UNIVERSITÉ
CAEN
NORMANDIE

- Rapport -
Projet 2
Jeu de Hex et bandits manchots

Amadou Bah
Celestin Mireux
Daouda TRAORE
Junior MISSOUP

L3 Informatique 2022-2023 Université Caen Basse-Normandie.

Table des matières

1 Introduction

Pour l'évaluation de la matière projet2 qui a pour but de nous inculquer les meilleures attitudes et compétences de programmation il nous a été proposé multiples sujets pour nous évaluer afin de mettre en pratique nos connaissances et nos compétences. Parmi tous ces sujets celui qui nous ont le plus Intéressé fut la coloration IA : Jeu d'Hex et bandits manchots. Ce jeu n'est pas très different de jeu qu'on a eu à réaliser mais la stratégie appliquée a ce jeu pour gagner une partie est un peu de différente, quand on parle de stratégie du point de vue d'une machine c'est l'algorithme qu'elle va utilider pour gagner une partie. On sait que les jeux extensifs, comme hex, le go ou les échecs, peuvent se représenter sous la forme d'arbre, et que l'essentiel du travail pour gagner est de prédire un score pour chaque nœud. Pour faire ça on analyse toutes les possibilités du jeu, ou on utilise des fonctions d'évaluation qui permet d'approximer l'état du jeu. Mais c'est méthodes peuvent avoir des limites.

Parmi ses limites, on a vu que certains jeux n'ont pas de fonctions d'évaluation évidentes et quand il y a trop de coups possibles, l'arbre de recherche devient beaucoup trop grand ce qui rend le temps de calcul d'un meilleur coup très long. C'est pourquoi pour ce projet, on va donc essayer de voir une autre méthode qui permette de traiter ces deux cas. Mais on est dans une situation pour le moins complexe : quelle que soit la profondeur à laquelle on ira chercher notre meilleur, nous n'avons pas de fonction d'évaluation, il nous sera impossible d'évaluer notre plateau.

Il va donc nous falloir trouver une autre solution. Il existe déjà un algorithme intéressant pour faire des estimations sur la valeur d'un coup sans avoir besoin d'évaluer le plateau, qui s'appelle **Monte-Carlo Tree search**.

Ce projet nous a donc amené à travailler sur des connaissances qui sont nouvelles pour nous, que ce soit les algorithmes de résolution ou encore des choses complexes du développement des applications. Ce rapport a pour objectif de présenter comment nous avons travaillé et quels ont été les multiples obstacles que nous avons dû faire face.
Le présent rapport est organisé en trois grandes parties. La première est consacrée à la présentation du jeu, puis nous parlerons de la répartition des

tâches, du développement du moteur, de l'implementation de MCTS et enfin de la création de l'interface graphique et des problèmes rencontrés.

2 Histoire et règle de l'hex

Inventé en 1942 par le poète et physicien danois Piet Hein, le jeu d'Hex est un jeu de stratégie abstrait, comme les échecs et le Go, par exemple.

Il est redécouvert, en 1948, par le célèbre économiste et mathématicien américain John Forbes Nash. Depuis, il a fasciné de nombreux amateurs de jeux combinatoires.

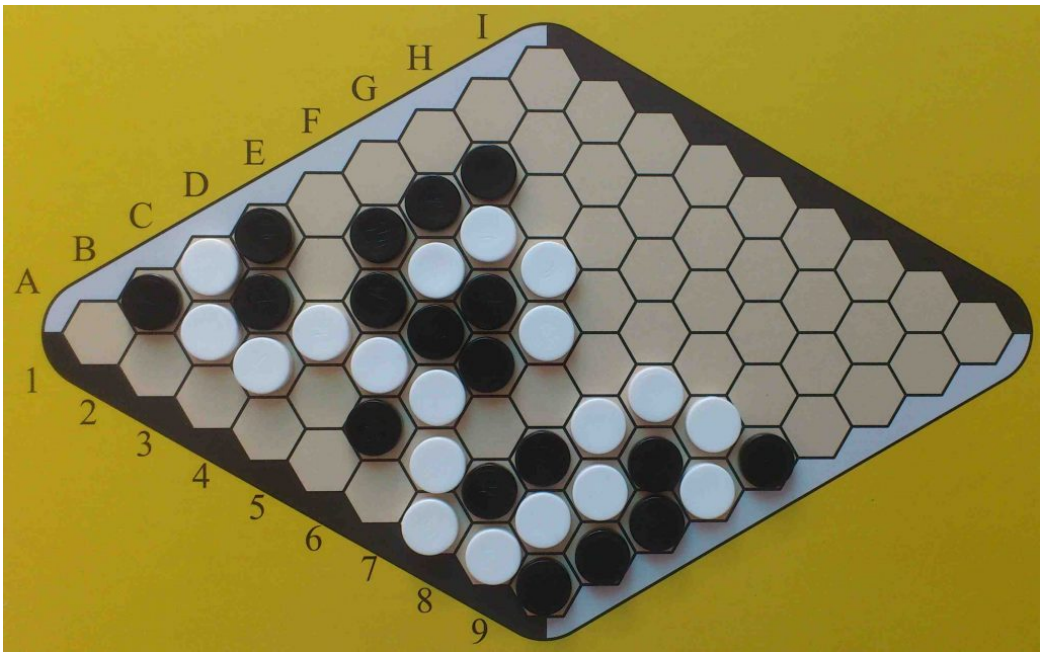


Figure 1: Hex

Le premier joueur utilise des pièces noires et le second joueur utilise des pièces blanches. Le but du premier joueur est de former une connexion avec les pièces du bord supérieur au bord inférieur, tandis que le second joueur essaiera de former une connexion entre le bord gauche et le bord droit.

