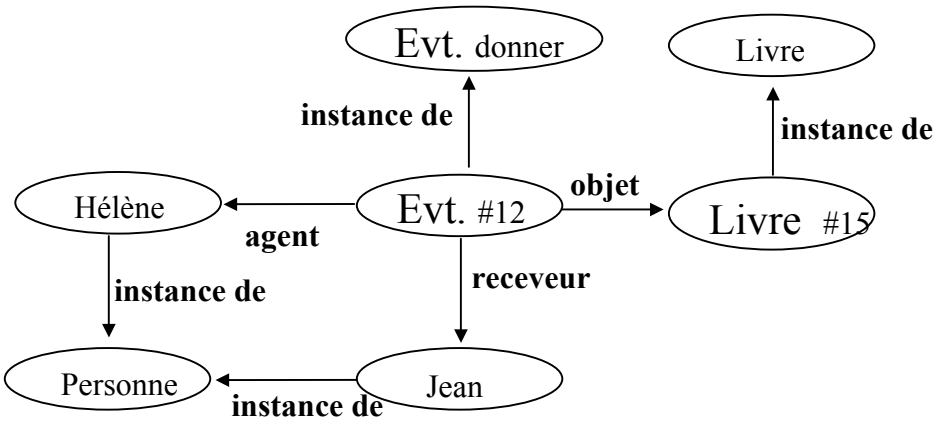
$\wedge$ nom(q, Hélène) $\wedge$ livre(r) $\wedge$ donne(q,r,p).

• Prolog

• Langages à base de réseaux :

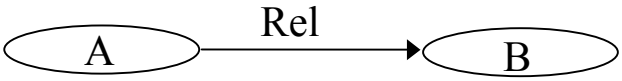
Un exemple de réseaux sémantiques :

Hélène donne un livre à jean



• 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 vrai et en plus il expliquera comment la base de connaissance doit être recherchée (attachements).

• Objets :

Une variante des calculs des prédicats.

Un exemple typique dans les langages "slot-filtre-frame" :

(une personne

(nom=jean)

(age = 25))

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

• Un réseau sémantique = une collection de frames

• Les frames = les réseaux sémantiques généralisés avec la notion de classe et d'instance est plus forte.

• Les frames contiennent des slots et des valeurs de slots;

• Peuvent décrire des instances et des classes (et autres relations);

../..

• Peuvent avoir des procédures d'accès, démons , réflexes , ...

➘ Démons : maintien des contraintes (e.g. when-constructed)

• Sont plus simples à comprendre mais sont moins expressifs.

➘ on ne peut pas dire que "le nom de la personne est Jean ou Jacques" ou "l'age n'est pas 30".

• **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, calculer des expressions non exprimées par les frames.

Caractéristiques d'une "bonne" représentation de connaissances :

- Objets et relations explicites ;
- Exprime les contraintes naturelles ;
- Supprime les détails inutiles ;
- La représentation est transparente ;
- Elle est complète et concise ;
- La manipulation est rapide;
- La représentation est calculable ;

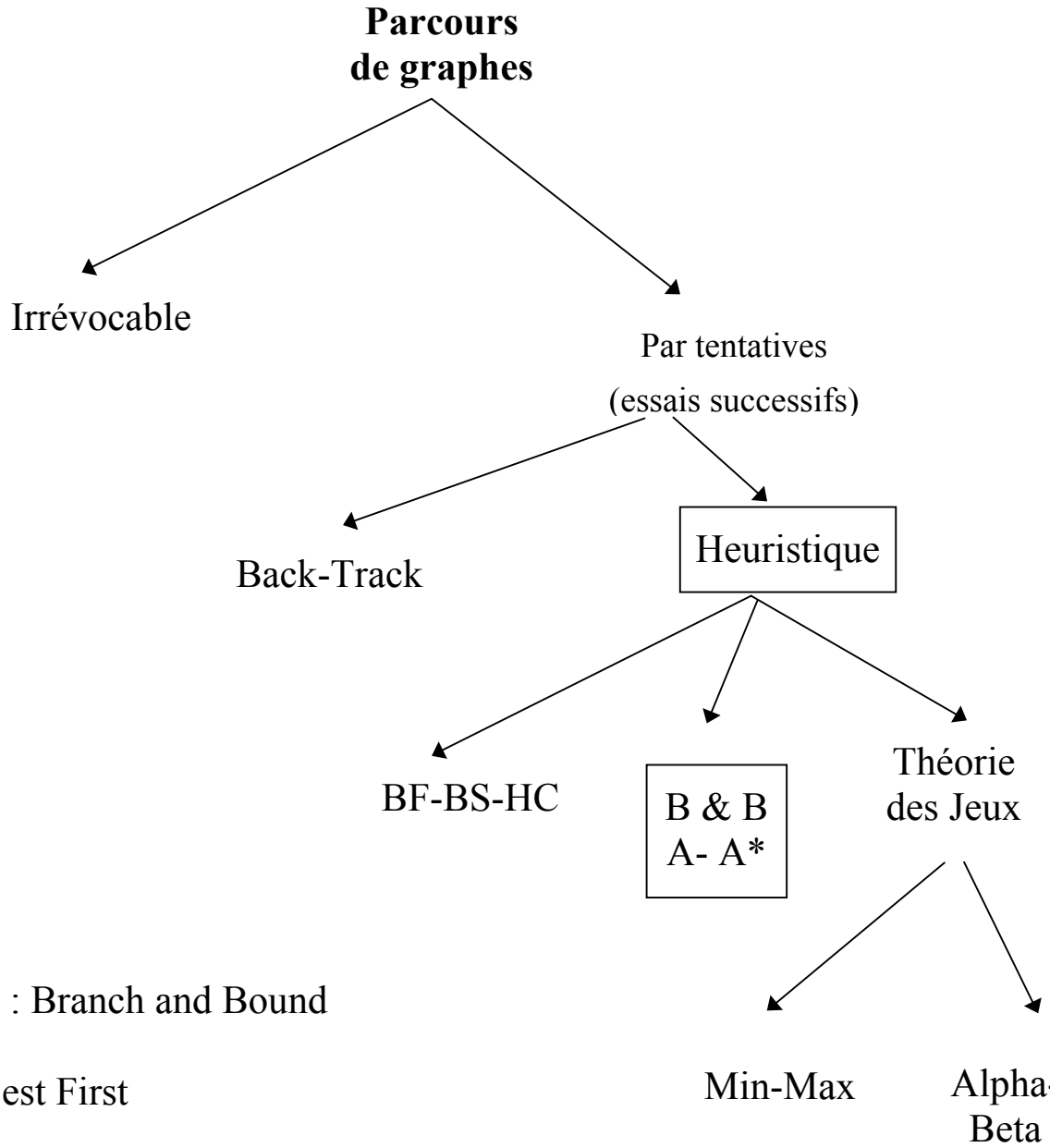
Eléments fondamentaux d'une représentation

- **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 procédurale.
- **sémantique** : signification.

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

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

Classement de quelques stratégies de recherche



B & B : Branch and Bound

BF : Best First

BS : Beam Search

HC : Hill Climbing

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

CHAPITRE 1 : Aperçu du langage Prolog

CHAPITRE 1

Aperçu du

langage de programmation

Prolog

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

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,
- Prototypage systèmes Experts
- ...

Exemples de relations

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)) :- X=0 , Y =0.	*le couple (X,Y) est l'origine si X=Y=0*
dans_cercle((X,Y), R) :- $X^2 + Y^2 < R^2$.	*le point (X,Y) est dans le cercle de rayon R si $X^2 + Y^2 < R^2$*
entier(0).	*0 est un entier*
entier(succ(N)) :- entier(N).	*si N est un entier alors son successeur l'est également*

Prolog : langage déclaratif avec une interprétation procédurale

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

Deux lectures possibles :

• **Lecture déclarative :**

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

• **Lecture procédurale :**

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 N, son prédécesseur.*

En Prolog :

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

Eléments 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.

Faits et questions élémentaires

Un fait est une assertion : une connaissance prouvée :

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,

?-entier(0)

pose la question : "est-ce que 0 est un entier ?".

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).
il_pleut.
homme(jean).
patron(emile, henri).
patron(bordet, jean).
eleve(durand, x25).

Questions simples

(question dont la réponse est oui ou non) :

?- pere(jean,pierre). **=> succès**
?- pere(jean,marie). **=> échec**
?- il_pleut(maintenant) . **=> échec**

Questions 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.

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

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

⇑ ⇑

vrai vrai

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

⇑ ⇑

faux vrai

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

⇑ ⇑

faux faux

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.

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

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

• Une variable Prolog :

- possède une valeur = représente un terme.

➘ Elle est dite instanciée. On ne peut pas changer sa valeur.

- ne possède pas de valeur => elle est dite libre.

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

Variables dans les questions

?- pere(jean, X).

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

Autre lecture :

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.

Utilisation des variables anonymes

?- pere(_, vincent) => succès

- *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 ?*

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.

Variables dans les conjonctions

?- pere(X, pierre) ,pere(X, helene).

↑ ↑

les X représentent le même individu

=> X= jean

- *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 helene sont frère et soeur ?*

• Les variables identiques dans une conjonction prennent la même valeur.

Variables dans les disjonctions

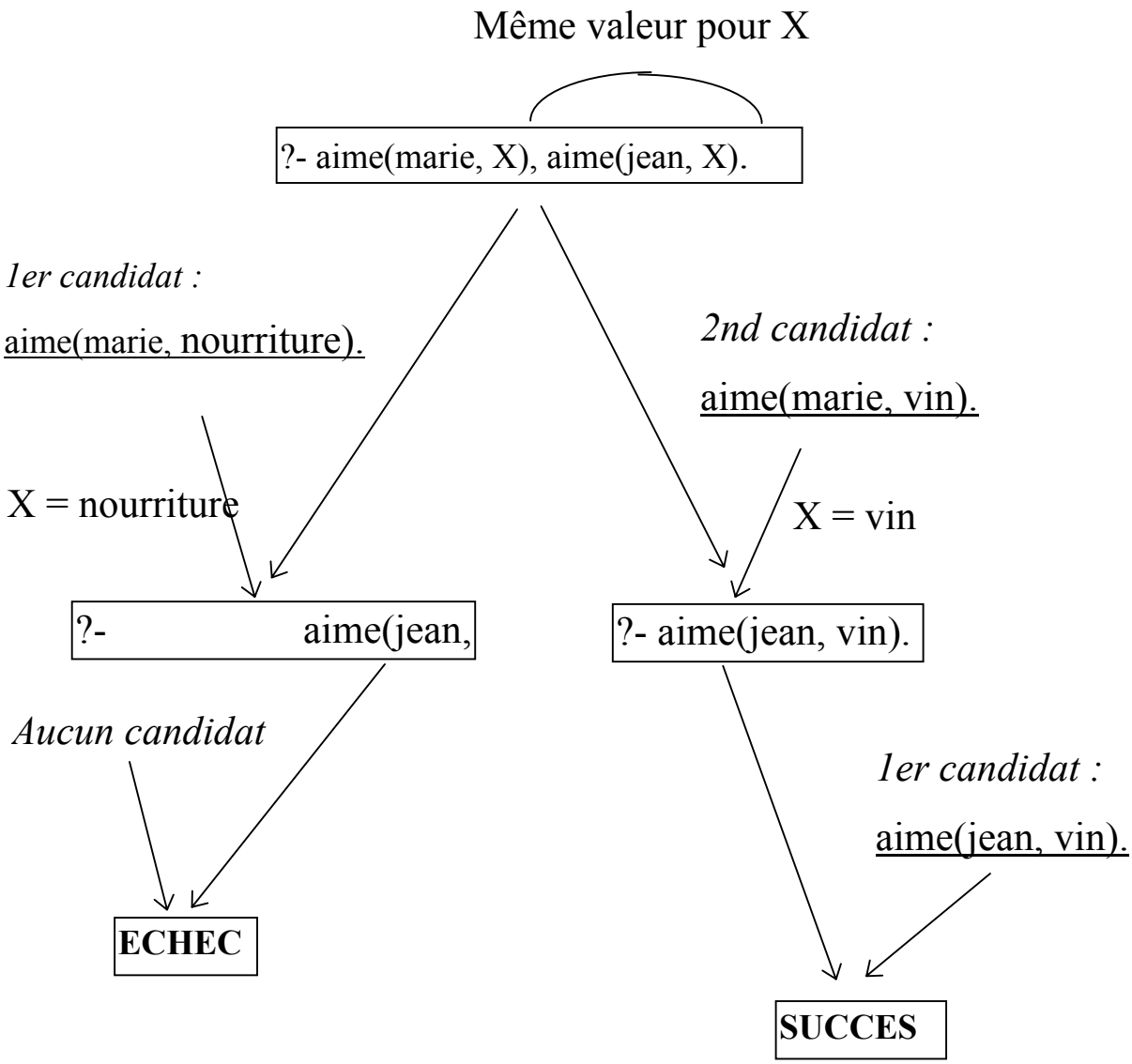
?- pere(X, Y) ; patron(X, _).

=> Les deux variables X sont différentes

=> On peut re formuler la même question par :

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

Trace de la question "?- aime(marie, X), aime(jean, X)."



Démarche de résolution :

Une (sous) question + un candidat	=>	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 modifiées et essayer d'autres candidats.

Principe de le Résolution en Prolog

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 b
- 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).

  $$pere(jean, paul) = \theta\,(pere(X,Y)) = (pere(X,Y))\,\theta$$
- Si d'autres réponses demandées,

  => θ est remis à {} et on retourne en arrière (**retour-arrière**)

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

Principe de l'Unification

Unifier deux termes t1 et t2 : trouver des valeurs pour les variables de ces termes qui rendent t1 et t2 identiques.

Principe de l'unification (sans les termes composés) :

- 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"}

Règles

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

  => Une solution : fournir une suite 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 de ces faits.

- 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 lecture procédurale d'une règle est : **conclusion SI conditions.**

- Lorsque le corps de la règle est vrai alors, la tête de la règle est 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 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 (sans disjonction), 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).

• *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.

ou *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 de définition des prédicats n'a pas d'importance.

Disjonction dans les règles

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

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 : X dans pere(X,Y) est différente du X dans mere(X,Y).***

La disjonction facilite seulement l'écriture des règles.

Par exemple, on peut réécrire :

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

en

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

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, soustr(N, 1, K), fact(K, L), mult(L, N, M).
```

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 pour plus de détails.

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

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

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

21- **X =? b** => θ={Y/a, X/b}

On a : θ(X) = b, θ(b) = b.

22- **b =? b**

Cette unification ne produit pas de lien;

Le résultat de l'unification (ce résultat est appelé une substitution):

θ={Y/a, X/b}

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.

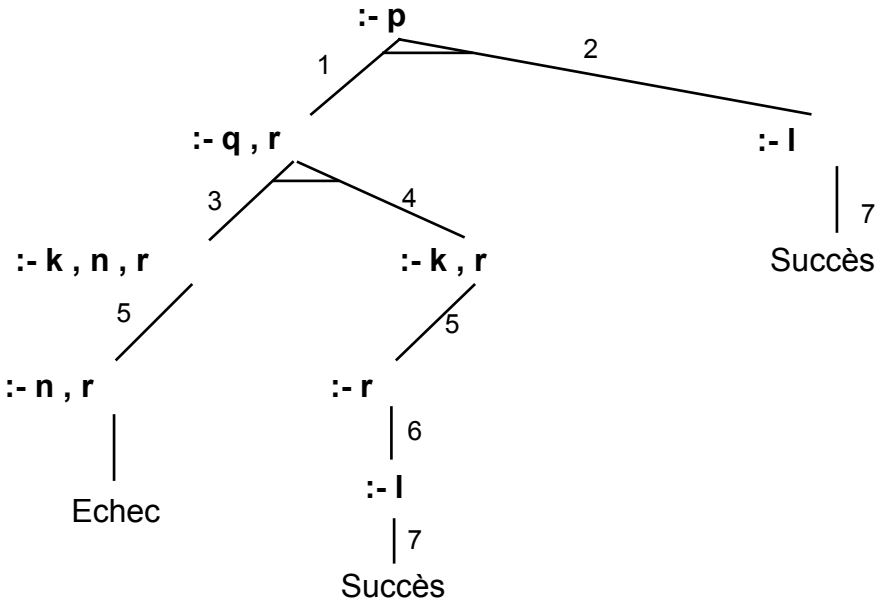
(4) q :- k.

(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).

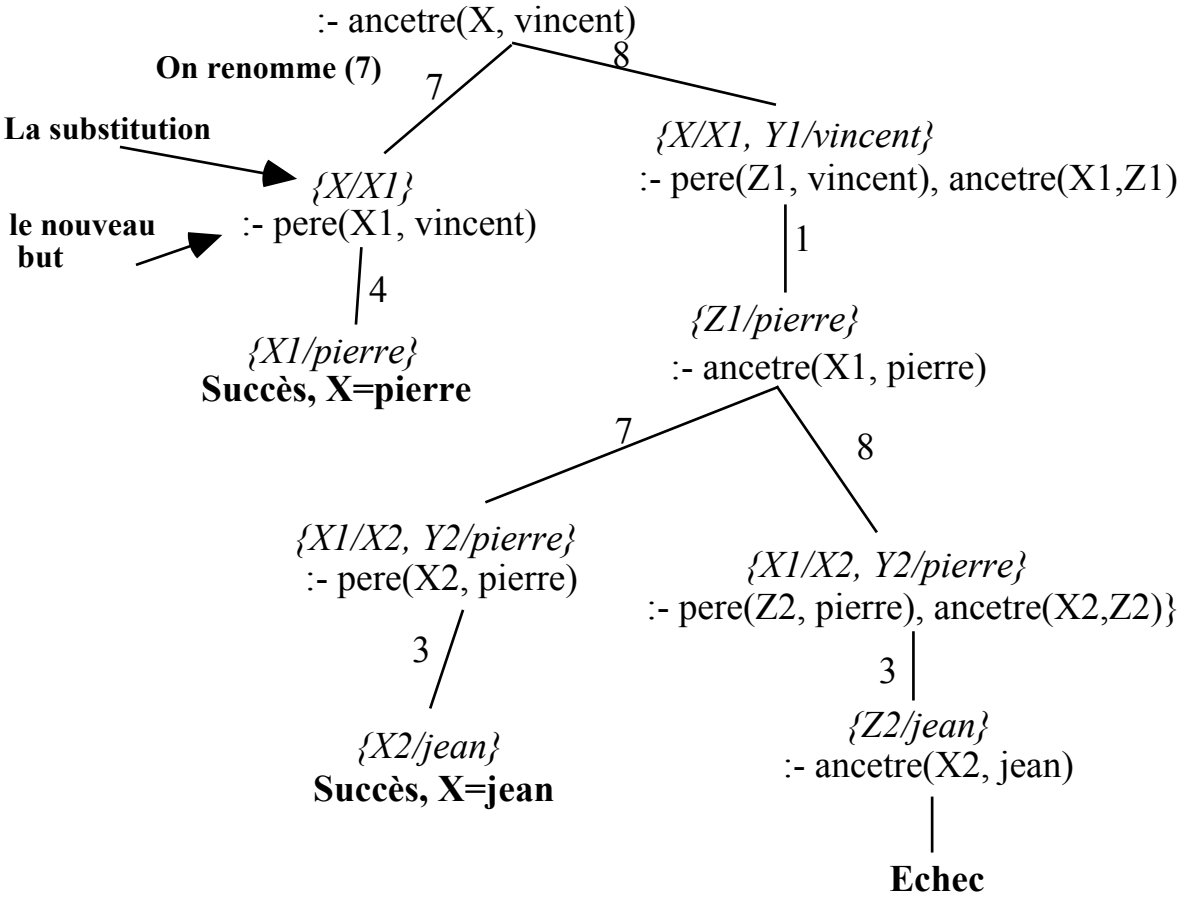
(6) pere(jacques,michel).

(7) ancetre(X,Y):- pere(X,Y).

(8) ancetre(X,Y):- pere(Z,Y) , ancetre(X,Z).

La question :

?- ancetre(X,vincent).



Hypothèse du monde fermé et 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:

femme(X) :- ***not*** *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.

Remarque sur la syntaxe : On peut écrire *not p* par **\+p**.

Contrôle de la résolution

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

Coupe-choix(!)

- 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)

Classification des cuts : Cut **vert** et Cut **rouge**

L'utilisation courante de cut

Cut peut être utilisé en trois types de situations.

Il peut être utilisé pour "dire" à Prolog :

- "En arrivant là (sur le cut), vous avez pris la règle qu'il fallait pour satisfaire ce but".

```
present(X, Liste) :-
        decompose(Liste, Tete, Reste), X=Tete, !.
present (X, Liste) :-
        decompose(Liste, Tete, Reste), present (X, Reste).
```

- "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).

```
diagnostic :-
    symptôme_négatif_présent,
    ! , fail.
diagnostic :-
    sympthôme_négatif_forcément_absent,
    continuer.
```

- "En arrivant là (sur le cut), vous avez 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.
```

Exemples d'utilisation de cut:

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

choix1(X, Y) :- age(X) , taille(Y).
choix2(X, Y) :- ! , age(X) , taille(Y).
choix3(X, Y) :- age(X) , ! , taille(Y).
choix4(X, Y) :- age(X) , taille(Y) , !.
```

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*.

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 n'essaye pas (2)

(3) q :- k , n.

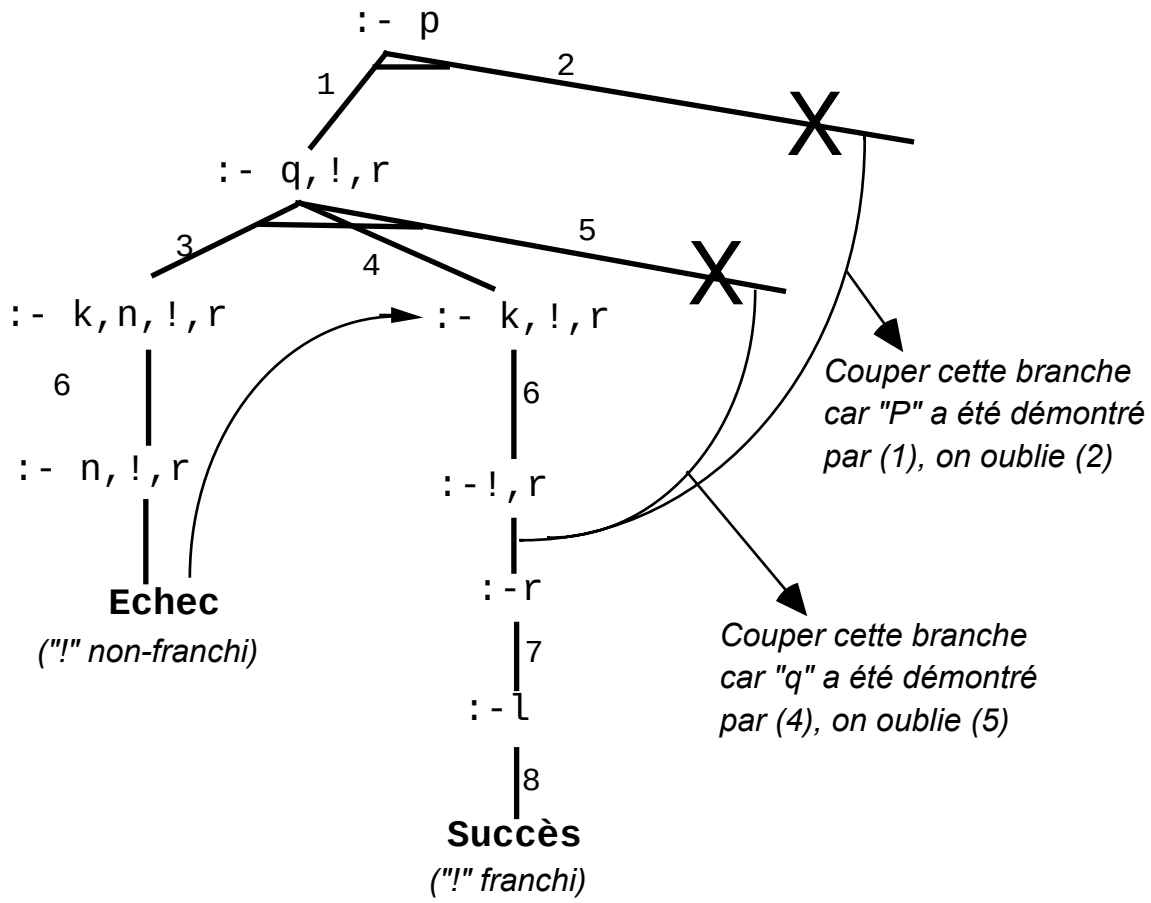
(4) q :- k.

(5) q.

(6) k.

(7) r :- l.

(8) l.



Effet de coupe-choix dans l'arbre de résolution

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)]]

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

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 = []

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 θ.

- **X \= Y** (ou \+***(X=Y)***) X et Y ne s'unifient pas (θ =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= '<'

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.

=:=	égalité
=\=	différence

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.

Exemples :

:- X is 2 , Y is X+1. => X=2, Y=3

:- 2 is 5-3 => succès

:- 4+3 =:= 10-3 => succès

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.

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

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

- Métaprédicats toutes-solutions

  bagof(Terme, But$_X$, Liste_Résultat).

  setof(Terme, But$_X$, Ens_Résultat).

  findall(Terme, But$_X$, Liste_Résultat).

- Schéma générateur / testeur

  pourtout(Q,P) :- not (Q , not P).

NB : voir le polycopié pour plus de détails.

Manipulation d'arbres

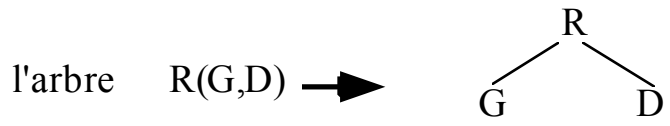
Exemple : parcours d'arbres binaires

Recherche et insertion dans un arbre binaire ordonné horizontalement

Convention:

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

  info(sag) < info(N) < info(sad).
- La représentation de différentes configurations:



Les feuilles (pas de descendant) :

F([], []) => F

```
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

CHAPITRE 2

Systèmes d'inférence

et

Stratégies de Recherche

Systèmes d'inférence

- Méthodes de preuve par Confirmation / Contradiction
- Exemple :

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

- Pour affirmer la proposition *aime(jean, marie)* par **confirmation**, on applique la règle *modus ponens* : *{B, (A <- B)} |- A.*
  Cette règle déduit toutes les conséquences du programme ci-dessus et permet de confirmer les faits suivants dont *aime(jean, marie)* fait partie : *aime(marie, logique) et aime(jean, marie)*

- Pour démontrer *aime(jean, marie)* par **contradiction**, on utilise la règle *modus tolens* : *{~A, (A <- B)} |- ~B.*

  Du même programme, on obtient par l'application de *modus tolens* :
  {~aime(jean, marie) & (aime(jean, marie) <- aime(marie, logique))}
  |- ~aime(marie, logique)

- Prolog utilise 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)

- Apporter la preuve par contradiction de la proposition A dans un programme P, c'est de prouver : $P \cup \{\sim A\} \models \Diamond$

  Donc, au lieu de dériver A de P, on peut dériver une contradiction de P & ~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 toutes 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 modus ponens dans une preuve par confirmation;

  => Prolog utilise les clauses du 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 ! "*

  - Utilisation dans diagnostic (de pannes) basé sur les symptômes.

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 :

  B → B1, B2, .. , Bn → B'1, .., B'm, B2, ... , Bn → ... → ◊

- Ils existent plusieurs façons de dériver la clause vide "◊" par la résolution (i.e. : 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 (i.e. trouver toutes les réponses).
- Une des méthodes de résolutions appliquée aux clauses définies est la **SLD-Résolution**.
- 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.
- 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 :

Q1 : <- gp(amenda, Z). gp **: grand parent**

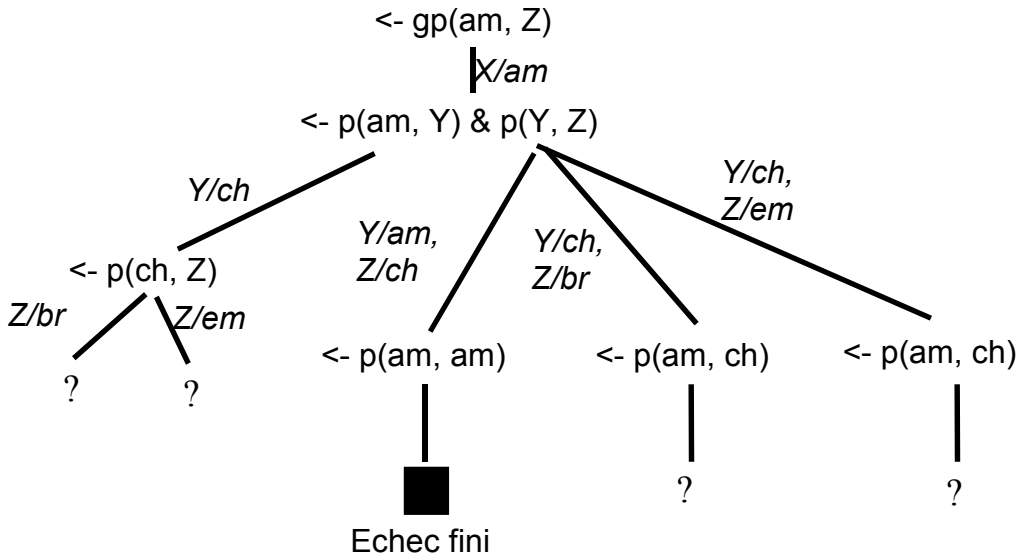
C1 : gp(X,Z) :- p(X, Y) & p(Y, Z). p **: parent**

C2 : p(amenda, christophe).

C3 : p(christophe, brigitte).

C4 : p(christophe, emile).

- 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 (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 :

  $Q : <- A_1 \&...\& A_i \&...\& A_n = \sim A_1 \mid ... \mid \sim A_n$ le résolvant

  $C : A <- B_1 \&...\& B_m$ une clause du programme

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

$Q' : <- (A_1 \&...\& A_{i-1} \& B_1 \&...\& B_m \& A_{i+1} \&...\& A_n)$ θ

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**.

Unification

L'unification de deux termes t1 et t2 consiste à trouver un unificateur le plus général θ tel que $t_1\theta = t_2\theta$.

L'algorithme suivant (algorithme de Robinson) produit une substitution θ (≠ 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 : ajouter {T1/T2} à θ; retourne(θ)
- T2 est une variable : ajouter {T2/T1} à θ; retourne(θ)
- T1 est une constante ou T2 est une constante : retourne(NULL);
- 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)

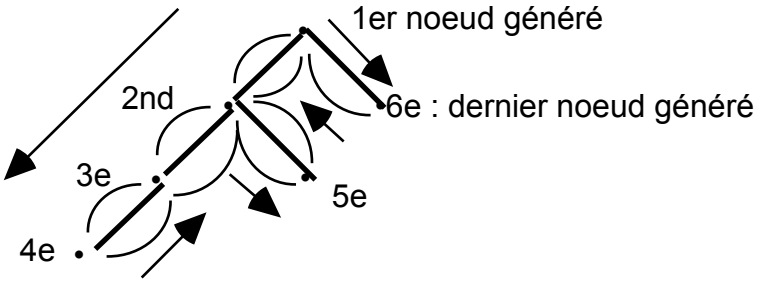
Fin cas

Fin unifier;

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 préservant la complétude de la SLD-résolution.

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 nœuds (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.

