Université de Lille

Travail Encadré de Recherche
-
création d'une intelligence artificielle pour le jeu othello avec la recherche arborescente de Monte Carlo

Encadrant : M WICKER

Thibaud BARGIBANT & Pierre JESUSPRET

Sommaire

1. Histoire et règle de l'Othello 3
2. Recherche arborescente de Monte Carlo 6
   2.1 L'arbre N-aire 6
   2.2 Algorithme de Monte Carlo Tree Search 7
   2.3 Exemple 9
3. Aspect théorique et expérimental de l'algorithme MCTS 11
   3.1 Positionnement du problème 11
   3.2 Théorème et preuve 12
   3.3 Expérimentation 16
4. Construction de l'intelligence artificielle du jeu 19
   4.1 Qu'est-ce qu'une intelligence artificielle ? 19
   4.2 Construction informatique du jeu et de l'algorithme de recherche arborescente 19
      4.2.1 Outils de développement 19
      4.2.2 Modules, programmes et fonctions développées 21
5. Conclusion 22
Annexes 23
   Bibliographie 23
   Code informatique 23
      Évaluation des coups 23
      Classe Noeud 26

1. Histoire et règle de l’Othello

L’othello est un jeu de stratégie qui se joue à 2 sur un tablier unicolore de 8 cases sur 8 cases. Le jeu a été inventé vers 1880 en Angleterre par Lewis Waterman. Chaque joueur joue l’un après l’autre, chacun possède une couleur, noir ou blanc.

Au début de la partie il y a 4 pions sur le plateau, deux blancs et deux noirs, les noirs commencent toujours. Ils sont disposés comme sur la figure ci-dessous :

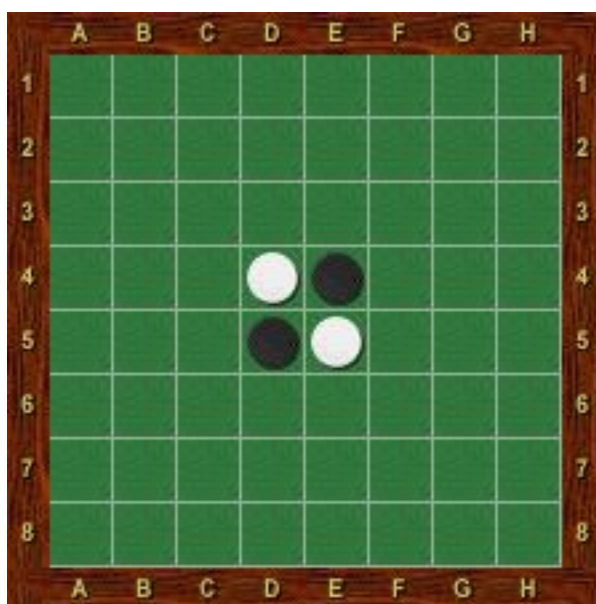


Figure 1 - Configuration initiale du jeu

Le joueur noir doit, pour pouvoir jouer, encadrer au moins un pion blanc entre un pion noir déjà sur le plateau et le nouveau pion qu’il place à son tour de jeu. Il est possible d’encadrer les pions en lignes, en colonnes ou en diagonales. Voici un exemple :

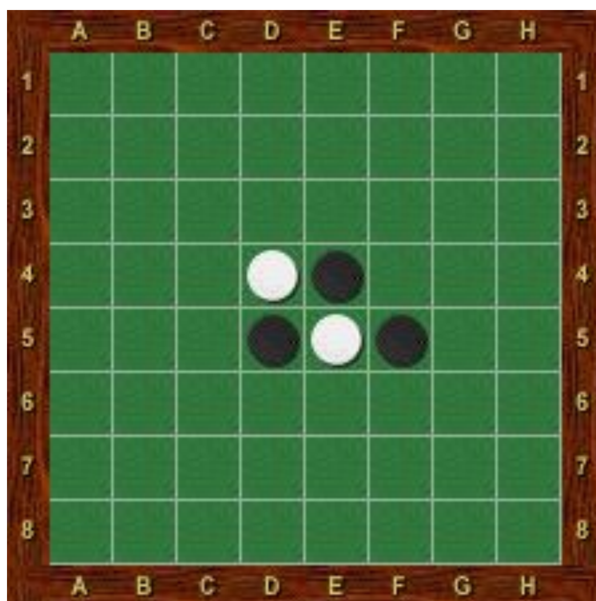


Figure 2 - Premier pion noir placé

Les pions blancs encadrés par les deux pions noirs deviennent des pions noirs, ce qui donne :

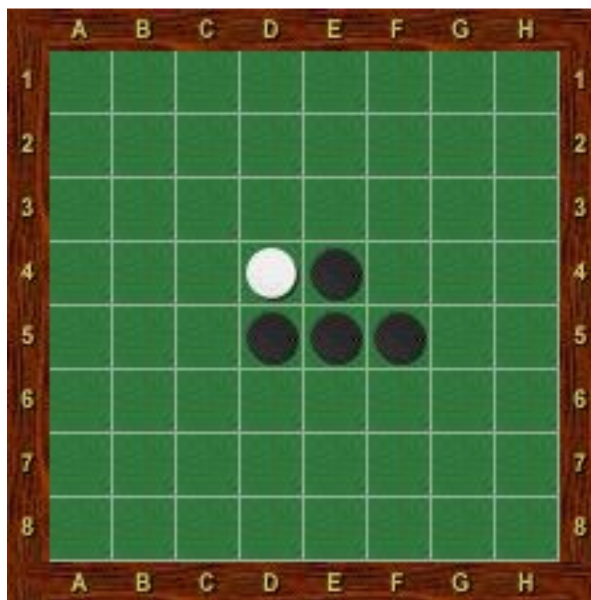


Figure 3 - Retournement du pion blanc en E5

C’est maintenant au tour des blancs de jouer, il suit le même procédé que les noirs et ainsi il peut jouer les différents coups suivants :

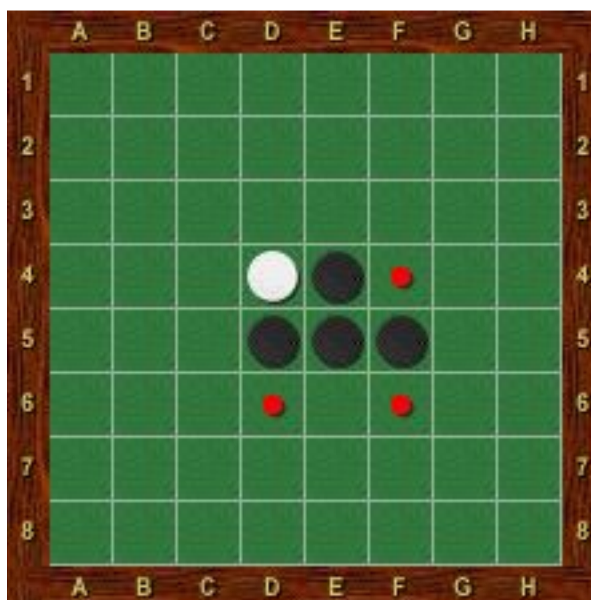


Figure 4 - Coups possibles pour les blancs

La partie se déroule suivant ce schéma. Il peut arriver qu’un joueur ne puisse pas placer de pion. Dans ce cas, l’autre couleur ayant joué au tour précédent continue son tour en plaçant un deuxième pion s’il en a la possibilité. En effet, il se peut que les deux joueurs ne puissent plus jouer (figure 5). Dans ces conditions, le jeu s’arrête même si le plateau n’est pas totalement occupé.

Figure 5 - Fin de partie

Sur la figure 5, aucune des deux couleurs ne peut jouer. Ainsi la partie prend fin en comptant le nombre de pions de chaque couleur et celle qui en a le plus sur le plateau a gagné.

2. Recherche arborescente de Monte Carlo

2.1 L'arbre N-aire

L'intelligence artificielle du jeu est basée sur la construction d'un arbre N-aire. On peut représenter le jeu par un arbre des possibles. Donnons un exemple d'arbre partiel correspondant à l'issue de deux coups joués :

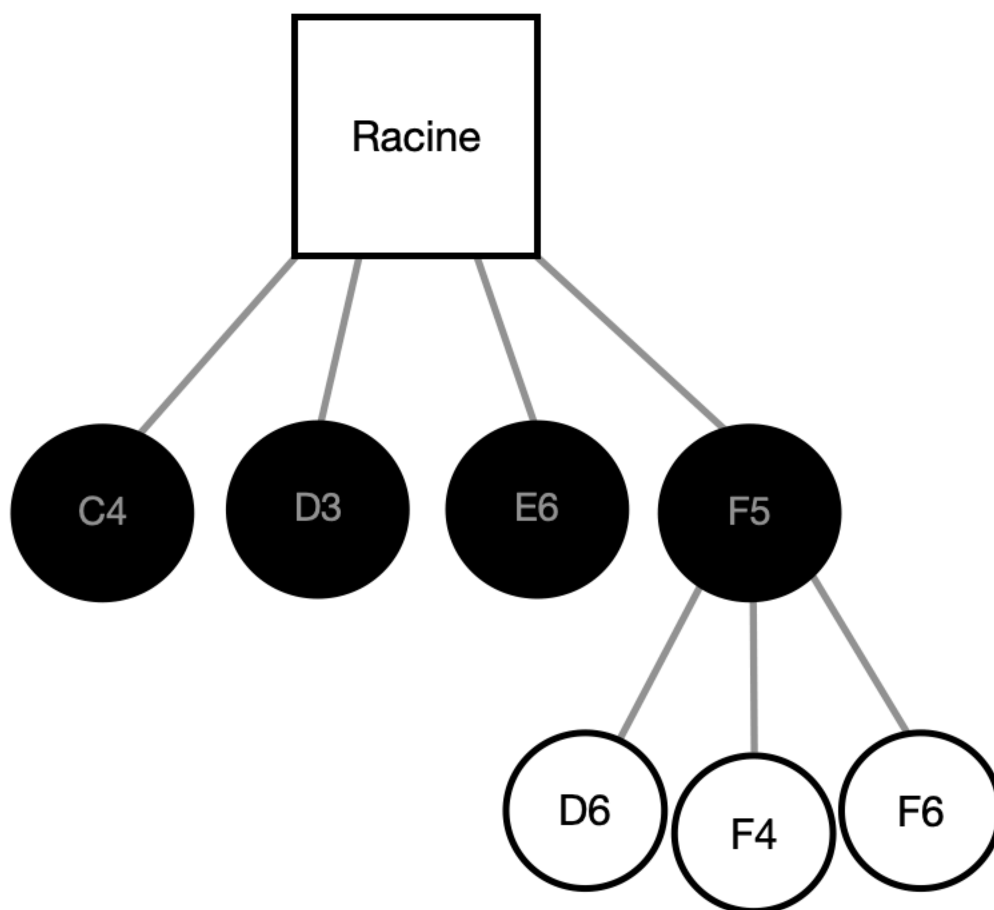


Figure 6 - Début de l'arbre de jeu

L'arbre est composé d'une racine, de noeuds, de feuilles et de branches. La racine correspond à la configuration initiale du jeu. Un noeud correspond dans notre cas à un coup possible ayant un père et ayant au moins un fils. Une feuille est un noeud qui n'a pas de fils. Enfin, une branche correspond à la série de noeuds présents sur le « chemin » entre une feuille et la racine. Dans notre exemple, seul F5 est un noeud. Les autres sont toutes des feuilles, elles ne possèdent pas de fils, et {racine, F5, F6} est une branche.

