Travaux Pratiques

Programmation logique

4 Année

Spécialité : Informatique

Option : IPS

Représentation et Modélisation des connaissances
Prolog – TP1

Objectif:
- Prise en main de l'environnement gprolog
- Comprendre le fonctionnement de Prolog

Exercice 1 : **Prise en main de l'environnement**
Ecrivez le programme suivant dans un éditeur et sauvegardez-le en menu.pro

```
/* composition des menus */
/* les entrées */
entree(crudites).
entree(terrine).
entree(melon).

/* les viandes (avec légumes associés) */
viande(steack).
viande(poulet).
viande(gigot).

/* les poissons (avec légumes associés) */
poisson(bar).
poisson(saumon).

/* les desserts */
dessert(sorbet).
dessert(creme).
dessert(tarte).

/* composition d'un menu simple : une entrée ET un plat ET un dessert */
menu_simple(E, P, D) :- entree(E), plat(P), dessert(D).

/* le plat de résistance : viande OU poisson */
plat(P) :- viande(P).
plat(P) :- poisson(P).
```

a – Lancement de l'interpréteur :
Vous devez faire appel à l'interpréteur (gprolog) pour pouvoir interroger votre programme.
Commandes de base :
- Lancer l'interpréteur en tapant la commande gprolog
- Choisir le fichier à interroger :
  - soit en écrivant : **?-[nomfichier].**
  - Soit en écrivant : **?-consult('nomfichier.pro').**
  - Soit en écrivant : **?-consult(nomfichier).**
- Pour recharger (effacer le programme précédent & charger les nouvelles règles) votre programme dès que vous le modifiez avec l'éditeur:
  - **?-reconsult(nomfichier).**

consult/1 et **reconsult/1** ne sont pas des prédicats logiques. Ils ont pour fonction d'exécuter des commandes d'enregistrement de programmes et sont appelés des *prédicats extra-logiques*.

Pour connaître tous les desserts contenus dans la base de faits, vous devez taper:

?-dessert(X).

Pour avoir les réponses suivantes, tapez "**;**" sinon tapez "**Entrée**" pour que la résolution s'arrête.
Attention, n'oubliez pas le point à la fin de chacune de vos requêtes.
Faites afficher tous les menus simples possibles.

b- Formalisez en Prolog les questions suivantes et testez vos requêtes:

- quels sont les menus simples avec des crudités en entrée ?
- peut-on avoir un menu avec des crudités et une mousse au chocolat ?
- quels sont les menus avec du poisson comme plat ?
- quels sont les menus avec du melon en entrée et du poisson comme plat ?

c- Que se passe-t-il si vous inversez l'ordre des buts dans vos deux dernières requêtes ?

Pour visualiser les arbres de résolution vus en TD, vous pouvez demander à l'interpréteur d'afficher son raisonnement pas à pas (les traces). Pour cela :

lancer la requête : **?-trace, <vos requêtes> .**

Instructions:

- creep (ou 'c' sous swi-prolog) : pour passer au pas de résolution suivant;
- exit : représente une démonstration réussie d'un prédicat;
- redo : le système essaie de démontrer le prédicat d'une autre manière en faisant un retour sur un point de choix;
- echec : le système a échoué à démontrer un prédicat.

Faites afficher progressivement les traces des 2 requêtes vues en TD en suivant en parallèle les arbres de résolution que vous avez construits en TD.

d- Que signifie la requête suivante ? Testez-la :

?- menu_simple(E, P, D), entree(crudites).