Algorithme de résolution de Prolog

- Soit le but **B= $b_1, b_2 , ... , b_n$** et un programme **P**
- Appel : θ <- résoudre (B , {});
- Retour : Si θ = NULL alors **échec**

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

Fonction résoudre (B : but ; θ : substitution) : substitution =

Si B = {} alors retourne (θ); Fin si;

(1) Considérer b_1 dans B; b_1 de la forme **$f(a_1,..., a_m)$**

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(**$\theta(a_i)$, $\theta(a'_i)$, θ**)** pour i=1..m

Si θ ≠ NULL Alors

B' = **A1, A2, ..., Ak** , b_2 , ... , b_n

θ <- résoudre(θ(B'), θ)

(3) Si θ ≠ NULL Alors retourne (θ) ; Fin si;

Fin si;

Fin Tant que;

Retourne (NULL);

Fin résoudre;

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

Cette version est déterministe en (3).

Un méta-interpréteur de Prolog en Prolog

Remarque :

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

```
effacer((B ,BS)) :-
    effacer_un_but(B) , effacer(BS).
effacer(B) :-
    not (B= (_, _)) , effacer_un_but(B).

effacer_un_but(true).
effacer_un_but(B) :-
    clause(B,Corps),
    effacer(Corps).
```

Question :

En Prolog : **?- la_question.**

Ici : **?- effacer(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.

Méthodes de résolution de problèmes : Stratégies de contrôle

Algorithme de base

MT : mémoire de travail = base de données initiale (état initial);

MP : mémoire de règles = base de règles (transitions, productions);

Condition d'arrêt : un fait (un état) qui dénote la fin du traitement

```
Tant que MT ne satisfait pas la condition d'arrêt
    CS <- trouver_candidats (MT, MP)
    R <- choisir (CS)
    MT <- appliquer(R, MT)
Fin TQ;
```

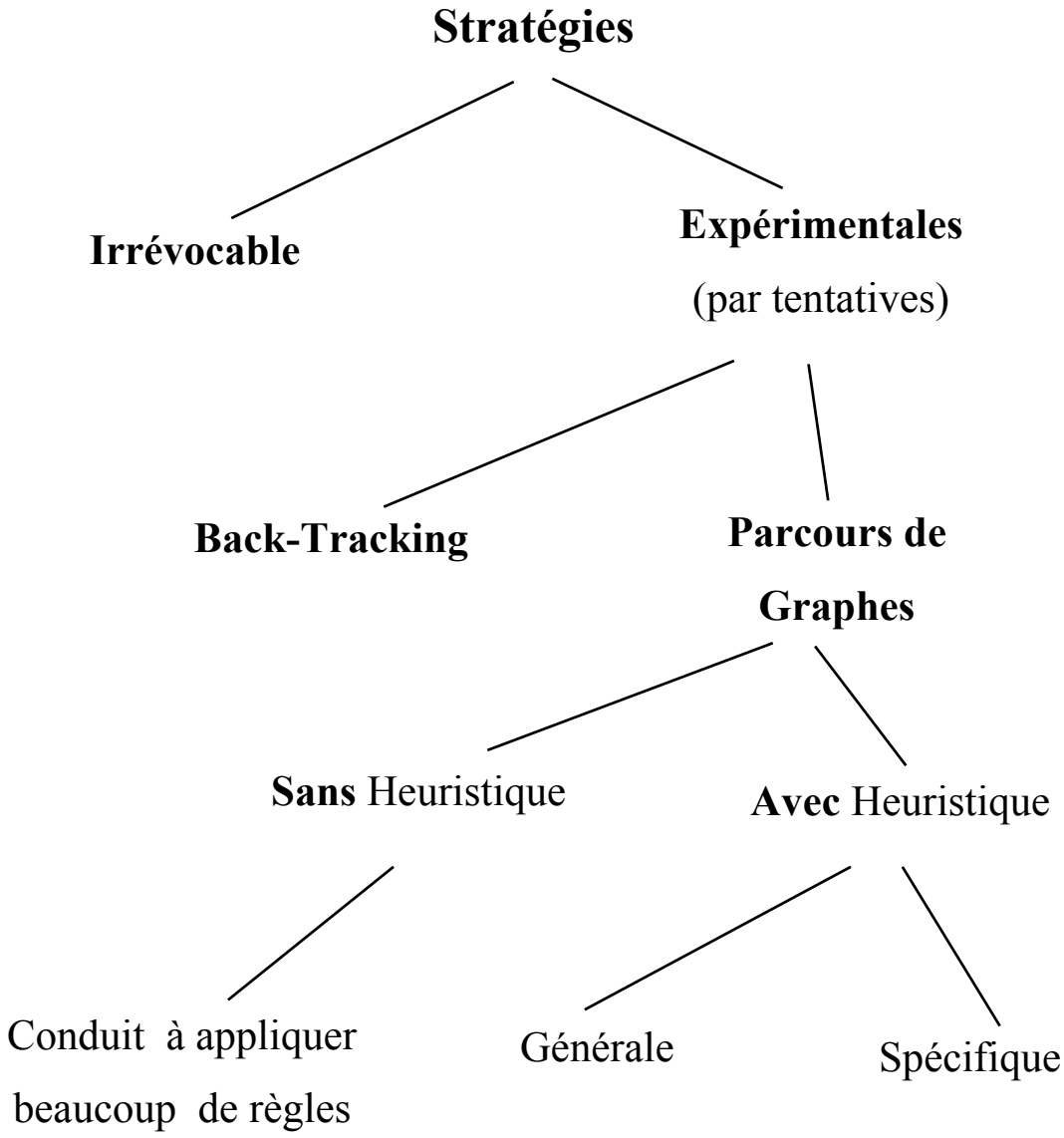
• Le choix des règles + la mémorisation des règles appliquées définissent la stratégie du système de contrôle (du système de production).

=> Le travail d'un système de production est un travail de recherche.

• Dans la plupart des applications IA, le système de contrôle ne dispose pas d'information satisfaisante pour choisir la bonne règle.

=> Le rôle du système de contrôle est un processus de recherche qui consiste à déterminer une suite de règles qui d'un MT0 initiale nous fait passer à MTf finale telle que MTf satisfait le problème.

Stratégies de Recherche



Remarques sur le graphe des stratégies :

- Dans chaque nœud de ce graphe, il y a toute la base de connaissance.

  => Une situation souhaitée serait d'avoir des règles qui décrivent les composants affectés par les changements d'états.

- Méthodes applicables à des problèmes <u>révocables</u> (domaines dans lesquels on peut défaire) :

  - Diagnostic médical
  - Robotique mobile, ...

Nous verrons plus loin les algorithme de décomposition et de planification appliquée aux méthodes révocables.

- Le retour arrière (Back Tracking) correspond à un schéma général de stratégie par essais successifs. Nous avons étudié un exemple de cette stratégie (cf. robot à la recherche de la cuisine depuis le salon).

  Les autres techniques, y compris les techniques de recherche informées utilisent le retour arrière.

- Dans une stratégie Irrévocable, on peut disposer d'une information qui permet de développer un arbre de recherche sans devoir faire des retours arrières. Cette information (locale) permet d'employer un schéma, par exemple Hill Climbing (voir plus loin) pour choisir une règle à appliquer à chaque étape sans retour arrière. Nous illustrons ce propos après avoir traité la stratégie Hill Climbing (voir plus loin).

Graphe et arbre de recherche

Avant de développer les stratégies de recherche, il convient de préciser les différences entre un "arbre" et un "graphe" de recherche, termes fréquemment utilisés dans les algorithmes qui suivent..

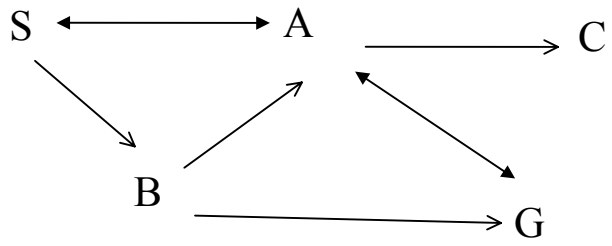
Graphe : les exemples sont souvent présentés sous forme d'un graphe qui précise les nœuds et les arcs. Un graphe peut être circulaire, avec ou sans cycle, connexe, fortement connexe, etc.

Les algorithme définissent (engendrent) potentiellement des graphes. Ils décrivent toute sort de développement, chemin (avec ou sans boucle), etc. Dans un graphe, on peut avoir des nœuds déjà visités, un nœud peut être le successeur de plusieurs nœuds,...

Parmi les chemins développés par les algorithmes et illustrés dans les graphes, seuls certains remplissant des conditions précises nous intéresseront : ce sont des arbres de recherche.

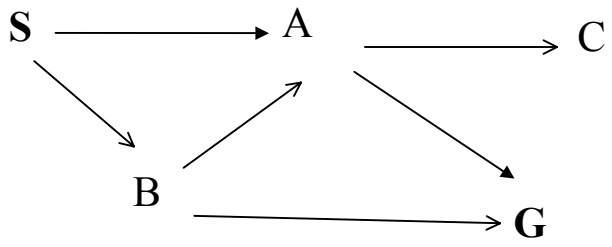
Un **arbre** est un graphe connexe sans circuit : tout nœud d'un arbre de recherche (sauf la racine) est le successeur d'un seul nœud. Ce nœud n'est généré que si son unique parent est développé (étendu). Un arbre de recherche est un sous ensemble d'un graphe de recherche. L'ensemble des nœuds d'un graphe de recherche et celui de son arbre correspondant sont identiques.

Un exemple : soit le graphe :



Pour aller de S à G, un arbre de recherche peut être (stratégie quelconque)

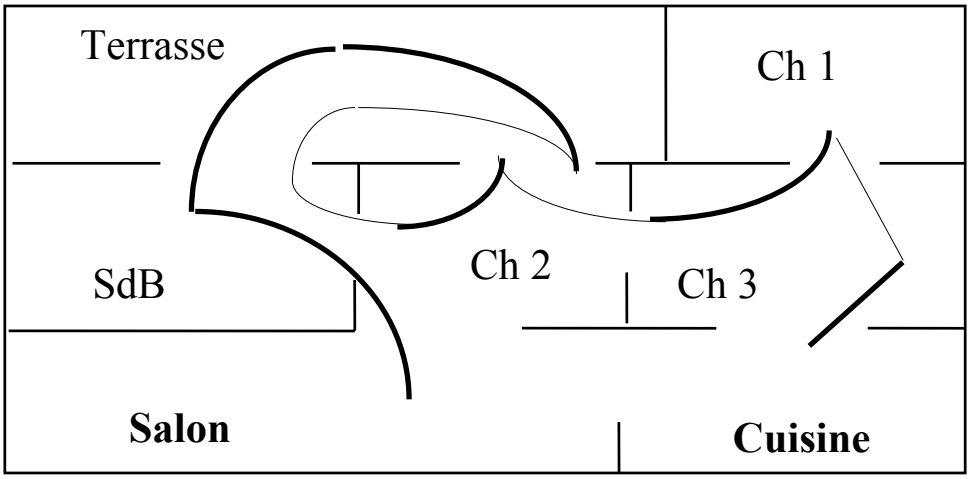
:



Cet arbre qui montre les nœuds visités a pu être développé par un algorithme de recherche qui évite les boucles.

Techniques de recherches : parcours de graphes

Une stratégie générale



But du robot : aller du salon à la cuisine (les lignes fines : backtrack)

Une stratégie ad hoc depuis le départ (visite dans le sens d'horloge) :

1. Marquer chaque porte comme "sortie"
2. Examiner la pièce actuelle
3. Si arrivée alors STOP.
4. Sinon, s'il y a des portes marquées "sortie" dans la pièce (**choix)**
   - 4a. Choisir une porte marquée "sortie"
   - 4b. Supprimer sa marque "sortie" (**Récupération mémoire)**
   - 4c. Emprunter cette porte
   - 4d. Si une des portes de cette pièce est déjà marquée "entrée" alors retourner dans la pièce précédente et aller à l'étape 4.**(boucle)**
   - 4d. Sinon marquer "entrée" la porte qu'on vient d'emprunter
   - 4e. Marquer toutes les autres portes "sortie"
   - 4f. Aller à l'étape 2.
5. Sinon emprunter la porte marquée "entrée" et aller à l'étape 4. (**ret_arr)**

Principe de l'algorithme général de recherche

Soit Agenda une liste de buts à prouver (effacer).

Agenda ← question initiale (le but).

1. Prendre le prochain nœud B d'Agenda (sortir B d'Agenda)
2. Si B = le but (état final) alors Stop
3. Sinon
   - 3a. Générer les successeurs de B
   - 3b. Mettre ces successeurs dans Agenda
   - 3c. Aller en 1.

Ce qui donne en Prolog :

```
% recherche(Agenda, Goal)   Goal est un nœud But à atteindre.
%                          C'est l'un des descendants des nœuds d'Agenda

recherche (Agenda, Goal):
        prochain(Agenda, Goal, Reste),          % Choix
        final(Goal).                            % état final atteint

recherche (Agenda, Goal):-
        prochain(Agenda, Courant, Reste),       % Choix
        successeurs(Courant, Successeurs),
        ajouter(Successeurs, Reste, NewAgenda),
        recherche (NewAgenda, Goal).
```

Plusieurs stratégies de recherche sont obtenues à partir de ce schéma de base.

Algorithmes de recherche purs : Schémas "En-Profondeur" et "En-Largeur"

La stratégie "en profondeur d'abord"

- La stratégie "en profondeur d'abord" est une variante de la stratégie de base précédente où l'on s'éloigne du but initial et l'on y revient qu'en cas de retour arrière.

- En Prolog :

```
% recherche en profondeur : à la Prolog
recherche_prof([Goal | Reste], Goal) :-
        goal(Goal).                          % Premier de la liste
recherche_prof([Courant | Reste], Goal):-
        successeurs(Courant, Successeurs),
        concat(Successeurs, Reste, NewAgenda),  % ou "append"
        recherche_prof(NewAgenda, Goal).
```

Version utilisant la pile de Prolog (au lieu d'Agenda)

On peut se servir de la pile utilisée pour la gestion des appels récursifs :

```
% recherche en profondeur avec retours arrières.
recherche_bachtrack(Goal, Goal):-
        final(Goal).
recherche_bachtrack t(Courant, Goal):-
        arc(Courant, Suivant),              % Retours Arrières
        recherche_bachtrack(Suivant, Goal).
```

Un exemple : le monde des blocs

On considère :

• une table sur laquelle on a 3 positions distinctes;

• un certain nombre de blocs sur cette table où ils sont empilés les uns sur les autres.

Comment déplacer ces bloc à partir d'un état initial pour obtenir un état final donné.

Seuls les blocs qui sont libres (avec aucun bloc sur eux) peuvent être déplacés.

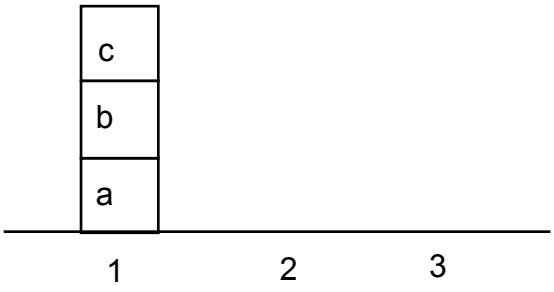
La représentation des états et des transitions

• Un terme fonctionnel ***etat/3*** pour représenter les trois positions de la table.

• Une constante *table* dénote la table.

• Enfin, *sur(X,Y)* représente le fait que X est sur Y.

Exemple: **etat(sur(c,sur(b,sur(a,table))), table, table)**
représente la configuration (l'état) :



où *sur(c, sur(b, sur(a, table)))* représente l'état de la position 1 et *table* et *table* ceux des positions 2 et 3.

Les Transitons :

Les actions générales suivantes sont possibles pour transformer les états:

• si la première position est occupée, le bloc le plus haut de cette position peut être déplacé vers la deuxième ou la troisième position;

• si la second position est occupée, le bloc le plus haut de cette position peut être déplacé vers la première ou la troisième position;

• si la troisième position est occupée, le bloc le plus haut de cette position peut être déplacé vers la première ou la second position.

Traduction en Prolog :

La première action peut s'exprimer par les deux clauses suivantes :

```
deplacer(  etat(sur(X, Nouv_X), Anc_Y, Z),
                etat(Nouv_X, sur(X, Anc_Y),Z)).
deplacer(  etat(sur(X, Nouv_X),Y, Anc_Z),
                etat(Nouv_X,Y,sur(X,Anc_Z))).
```

Remarque :

deplacer(X,Y) = une transition.

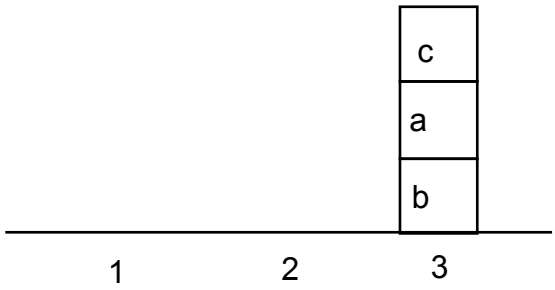
=> *déplacer(X, Y)* a le même sens que le prédicat général *arc(X,Y)* du parcours de graphe.

⇨ on complète les deux autres actions d'une manière analogue.

Le code ci-dessus du parcours en profondeur (recherche_bachtrack = prof_bt) utilise un accumulateur pour éviter les boucles. Il utilise la pile Prolog pour les retours arrières.

```
prof_bt (X,Y,Plan) :- prof_bt (X,Y,[X], Plan).

prof_bt (X,X,Plan,Plan).
prof_bt (X,Z,Memoire,Plan) :-
        arc(X,Y),  not membre(Y, Memoire),
        prof_bt (Y,Z,[Y|Memoire],Plan).
```

Exemple, si l'état final recherché est :



On posera la question (aucun état n'a été figé) :

```
?- prof_bt (etat(sur(c,sur(b,sur(a,table))), table, table),
            etat(table, table, sur(c,sur(a,sur(b,table)))) ,X).
```

Une réponse est :

```
X = [   etat(table, table, sur(c,sur(a,sur(b,table))))
        etat(table, sur(c,table), sur(a,sur(b,table))),
        etat(sur(a,table), on(c,table), sur(b,table)),
        etat(sur(b,sur(a,table)), sur(c,table),table),
        etat(sur(c,sur(b,sur(a,table))), table, table)].
```

- déplacer **c** en position 2;
- deplacer b en position 3 et le bloc **a** sur le bloc **b**
- deplacer **c** sur **a**.

Améliorations du parcours "en profondeur"

Avant d'étudier ces amélioration, définissons le prédicat d'obtention de tous les successeurs dont on aura besoin plus bas.

Obtention de tous les successeurs d'un nœud :

Afin de procéder aux divers traitements (tri, sélection, etc) des successeurs, on peut obtenir la liste des successeurs d'un nœud.

Appliquée à un graphe avec la relation **arc** entre les nœuds, les successeurs peuvent être trouvés (d'un seul coup) par le prédicat **findall**. (*findall* est comme bagof mais en cas d'échec du but appelé, le résultat de findall sera une liste vide).

```
successeurs(Noeud, Successeurs):-
    findall(C,  arc(Noeud,C), Successeurs).
```

On verra plus loin l'utilisation de la liste complète des successeurs.

1• On peut produire le trajet en cas de succès.

Au lieu de mettre des simples nœuds dans Agenda, on peut y ranger le trajet partiel depuis le début jusqu'à ce nœud. Le prédicat successeurs qui fournit la liste des successeurs d'un nœud devient :

```
successeurs([Noeud | Chemin], Successeurs):-
    findall([C,  Noeud | Chemin], arc(Noeud, C), Successeurs).
```

Dans ce cas, la question initiale doit être posée sous la forme :

```
?- recherche_prof([[Noeud_Initial]],CheminAuBut).
```

2• Seconde amélioration : détection des boucles par l'examen des nœuds déjà visités. On ajoutera à Agenda seuls les nœuds non encore visités (et ceux qui ne sont pas déjà dans Agenda actuel).

```
successeurs([Noeud | Chemin], Successeurs):-
    findall([C,  Noeud | Chemin],
            (arc(Noeud, C), not membre(C, [Noeud | Chemin])),
            Successeurs).
```

Le prédicat membre est définit dans la partie Prolog.

Si le chemin partiel n'est pas disponible, il faudra ajouter un paramètre supplémentaire qui contiendra l'accumulation des nœuds visités.

3• Une dernière amélioration : au lieu de conserver le trajet partiel pour chaque nœud dans Agenda, il suffit de conserver le nœud et son parent. Lorsque le But est atteint, tous ses parents sont dans la liste des nœuds déjà visités. On pourra alors reconstruire le trajet.

Variations de la recherche "en profondeur"

- Le problème de boucle peut être résolu par le contrôle de la profondeur du graphe.
- Soit le prédicat général de recherche en profondeur (avec backtrack)

```
prof_backtrack(But, But) :-
    final(But).                    % état final
prof_backtrack(Courrant, But) :-
    arc (Courrant, Suivant),       % état intermédiaire (transition)
    prof_backtrack (Suivant, But).
```

- On introduit un niveau de profondeur par le paramètre N :

```
prof_niv(N, But, But) :-
    final(But).
prof_niv(N, Courrant, But) :-
    N > 0, N1 is N-1,
    arc (Courrant, Suivant),       % état intermédiaire
    prof_niv (N1, Suivant, But).
```

- Contrairement à *prof_backtrack*, prof_niv s'arrête mais les solutions au delà du niveau imposé peuvent être ignorées.
- Une amélioration de prof_niv est le contrôle du niveau variable (**iterative deeping**). Dans cette procédure, après avoir tenté la recherche avec un niveau limité à N, le niveau est itérativement augmenté de D et la recherche reprend.

- Pour une variation de D=1, on aura :

```
prof_niv_iter(Premier, But) :-
    prof_niv_iter(1, Premier, But) .

prof_niv_iter(N, Courrant, But) :-
    prof_niv(N, Courrant, But).        % appel de prof_niv

prof_niv_iter(N, Courrant, But) :-
    N1 is N+1,
    prof_niv_iter(N1, Courrant, But).
```

- Un avantage important de *prof_niv_iter* par rapport à *prof_backtrack* est que cette variation itérative de niveau donne une procédure complète (trouve les réponses s'il en existe) : la recherche prouvera les buts d'un niveau N et inférieur avant d'aller au niveau N+D.

- De plus, pour D=1, *prof_niv_iter* est optimal car il trouvera les chemins les plus courts.

- Un inconvénient de cette méthode est l'examen répété des niveaux supérieurs de l'arbre de recherche.

La stratégie Hill Climbing (HC)

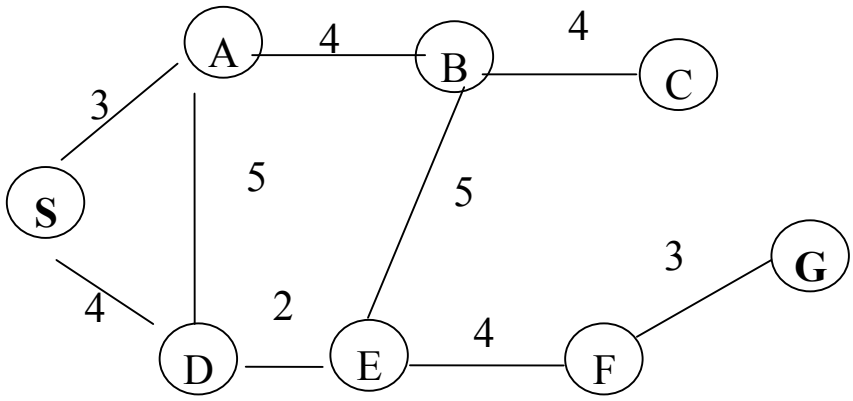
- Est une technique de parcours en profondeur d'abord informé.

- Pour se déplacer dans un arbre de recherche par cette technique, on procède comme dans la technique "en profondeur d'abord" mais on ordonne les successeurs selon une mesure heuristique de la distance jusqu'au but à atteindre.

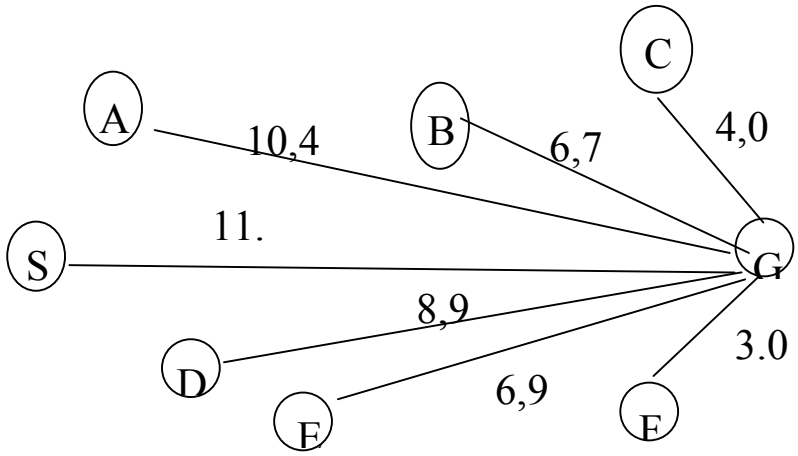
- Pour une heuristique néant, on aura le parcours en profondeur.

- La ligne droite (ou la distance à vol d'oiseaux) jusqu'au but est un exemple d'heuristique.

Exemple : soit le graphe suivant où l'on cherche un chemin de S à G.



Supposons les distances à vol d'oiseaux (ou ligne droite) suivantes :



Pour atteindre le nœud G depuis S, il faut se trouver dans un nœud proche de G.

"C" est un nœud proche mais il n'y a pas de lien de C à G.

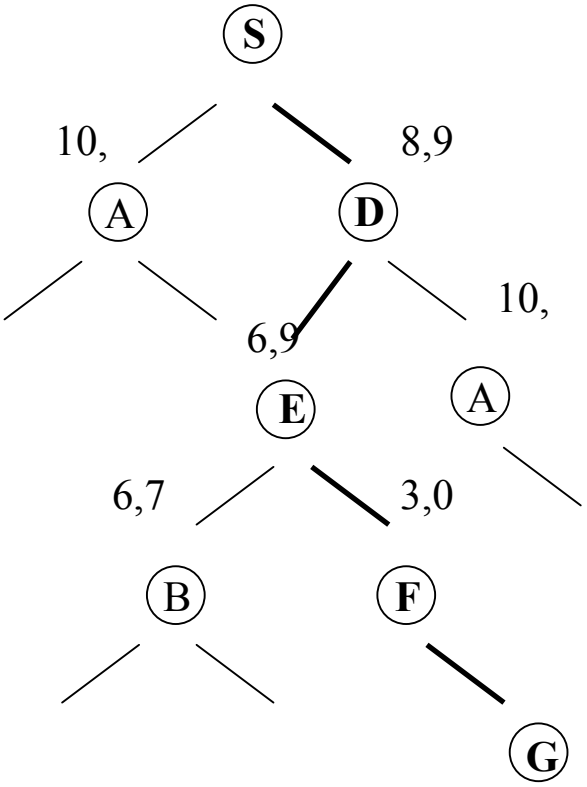
Application de Hill Climbing en partant de S pour aller à G :

- adjacents(S) = {A, D} mais D est plus proche de G que A; D est examinés d'abord;
- adjacents(D) = {A, E} mais E est plus proche de G que A (à vol d'oiseaux);
- adjacents(E) = {B, F}, on choisit F;
- adjacents(F) = {G}; le but
- succès

Remarque : ce parcours tient évidemment compte des circuits.

Exemple

L'arbre de recherche suivant donne le parcours :



Le chemin en gras dans ce graphe ressemblerait à un parcours de dunes !

L'algorithme HC

L'algorithme suivant utilise une pile pour trouver un chemin depuis le départ jusqu'au But. La pile contient les chemins partiels parcourus :

1- Pile = [Départ]
2- Tant que sommet(Pile) ≠ But et Pile ≠ vide
2.1- Chemin = dépiler; Chemin est une liste L de nœuds
2.2- Etendre Chemin en y ajoutant tous les adjacents du tête(L)
2.3- En supprimer tous les chemins contenant une boucle
2.4- Trier ces nouveaux chemins sur la distance au But
2.5- Empiler le résultat
3- Si le But est atteint
Alors succès; sommet(Pile) = le chemin recherché
Sinon échec.
Fin Hill Climbing

Les inconvénients de HC

L'algorithme de Hill Climbing souffre de plusieurs problèmes qui le rendent peu efficace dans la version précédente.

Ces inefficacités sont plus évidentes lorsque Hill Climbing est utilisé pour optimiser des paramètres.

Une méthode d'optimisation des paramètres consiste à se servir de certains outils pour "ajuster les paramètres" de la recherche (le parcours) en observant certaines quantités qui nous permettrons de mesurer la qualité des performances associées aux différents réglages des paramètres qu'on effectuera.

Exemple : supposons que l'on grimpe une montagne lorsqu'un brouillard épais apparaît. On ne dispose pas de plan ni de sentier à suivre. Mais on a un compas, un altimètre. Le but est d'atteindre le sommet.

L'application de "l'ajustage des paramètres" à cet exemple serait la suivante : le paramètres ajustable est notre emplacement et on peut se servir de l'altimètre pour savoir si on progresse vers le sommet.

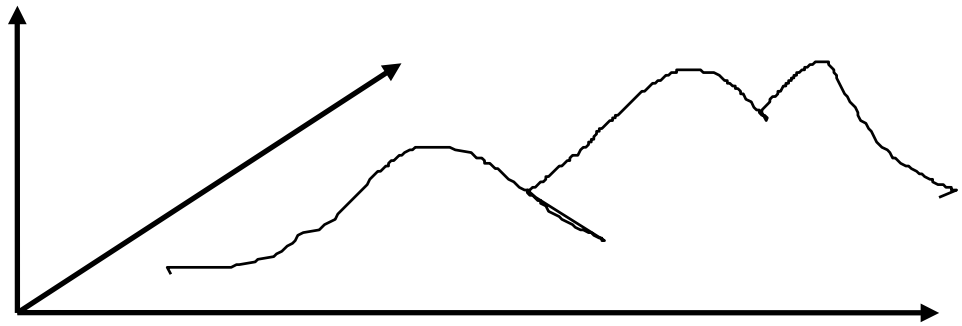
Pour grimper selon la méthode "orienté paramètres" de Hill Climbing, on peut essayer de faire un pas vers le nord, reculer et faire un pas vers l'est. Puis on fera un pas vers le ouest et enfin vers le sud. On choisira ensuite le pas qui augmente le plus notre altitude.

On répétera cette action jusqu'à ce que toutes les tentatives diminuent notre altitude : ce sera le sommet.

D'une manière plus générale, pour effectuer un Hill Climbing "orienté paramètres", on applique une étape d'ajustement, haut et bas, à chaque paramètre, on prend le meilleur résultat (selon la mesure de la qualité ou des performances) et on répète jusqu'à trouver une combinaison des paramètres qui produit la meilleur performance (qualité) par rapport à tous les alternatives possibles.

On peut cependant rencontrer de sérieux problèmes avec cette méthode :

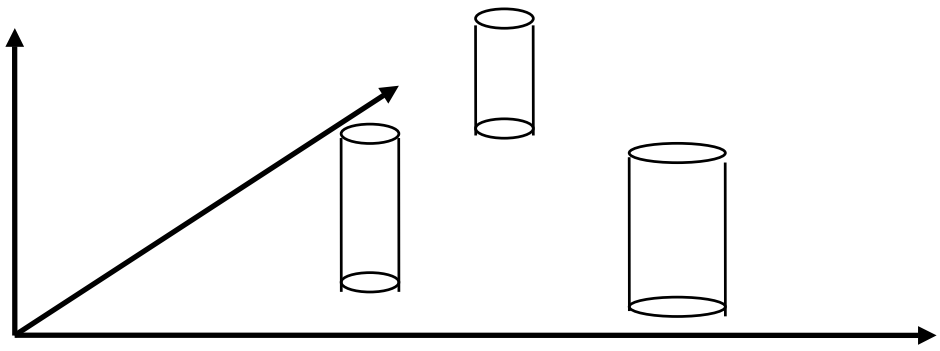
1- on risque de trouver un "foothill" lorsqu'il y a des sommets secondaires comme dans la figure suivante :



Dans ce cas de figure, on peut trouver un optimum local au lieu d'un optimum global alors que la solution trouvée semble correcte. Des retours arrières éventuels (recherche non déterministe) pour tenter d'autres possibilités risquent d'aboutir au même maximum local. En toute rigueur, la méthode naïve de Hill Climbing donnerait dans ce cas de figure de meilleurs résultats que la méthode orientée paramètres.