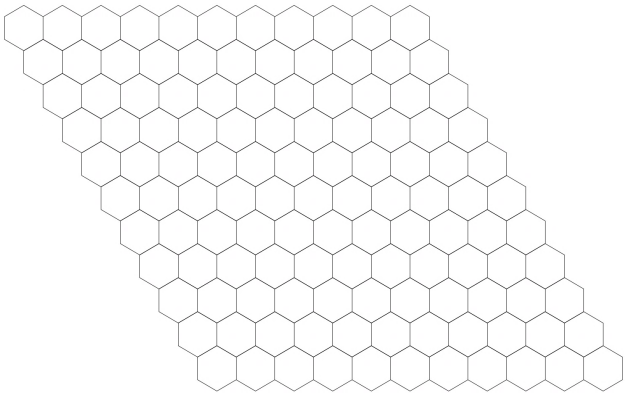


Figure 2: Tableau hexagonal 11x11 vide

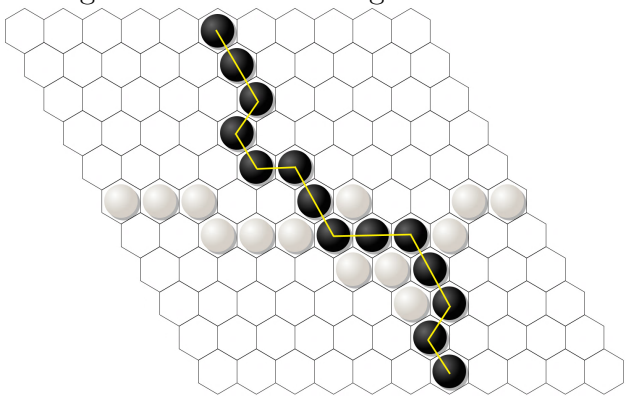


Figure 3: Exemple d'une partie

Ici on peut observer que c'est le joueur avec les pièces noirs qui a gagné la partie.

Une partie d'Hex ne peut jamais se terminer par un match nul. En tant que tel, le seul moyen de bloquer complètement l'adversaire est de former une connexion avec ses propres pièces. La recherche d'une où des stratégies gagnantes deviennent de plus en plus difficiles lorsque la taille du plateau augmente jusqu'à des plateaux, en général la taille d'un plateau est de 14 x

14. En raison de l'avantage significatif que le premier joueur à placer sa pièce a, le jeu est couramment joué avec la règle de l'échange. Cette règle stipule qu'après que le joueur Noir est placé sa première pièce, le joueur Blanc peut choisir d'échanger cette pièce contre une pièce blanche ou pas. Le jeu se poursuit alors avec les noirs, qui sont devenus le deuxième joueur.

3 Organisation

Pour la répartition des tâches elle a été faite sans véritable organisation mais nous pouvons considérons la répartition suivante:
celestion Mireux :

- Partie Graphique
- rapport

Daouda TRAORE :

- Réalisation du moteur
- Partie Graphique
- Rapport

Amadou Bah :

- Réalisation du moteur
- Tests
- Rapport

Junior MISSOUP :

- Rapport

Pour la prochaine fois une meilleure organisation sera notre premier objectif, mais étant des amateurs nous avons un petit peu avancé à l'aveuglette.

3.1 Gestion du projet

Afin de faciliter la communication et le bon déroulement de la conception de notre jeu, divers moyens ont été mis en oeuvre.

3.1.1 Hébergement du code

Afin de faciliter la gestion du projet, nous avons utilisé à la fois la Forge d'Unicaen qui permet de créer et d'administrer des dépôts sous Git très facilement par l'intermédiaire d'une interface web. D'ailleurs pour tous les membres l'utilisation de GIT fut une première, donc un temps d'acclimatisation et d'apprentissage nous a été nécessaire. D'autres fonctionnalités sont disponibles sur cette plateforme comme une gestion des permissions, une visualisation des différents commits, la visualisation de l'activité du projet, etc. L'utilisation supplémentaire de Github permettait de centraliser les projets du cursus de la licence sur une seule plateforme(plus à but personnel).

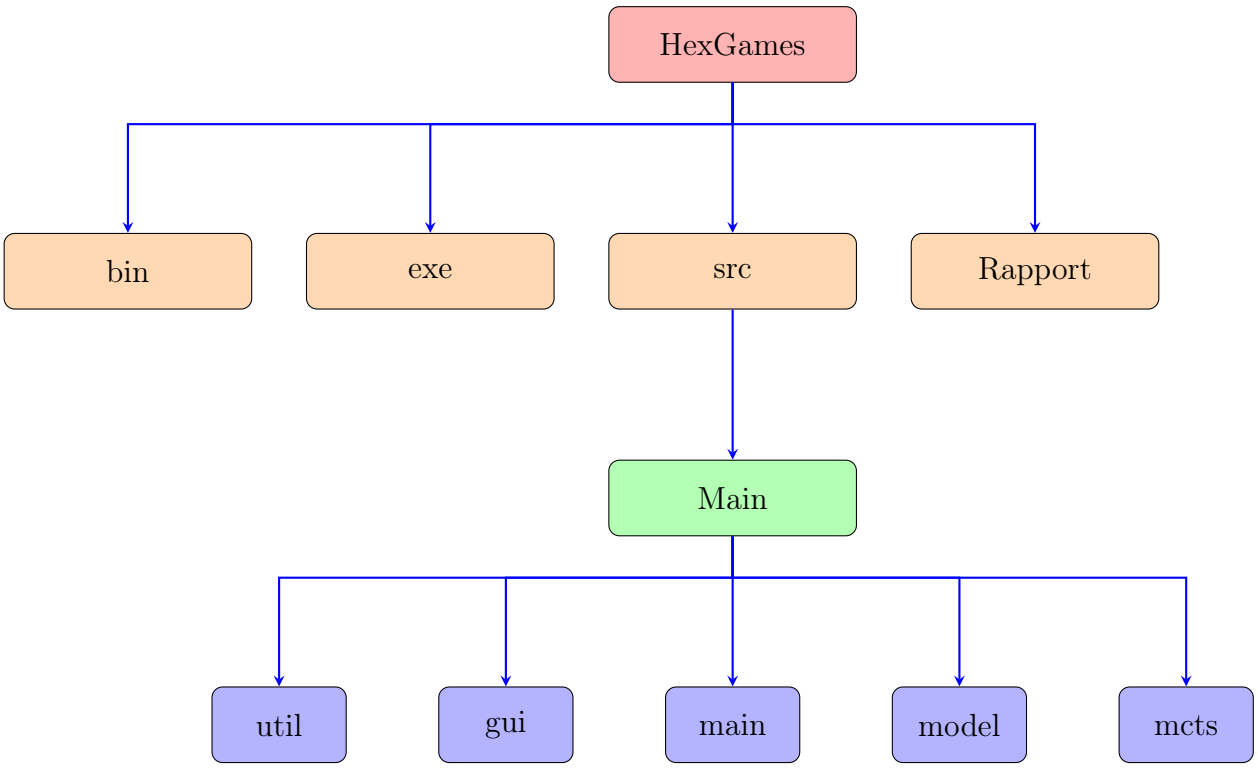
3.1.2 Gestionnaire de version

Nous avons utilisé un gestionnaire de version afin de permettre la centralisation du code et rendre le travail en équipe de manière bien plus efficace. Nous avons opté pour Git comme gestionnaire de fichiers sans raison particulière, qui en fin de compte s'avérait être un bon choix. Sur Git l'utilisation des branches est très facile, elle permet de faire des commits sans pour autant être connecté sur le serveur. Cela permet de faire plus de commits, qui sont enregistrés localement et de les envoyer sur le serveur en une seule fois, au moment où nous sommes sûrs que la fonctionnalité ajoutée fonctionne. Git permet également de transférer facilement son code vers un autre hébergeur en ajoutant simplement une "route" (remote), tout en conservant la totalité des commits réalisés.