Exercice 2 : Les relations familiales : utilisation de la Récursivité

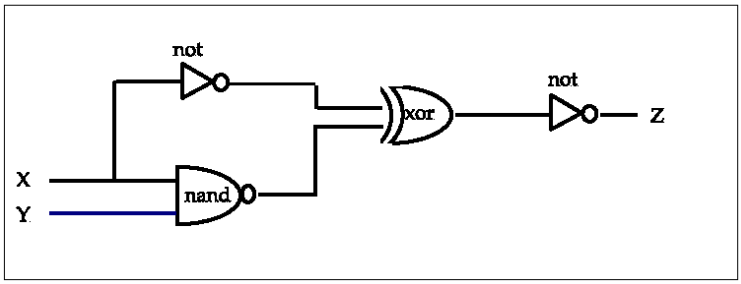
a - Récupérez la base de faits **familleBDF.pro**
Dessinez l'arbre généalogique donné par la base de faits afin de vérifiez les résultats suivants.

b - Définissez et testez les prédicats suivants:

- parent(X,Y) : « X est parent de Y »
- enfant(X,Y) : « X est enfant de Y »

c – Trouvez les 3 solutions possibles pour écrire le prédicat **ancetre/2**. Testez vos résultats en interrogeant le système sur les ancêtres du Duc d'Anjou.
Que constatez-vous ?

Exercice 3 : Le circuit logique



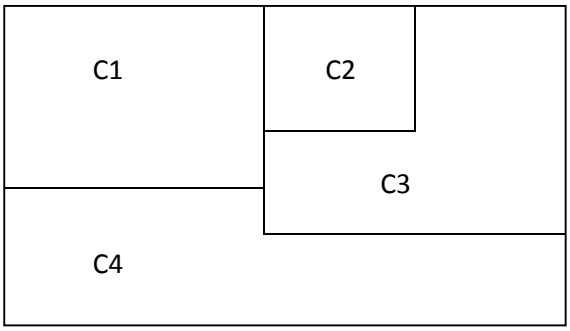
a- Définissez les **prédicats** correspondant aux **composants** (ou connecteurs) logiques. Prenez pour valeurs possibles des entrées ou sorties 0 ou 1 (respectivement pour faux ou vrai).

b- Définissez le **prédicat circuit/3**, tel que circuit(X,Y,Z) est vrai si et seulement si X et Y correspondent à des valeurs d'entrée pour lesquelles Z est la valeur de sortie du circuit.

c- Testez les différentes **requêtes possibles** en faisant varier la valeur des arguments (entrée ou sortie). Déduisez-en la table de vérité du circuit.

Exercice 4 : Gestion des entrées/sorties

On se propose de définir un prédicat permettant de colorier la carte suivante :



Les règles sont les suivantes :

- On dispose de trois couleurs qui sont : vert, jaune et rouge ;
- Deux zones contiguës doivent avoir des couleurs différentes.

a- Ecrivez un prédicat **coloriage(C1, C2, C3, C4)** dont les littéraux sont ordonnés en deux temps. Tout d'abord on génère toutes les valeurs possibles de C1, C2, C3 et C4. Ensuite on vérifie si les colorations obtenues sont conformes à la carte par l'utilisation du prédicat **X \== Y** sur les couleurs des zones contiguës.

b- Reprenez ce prédicat et modifiez le programme en plaçant les tests de différence de couleurs le plus tôt possible dans l'écriture du prédicat, c'est-à-dire en vérifiant les différences de couleurs dès que celles-ci sont instanciées. Quelle en est la conséquence ?

c- On veut permettre à l'utilisateur de choisir lui-même une couleur pour une des positions, s'il en a envie. Pour cela vous devez faire un menu lui laissant le choix entre laisser le système donner une réponse immédiatement ou faire d'abord le choix d'une couleur. Le résultat doit alors tenir compte du choix de l'utilisateur.

Exercice 5 – Résolution d'une énigme policière

Jean a été tué mardi. Les suspects sont : Luc, Paul, Alain, Bernard et Louis. L'assassin est quelqu'un qui peut désirer tuer Jean, qui possède une arme et qui n'a pas vraiment d'alibi pour mardi. On peut désirer tuer quelqu'un, soit par vengeance, soit par intérêt. Pour avoir intérêt de tuer Jean on peut être son héritier, lui devoir de l'argent, ou avoir été surpris par lui en train de commettre un crime. Un alibi donné par quelqu'un de douteux ne peut pas être sérieusement pris en compte.

L'enquête détaillée a permis d'établir les faits suivants :

- Luc a un alibi pour mardi donné par Bernard.
- Paul a un alibi pour mardi donné par Bernard.
- Louis a un alibi pour mardi donné par Luc.
- Alain a un alibi pour jeudi donné par Luc.