Sur l'exemple de la figure 6, on démarre de la racine qui correspond, comme précisé antérieurement, aux quatre pions placés à l'initial. Les noirs commencent la partie et ont la possibilité de choisir l'une des cases parmi C4, D3, E6 ou F5. Ils choisissent la case F5. C'est maintenant aux blancs de jouer. Ils peuvent choisir pour jouer l'une des cases suivantes : D6, F4 ou F6 et ainsi de suite. Ici, les noeuds, représentés par les disques pleins noirs correspondent aux coups potentiels de la couleur noire et ceux représentés par les disques blancs sont les coups possibles de la couleur blanche.

2.2 Algorithme de Monte Carlo Tree Search

L'algorithme Monte Carlo Tree Search (MCTS), ou l'algorithme UCB1, est un algorithme d'apprentissage par renforcement qui consiste en une recherche heuristique pour permettre une certaine prise de décision. C'est un procédé qui cherche à faire une balance entre l'exploration, c'est à dire trouver des actions potentiellement profitables et l'exploitation, qui consiste à choisir la meilleure action empirique le plus souvent possible. Ceci dans le but de prendre empiriquement la meilleure action aussi souvent que possible. Cette méthode a pour support l'arbre présenté dans la section précédente. Dans cet algorithme chaque noeud (ou feuille) et la racine possèdent certaines quantités exploitées par l'algorithme qui interviennent dans la formule suivante : $UCB1(N,c) = \frac{\omega}{n} + c\sqrt{\frac{\ln(N)}{n}}$ avec N le nombre de parties totales simulées. Ces quantités sont :

- n_i : le nombre de fois où le noeud i a été visité
- ω_i : le nombre de parties remportées par la couleur du noeud i
- la valeur de UCB1

L'algorithme MCTS est composé de 4 étapes :

- Sélection
- Expansion
- Simulation
- Backpropagation

La **sélection** consiste à sélectionner une série de noeuds depuis la racine jusqu'à atteindre une feuille. Cette sélection s'effectue en gardant un compromis entre le choix d'un noeud qui a été prouvé comme prometteur (ou exploitation qui est représentée par $\frac{\omega}{n}$ dans la formule) et le choix d'un nouveau noeud qui paraît prometteur (ou exploration correspondant à $\sqrt{\left(\frac{\ln(N)}{n}\right)}$).

Lors de l'**expansion**, on parcourt en respectant la sélection jusqu'à atteindre une feuille. Si cette feuille n'est pas finale (si elle ne correspond pas à une configuration finale de jeu) alors en suivant les règles du jeu on ajoute autant de noeuds fils à cette feuille que de coups possibles.

La **simulation** consiste à terminer la partie en partant de la configuration donnée par la feuille.

Enfin la **back propagation** modifie les données des noeuds en remontant la branche concernée. Ce procédé consiste à incrémenter de 1 le nombre de visites des noeuds sur la branche et à incrémenter de 1 la donnée ω qui correspond au nombre de parties remportées par la couleur du noeud (ω_{racine} est incrémenté de 1 dans tous les cas).
La difficulté principale est la phase de sélection. Pour construire l'arbre, il faut choisir un noeud fils tout en maintenant le compromis entre exploitation et exploration. Ce compromis se traduit par le choix du noeud k qui maximise la formule UCB1. Voici ci-dessous le résumé de l'algorithme MCTS :

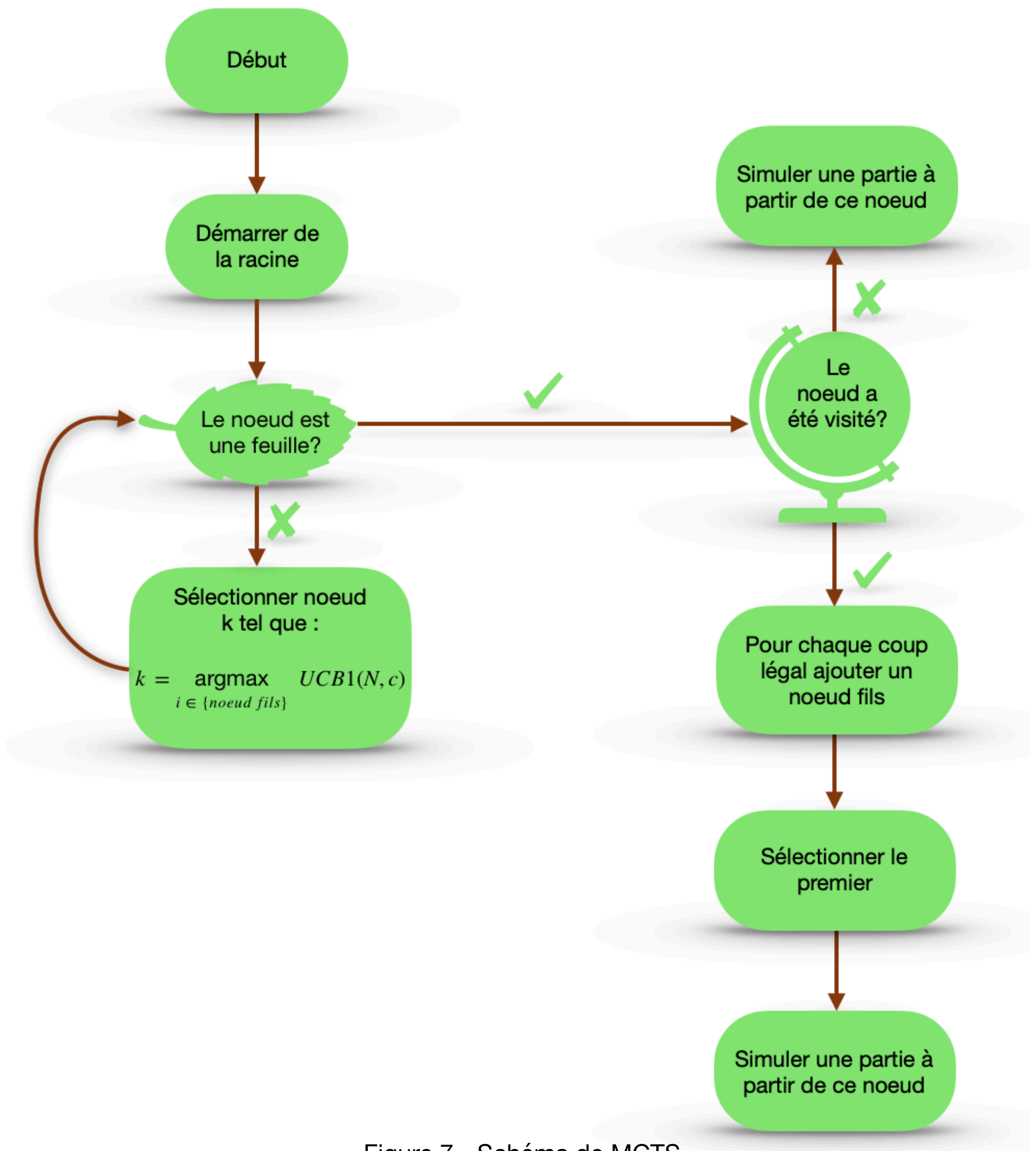


Figure 7 - Schéma de MCTS

Lorsque l'on ajoute un noeud fils par coup possible on sélectionne le premier noeud pour simuler la partie. Cette règle provient du fait que le nombre de visites de chacun de ces noeuds est nul ce qui implique que chaque noeud est évalué en l'infini par la formule UCB1. La règle devient alors choisir le noeud par ordre numérique.

2.3 Exemple

Nous allons donner un exemple d'application de l'algorithme. Dans cet exemple, on pourra construire un arbre à deux niveaux.