4 Conception

4.1 Arborescence du projet

Jeu de Hex et bandits manchots :
Réprésente l'espace de travail.
Rapport:
Le répertoire contenant les différents éléments du rapport.
exe:
Contient le jar.
bin:
Répertoire contenant les .class.
Au fur et à mesure de l'avancement le projet a pris pour arborescence la suivante 4.1 :



4.2 Architecture du programme

4.2.1 Diaggramme des packages

Le diagramme suivant est sa forme finale prise une fois que tous les éléments ont été correctement implémenté.

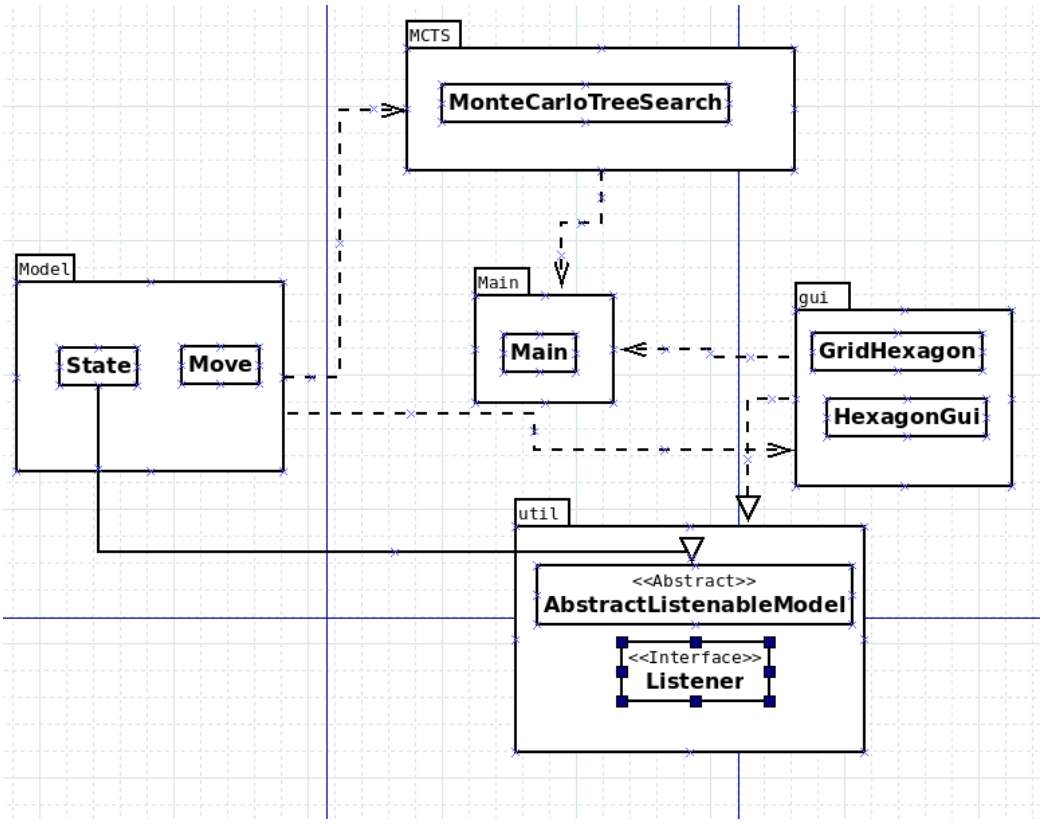


Figure 4: Diagramme UML

Comme nous pouvons le voir le code source est décomposé en 5 packages avec le pattern **MVC** en plus pour une plus facile implémentation de l'interface graphique.

- Le package **model** constitue le moteur du jeu .

- Le package **gui** constitue la partie **vue** + **controlleur**
- Le package **Mcts** contient le l'agorithme de MCTS.

5 Réalisation du Moteur

La réalisation du moteur était la première étape que nous avions entamée et la dernière à être achevée car jusqu'au bout de multiples bugs ont dû être corrigé et de nouvelles fonctionnalités ont été ajoutées afin que les autres parties puissent fonctionner correctement.