- Alain est un personnage douteux.

- Paul désire se venger de Jean.
- Luc désire se venger de Jean.

- Bernard est l'héritier de Jean.
- Jean est l'héritier de Louis.
- Louis doit de l'argent à Jean.
- Luc doit de l'argent à Jean.

- Jean a vu Alain commettre un crime.

- Luc possède une arme.
- Louis possède une arme.
- Alain possède une arme.

→ Écrivez un programme Prolog qui trouve le meurtrier de Jean.

Vous aurez besoin du *méta-prédicat* **not(P)** qui retourne vrai si la résolution du but **P** échoue. Pour satisfaire **not(P)** il faut que toutes les variables de **P** soient liées :

Autorisé : sportif(X) :- fort(X), not(petit(X)).
Interdit : moyen(X) :- not(grand(X)), not(petit(X)).
Autorisé : moyen(X) :- personne(X), not(grand(X)), not(petit(X)).

Représentation et Modélisation des connaissances

Prolog – TP2 & TP3

Objectif:

- formalisation de problèmes en Prolog
- planification
- les listes en Prolog

Exercice 1 – Listes et trace

- Testez les différents prédicats sur les listes vues en cours et en TD.
- Pour les différentes versions possibles du prédicat ***creeListe***, testez le mode d'affichage des traces en posant la requête

?-creeListe(3,[]).

Comparez ces traces avec les arbres de résolution vus en TD.

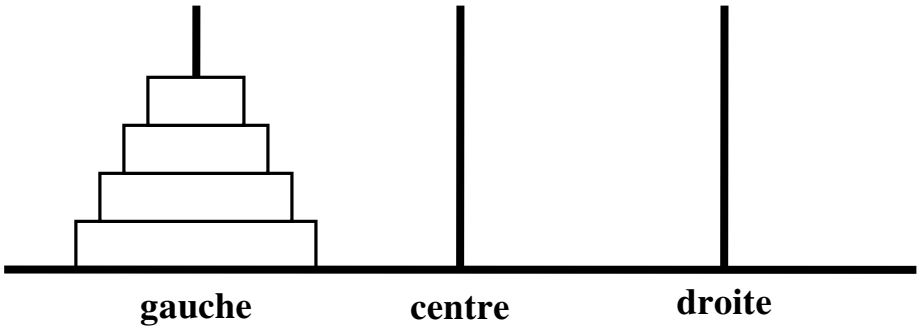
Exercice 2 – Automates finis

On représente les mots sur un alphabet A par des listes, ainsi le mot « abaa » par [a, b, a, a] sur A = {a, b}. Un automate sur A est un ensemble d'états Q = {q0, q1, ...} ayant un état initial q0, un (ou plusieurs) état final, et une relation de transition donnée par des triplets (q, x, q') où x A et q, q' Î Q. Un mot m est reconnu par l'automate s'il existe une suite de transitions de l'état initial à un état final au moyen des lettres de ce mot.

- Ecrire les clauses générales pour cette reconnaissance.
- Ecrire les clauses particulières décrivant l'automate à deux états q0, q1, et la transition définie par tr(q0, a) = q1, tr(q1, b) = q0 avec q0 à la fois initial et final.
- Décrire l'automate reconnaissant les mots contenant le sous-mot « iie » sur l'alphabet A = {a, e, i, o, u}.

Exercice 3 – Les tours de Hanoï

Cet exercice consiste à programmer un « classique » de la récursivité : les « tours de Hanoï ». Ce jeu consiste à transférer tous les disques du poteau de gauche vers le poteau de droite, en ne déplaçant qu'un disque à la fois, et en ne plaçant jamais un disque sur un autre disque plus petit que lui :



Vous aurez besoin d'un prédicat **transferer(N,A,B,I)** où :

- N est le nombre de disques à transférer ;
- A est le poteau où les disques sont placés ;
- B est le poteau de destination ;

– I est le poteau intermédiaire.

Et d'un prédicat **deplacer(A,B)** qui ne fera qu'afficher :
– « On déplace un disque de … vers … ».

