Université de Franche-Comté
Département Informatique

Master 1ère année
Module MOIA

Projet de jeu Ultimate Tic-Tac-Toe
M1 MOIA

BINOME		
Nom	**Prénom**	**Trigramme**
LOTOY BENDENGE	Vianney	LBV
OUSLEYEH BILEH	Yassin	OBY

Année Universitaire 2015-2016

Sommaire :

I- Introduction :

Ultimate Tic-Tac-Toe est un jeu qui se compose d'un Plateau 3x3, dont à l'intérieur de chaque case du plateau se trouve des sous-plateaux de 3x3 , sur lequel à tour de rôle, 2 joueurs marquent une case (ou cellule), respectivement par 'X' et 'O'.
Le premier joueur à avoir aligné 3 mini sous-plateaux gagné horizontalement, verticalement ou diagonalement dans le plateau, est le gagnant. Dans le cas où tous les sous-plateaux sont soit gagnés ou plein, plus aucun sous-plateau n'est libre et aucun des joueurs n'a gagné, la partie est un match nul. Les Sous-Plateaux déjà gagnés ou pleins sont interdits d'y jouer.

Le premier joueur peut inscrire son symbole (X) dans n'importe quelle case de n'importe quel sous-plateau. Par contre les coups suivant se joueront dans des sous-plateaux particuliers.
C'est à dire, la case jouée par le coup précédent conditionne le sous-plateau sur lequel le coup suivant sera jouée.
Exemple :
X joue [Sous-Plateau A, Case 2], le O doit joué dans le sous-plateau B.
et si O joue [Sous-Plateau B , Case 5]. Alors le A va joué à son tour dans le sous-plateau E.

Si jamais un joueur est envoyé dans un sous-plateau déjà gagné ou plein alors, il aura le choix de choisir un sous-plateau non gagné et non plein.
Par convention le joueur 'X' débute la partie, et les joueurs seront identifiés par leur seul caractère.
L'objectif du projet est :
- La Réalisation d'une IA en Sicstus Prolog jouant au "Ultimate Tic-tac- toe".
- La Réalisation d'une API en Java pour invoquer l'IA Prolog et pour communiquer avec la partie codé en C.

Figure *(Un extrait de l'exécution du jeu sur un Terminal)*

II-Réalisation de la IA en Prolog.

1- Les Prédicats de base.

- changeSigneCase/4, prend en paramètre une case, un sous-plateau, un signe et un nouveau sous-plateau. Il modifie le sous-plateau donné en plaçant le signe donné dans la case donné et retourne le nouveau sous-plateau.

```
{
changeSigneCase(1, [_|SousPlateau], Signe, [Signe|SousPlateau]).
changeSigneCase(N, [X|SousPlateau], Signe, [X|NSousPlateau]) :-   N2 is N – 1,
                changeSigneCase(N2, SousPlateau, Signe, NSousPlateau).
            }
```

- signeCaseSousPlateau/3, ce prédicat prend en paramètre une case, un sous-plateau et un signe. Et permet de vérifier si le signe donné se trouve dans la case donnée du sous-plateau donné.

```
{
                signeCaseSousPlateau(N, SousPlateau, Signe) :-
                 nth1(N, SousPlateau, Signe).
            }
```

- sousplateauGagnant/2, vérifie si le signe passé en paramètre est présent 3x horizontalement, verticalement ou diagonalement dans les sous-plateau donné.

```
{
sousplateauGagnant([Signe,Signe,Signe|_],Signe):-!.
               sousplateauGagnant([_,_,_,Signe,Signe,Signe|_],Signe):-!.
               sousplateauGagnant([_,_,_,_,_,_,Signe,Signe,Signe|_],Signe):-!.
               sousplateauGagnant([Signe,_,_,_,Signe,_,_,_,Signe],Signe):-!.
               sousplateauGagnant([_,_,Signe,_,Signe,_,Signe,_,_],Signe):-!.
               sousplateauGagnant([Signe,_,_,Signe,_,_,Signe,_,_],Signe):-!.
               sousplateauGagnant([_,Signe,_,_,Signe,_,_,Signe,_],Signe):-!.
               sousplateauGagnant([_,_,Signe,_,_,Signe,_,_,Signe],Signe).
            }
```

- typeCoup/6, détermine le type du coup joué et retourne 'd' pour partie CONTINUE, 'g' GAGNANT, 'n' NULLE et 'p' PERDANT.