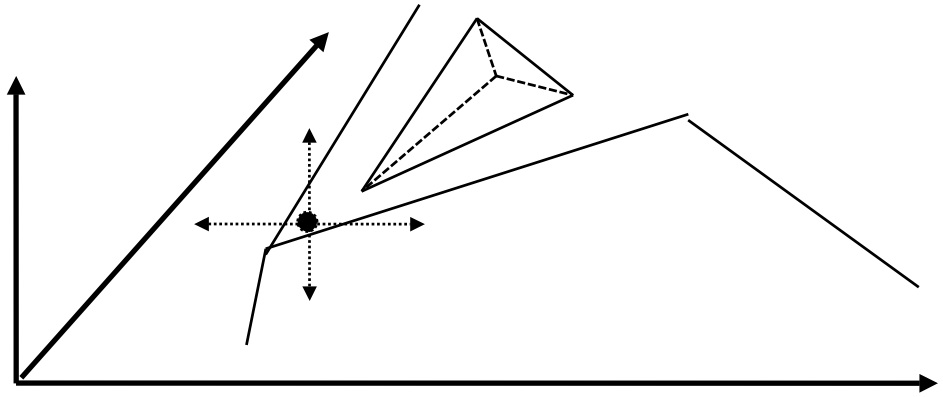
Cependant, en utilisant des retours arrières, avec un nombre aléatoire de pas de tailles aléatoires et dans des directions aléatoires pourraient nous sortir d'un maximum local.

2- Le problème de "plateau" comme dans la figure suivante :



où un plateau entoure les sommets. Dans ce cas, l'amélioration local de la méthode n'est pas utile et ne modifie presque rien.

3- Il y a un troisième cas (ridge) plus subtile. On peut imaginer, sur une route en spiral montante autour d'une montagne, alors qu'on est coincé dans un angle très fermé. Les 4 directions habituelles rencontrent une baisse d'altitude.



Depuis l'emplacement marqué d'une croix (les 4 directions), toute tentative donne une baisse de qualité (altitude).

Une solution à ce problème serait d'essayer plusieurs directions supplémentaires (plus de 4).

Codage de l'algorithme HC

Pour la méthode orientée paramètres, les directions (alternatives d'une étape) sont les adjacents possibles d'un nœud. On se donnera une fonction de mesure de qualité pour choisir la meilleur direction (prédicat "choisir_meilleur").

Principe de codage

- Soit le nœud N le premier élément du sommet de la Pile des trajets.
- Si N=but alors succès; le chemin est au sommet de la Pile.
- Sinon
    . Trouver les adjacents du nœud N;
    . Classer les adjacents en mesurant les performances (e.g. distance au
        . but) tel que le meilleur successeur de N soit en tête;
    Refaire.

Remarque : Si le meilleur choix n'aboutit pas alors on essayera le 2nd.

Remarque : Si l'on se contente d'un seul meilleur choix, on aura une méthode "irrévocable".

Version utilisant la pile de la récursivité :

```
recherche_hc(Goal,Goal):-
    goal(Goal).
recherche_hc(Courant,Goal):-
    successeurs(Courant, Successeurs),
    choisir_meilleur(Successeurs, Best),
    recherche_hc(Best, Goal).
```

Remarque : si "choisir_meilleur" est déterministe, "recherche_hc" sera irrévocable.

Version utilisant une pile avec gestion des chemins partiels :

```
recherche_hc([[X|Xs]|_], X, [X|Xs]).
recherche_hc([[X|Xs]| Xs1], Y, Plan) :-
    findall([N,X|Xs], (arc(X,N),not membre(N,[X|Xs])), Fils_de_X),
    classer(Fils_de_X, Fils_de_X_classes),
    concat(Fils_de_X_classes, Xs1,  Z),
    recherche_hc(Z,Y,Plan).
```

Le prédicat "classer" ordonne la liste des successeurs de X telle que le premier successeurs répond aux critères de sélection du prédicat "choisir_meilleur" ci-dessus. La version ci-dessus tient compte de tous les descendants (révocable).

L'action de "findall" :

L'appel à "findall" permet de trouver les successeurs N du nœud X (ceux qui ne sont pas dans le chemin partiel "départ –> X") et d'étendre ce chemin partiel en y ajoutant les nœuds N. Si X possède k successeurs {N1, N2, ..., Nk}, il y aura k extensions du chemin partiel {départ –> N1, .., départ –> Nk}, lesquelles remplaceront "départ –> X" dans la pile.

Pour plus de détails sur "findall", voir la stratégie "en largeur" ci-dessous.

HC et la stratégie irrévocable

Dans une stratégie de contrôle **irrévocable**, un seul chemin dans l'arbre de recherche est suivi.

Une telle stratégie possède suffisamment d'information pour résoudre un problème de sorte que l'on sache appliquer une transition à chaque étape et aboutir à une solution (à un échec ou à l'absence de solution). Cette stratégie aurait alors la solution du problème traité (ainsi que d'autres). Dans ce cas, on ne peut pas véritablement dire que l'on a résolu le problème car la solution était en quelque sorte déjà connue et codée dans l'algorithme même de recherche.

De telles stratégies (générales) applicables aux problèmes complexes n'existent pas (cf. les problèmes NP-Complets).

Par contre, on peut distinguer entre une information **locale** explicite permettant d'effectuer des étapes vers une solution et une information **globale** implicite sur la solution. Lorsqu'une information locale infaillible et sûre est disponible, on peut appliquer une stratégie irrévocable pour construire une information globale explicite permettant d'aboutir à une solution et ce sans avoir eu une connaissance globale de la solution.

Le schéma HC est un cas typique d'utilisation d'information locale pour construire une solution globale. Dans ce schéma, on examine à chaque étape les différentes "directions" (informations locales) pour

trouver éventuellement le maximum d'une fonction (information globale).

Pour certaines fonctions (celles avec un seul maximum possédant également d'autres propriétés), la connaissance de la bonne "direction" est suffisante pour trouver une solution.

Le schéma HC peut ainsi être appliqué dans une stratégie irrévocable.

Exemple d'application au Taquin

Une stratégie irrévocable utilisant le schéma HC exploite à chaque étape la plus grande valeur de la fonction à maximiser pour effectuer une transition, de façon irrévocable.

Pour illustrer le propos, reprenons l'exemple classique du Taquin dont la complexité du graphe des possibilités en fait un exemple non trivial.

Cet exemple sera complément traité plus loin.

Pour résoudre le puzzle Taquin par une stratégie irrévocable avec le HC, on utilisera la négation du nombre de pions mal placés comme la fonction que le schéma HC maximise à chaque étape.

Les "directions" d'un schéma HC sont les mouvements possibles et le HC choisit celui qui maximise la valeur de la fonction ci-dessus.

La valeur du But est 0.

Si dans un état, la valeur de la fonction ne peut pas être améliorée, on appliquera une transition que ne diminue pas cette valeur.

La résolution s'arrête si une telle transition n'est pas possible; un tel état équivaut à un maximum (local).

La figure suivante donne les différents états de la résolution ainsi que la valeur de cette fonction.

La progression gauche droite et du haut vers le bas illustre l'application de la stratégie irrévocable.

Dans la figure, la valeur de la fonction à maximiser est notée au dessus de chaque état.

-4

2	8	3
1	6	4
7		5

-3

2	8	3
1		4
7	6	5

-3

2		3
1	8	4
7	6	5

-2

	2	3
1	8	4
7	6	5

-1

1	2	3
	8	4
7	6	5

0

1	2	3
8		4
7	6	5

Comme nous l'avons vu plus haut, le schéma HC souffre de plusieurs cas d'insuffisance. Le cas ci-dessous montre le problème de multiple maxima locaux où la fonction associée à HC ne peut améliorer sa valeur. Dans ce cas, l'état initial est un maximum local.

1	2	5
	7	4
8	6	3

état initial

1	2	3
	7	4
8	6	5

état final

Remarque :

L'exemple de Taquin sera traité complément plus loin.

Stratégie "en largeur d'abord"

- On peut gérer l'Agenda selon le mode FIFO (queue, file).
  => La stratégie "en largeur d'abord".

- Complétude de la stratégie, coût

- "En largeur" de base en Prolog

```
% Recherche en largeur d'abord (version de base)

recherche_largeur([Goal | Reste],Goal):-
        goal(Goal).

recherche_largeur([Courant|Reste],Goal):-
        successeurs (Courant, Successeurs),
        concat (Reste, Successeurs, NewAgenda),      % Ajout en queue
        recherche_largeur(NewAgenda,Goal).
```

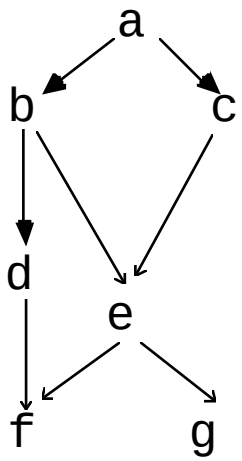
- On peut appliquer les variations de niveaux à cette stratégie.

- Les détails de la stratégie en largeur sont donnés dans la suite.

Un exemple détaillé

• Dans un parcours en largeur d'abord, pour un nœud donné, on construit l'ensemble de ses successeurs que l'on ajoute à l'ensemble d'états non encore explorés.

```
arc(a,b).
arc(a,c).
arc(b,d).
arc(b,e).
arc(c,e).
arc(d,f).
arc(e,f).
arc(e,g).
```



Pour la question **?-chemin(a,g)** l'état de la file est :

[a]	a n'est pas la destination, enlever a, ajouter ses descendants (b et c) en queue de la liste.
[b,c]	b n'est pas la destination, l'enlever et ajouter ses descendants (d et e) en queue de la liste [c]
[c,d,e]	enlever c, ajouter ses descendants [e]
[d,e,e]	enlever d, ajouter [f]
[e,e,f]	enlever e, ajouter [f , g]
[e,f,f,g]	enlever e, ajouter [f , g]
[f,f,g,f,g]	enlever f, ajouter []
[f,g,f,g]	enlever f, ajouter []
[g,f,g]	=> g = la destination, **succès**

• L'ordre : a→b→c→d→e→e→f→f→g

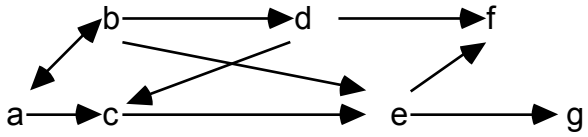
Le prédicat *chemin_largeur* :

```
chemin_largeur(X,Y) :-
        chemin_l([X], Y).

chemin_l([X|_],X).                      % 1er but = l'état final
chemin_l([X|Xs],Y) :-
        findall(N, arc(X,N), Fils_de_X),  % Tous les successeurs
        concaten(Xs, Fils_de_X, Z),       % Ajout en queue
        chemin_l(Z,Y).
```

- La liste des successeurs est construit à l'aide du prédicat **findall**.
  Format : findall(X,But,Toutes_les_réponses).

- Le sommet à explorer X est le premier élément de la liste des sommets à explorer.
- Si X est égal au sommet à atteindre Y, le but chemin_largeur(X,Y) réussit; sinon l'on retire X de la liste des sommets, on calcule ses successeurs et on les met à la queue de la liste des sommets à explorer.
- La résolution se terminera quand cette liste de sommets sera vide.
- On pourra modifier le programme pour tenir compte des doublons que l'on traite plusieurs fois.

Cas de graphe avec circuits



La relation arc est donnée par:

arc(a,b).
arc(b,a).
arc(d,c)
.....

A la question ?-chemin_largeur(a,d).

On obtient une infinité de réponses:

[a] → [b,c] → [c,a,d,e] → [a,d,e,e] → [d,e,e,b,c]
Une infinité d'autres réponses sont possibles car
a↔b et c↔d

L'exploration des sommet ne se terminera pas car la liste des sommets n'est jamais égale à la liste vide.

L'intérêt d'un parcours en largeur d'abord par rapport au parcours en profondeur d'abord est que, même si le graphe possède des circuits, on trouvera au moins une solution lorsqu'elle existe.

Gestion des circuits

Gestion simple :

```
chemin_largeur(X,Y) :-
    chemin_l([X], Y,[]).

chemin_l([X|_],X,_).
chemin_l([X|Xs],Y,Visites) :-
    findall(N,
        (arc(X,N), not membre(N,Visites)),    % éviter les boucles
        Fils_de_X),
    concaten(Xs,Fils_de_X, Z),
    chemin_l(Z,Y,[X|Visites]).
```

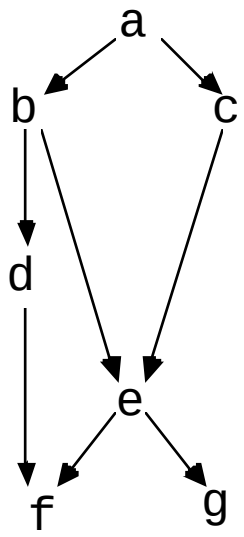
La liste *Visites* contient les nœuds que l'on a visités avant d'arriver au point X.

Ainsi, pour éviter de traiter un état déjà traité, on regarde si celui-ci n'est pas dans la liste *Visites* auquel cas on ne l'insère pas.

Génération du plan de parcours

On maintient, pour chaque nœud à explorer, tout le chemin parcouru depuis le début jusqu'à ce nœud.

```
arc(a,b).
arc(a,c).
arc(b,d).
arc(b,e).
arc(c,e).
arc(d,f).
arc(e,f).
arc(e,g).
```



Pour la question ?-chemin(a,e), on obtient successivement :

[[a]]

[[b,a],[c,a]]

[[c,a],[d,b,a],[e,b,a]]

[[d,b,a],[e,b,a],[e,c,a]]

[[e,b,a],[e,c,a],[f,d,b,a]]

⇨ e = la destination, **succès**

Le prédicat *chemin_largeur* final est :

```
chemin_largeur(X,Y,Plan) :- chemin_l([[X]], Y, Plan).

chemin_l([[X|Xs]|_], X, [X|Xs]).
chemin_l([[X|Xs]|Xs1],Y,Plan) :-
    findall([N,X|Xs], (arc(X,N),not membre(N,[X|Xs])), Fils_de_X),
    concaten(Xs1,Fils_de_X, Z),
    chemin_l(Z, Y, Plan).
```

Stratégie Beam Search (recherche partielle)

- Une variante de la stratégie en largeur

- Développement de l'arbre de recherche avec une limite sur les nœuds de l'Agenda.

  => seule une partie de l'espace est recherchée.

- Si l'on limite la taille de l'Agenda à 1, on aboutit à la stratégie **Hill Climbing**.
- Beam search est une variante de la stratégie "en largeur d'abord" et progresse par niveau. L'arbre de recherche est développé en ne conservant seulement D **meilleurs** successeurs à chaque niveau, les autres successeurs seront ignorés.

Détails de codage de beam search

- On peut maintenir 2 Agendas : un pour le niveau courant et un pour les successeurs des nœuds du niveau courant.

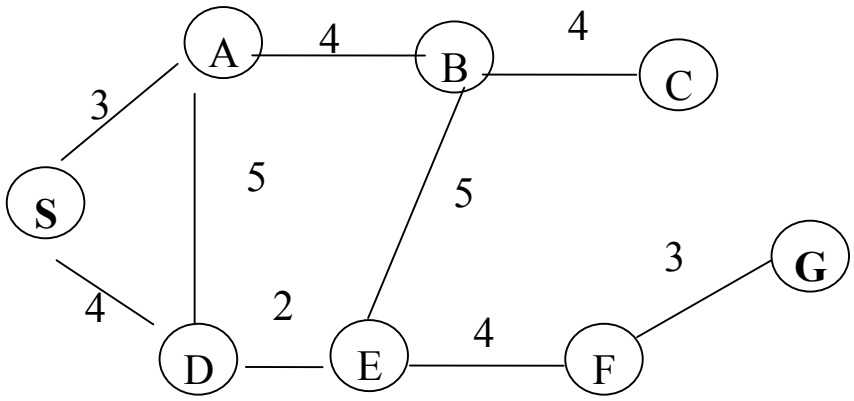
  Lorsque le niveau actuel est couvert, les Agendas seront permutés et le compteur du niveau est augmenté.

  Le niveau peut être passé en paramètre au prédicat d'ajout des successeurs à l'Agenda pour décider combien de meilleurs successeurs ajouter à Agenda du prochain niveau (le niveau peut définir le nombre des successeurs).
- En Prolog : le paramètre D détermine le nombre de nœuds à retenir.

recherche_beam(Agenda, Goal):-

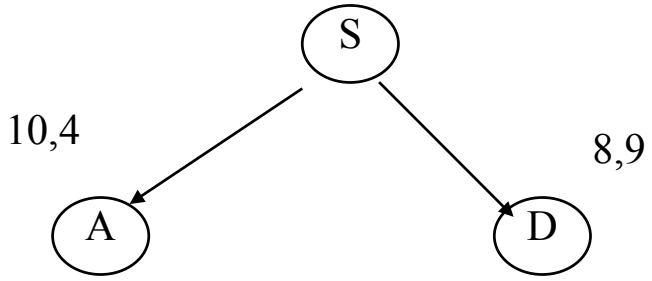
```
        recherche_beam(1, Agenda, [],  Goal).
recherche_beam(D,[], Prochain_Niveau, Goal):-      % permutation
        D1 is D+1,
        recherche_beam(D1, Prochain_Niveau, [], Goal).

recherche_beam(D, [Goal | Reste], Prochain_Niveau, Goal):-
        etat_final(Goal).                          % succès
recherche_beam(D, [Courant | Reste], Prochain_Niveau, Goal):-
        successeurs(Courant, Successeurs),
        ajouter_meilleurs(D, Successeurs,
                          Prochain_Niveau,Prochain_Niveau_bis),
        recherche_beam(D, Reste, Prochain_Niveau_bis, Goal).
```

Exemple : soit le graphe suivant où l'on cherche un chemin de S à G.

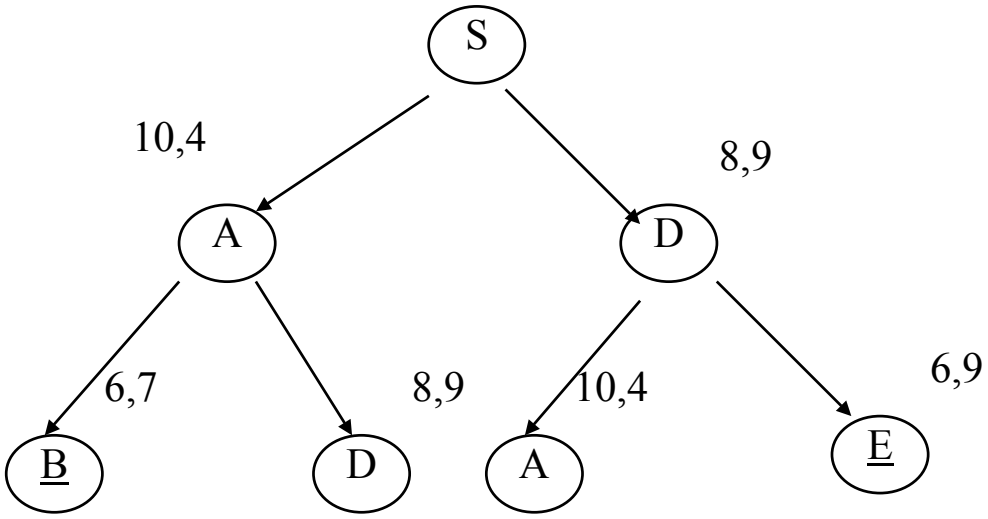


Le critère de choix des meilleurs nœuds sera la distance à vol d'oiseaux.

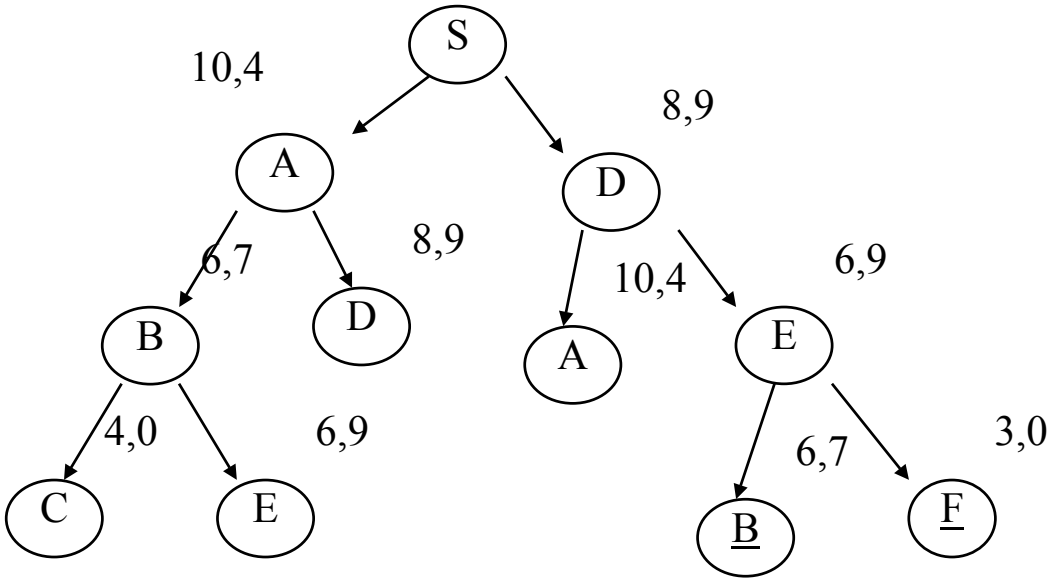
Les divers développements avec le nombre de nœuds développés D= 2 sont donnés ci-dessous dont le développement du nœud S donne :



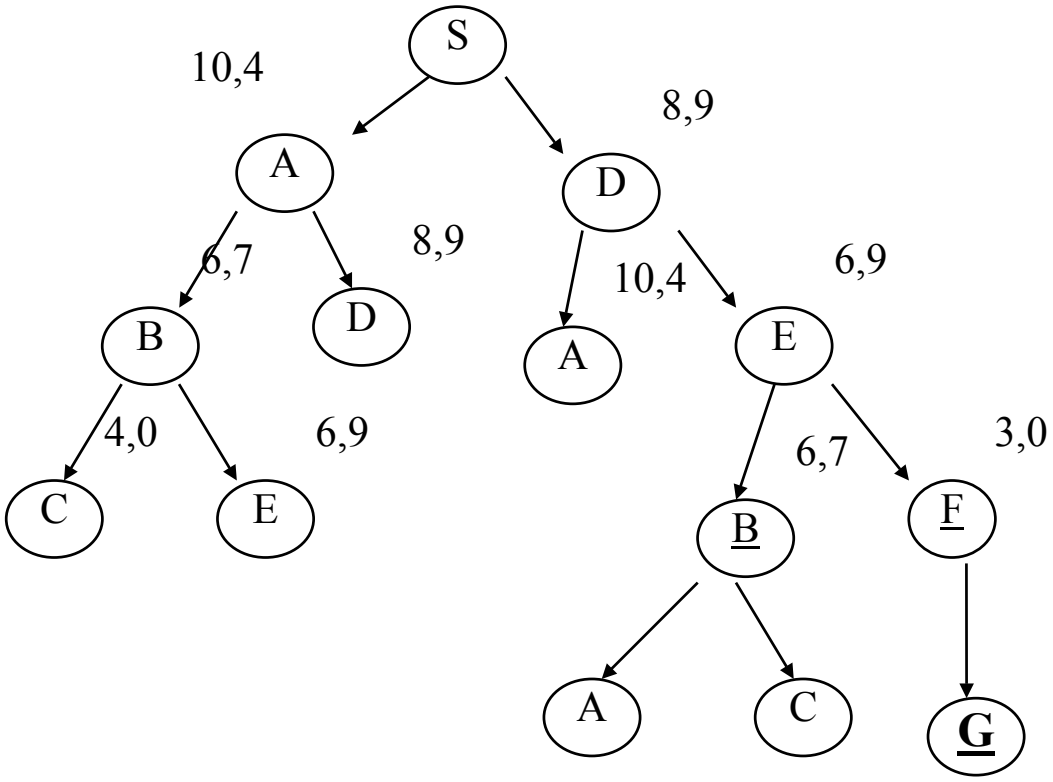
Deuxième niveau : on conserve les deux meilleurs nœuds B et E. Les deux autres sont rejetés.



Troisième niveau : le nœud C n'aboutit à rien (sans successeur). On conservera les nœuds F et B.



Quatrième niveau : le nœud F aboutit au but G.



Stratégies informées non exhaustives

- Les stratégies vues jusqu'ici sont exhaustives
- Elles exploitent les espaces complets de recherche (dans la cas pire).
- Elles ajoutent **tous** les descendants d'un nœud à Agenda.
- La taille de l'Agenda augmente exponentiellement avec le trajet.
- Les heuristiques permettent une sélection des nœuds.
- Ces stratégies ne sont pas forcément complètes; il faut une bonne heuristique informée.

Un principe : "Pour trouver une meilleure méthode de recherche pour un problème, trouver un autre espace de recherche pour ce problème!"

- Dans un parcours **aveugle**, on ne fait aucune hypothèse sur la plausibilité d'un nœud qui pourrait mener à une solution.

- Dans une stratégie informée, on exploite certaine information (sous forme d'une fonction **h**) appelée **heuristique**.

Cette information permet de savoir par exemple à quelle distance du but (état) final se trouve l'état actuel.

Elle peut être utilisée pour ordonner / trier les nœuds et en choisir le meilleur : méthode Best-First

Stratégie Best-First (BF)

- Soit le prédicat **eval/2** qui pour un nœud donné renvoie la distance estimée entre ce nœud et le but final.
- Soit Agenda une liste de sous buts (résolvants).
  pour un nœud donné, les successeurs sont ajoutés à l'Agenda.

Prédicat best-first :

```
best_first([Goal | Reste],Goal):-        % Le premier but
    etat_final(Goal).                    %  état final atteint

best_first([Courant|Rest],Goal):-
    successeurs (Courant,Successeurs),
    ajouter_bstf (Successeurs, Reste, NewAgenda),
    best_first (NewAgenda, Goal).
```

ajouter_bstf (New_successeurs, Agenda, New_Agenda)

peut fonctionner comme une procédure insere_trié : insertion un par un et à leur place des successeurs dans Agenda (selon eval/2).

Exemple d'application

On considère le tableau ci-dessus (état initial)

- **Le but** (état final) est de déplacer les pions de sorte que les pions noirs soient à droite des blancs.
- La position du la case vide sera sans importance.

- **Le mouvement autorisé** : un pion va dans la case vide en sautant au plus 2 autres pions.

- Le coût d'un déplacement =
  - 1 s'il n'y a pas de pion sauté;
  - N sinon (N = le nombre de pions sautés $\neq 0$).

- L'espace d'états
  - les nœuds sont les différentes positions du tableau
  - un arc (une transition) est un déplacement de pion (mouvement).

- Représentation des données :
  On peut présenter le tableau par une liste de **b**, **w** et **e** :
  - b : black
  - w : white
  - e : empty

  par exemple, la figure précédente : [b, b, b, e, w, w, w]

Utilisation de la procédure Best First (BF)

- On construit une séquence de déplacements par lesquels le but sera atteint.

- Les nœuds examinés pendant la recherche sont collectés dans une liste appelée *Visitées*.

  Lorsqu'une position de But (état final) est trouvée, la solution (le trajet et son coût) est construite à partir de la liste Visitées.

- Les éléments d'Agenda sont sous la forme de paires

  val(Valeur, Déplacement)

  où Valeur est l'évaluation heuristique de la position atteinte par Déplacement.

- Les successeurs d'un nœud sont générés par setof/3 qui produit une liste **triée**.

  Pour obtenir le tri sur les valeurs, on place Valeur au début du terme val,

  Ainsi, la liste des successeurs sera simplement "fusionné" avec Agenda (pas besoin de tri par insertion).

- Un déplacement de OldPos vers NewPos est représenté par le triplet :

  m(OldPos, NewPos, Cout)

  où Cout est le coût donné plus haut.

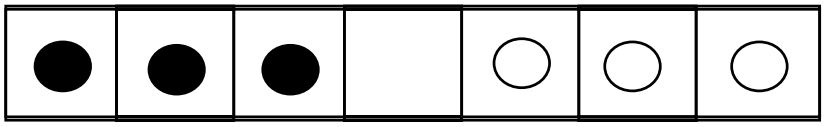
Exemple de fonction heuristique

- On compte, pour tout pion blanc, le nombre de pions noirs à sa gauche.

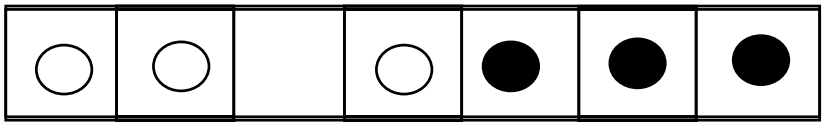
  Pour l'état final (les pions noirs à droite des blancs), ce nombre = 0.

  Ceci caractérise l'état final.

- L'état initial : **start(m(noparent, [b,b,b,e,w,w,w], 0)).**



- Exemple d'état final :



Exemple de question posée :

?- pions([b, b, b, e, w, w, w], M, C).

Réponses produites :

[b, b, b, e, w, w, w] - 9

[b, b, b, w, e, w, w] - 9

[b, b, e, w, b, w, w] - 8

[b, b, w, w, b, e, w] - 7

[b, b, w, w, b, w, e] - 7

[b, b, w, w, e, w, b] - 6

[b, e, w, w, b, w, b] - 4

[b, w, e, w, b, w, b] - 4

[e, w, b, w, b, w, b] - 3

[w, w, b, e, b, w, b] - 2

[w, w, b, w, b, e, b] - 1

M = [[b, b, b, e, w, w, w], [b, b, b, w, e, w, w], [b, b, e, w, b, w, w],

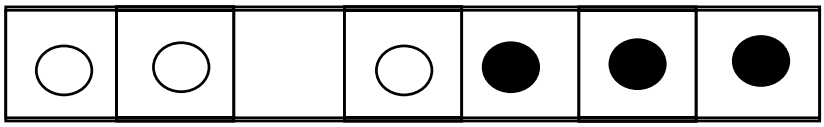
[b, b, w, w, b, e, w] , [b, b, w, w, b, w, e], [b, b, w, w, e, w, b] ,

[b, e, w, w, b, w, b] , [b, w, e, w, b, w, b], [e, w, b, w, b, w, b],

[w, w, b, e, b, w, b], [w, w, b, w, b, e, b], [w, w, e, w, b, b, b]]

C = 15.

Correspond à l'état final :



• Les déplacements considérés sont ceux qui aboutissent à la solution. On n'a donc pas besoin de retour arrière (une heuristique directe peut remplacer les retours arrières).

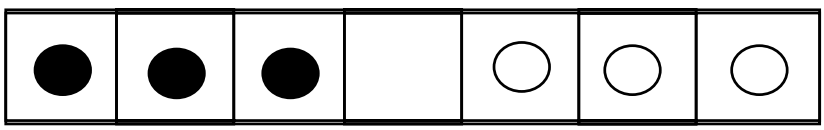
Une autre heuristique

• On peut compter les pions **mal placé**s :

=> Un pion mal placé sur le premier ou le 7e carrée donne 3;

sur le 2nd ou 6e donnerait 2 et

sur le 3e ou 5e carrée donnerait 1:



• On obtient pour ce cas une solution avec C= 14 (voir figure page

suivante, partie du milieu)

- Il y a quelque différences avec l'heuristique précédente :
  - on utilise les retours arrières
  - la solution nécessite 2 déplacements en moins (moins chère).