Les questions que vous devez vous poser sont :
– Que faire s'il n'y a qu'un seul disque ?
– Que faire s'il y en a davantage ?
– Quelle est la condition d'arrêt ?

Pour vous aider, disons que pour transférer N disques, il suffit de disposer d'un prédicat qui sait déplacer N-1 disques. Selon la figure, il faut transférer trois disques de la gauche vers le centre, de déplacer le grand disque de la gauche vers la droite, puis de transférer les trois disques du centre vers la droite.

Exemple d'exécution pour N=3 :

?- transfer(3, gauche, droite, centre).

On déplace un disque de gauche vers droite
On déplace un disque de gauche vers centre
On déplace un disque de droite vers centre
On déplace un disque de gauche vers droite
On déplace un disque de centre vers gauche
On déplace un disque de centre vers droite
On déplace un disque de gauche vers droite
Yes
?-

Exercice 4 – Le loup, la chèvre et le chou

Le problème *« du loup, de la chèvre et du chou »* est un exemple typique de planification par recherche dans un graphe d'états. Nous ne mettrons pas en œuvre d'heuristique car l'espace de recherche, assez restreint, ne provoque pas d'explosion combinatoire.

Un batelier doit transborder un loup, une chèvre et un chou de la rive gauche à la rive droite d'un fleuve. À chaque traversée, il peut prendre au maximum un seul des trois personnages. De plus, pour éviter des disparitions, il ne peut laisser dans surveillance ni le loup et la chèvre ensemble, ni la chèvre et le chou ensemble. En effet, en son absence, le loup mangerait la chèvre et la chèvre le chou.

On choisit d'implémenter les situations-types (nœuds du graphe) à l'aide de structures de données de type liste **[A,B,C,D]** où :

– A est la position du batelier (gauche ou droite) ;
– B est la position du loup (idem) ;
– C est la position de la chèvre (idem) ;
– D est la position du chou (idem).

La situation initiale est **[gauche,gauche,gauche,gauche]**
La situation finale est **[droite,droite,droite,droite]**

Les contraintes sont les suivantes :

- B doit être à l'opposé de C ;
- C doit être à l'opposé de D.

Les règles de transformations sont représentées à l'aide d'un prédicat **transition/3** :

```
transition('Le batelier traverse seul', [X,B,C,D], [Y,B,C,D]) :-
        oppose(X, Y), oppose(B, C), oppose(C, D).
transition('Le batelier traverse avec le choux', [X,B,C,X], [Y,B,C,Y]) :-
        oppose(X, Y), oppose(B, C).
transition('Le batelier traverse avec le loup', [X,X,C,D], [Y,Y,C,D]) :-
        oppose(X, Y), oppose(C, D).
transition('Le batelier traverse avec la chèvre', [X,B,X,D], [Y,B,Y,D]) :-
        oppose(X, Y).
```

1. Écrivez le prédicat **oppose/2** sachant que les deux valeurs sont *gauche* et *droite*.
2. Explicitez ces règles de transformation en français. Comment les tester ?
3. Écrivez un prédicat récursif **planifie(A,B,C)** qui fonctionne de la façon suivante :
   - A est la situation de départ ;
   - B est la situation d'arrivée ;
   - C est une liste (construite dans le membre gauche du prédicat planifie) qui doit contenir au final la liste des actions décrites dans les transitions.
   - Pour planifier un parcours de A à B, il suffit de trouver une transition qui permet d'aller de A à C, et de planifier un parcours de C à B.
4. Écrivez le prédicat très simple **run/0** qui lance la planification.
5. Lancez la planification. Que remarquez-vous ?
6. Ajoutez un quatrième argument au prédicat planifie afin de construire une liste de situations déjà explorées, et prenez garde de ne pas planifier de parcours de C à B si la situation C fait déjà partie de cette liste !

Attention : Cette liste doit être construite dans le membre droit du prédicat **planifie/4** car elle ne dépend pas d'un but, contrairement à la liste des actions !

Indication : Utilisez le prédicat de liste **is_member/2** avec une négation.