{

```
typeCoup(Plateau, _Signe, SigneAdv, _Tour, _TourMax,  p) :-
      sousplateauGagnant(Plateau, SigneAdv),
      !.
typeCoup(Plateau, Signe, _SigneAdv, _Tour, _TourMax, g) :-
       sousplateauGagnant(Plateau, Signe),
      !.
typeCoup(_Plateau, _Signe, _SigneAdv, TourMax, TourMax, n)  :-
      !.
typeCoup(_Plateau, _Signe, _SigneAdv, _Tour, _TourMax, d).
```

}

- genereCoup/9, génère un coup à partir de la case, du sous-plateau et du signe et retourne le type du coup joué et du nouveau sous-plateau.

{

```
genereCoup(N,Plateau,Signe,NotreSigne,SigneAdv,Tour,TourMax,TypeCoup,NPlateau):-
                  signeCaseSousPlateau(N, Plateau, v),
                  changeSigneCase(N,Plateau,Signe,NPlateau),
                  signeCaseSousPlateau(N, NPlateau, Signe),
                  typeCoup(NPlateau,NotreSigne,SigneAdv,Tour,TourMax,
                  TypeCoup).
```

}

- plateauxSuivant/7, Renvoie tous les coups possibles à partir d'un sous-plateau dans une liste.

{

```
plateauxSuivants(Plateau, SigneCoup, SigneJoueur, SigneAdv,
      Tour, TourMax, R) :-
      findall([NPlateau, Case, TypeCoup], genereCoup(Case,
      Plateau, SigneCoup, SigneJoueur, SigneAdv, Tour,
      TourMax, TypeCoup, NPlateau), R).
```

}

2- Fonction d'évaluation .

Définir un prédicat evalue/5 qui rend une évaluation du plateau de jeu pour le joueur J.
On se limitera à l'heuristique vue en cours : on compte le nombre d'alignements de 2 marques complétables à 3 par le joueur J s'il jouait maintenant. On retire à ce nombre celui des mêmes alignements complétables par le joueur adverse s'il jouait maintenant. Cette fonction d'évaluation est donc statique : elle ne dépend que de la configuration actuelle du plateau et pas des décisions futures des joueurs. On notera également qu'on pourra décomposer dans les différents types d'alignements (verticaux, ...), comme pour le prédicat sousplateauGagnant/2.

- creerListeSigne/3, crée une liste de 3 éléments contenant N fois le signe donné.

{

```
creerListeSigne(1, Signe, [Signe,_,_]).
creerListeSigne(2, Signe, [Signe,Signe,_]).
creerListeSigne(3, Signe, [Signe,Signe,Signe]).
```

}

- nbSigneLigne/5, vérifie si le signe passé en paramètre est présent N fois dans les lignes depuis Dep dans le sous-plateau donné.

{

```
nbSigneLigne(SousPlateau, Dep, Signe, N, 1) :-
        Dep2 is Dep + 1,
        Dep3 is Dep + 2,
        nth1(Dep, SousPlateau, C1),
        nth1(Dep2, SousPlateau, C2),
        nth1(Dep3, SousPlateau, C3),
        creerListeSigne(N, Signe, L),
        permutation(L, [C1,C2,C3]),
        !.

nbSigneLigne(_, _, _, _, 0).
```

}

- nbSigneCol/5, vérifie si le signe passé en paramètre est présent N fois dans les colonnes depuis Dep dans le sous-plateau donné.