- Ceci permettrait de penser qu'un pion mal placé provoque une pénalité qui peut mener à une solution meilleure (moins chère).

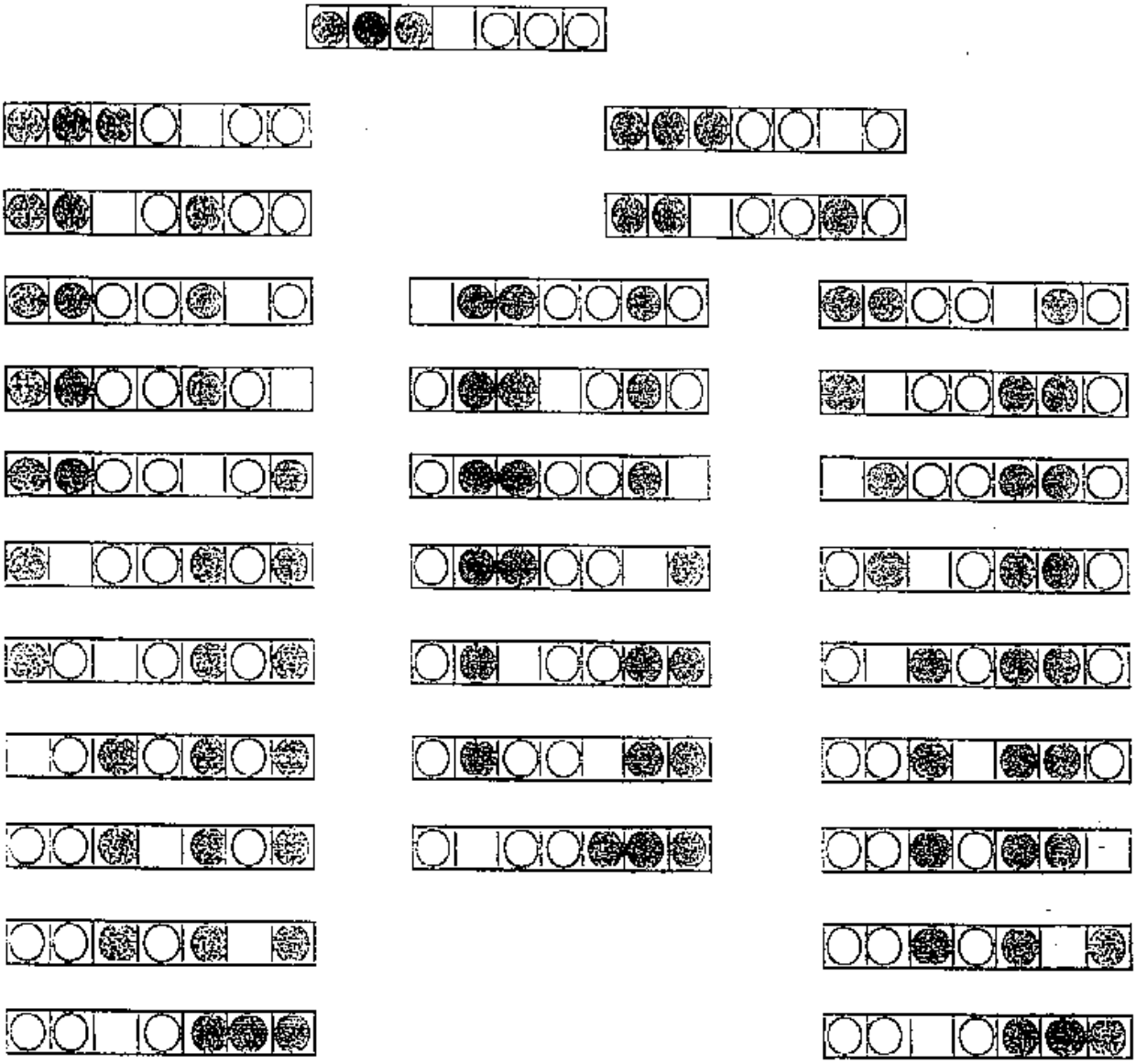
=> On décide d'augmenter la pénalité :

4, 3 et 2 au lieu de 3-2-1 ci-dessus !.

Ceci donnera une autre solution (voir figure milieu) avec C=15 et plus de déplacements (2 de plus) que le précédent.

Constat : Cette heuristique n'améliore pas le coût !

Question : y a-t-il une heuristique optimale ?



Stratégie Branch and Bound (B & B)

- La stratégie BF précédente s'inscrit dans un cadre plus général de la méthode **Branch & Bound**.
- Dans la méthode Branch & Bound, on s'intéresse à évaluer (et à obtenir en cas de succès) un **chemin** optimal. Les étapes du développement d'un arbre de recherche effectuent des extensions des **chemins partiels** jusqu'à là obtenus.
- Rappelons que pour obtenir le meilleur chemin, la méthode naïve de base (appelée la méthode "British Museum") extrait tous les chemins puis choisit le meilleur. La méthode (aveugle) de développement d'arbre de recherche employé peut être en profondeur ou en largeur.

Cette méthode se heurte évidemment au problème d'explosion combinatoire.

En supposant que le facteur de branchement **b** (nombre de successeurs d'un nœud) soit complètement uniforme, le nombre de chemin pour le niveau **n** de l'arbre de recherche sera $b^{\mathbf{n}}$.

Il y a (heureusement) des stratégies permettant de trouver le meilleur chemin sans les énumérer d'abord.

La méthode Branch & Bound

Cette méthode est similaire à la stratégie Best First mais classe les chemins partiels et fournit un chemin optimal (mais pas forcément le meilleur).

L'algorithme de principe de B&B

Pour aller du nœud *Départ* à un nœud *But* :

- Pile des chemins = chemin de longueur 0 contenant Départ

- Tant que Pile ≠ Vide et le nœud terminal N du sommet(Pile) ≠ But

- Enlever le premier chemin partiel CH de la Pile (dépiler)

- Etendre CH en ajoutant les successeurs de N

- Eliminer de ces nouveaux chemins ceux contenant une boucle

- Ajouter les chemins restant à la Pile

- Trier toute la Pile sur les coûts des chemins (longueur du chemin), le meilleur (minimum) en tête

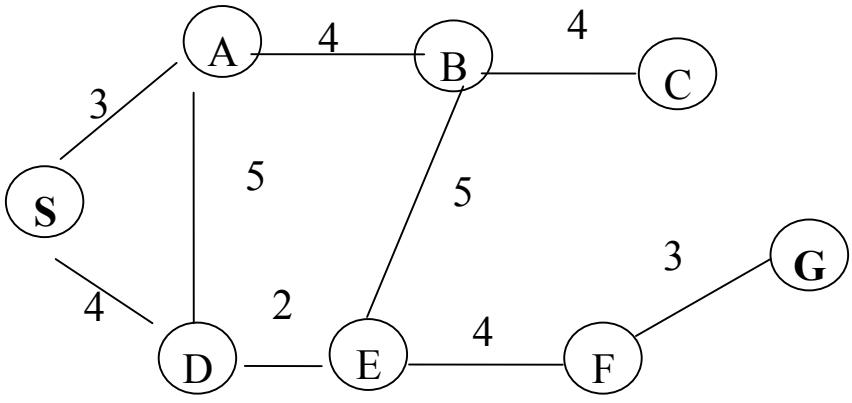
- Fin Tant que

- Si le But est atteint alors succès Sinon échec.

On remarque bien que cet algorithme est très similaire à celui de la méthode *Best First*.

La méthode B& B peut utiliser, pour évaluer le coût, une fonction heuristique **h** basée sur le trajet depuis le départ (la longueur du chemin) et compare les chemins partiels et fournit un des meilleurs chemins.

Exemple : on reprend l'exemple du graphe :



Les états successifs de la Pile sont donnés ci-dessous. Les chemins

contenant une boucles sont évités) :

• [[S]]

• [[SA3], [SD4]] *Développement de S*

• [[SAB7], [SAD8], [SD4]] *Développement de SA3*

• [[SD4], [SAB7], [SAD8]] *Tri*

• [[SDA9], [SDE6], [SAB7], [SAD8]] *Développement de SD4*

• [[SDE6], [SAB7], [SAD8], [SDA9]] *Tri*

• [[SDEB11], [SDEF10], [SAB7], [SAD8], [SDA9]] *Dévt.*

• [[SAB7], [SAD8], [SDA9] , [SDEF10], [SDEB11]] *Tri*

• [[SABC11], [SABE12], [SAD8], [SDA9] , [SDEF10], [SDEB11]] *Dévt*

• [[SAD8], [SDA9] , [SDEF10], [SABC11], [SDEB11] ,[SABE12]] *Tri*

• [[SADE10], [SDA9] , [SDEF10], [SABC11], [SDEB11] ,[SABE12]]

• [[SDA9] , [SADE10], [SDEF10], [SABC11], [SDEB11] ,[SABE12]]

• [[SDA9] , [SADE10], [SDEF10], [SABC11], [SDEB11] ,[SABE12]]

• [[SDAB13] , [SADE10], [SDEF10], [SABC11], [SDEB11] ,[SABE12]]

• [[SADE10], [SDEF10], [SABC11], [SDEB11] ,[SABE12], [SDAB13]]

• [[SADEB15], [SADEF14], [SDEF10], [SABC11], [SDEB11] , [SABE12] , [SDAB13]]

• [[SDEF10], [SABC11], [SDEB11] , [SABE12] , [SDAB13], [SADEF14], [SADEB15]]

• [[SDEF**G**13], [SABC11], [SDEB11] , [SABE12] , [SDAB13], [SADEF14], [SADEB15]]

Le chemin **S**DEF**G** est un des plus court chemin. Etant donné que B&B ne fournit pas forcément le meilleur chemin (critère local de

classement), on peut développer les autres nœuds disponibles (B et E). La recherche s'arrête lorsque les autres nœuds développés (SDEB et SABE, pour SABC, le nœud C est une impasse) donnent des chemins de longueur égales ou plus grande que **S**DEF**G** (SDEBA15, SDEBC15, SABED14 et SABEF16).

Stratégie B & B optimale (A, A*)

• B&B (ou Best First) est une stratégie exhaustive avec une variation, selon l'heuristique utilisée, de comportement allant de la stratégie "en profondeur" jusqu'à la stratégie "en largeur".

• Best First n'est pas complet : l'heuristique peut mener à une branche infinie en affichant le meilleur coût à chaque étape.

La raison en est que l'heuristique est basée sur la distance (locale) au But alors qu'on est intéressé par minimiser le coût total d'une solution donnée.

Pour B&B, nous avons vu que le chemin proposé n'est pas forcément le plus court.

• Pour obtenir une recherche en largeur complète, on peut utiliser une fonction d'évaluation **f** avec deux parties :

f(n) = g(n) + h(n)

h(n) est l'estimation heuristique pour atteindre le But depuis un nœud **n** (comme plus haut).

g(n) est le coût actuel pour atteindre le nœud **n** depuis le début.

La somme est utilisée pour ordonner les nœud de la liste Agenda.

• Un algorithme de recherche qui utilise une telle fonction d'évaluation **f** pour estimer le coût d'une solution est un **algorithme-A**.

• Un algorithme-A est complet puisque la valeur de la profondeur **g(n)**

permettra d'éviter une branche infinie.

Dans ce sans, **g(n)** permet d'avoir un algorithme "plus" en largeur.

• Un algorithme "en largeur" est une sorte d'algorithme-A avec **h(n)=0** pour tout nœud n => tous les nœuds ont la même h-valeur=0.

• Un inconvénient d'algorithme-A est sa faible efficacité du fait même qu'il est plutôt du type "en largeur".

La stratégie "en largeur" est non seulement complète, mais aussi optimale : elle produit toujours le meilleur chemin.

• L'algorithme-A hérite-t-il de cette propriété selon la fonction **h** ?

=> si un nœud *n1* sur le chemin le moins cher obtient une *h-valeur* très élevée, les autres nœuds seront traités d'abord et une solution non optimale peut être trouvée.

On dit que l'heuristique est **pessimiste** eu égard au *n1*.

• Inversement, une heuristique qui n'affecte jamais à un nœud une valeur plus élevée que le vrai coût d'une solution depuis ce nœud est **optimiste**.

Exemple du tableau précédent

- Soit la 1ère heuristique qui compte pour un pion blanc les pions noirs à sa gauche.
- Supposons qu'un pion noir ait **w** pions blancs à sa droite; ceci ajoute **w** à l'heuristique pour cette position (h-valeur).

Pour atteindre une position But, le pion noir doit sauter par dessus quelques uns des pions blancs alors que les blancs restants doivent sauter par dessus le pion noir.

Le coût de ces déplacements est au moins w => heuristique optimiste.

- La seconde heuristique qui calcule une somme pondérée des pions mal placés est également optimiste.

Exemple : un pion noir sur le 1er carré avec 3 blancs à sa droite par dessus lesquels il doit sauter (coût = 3) .

De même, un pion noir dans le 2nd carré => il y a au moins deux blancs à sauter (coût = 2).

- Par contre, les poids utilisés dans la 3ème heuristique sont trop élevés.

Un résultat important : A*

- **Un algorithme-A avec une heuristique optimiste trouve toujours une solution optimale.**

L'algorithme résultant est appelé **A* (A-star).**

- A* et les heuristiques optimistes sont appelés admissibles.

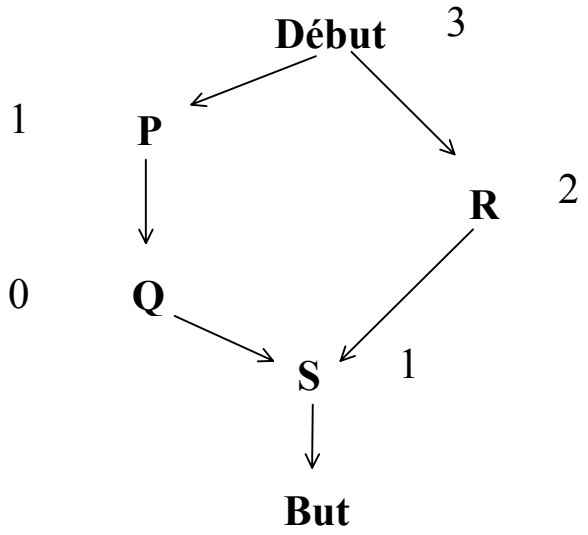
- Erreur à ne pas faire : les meilleures heuristiques ne sont pas les plus optimistes !

Une bonne heuristique est aussi pessimiste que possible sans devenir non-admissible.

En général, si h1(n) >= h2(n) pour tout nœud n, alors h1 est dite au moins aussi **informée** que h2.

Plus une heuristique est informée, moins l'on cherche dans l'espace des états.

Un exemple (heuristique non monotone)



f = g+h : Début_à_Nœud + Nœud_à_But

• Les **h**-valeurs indiquées au niveau des nœuds; le coût par arc est 1

• L'heuristique est optimiste et donc A* renvoie le meilleur chemin :

Début –> R –> S –> But

• Mais cette solution n'est pas trouvée immédiatement :

- [Début]
- $[P_{debut}, R_{debut}]$ l'indice désigne le père du nœud
  $[P_{debut}, R_{debut}]$ Agenda Ordonné car :
  $f(P_{debut}) = g(P_{debut}) + h(P_{debut}) = 1+1=2$
  $f(R_{debut}) = g(R_{debut}) + h(R_{debut}) = 1+2=3$
- P_{debut} traité, Agenda ordonné = $[Q_P, R_{debut}]$ $f(Q_P)=2$
- Q_P traité, Agenda ordonné = $[R_{debut}, S_Q]$ $f(S_Q)=4$
- R_{debut} traité, Agenda ordonné = $[S_R, S_Q]$ $f(S_R)=3$
- S_R traité, Agenda contient But => **FIN**

• Donc, même si les recherches admissibles aboutissent aux solutions

optimales, il n'est pas certain que les nœuds d'un chemin optimal soient immédiatement exploités sur ce chemin.

- Dans l'exemple du tableau des pions,

le problème vient du pessimisme local de l'heuristique qui évalue le coût du déplacement de Début à P par 3-1 = 2 (3 pour Début à But, 1 pour P à But) alors que le "vrai" coût est 1 (1 arc).

Si h(P) était = 2, S serait atteint la 1ère fois par le chemin le plus court.

- Ce qui est Vrai en général :

Si une heuristique estime le coût de déplacement d'un nœud à un autre de façon optimiste, alors les nœuds seront atteints d'abord par le chemin le moins cher.

Cette propriété est appelée la **monotonie** car on peut montrer que les f-valeurs sont monotones non décroissantes le long du meilleur chemin.

- La 1ère heuristique de l'exemple du tableau (pions) est monotone, la 2nde ne l'est pas. Ce qui peut être montré par les évaluations

[b, b, e, w, w, b, w] - 9 saut de b de 1 à 3

[e, b, b, w, w, b, w] - 7

L'heuristique estime le coût de ce déplacement par 9-7=2 alors que le coût actuel est 1.

- La monotonie implique l'admissibilité : la 3ème heuristique n'est pas monotone non plus (elle est pessimiste).

De B & B à A* : algorithmes

On peut présenter une approche constructive permettant de détailler l'algorithme et la stratégie A*.

Intuitivement, il semble évident que tenir compte d'une estimation de la distance restante au But et d'une estimation de la distance déjà parcourue peuvent donner de meilleurs résultats pour certains problèmes de recherche. Après tout, si une estimation de la distance restante est utile, alors l'ajout d'une estimation de la distance effective déjà parcourue doit l'être également.

Cette remarque permet de modifier l'algorithme de B& B en modifiant la ligne de classement :

le classement se fera sur la **somme** de la longueur du chemin partiel et de l'estimation de la borne inférieure de la distance restante (sous estimation). Si cette borne inférieure est proche de la réalité, alors on sera en présence d'une heuristique globale et un algorithme irrévocable est applicable. Par contre, mieux vaut une estimation NULLE (zéro) qu'une mauvaise estimation, zéro est la limite d'une (sous) estimation de la distance restante.

Traitement des chemins partiels redondants

Soit un chemin partiel partant de Départ et aboutissant au nœud N (noté Départ –> N). Supposons que ce chemin traverse plusieurs nœuds : Départ –> X1 –> X2 –> .. –> N.

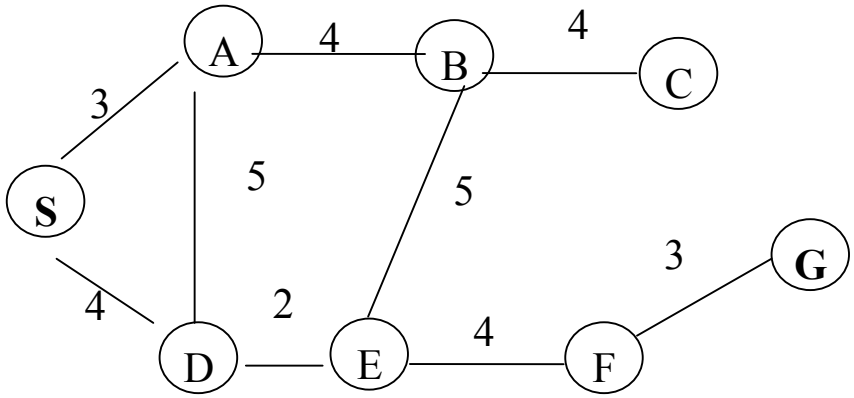
Deux chemins partiels sont redondants si, en partant de Départ, ils aboutissent au même nœud N en empruntant deux trajets différents : par

exemple :

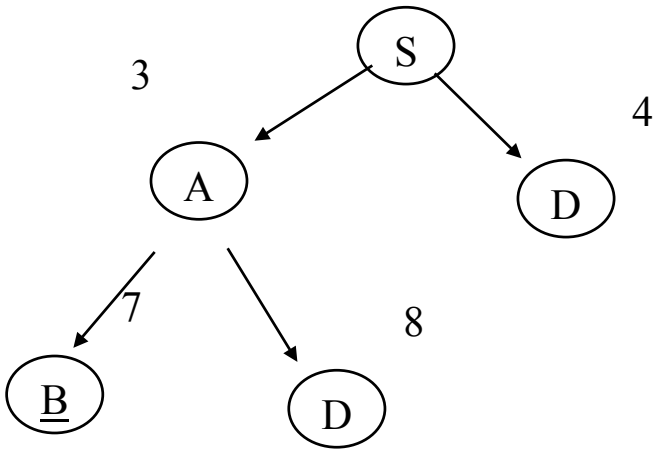
Départ –> X1 –> X2 –> .. –> N et Départ –> Y1 –> Y2 –> .. –> N.

Lors des extensions des chemins partiels, si deux chemins partiels redondants sont trouvés, il semble évident que le trajet optimal final ne peut contenir que le plus court de ces deux chemins partiels.

Exemple : pour le graphe



On aura :



Il est évident que le chemin partiel SAD8 ne sera pas développé car SD4 est plus court.

Importance des nœuds intermédiaires

A partir de cet exemple, on peut rappeler la stratégie appelée le "**Principe de la Programmation Dynamique"** qui stipule que pour trouver un chemin optimal entre Départ et But, on peut ignorer tous les chemins depuis Départ à tout nœud intermédiaire I sauf le plus court de Départ à I.

Le principe de la Programmation Dynamique s'énonce de la manière suivante :

"Le meilleur chemin en passant un nœud intermédiaire particulier I est le meilleur chemin jusqu'à I depuis Départ suivi du meilleur chemin de I à But".

L'algorithme suivant est une version modifiée de B&B qui tient compte du principe ci-dessus.

L'algorithme B& B modifié

Si l'on applique ce principe à l'algorithme B& B, on aura :

- Pile des chemins = chemin de longueur 0 contenant Départ

- Tant que Pile ≠ Vide et le nœud terminal N du sommet(Pile) ≠ But

- Enlever le premier chemin partiel CH de la Pile (dépiler)

- Etendre CH en ajoutant les successeurs de N

- Eliminer de ces nouveaux chemins ceux contenant une boucle

- Ajouter les chemins restant à la Pile

- Si deux (ou plus) chemins aboutissent au même nœud, les supprimer tous sauf celui qui atteint ce noeud commun avec un coût minimum

- Trier toute la Pile sur les coûts des chemins (la longueur du chemin partiel) avec le meilleur en tête

- Fin Tant que

- Si le But est atteint alors succès Sinon échec.

Remarques :

La partie modifié est en italique.

Dans l'algorithme ci-dessus, le coût d'un chemin est la composante g(n) de f(n). La composante h(n) sera requise dans l'algorithme A*.

Exemple

Pour le graphe de la page précédent, un des meilleurs chemin (pas forcément le meilleur puisque la composante h(n) est absente) pour aller de S à G est :

• [[S]]	
• [[SA3], [SD4]]	Développement de S
• [[SAB7], [SA<u>D</u>8], [S<u>D</u>4]]	Développement de SA3
• [[SD4], [SAB7]]	Chemins redondants (D); suppression de SAD8; Tri
• [[SD<u>A</u>9], [SDE6], [S<u>A</u>B7]]	Développement de SD4 Nœud A redondant; SDA9 ne
• [[SDE6], [SAB7]	sera pas développé; Tri
• [[SDE<u>B</u>11], [SDEF10], [SA<u>B</u>7]]	Développement
• [[SAB7], [SDEF10]]	B Redondant; suppr et tri
• [[SABC11], [SABE12], [SDEF10]]	Dévt. "C" est une impasse.
• [[SDEF10]	"E" est redondant. Tri.
• [[SDEF**G**13]]	Succès.

Remarques : on note bien que si un nœud <u>terminal</u> N (noté N_1) est redondant à cause de la présence du même nœud (N_2), le chemin partiel aboutissant à N_1 est considéré redondant même si N_2 n'est pas terminal. Si le coût du chemin partiel aboutissant à N_1 est supérieur à celui de N_2 alors le nœud N_1 sera abandonné (cf. les cas redondants ci-dessus). Dans le cas contraire, le nœud (éventuellement non terminal N_2) sera abandonné même si N_2 est la racine d'un sous arbre.

Algorithme A*

L'algorithme A* est une procédure B&B avec une estimation de la distance au But (composante h de la fonction heuristique f) et tenant compte du principe de la Programmation Dynamique.

Rappelons que si l'estimation du coût au But est une borne inférieure de la distance réelle, alors cet algorithme produit un chemin optimal.

On remarquera la similitude entre l'algorithme ci-dessous et celui de B&B avec le principe de la Programmation Dynamique, la différence est dans le calcul du coût du chemin partiel.

- Pile des chemins = chemin de longueur 0 contenant Départ

- Tant que Pile ≠ Vide et le nœud terminal N du sommet(Pile) ≠ But

- Enlever le premier chemin partiel CH de la Pile (dépiler)

- Etendre CH en ajoutant les successeurs de N

- Eliminer de ces nouveaux chemins ceux contenant une boucle

- Ajouter les chemins restant à la Pile

- Si deux (ou plus) chemins aboutissent au même nœud, les supprimer tous sauf celui qui atteint ce noeud commun avec un coût minimum

- Trier toute la Pile sur la somme de la longueur du chemin partiel et de l'estimation de la borne inférieure de la distance restante, le meilleur en tête

- Fin Tant que

- Si le But est atteint alors succès Sinon échec.

Codage en Prolog

Le prédicat suivant implante la stratégie A* en Prolog.

Le code du programme en Prolog du schéma A* ainsi que le traitement détaillé de l'exemple Taquin sont donnés plus loin.

Dans le prédicat *a_star* ci-dessous, chaque nœud de l'arbre de recherche est représenté par le triplet *<Nœud, Valeur_de_f, Niveau_du_nœud>*.

On suppose ici que la valeur de g dans *f(n)=g(n)+h(n)* est la longueur du chemin partiel (le niveau du nœud dans l'arbre de recherche).

Les prédicats *calcul_f* et classer dépendent de l'exemple traités.

```
a_star([[X/F/N|Xs] | _], [X/F/N|Xs]):-      % condition d'arrêt
    final(X).                               % Etat final (But) atteint ?
a_star([[X/F/Niv|Xs]|Xs1],Plan) :-
    findall(N,
          (transition(X,N), not dans_l_ancetre(N, [X/F/Niv|Xs])),
                Succ_de_X),
          Niv1 is Niv+1,
        % calcul de la valeur de f=g+h pour les successeurs du noeud  X
        calcul_f(Succ_de_X, Niv1, Succ_de_X_avec_f),
        findall([K,X/F/Niv|Xs],
             K ^ membre(K, Succ_de_X_avec_f),
             New_path),
    concaten(Xs1, New_path, New_agenda),    % ajout des succs
    classer(New_agenda,  Agenda_classes),
    a_star(Agenda_classes, Plan).
```

L'importance de l'heuristique

Une heuristique (f=g+h) avec h=0 est admissible mais l'algorithme A* devient alors un simple parcours en largeur et donc peu efficace. Si h est la plus grande borne inférieure de h*, on développe moins de nœuds dans l'arbre de recherche tout en restant admissible.

Parfois, on peut sacrifier l'admissibilité sans que h soit une borne inférieure de h* pour gagner en efficacité de l'heuristique. Ce type d'heuristique permet de résoudre des problèmes plus difficiles.

Pour prendre un exemple non trivial, on considère le jeu Taquin (cet exemple sera développé en détails plus loin).

Dans Taquin, pour f(n)=g(n)+h(n), une heuristique couramment employée est h(n)=W(n) avec W(n) le nombre de cases mal placées. h(n) est une borne inférieure de h*(n) mais ne fournit pas une bonne estimation de la difficulté en nombre d'étapes pour résoudre le puzzle. Une meilleure estimation peut être P(n) : la somme des distances de chaque case mal placée jusqu'à l'état final (on ignore d'intervertir des pions). Cette heuristique n'aggrave pas la difficulté d'échanger les positions de deux cases adjacents.

Un estimation qui marche bien pour Taquin est $h(n)=P(n)+3S(n)$ où S(n) est le score de la séquence obtenu en vérifiant les cases non centrales et en donnant la valeur 2 à un pion qui n'est pas suivi par son propre successeur et 0 pour les autres; le pion central ayant la valeur 1.

Cette heuristique n'est pas une borne inférieure de h*. Pourtant, en tenant compte de cette valeur de h(n) dans f(n)=g(n)+h(n), on résout des puzzles plus difficiles (avec des solutions pas optimales).

Rappel des propriétés de l'algorithme A*

• Propriétés de la fonction heuristique f(n) = g(n) + h(n) :

- Soit $k(N_i, N_j)$ le coût entre deux nœuds N_i et N_j reliables (s'il n'y a pas de chemin entre N_i et N_j alors k n'est pas défini).

- $k(N, G_i)$ est la valeur du chemin depuis le nœud N à un but G_i

- $h^*(N)$ = minimum de tout $k(N, G_i)$ sur l'ensemble de tous les buts $G=\{G_i\}$,

$h^*(N)$ est le coût du chemin de coût minimal de N à un but.

- Tout chemin depuis N à un but qui possède $h^*(N)$ est un chemin **optimal** de N à ce but.

h^* n'est pas défini pour un N qui n'aboutit pas à un but.

- Pour un chemin optimal du départ S à un nœud N, on a

$g^*(N) = k(S, N)$ pour tout nœud N accessible depuis S.

- Soit g(N) le coût du chemin de S à N donné. Si g(N) est la somme des coûts des arcs allant de S a N (ce que l'on utilise fréquemment), alors $g(N) \geq g^*(N)$ où g = une estimation, g^* = la coût réelle de S à N. La raison en est que g(N) est le plus petit coût d'un chemin de S à N trouvé par l'algorithme à l'étape présente. Cette valeur pourrait éventuellement diminuer selon le traitement des nœuds redondants.

- $f^*(N) = g^*(N) + h^*(N)$: chemin optimal de S à N + le chemin optimal de N au But.

$f^*(N)$ est le coût d'un chemin optimal depuis S contraint à passer par

N.

f*(S) = h*(S) représente la même valeur sans la contrainte de passer par N.

- Dans f(N) = g(N) + h(N), la fonction heuristique couramment employée dans A*, les valeurs f,g et h sont des estimations de f*, de g* et de h*.

- Pour h une estimation de h*, on utilise une information heuristique telle que celle utilisée dans Taquin.

- Si l'algorithme B&B utilise f(N), alors on a un **A_algorithme**.

- Si h(N) ≤ h*(N) pour tout N, alors l'A_algorithme trouve un chemin optimal.

- Lorsque l'A_algorithme utilise un h qui est une borne inférieure de h*, l'algorithme est dit **A***.

- On peut prendre un parcours en largeur avec h=0, g=niveau de profondeur du nœud.

Démontrer que le parcours en largeur trouve un chemin minimal est déduit de cette propriété de A* car h est le minimum de tout h* (h=0).

• Quelques propriétés de l'algorithme A* :

- A* termine toujours pour les graphes finis.

- S'il existe un chemin entre Départ et But, A* termine.