5.1 Réalisation du Board

5.1.1 UML des classes

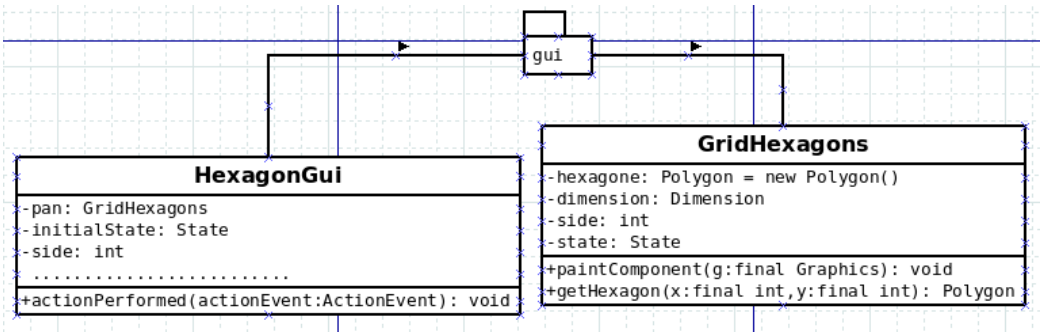
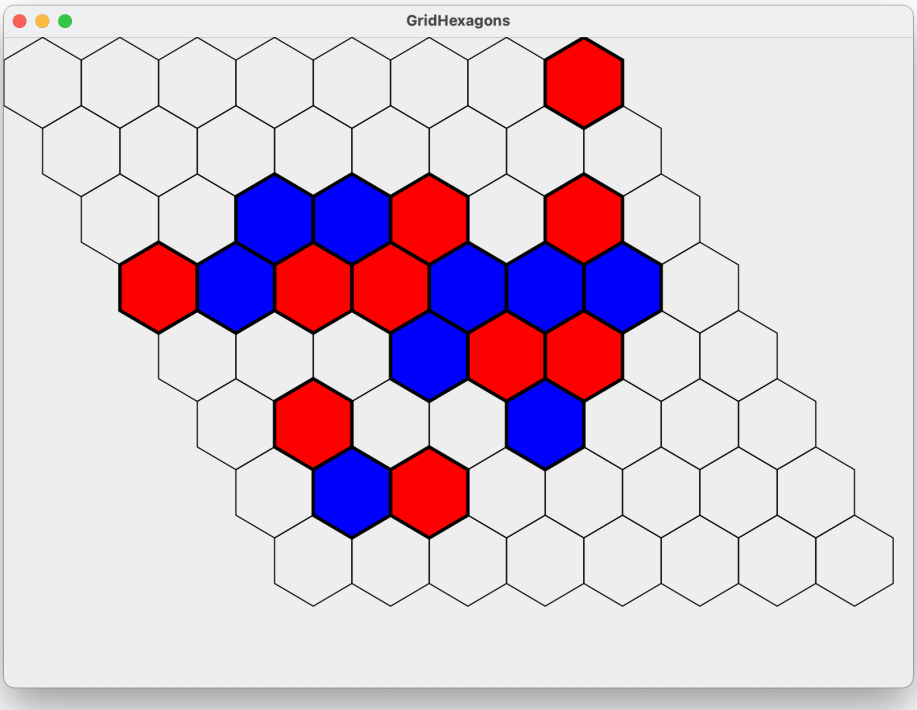


Figure 5: Diagramme UML du package tab



5.1.2 Création du Plateau

Pour concevoir le jeu de Hex, la première question que nous nous sommes posé à été: **comment devons-nous créer le plateau?**. Le plateau de Hex est un plateau hexagonal dont les cases sont hexagonales
Pour créer le plateau, on avait plusieurs d'idées, on voulait d'abord partir sur une représentation en 3 dimensions qui était un peu complexe à comprendre. Après mur réflexion on s'est décidé à faire un tableau 2d avec une fonction de voisinage pour "transformer" les cases carrées des tableaux qui ont 4 voisins en case hexagonale qui en ont six.

5.2 Fonctionnement du moteur

Une fois le plateau fait, nous nous sommes penché sur le coeur de l'application c'est-à-dire le moteur du jeu dont la majorité des fonctionnalités appartiennent à la classe **State** qui également joue le rôle de modèle dans l'architecture MVC.

5.2.1 Description

5.2.2 UML des classes

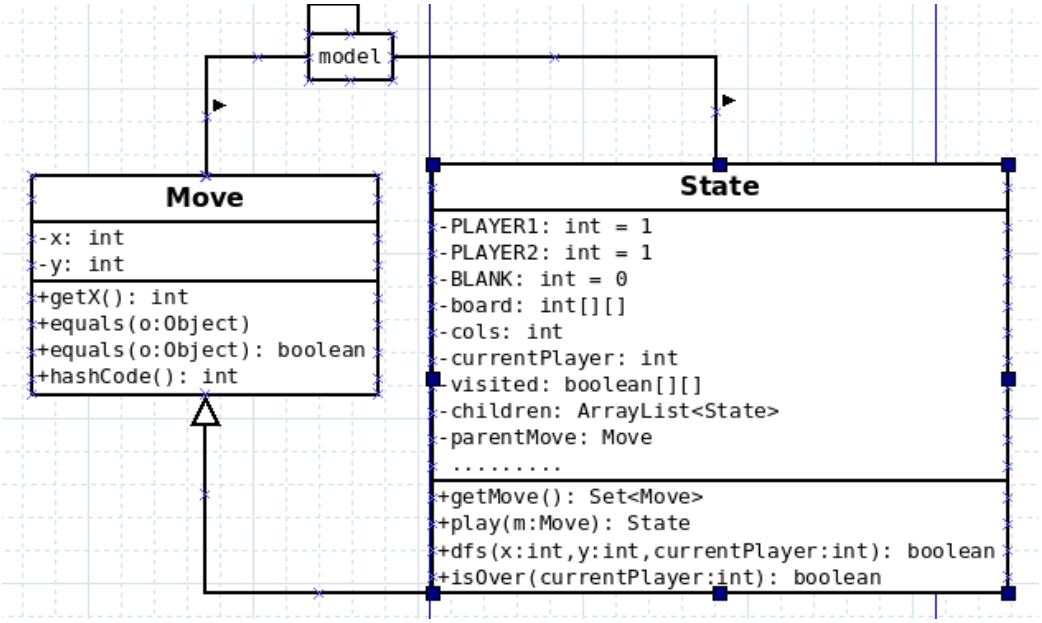


Figure 6: Diagramme UML du package tab

La classe **State** donne l'état du jeu à tout instant donné. Elle initialise le jeu en créant un plateau vide. Dans la fenêtre, nous faisons apparaitre un plateau de jeu 11 x 11. Pour pouvoir placer les pions, nous avons développé certaines fonctions qui permettent de placer le pion dans une case. Concernant l'IA, pour pouvoir jouer, le programme fournit les coordonnées du coup qu'il 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 11 x 11 par une matrice à 11 lignes et 11 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 Rouge et les pions Bleus 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 bleu se trouve sur la case (3,4). Ainsi, Pour pouvoir jouer, le joueur recupere un coup dans un set de moves(**classe Move**) . 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 via la classe **MonteCarloTreeSearch** plusieurs fois (par exemple 1000 fois pour qu'il y ait assez de simulations de parties pour pouvoir donner un coup suffisamment efficace).
Cependant pour déterminer si un joueur a gagné on utilise une recherche en profondeur, contrairement au parcours en largeur, lorsque l'on fait un parcours en profondeur à partir d'une case donnée, on tente d'avancer le plus loin possible dans le jeu, et ce n'est que lorsque toutes les possibilités de progression sont bloquées que l'on revient (étape de backtrack) pour explorer un nouveau chemin ou une nouvelle chaîne. Le parcours en profondeur correspond aussi à l'exploration d'un labyrinthe. Les applications de ce parcours sont peut-être moins évidentes que pour le parcours en largeur, mais le parcours en profondeur permet de résoudre efficacement des problèmes plus difficiles.Voici l'algo qu'on a implémenté