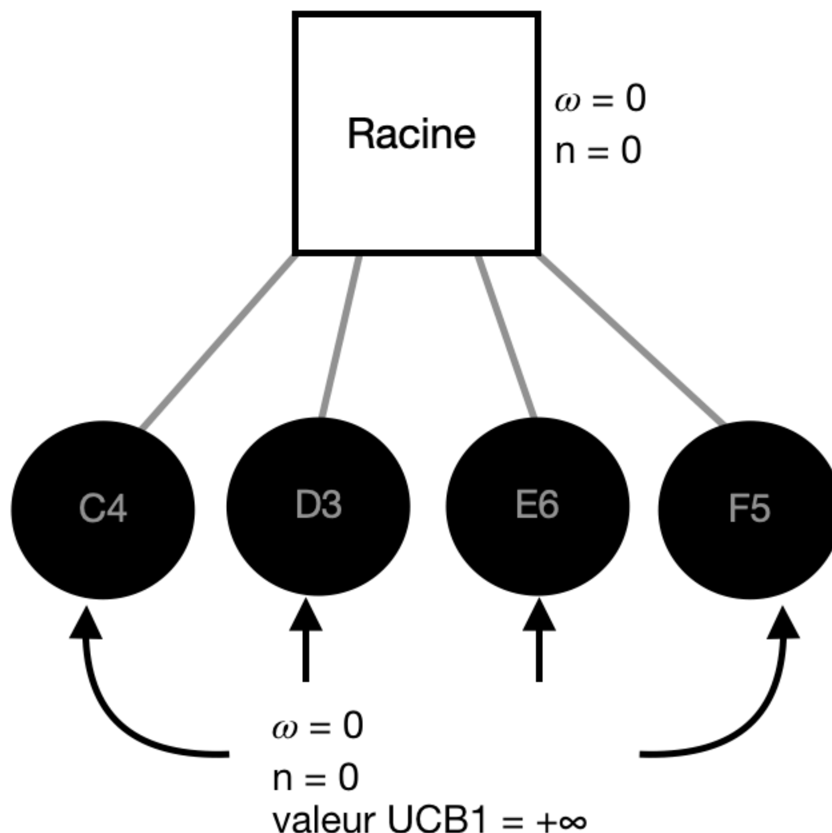


Figure 8.a - Arbre de début de jeu

On applique l'algorithme de MCTS. Les noirs commencent à jouer. La racine à ce moment est une feuille. Les noirs peuvent jouer sur 4 cases, on ajoute alors 4 noeuds à la racine. Tous ces noeuds possèdent les mêmes données alors on choisit le premier noeud. Comme ce noeud est une feuille qui n'a pas été visitée, on simule la partie entière. On suppose que les noirs ont gagné cette partie. On procède alors maintenant à la backpropagation et on met à jour les données du noeud sélectionné. Ainsi $\omega_{C4} = 1$, $n_{C4} = 1$ de même que $\omega_{racine} = 1$ et $n_{racine} = 1$. Lors des 3 itérations suivantes, les noeuds D3, E6 et F5 seront sélectionnés et mis à jour car ils n'avaient pas encore été visités (ainsi leur valeur UCB1 était infinie d'où leur choix). On suppose que ces trois simulations aboutissent à des parties perdues par les noirs. À ce stade, les 4 noeuds ont été visités

donc lors de la cinquième itération on utilise la maximisation de la formule UCB1 pour choisir le noeud. C'est donc le noeud C4 qui maximise la formule.

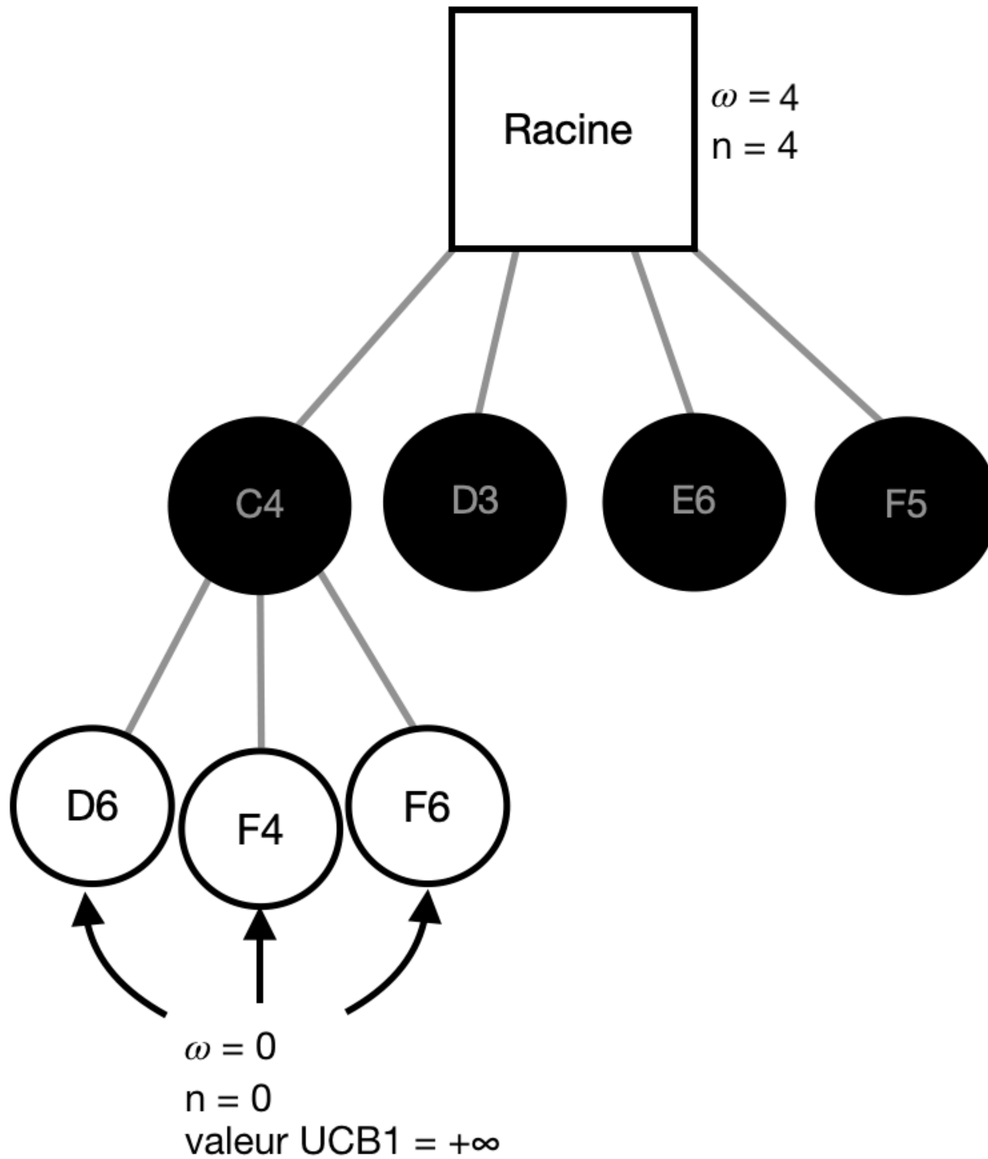


Figure 8.b - cinquième itération de MCTS

Le noeud C4 a été sélectionné, c'est une feuille qui a déjà été visitée donc on lui ajoute 3 noeuds fils qui correspondent aux coups légaux disponibles pour les blancs. On sélectionne alors le premier (D6) et on simule la partie. On abouti à une victoire pour les blancs alors $\omega_{D6} = 1$ et $n_{D6} = 1$. Et ainsi de suite.

3. Aspect théorique et expérimental de l'algorithme MCTS

3.1 Positionnement du problème

L'algorithme UCB1 est un algorithme heuristique. Nous n'avons donc aucune certitude sur l'efficacité du procédé présenté. Il existe un moyen de mesurer le succès d'une méthode en considérant le regret. Le regret est la perte due au fait que le coup optimal n'est pas systématiquement choisi. L'un des exemples les plus parlant pour le dilemme d'exploration/exploitation est l'exemple du problème du bandit manchot multi bras. Nous allons présenter ce problème sous forme imagée de la façon suivante :
Un joueur se tient devant une rangée de machines à sous qui a pour objectif de maximiser ses gains. Pour cela il doit décider quelle machine jouer. Chaque machine donne une récompense moyenne que le joueur ne connait pas au préalable. Pour atteindre au mieux son objectif, le joueur va adopter la politique de recherche de compromis entre exploration et exploitation. L'exploration consiste à tester plusieurs machines et enregistrer les gains qu'elles lui donnent, et l'exploitation traduit le fait que le joueur choisi la machine qui lui semble la plus rentable.

Donnons une formalisation mathématique du problème. Un problème de bandit manchot à K bras est défini par des variables aléatoires $X_{i,n}$ avec $1 \leq i \leq K$ et $1 \leq n$, où i est l'indice de la machine à sous (c'est à dire le bras i du bandit). Plusieurs tours de jeu successifs sur la machine i conduisent aux gains $X_{i,1}, X_{i,2}, \ldots$ qui sont indépendants et identiquement distribués selon une loi de probabilité inconnue avec une espérance inconnue du joueur. L'indépendance a aussi lieu entre les gains de différentes machines, c'est à dire que $X_{i,n}$ et $X_{j,m}$ sont indépendantes (et généralement pas identiquement distribuées). Une politique, typiquement ici l'*Upper confidence Bound 1* ou UCB1, est un algorithme qui choisit la prochaine machine à jouer en se basant sur les parties jouées précédentes et les gains obtenus durant ces jeux. Soit $T_i(n)$ le nombre de fois où la machine i a été jouée par UCB1 sur n parties. Alors le regret après n parties est défini par:

$n\mu^* - \sum_{j=1}^{K} \mu_j \mathbb{E}[T_j(n)]$ où $\mu^* = \max_{1 \leq i \leq K} \mu_i$ et μ_k est l'espérance du gain sur la machine k.

De ce fait, ci dessus on définit le regret comme la perte attendue du au fait que la machine donnant le plus de gains n'est pas systématiquement choisi par l'algorithme UCB1. Nous allons voir que nous pouvons borner le regret. Par la suite on désignera avec $*$ les quantités se rapportant à la machine donnant le plus de gains. Ainsi, par exemple on suppose que l'indice de la machine optimale est s alors on notera $X_{s,n}$ le gain de la machine optimale après n parties par X_n^*.

