Introduction :

Ce projet a pour but de créer trois intelligences artificielles capable de jouer au jeu Ultimate Tic-Tac-Toe, l'implémentation s'appuie sur le papier :

Sylvain Gelly (Université Paris Sud, LRI, CNRS, INRIA), David Silver (University College London) "Monte-Carlo Tree Search and Rapid Action Value Estimation in Computer Go" March 2011.

Je crée trois IA différentes utilisant trois stratégies pour jouer.

La première utilisera UCT (Upper confidence Bounds for Trees)
La seconde utilisera RAVE (Rapid Action Value Estimation)
Et la troisième sera composé d'un mixte des deux premières stratégies.

Je fais jouer ces trois IA face à une stratégie random et entre elles. Après simulation de 1000 parties du jeu je récapitule le taux de gain de chaque méthode.

I) Jeu utilisé : Ultimate Tic-Tac-Toe

Ultimate tic-tac-toe est un jeu de société de neuf tic-tac-toe disposées dans une grille 3 × 3.

- Le premier joueur choisi un board et une case comme il lui convient.
- Cette case va alors déterminer sur quel board le joueur 2 va pouvoir jouer : sur la figure 1, le joueur 1 a joué sur le board central à la case en haut à droite, donc le joueur suivant va devoir jouer dans le board en haut à droite et ainsi de suite.
- Pour gagner, il faut gagner 3 boards et que ces 3 boards soient alignés. Un exemple se trouve dans la figure 2.

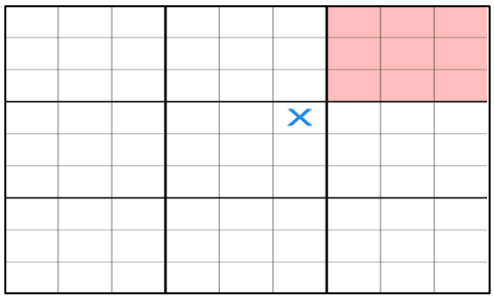

Figure 1

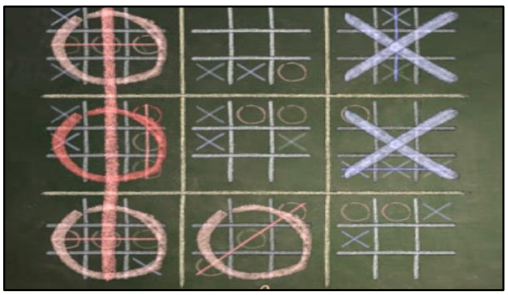

Figure 2

II) Expériences de recherche Monte Carlo sur Ultimate Tic-Tac-Toe

a) Structure du code

- **Une classe Game** qui donne le board, les actions, les jeux gagnants, comment faire une action et obtenir les résultats
- **Une classe Node** qui construits les noeuds. Chaque noeud a des actions possibles, des noeuds fils, un nœud parent, ses statistiques de gains et de visites. On sélectionne ce noeud avec différentes méthodes *SelectChild(),* on lui ajoute des fils *AddChild(), o*n update ces statistiques de gains et de visites *Update().* Le reste des fonctions sont des fonctions de représentation.

• **Une fonction en lien avec la méthode utilisée** qui se déroule en quatre temps : Selection Expansion, RollOut, Propagation rétroactive.
• Un algorithme permettant de **faire jouer deux joueurs**.

b) UCB

La classe NodeUCB utilise **la méthode UCT** :
Sélectionne l'action qui maximise la formule UCB à partir d'un état *UCTSelectChild()* :

```
s = sorted(self.childNodes, key=lambda c: c.wins / c.visits + sqrt(b *
log(self.visits) / c.visits))[-1]
```

c) RAVE

La classe NodeRave utilise **la méthode Rave :**
Sélectionne l'action qui maximise la valeur de l'état avec la formue RAVE *RaveSelectChild()* :

Calcul du beta: `self.beta = self.visitsAMAF/ (self.visits + self.visitsAMAF + 1e-5 * self.visits * self.visitsAMAF)`
Calcul de la meilleure action:

```
dict1[c]=((1-c.beta)*(c.wins / c.visits)+c.beta*(c.winsAMAF / c.visitsAMAF))
s=list({k: v for k, v in sorted(dict1.items(), key=lambda item: item[1])})[-1]
```

La fonction *Update()* doit mettre à jour les statistiques AMAF, ie les gains d'une action et ses visites qui est contenu dans tous les playouts simulés.

```
if self.move in l:
        self.winsAMAF += result
        self.visitsAMAF += 1
```

Enfin dans la **fonction Rave**, il faut créer lors du **RollOut** une liste permettant de **récupérer toutes les actions joués lors d'un playout** afin de pouvoir updater les statistiques AMAF.

```
l=[]
while state.GetMoves() != []: # while state is non-terminal
        x=random.choice(state.GetMoves())
        state.DoMove(x)
       if x not in l: l.append(x)
```

d) Un mélange de UCB et RAVE : UCB-RAVE

La classe Node_UCB_Rave ajoute **un bonus exploration à la formule Rave** pour sélectionner l'action à faire à partir d'un état *UCBRaveSelectChild()* :

```
dict1[c]=(1-c.beta)*(c.wins / c.visits)+c.beta*(c.winsAMAF / c.visitsAMAF)+ sqrt(b
* log(self.visits) / c.visits)
```

```
s=list({k: v for k, v in sorted(dict1.items(), key=lambda item: item[1])})[-1]
```

Le reste est identique à la méthode Rave.

II) Résultats sur 1000 parties

	RAVE	UCB-RAVE c=sqrt(2)	UCB-RAVE c=1	Random
UCB c=sqrt(2)	799/201	724/276	607/393	994/6
UCB c=1	875/125	801/199		998/2
RAVE		499/501	391/609	979/21
UCB-RAVE c=sqrt(2)				982/18
UCB-RAVE c=1				989/11
Random				

- On remarque tout d'abord que les stratégies UCB (c=sqrt(2)/c=1), RAVE et UCB-RAVE (c=sqrt(2)/c=1) gagnent presque tout le temps face au random player (994/6, 998/2, 979/21, 982/18) ce qui est normal puisque le deuxième joueur n'a aucune stratégie, il joue au hasard.

- On remarque que le joueur UCB gagne 80% du temps contre RAVE. On pourrait s'attendre à un résultat contraire mais nous sommes dans un jeu simple avec un arbre peu profond, c'est une situation très favorable à UCB. En effet, UCB peut être moins performant lorsque l'arbre est profond s'il prend un mauvais chemin il peut être difficile de revenir à des états du débuts grâce au terme d'exploration. Ici, le jeu est trop simple pour bénéficier des avantages qu'offre la méthode RAVE.

- On remarque que UCB reste plus performant que UCB-RAVE mais gagne moins souvent que contre RAVE. De plus UCB-RAVE gagne contre RAVE. Ainsi, UCB-RAVE est une amélioration de RAVE pour ce jeu.

- On remarque que lorsque la constance *c* est plus petite donc qu'on explore moins, UCB(c=1) et UCBRAVE(c=1) obtiennent de meilleurs résultats. Ceci est logique puisque le jeu est simple, explorer des états fait perdre du temps au joueur. D'ailleurs lorsque UCB (c=sqrt(2)) joue contre UCB-RAVE (c=1), il gagne toujours mais moins souvent 60% contre 75% du temps.