Algorithm 1: DFS pour le jeu Hex

```
   Input: x, y : position du joueur courant
   Input: currentPlayer : joueur courant
   Output: true si le joueur courant a gagné, sinon false
 1 if currentPlayer == PLAYER1 and x == rows - 1 then
 2 |   return true;
 3 end if
 4 else if currentPlayer == PLAYER2 and y == cols - 1 then
 5 |   return true;
 6 end if
 7 visited[x][y] = true;
 8 if GetCell(x - 1, y) == currentPlayer and visited[x - 1][y] == false
    then
 9 |   if dfs(x - 1, y, currentPlayer) == true then
10 |   |   return true;
11 |   end if
12 end if
13 if GetCell(x - 1, y + 1) == currentPlayer and visited[x - 1][y + 1]
    == false then
14 |   if dfs(x - 1, y + 1, currentPlayer) == true then
15 |   |   return true;
16 |   end if
17 end if
18 if GetCell(x, y - 1) == currentPlayer and visited[x][y - 1] == false
    then
19 |   if dfs(x, y - 1, currentPlayer) == true then
20 |   |   return true;
21 |   end if
22 end if
23 if GetCell(x, y + 1) == currentPlayer and visited[x][y + 1] ==
    false then
24 |   if dfs(x, y + 1, currentPlayer) == true then
25 |   |   return true;
26 |   end if
27 end if
28 if GetCell(x + 1, y - 1) == currentPlayer and visited[x + 1][y - 1]
    == false then
29 |   if dfs(x + 1, y - 1, currentPlayer) == true then
30 |   |   return true;
31 |   end if
32 end if
33 if GetCell(x + 1, y) == currentPlayer and visited[x + 1][y] ==
    false then
34 |   if dfs(x + 1, y, currentPlayer) == true then
35 |   |   return true;
   |   end if
```

cette méthode, est une fonction récursive qui utilise la recherche en profondeur pour explorer les cellules adjacentes à une cellule donnée et vérifie si elles sont de la même couleur que le joueur en cours. La méthode prend deux paramètres: les coordonnées de la cellule actuelle (x, y) et l'identifiant du joueur actuel (PLAYER1 ou PLAYER2). La méthode marque chaque cellule visitée en utilisant un tableau **visited** et s'arrête et renvoie **true** si elle trouve une cellule qui correspond à la couleur du joueur qui forme une connexion ininterrompue jusqu'à la rangée opposée.

Algorithm 2: fonction isOver

```
   Input: currentPlayer : joueur courant
   Output: true si le joueur courant a gagné, sinon false
 1 boolean gameOver = false;
 2 int y = 0, x = 0;
 3 int end = (currentPlayer == PLAYER1 ? cols : rows);
 4 for int i = 0; i < cols; i++ do
 5 |   for int j = 0; j < rows; j++ do
 6 |   |   visited[i][j] = false;
 7 |   end for
 8 end for
 9 for int i = 0; i < end; ++i do
10 |   if GetCell(x, y) == currentPlayer and dfs(x, y, currentPlayer)
   |     == true then
11 |   |   gameOver = true;
12 |   |   break;
13 |   end if
14 |   if currentPlayer == PLAYER1 then
15 |   |   y++;
16 |   end if
17 |   else
18 |   |   x++;
19 |   end if
20 end for
21 return gameOver;
```

La méthode, **isOver**, prend également un paramètre **currentPlayer** et vérifie si le joueur a gagné la partie en appelant la méthode **dfs** pour chaque cellule de la première colonne ou ligne en fonction du joueur actuel. Elle parcourt chaque cellule de la première colonne ou ligne selon le joueur actuel et vérifie si elle est de la même couleur que le joueur. Si la méthode **dfs** renvoie **true** pour l'une de ces cellules, cela signifie que le joueur a gagné et la méthode renvoie **true**. Sinon, la méthode renvoie **false**.

6 Recherche arborescente de Monte Carlo

6.1 Recherche arborescente

Dans un plateau Hex de taille n × n, la configuration de départ vide a n × n mouvements possibles. Chaque coup donne une nouvelle configuration unique des pièces sur l'échiquier. Ces configurations peuvent être organisées hiérarchiquement dans une structure arborescente. La configuration de départ était le nœud racine, chaque position de l'échiquier atteignable à partir de cette position sera des enfants rattachés au nœud. Cet arbre se poursuit jusqu'aux nœuds terminaux, c'est-à-dire les positions où le jeu est terminé. La figure ci-dessous montre comment un arbre de 2 × 2 Hex qui se développe.

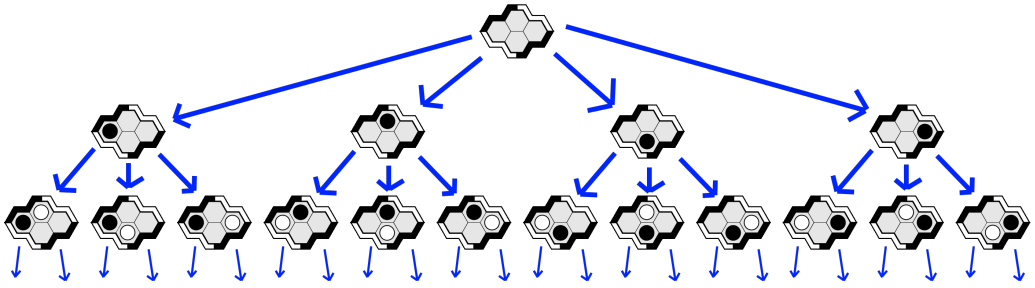


Figure 7: Structure arborescente 2×2 Hex

La recherche arborescente est une méthode de recherche d'un arbre à partir d'une position de départ donnée, afin de trouver des positions de conseil bénéfiques plus bas dans l'arbre. Certaines stratégies de recherche arborescente

pour explorer l'arbre sont la recherche en largeur d'abord et la recherche en profondeur d'abord.