3.2 Théorème et preuve

L'algorithme considéré est le procédé UCB1. On initialise l'algorithme en visitant une fois chaque machine à sous. Ensuite, la boucle consiste à choisir la machine j telle que $j = \underset{1 \leq i \leq K}{argmax} \left[\frac{\omega_i}{n_i} + c\sqrt{\frac{\ln(N)}{n_i}} \right]$

De plus, on adoptera les notations suivantes :

$\Delta_i := \mu^* - \mu_i$

$\bar{X}_{i,n} = \frac{1}{n} S_{i,n}$ avec $S_{i,n} = \sum_{j=1}^{n} X_{i,j}$

Enfin, on admettra la formule de la borne de Chernoff-Hoeffding :

Soit $X_1, X_2, \ldots, X_n$ des variables aléatoires à support dans $[0,1]$ et telles que $\mathbb{E}\left[X_t | X_1, \ldots, X_{t-1}\right] = \mu$. Soit $S_n = X_1 + \ldots + X_n$. Alors pour tout $a \geq 0$:

$$\mathbb{P}(S_n \geq n\mu + a) \leq e^{-2a^2/n} \text{ et } \mathbb{P}(S_n \leq n\mu - a) \leq e^{-2a^2/n}$$

Théorème :

Pour tout $K > 1$, si on applique UCB1 sur K machines ayant une distribution de gain arbitraire à support dans $[0,1]$ alors l'espérance du regret après $n \in \mathbb{N}$ parties est au plus $\left[8 \sum_{i:\mu_i<\mu^*} \left(\frac{\ln(n)}{\Delta_i} \right) \right] + \left(1 + \frac{\pi^2}{3} \right) \left(\sum_{j=1}^{K} \Delta_i \right)$

Preuve :

On notera par souci de lisibilité $1_{\{x=y\}}$ par $\{x = y\}$. C'est à dire que si $x = y$ alors $\{x = y\} = 1$ sinon, $\{x = y\} = 0$. On rappelle que $T_i(n)$ est le nombre de fois où la machine i a été jouée par UCB1 sur n parties. Enfin on note par $I_t = i$ l'événement « le joueur se trouve sur la machine i au temps t. Par ces informations on en déduit que :

$$T_i(n) = \sum_{t=1}^{n} \{I_t = i\}$$

Or chaque coup est joué au moins une fois donc :

$$T_i(n) = 1 + \sum_{t=K+1}^{n} \{I_t = i\}$$

$$= 1 + \sum_{t=K+1}^{n} \{I_t = i, T_i(t-1) < l\} + \{I_t = i, T_i(t-1) \geq l\}$$

Montrons que pour $l \in \mathbb{N}$: $\sum_{t=K+1}^{n} \{I_t = i, T_i(t-1) < l\} \leq l - 1$.

Soit $l \in \mathbb{N}^*$. Dans le cas où cette somme est maximale, c'est à dire dans le cas où on reste toujours sur la même machine i alors $\{I_t = i\} = 1$. Comme chaque machine est jouée une fois sur les K premières parties, on a alors $T_i(K) = 1$. Et pour $t \in [K+1, n]$:

Quand $t = K+1$, $T_i(t-1) = T_i(K) = 1$ donc $\{T_i(K) < l\} = 1$

Quand $t = K+2$, $T_i(K+1) = 2$ donc $\{T_i(K+1) < l\} = 1$

$\vdots$

Quand $t = K+l-1$, $T_i(K+(l-1)) = l-1$ donc $\{T_i(K+l-1) < l\} = 1$

Quand $t = K+l$, $T_i(K+l) = l$ donc $\{T_i(K+l) < l\} = 0$

Donc au mieux, $\sum_{t=K+1}^{n} \{I_t = i, T_i(t-1) \geq l\} = l - 1$

Ce qui donne :

$$T_i(n) \leq 1 + (l-1) + \sum_{t=K+1}^{n} \{I_t = i, T_i(t-1) \geq l\} = l + \sum_{t=K+1}^{n} \{I_t = i, T_i(t-1) \geq l\}$$

De plus, être sur la machine i ($\{I_t = i\} = 1$) résulte du fait que i est l'argmax de la formule UCB1. On obtient alors en particulier :

$$T_i(n) \leq l + \sum_{t=K+1}^{n} \left\{\bar{X}^*_{T^*(t-1)} + c_{t-1,T^*(t-1)} \leq \bar{X}_{i,T_i(t-1)} + c_{t-1,T_i(t-1)},\ T_i(t-1) \geq l\right\}$$

Il est clair par ci-dessus, en supposant que $\{T_i(t-1) \geq l\} = 1$ pour tout t, que :

$$T_i(n) \leq l + \sum_{t=K+1}^{n} \left\{\bar{X}^*_{T^*(t-1)} + c_{t-1,T^*(t-1)} \leq \bar{X}_{i,T_i(t-1)} + c_{t-1,T_i(t-1)}\right\}$$

Et donc :

$$T_i(n) \leq l + \sum_{t=K+1}^{n} \left\{\min_{0<s<t} (\bar{X}^*_s + c_{t-1,s}) \leq \max_{l<s_i<t} (\bar{X}_{i,s_i} + c_{t-1,s_i})\right\}$$

On remarque que :

$$\left\{\min_{0<s<t} (\bar{X}^*_s + c_{t-1,s}) \leq \max_{l<s_i<t} (\bar{X}_{i,s_i} + c_{t-1,s_i})\right\} \leq \sum_{s_i=l}^{t-1} \left\{\min_{0<s<t} (\bar{X}^*_s + c_{t-1,s}) \leq \bar{X}_{i,s_i} + c_{t-1,s_i}\right\}$$

$$\leq \sum_{s=1}^{t-1} \sum_{s_i=l}^{t-1} \left\{\bar{X}^*_s + c_{t-1,s} \leq \bar{X}_{i,s_i} + c_{t-1,s_i}\right\}$$

car le max est l'un des termes de la première somme et le min est l'un des termes de la deuxième.

On trouve finalement que :

$$T_i(n) \leq l + \sum_{t=1}^{\infty} \sum_{s=1}^{t-1} \sum_{s_i=l}^{t-1} \left\{\bar{X}^*_s + c_{t,s} \leq \bar{X}_{i,s_i} + c_{t,s_i}\right\}$$

Maintenant, on observe que si :

$$\bar{X}^*_s + c_{t,s} \leq \bar{X}_{i,s_i} + c_{t,s_i} \quad (1)$$

$$\bar{X}_s^* \leq \bar{\mu}^* - c_{t,s} \quad (2)$$
$$\bar{X}_{i,s_i} \geq \mu_i + c_{t,s_i} \quad (3)$$
$$\mu^* < \mu_i + 2c_{t,s_i} \quad (4)$$

Alors l'une des inégalités suivantes est vérifiée :

$$\bar{X}_s^* > \mu^* - c_{t,s} \quad (5)$$
$$\bar{X}_{i,s_i} < \mu_i + c_{t,s_i} \quad (6)$$
$$\mu^* \geq \mu_i + 2c_{t,s_i} \quad (7)$$

En effet, pour prouver cela on suppose que les trois inégalités ci-dessus sont fausses c'est à dire :

(5) implique :

$$\bar{X}_s^* + c_{t,s} > \bar{\mu}^*$$

(7) nous donne, grâce à cette inégalité :

$$\bar{X}_s^* + c_{t,s} > \mu_i + 2c_{t,s_i}$$

Et finalement de (6) on obtient :

$$\bar{X}_s^* + c_{t,s} > \bar{X}_{i,s_i} + c_{t,s_i}$$

On abouti à une contradiction, ce qui prouve le résultat énoncé au dessus.

Ensuite, on voit que pour $s_i \geq 8(\ln n)/\Delta_i^2$ l'équation (4) est fausse car :

$\mu^* - \mu_i - 2c_{t,s_i} = \mu^* - \mu_i - 2\sqrt{2(\ln n)/s_i} \geq \mu^* - \mu_i - \Delta_i = 0$ d'après la définition de Δ_i.
On va voir de plus, grâce à la borne de Chernoff-Hoeffding qu'on obtient les majorations suivantes :

$$\mathbb{P}(\bar{X}_{i,s_i} \geq \mu_i + c_{t,s_i}) \leq e^{-4\ln t} = t^{-4} \text{ et } \mathbb{P}(\bar{X}_s^* \leq \mu^* - c_{t,s}) \leq e^{-4\ln t} = t^{-4}$$

En effet, Chernoff-Hoeffding nous dit que :
$\mathbb{P}(S_n \geq n\mu + c_{t,s}) \leq e^{-2c_{t,s}^2/n}$ ainsi que $\mathbb{P}(S_s^* \leq n\mu^* - c_{t,s}) \leq e^{-2c_{t,s}^2/n}$

$$
\begin{aligned}
\mathbb{P}(\bar{X}_s^* \le \mu^* - c_{t,s}) &= \mathbb{P}(s\bar{X}_s^* \le s\mu^* - sc_{t,s}) \\
&= \mathbb{P}(S_s^* \le s\mu^* - sc_{t,s}) \\
&\le e^{-2(sc_{t,s})^2/s} \\
&= e^{-2s\left(\sqrt{\frac{2\ln t}{s}}\right)^2} \\
&= e^{-4\ln t} \\
&= t^{-4}
\end{aligned}
$$

Et :

De la même manière on obtient : $\mathbb{P}(\bar{X}_{i,s_i} \ge \mu_i + c_{t,s_i}) \le t^{-4}$ (8).

En utilisant la majoration de $T_i(n)$ et $l = \lceil 8(\ln n)/\Delta_i^2 \rceil$, on peut écrire :

$$T_i(n) \le \lceil 8(\ln n)/\Delta_i^2 \rceil + \sum_{t=1}^{\infty} \sum_{s=1}^{t-1} \sum_{s_i=\lceil 8(\ln n)/\Delta_i^2 \rceil}^{t-1} \left\{ \bar{X}_s^* + c_{t,s} \le \bar{X}_{i,s_i} + c_{t,s_i} \right\}$$

On a vu au dessus que si on avait (1) alors une au moins des inégalités $(2), (3)$ ou (4) était vraie, or ici la (4) est fausse donc au moins une des deux autres inégalités est vraie. Ce qui permet d'écrire :

$$\left\{ \bar{X}_s^* + c_{t,s} \le \bar{X}_{i,s_i} + c_{t,s_i} \right\} \le \left\{ \bar{X}_{i,s_i} \ge \mu_i + c_{t,s_i} \right\} + \left\{ \bar{X}_s^* \le \bar{\mu}^* - c_{t,s} \right\}$$

En utilisant la linéarité de l'espérance ainsi que :

$$\mathbb{E}\left(\left\{ \bar{X}_{i,s_i} \ge \mu_i + c_{t,s_i} \right\} + \left\{ \bar{X}_s^* \le \bar{\mu}^* - c_{t,s} \right\} \right) = \mathbb{P}(\bar{X}_s^* \le \bar{\mu}^* - c_{t,s}) + \mathbb{P}(\bar{X}_{i,s_i} \ge \mu_i + c_{t,s_i})$$