Exercice 5 – Le zèbre

Dans la rue il y a cinq maisons. Les couleurs des maisons sont toutes différentes. Les propriétaires sont de nationalités différentes, leurs prénoms sont différents et leurs boissons préférées sont différentes. Dans chaque propriété vit un animal différent.

1. L'Anglais vit dans la maison rouge.
2. Le Suédois a un chien.
3. L'habitant de la maison verte boit du café.
4. La maison de Jean est voisine de celle du chat.
5. Le Danois boit du thé.
6. La maison verte est à droite de la maison blanche.
7. Pierre a un oiseau.
8. Paul habite la maison jaune.
9. Le buveur de lait habite la maison du milieu.
10. Le Norvégien habite la première maison à gauche.

11. Paul est voisin du cheval.
12. Jacques boit de la bière.
13. L'Allemand se prénomme Hans.
14. Le Norvégien vit près de la maison bleue.
15. La personne qui boit de l'eau est voisine de Jean.

Ecrivez un programme Prolog qui répond aux questions suivantes:

- Qui a un zèbre ?
- Qui boit du lait ?

Exercice 6 – Le singe et la banane

Autre problème classique de planification : *Le singe et la banane…*

Dans une pièce il y a un singe, une boite, et une banane suspendue au milieu du plafond. Le singe est au niveau de la porte et la boite est au niveau de la fenêtre. Pour manger la banane, le singe doit donc **marcher** de la porte vers la fenêtre, **pousser** la boite de la fenêtre vers le milieu de la pièce, **grimper** sur la boite, et prendre la banane. Lorsque le singe a pris la banane, on considère que son but est atteint.

1. Formalisez la « situation type ».
2. Écrivez une base de clauses Prolog qui contient un état initial, un état final, et les transitions nécessaires à la planification de ce problème.
3. Lancez le prédicat de planification de l'exercice n°1 sur votre base de clauses.

Représentation et Modélisation des connaissances
Prolog - TP3 & TP4

Objectif:

- Jeux à deux joueurs en Prolog

Contrôle continu :

Les TP 3 et 4 feront l'objet d'évaluation. Pour cela:

- Vous pouvez travailler en binôme;
- Indiquez vos 2 noms en haut de chacun de vos fichiers;
- Commentez votre code en expliquant chacune de vos requêtes ainsi que les conditions d'arrêt. Indiquez en commentaire un exemple d'utilisation de chacune de vos requêtes.
- Envoyez **une archive nom1-nom2.zip** regroupant vos fichiers à la **fin du TP** @ : youssef.serrestou@univ-lemans.fr

---

Etapes dans la réalisation d'un joueur artificiel :

1. définir la représentation du jeu ;
2. gérer la « visualisation » du jeu ;
3. gérer les coups qu'un joueur peut jouer ;
4. créer le moteur qui va permettre de lancer le jeu et de faire jouer un joueur contre un joueur artificiel ;
5. **amélioration du « joueur » artificiel : min-max , alpha-beta,…**

Etape 1 à 4 :

Ces premières étapes consistent à programmer les différents prédicats vus en TD3 et TD4. Vous pouvez récupérer le programme écrit en ces TD sous tictactoe.pro. Testez ce programme et modifier et/ou compléter le en cas de besoin. Si vous aviez écrit des prédicats différemment, testez vos prédicats.

Etape 5 : Amélioration du joueur artificiel

Jusqu'à présent, le joueur artificiel joue le premier coup qu'il trouve dans la liste des coups. Vous devez rajouter des stratégies de jeu au joueur artificiel afin qu'il soit plus « intelligent ». Pour cela vous devrez créer un arbre de recherche à une certaine profondeur et implémenter une fonction d'évaluation. Vous pourrez ensuite implémenter l'algorithme *MiniMax* pour aider le joueur artificiel dans le choix de ces coups. Enfin la dernière étape pourra être d'optimiser les choix de votre joueur en implémentant l'algorithme alpha-beta.