6.2 Algorithme de recherche arborescente de Monte Carlo

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 :

$$UCB1(N, c) = \frac{\omega}{n} + c\sqrt{\frac{\ln(N)}{n}} \tag{1}$$

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 là 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 ($\omega_r acine$ est incrémenté de 1 dans tous les cas). a 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 :

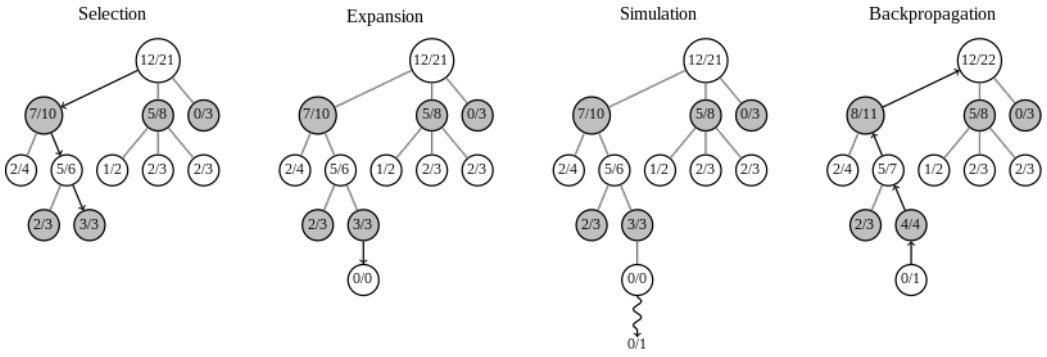


Figure 8: Formule de calcul du coût

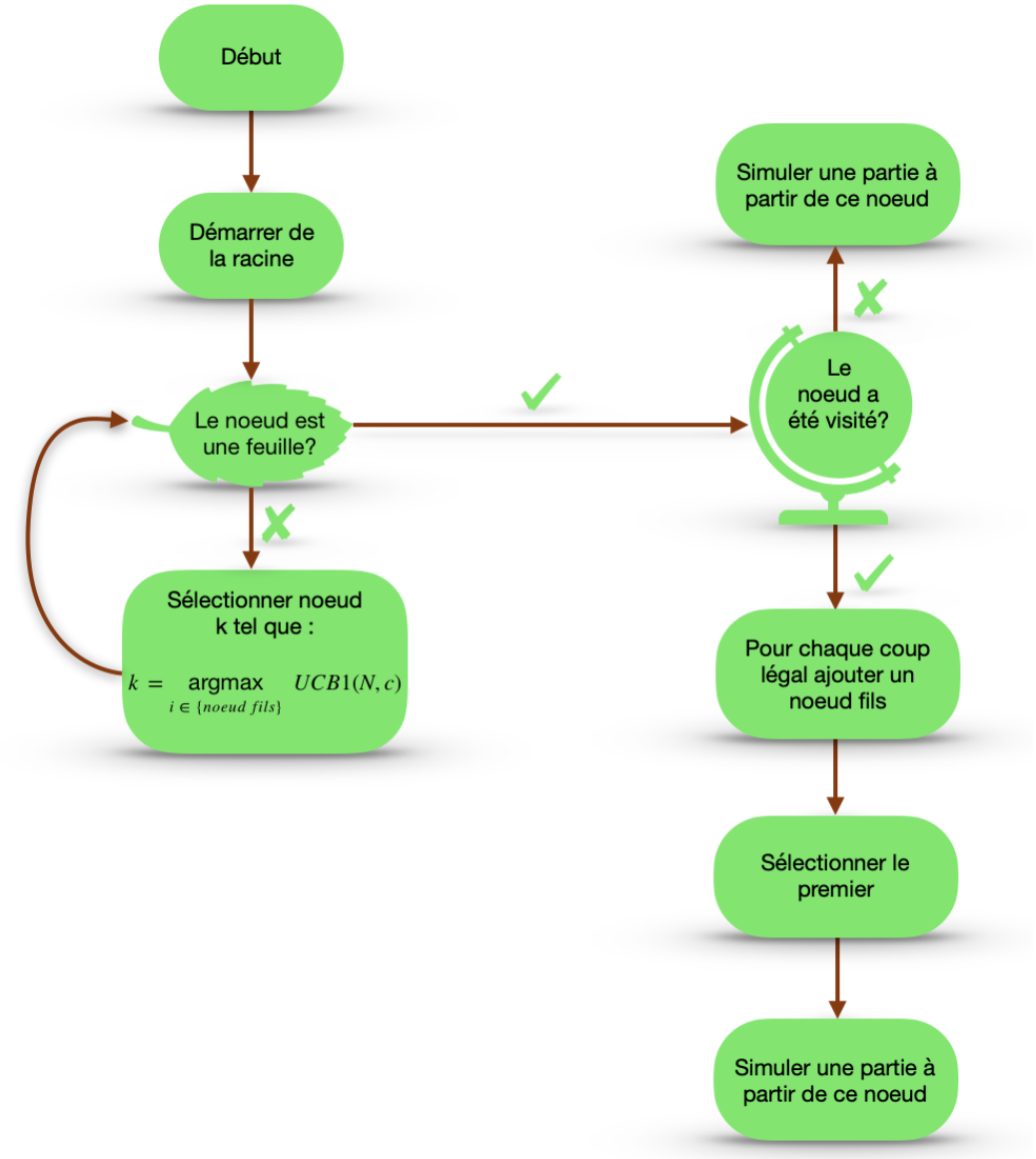


Figure 9: Schéma de MCTS

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.