On a :

$$\mathbb{E}[T_i(n)] \le \lceil 8(\ln n)/\Delta_i^2 \rceil + \sum_{t=1}^{\infty} \sum_{s=1}^{t-1} \sum_{s_i=\lceil 8(\ln n)/\Delta_i^2 \rceil}^{t-1} \left[\mathbb{P}(\bar{X}_s^* \le \bar{\mu}^* - c_{t,s}) + \mathbb{P}(\bar{X}_{i,s_i} \ge \mu_i + c_{t,s_i}) \right]$$

Puis en appliquant la borne de Chernoff-Hoeffding, en montrant l'inégalité (8) :

$$E[T_i(n)] \le \lceil 8(\ln n)/\Delta_i^2 \rceil + \sum_{t=1}^{\infty} \sum_{s=1}^{t} \sum_{s_i=1}^{t} 2t^{-4}$$

$$E[T_i(n)] \le 8(\ln n)/\Delta_i^2 + 1 + \frac{\pi^2}{3} \quad \text{car} \quad \sum_{t=1}^{\infty} t^{-2} = \frac{\pi^2}{6}$$

Pour conclure la preuve, on remarque que le regret peut s'écrire de la manière suivante : $\sum_{j=\mu_j<\mu^*} \Delta_j \mathbb{E}(T_j(n))$. En effet, le regret est $n\mu^* - \sum_{j=1}^{K} \mu_j \mathbb{E}[T_j(n)]$. Mais puisque

$\sum_{j=1}^{K} T_i(n) = n$ (la somme des nombres de visites de chaque machine sur n parties est égale à n) on peut dire que le regret s'écrit de la façon suivante :
$\mu^* \sum_{i=1}^{K} T_i(n) - \sum_{i=1}^{K} \mu_i \mathbb{E}(T_i(n))$ et en prenant l'espérance de cette expression on obtient la formule : $\sum_{i=1}^{K} (\mu^* - \mu_i)\mathbb{E}(T_i(n))$ c'est à dire : $\sum_{i=1}^{K} \Delta_i \mathbb{E}(T_i(n))$. Enfin, $T_i(n)$ fait intervenir Δ_i au dénominateur et $\Delta^* = \mu^* - \mu^* = 0$ alors on enlève l'indice qui correspond à la machine optimale. L'espérance du regret devient alors $\sum_{i:\mu_i<\mu^*} \Delta_i \mathbb{E}(T_i(n))$.

Finalement,

$$\sum_{i=\mu_i<\mu^*} \Delta_i \mathbb{E}(T_i(n)) \leq \sum_{i=\mu_i<\mu^*} \Delta_i \left[8 \frac{\ln n}{\Delta_i^2} + 1 + \frac{\pi^2}{3} \right]$$
$$\leq 8 \times \sum_{i:\mu_i<\mu^*} \frac{\ln n}{\Delta_i} + (1 + \frac{\pi^2}{3})(\sum_{i=1}^{K} \Delta_i)$$

Ce qu'il fallait démontrer.

3.3 Expérimentation

Lors de la présentation de la méthode, nous avons précisé que celle-ci était heuristique. Est-ce que la recherche arborescente de Monte Carlo vaut la peine d'être implémentée? Est-ce plus efficace qu'un jeu aléatoire?

Pour y répondre nous avons voulu vérifier notre hypothèse selon laquelle le programme pouvait mettre en oeuvre une certaine stratégie favorisant la victoire.

Pour initialiser l'expérience nous avons commencé par faire jouer l'ordinateur contre lui même d'une manière aléatoire en guise de témoin. On a compté (sur 100 parties jouées) le nombre de parties remportées par l'une des deux couleurs en simulant 10 fois l'expérience et nous avons obtenu une moyenne de 45 parties remportées. Cela illustre assez bien le fait que les adversaires jouent aléatoirement et remportent chacun environ le même nombre de parties. La deuxième partie de l'expérience consiste à remplacer un des deux joueurs aléatoires par l'IA.

Nous avons ainsi reproduit l'expérience présentée ci-dessus avec l'intelligence artificielle présentant différents degrés d'entraînement (croissants) du programme. Comme nous l'attendions l'IA a une réelle efficacité. En effet, plus elle était entrainée plus elle remportait de parties.

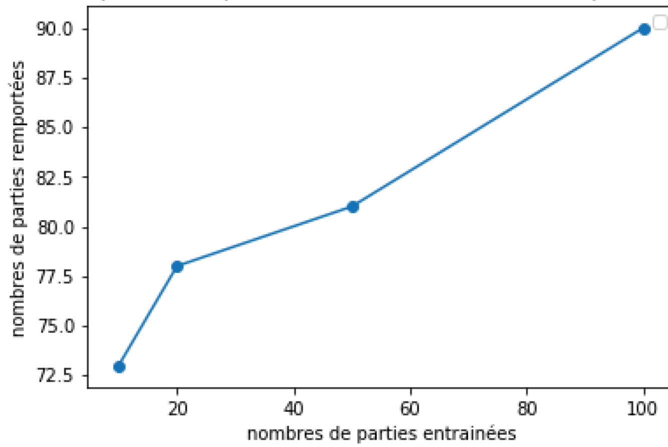


Figure 9

Le graphique représenté sur la figure 9 illustre bien le fait que l'IA a un réel effet. Le nombre de parties entrainées favorise le nombre de parties gagnées. Ainsi en ayant un entrainement de 10 parties, le programme a remporté 72 parties contre le jeu aléatoire. Avec un entrainement plus poussé allant jusqu'à 100 parties, l'IA est capable de gagner plus de 90% des parties simulées.

Le programme ayant validé l'hypothèse précédente, nous avons voulu étudier l'influence du paramètre c dans la formule UCB1. Pour cela, nous avons réalisé une expérience proche de la première en simulant 100 parties avec un programme jouant aléatoirement contre l'algorithme MCTS entrainé sur 10 parties (dans un souci de temps d'exécution du programme). Ceci répété plusieurs fois avec un paramètre c différent. Nous obtenons les résultats suivants :

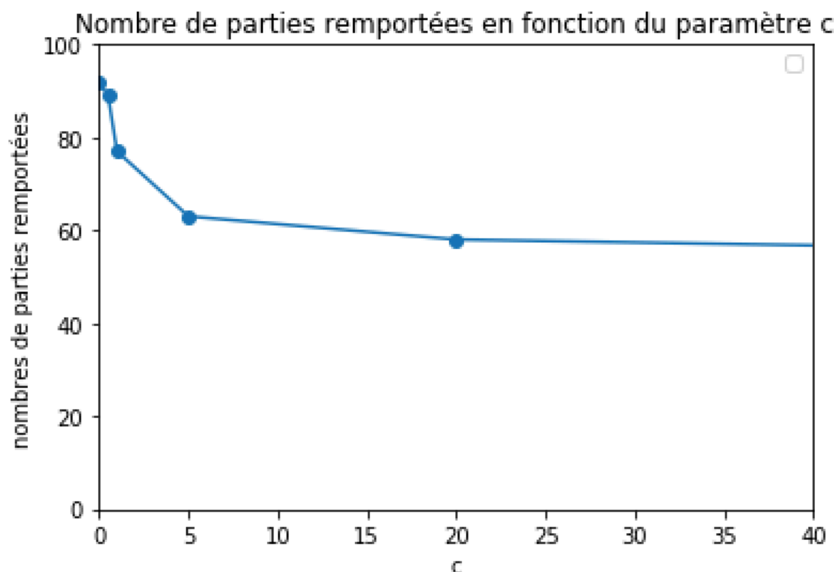


Figure 10

Sur la figure 10, on voit que plus le paramètre c est grand plus le nombre de victoires se rapproche de 50. Ce qui montre que plus l'exploration est privilégiée moins le programme est fort. En effet, pour $c = 1000$, le programme n'a remporté que 52 parties. Favoriser grandement l'exploration s'apparente à faire jouer le programme aléatoirement. Au contraire, privilégier une petite valeur du paramètre semble accorder une victoire bien

plus certaine. Pour $c = 0$ le nombre de parties remportées est maximal. Le programme ne fait qu'exploiter les connaissances qu'il a acquises lors des simulations. Cela compromettrait la pertinence de l'exploration. Cependant, nous ne pouvons pas affirmer cela car notre expérience ne repose que sur quelques parties simulées et nécessite une étude approfondie. Pour cela, il faudrait entrainer le programme sur beaucoup plus de parties et simuler bien plus que 100 parties jouées.

4. Construction de l'intelligence artificielle du jeu

4.1 Qu'est-ce qu'une intelligence artificielle ?

L'Intelligence Artificielle (IA) est le domaine qui permet à une machine de simuler l'intelligence. L'IA trouve ses origines dans les années 1950 à travers les travaux d'Alan Turing (1912-1954, mathématicien britannique) qui se demande si une machine peut « penser ». Ce domaine repose sur des connaissances informatiques mais aussi sur certains domaines des sciences du vivant comme la neurobiologie computationnelle ainsi que les mathématiques.
De nos jours, l'IA est déjà très utilisée. En effet, on en retrouve dans plusieurs champs d'application comme celui de l'automobile, l'aéronautique ou encore les jeux vidéos. De plus en plus de studios de développement de jeux videos incluent de l'IA dans leurs projets pour fournir aux joueurs une expérience plus complète, par exemple en permettant au joueur de choisir l'évolution du jeu en fonction de ses propres choix au cours de ses parties (plusieurs exemples peuvent être donnés comme Assassin's Creed Odyssey, Fallout 4…). Ainsi, l'algorithme MCTS étudié est implanté dans le jeu video Total War : Rome II (jeu de stratégie) sorti en 2004. Enfin, l'IA a aussi été testée et utilisée dans le domaine du jeu de société. Plusieurs programmes informatiques ont été produits pour notamment le jeu de GO (l'AlphaGo), le jeu d'Echec ou récemment le poker. En 2017, l'AlphaGo, un programme informatique développé par Google, a battu plusieurs fois le champion du monde et montre ainsi ses capacités.