Pour implémenter l'algorithme *MiniMax*, vous pourrez avoir besoin de:

- Déterminer les prédicats nécessaires pour écrire une **fonction d'évaluation** (en choisir une simple pour commencer). En particulier, vous devrez définir le prédicat **evaluePosition(+Grille,+Camp, ?Valeur)** qui permet d'associer une valeur à une grille, ce prédicat est satisfait si Valeur est le résultat de la fonction d'évaluation du programme pour la Grille, pour le joueur Camp.
- Déterminer un prédicat qui permet de trouver le meilleur coup parmi une liste de coups jouables pour un camp donné (utile pour le Max) ;
- De même, déterminer un prédicat qui permet de trouver le plus mauvais coup parmi une liste de coups (utile pour le Min) ;

- Déterminer un prédicat (et les prédicats nécessaires pour conduire à ce prédicat) **evalueMinMax(+Grille, +Camp, +Profondeur, ?Case, ?Valeur)** qui est satisfait si Case correspond aux coordonnées de la case de la Grille où Camp peut jouer ; Case est trouvée en utilisant l'algorithme minimax à la profondeur maximale Profondeur ; Valeur est l'évaluation de la position.
- Modifier votre programme initial afin d'intégrer l'algorithme *MiniMax* dans la gestion des coups du joueur artificiel.
- Proposez une autre fonction d'évaluation pour l'algorithme *MiniMax*;
- Vous pouvez augmenter la taille de la grille de jeu (et donc modifier le programme en conséquence) afin de mieux tester l'évolution de votre joueur artificiel. Testez cet algorithme en augmentant la taille de la grille de jeu. Pouvez-vous faire varier la profondeur de calcul de votre algorithme afin d'avoir un jeu toujours gagnant pour l'ordinateur ?

Quelques rappels (adaptés aux Jeux à deux joueurs):

Un arbre de jeu :

- un arbre de jeu est un graphe d'états dont
  - les sommets correspondent aux différentes configurations du jeu,
  - les arcs correspondent aux coups joués (i.e. aux transitions)
  - et les positions terminales correspondent à « gagné », « perdu » ou « match nul ».
- un arbre de jeu permet de décrire toutes les positions possibles.

Un arbre de recherche :

Pour limiter la taille de l'arbre de jeu, on génère l'arbre de recherche à partir d'une position donnée afin de trouver les prochains coups à jouer (i.e. on construit dynamiquement un sous-arbre de jeu). Pour restreindre les alternatives à considérer, on utilise des stratégies qui vont permettre de choisir le développement de tel ou tel nœud. Parmi ces stratégies, on trouve l'algorithme *MiniMax* et *Alpha-Beta.*

Le principe de l'algorithme *MiniMax* :

Cet algorithme permet de choisir le coup le plus avantageux à jouer au niveau de la racine de l'arbre (i.e. du coup à jouer) selon une stratégie qui essaie de **maximiser** les chances de succès du joueur, tout en **minimisant** les chances de succès de son adversaire. (On suppose que les deux joueurs utilisent la même stratégie i.e. la même fonction d'évaluation pour évaluer leurs positions). L'algorithme est donc le suivant :

- MiniMax(p)=f(p) si p est une position terminale et f une fonction d'évaluation ;
- MiniMax(p)=max(MiniMax(O_1),…, MiniMax(O_n)) si p est une position joueur Ami avec O_1, … , O_n comme fils ;
- MiniMax(p)=min(MiniMax(J_1),…, MiniMax(J_n)) si p est une position Ennemi avec J_1, … , J_n comme fils.