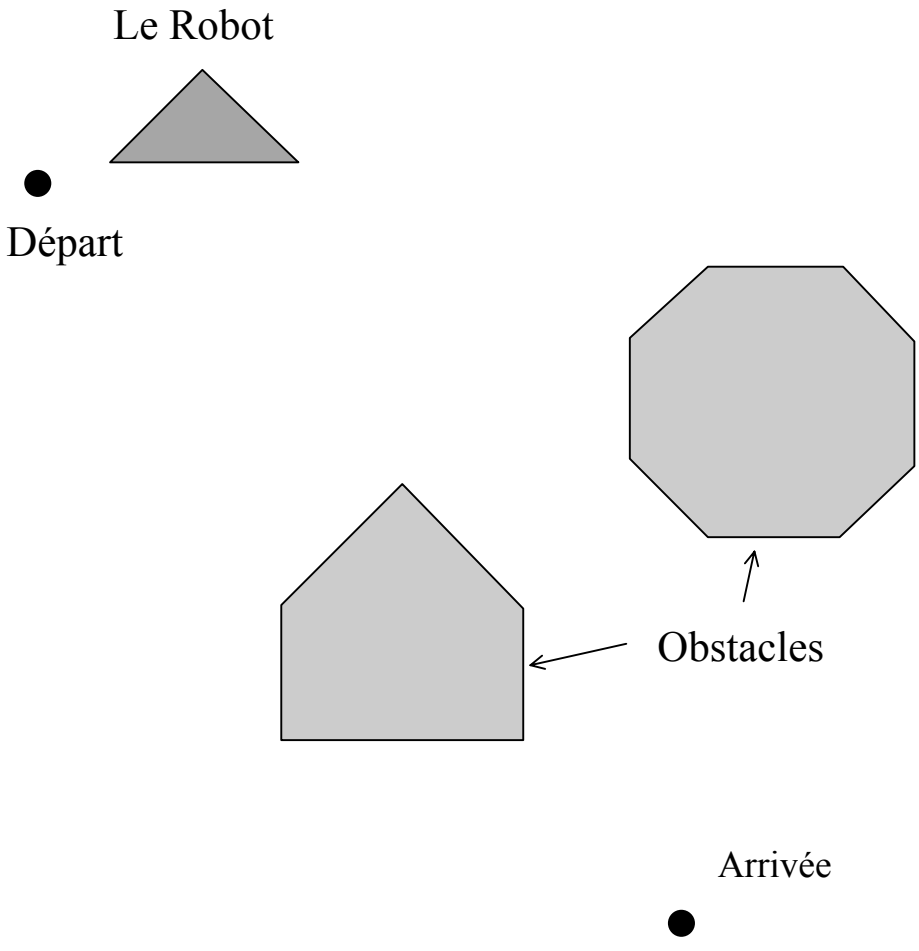
- A* est admissible : s'il y a un chemin de Départ à But, A* termine et trouve un chemin optimal.

Un exemple : le robot

Dans l'exemple suivant, le problème de recherche de chemin d'un Robot est ramené à un problème de recherche dans un graphe d'états que l'on résout par une stratégie informé telle que A*.

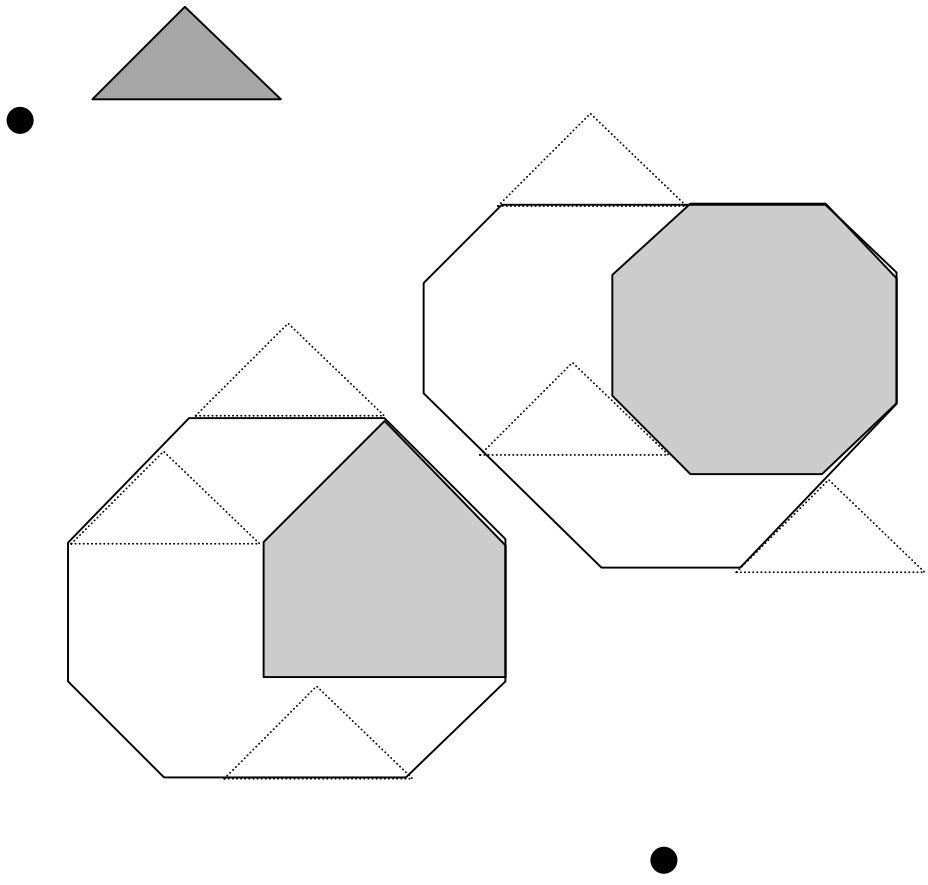
Soit un Robot triangulaire dans un espace contenant des obstacles. Pour simplifier, on traite le problème dans un espace à 2 dimensions.

Le but du Robot est d'atteindre une destination donnée en partant d'un point de départ par **le plus court chemin**. Pour simplifier, on suppose que le Robot (le triangle) ne fera aucune rotation.



Avant de trouver le chemin le plus court, on s'assure que le Robot peut passer par tous les endroits, y compris entre les deux obstacles.

En faisant glisser le triangle représenté par le Robot, on obtient la figure suivant e (appelée l'espace de configuration). Les deux objets ajoutés à la figure précédente (qui forment deux grossissements des obstacles) représentent l'espace nécessaire au Robot pour faire mouvement en "rasant" l'un de ces obstacles. On constate que le Robot a suffisamment d'espace de passage entre ces deux objets. Quelques exemples de mouvements du Robot (sous la forme de triangles hachurés) sont représentés dans la figure.

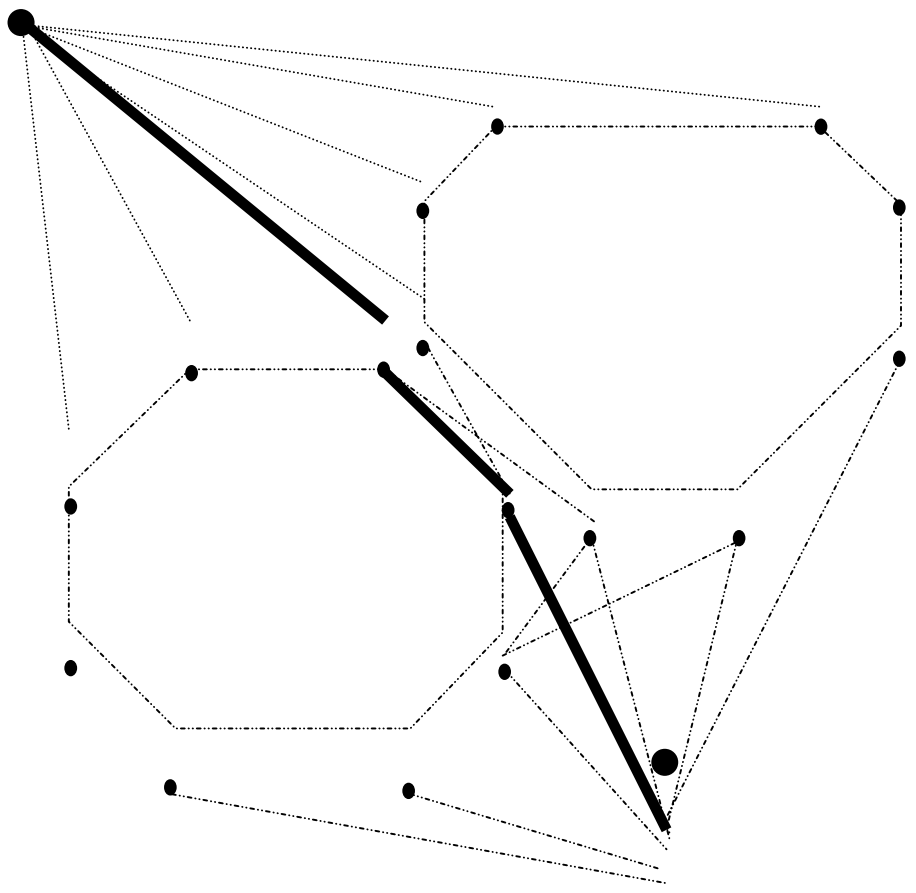


Si le Robot pouvait "voir" le point d'arrivée, le plus court chemin serait la ligne directe entre le départ et l'arrivée. Puisque le Robot ne peut

pas voir directement le point d'arrivée, le seul mouvement plausible (raisonnable) est de se déplacer vers un sommet visible puis essayer d'atteindre l'arrivée de la même manière.

On constate qu'un chemin acceptable sera trouvé par un déplacement sommet à sommet (sauf le tout premier mouvement et le tout dernier). Ainsi, le Départ, l'Arrivée et les sommets de l'espace de configuration représenteront les nœuds d'un graphe (appelé le graphe de visibilité).

La figure suivante représente ce graphe. Pour éviter un dessin trop touffu, on n'a pas dessinés tous les arcs (de sommet à sommet) possibles.



Dans la figure ci-dessus, les 3 segments en traits pleins représentent le chemin le plus court obtenu par A*.

Parmi les heuristique possibles, on pourrait envisager le fait que la somme des deux cotés d'un triangle est supérieure au 3ème coté.

Un exemple complet : le Taquin

- Un échiquier (3 x 3).
- Une des cases de l'échiquier est vide;

  les autres cases contiennent des entiers de un à huit.

2	1	6
4	8	
7	5	3

figure - 1 : état initial

- On peut déplacer horizontalement ou verticalement un entier occupant une case non vide vers la case vide.
- Dans l'exemple, on peut déplacer 6,8 et 3.
- **Le problème** du taquin est de trouver une suite de déplacements permettant d'obtenir, à partir d'un état initial, la configuration finale :

1	2	3
8		4
7	6	5

figure - 2 : état final

Formalisation par un graphe d'états

- Un état = la description du contenu des cases de l'échiquier.
- Il existe un transition entre deux états s'il existe une action permettant de passer d'un état à un autre.
- Les actions possibles seront les déplacements des cases vers la case vide.

• Par exemple, pour l'état initial ci-dessus, on peut envisager trois successeurs.

• Le premier est

2	1	6
4		8
7	5	3

figure - 3

• Le deuxième :

2	1	
4	8	6
7	5	3

figure - 4

• et le troisième :

2	1	6
4	8	3
7	5	

figure - 5

Remarque :

• Il est plus simple de considérer que les actions possibles sont les déplacements de la case vide vers la droite, gauche, haut et bas.

=> On nommera ces actions par **droite**, **gauche**, **haut** et **bas.**

• Ces actions ne seront pas toutes applicables à un état donné. Dans la figure-1 ci-dessus, les actions applicables seront droite, bas et haut.

Description des états

- On caractérise un état de l'échiquier par une liste de 9 éléments.

Le premier élément de cette liste représente la position du blanc (la case vide) et les huit autres éléments les positions des entiers 1 à 8.

=> le deuxième élément de la liste donne la position de l'entier 1 sur l'échiquier, le troisième élément de la liste donne la position de l'entier 2, etc.

- La position d'un entier est représentée par le couple de coordonnées (X/Y). Les coordonnées de chaque case sont données par le schéma suivant :

1/3	2/3	3/3
1/2	2/2	3/2
1/1	2/1	3/1

figure - 6

- Par exemple, pour représenter la configuration suivante :

2	8	3
1	6	4
7		5

figure - 7

on aura la liste [**2/1**, 1/2, 1/3, 3/3, 3/2, 3/1, 2/2, 1/1, 2/3].

Description des transitions

• Dans le passage d'un état E1 à un état E2, seules deux cases sont en jeu: la case vide qui se déplace vers une case occupée O et la case O qui prend la place de la case vide.

• Les possibilités de déplacement pour la case vide :

Le prédicat *possible* a deux arguments dont le premier est les coordonnées de la case vide et le deuxième les mouvement applicables à cette position de la case vide :

possible(1/3, [droite,bas]).
possible(1/2, [haut,droite,bas]).
possible(1/1, [haut,droite]).
possible(2/3, [droite,gauche,bas]).
possible(3/3, [gauche,bas]).
possible(2/2, [haut,droite,gauche,bas]).
possible(3/2, [haut,bas,gauche]).
possible(2/1, [gauche,droite,haut]).
possible(3/1, [haut,gauche]).

• Pour définir une transition d'un état E1 vers un état E2, on procède de la manière suivante :

- Pour la tête de E1 (disons BLANC), extraire les mouvements possibles MOVES par le prédicat possible;

- Pour tout mouvement possible M dans MOVES :

+ appliquer le mouvement M à BLANC pour obtenir BLANC1 les nouvelles coordonnées de la case vide;

+ Trouver dans E1 l'élément dont les coordonnées sont identiques à BLANC1.

Cette case-là devra prendre comme coordonnées celles de BLANC.

+ Remplacer (le prédicat *remplacer*) cet élément par BLANC

```
transition(E1,E2) définit la transition E1 --> E2
transition([Blanc|Reste],N_Etat) :-
    possible(Blanc,Moves),
    appliquer(Moves,[Blanc|Reste],N_Etat).

appliquer(MOVES, E1, E2) :
    applique chacun des mouvements à E1 pour  obtenir E2
appliquer([M|_], E1,E2) :- faire_mouvement(M,E1,E2).
appliquer([_|Ms], E1,E2) :- appliquer(Ms, E1,E2).

faire_mouvement(Move,E1, E2) :
    applique un mouvement à E1 pour obtenir E2
faire_mouvement(Move,[Blanc|Reste],[Blanc1|Reste1]) :-
    G =.. [Move,Blanc,Blanc1],  call(G),
    remplacer(Blanc,Blanc1, Reste, Reste1).

remplacer(BLANC, BLANC1, RESTE, RESTE1) :
     remplacer BLANC1 par  BLANC dans RESTE pour obtenir RESTE1.
    Il s'agit de remplacer un élément par un autre dans une liste : on donne
les coordonnées BLANC à l'élément de RESTE dont les coordonnées sont
égales à celles de BLANC1
remplacer(Y,X,[X|Zs],[Y|Zs]) :- !.
remplacer(Y,X,[Z|Zs],[Z|Us]) :- remplacer(Y,X,Zs,Us).
```

application effective d'un mouvement à un couple de coordonnées

(voir call(G) ci-dessus)

```
haut(X/Y, X/Y1) :- Y1 is Y+1 .
bas(X/Y, X/Y1) :- Y1 is Y - 1.
droite(X/Y,X1/Y) :- X1 is X+1 .
gauche(X/Y,X1/Y) :- X1 is X-1 .
```

• Il reste à définir les prédicats utilitaires d'usage. Le prédicat **recherche** implante une des stratégies de parcours de graphe (informée ou aveugle).

```
jouer(Plan) :-
    initial(X), final(Y),
    recherche(X,Y,Plan).
initial([2/1, 1/2, 1/3, 3/3, 3/2, 3/1, 2/2, 1/1, 2/3]).
```

L'état initial (le fait initial ci-dessus) correspond à :

2	8	3
1	6	4
7		5

final([2/2, 1/3, 2/3, 3/3, 3/2, 3/1, 2/1, 1/1 , 1/2]).

L'état final correspond à :

1	2	3
8		4
7	6	5

Pour une stratégie telle qu'un parcours simple **en largeur**, on obtient une première réponse à la question **?-jouer(Plan)**.

Plan = [[2/2, 1/3, 2/3, 3/3, 3/2, 3/1, 2/1, 1/1, 1/2],
[1/2, 1/3, 2/3, 3/3, 3/2, 3/1, 2/1, 1/1, 2/2],
[1/3, 1/2, 2/3, 3/3, 3/2, 3/1, 2/1, 1/1, 2/2],
[2/3, 1/2, 1/3, 3/3, 3/2, 3/1, 2/1, 1/1, 2/2],
[2/2, 1/2, 1/3, 3/3, 3/2, 3/1, 2/1, 1/1, 2/3],
[2/1, 1/2, 1/3, 3/3, 3/2, 3/1, 2/2, 1/1, 2/3]]

Ce qui est équivalent à la séquence de mouvements suivante à faire :
[haut, haut, gauche, bas, droite].

Heuristiques dans Taquin

- Le défaut des parcours en profondeur / en largeur d'abord :

parcours **aveugles**, trop généraux, n'utilisent pas de connaissances propres au problème à résoudre pour choisir la transition à appliquer à un état.

- on peut choisir une action (un transition) menant à un sommet qui représente plus d'intérêt : cas de prédicat **valeur(Etat, Valeur)**.
- Par exemple, dans une stratégie simple pour le cas du Taquin le prédicat *valeur* compte le nombre de cases bien placées.
- D'autres heuristiques telles que la distance à vol d'oiseau d'une case par rapport à sa bonne place sont possibles.
- Cas simple d'heuristique (et choix pour les mêmes valeurs ci-dessus) : préférer déplacer la case vide vers la gauche plutôt qu'ailleurs quand cela est possible.
- Codage en ordonnant la liste qui figure en deuxième paramètre du prédicat *possible*.
- La mise en place d'une heuristique est plus simple dans un parcours en largeur d'abord car on organise la gestion des successeurs d'un nœud, ce qui permet d'échapper aux retours en arrières du moteur de Prolog.

- Pour un schéma simple (par exemple HC), on peut classer la liste des transitions applicables (insérées en queue de la liste dans chemin_l).

Exemple de prédicat valeur :

La valeur d'un état sera définie en fonction du nombre de cases bien placées :

```
valeur(Etat, Valeur) :-
    final(F),                          % l'état final , l'état de référence
    bien_placees(Etat, F, Valeur).     % nombre de bien_placés
bien_placees([],[ ], 0).
bien_placees([X|Xs],[X|Ys], N) :-      % on a trouvé un bien_placé
    ! , bien_placees(Xs, Ys, N1), N is N1 + 1  .
bien_placees([_|Xs], [_|Ys], N) :-
    bien_placees(Xs, Ys, N).
```

- Le classement des états candidats à la succession d'un état sera fait par le prédicat *classer* suivant.
- On obtiendra une liste de termes *paire(Etat, Valeur_de_l_Etat)* :

```
classer(Etats, Etats_classes) :-
    findall(paire(Etat,Valeur), (membre(Etat, Etats),
            valeur(Etat, Valeur)), Liste),
    trier(Liste, Paires_classes),
    findall (Etat, Valeur^ membre(paire(Etat,Valeur),
            Paires_classes), Etats_classes).
```

Remarques :

- Le dernier appel à findall permet d'extraire les Etats de la liste des *paire(Etat, Valeur).*
- L'expression *Valeur^membre...* (il existe Valeur tel que membre...) permet d'obtenir toutes les réponses (la variable Valeur est libre).
- Le prédicat *trier* appliquera une méthode classique de tri (par exemple le tri rapide) pour ordonner les états selon leur valeur.

Pour insérer cette stratégie dans un parcours en largeur et obtenir un schéma équivalent à HC, on modifie le prédicat *chemin_l* pour obtenir :

```
chemin_l([[X|Xs]|_], X, [X|Xs]).
chemin_l([[X|Xs]|Xs1],Y,Plan) :-
    findall([N,X|Xs], (transition(X,N), not membre(N,[X|Xs])),
    Succ_de_X),
    classer(Succ_de_X, Succ_de_X_classes),
    concaten(Xs1, Succ_de_X_classes, Z),
    chemin_l(Z, Y, Plan).
```

- Le calcul des valeurs et le classement des états revient à considérer un graphe valué d'états.
- Dans le cas du graphe simple précédent, le nœud Y sera le successeur choisi du nœud X plutôt que le nœud Z si la valeur portée par la transition (X,Y) est supérieure à celle de la transition (X,Z)

Résolution par le schéma A* : programme Prolog

- L'heuristique mise en place par le prédicat ci-dessus est une heuristique de type HC.

  => on choisira le meilleur nœud parmi ceux d'un niveau donné.
- On peut utiliser une heuristique du type Best-First ou A* qui permettent de choisir le nœud le plus prometteur de tout le graphe.

  => il faudra classer l'ensemble des nœuds *Xs1* et *Succ_de_X*.

  => Dans ce cas, le parcours ne sera plus nécessairement en largeur et par niveau puisque le nœud le plus prometteur peut être choisi parmi ceux des niveaux précédents.

=> Il faut vérifier que la transition est toujours possible.

```
% recherche(Etat initial,  Plan à construire) :    prédicat de lancement
%   chaque nœud de l'arbre de recherche est représenté par un triplet
%   <Nœud, Valeur de la fonction f, Niveau)      avec f = h + g de A*

recherche(X, Plan):-
      calcul_f([X], 0, X_avec_f_et_niv),       % calcul de f pour l'état initial
      a_star ([X_avec_f_et_niv], Plan).

% Le  schéma A*

a_star([[X/F/N|Xs] | _], [X/F/N|Xs]):-       % condition d'arrêt
      final(X).
a_star([[X/F/Niv|Xs]|Xs1],Plan) :-
      findall(N,
            (action(X,N), not dans_l_ancetre(N, [X/F/Niv|Xs])),
            Succ_de_X),
      Niv1 is Niv+1,
          % calcul de la valeur de f=g+h pour les successeurs du noeud  X
      calcul_f(Succ_de_X, Niv1, Succ_de_X_avec_f),
      findall([K,X/F/Niv|Xs],
            K ^ membre(K, Succ_de_X_avec_f),
            New_path),
      concaten(Xs1, New_path, New_agenda),     % ajout des succs
      classer(New_agenda,  Agenda_classes),
      a_star(Agenda_classes, Plan).

% Pour éviter les boucles, vérifier que le nœud N ne soit pas sur le chemin
% depuis le départ (Ancêtres). Le chemin partiel depuis le départ (Ancêtres)
% est une liste de triplets  <Nœud/F/Niv> et N est un nœud simple
dans_l_ancetre(N,[N/_F/_Niv|_]).
dans_l_ancetre(N,[_X|Xs]) :-
      dans_l_ancetre(N,Xs).
% Classement global. Les éléments de la liste sont des chemins partiels.
```

```
% trier_avec_path est une adaptation de Quick Sort adapté à notre cas
classer(New_agenda,  Agenda_classes) :-
      trier_avec_path(New_agenda,  Agenda_classes).


% calcul_f(Nœuds, Niv, Nœuds_avec_f)
% Nœuds est une liste de nœuds.
% calculer la valeur de f=g+h pour chaque nœud
calcul_f(Noeuds, Niv, Noeuds_avec_f) :-
       findall(Noeud / Valeur / Niv,
                 (membre(Noeud, Noeuds),
                  valeur_taquin(Noeud, Niv, Valeur)),  % f(n)=g(n)+h(n)
                Noeuds_avec_f).


% valeur f(n)=g(n)+h(n)  pour l'exemple Taquin
valeur_taquin(Etat, Niv, Valeur) :-
      final(F),                          % l'état final
      misplaced(Etat, F, Val_h),         % nombre de bien_placés
      Valeur is Val_h + Niv.             % Niv = g(n), f=g+h


% la case blanche ne compte pas dans g(n) = 8-mal placés
misplaced([_Blac|L],[_Blanc1|F], Val_h) :-
      bien_placees(L, F, Val_h_inv),      % nombre de bien_placés: 0.. 8
      Val_h is 8 - Val_h_inv.
bien_placees([],[], 0).
bien_placees([X|Xs],[X|Ys], N) :-       % on a trouvé un bien_placé
      ! , bien_placees(Xs, Ys, N1), N is N1 + 1  .
bien_placees([_|Xs], [_|Ys], N) :-  bien_placees(Xs, Ys, N).
```

```
% Les états
initial([2/1, 1/2, 1/3, 3/3, 3/2, 3/1, 2/2, 1/1, 2/3]).
final([2/2, 1/3, 2/3, 3/3, 3/2, 3/1, 2/1, 1/1 , 1/2]).

% Prédicat de lancement
go_a_star(Plan) :-
    initial(X),  a_star (X,Plan).
```

Remarques :

% Le prédicat possible est inchangé. L'ordre des mouvements : g,h,d,b

% Les prédicats action, appliquer, faire_move et remplacer inchangés

% Les prédicats de déplacement h, g, d et b inchangés.

Le code ci-dessus ne traite pas l'élimination des nœuds communs (voir plus haut le principe de la programmation dynamique).

Exemple d'utilisation:

?- go_a_star (P).

Sur SWI-Prolog sur un PC avec des options simples, on obtient quatre réponses puis on se heurte au problème d'insuffisance de mémoire qui empêche de poursuivre (stack overflow) : l'exemple est gros !

P = [[2/2, 1/3, 2/3, 3/3, 3/2, 3/1, 2/1, 1/1, 1/2]/5/5,
[1/2, 1/3, 2/3, 3/3, 3/2, 3/1, 2/1, 1/1, 2/2]/5/4,
[1/3, 1/2, 2/3, 3/3, 3/2, 3/1, 2/1, 1/1, 2/2]/5/3,
[2/3, 1/2, 1/3, 3/3, 3/2, 3/1, 2/1, 1/1, 2/2]/5/2,
[2/2, 1/2, 1/3, 3/3, 3/2, 3/1, 2/1, 1/1, 2/3]/4/1,
[2/1, 1/2, 1/3, 3/3, 3/2, 3/1, 2/2, 1/1, 2/3]/4/0] ;

P = [[2/2, 1/3, 2/3, 3/3, 3/2, 3/1, 2/1, 1/1, 1/2]/9/9,
[2/3, 1/3, 2/2, 3/3, 3/2, 3/1, 2/1, 1/1, 1/2]/9/8,
[1/3, 2/3, 2/2, 3/3, 3/2, 3/1, 2/1, 1/1, 1/2]/9/7,
[1/2, 2/3, 2/2, 3/3, 3/2, 3/1, 2/1, 1/1, 1/3]/9/6,
[2/2, 2/3, 1/2, 3/3, 3/2, 3/1, 2/1, 1/1, 1/3]/8/5,
[2/3, 2/2, 1/2, 3/3, 3/2, 3/1, 2/1, 1/1, 1/3]/7/4,
[1/3, 2/2, 1/2, 3/3, 3/2, 3/1, 2/1, 1/1, 2/3]/6/3,
[1/2, 2/2, 1/3, 3/3, 3/2, 3/1, 2/1, 1/1, 2/3]/5/2,
[2/2, 1/2, 1/3, 3/3, 3/2, 3/1, 2/1, 1/1, 2/3]/4/1,
[2/1, 1/2, 1/3, 3/3, 3/2, 3/1, 2/2, 1/1, 2/3]/4/0] ;

P = [[2/2, 1/3, 2/3, 3/3, 3/2, 3/1, 2/1, 1/1, 1/2]/11/11,

[2/1, 1/3, 2/3, 3/3, 3/2, 3/1, 2/2, 1/1, 1/2]/11/10,

[3/1, 1/3, 2/3, 3/3, 3/2, 2/1, 2/2, 1/1, 1/2]/11/9,

[3/2, 1/3, 2/3, 3/3, 3/1, 2/1, 2/2, 1/1, 1/2]/11/8,

[2/2, 1/3, 2/3, 3/3, 3/1, 2/1, 3/2, 1/1, 1/2]/10/7,

[1/2, 1/3, 2/3, 3/3, 3/1, 2/1, 3/2, 1/1, 2/2]/10/6,

[1/3, 1/2, 2/3, 3/3, 3/1, 2/1, 3/2, 1/1, 2/2]/10/5,

[2/3, 1/2, 1/3, 3/3, 3/1, 2/1, 3/2, 1/1, 2/2]/10/4,

[2/2, 1/2, 1/3, 3/3, 3/1, 2/1, 3/2, 1/1, 2/3]/9/3,

[3/2, 1/2, 1/3, 3/3, 3/1, 2/1, 2/2, 1/1, 2/3]/8/2,

[3/1, 1/2, 1/3, 3/3, 3/2, 2/1, 2/2, 1/1, 2/3]/6/1,

[2/1, 1/2, 1/3, 3/3, 3/2, 3/1, 2/2, 1/1, 2/3]/4/0] ;

P = [[2/2, 1/3, 2/3, 3/3, 3/2, 3/1, 2/1, 1/1, 1/2]/13/13,

[3/2, 1/3, 2/3, 3/3, 2/2, 3/1, 2/1, 1/1, 1/2]/13/12,

[3/1, 1/3, 2/3, 3/3, 2/2, 3/2, 2/1, 1/1, 1/2]/13/11,

[2/1, 1/3, 2/3, 3/3, 2/2, 3/2, 3/1, 1/1, 1/2]/13/10,

[2/2, 1/3, 2/3, 3/3, 2/1, 3/2, 3/1, 1/1, 1/2]/12/9,

[1/2, 1/3, 2/3, 3/3, 2/1, 3/2, 3/1, 1/1, 2/2]/12/8,

[1/3, 1/2, 2/3, 3/3, 2/1, 3/2, 3/1, 1/1, 2/2]/12/7,

[2/3, 1/2, 1/3, 3/3, 2/1, 3/2, 3/1, 1/1, 2/2]/12/6,

[2/2, 1/2, 1/3, 3/3, 2/1, 3/2, 3/1, 1/1, 2/3]/11/5,

[2/1, 1/2, 1/3, 3/3, 2/2, 3/2, 3/1, 1/1, 2/3]/10/4,

[3/1, 1/2, 1/3, 3/3, 2/2, 3/2, 2/1, 1/1, 2/3]/8/3,

[3/2, 1/2, 1/3, 3/3, 2/2, 3/1, 2/1, 1/1, 2/3]/6/2,

[2/2, 1/2, 1/3, 3/3, 3/2, 3/1, 2/1, 1/1, 2/3]/4/1,

[2/1, 1/2, 1/3, 3/3, 3/2, 3/1, 2/2, 1/1, 2/3]/4/0] ;

Systèmes de règles de production

1- Introduction

On considère une règle **tête :- corps.**

- Fonctionnement du moteur Prolog

  ?- but. => réponses

⇒ *Moteur d'inférence à chaînage arrière*

Définition : Un moteur d'inférence qui commence avec un but et travaille avec les faits pour prouver ce but.

⇒ Une autre manière d'utiliser les règles :

⇒ Travailler en partant des faits, c'est à dire en partant des connaissances que l'on a sur le domaine.

⇒ On considère les règles sous l'angle :

action_ou_conclusion :- conditions.

⇒ On ne donne pas de but au moteur d'inférence;

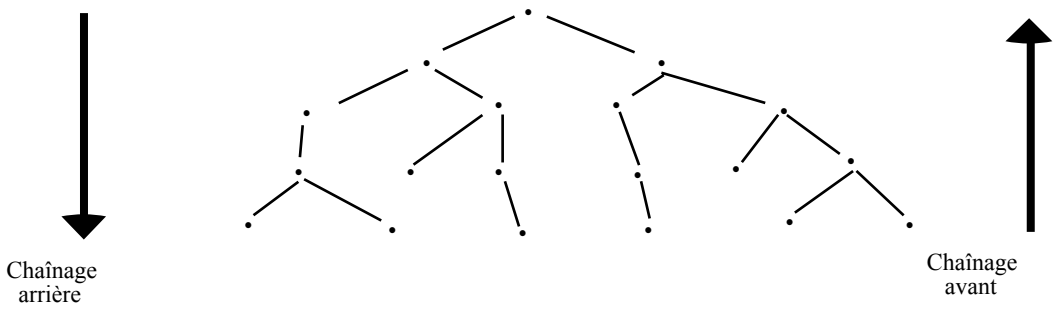
⇒ Le moteur travaille avec les faits déjà établis;

⇒ Si les faits disponibles à un moment donné satisfont les conditions d'application d'une règle, alors le moteur active cette règle et effectue l'action spécifiée par celle-ci.