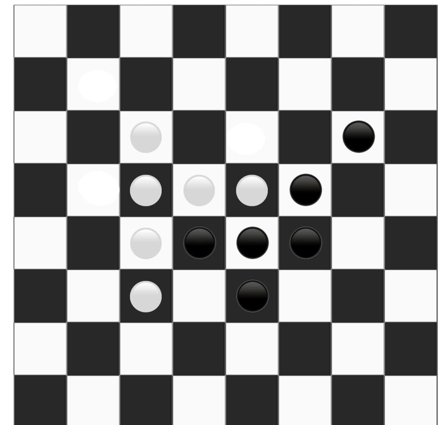
4.2 Construction informatique du jeu et de l'algorithme de recherche arborescente

4.2.1 Outils de développement

La construction du jeu de société nécessite de pouvoir produire une interface graphique. Nous nous étions dans un premier temps dirigés vers le moteur de jeu multiplateforme Godot Engine. Après avoir tenté de comprendre comment fonctionnait le logiciel, nous nous sommes rendus compte qu'il fallait avoir des compétences développées en informatique lesquelles nous aurions mis beaucoup trop de temps à acquérir. Par conséquent, nous nous sommes réorientés vers le C++ ce qui nous permettait d'utiliser les cours de C++ que nous avions eus et ainsi pouvoir développer nos compétences dans ce langage. Plus précisément, nous avons utilisé la SFML (Simple and Fast Multimédia Library) qui est une interface de programmation pour construire des jeux videos en 2D. Cette librairie nous permet de fournir une fenêtre graphique sur laquelle nous pouvons jouer.
Dans la fenêtre, nous faisons apparaitre un plateau de jeu de 64 cases. Pour pouvoir placer les pions, nous avons développé certaines fonctions qui permettent de placer le pion au centre d'une case donnée en cliquant dessus. Concernant l'IA, pour pouvoir jouer, le programme fournit les coordonnées du coup qui veut réaliser, et parmi les modules créés se trouvent certaines fonctions permettant de placer le pion automatiquement avec ces coordonnées.

Le jeu repose sur de la manipulation de matrice. En effet, nous avons représenté le plateau de jeu de 8x8 cases par une matrice à 8 lignes et 8 colonnes. Les éléments de cette matrice sont soit 0, 1 ou 2. 0 correspond à une case vide, 1 correspond à une case sur laquelle se trouve un pion noir et les pions blancs se trouvent sur la case marquée de l'élément 2. Par exemple, si l'élément (3,4) de la matrice est 2, alors un pion blanc se trouve sur la case (3,4) (ou sur la case D3). Ainsi,

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 2 & 2 & 2 & 1 & 0 & 0 \\ 0 & 0 & 2 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 2 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$



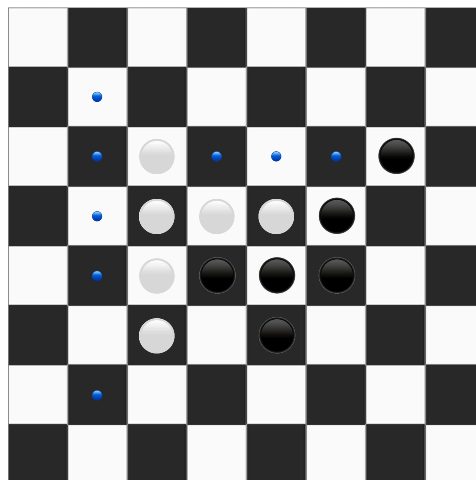


Figure 11 - C'est au noir de jouer

Pour pouvoir jouer, le joueur physique dispose d'indicateurs (ronds bleus) pour pouvoir choisir le coup à jouer comme ci-dessus. Quant à l'ordinateur, il joue ses coups en fournissant à la fonction d'affichage les coordonnées du coup qu'il a sélectionné après avoir appliqué l'algorithme MCTS plusieurs fois (par exemple 500 fois pour qu'il y ait assez de simulations de parties pour pouvoir donner un coup suffisamment efficace). Dans la partie suivante, nous allons expliquer comment l'ordinateur est capable de rechercher, choisir et fournir un coup à jouer et ainsi placer le pion correspondant sur le plateau.

4.2.2 Modules, programmes et fonctions développées

Le projet comporte plusieurs fichiers de code. Le jeu étant basé sur la manipulation de matrice, nous avons pensé qu'il était intéressant d'utiliser la classe « vecteur » et la classe « matrice » développées en cours de POOCNS en début d'année. Le projet nécessitait plusieurs autres fonctions pour manipuler les matrices (ajout de ligne, ne garder qu'un exemplaire d'une ou plusieurs lignes en plusieurs exemplaires dans la matrice etc) alors nous avons enrichi ces classes.
Le jeu reposant sur une arborescence, nous avons alors dû créer une classe « noeud » qui permet d'implémenter des arbres N-aire et ainsi pouvoir construire notre arborescence. Cette partie du projet nous a été particulièrement difficile, nous avons trouvé en open source une classe « noeud » qui compose la base de notre fichier. En effet, nous avons compris comment elle fonctionnait et nous avons alors pu l'enrichir de nos propres besoins en fonctions.
De plus, cette classe nous a permis de manipuler et comprendre mieux les pointeurs. En effet, chaque noeud (auquel sont rattachées plusieurs données comme le nombre de visite du noeud, le nombre de fils, les éventuels fils, l'éventuel père), est lié à son père (si ce noeud n'est pas la racine, auquel cas il n'a pas de père) et à ses fils s'il en a. Ces liens sont possibles ici par l'utilisation de pointeurs. Ils peuvent être vus comme les « bras » qui relient les noeuds entre eux comme par exemple sur la figure 8.a. Ces pointeurs sont particulièrement utiles pour la backpropagation expliquée dans la section 2.1.
Enfin, ces classes nous ont permis de construire le module « d'automatisation » qui contient les différentes fonctions permettant la construction de l'arbre et la recherche dans celui-ci.

Grâce à la construction de ces programmes, nous avons découvert la fuite de mémoire. Ce phénomène est en lien étroit avec l'utilisation des pointeurs. En effet, la fuite de mémoire apparait quand nous créons un programme qui ne supprime pas la mémoire. Chaque variable déclarée est libérée de la mémoire lorsque la fonction est finie d'être appelée. Un pointeur étant une variable qui stocke une adresse à laquelle sont stockées des informations (des entiers, une matrice etc), celui-ci est supprimé lorsque le programme a fini son exécution… Mais pas le contenu stocké à l'endroit vers lequel le pointeur fait référence. Par conséquent le pointeur et supprimé mais pas le contenu, lequel est impossible de retrouver sans le pointeur. La mémoire occupée par l'objet est alors perdue. Heureusement, à la fin du programme, le compilateur est capable de libérer la mémoire perdue. Le problème apparait quand on a besoin de faire fonctionner le programme sur de longues périodes. Effectivement, si perte de mémoire il y a, plus le programme va tourner plus la mémoire sera perdue et ainsi les performances du programme seront considérablement amoindries.
Nous avons rencontré le problème lors de l'exécution du jeu. En effet, en faisant tourner le projet nous nous sommes vite rendus compte que l'occupation de la mémoire était rapidement importante (de l'ordre du Giga en 1 minute d'exécution pour un programme qui n'occupe que quelques Mo). Après avoir réparé toutes les fuites de mémoire, le programme n'occupait plus que 100Mo de mémoire après le même temps d'utilisation. D'où l'importance de régler le problème, ce qui n'a pas été facile et a pris beaucoup de temps.

5. Conclusion

Lors de notre projet, nous avons appris les axes principaux de l'intelligence artificielle. En effet, notre exemple d'application n'est en fait qu'un exemple très simple dans lequel ce type de technologie peut être utilisé.
Nous avons vu que l'IA implémentée grâce à la recherche arborescente de Monte Carlo avait une réelle efficacité et pouvait être un adversaire donnant l'impression d'une mise en place d'une stratégie pour gagner. Ainsi, le jeu fournit l'expérience attendu d'un jeu de réflexion. En effet, outre le programme adversaire, le jeu a subi au cours de la construction du projet, des modifications ergonomiques pour améliorer l'expérience de jeu. Bien que cela n'ait pas bénéficier de notre pleine attention, nous voulions fournir une interface de jeu acceptable et ludique.

De plus, bien que la méthode soit heuristique, nous avons vu que le regret, correspondant à la perte attendue du fait des mauvais choix pris, n'explosait pas et pouvait être borné. Cela garantit que la politique UCB1 n'est pas sans intérêt.
Enfin, les compétences de l'intelligence artificielle sont assez étonnantes. Néanmoins, Celles-ci pourraient l'être encore plus en optimisant le code source. Mais aussi, nous pourrions rendre le programme encore plus fort en faisant entrer plus de paramètres. Notamment, il est possible de pondérer certain coups, ce qui permettrait de les favoriser et conduire à une victoire encore plus certaine. La méthode implémentée peut être améliorée et nous disposons aujourd'hui de travaux qui améliore les compétences de l'algorithme.

Annexes

Bibliographie

Page wikipédia recherche arborescente de Monte Carlo :
https://fr.wikipedia.org/wiki/Recherche_arborescente_Monte-Carlo

Article de recherche « Finite time analysis of the multiarmed bandit problem » de Peter AUER, Nicolo CESA-BIANCHI et Paul FISCHER.

Figures 1, 2, 3, 4, 5 : http://www.ffothello.org/othello/regles-du-jeu/

Autres figures : réalisations personnelles

Code informatique

L'ensemble des fichiers composant le projet représente environ 3000 lignes de codes. Voici ci-dessous les parties les plus importantes.