{

```
nbSigneCol(Plateau, Dep, Signe, N, 1) :-
       Dep2 is Dep + 3,
       Dep3 is Dep + 6,
       nth1(Dep, Plateau, C1),
       nth1(Dep2, Plateau, C2),
       nth1(Dep3, Plateau, C3),
```

```
            creerListeSigne(N, Signe, L),
            permutation(L, [C1,C2,C3]),
            !.
      nbSigneCol(_, _, _, _, 0).
}
```

- nbSigneDiag1/4, vérifie si le signe passé en paramètre est présent N fois dans la diagonale depuis la case 1 dans le sous-plateau donné.

{

```
      nbSigneDiag1(Plateau, Signe, N, 1) :-
        Dep is 1,
        Dep2 is Dep + 4,
        Dep3 is Dep + 8,
       nth1(Dep, Plateau, C1),
       nth1(Dep2, Plateau, C2),
       nth1(Dep3, Plateau, C3),
       creerListeSigne(N, Signe, L),
       permutation(L, [C1,C2,C3]),
       !.
      nbSigneDiag1(_, _, _, 0).
}
```

- nbSigneDiag2/4, vérifie si le signe passé en paramètre est présent N fois dans la diagonale depuis la case 3 dans le sous-plateau donné.

{

```
      nbSigneDiag2(Plateau, Signe, N, 1) :-
            Dep is 3,
            Dep2 is Dep + 2,
            Dep3 is Dep + 4,
            nth1(Dep, Plateau, C1),
            nth1(Dep2, Plateau, C2),
            nth1(Dep3, Plateau, C3),
            creerListeSigne(N, Signe, L),
            permutation(L, [C1,C2,C3]),
            !.
      nbSigneDiag2(_, _, _, 0).
}
```

- nb1Signe/3, Vrai si le Plateau contient R lignes/colonnes/diagonales comportant 1 fois ce Signe

- nb2Signe/3, Vrai si le Plateau contient R lignes/colonnes/diagonales comportant 2 fois ce Signe

- evalue/5, 1000 si plateau gagnant, -1000 si perdant, 0 si match nul, *sinon f(Plateau) = 2 * (NB2 - NB2Adv) + 1 * (NB1 - NB1Adv)*

{

```
evalue([_,_,p], _SigneJoueur, _SigneAdv, Prof, Eval) :-
       Eval is -1000 + Prof,
       !.
evalue([_,_,g], _SigneJoueur, _SigneAdv, Prof, Eval) :-
       Eval is 1000 - Prof,
       !.
evalue([_,_,n], _SigneJoueur, _SigneAdv, _Prof, 0) :- !.
evalue([Plateau,_,_], SigneJoueur, SigneAdv, _Prof, Eval) :-
       nb2Signe(Plateau, SigneJoueur, NB2),
       nb2Signe(Plateau,SigneAdv, NB2Adv),
       nb1Signe(Plateau, SigneJoueur, NB1),
       nb1Signe(Plateau,SigneAdv, NB1Adv),
       Eval is  2 * (NB2 - NB2Adv) + 1 * (NB1 - NB1Adv).
```

}

3- Minimax & Alpha-Beta.

Définir un fait sous la forme d'une variable **Prof** qui permettra à minimax de tester s'il est arrivé à la profondeur maximum.

Implantation du prédicat
`minEvalPlateaux (P, SJ, SA, Tour, TourMax, Prof, ProfParcourue, Alpha, Beta, Acc, Acc) :`

◦ P est le sous-plateaux actuel.
◦ SJ est le signe de notre joueur.
◦ SA est le signe de l'adversaire.
◦ Tour est le nombre de coup jouer.
◦ TourMax est le nombre de coup maximum possible de jouer initialiser à 81.
◦ Prof est la profondeur du noeud de l'arbre où on appelle récursivement minimax .
◦ ProfParcourue est la profondeur actuel à chaque coup joué.
◦ Alpha est une variable initialiser à -1000.
◦ Beta est une variable initialiser à 1000.
◦ Acc sont des accumulateur.