6.3 Exemple

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

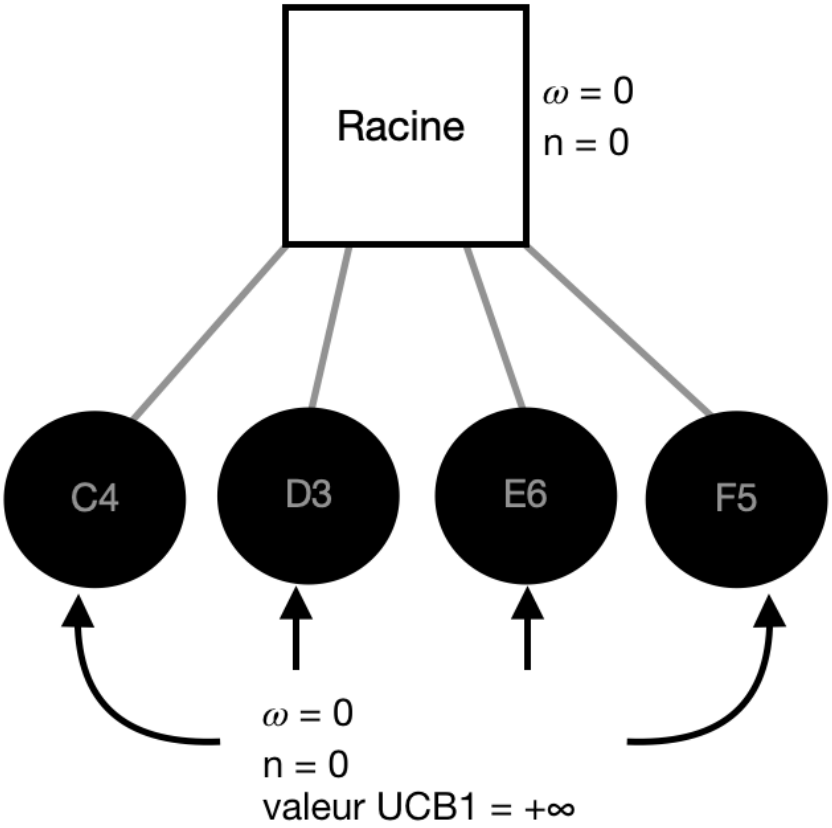


Figure 10: Arbre de début de jeu

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 meme 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.

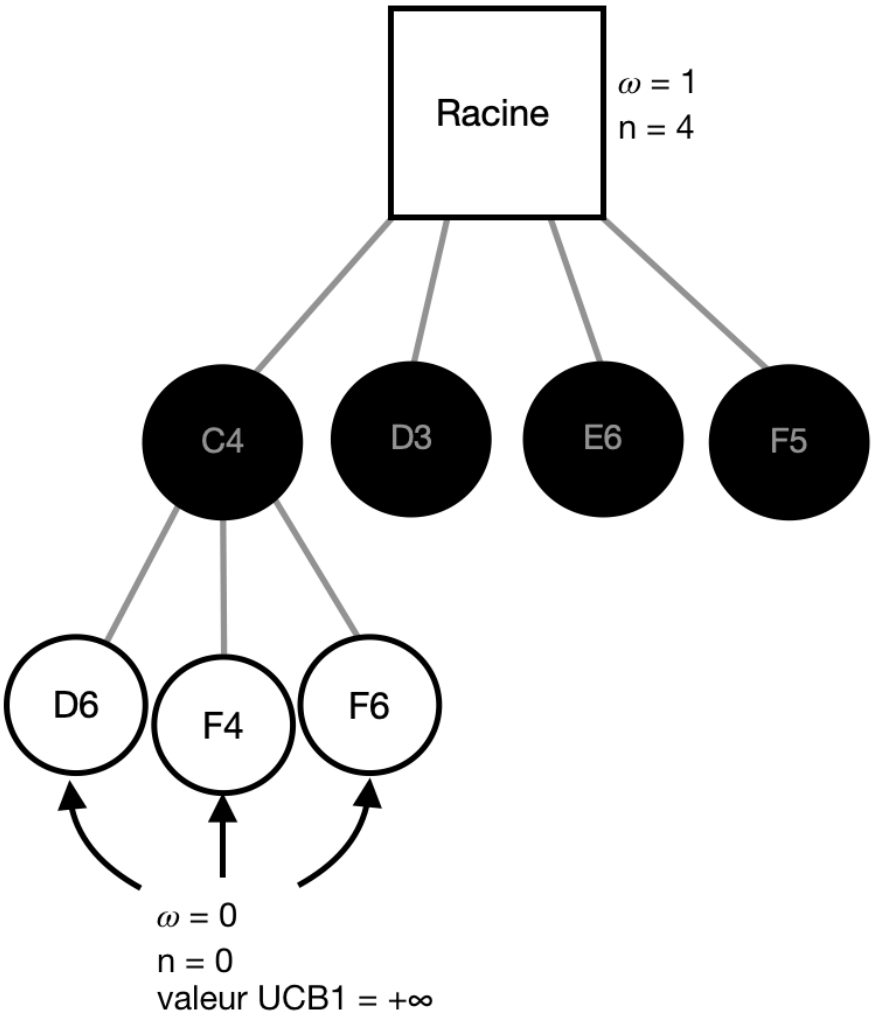


Figure 11: cinquième itération de MCTS

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 aboutit à une victoire pour les blancs alors $\omega_{D6} = 1 et n_{D6} = 1$.Et ainsi de suite.

6.3.1 UML

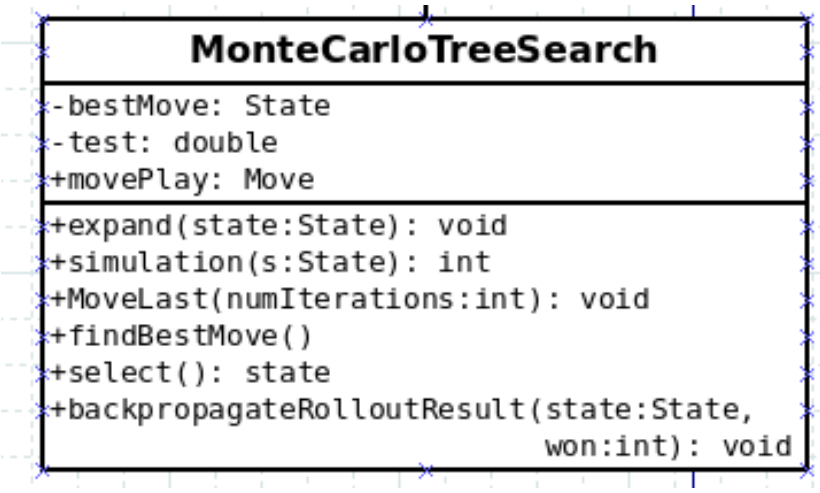


Figure 12: UML du package Algo

6.4 Implémentation par rapport à notre Jeu de Hex

L'implémentation de MCTS s'est faite en 4 étapes comme nous l'avions précisé un peu plus tôt, mais on a une cinquième étape qui nous sert de d'un genre de fonction auxiliaire .cette fonction utilise nos 4 etapes pour simuler plusieurs parties aléatoires à partir du nœud racine et utilise l'information obtenue pour déterminer le meilleur coup à jouer. Les algorithmes suivants sont directement inspiré de ceux présents sur internet, les refernces sont bien données.