Évaluation des coups

Voici le code de la fonction d'évaluation des coups possibles verticalement. Les trois autres directions (horizontale, diagonale et antidiagonale) sont codées de manière similaire.

```
//-----------------------------------//
//Cette fonction évalue les coups possibles verticalement à un stade de la partie donnée
dans la matrice partie.
//L'argument couleur est celui qui définit à qui le tour est.

matrice coups_possibles_verticaux(matrice& partie, int const couleur) {
    //EXEMPLE : Pour comprendre le code, dans les commentaires, la variable couleur sera
egale à 1 c'est à dire la couleur noir.

    //Variables globales
    int couleur_adverse = Autre_Couleur(couleur);

    matrice coord_coups(1,2);
    //On récupère le nombre de pions pions et leurs coordonnées
    matrice coord_pion_couleur (1,2);
    for (int i=0; i<partie.dim1(); i++){
        for(int j=0; j<partie.dim2(); j++) {
            if(partie(i,j) == couleur) {
                coord_pion_couleur.ajout_ligne();
                coord_pion_couleur(coord_pion_couleur.dim1()-1,0) = i;
                coord_pion_couleur(coord_pion_couleur.dim1()-1,1) = j;
            }
        }
    }
    coord_pion_couleur.enlever_premiere_ligne(); //La premiere ligne est (0,0) et doit
être enlevée (ce n'est pas forcement un coup)
    int nb_pion_couleur = 0;
    nb_pion_couleur = coord_pion_couleur.dim1();
    //Pour chaque pion couleur
    for(int i=0; i<nb_pion_couleur; i++) {

        //il y a des blancs au DESSUS? (faut aussi qu'il n'y a pas de trous entre les
deux)
```

```
        //combien des blancs au dessus
        int couleur_adverse_dessus = 0;
        int ordonnee_couleur_adverse_dessus = 0;
        for(int ii=coord_pion_couleur(i,0)-1; ii>=0; ii--) {
            if(partie(ii,coord_pion_couleur(i,1)) == couleur_adverse) {
                couleur_adverse_dessus += 1;
                ordonnee_couleur_adverse_dessus = ii;
            }
        }

        //Faut regarder s'il y a des trous entre les deux
        //comptage de trou
        int trous_entre_dessus = 0;
        for(int ii=ordonnee_couleur_adverse_dessus; ii<coord_pion_couleur(i,0); ii++) {
           if(partie(ii,coord_pion_couleur(i,1)) == 0) trous_entre_dessus +=1;
        }

        //S'il y a des pions au dessus et s'il n'y a pas de trous :
        if(couleur_adverse_dessus != 0 && trous_entre_dessus == 0) {
            if(ordonnee_couleur_adverse_dessus>0 &&
partie(ordonnee_couleur_adverse_dessus - 1,coord_pion_couleur(i,1)) == 0) {
                coord_coups.ajout_ligne();
                coord_coups(coord_coups.dim1()-1,1) = ordonnee_couleur_adverse_dessus -
1; // remplissage à la fin
                coord_coups(coord_coups.dim1()-1,0) = coord_pion_couleur(i,1); //
coordonnées echec (premiere est colonne)
            }
        }
        else {
            //pour le cas blanc de jouer et blanc noir trou noir alors coup sur le trou
            if(couleur_adverse_dessus != 0 && trous_entre_dessus !=0) { //s'il y a des
blancs mais aussi des trous
                //on regarde si le pion juste apres est un pion adverse sinon coup
impossible
                if((partie(coord_pion_couleur(i,0)-1, coord_pion_couleur(i,1)) ==
couleur_adverse) && (trous_entre_dessus != 0)) {
                    //compte combien de blancs au dessus du noir avant le 1er trou
                    int trous_dessus = 0;
                    int mmcouleur_adverse_consecutif_dessus = 0;
                    for(int ii=coord_pion_couleur(i,0)-1; ii>=0; ii--){
                        if(partie(ii,coord_pion_couleur(i,1)) == 0) trous_dessus += 1;
                        if(partie(ii,coord_pion_couleur(i,1)) == couleur_adverse &&
trous_dessus == 0){
                            mmcouleur_adverse_consecutif_dessus+=1;
                        }
                    }

                    if(mmcouleur_adverse_consecutif_dessus != coord_pion_couleur(i,0)) {
//on regarde si la colonne n'est pas remplie
                        int trous = 0;
                        int position_trou_dessus = coord_pion_couleur(i,0);

                        while(trous == 0) {
                            position_trou_dessus -= 1;
                            if(partie(position_trou_dessus, coord_pion_couleur(i,1)) ==
0) trous +=1; //pour arreter while
                        }
                        //Si le pion juste avant le trou est noir alors coup illégal
                        if(partie(position_trou_dessus+1,coord_pion_couleur(i,1)) !=
couleur) {
                            coord_coups.ajout_ligne();
                            coord_coups(coord_coups.dim1()-1,1) = position_trou_dessus;
                            coord_coups(coord_coups.dim1()-1,0) = coord_pion_couleur(i,
1);
                        }
                    }
                }
            }
        } //fin else

        //il y a des blancs en DESSOUS? On en recupère le nombre de blancs et l'ordonnee
du dernier
```

```
        int couleur_adverse_dessous = 0;
        int ordonnee_couleur_adverse_dessous = 0;
        for(int ii=coord_pion_couleur(i,0)+1; ii<8; ii++) {
            if(partie(ii, coord_pion_couleur(i,1)) == couleur_adverse) {
                couleur_adverse_dessous += 1;
                ordonnee_couleur_adverse_dessous = ii;
            }
        }

        //Il faut regarder s'il y a des trous entre les deux
        //comptage de trous
        int trous_entre_dessous = 0;
        for(int ii=coord_pion_couleur(i,0)+1; ii < ordonnee_couleur_adverse_dessous; ii+
+) { //regarde si trous entre eux
            if(partie(ii,coord_pion_couleur(i,1)) == 0) trous_entre_dessous += 1;
        }
        if(couleur_adverse_dessous != 0 && trous_entre_dessous == 0) {
            if(ordonnee_couleur_adverse_dessous<7 &&
partie(ordonnee_couleur_adverse_dessous + 1,coord_pion_couleur(i,1)) == 0) { //
typiquement pour le cas un blanc entre 2 noirs
                    coord_coups.ajout_ligne();
                    coord_coups(coord_coups.dim1()-1,1) =
ordonnee_couleur_adverse_dessous + 1;
                    coord_coups(coord_coups.dim1()-1,0) = coord_pion_couleur(i,1); //
coordonnées echec (premiere est colonne)
            }
        }
        else {
            //pour le cas blanc de jouer et blanc noir trou noir alors coup sur le trou
            if(couleur_adverse_dessous != 0 && trous_entre_dessous !=0) { //s'il y a des
blancs mais aussi des trous
                //on regarde s'il y a un pion juste après sinon coup impossible
                if((partie(coord_pion_couleur(i,0)+1, coord_pion_couleur(i,1)) != 0) &&
(trous_entre_dessous != 0)) {
                    //combien de blancs au dessus du noir avant le 1er trou
                    int trous_dessous = 0;
                    int mmcouleur_adverse_consecutif_dessous = 0;
                    for(int ii=coord_pion_couleur(i,0)+1; ii<8; ii++){
                        if(partie(ii,coord_pion_couleur(i,1)) == 0) trous_dessous += 1;
                        if(partie(ii,coord_pion_couleur(i,1)) == couleur_adverse &&
trous_dessous == 0){
                            mmcouleur_adverse_consecutif_dessous+=1;
                        }
                    }

                    if(mmcouleur_adverse_consecutif_dessous != coord_pion_couleur(i,0))
{ //on regarde si la colonne n'est pas remplie
                        int trous = 0;
                        int position_trou_dessous = coord_pion_couleur(i,0);
                        while(trous == 0) {
                            position_trou_dessous += 1;
                            if(partie(position_trou_dessous, coord_pion_couleur(i,1)) ==
0) trous +=1; //pour arreter while

                        }
                        if(partie(position_trou_dessous-1,coord_pion_couleur(i,1)) !
=couleur) {
                            coord_coups.ajout_ligne();
                            coord_coups(coord_coups.dim1()-1,1) = position_trou_dessous;
                            coord_coups(coord_coups.dim1()-1,0) = coord_pion_couleur(i,
1);
                        }
                    }
                }
            }

        } // fin du else
    }//fin pour chaque pion couleur
    if (coord_coups.dim1()>1) coord_coups.enlever_premiere_ligne(); //si il y a au moins
un coup on peut enlever la premiere ligne (0,0)
    //Le coup (0,0) existe (case en haut à gauche) et le constructeur de la matrice la
rempli de 0.
```

```
    //Il faut donc enlever la ligne (0,0) et la remplacer par une autre puisque la 
matrice ne peut
    //pas être de dimension nulle. On choisit de remplir la premiere (et seule) ligne 
par (20,20).
    else {
        coord_coups(0,0) = 20;
        coord_coups(0,1) = 20;
    }
    return coord_coups;
} // fin fonction
```