Implantation du prédicat
`alphaBeta(P, SJ, SA, Tour, TourMax, Prof, MeilleurCoup) :`

◦ P est le sous-plateaux actuel.
◦ SJ est le signe de notre joueur.
◦ SA est le signe de l'adversaire.
◦ Tour est le nombre de coup jouer.
◦ TourMax est le nombre de coup maximum possible de jouer initialiser à 81.
◦ Prof est la profondeur du noeud de l'arbre où on appelle récursivement minimax .
◦ MeilleurCoup est le coup préconisé et son évaluation minimax.

• On notera qu'on appelle toujours minimax pour jouer un coup. Grosso-modo pour trouver le meilleur coup à jouer à un stade donné d'un sous-plateau.
Soit pour ne pas perdre un sous-plateau ou pour gagner un sous-plateau.

• Et de la même manière on utilise minimax pour trouver le meilleur sous-plateau à jouer dans le cas où notre adversaire nous envoie dans un sous-plateau gagné, perdu ou plein, pour cela nous avions juste développé une classe java qui prend tous les états des sous-plateaux.
C'est à dire si on a un sous-plateaux est gagné, on le marque avec notre signe et vice versa pour notre adversaire.
Et on envoie la liste des états des sous-plateaux à notre IA et ainsi elle nous indique quel sous plateau faudra joué. Soit pour gagner tout le plateau ou soit pour éviter de perdre tout le plateau.

• Pour déterminer le coup suivant de l'IA, on appelle minimax avec en paramètre la `ProfParcourue` 1 (on est à la racine de l'arbre). Le symbole global Prof doit être supérieur ou égal à 1 : s'il était égal à 0 alors minimax s'arrêterait tout de suite en donnant l'évaluation du plateau courant et un coup vide, ce qui ne permet pas de jouer le coup suivant.

4- Lancement de la partie

Le prédicat de lancement est :

`lancement(Plateau,SigneSousPlateau, SigneJoueur, SigneAdv, Tour, TourMax, Case, TypeCoup, NPlateau):`

- Plateau est le plateau du jeu.
- SigneSousPlateau est le signe du sous-plateau (A ou B ou C ou D …..).
- SigneJoueur est le signe de notre joueur.
- SigneAdv est le signe de l'adversaire.
- Tour est le nombre de tour actuellement joué.
- TourMax est le nombre de tour maximum possible (81).
- Case est la case joué par la IA.
- TypeCoup est le type du coup (Perdu, Gagné, Continue, Nulle) retourné par la IA.
- NPlateau est le nouveau plateau retourné par la IA.

Il est à noter que la IA joue le coup en se basant que sur le sous-plateau sur lequel on doit jouer.
Le sous-plateau sur lequel, elle doit jouer son coup lui est indiqué par SigneSousPlateau passé en paramètre. Et c'est dans la partie java que SigneSousPlateau est géré.

La communication entre la IA (Prolog-Sicstus) et l'API java est possible par le biais de la librairie jasper.
Après la création de la requête (de type String) correspondant exactement au prédicat de lancement de la IA `lancement/9`.

Et pour finir :

- On lance la requête, avec un objet (de type SPQuery).
- On récupère, le résultat de la IA dans un objet de type (HashMap).

```
Map<String, SPTerm> resultQuery = new HashMap<String, SPTerm>();
String req = "lancement("+plateau.getStringSPTerm()+","+numplateau.getStringSPTerm()+",
       "+symbol.getStringSPTerm()+", "+symbolAdverse.getStringSPTerm()+", "+numeroTour+",
       "+nbToursMax+", Case, TypeCoup, NPlateau).";
SPQuery query = this.sp.openQuery(req, resultQuery);
```

II-Conclusion

Au cours de la réalisation de ce projet, nous avons eu l'occasion de mettre en pratique ce que nous avons pu apprendre en cours, telles que la programmation récursive en prolog, l'implémentation d'une heuristique, etc. Ce projet nous a notamment permis de voir la complexité des algorithmes efficaces dans la résolution des problèmes dans les jeux.
De plus, le jeu Ultimate Tic-Tac-Toe rajoute de la difficulté par rapport à gestion du plateau pour la mise en place d'un algorithme capable d'être efficace tout en étant rapide, en prenant en compte les règles et contraintes de ce jeu, ce qui n'a pas été évident à réaliser.