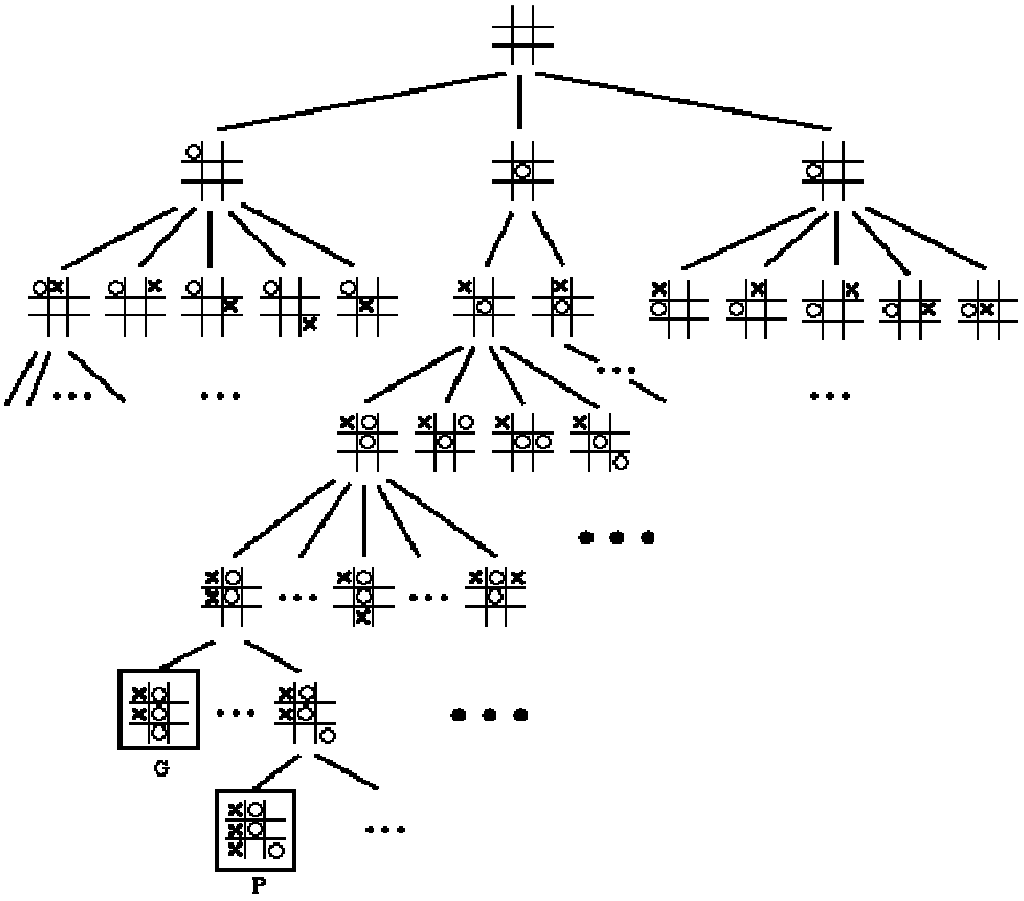


Figure 1 : Exemple d'un arbre de recherche pour le tic-tac-toe

Exemples de fonctions d'évaluation pour le Tic-tac-toe :

Les fonctions d'évaluation permettent d'évaluer au mieux une position (non terminale) et de lui attribuer une valeur afin de pouvoir déterminer le meilleur coup parmi une liste de coups possibles.

Solution 1 :

Si le joueur de référence est 'x', on peut déterminer la fonction d'évaluation V telle que V= V1-V2, où V1 est la somme :

- du nombre d'occurrences de 3 'x' sur une ligne/colonne/diagonale multiplié par 10
- du nombre d'occurrences de 2 'x' sur une ligne/colonne/diagonale avec une 3ième position vide multiplié par 4
- du nombre d'occurrences de 1 'x' sur une ligne/colonne/diagonale avec deux autres positions vides

Et V2 la même somme pour 'o'.

Solution 2 :

Si le joueur de référence est 'x', on peut déterminer la fonction d'évaluation V telle que V= V1-V2, où V1 est le nombre de ligne/colonne/diagonale encore ouverte pour 'x' , et V2 est le nombre de ligne/colonne/diagonale encore ouverte pour 'o'.

Solution 3 :

Reprendre les solutions précédentes en ajoutant la reconnaissance des positions gagnantes et perdantes : V = +∞ si 'x' est gagnant et V = -∞ si 'o' est gagnant.

Exemples :

	o	x
	o	x
o		x

o		x
	o	x
	x	o

o	o	x
x	o	
x		

Solution 1 : V=5 V=-11 V=-6

Solution 2 : V=-1 V=-2 V=0

Solution 3 : V=+∞ V=-∞ V=-6