Classe Noeud

Cette classe permet de créer l’arborescence recherchée.

```
class Noeud;

typedef Noeud * pNoeud;
typedef pNoeud * ppNoeud;

const pNoeud pN_NULL=0;
const ppNoeud ppN_NULL=0;


// Classe Noeud:
class Noeud {

     private:

          // Informations du noeud:
          int Niveau;                        // Niveau du noeud
          int nb_fils;                // Nombre de noeuds fils
        int couleur;                //La couleur pour lequel le noeud a joué
        double w;
        double nb_visite;
        double value; //Valeur de l'UCB1
        double indice_coup_l; //Indice ligne du coup
        double indice_coup_c; //Indice colonne du coup

          // Méthodes privées:
          void Calculer_Niveau();  // Mutateur descendant

          // Gestion dynamique des adresses des noeuds fils:
          void Initialiser_Fils(int _nb_fils);
          void Supprimer_Fils();

     public:

          // Liens:
          pNoeud Pere;        // Pointe le noeud pere
          ppNoeud Fils;       // Pointe vers le tableau des noeuds fils

          // Constructeur:
          Noeud() {
               Niveau=0;
               nb_fils=0;
            couleur = 0;
               Pere=pN_NULL;
               Fils=ppN_NULL;
            w=0;
            nb_visite=0;
            value=0;
            indice_coup_l=50;
            indice_coup_c=50;
          }

          // Destructeur:
          ~Noeud() {
```

```
            Supprimer_Noeuds_Fils(); // Appel récurssif de la méthode!
          //std::cout << "destructeur du noeud: " << this << std::endl;
        }

        // Méthodes associées:

        // Mutateurs:
        void Definir_Niveau(int);
        void Definir_nb_fils(int);
      void Definir_w(double);
      void back_prop_score(double);
      void Definir_nb_visite(double);
      void noeud_visite(); //Pour la back_propagation
      void Definir_icl(double);
      void Definir_icc(double);
      void Definir_couleur(int);
      void Definir_value_initiale();

        // Accesseurs:
        int Lire_Niveau() const;
        int Lire_nb_fils() const;
      int Lire_couleur() const;
      double Lire_w() const;
      double Lire_nb_visite() const;
      double Lire_icl() const;
      double Lire_icc() const;

      //UCB1:
      double calcul_UCB1(double, double);
      double Lire_value();

        // Affichage:
        void Afficher_Noeud();
        void Afficher_Arbre();

        // Gestion des noeuds fils:
        void Creer_Noeuds_Fils(matrice&, int);
      void Ajouter_Noeud(matrice&, int);
        void Supprimer_Noeuds_Fils();
      void Ajouter_Noeud_Fils(matrice&, int, Noeud);

      //Autre
      bool est_une_feuille();
      bool est_la_racine();
      int calcul_meilleur_indice_fils_UCB1(double, double);
      int retourne_meilleur_indice_fils_UCB1();
      int ind_noeud_fils_coup_corresp(int, int);
      void back_propagation(double, Noeud*, double, double);


};

// Méthodes déportées de la classe Noeud:

// Définit le niveau du noeud:
void Noeud::Definir_Niveau(int _Niveau) {
     Niveau=_Niveau;
}


// Retourne le niveau du noeud:
int Noeud::Lire_Niveau() const {
     return Niveau;
}

// Calcule le niveau d'un noeud dans l'arbre: (méthode descendante)
void Noeud::Calculer_Niveau() {
     if(Pere!=pN_NULL) {
          Niveau=Pere->Niveau+1;
     } else {
          Niveau=0;
     }
```

```
}

//Méthodes liées au w
void Noeud::Definir_w(double _w) {
    w = _w;
}

double Noeud::Lire_w() const {
    return w;
}

void Noeud::back_prop_score(double gagnant) {
        if(couleur == gagnant) w+=1; //On a gagné une partie
}

void back_propagation(double gagnant, Noeud* pointeur, double N, double c) {
    pointeur->noeud_visite();
    pointeur->back_prop_score(gagnant);
    pointeur->calcul_UCB1(N, c);
    while(!(pointeur->est_la_racine())) {
        pointeur = pointeur->Pere;              //Si on met cette ligne à la fin, la
racine ne sera pas mise à jour
        pointeur->noeud_visite();
        pointeur->back_prop_score(gagnant);
        pointeur->calcul_UCB1(N, c);
    }
}


//Méthodes liées au n
void Noeud::Definir_nb_visite(double _nb_visite) {
    nb_visite = _nb_visite;
}

double Noeud::Lire_nb_visite() const {
    return nb_visite;
}

//Cette fonction va permettre d'incrémenter de 1 le nombre de visite du noeud
void Noeud::noeud_visite() {
    nb_visite+=1;
}

//Méthodes liées à la couleur
void Noeud::Definir_couleur(int _couleur) {
    couleur = _couleur;
}

int Noeud::Lire_couleur() const {
    return couleur;
}

//Permet de calculer la valeur UCB1
double Noeud::calcul_UCB1(double N, double c) {
    if(nb_visite!=0) value = w/nb_visite + c*sqrt(log(N)/nb_visite);
    else value = 1e7; //Valeur élevée qui résulte de la division par zero quand le noeud
n'a pas été visité. La valeur UCB1 = Inf
    return value;
}

double Noeud::Lire_value() {
    return value;
}

void Noeud::Definir_value_initiale() {
    if(Lire_value() == 0) value = 1e7;
}

//Pour les indices coups
void Noeud::Definir_icl(double icl) {
    indice_coup_l = icl;
```

```
}

void Noeud::Definir_icc(double icc) {
    indice_coup_c = icc;
}

double Noeud::Lire_icl() const {
    return indice_coup_l;
}

double Noeud::Lire_icc() const {
    return indice_coup_c;
}

// Définit le nombre de noeuds fils:
void Noeud::Definir_nb_fils(int _nb_fils) {
     nb_fils=_nb_fils;
}

// Retourne le nombre de noeuds fils:
int Noeud::Lire_nb_fils() const {
     return nb_fils;
}



//---------------------------------//

// Affiche nbCar fois le caractère Car:
void Barre(char Car, int nbCar) {
     for(int i=0; i<nbCar; i++) {
          cout << Car;
     }
}

// Affiche le noeud:
void Noeud::Afficher_Noeud() {
     Barre(' ',2*Niveau);
     cout << "+ Noeud (" << this << ") :" << endl;
     Barre(' ',2*Niveau);
    cout << " - Pere = " << Pere << endl;
    Barre(' ',2*Niveau);
     cout << " - Niveau = " << Niveau << endl;
     Barre(' ',2*Niveau);
     cout << " - nb_fils = " << nb_fils << endl;
    Barre(' ', 2*Niveau);
    cout << " - w = " << w << endl;
    Barre(' ', 2*Niveau);
    cout << " - nb_visite = " << nb_visite << endl;
    Barre(' ', 2*Niveau);
    cout << " - value UCB1 = " << value << endl;
    Barre(' ', 2*Niveau);
    cout << " - icl = " << indice_coup_l << endl;
    Barre(' ', 2*Niveau);
    cout << " - icc = " << indice_coup_c << endl;
    Barre(' ',2*Niveau);
    cout << " - couleur = " << couleur << endl;
     Barre(' ',2*Niveau);
     cout << " - Fils[" << nb_fils  << "] = { ";
     for(int i=0; i<nb_fils; i++) {
          if(i>0) cout << ", ";
          cout << Fils[i];
     }
     cout << " }" << endl;
     cout << endl;
}

// Affiche le noeud et le sous-arbre découlant du noeud: (méthode récursive!)
void Noeud::Afficher_Arbre() {
     Afficher_Noeud();
     for(int i=0; i<nb_fils; i++) {
```

```
            Fils[i]->Afficher_Arbre();
      }
}

//---------------------------------//

// Initialise le tableau des Fils pour n noeuds fils: (supprime les noeuds fils
existants)
// Est utilisée dans la fonction de Creer_Noeud_fils
void Noeud::Initialiser_Fils(int _nb_fils) {
      if(Fils!=ppN_NULL) {
            Supprimer_Noeuds_Fils();
      }
      nb_fils=_nb_fils;
      Fils = new pNoeud[nb_fils];
}




// Supprime le tableau des Fils: (ne supprime pas les noeuds fils)
void Noeud::Supprimer_Fils() {
      if(Fils!=ppN_NULL) {
            delete [] Fils;
      }
      nb_fils=0;
      Fils=ppN_NULL;
}

// Crée n noeuds fils au noeud courant:
//La fonction initialiser_fils suppriment tous les fils existants. C'est pas grave. A
priori, une fois qu'un noeud a tous ces
//Noeuds fils de créé on ne devra plus en créer d'autres (unicité du chemin pour arriver
au noeud)
/*void Noeud::Creer_Noeuds_Fils(int _nb_fils) {
     int stock_nb_fils = nb_fils;
      Initialiser_Fils(_nb_fils);
      for(int i=stock_nb_fils; i<_nb_fils; i++) {
            Fils[i] = new Noeud;
            Fils[i]->Pere=this;
            Fils[i]->Calculer_Niveau();
      }
}*/

//Ici la couleur est la dernière couleur ayant joué pour arriver à la configuration de
la partie dans matrice_coups
void Noeud::Creer_Noeuds_Fils(matrice& matrice_coups, int couleur) {
     int stock_nb_fils = matrice_coups.dim1();
     Initialiser_Fils(stock_nb_fils);
     for(int i=0; i<stock_nb_fils; i++) {
         Fils[i] = new Noeud;
         Fils[i]->Pere=this;
         Fils[i]->Calculer_Niveau();
         Fils[i]->Definir_icl(matrice_coups(i,0));
         Fils[i]->Definir_icc(matrice_coups(i,1));
         Fils[i]->Definir_couleur(couleur);
         Fils[i]->Definir_value_initiale();
     }
}

void Noeud::Ajouter_Noeud_Fils(matrice& matrice_coups, int couleur, Noeud noeud_courant)
{
     int nb_coups = matrice_coups.dim1();
     if(nb_coups!=0) {
         srand(time(NULL));
         int coups_i = rand() % nb_coups;
         Initialiser_Fils(1);
         int nb_fils_noeud_courant = noeud_courant.Lire_nb_fils();
         Fils[nb_fils_noeud_courant-1] = new Noeud;
         Fils[nb_fils_noeud_courant-1]->Calculer_Niveau();
         Fils[nb_fils_noeud_courant-1]->Definir_icl(matrice_coups(coups_i,0));
         Fils[nb_fils_noeud_courant-1]->Definir_icc(matrice_coups(coups_i,1));
```

```
        Fils[nb_fils_noeud_courant-1]->Definir_couleur(couleur);
    }
}
// Supprime les noeuds enfants du noeud courant: (méthode récursive! cf.: déstructeur de
la classe Noeud)
// Cette méthode supprime le sous-arbre de noeuds découlant du noeud courant
void Noeud::Supprimer_Noeuds_Fils() {
     for(int i=0; i<nb_fils; i++) {
          delete Fils[i];// Lors de la destruction d'un noeud, cette fonction est
appellée par le déstructeur du noeud
     }
     Supprimer_Fils();
}

bool Noeud::est_une_feuille() {
    if (nb_fils==0) return true;
    else return false;
}

bool Noeud::est_la_racine() {
    bool sortie = false;
    if(Pere == pN_NULL) sortie = true;
    return sortie;
}

//Retourne l'indice du fils qui a le meilleur UCB1 calculé dans cette meme fonction
int Noeud :: calcul_meilleur_indice_fils_UCB1(double N, double c) {
    vecteur val_UCB1(Lire_nb_fils());
    for (int i=0; i<Lire_nb_fils(); i++) {
        Fils[i]->calcul_UCB1(N, c);
        val_UCB1(i) = Fils[i]->Lire_value();
    }
    int sortie = 0;
    sortie = val_UCB1.indice_du_max();
    return sortie;
}

int Noeud :: retourne_meilleur_indice_fils_UCB1() {
    vecteur val_UCB1(Lire_nb_fils());
    for (int i=0; i<Lire_nb_fils(); i++) {
        val_UCB1(i) = Fils[i]->Lire_value();
    }
    int sortie = 0;
    sortie = val_UCB1.indice_du_max();
    return sortie;
}

int Noeud :: ind_noeud_fils_coup_corresp(int icl, int icc) {
    for(int i=0; i<nb_fils; i++) {
        if((Fils[i]->Lire_icl() == icl) && (Fils[i]->Lire_icc()==icc)) {
            return i;
        }
    }
}
```