⇒ *Moteur d'inférence à chaînage en avant*

Définition : Un moteur d'inférence qui fonctionne à partir des faits vers les conséquences.

Sur l'arbre de recherche :



Dans un système fonctionnant en mode de chaînage en avant :

- L'action spécifiée par une (tête) de règle est plus complexe que la tête d'une règle Prolog.

- Cette action peut :
  -ajouter plusieurs faits à la base;
  - supprimer quelques faits de la base;
  - afficher des messages
  - effectuer d'autres actions.

Autrement dit, faire des "effets de bords" sur une zone appelée la *mémoire de travail* ou la *mémoire temporaire*.

- Le système manipule des **règles de production** :

  L'activation d'une règle produit une nouvelle situation.

Différentes sortes de systèmes :

- Ordre 0, Ordre 1, ...
- Orienté données (Data Driven)
- Orienté buts (Goal Driven)
- Chaînage avant / Chaînage arrière (Mixte)

$\Rightarrow$ Cas pragmatique : les algorithmes sont dirigés par un (ou plusieurs) but à atteindre.

Algorithme général de fonctionnement :

(On dispose d'un but à atteindre)

- Trouver une règle activable;
- Activer la règle et appliquer ses conséquences;
- Réitérer.

$\Rightarrow$ On s'arrête si plus aucune règle n'est activable.

L'activation des règles de production peut causer l'ajout de nouvelles informations (**faits**) et de nouveaux **buts** intermédiaires à la mémoire de travail.

Exemple :

Un système de règles de productions pour un robot qui met en marche une lampe.

• Une règle de production aura trois parties :

identificateur, **condition** et **action.**

• Le but sera de la forme *allumer_la_lampe.*

• La situation sera décrite par les faits :

lampe_allumée ou *lampe_éteinte.*

•A l'état initial, la base contient le but et l'un de ces faits.

Raisonnement

Si l'on trouve le but *allumer_la_lampe* dans la base :

• si la lampe est déjà allumée, ne rien faire.

$\Rightarrow$ Ce qui donne la règle :

règle-1 : Si le but *allumer_la_lampe* et
le fait *lampe_allumée*
sont dans la base
Alors
supprimer le but *allumer_la_lampe* de la base.

- si la lampe n'est pas allumée :

règle-2 : Si le but *allumer_la_lampe* et
le fait *lampe_éteinte*
sont dans la base
Alors
ajouter le but *tourner_interrupteur* à la base.

Les autres règles :

règle-3 : Si le but *tourner_interrupteur* et
le fait *lampe_allumée*
sont dans la base
Alors
"tourner l'interrupteur"
supprimer le but *tourner_interrupteur* et
supprimer le fait *lampe_allumée* de la base et
ajouter le fait *lampe_éteinte* à la base.

règle-4 : Si le but *tourner_interrupteur* et
le fait *lampe_éteinte* sont dans la base
Alors
"tourner l'interrupteur"
supprimer le but *tourner_interrupteur* et
supprimer le fait *lampe_éteinte* de la base et
ajouter le fait *lampe_allumée* à la base.

Essai-1 : On se donne l'état initial

- le but *allumer_la_lampe* et
- le fait *lampe_allumée* .

Seule la condition de la règle-1 est satisfaite

$\Rightarrow$ le but *allumer_la_lampe* est supprimé de la base.

Le moteur cherche une autre règle à activer

$\Rightarrow$ Toutes les règles ont un but dans leur partie condition mais il n'y a plus de but dans la base.

$\Rightarrow$ *Le moteur s'arrête faute de règle activable.*

Essai-2 : On se donne l'état initial :

- le but *allumer_la_lampe* et
- le fait *lampe_éteinte.*

Seule la règle-2 s'applique

$\Rightarrow$ le but *tourner_interrupteur* est ajouté à la base.

Au prochain tour, les règles 2 et 4 peuvent s'appliquer puisque leur conditions sont satisfaites.

Laquelle faut-il activer? Nous voudrions que ça soit la règle-4 mais pourquoi pas la règle-2 ?

Ensemble de conflits :

Définition : L'ensemble de règles dont les conditions sont satisfiables à un moment donné.

- Lorsque cet ensemble est vide, le moteur d'inférence s'arrête;
- Quand il contient un seul élément, celui-ci sera activé.
- Lorsque l'ensemble de conflits contient plusieurs règles, il faut décider de la règle à activer.

$\Rightarrow$ Cette décision peut être très importante :

Si nous activons la règle-2 à chaque fois que cela est possible, on boucle en ajoutant le but *tourner_interrupteur* à chaque cycle à la base et le robot ne tournera jamais l'interrupteur.

$\Rightarrow$ Déduisons deux principes simples:

- *Appliquer une règle dès que sa condition est satisfaite.*

- *Toute règle doit modifier la base de telle sorte que sa propre condition ne soit plus satisfaite.*

Ce qui donne (à la place de la règle-2):

règle-2': Si le but *allumer_la_lampe* et
le fait *lampe_éteinte*
sont dans la base et
le but *tourner_interrupteur*
n'est pas dans la base
Alors ajouter le but *tourner_interrupteur* à la base.

NB : on peut garder Regle-2 mias en supprimer le but "allumer_la_lampe".

Reprenons l'algorithme du moteur :

• Après l'activation de la règle-2', seule la condition de la règle-4 sera satisfaite. La base contiendra le but *tourner_interrupteur* et le fait *lampe_allumée* .

• La règle-1 s'appliquera ensuite. Le but *allumer_la_lampe* est supprimé de la base.

• Plus aucune règle n'aura sa condition satisfaite. Le moteur d'inférence (le robot) s'arrête en ayant activé les règles 2', 4 et 1.

⇒ Le problème rencontré avec la règle-2 soulève d'autres principes généraux:

- *Toutes les règles d'un système de production doivent être indépendantes les unes des autres.*

- *La même règle doit être activée dans les même conditions reproduites quelque soit l'ordre d'entrée des règles dans la base;*

⇒ Ce n'est pas le cas en Prolog.

⇒ Il faut garantir l'indépendance des règles de production.

⇒ Question de retour arrière

⇒ Dans le cas simplifié de ce TD, les règles seront décrites de sorte qu'il n'y ait pas de situation où les parties conditions de deux ou plusieurs règles soient satisfaites en même temps.

Si une telle situation se produit, l'ordre d'entrée des règles provoquera les règles différentes à activer.

Programmation d'un moteur d'inférence en chaînage avant simple

2-1 Syntaxe des règles en Prolog

Format général d'une règle de production :

Identificateur :-

Condition , alors, Action.

• **identificateur** :

sous la forme *regle(ID)* où ID est un nombre (ou un atome, une chaîne ou autre).

• La **Condition** et l'**Action** dans le corps de la règle.

Etant donné que Prolog procède de gauche à droite, les actions ne seront exécutées que si les conditions se vérifient.

⇒ *L'action doit être un sous-but qui réussit.*

La condition ne doit pas modifier la base.

2-2- Transcription en Prolog

• Les buts sous la forme *but(...)* et les faits sous la forme *fait(...)*.

Exemples:

• “le but *allumer_la_lampe...*”
sera réécrit en *but(allumer_lampe)*.

• “le fait *lampe_éteinte...*”
devient *fait(lampe_eteinte)*.

• Les “Si” disparaissent;

• Les “et” sont remplacés par des virgules (conjonction) de Prolog.

• La négation devient *not*.

• Pour une meilleur lisibilité des règles, on conserve “alors” qui sera un prédicat qui réussit toujours.

• Les prédicats *ajouter/1* et *supprimer/1* pour ajouter ou supprimer un des faits et des buts.

Les règles de l'exemple précédent :

- Ordre 0 (sans variable),
- Orienté but

- regle(1) :- but(allumer_lampe),
  fait(lampe_allumee), %noter la “,” ici
  alors, supprimer(but(allumer_lampe)). % et là

- regle(2) :- but(eteindre_lampe),
  fait(lampe_eteinte),
  alors, supprimer(but(eteindre_lampe)).

- regle(3) :- but(allumer_lampe), % règle-2’ précédente
  fait(lampe_eteinte) ,
  not but(tourner_interrupteur), %pas encore de marquage
  alors, ajouter but(tourner_interrupteur). % les règles

- regle(4) :- but(eteindre_lampe), % règle nouvelle
  fait(lampe_allumee) , % pour prévoir toute situation
  not but(tourner_interrupteur),
  alors, ajouter(but(tourner_interrupteur)).

- regle(5) :- but(tourner_interrupteur),
  fait(lampe_allumee),
  alors, supprimer(but(tourner_interrupteur)),
  supprimer(fait(lampe_allumee)),
  ajouter(fait(lampe_eteinte)).

- regle(6) :- but(tourner_interrupteur)),
  fait(lampe_eteinte),
  alors, supprimer(but(tourner_interrupteur)) ,
  supprimer(fait(lampe_eteinte)),
  ajouter(fait(lampe_allumee)).

• L'état initial sera présenté par :

but(allumer_lampe) et *fait(lampe_eteinte).*

Les réponses obtenus pour cet état initial sont : => **3, 6, 1**

Et pour :

but(eteindre_lampe)) et *fait(lampe_allumee).* => **4, 5, 2**

2-3 Le programme de chaînage en avant :

```
/* lancement */
go :- chainage_avant.
go :- write('fini'), nl.

chainage_avant :-   regle(ID),
                    write('règle activée: '),
                    write(ID),write('.'), nl, !,
                    chainage_avant.

/* prédicats de gestion de la base */
ajouter(X) :-       assert(X).
supprimer(X) :-     retract(X).

/* réussit toujours et ne fait rien */
alors.
si.                 % éventuellement
/* nettoyage et rechargement */
charger(X) :-
    abolish(fait/1),abolish(but/1), abolisg(regle/1), reconsult(X).
```

4- Amélioration de la syntaxe :

Le format général d'une règle est :

regle ID : si $Cond_1$ et ... et $cond_n$
alors $Action_1$ et .... et $Action_m$.

NB : pour éviter la ré-application d'une même règle dans les mêmes conditions de la base, on peut marquer les règles : noter que telle règle sous telles conditions a déjà été appliquée pour éviter de la ré-appliquer. Ce qui représente un état de boucle.

déclaration des opérateurs en prolog

```
:-op(1200,fx,regle).
:-op(1190,xfx,':').
:-op(1100,fx,si).
:-op(980,xfx,alors).
:-op(954,xfy,et).
```

Lancement :

```
go :- chainage_avant.
go :- write('fini'), nl.
```

nettoyage de la base :

```
charger(X) :-
    abolish(fait/1),abolish(but/1), abolish(deja_fait/2), abolish(regle/1),
    reconsult(X).
```

Le moteur :

```
chainage_avant :-
    regle((ID : si Cond alors Act)),
    executer(Cond) ,            % aucune modification de la base
    not deja_fait(ID,Cond),     % la place entre les deux
    executer(Act),              % est important
    write('regle activee : '),
    write(ID), write('.'), nl, !,
    ajouter(deja_fait(ID,Cond)),
    chainage_avant.
```

Remarque : deja_fait pour la gestion de marquage des règles.

Exécuter une Action, vérifier une condition :

```
executer(A et B) :- !, executer(A),executer(B).
executer(A) :- call(A).
```

Modification de la base :

On prend en compte le cas d'échec et on évite les doublons

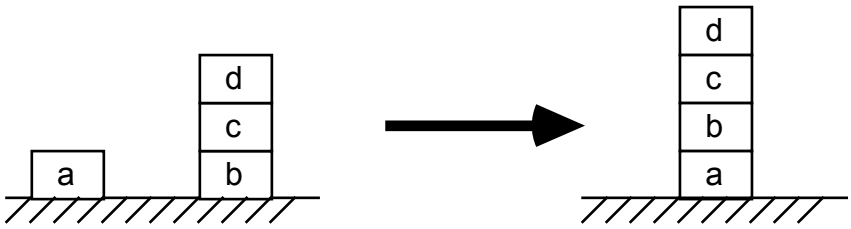
```
ajouter(X) :- not call(X) , assert(X), !.
ajouter(_).

supprimer(X) :- call(X) , retract(X), !.   % on enlève X que
supprimer(_).                              % s'il est là
```

Développement de l'exemple des blocs (Ordre 1, orienté but)

Un robot a pour tâche d'empiler des blocs (des cartons) a,b,c et d. Initialement, le robot sait où sont les blocs et comment ils sont empilés.

Supposons que les blocs a et b sont au sol, b est sous c et c est sous d. Le but du robot est d'empiler a sous b sous c sous d. Il ne peut déplacer qu'un bloc à la fois.

$\Rightarrow$ Le bloc "a" est toujours au sol.



L'état initial est donné par :

fait(est_sous(sol,a)).

fait(est_sous(sol,b)).

fait(est_sous(b,c)).

fait(est_sous(c,d)).

but(empiler([a,b,c,d])).

- Le premier but sera de mettre b sur a. Si le deuxième bloc est déjà sur le

premier, on ne fait rien :

```
regle 1  : si   but(empiler([X,Y|Reste])) et
                fait(est_sous(X,Y))
           alors
                supprimer_but(empiler([X,Y|Reste])) et
                ajouter_but(empiler([Y|Reste])).
```

- Pour placer un bloc sur un autre, il faut que les deux blocs soient libres.

  La règle suivante permet d'effectuer cette action lorsqu'elle est possible.

```
/**** Y est sur un bloc Z, X est libre, mettre Y sur X ****/
regle 2  : si   but(empiler([X,Y|Reste])) et
                not fait(est_sous(X,_)) et
                not fait(est_sous(Y,_)) et
                fait(est_sous(Z,Y))
           alors
                supprimer_fait(est_sous(Z,Y)) et
                ajouter_fait(est_sous(X,Y)) et
                supprimer_but(empiler([X,Y|Reste])) et
                ajouter_but(empiler([Y|Reste])).
```

- S'il y a un autre bloc sur l'un des deux blocs à empiler, le robot doit d'abord enlever cet obstacle (libérer les blocs à empiler X et Y de tout bloc qui les couvre):

```
/******* X est sous Z, libérer X **********/
regle 3  : si   but(empiler([X,Y|Reste])) et
                fait(est_sous(X,Z)) et
                Y \= Z et
                not but(enlever(Z))
           alors
                ajouter_but(enlever(Z)).

/******* X est sous Z, libérer X **********/
regle 4  : si   but(empiler([_,X|Reste]))  et
                fait(est_sous(X,Y)) et
                not but(enlever(Y))
           alors
                ajouter_but(enlever(Y)).
```

- Maintenant, on précise comment enlever un bloc qui nous gène. Remarquons que celui-ci peut être sous un autre bloc :

```
regle 5  : si   but(enlever(X)) et
                fait(est_sous(X,Y)) et
                not but(enlever(Y))
           alors
                ajouter_but(enlever(Y)).

regle 6  : si   but(enlever(X))  et
                not fait(est_sous(X,_)) et
                fait(est_sous(Y,X))
           alors
                supprimer_fait(est_sous(Y,X)) et
                ajouter_fait(est_sous(sol,X)) et
                supprimer_but(enlever(X)).
```

- Finalement, la condition d'arrêt est atteinte lorsqu'il existe un seul bloc dans la liste de blocs à empiler :

```
regle 7  : si   but(empiler([X]))
           alors   supprimer_but(empiler([X])).
```

Exemple d'exécution :

```
/*  état initial  */
fait(est_sous(sol,a)).   fait(est_sous(sol,b)).
fait(est_sous(b,c)).     fait(est_sous(c,d)).
but(empiler([a,b,c,d])).

?- go.
règle activée : 4.
règle activée : 5.
règle activée : 6.
règle activée : 6.
règle activée : 2.
règle activée : 2.
règle activée : 2.
règle activée : 7.        fini.
```

Comment écrire un système de règles

Exemple de voiture : Second choix du travail à rendre

1- Approche orientée données :

- Les faits que l'on manipule (xxx_ok : xxx fonctionne)

klaxon_audible **batterie_ok** **phares_ok** **voiture_ok**
carburant_visible *on sait qu'il y a de l'essence; on en a mis hier !*
demarreur_ok **bruit_demarrage** **jauge_ok**
jauge_positive **jauge_zero** :
la jauge fonctionne et on a de l'essence ou on n'en a pas

- Les phares et le klaxon ne seront jamais en panne.
- Les éléments pouvant causer une panne :

manque de batterie, la jauge, manque d'Essence, le démarreur, la voiture

- Ce que l'on vérifie de visu (ou de manière auditive) :

carburant_visible, jauge_positive, jauge_zero,
klaxon_audible, phares_ok

- Identification les faits et ce qu'ils impliquent (la virgule = et):

klaxon_audible => batterie_ok
phares_ok => batterie_ok
carburant_visible => essence_ok
bruit_demarrage => batterie_ok, demarreur_ok

jauge_positive => batterie_ok, essence_ok, jauge_ok

jauge_zero => batterie_ok, jauge_ok

essence_ok, demarreur_ok => voiture_ok (ça peut rouler)

• Etant données des symptômes, on déduit le fait : voiture_ok ou pas.

2- Approche orientée buts :

• possibilité de diagnostiquer une panne particulière (par exemple de la batterie)

• Identifier les dépendances des buts et les faits qui satisfont ces buts :

vérification de la voiture => vérification de:
la batterie (but)
le démarreur(but)
l'essence(but)

vérification du carburant => vérification de:
la jauge positive (fait) ou
l'essence dans le réservoir(fait)

vérification de la batterie => vérification :
des phares(fait)
bruits au démarrage (fait)
jauge d'essence positive ou zéro (faits)
klaxon audible (fait)

vérification du démarreur => vérification :

bruit de démarreur (fait)

• On peut éventuellement découper le but de vérification de la jauge d'essence :

vérification de la jauge => vérification :
jauge d'essence positive ou zéro (faits)

• L'objectif étant de savoir si la voiture est en panne ou pas (le but).
On planifie en procédant par différentes vérifications (sous-buts).
Les données aideront à satisfaire ces sous-buts.

Les moteurs à chaînage mixte :

• Les moteurs fonctionnant en *chaînage arrière* (comme en Prolog).
• Considèrent les parties conclusions des règles.

• *Chaînage mixte* : on peut au besoin demander explicitement la démonstration d'un fait (but).

règle xx :
si tel fait est disponible
et il s'avère que tel autre fait peut être prouvé (peut être déduit)
alors conclusion.

• On active un fonctionnement en arrière (à la Prolog) par **déduit** :

règle 'ca roule' :

si fait(essence_ok)
et **déduit**(batterie_ok)
alors ajouter(fait(voiture_ok)).

règle 'batterie_ok car klaxon audible' :
si fait(klaxon_ok)
alors ajouter(fait(batterie_ok)).

règle 'batterie_ok car les phares s allument' :
si fait(phares_ok)
alors ajouter(fait(batterie_ok)).

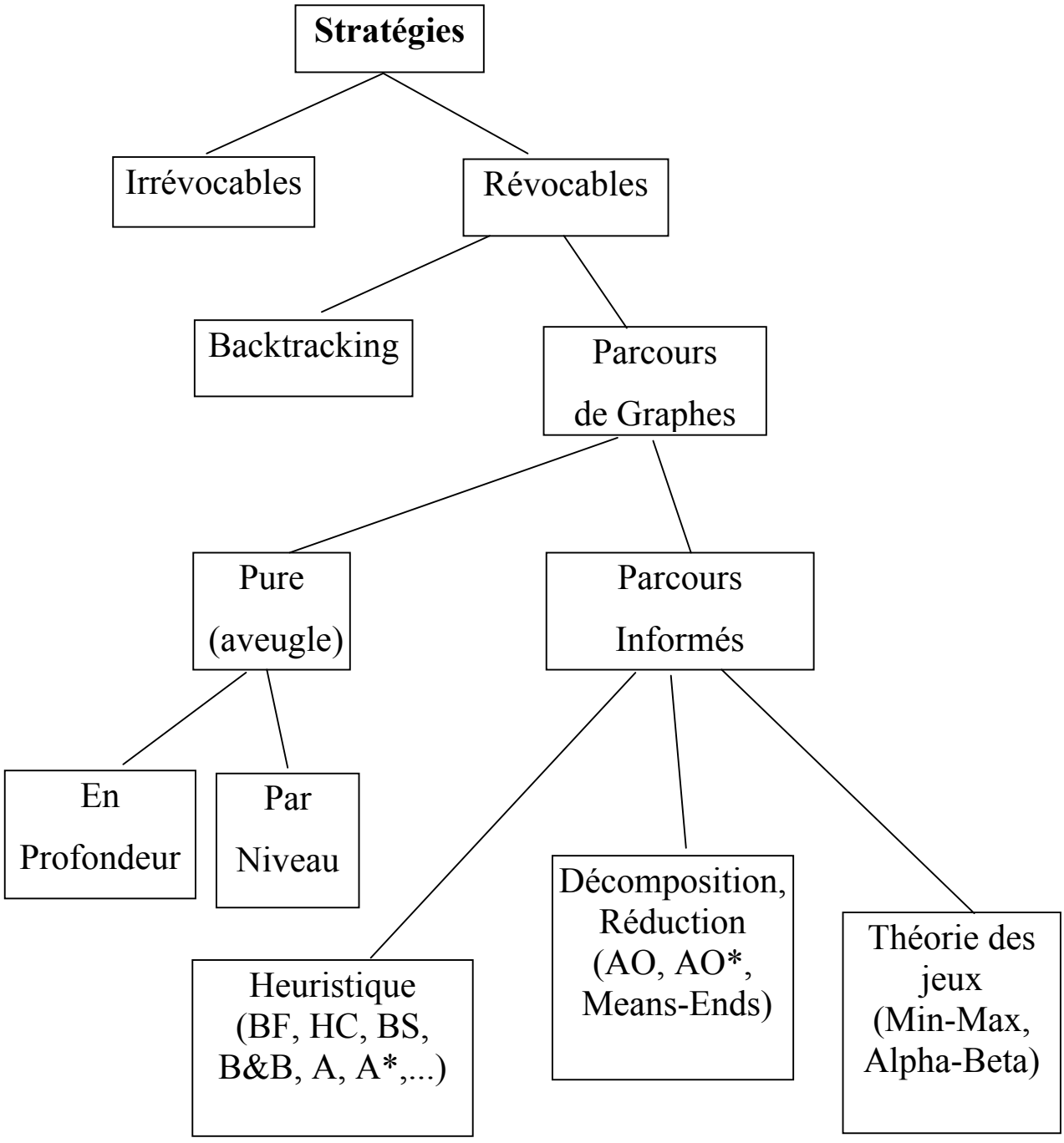
• Ce mode de chaînage produit un fait précis nécessaire à la continuité du fonctionnement (par opposition au mode de chaînage en avant).

• Le système peut être lancé dès le début par un **déduit** pour fonctionner à la Prolog.

Planification et génération de plans

Définition et classification des stratégies

Classement par famille de procédures.



Remarques :

• Dans le graphe ci-dessus, **AO** (et AO*) désigne l'ensemble des méthodes de déductions basées sur les graphes ET-OU (And-Or).

• La méthode **M**eans-**E**nds est développée dans la suite.

• **BF** : Best-First, **HC** : Hill Climbing, **BS** :

Rappel des remarques sur le graphe des stratégies :

• Dans chaque nœud de ce graphe, il y a toute la base de connaissance.

=> Une situation souhaitée serait d'avoir des règles qui décrivent les composants affectés par les changements d'états.

• Méthodes applicables à des problèmes <u>révocables</u> (domaines dans lesquels on peut défaire) :

- Diagnostic médical
- Robotique mobile, ...

D'où l'idée de faire de la planification (appliquée aux méthodes révocables).

Décomposition et Planification

Procédé qui construit une suite d'étapes conduisant d'un état initial à un état final.

Se réalise :

• Sans la réaction de l'univers.

• Sans affecter les étapes sur l'univers réel (simulation, construction de plans avec l'état révocable - en paramètre des algorithmes -).

• Est une stratégie révocable pour un univers irrévocable.

• Fonctionne bien dans un univers prévisible.

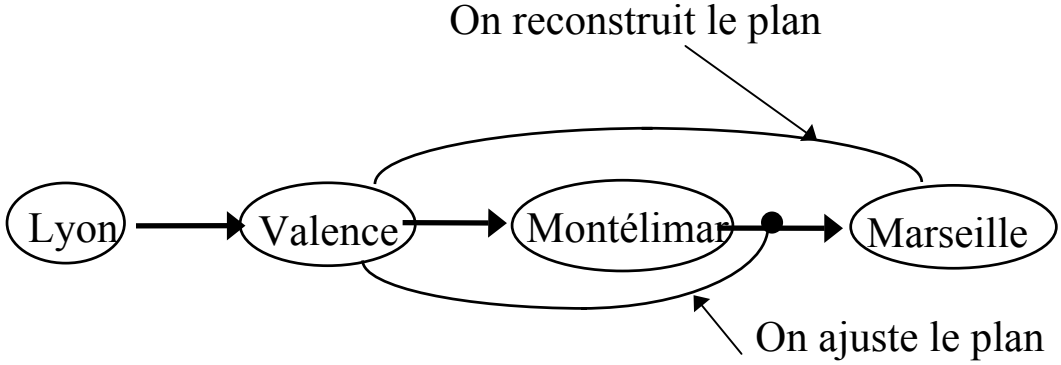
Planification d'univers imprévisible

Deux solutions :

1- Construire un plan en partant de la situation d'échec, ou

2- Ajuster le plan pour contourner la situation d'échec puis reprendre le plan initial.

Exemple :



Les implications de l'ajustement dynamique de plans

1- Décomposition du problème en sous problèmes aussi indépendants que possible (modularité)

a) l'échec d'un sous problème ne remet pas en cause les sous problèmes indépendants

b) réduit la complexité combinatoire du problème (réduit les dimensions de l'univers).

2- Enregistrer les étapes et les raisons de ces étapes. Si l'étape échoue, il est facile de déterminer les parties du plan dépendantes qui nécessitent d'être revues.

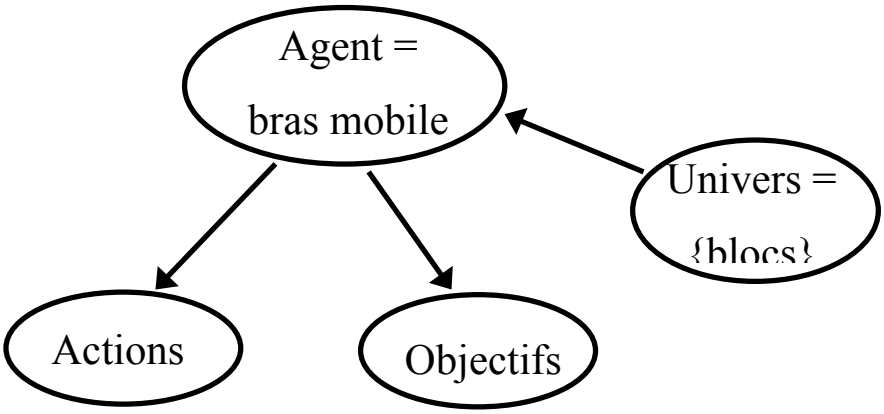
(Cela est naturel avec le chaînage arrière (e.g. Emycin)).

La planification définit des schémas idéaux que l'on devra réajuster avec les conditions réelles.

Strip

Planification dans l'univers particulier par le schéma *si-ajout-supprime* : le monde des blocs (STRIP, bras mobile) .

- Univers composé d'objets cubiques et une table.
- Table fixe, blocs mobiles (manipulation par un bras mobile)
- Bras mobile muni d'une pince; ne peut prendre au plus qu'un bloc à la fois.
- Quand un bloc n'est pas déplacé, soit
  - il repose sur la table
  - il est sur un autre bloc
  - il est dans la pince

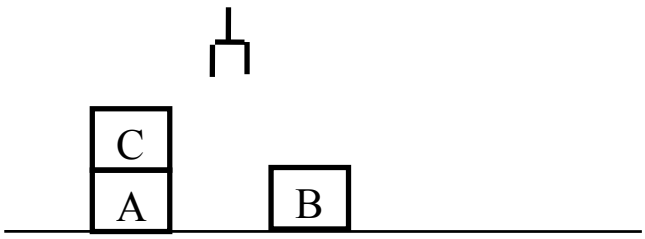


Modélisation de la connaissance

Les connaissances : Univers, But, Actions,

1- L'univers = modélisation des états de l'univers.

Exemple :



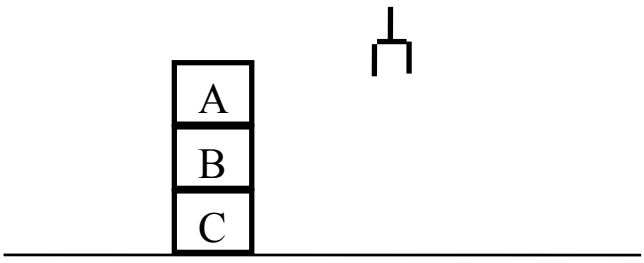
Cet état est modélisé par exemple avec le calcul des prédicats :

sur(C, A).	**libre**(C).
surtable(A).	**libre**(C).
surtable(B).	**mainvide.**

Si la pince tient le cube X, le prédicat **tenu**(X) est vrai.

2 - Le But

Exemple :



Etat représenté par une formule du calcul des prédicats = conjonction

de littéraux :

sur(A,B) & sur(B, C) & surtable(C) & mainvide & libre(A).

3- Les actions

D'une manière générale, dans un système de représentation de connaissances, les actions sont décrites par :

- soit le COMMENT de l'action (procédural)
- soit les EFFETS de l'action (effets de bord, non procédural)

Ici, les actions sont décrites par leur effet et non par leur comment

=> la description n'est pas procédurale.

Une action est décrite par :

1- **Précondition** d'activation dans l'état courant = une formule du calcul des prédicats.

2- Effets de l'action sur l'univers par :

- Une liste des **ajouts** : liste de formules qui deviendraient vraies (une fois l'action accomplie);
- Une liste des **retraits** : liste de formules qui ne sont plus vraies.

Cette liste correspond en général à la liste des préconditions que l'on retire.

Par ses influences sur l'univers, une action est également appelée un OPERATEUR.

Exemples d'actions de STRIP pour le bras mobile et blocs

a) **Prendre(X)** : *X est un bloc sur la table*

Préconditions : surtable(X) & mainvide & libre(X)

Retraits : surtable(X) & mainvide & libre(X)

Ajouts : tenu(X).

b) **Poser(X)** : *X est posé sur la table*

Préconditions : tenu(X)

Retraits : tenu(X)

Ajouts : surtable(X) & libre(X) & mainvide.

c) **Empiler(X, Y)** : *Placer X sur Y*

Préconditions : tenu(X) & libre(Y)

Retraits : tenu(X) & libre(Y)

Ajouts : sur(X, Y) & libre(X) & mainvide.

d) **Dépiler(X, Y)** : *X est sur Y*

Préconditions : sur(X, Y) & libre(X) & mainvide.

Retraits : sur(X, Y) & libre(X) & mainvide.

Ajouts : libre(Y) & tenu(X).

• **Un plan** = une liste d'opérateurs instanciés (valeurs affectées aux variables qui figurent dans le schéma d'une règle).

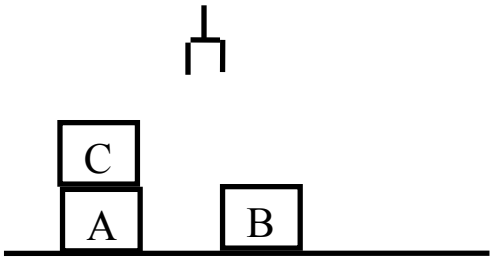
Trouver le plan = trouver un chemin dans un graphe.

[l'état initial + {opérateurs}] = définition implicite d'un graphe.

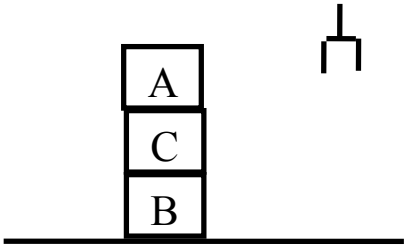
dont les sommets sont les états

et les arcs sont des opérateurs instanciés.

Un exemple :



Etat initial



Etat final

Stratégie Means-Ends : mise en relation "des fins et des moyens"

Principe : On part du but (chaînage arrière, orienté but). On considère l'état courant et on applique l'opérateur qui nous rapproche du but (état final).

Exemple :

Plan		**Pile de Buts** (de haut vers le bas)	
nil		sur(A,C) & sur(C,B)	
Depiler(C,A)	**(1)**	sur(A,C)	
décomposition			
Empiler(C,B)		sur(C,B)	
...			
...		Empiler(C,B)	*sur(C,B) figure*
...		tenu(C) & libre(B)	*dans les ajouts*
...	**(2)**	libre(B)	*d'Empiler(C,B)*
		tenu(C)	
		Depiler(C,A)	
		mainvide & sur(c,y=A) & libre(C)	
	(3)	mainvide	
		sur(C, A)	
		libre(C)	
		...	

Constat : selon le plan partiel ci-dessus, les opérations Dépiler(c, a) & Empiler(c, b) permettent de mettre C sur B (étape 2). A cette étape, il reste à réaliser "sur(a, c)" de l'étape (1).

en (1) : Empiler(x, y) avec x=C, y=B

permettrais d'atteindre sur(c, b) mais cet opérateur n'est pas directement applicable. Pour ce faire, sa précondition doit être empilée dans la pile des buts.

en (2) : alternative entre Prendre et Dépiler pour atteindre "tenu(c)"

on choisit la première règle dans l'ordre d'entrée (Dépiler) et on dépile.

On peut atteindre (réaliser) un but simple qui peut être défait pour atteindre un autre but : cas d'**échec complexe.**

Remarque :

une stratégie orientée données serait trop "naïve" : partir des faits et appliquer les opérateurs qui peuvent l'être. Manipuler les blocs sans objectif (voire inutilement) jusqu'à arriver éventuellement au but (explosion combinatoire)

Algorithmes et Programmes Prolog

Algorithme sans cas d'échec (sans retour arrière)

On se servira d'une pile des buts dans laquelle on empile des buts composés, leur décomposition (des buts simples) ainsi que les opérateurs.

Le plan est une liste qui contiendra uniquement des opérateurs.

Procédure FIN-MOYENS =

Etat <- état initial

Plan <- NULL *Le plan est une liste d'opérateurs*

Empiler But final *Dans la Pile des buts*

BUT-COMPOSE

retourne Plan.

Fin FIN-MOYENS**;**

Remarques :

En cas de succès, le Plan contiendra les opération à appliquer à l'état initial pour atteindre l'état final.

Le but initial n'est pas forcément composé (voir les remarques plus loin).

Procédure BUT-COMPOSE =

B <- sommet de la Pile *sans modifier la Pile*

Si B est atteint dans Etat Alors

Dépiler

OPERATEUR *Applique opérateur*

Sinon

(a) Choisir une décomposition de B = B1 & B2 & ... & Bn

Empiler les buts simples Bi, 1≤ i ≤ n, dans l'ordre de la décomposition

BUT-SIMPLE

Fin si

Fin BUT-COMPOSE;

Remarques :

• La décomposition représente une stratégie de contrôle. On peut envisager :

- une décomposition de gauche à droite simple.

=> Elle peut poser des problèmes;

- une décomposition où les buts simples actuellement vérifiés seront moins prioritaires; on donne la priorité aux buts simples non vérifiés;

- une décomposition qui tiendrait compte d'un fonction heuristique;

• Traitement lorsque le but n'est pas composé (voir plus bas)

```
Procédure BUT-SIMPLE =
    B <- sommet de la Pile des buts
    Si B est atteint dans Etat Alors
        Dépiler
        Si sommet de Pile est un but composé

    (b) Alors                    on suppose que le but composé est atteint
            Dépiler                  après ce but composé, il y a un
            OPERATEUR                opérateur(sauf pour le but initial)
        Sinon
            BUT-SIMPLE
        Fin si
    Sinon

    (c) Choisir un opérateur F dont la liste d'ajouts contient un littéral qui
                                                          s'unifie avec B
        Empiler (F)

    (d) Empiler (préconditions de F)
        BUT-COMPOSE
    Fin si
Fin BUT-SIMPLE;
```

Remarques :

(c) point de choix modifiable selon la stratégie de recherche.

Par exemple, on peut préférer les opérateurs dont les préconditions sont plus simples ! (e.g. Celles de Prendre sont incluses dans celles de Dépiler)

NB : mainvide fait partie de plusieurs liste d'ajouts (e.g. Poser et Empiler).

=> Pour réaliser mainvide, on pourrait utiliser Empiler avant Poser.

(b, d) Les préconditions de certains opérateurs ne sont pas sous une forme composée (cf. Poser(X) avec la précondition tenu(X)). Pourtant, l'algorithme supposera tenu(X) comme un but composé (comme n'importe quelle précondition).

Pour palier ce problème, on peut ajouter le fait **true** (toujours vrai) aux préconditions simples.

- Autre problème en **(b)** : le but composé qui donne lieu à une décomposition n'est pas forcément resté vérifié lorsque ses sous buts se réalisent.

Par exemple, soit les préconditions d'un opérateur de la forme :

[b1 & b2 & b3, ...] *la pile des buts*

Une décomposition de ce but composé donne par exemple :

[b1, b2, b3, b1 & b2 & b3, ...] *la pile des buts*

Si b1 est vérifié; il est dépilé puis, on s'occupe du but simple b2....

Si la réalisation de b2 modifie l'Etat tel que b1 ne soit plus vérifié (e.g. b1 = mainvide et b2 nécessite de dépiler; ce qui supprime mainvide).

=> pour réaliser b1 & b2 & b3, la réalisation de b2 annule une précondition de b1. Lors de retour au but composé b1 & b2 & b3, on aura

une incohérence.

Par ailleurs, si on re-vérifie ce but composé, on risque de recommencer les choses déjà faites.

Une solution partielle à cette situation peut se trouver dans la stratégie de décomposition : vérifier et préserver les sous buts déjà avérés et s'occuper de ceux non encore vérifiés. Mais le problème persiste : la réalisation d'un sous but peut être remise en cause par la réalisation d'un autre sous but qui le suit.

Voir plus loin (Exemple 6).

```
Procédure OPERATEUR=
    Si Pile des buts n'est pas vide alors
        F <- sommet de Pile          sans modification de la pile
        Plan <- cons(F, Plan)        ajout en tête

    (e) Etat <- appliquer(F, Etat)   application des ajouts et retraits
        Dépiler                      l'opérateur F
        BUT-SIMPLE
    Sinon
        C'est gagné
    Fin si
Fin OPERATEUR
```

Remarque :

Un retour arrière sur le point **(e)** est possible. L'Etat doit donc être révocable (passé en paramètre).

Remarque sur les boucles et opérateurs répétés :

Pour éviter les boucles, on utilisera une liste d'opérateurs déjà envisagés dans le plan. Une instance d'un opérateur sera prise en compte si elle ne figure pas dans le plan partiel (ou dans la liste commulative des opérateurs).

Cependant, cette recherche d'opérateur déjà existant doit tenir compte de l'état d'instanciation des variables de l'opérateur (voir le prédicat pas_dans).

La différence entre le Plan partiel et la liste cumulative des opérateurs (Liste_Accu_op) : la pile des opérateurs est enrichie quand on choisit un opérateur (pas encore appliqué) alors que le Plan, qui contient lui aussi des opérateurs, est modifié lorsque l'on applique un opérateur et modifie Etat.

Programme Prolog

Version SWI Prolog sur PC

```
?- op(1200, fx, règle).
?- op(1190, xfx, '::').
?- op(1100, fx, si).
?- op(980, xfx, alors).
?- op(954, xfy, et).

% Le prédicat principal
fin_moyens(Plan_final, Etatout) :-
        recharger_base(But,  Etat),          % extraction du But et l'état initial
        Pile_but=[But], Plan=[],             % Plan des opérateurs
        Liste_Accu_op=[],                    % les OP déjà prévues. Pour éviter les boucles.
        but_compose(Pile_but,  Etat, Etatout,  Plan,  Plan_final,  Liste_Accu_op).

%---------------------------------------------------------------
% dépend du format dans lequel l'état initial est donné.
recharger_base(But,  Etat) :-
        abolish(règle, 1),  abolish(but, 1),
        extraire_faits(Etat),                    % extraction des faits
        but(But).                                % le But

%---------------------------------------------------------------
but_compose([B|Bs], Etatin, Etatout,  Plan_in,  Plan_out,  Liste_Accu_op) :-
        est_verifie(B, Etatin)
         –> applique_operateur(Bs,  Etatin,  Etatout,  Plan_in,  Plan_out,  Liste_Accu_op)
                                         % sommet de BS est un opérateur
        ; (decomposer_empiler_heuristique(B,  Etatin, [B|Bs],  Pile2),
                                         % la décomposition de B s'ajoute  à  [B|Bs]
          but_simple(Pile2, Etatin, Etatout,  Plan_in,  Plan_out,  Liste_Accu_op)
         ) .
```

```
%-------------------------------------------------------------
but_simple([], Etat, Etat,  Plan ,  Plan,  _Liste_Accu_op ) :-
        write(' but_simple; pile vide. A-t-on fini ? OUIIIIIII '),  nl.

but_simple([B|Bs], Etatin, Etatout,  Plan_in,  Plan_out,  Liste_Accu_op) :-
        est_verifie(B, Etatin)
        –>
        ((Bs=[B1|B1s],  B1 = (_ et _))                % B1 est compose' : format B1 et B2
         –> applique_operateur(B1s, Etatin, Etatout,  Plan_in,  Plan_out,  Liste_Accu_op)
         ;   but_simple(Bs,  Etatin, Etatout, Plan_in,  Plan_out,  Liste_Accu_op)
        )
        ; ((regle ID :: si Cond alors Act),             % échec possible
          dans_ajouter_de_postecond(B,  Act),
          instancier_regle(ID, Etatin),               % Instanciation de l'opérateur dans Etatin
          n_annule_pas_prec(ID,Bs),         % pb d'annulation d'effets. voir trace plus bas
          pas_dans(ID, Liste_Accu_op),              % on évite les boucles
          Pile_temp=[Cond, ID|[B|Bs]],              % ID sera l'opérateur a' appliquer
          but_compose(Pile_temp,  Etatin, Etatout,  Plan_in,  Plan_out,  [ID|Liste_Accu_op])
        ).

%-------------------------------------------------------------
% Instancier la regle ID dans Etat pour éviter les variables (pas toujours possible)
instancier_regle(ID, Etat) :-
        copy_term(Etat, Copy_Etat),
        (regle ID :: si Cond alors Act),
        est_verifie(Cond, Copy_Etat),
        executer(Act, Copy_Etat, _Etatemp).

instancier_regle(_ID, _Etat).                       % tout op n'est pas instanciable (ca dépend d'Etat)

%-------------------------------------------------------------
% Applique_operateur(Liste,  Plan_in,  Plan_out)
% Le sommet de Liste contient un opérateur à appliquer
```

```
applique_operateur([], Etat, Etat,  Plan,  Plan,  Pile_actions) :-
        write('c est gagne,  plan = '),  write(Plan),  nl,
        write('la Pile des actions est : '),  write(Pile_actions),  nl.

applique_operateur([B|Bs],  Etatin, Etatout, Plan_in,  Plan_out,  Liste_Accu_op) :-
        append(Plan_in, [B],  Plan_temp),          % Plan enrichi
        (règle B :: si Cond alors Act),
        est_verifie(Cond,  Etatin),                % Instanciation des variables
        executer(Act,  Etatin, Etatemp),
        but_simple(Bs,  Etatemp, Etatout,  Plan_temp,  Plan_out,  Liste_Accu_op) .
%---------------------------------------------------------------
% dans_ajouter_de_postecond(B,  Act) : B peut être une conjonction B1 et B2..Bn,
% Act idem. Vérifier que B est dans la partie ajouter de l'Action Act

dans_ajouter_de_postecond((B et Bs), Act) :-
        !, b_dans_ajouter_de_postecond_Act(B, Act) ,
        dans_ajouter_de_postecond(Bs,Act).
dans_ajouter_de_postecond(B, Act) :-
        b_dans_ajouter_de_postecond_Act(B, Act).

b_dans_ajouter_de_postecond_Act(B, (B1 et _)):-
        B=B1 .
b_dans_ajouter_de_postecond_Act(B, (B1 et _)):-
        B1=ajouter(B).
b_dans_ajouter_de_postecond_Act(B, (_ et Bs)) :-
        b_dans_ajouter_de_postecond_Act(B, Bs).
b_dans_ajouter_de_postecond_Act(B, Act) :-
        not(Act = (_ et _)) , (Act=ajouter(B) ; B=Act).
%---------------------------------------------------------------
% Problème d'annulation des effets. eg. on doit réaliser "mainvide & libre(a)".
% on dépile(b,a) pour avoir libre(a) puis pour réaliser mainvide, on empile(b,a)
% mais après empiler(b,a), libre(a) est annulé. Or, on ne le voit pas. Il faut
% constater que empiler(b,a) supprime libre(a) qui est la raison d'être de dépiler(b,a).
```

```
% On a un sous but B pour lequel on a trouvé un opérateur Op.
% Il faut regarder si cet OP n'annule pas, par sa partie supprimer, la raison
% d'être de l'opérateur précédent dont B est une précondition.
% Algorithme : remonter la pile des Buts, sur le 1er op OP1 rencontré, regarder si la
% partie supprimer d'OP n'enlève pas la raison d'être d'OP1 (qui précède OP1)

n_annule_pas_prec(_Op, []).
n_annule_pas_prec(Op, Bs) :-                    % B est un BUT
        raison_d_op_precedent(Bs, Raison),      % Raison est un BUT
        suite_n_annule_pas_prec(Raison, Op).

suite_n_annule_pas_prec(true, _) :- !.
suite_n_annule_pas_prec(Raison, Op):-
        (regle Op :: si _Cond alors Act),
        pas_dans_suppr_de_postecond(Raison, Act).

raison_d_op_precedent([], true).                % pour le but true, il ,'y a jamais d'op
raison_d_op_precedent([X|L],  Raison) :-
        est_un_op(X)                            % A existe sauf au départ => pas d'op
        –> L=[Raison|_]
        ; raison_d_op_precedent(L, Raison).

est_un_op(X) :-
        (regle ID :: si _Cond alors _Act), X=ID.    % pour éviter Exception de SWI

pas_dans_suppr_de_postecond(B, Act):-
        not(dans_suppr_de_postecond(B, Act)).

dans_suppr_de_postecond((B et Bs), Act) :-
        !, b_dans_suppr_de_postecond(B, Act) ,
        dans_suppr_de_postecond(Bs, Act).

dans_suppr_de_postecond(B, Act) :-
        b_dans_suppr_de_postecond(B, Act).
```

```
b_dans_suppr_de_postecond(B, (B1 et _)):-
        B1=supprimer(B), ! .
b_dans_suppr_de_postecond(B, (_ et Bs)) :-
        b_dans_suppr_de_postecond(B, Bs).
b_dans_suppr_de_postecond(B, Act) :-
        not(Act = (_ et _)) ,Act=supprimer(B).
%-------------------------------------------------------------
% pas_dans(B, Pile_op)        Vérifier que B n'est pas dans Pile_op
% Réussit pour le même foncteur F et pour tout couple d'argument X et Y figurant dans Pile_op.
% il faut avoir  : variable contre variable ou constante contre la même constante.
% Si var contre constante => échec ?      car il peut y avoir une Instanciation différente de var

pas_dans(_B,[]).
pas_dans(B,[X|L]) :-
        pas_compatibles(B,X),
        pas_dans(B,L),
        pas_dans(X,L).                          % des choses instanciées. Refaire le test total

pas_compatibles(X,Y) :-
        not(buts_compatibles(X,Y)).

buts_compatibles(X,Y) :-
        functor(X,F,N), functor(Y,F,N),
        X =.. [F|L], Y =.. [F|L1],              % faut le même foncteur
        args_compatibles(L, L1), !.

args_compatibles([],[]).
args_compatibles([X|L], [Y|L1]) :-
        (var(X), var(Y) , X=Y ;                 % même si on a not plus haut. Prédicat réutilisable
        nonvar(X), nonvar(Y), X=Y),             % Mais pas var contre non var qui s'unifient
        args_compatibles(L, L1).

%-------------------------------------------------------------
% vérifier er instancier les variables de l'opérateur
```

```
est_verifie((C et Cs), Etat) :-          % Le but est vérifié dans l'état actuel de la base
        !,  est_verifie(C, Etat) ,       % le ET est associatif à droite mais '()' possible
        est_verifie(Cs, Etat).

est_verifie(C, Etat) :-
        verifier(C, Etat).

% verifier(B) : verifier B sans modifier la base.
verifier(true, _) :-!.                   % pour fabriquer des but composés artificiels (voir PINCE)
verifier(ajouter(_), _) :-!.             % ne rien faire
verifier(supprimer(_), _):- !.           % ne rien faire
verifier(C, Etat) :-
        membre(C, Etat).                 % on laisse instancier ??

check(X) :-                              % verifier si la clause existe sinon Exception de SWI
        clause(X, _),  !,  X.            % on utilise clause car l'absence donne exception

%-------------------------------------------------------------
executer((A et B),  Ein,  Eout) :-
        !,
        executer(A,  Ein,  Etemp),
        executer(B, Etemp,  Eout).

executer(ajouter(X), Ein,  Eout) :-
        insere(X,  Ein,  Eout).
executer(supprimer(X), Ein,  Eout) :-
        delete(X,  Ein,  Eout).

%-------------------------------------------------------------
% decomposition heuristique. On met en queue les buts déjà vrais.
% décomposer B et empiler chaque composante dans Pile_in => Pile_out
% Si le but a décomposer est SIMPLE,  ne pas remettre ce but simple
% au sommet,  ca fera 2 fois la même chose au sommet de la pile des buts
```

```
decomposer_empiler_heuristique(But, Etat,  Pile_in,  Pile_out) :-
        But=(_ et _) ,  ! ,
        decomposer_heuristique(But,  Etat,  [],  Decomposition),
        append(Decomposition,  Pile_in,  Pile_out).

% But simple. ne pas décomposer
decomposer_empiler_heuristique(_But,  _Etat,  Pile,  Pile).
decomposer_heuristique(But, Etat,  Queue,  Decomposition) :-
        But=(A et B) ,  ! ,  but_a_sa_place(A, Etat,  Queue,  Decomp_temp),
        decomposer_heuristique(B, Etat,  Decomp_temp, Decomposition).

decomposer_heuristique(B, Etat,  Queue,  Decomposition):-       % arrive au dernier
        but_a_sa_place(B,  Etat,  Queue,  Decomposition).

but_a_sa_place(B, Etat,  Queue, Queue_bis) :-                   % en queue si vérifiée
        est_verifie(B, Etat) ,  !,
        append(Queue, [B], Queue_bis).
but_a_sa_place(B, _Etat, Queue, [B|Queue]).                     % en tète si non vérifiée

%-------------------------------------------------------------
insere(X,  Ein, [X | Ein]).                     % cons

delete(X,  [X|L],  L):- !.
delete(X,  [Y|L],  [Y|L1]) :-
        delete(X,  L,  L1).
append([], X, X).
append([X|L], L1, [X|L2]) :-
        append(L, L1, L2).
```

Exemples d'application

On applique ce moteur à l'exemple des blocs dont les règles sont :

règle depiler(X,Y) :: *% X est sur Y, on prend X*

si sur(X,Y)
et libre(X)
et mainvide
alors supprimer(sur(X,Y))
et supprimer(libre(X))
et supprimer(mainvide)
et ajouter(libre(Y))
et ajouter(tenu(X)).

règle prendre(X) :: *% X est sur la table*

si surtable(X)
et mainvide
et libre(X)
alors supprimer(surtable(X))
et supprimer(mainvide)
et supprimer(libre(X))
et ajouter(tenu(X)).

règle empiler(X,Y) :: *% placer X sur Y*

si tenu(X)
et libre(Y)
alors supprimer(tenu(X))
et supprimer(libre(Y))
et ajouter(libre(X))
et ajouter(sur(X,Y))
et ajouter(mainvide).

règle poser(X) :: % X est pose' sur la table
si tenu(X)
et **true** % problème de but composé artificiel (voir remarque)
alors supprimer(tenu(X))
et ajouter(surtable(X))
et ajouter(libre(X))
et ajouter(mainvide).

Remarque :

lorsque la précondition d'un opérateur est simple, on ajoute un élément neutre (true, toujours vrai) car, la procédure BUT-SIMPLE suppose que toute précondition est un but composé que l'on décompose en sous-buts. Eviter true aurait compliqué inutilement les algorithmes.

Exemples d'exécution

Exemple-1:

but(surtable(a) et surtable(b)).

sur(a,b). libre(a). mainvide. surtable(b).

=> plan = [depiler(a, b), poser(a)]

Exemple-2 :

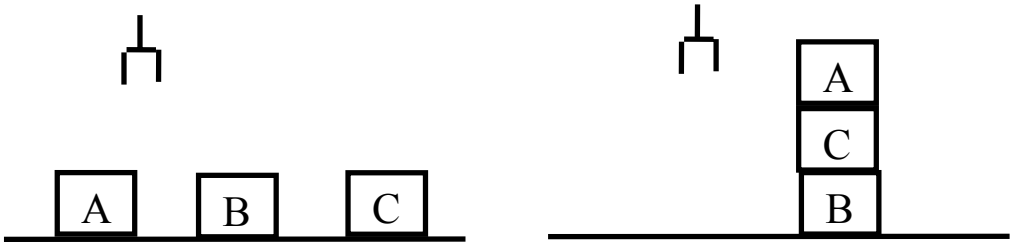
but(sur(a,c) et sur(c,b) et surtable(b)).

sur(c,b). surtable(a). surtable(b).

libre(c). libre(a). mainvide.

=> plan = [prendre(a), empiler(a,c)]

Exemple-3:



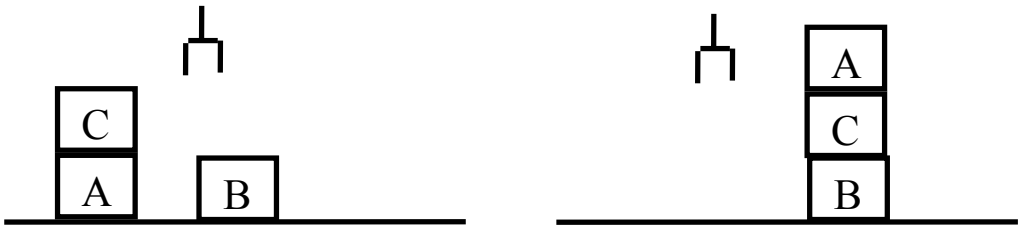
but(sur(a,c) et sur(c,b) et surtable(b)).

surtable(a). surtable(b). surtable(c).

libre(c). libre(a). libre(b). mainvide.

=> plan = [prendre(c), empiler(c,b), prendre(a), empiler(a,c)]

Exemple-4:



but(sur(a,c) et sur(c,b) et surtable(b)).

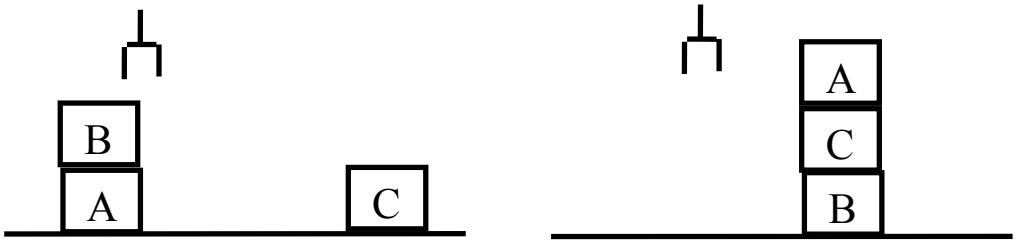
sur(c,a). surtable(a). surtable(b).

libre(c). libre(b). mainvide.

=> plan = [depiler(c, a), empiler(c, b), prendre(a), empiler(a, c)]

=> plan = [depiler(c,a), poser(c), prendre(c), empiler(c, b), prendre(a), empiler(a, c)]

Exemple-5 :



but(sur(a,c) et sur(c,b) et surtable(b)).

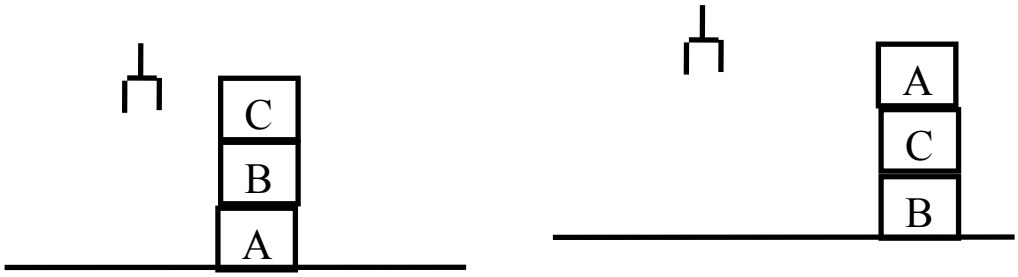
surtable(c). surtable(a). sur(b,a).

libre(c). libre(b). mainvide. % idem ?

=> Plan = [depiler(b, a), poser(b), prendre(c), empiler(c, b), prendre(a), empiler(a, c)]

Exemple-6:

Pour la configuration suivante, on atteint les limites de la version Prolog du système alors que cet exemple est peu différent du précédent.



but(sur(a,c) et sur(c,b) et surtable(b)).

sur(c,b). sur(b,a). surtable(a).

libre(c). mainvide.

Résultat aberrant :

=> Plan = [depiler(c, b), poser(c), depiler(b, a), poser(b), prendre(a), empiler(a, c),
depiler(a, c), poser(a), prendre(c), empiler(c, b)]

Idem si l'état initial est tel que B est sur C et C sur A.

Remarque :

Le problème de cet exemple vient de la décomposition et d'un contradiction avec le fait de préserver les buts déjà réalisés.

Nous avons vu qu'il fallait garantir que l'opérateur choisi n'entre pas en interférence avec le plan partiel actuel ni avec l'Etat actuel.

Le problème de cet exemple est que la décomposition du but initial donne "réaliser sur(a,c) puis sur(c,b)". Or, si on réalise sur(a,c), on n'a pas les moyens de réaliser sur(b,c) si l'on ne veut pas qu'une action annule les réalisations de l'actions précédente. Donc, s'il faut pas défaire sur(a,c), alors comment peut-on réaliser sur(b,c).?

Trace de la résolution

Pour constater ces points, on étudie une partie de la trace de l'exécution de l'exemple-6:

Plan		**Pile de Buts** (de haut vers le bas)	
nil		(1) sur(a,c) & sur(c,b) & surtable(b)	
		(2) sur(c,b)	***décomposition***
||		(3) sur(a,c)	
		(4) surtable(b)	
||		(5) **poser(b)**	*surtable(b) figure*
		(6) tenu(b) & ***true***	*dans les ajouts*
||		(7) true	*de poser(b)*
		(8) tenu(b)	
		(9) **depiler(b, X)**	
		(10) sur(b, X) & libre(b) & mainvide	
		(11) mainvide	
		(12) sur(b, X=a)	
		(13) libre(b)	
depiler(c,b)	<=	(14) **depiler(c, b)**	
		(15) sur(X, b) & libre(X) & mainvide	***vérifié***
		On exécute **depiler(c,b)** qui réalise **libre(b)**; on reprend à (12)	
		(12) sur(b, a)	***vérifié*** *(grâce à depiler(c,b))*
		(11) mainvide	*non vérifié*
		(16) **empiler(c,b)**	*requis pour réaliser* ***mainvide***
		(17) tenu(X) & libre(Y)	***vérifié***

empiler(c,b) <= On exécute **empiler(c,b).**

Avec ces deux actions **depiler(c,b), empiler(c,b)**, on peut constater que **Etat=Etatinit** et **libre(b)** n'est plus vrai, or la décomposition en actions et les exécutions étape par étape déduisent le contraire.

(11) mainvide *réalisé grâce à empiler(c, b)*

(10) Reprise de (10); ce but composé est considéré vrai (!), on va donc exécuter dépiler(b,a); Or, c'est une **erreur** Appliquer_operateur revérifie le but (10) => **échec**

défaire empiler(c,b) Remise en cause jusqu'à (11). Pour satisfaire mainvide, on va utiliser la règle poser(X)

poser(c) <= (18) **poser(c)** *Réalisation de **surtable(b)**.*

(19) tenu(c) & true ***vérifié***

On exécute poser(c).

(11) mainvide *réalisé grâce à poser(c)*

(10) sur(b, a) & libre(b) & mainvide est considéré vrai (ok)

depiler(b,a) <= (9) On exécute **depiler(b,a)**

(8,7,6) tenu(b) & true ***vérifié***

poser(b) <= (5) On exécute **poser(b)**

(4) surtable(b) ***vérifié***

(3) sur(a,c)

prendre(a) <= Pour obtenir ce but, on appliquera :

empiler(a, c) <= **prendre(a), empiler(a, c)** *Réalisation de **sur(a, c)**.*

(2) sur(c,b)

Pour ceci, on appliquera :
depiler(a, c), poser(a), prendre(c), empiler(c, b)

depiler(a, c) <= On constate qu'un état a été modifié pour réaliser un
depiler(a, c) <= autre sous but : sur(a,c) est défait par l'application de:
depiler(a, c) <= ***depiler(a, c), poser(a), prendre(c), empiler(c, b***

(1) sur(a,c) & sur(c,b) & surtable(b)

Ce but composé est supposé vrai (erreur) mais puisqu'il n'y a pas d'opérateur au niveau 0. Même si l'on tente de vérifier ce but composé avant de conclure sur "fin des traitements", on obtient un échec.

Résumé : l'algorithme simple proposé montre ses limites. Cette insuffisance vient du fait que certains opérateurs annulent les pré requis réalisées par les opérateurs qui ont déjà été exécutés.

Proposition :

• revoir la stratégie de décomposition et envisager toutes les permutations !.

• la réalisation d'un sous but ne doit pas défaire les effets (résultats) des sous buts frères (du même but composé).

Nous verrons plus loin une méthodes plus élaborée pour palier ces insuffisances.

Remarques sur la trace précédente

décomposition :

Décomposition heuristique où l'on place en tête (priorité dans les réalisations) les sous buts non actuellement vérifiés dans Etat. C'est ainsi que "sur(c,b)" est considéré vrai et sera traité après "surtable(b)" et "sur(a,c)".

true :

Ce sous but fictif a été ajouté aux préconditions simple (non composées) de certains opérateurs. Ceci permet de :

- après avoir réaliser des sous buts simple, de conclure que l'on est en présence d'un but composé (avec une conjonction) et d'appliquer l'opérateur qui le précède plutôt que d'appeler BUT_SIMPLE qui serait une erreur de traitement.

Améliorations : ordonnancement des opérateurs

• L'algorithme appliqué est simple mais trop exhaustif et insuffisant.

• C'est du chaînage arrière orienté but.

• On pourrait adopter un schéma En-Largeur plutôt que le schéma En-Profondeur employé; avec éventuellement l'algorithme itérative-deeping (profondeur bornée).

• Dans l'algorithme précédent, on risque d'effectuer plusieurs actions qui seront en fin de compte à défaire à cause d'un échec (coût élevé en fonction du nombre de blocs, ...).

Dans le cas d'une scène complexe, on risque l'explosion combinatoire.

Proposition :

Etudier (sans appliquer les opérateurs) les liens entres les opérateurs et établir une séquence ordonnée d'actions (une relation d'ordre partiel).

Pour satisfaire un but B, on choisit une instance d'un opérateur OP1 dont la partie "ajouter" contient B. On cherche ensuite un (ou plusieurs) opérateur OP2 dont la partie "ajouter" contient les préconditions d 'OP1. Ainsi, on établit un lien "réalise" entre OP1 et OP2 (détails ci-dessous)

Mais avant toute chose, pour pouvoir aborder des problèmes plus complexes du monde des blocs, on modifie l'ensemble des opérateurs pour

en proposer une version plus élaborés sans la prise en compte de la pince :

1 • depiler_x_y_empiler_x_z(X,Y,Z).	*déplacer X de Y sur Z*
2 • depiler_x_y_poser_y(X,Y).	*déplacer X de Y sur la table*
3 • empiler(X,Y).	*mettre X sur Y*

L'environnement des blocs sera décrit par les mêmes prédicats que précédemment. Cependant, pour simplifier, la pince est supprimée et le prédicat mainvide n'est plus utile.

Les règles (nouveaux opérateurs)

op1- depiler_x_y_empiler_x_z(X,Y,Z) : *% déplacer X de Y sur Z*

si sur(X, Y) *% équivalent à (sans la pince) :*
et libre(X) *% depiler(X,Y), empiler(X,Z)*
et libre(Z)
alors ajouter(sur(X, Z))
et ajouter(libre(Y))
et supprimer(sur(X,Y))
et supprimer(libre(Z)).

op2- depiler_x_y_poser_y(X,Y) : *% déplacer X de Y sur la table*

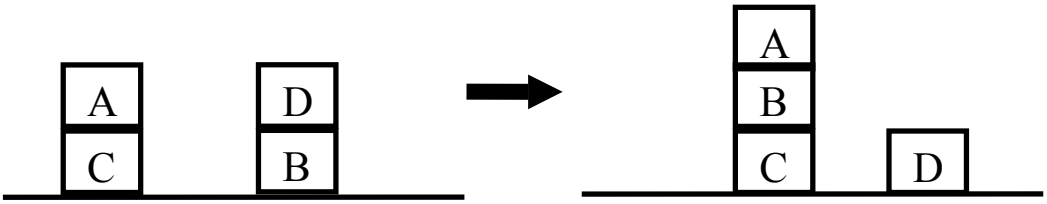
si sur(X, Y) *% équivalent à (sans la pince) :*
et libre(X) *% depiler(X,Y), poser(Y)*
alors ajouter(surtable(X))
et ajouter(libre(Y))
et supprimer(sur(X,Y)).

op3- empiler(X,Y) : *% mettre X sur Y*

si surtable(X)
et libre(X)
et libre(Y)
alors ajouter(sur(X, Y))
et supprimer(surtable(X))
et supprimer(libre(Y)).

Exemple

Considérons le but **sur(a, b)** dans un premier temps.
Supposons que l'on doive déplacer le bloc A sur le bloc B.



trace : (del : supprimer, add : ajouter, **&** : et)

Etat Initial
sur(a,c)
libre(a)
sur(d,b)
libre(d)
surtable(c)
surtable(b)

op1 :
if sur(a,c)
& libre(a)
& libre(b)
add sur(a,b)
& libre(c)
del sur(a,c)
& libre(b)

but
sur(a,b)

Etat Initial
sur(a,c)
libre(a)
sur(d,b)
libre(d)
surtable(c)
surtable(b)

op2 :
if sur(d,b)
& libre(d)
add surtable(d)
& libre(b)
del sur(d,b)

op1 :
if sur(a,c)
& libre(a)
& libre(b)
add sur(a,b)
& libre(c)
del sur(a,c)
& libre(b)

but
sur(a,b)

Etat Initial
sur(a,c)
libre(a)
sur(d,b)
libre(d)
surtable(c)
surtable(b)

op2 :
if sur(d,b)
& libre(d)
add surtable(d)
& libre(b)
del sur(d,b)

op1 :
if sur(a,c)
& libre(a)
& libre(b)
add sur(a,b)
& libre(c)
del sur(a,c)
& libre(b)

but
sur(a,b)

Figure ci-dessus :

Fig-1 : trois étapes de construction de plan pour réaliser un but simple par le chaînage en arrière. Les liens sont de type "réalise".

Remarque :

• Le déplacement du bloc A depuis le bloc C vers B (op-1) n'est pas la seule possibilité. En principe, A peut provenir d'un autre bloc ou de la table. Si ce déplacement n'aboutit pas au résultat final, on essaiera les autres possibilités.

• Comme on peut le constater (fig-1), pour déplacer A depuis C vers B, il y a deux autres préconditions à satisfaire : libre(a) et libre(b).

Un solution pour libérer B est de déplacer D depuis B sur la table (op-2 utilisé au milieu de la figure-1). Ce déplacement établit un lien "réalise" entre les deux opérateurs : la flèche entre libre(b) de l'op-2 vers libre(b) de l'op-1.

- **Un lien "réalise" veut dire que l'opérateur qui établit l'assertion (e.g. op-2) doit apparaître, dans le plan final, avant l'opérateur qui a besoin de cette assertion (e.g. op-1).**

• L'opérateur qui établit l'assertion (e.g. op-2 qui ajoutera libre(b)) ne doit pas forcément être exécuté immédiatement avant l'opérateur bénéficiaire (e.g. op-1 qui utilise libre(b)).

=> Il peut y avoir d'autres opérateurs entre les deux à condition que ces autres opérateurs n'ajoutent pas et ne suppriment pas le fait établi (e.g. le fait libre(b)).

=> De même, si deux opérateurs utilisent un même fait initial déjà établi, l'opérateur qui sera exécuté avant le second ne doit pas supprimer ce fait.

• On peut remarquer (fig-1) que les autres préconditions (de l'op-2 et de l'op-1) sont acquises par l'état initial.

=> Il peut y avoir d'autres moyens d'établir ces préconditions : à l'aide d'autres opérateurs qui réaliseraient des actions plus longues.

par exemple : mettre/enlever le bloc D sur/du bloc B !.

La trace de la Fig-1 donne une solution courte pour atteindre le but final.

• Un plan partiel (milieu de la Fig-1) peut ne pas aboutir à cause des interactions des opérateurs.

Exemple :

soit le but composé **sur(A,B) & sur(B,C)** pour lequel on veut un plan.

Soit le plan partiel de Fig-2, la partie haute pour réaliser sur(A,B).

La figure suivante montre les deux étapes pour réaliser le but.

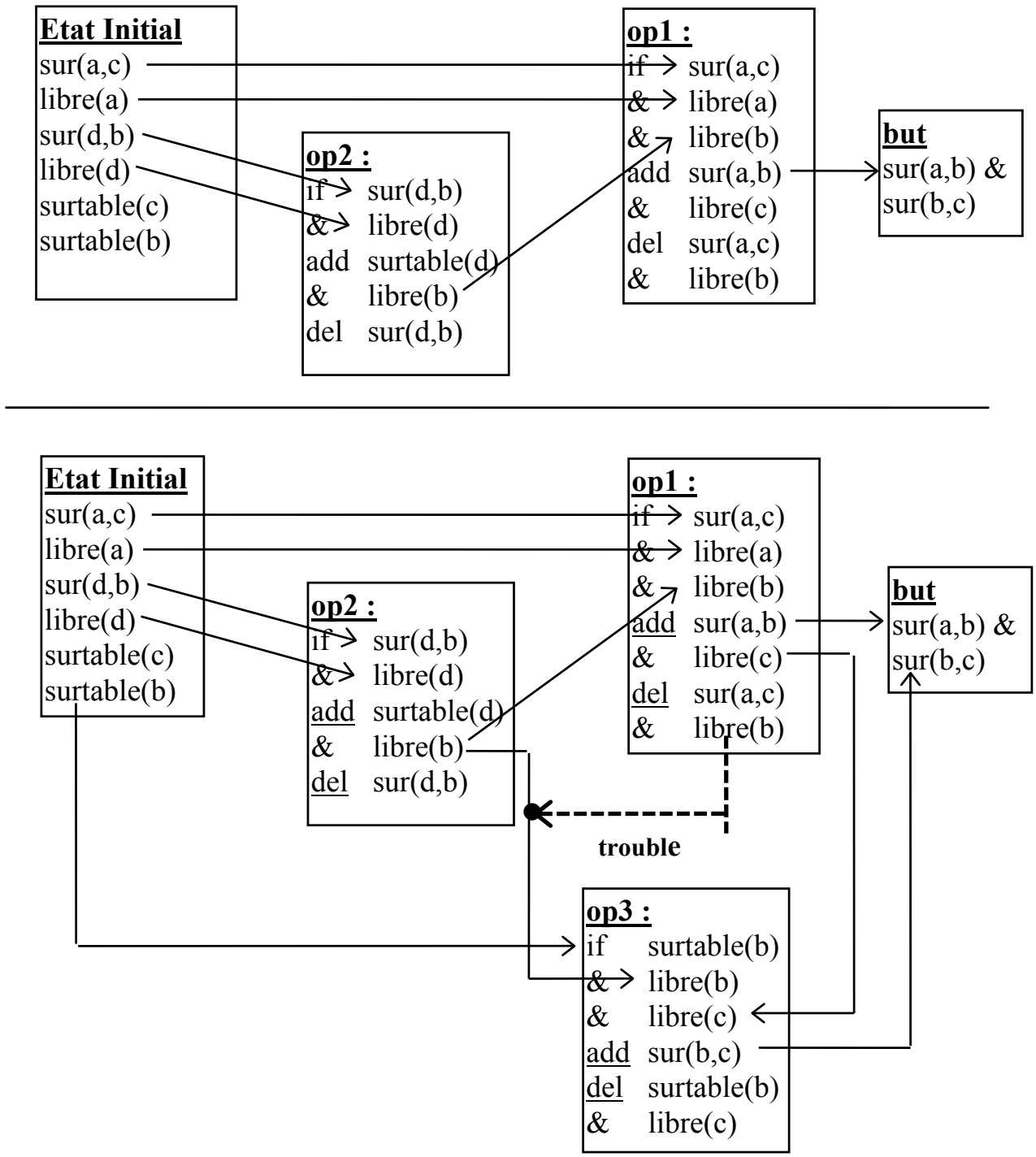


Fig-2 : Le plan est inconsistant car op-1 interfère avec un lien "réalise" entre op-2 et op-3.

La satisfaction du but sur(a, b) par op-1 supprime libre(b) dont op-3 a besoin.

Le problème rencontré est le suivant :

Lorsque l'on ajoute une assertion nécessaire par les préconditions d'un opérateur, on risque de supprimer une assertion ajoutée précédemment pour satisfaire un but ou pour satisfaire les préconditions d'un autre opérateur.

=> il faut une méthode pour détecter de telles interférences aussi tôt que possible pour éliminer les plans partiels inutiles.

Pour cela, on examine chaque lien "réalise" pour vérifier si l'un des opérateurs en jeu peut invalider ce lien (le point "trouble" de Fig-2). Si l'on trouve un tel opérateur, on ajoutera une contrainte d'ordre qui permettra d'éviter la contradiction.

Dans Fig-2, Op-1 supprimerait libre(b) s'il devait figurer après Op-2 et avant Op-3. Pour éviter cette situation, l'opérateur "perturbateur" (ici, Op-1) doit apparaître **avant** l'opérateur qui ajoute l'assertion (ici Op-2) **ou après** l'opérateur utilisant l'assertion (ici Op-3)**. Il ne doit pas figurer entre les deux.**

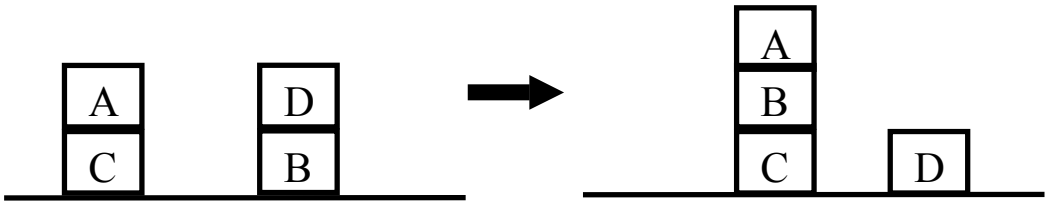
Il peut arriver qu'aucun de ces ordres ne fonctionne. Dans Fig-3, en conséquence du problème posé par Op-1 ("perturbateur") et l'ordre imposé, les liens "réalise" sont remplacés par des liens "avant" dénotant un ordre simple.

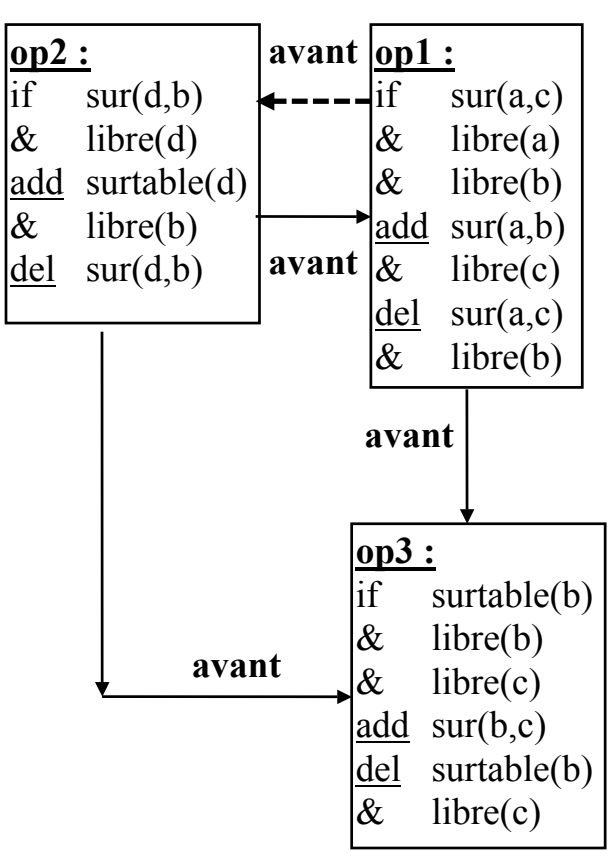
On peut voir dans Fig-3-a que l'Op-1 ne peut pas figurer avant Op-2 car le lien "réalise" d'Op-2 à Op-1 impose l'ordre <Op-2 avant Op-1>.

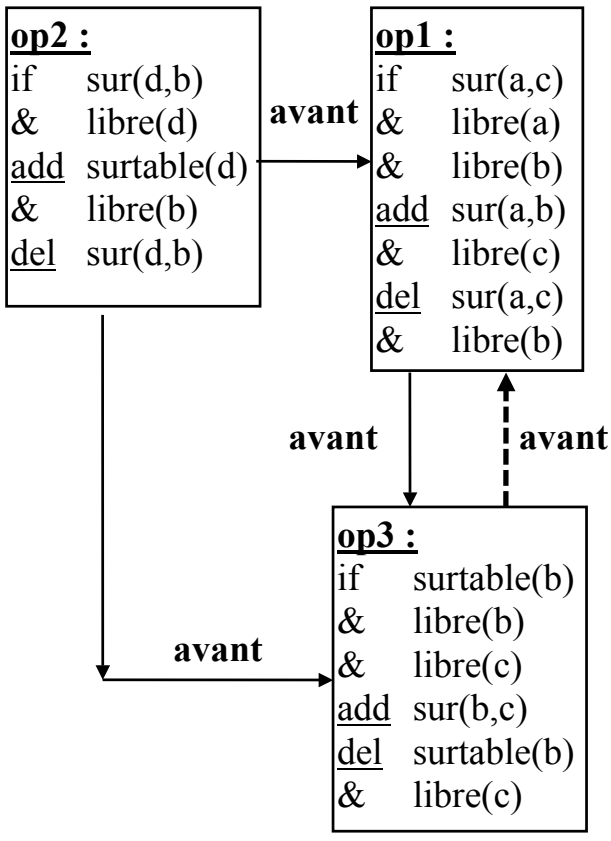
De même, dans Fig-3-b, l'op-3 ne peut pas figurer avant op-1 car le lien "réalise" d'Op-1 à Op-3 impose <Op-1 avant Op-3>. Ce qui rend ce plan partiel incohérent.

On constate donc que cette stratégie de construction de plans permet de **détecter et d'écarter les plans impossibles.**

Rappel du but et de l'état initial :







(a) **(b)**

Figure ci-dessus :

Fig-3 : On peut protéger un lien "réalise" en ajoutant un lien "avant" supplémentaire. Mais on constate que le problème de précédence ne peut être résolu ici sans créer une boucle de relations "avant".

Remarque :

La possibilité d'interférence n'entraîne pas forcément une interférence comme on peut le voir dans Fig-4. Avec la possibilité d'interférence, il faut établir un ordre "avant" qui peut aboutir à une solution qui n'aboutit pas à un contradiction.

Dans la partie haute de la Fig-4, on décide de déplacer A de la table sur B par une instance de l'opérateur 3 (op-3a) au lieu d'envisager l'état initial où A est sur C (comme on l'avait fait dans Fig-1). Ce qui correspond à envisager un autre choix possible pour aboutir à sur(a, b). On constate également comment satisfaire les préconditions d'op-3a à l'aide de deux instances de l'opérateur-2 (op-2a et op-2b).

En bas de Fig-4, on constate la présence d'un opérateur "perturbateur" pour réaliser le but "sur(b,c)" à l'aide d'op-3b. Cette fois, "libre(b)" est ajouté par op-2a, utilisé par op-3a mais supprimé par op-3a alors qu'op-3b en a besoin.

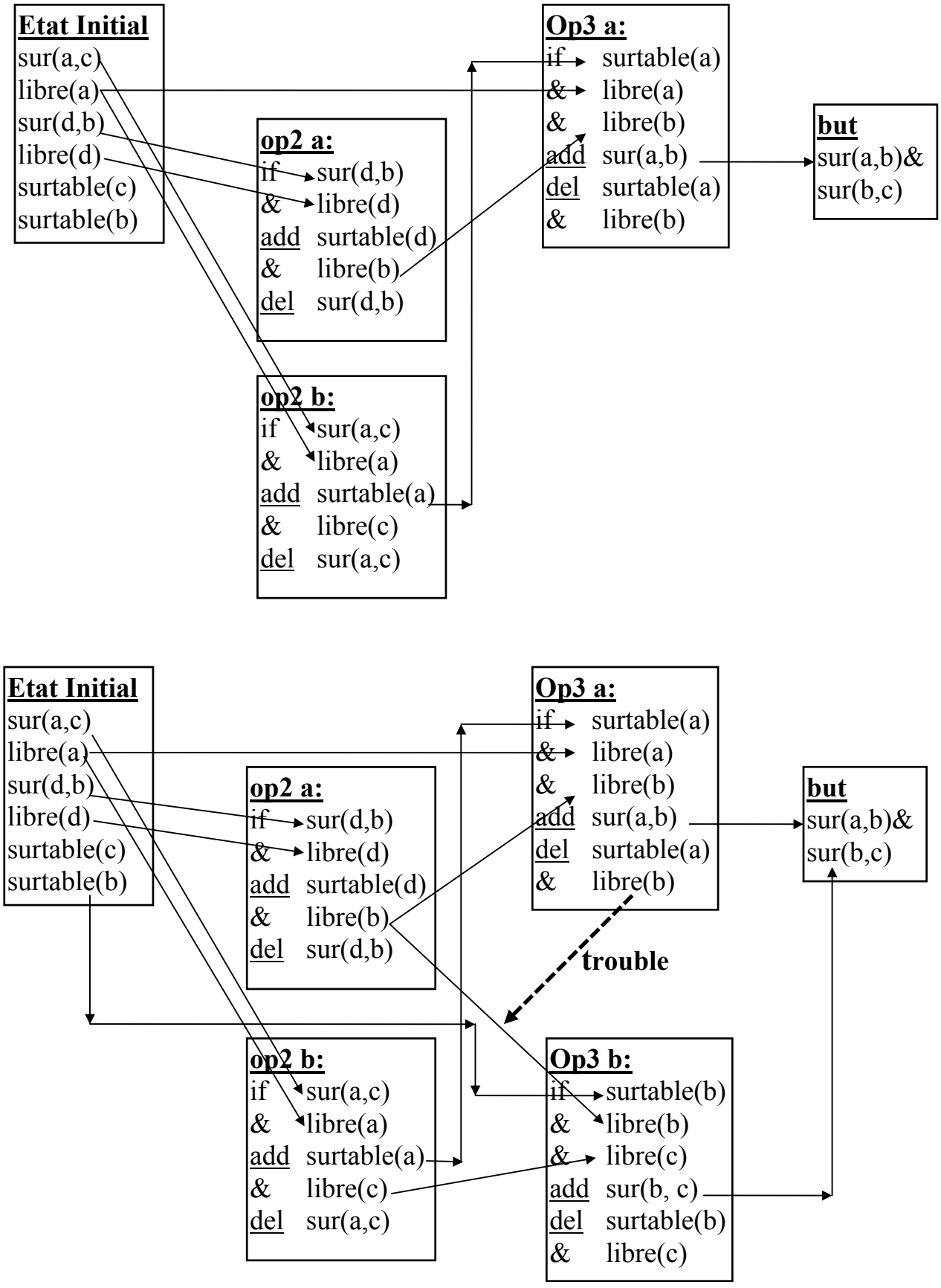


Fig-4 : un autre plan pour réaliser le but "sur(a,b) & sur(b,c)".

D'après Fig-4, Op-3a "trouble" le lien "réalise" entre op-2a et op-3b => **op-3a doit être placé avant op-2a ou après op-3b**.

Mais, op-3a ne peut pas être placé avant op-2a car op-2a réalise une des préconditions d'op-3a.

Mais, op-3a peut être placé après op-3b sans créer de boucle de relation "avant". Remarquons que si cela n'était pas possible, on devrait remettre en cause le plan qui aura conduit à une impasse et essayer les autres possibilités.

Le **plan complet** obtenu est présenté en Fig-5. On remarque que toutes les préconditions sont réalisées et il n'y a pas de boucle "avant".

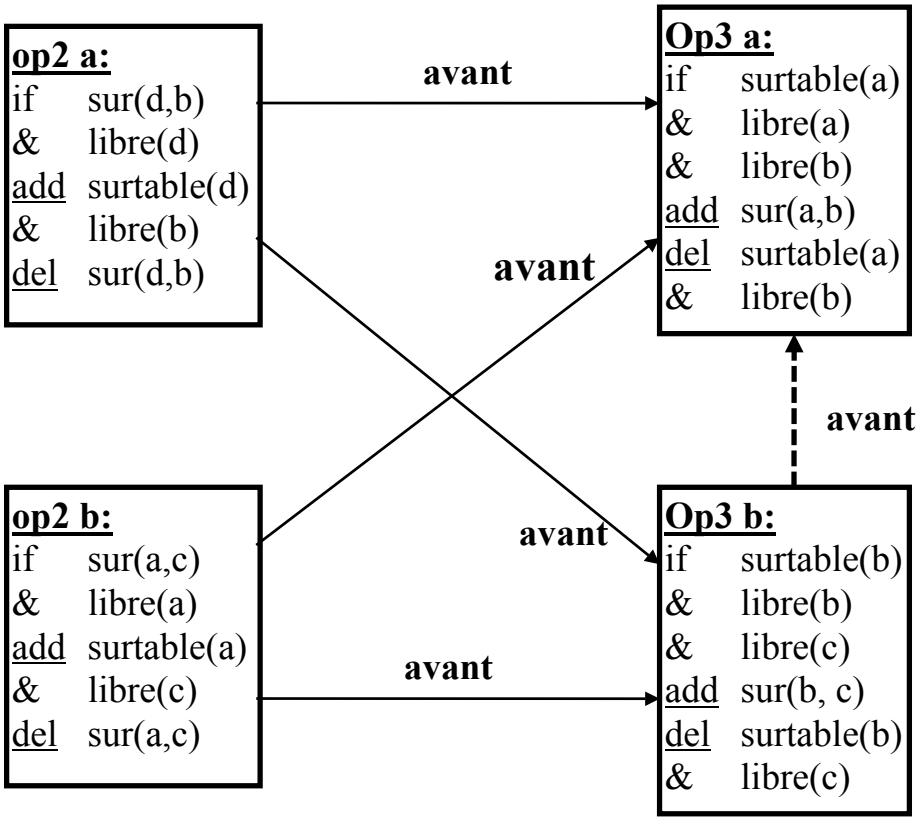


Fig-5 : Le lien "réalise" entre op-2a et op-3b a été protégé par l'ajout

d'un **nouveau** lien "avant" qui force op-3b de figurer avant op-3a pour éviter les troubles.

Remarque : un plan complet ne fournit pas un ordre exact des actions à appliquer pour réaliser le but initial. Ici, le bloc A peut être déplacé sur la table avant D ou vice versa. Ce qui donne les séquences suivantes (au choix) :

Plan-1 :

déplacer A sur la table
déplacer D sur la table
mettre B sue C
mettre A sur B

Plan-2 :

déplacer D sur la table
déplacer A sur la table
mettre B sue C
mettre A sur B

Les **plans linéaires** ci-dessus peuvent être obtenus par l'application d'un tri topologique sur les opérateurs qui possèdent un ordre partiel entre eux.

Algorithme sommaire de construction de plan

• Pour construire un plan :

étendre un plan partiel ne contenant pas d'opérateur jusqu'à obtenir un plan complet (retour arrière en cas d'échec).

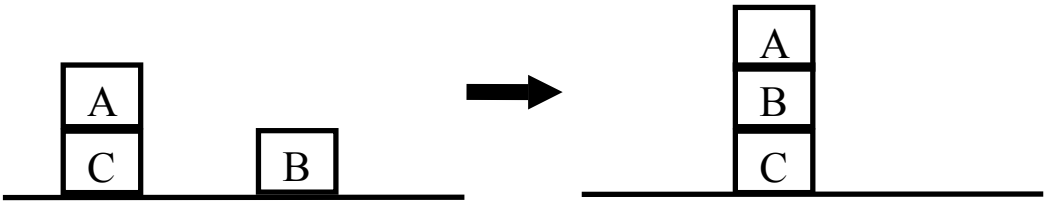
• Pour étendre un plan partiel :

- Si le plan contient une boucle "avant" alors échec
- Si le plan est complet alors succès
- S'il y a un opérateur Opt qui "trouble" un lien "réalise" entre deux opérateurs Op1 et Op2 alors :
  . soit placer un lien "avant" entre Opt et Op1 et étendre le plan
  . soit placer un lien "avant" entre Op2 et Opt et étendre le plan

- Sinon, choisir une précondition C non satisfaite et :
  . soit trouver un opérateur déjà existant qui ajoute C, installer un lien "réalise" et étendre le plan
  . soit instancier un opérateur qui ajoute C, installer un lien "réalise" et étendre le plan.

Exemple récapitulatif simple

Dans l'exemple suivant, on abandonne le bloc D pour montrer de manière plus simple la logique de l'algorithme précédente.



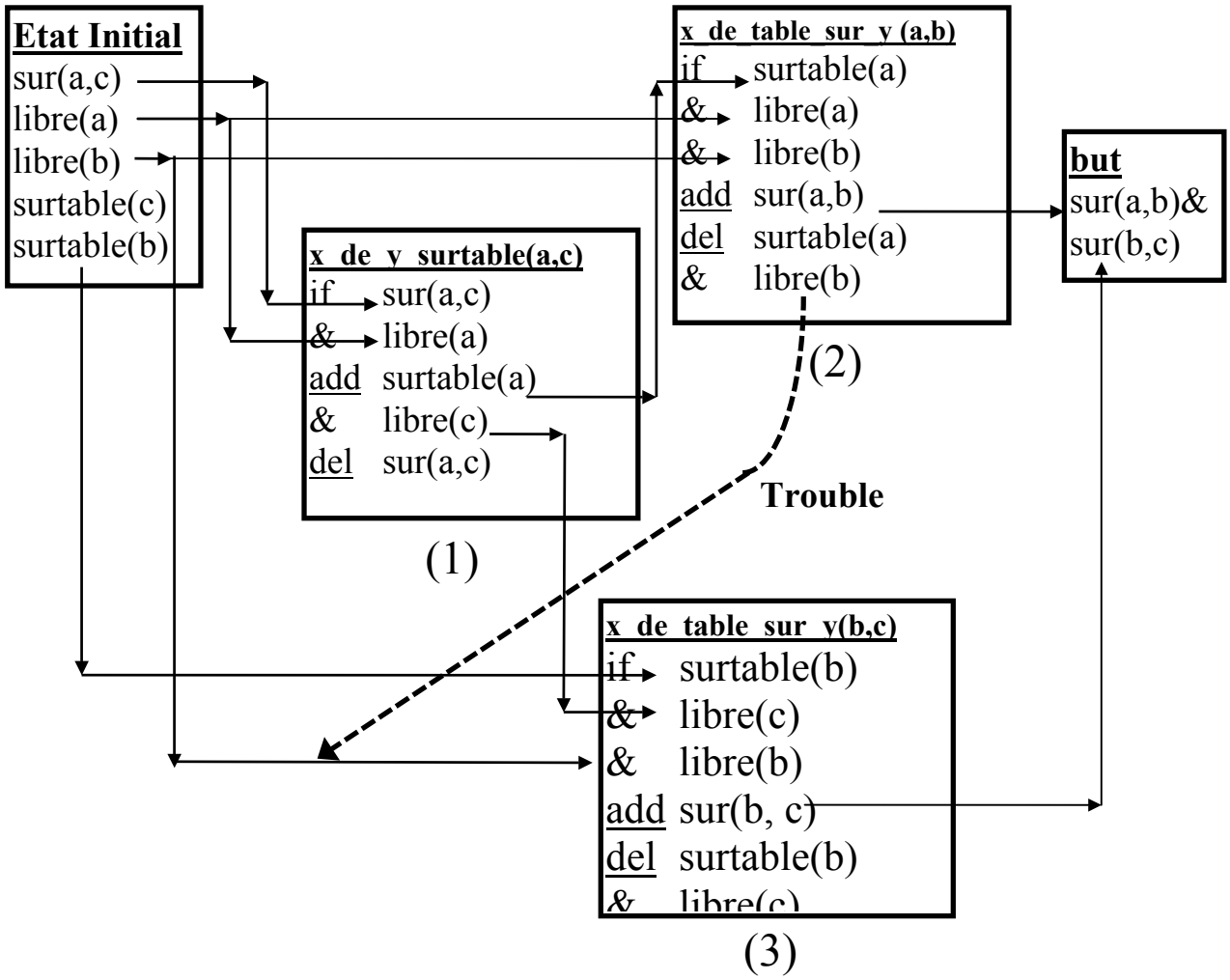


Fig-6 : l'opérateur (2) "**trouble**" le lien (sur libre(b)) de l'état initial à (3).

On a les liens "réalise" suivants :

(1)------- surtable(a) ---→(2) ET (1)-------- libre(c) -----→(3)

Il s'agit d'obtenir, pour un plan complet, un ordre entre (3) et (2).

Or, on constate que (2) supprime libre(b) dont (3) a besoin. (2) devrait être effectuée après (3). Remarquons que (2) ne peut être placée avant (1).

On obtient par l'ajout d'un nouveau lien "avant" (3) → (2) la séquence ordonnée (1) → (3) -→ (2)

Un plan final complet :

Déplacer A depuis C sur la table,

Déplacer B de la table sur C,

Déplacer A de la table sur B,