Algorithm 3: findBestMove(numIterations)

```
   Input: numIterations : nombre d'itérations
   Result: Le meilleur coup à jouer
 1 for i ← 1 to numIterations do
 2 |   nodePromise ← select(rootNode);
 3 |   won ← simulation(nodePromise);
 4 |   backpropagateRolloutResult(nodePromise, won);
 5 end for
 6 victories ← 0;
 7 ; for child ∈ rootNode.getChildren() do
 8 |   if child.getVictories() ≥ victories then
 9 |   |   bestMove ← child;
10 |   |   movePlay ← bestMove.getLastMove();
11 |   |   victories ← child.getVictories();
12 |   end if
13 end for
```

6.5 Expérimentation

Lors de la présentation de la méthode, nous avons précisé qu'on utilisait une heuristique qui est le UCB(Upper Confident Bounded).
Est-ce que la recherche arborescente de Monte-Carlo vaut la peine d'être implémentée?
Quels sont les paramètres de l'algorithme MCTS que nous pourrions ajuster

pour améliorer les performances de l'agent de jeu ?
Comment pourrions-nous évaluer les performances de l'agent de jeu sur le jeu de Hex ?
Est-ce plus efficace qu'un jeu aléatoire?

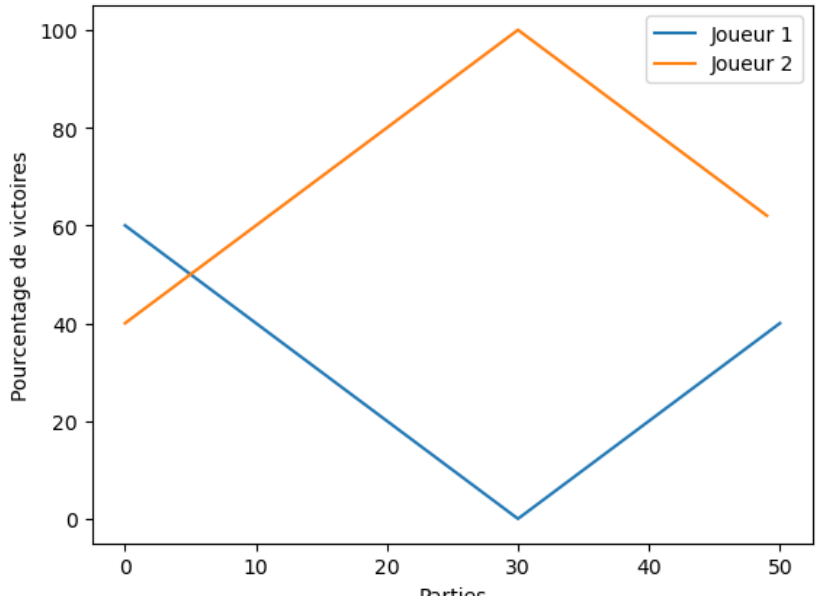


Figure 13:

Pour initialiser mon expérience nous voulions être d'accord que lorsque plus le budget est grand mieux sont les performances de l'algorithme. sur la figure ci-dessus on a le joueur 1 et le joueur 2 qui jouent avec le même budget(1000). Et on remarque ils ont presque le même taux de victoires, en effet pour le joueur 1 on remarque plus de victoires de son coté, c'est normal vu que c'est le premier qui commence à jouer à chaque fois à ne pas oublier sur le jeu d'hex le joueur qui commence en premier à un avantage.

On remarque sur la figure que le joueur 2 a gagné plus souvent que le joueur 1 lorsqu'on a augmenté la constante d'exploration du joueur 1 à 100 tandis que le joueur 2 est resté à une constante d'exploration de 2, cela peut prouver que le joueur 1 a exploré davantage de coups que le joueur 2. Bien que cela puisse sembler avantageux, explorer davantage de coups ne garantit pas toujours la victoire, car le choix des coups explorés peut également affecter le résultat

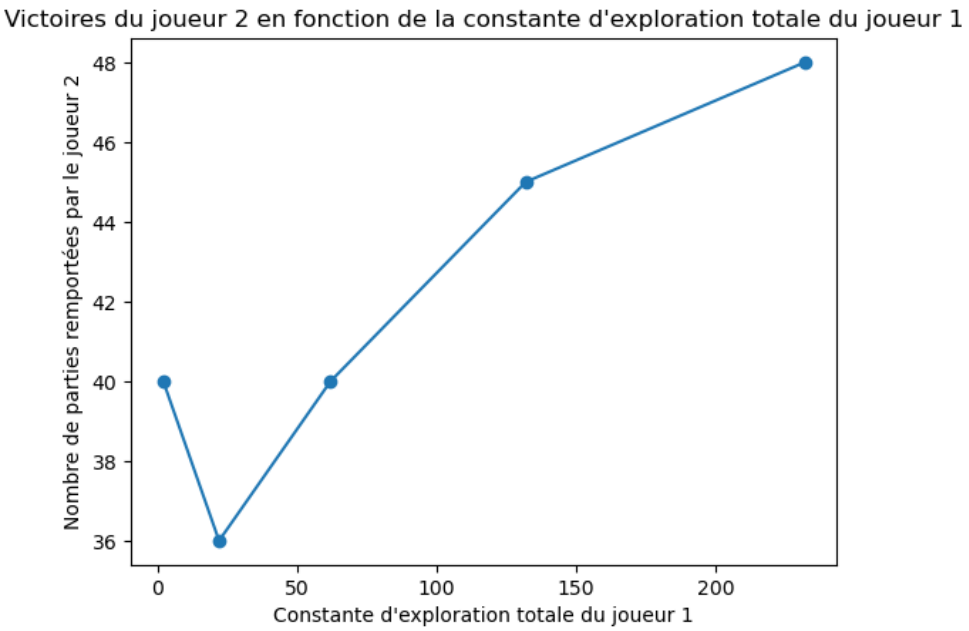


Figure 14:

du jeu.Pour c=0, on a un taux de victoire maximal pour le joueur 1, cela signifie que la recherche de l'arbre de recherche se concentrera uniquement sur l'exploitation des nœuds qui ont la meilleure estimation de la probabilité de gagner, sans prendre en compte l'exploration de nouvelles options qui pourraient être prometteuses à long terme.

6.6 Problèmes rencontés

Durant la mise en place de l'interface graphique, Cette partie du projet nous a été particulièrement difficile, nous avons trouvé en open source une classe « Hexagon » qui compose notre fichier. En effet, nous avons compris comment elle fonctionnait et nous avons alors pu l'enrichir de nos propres besoins en fonction.
Cependant dans l'implementation de l'algorithme on a eu plusieurs dysfonctionnements qui a duré plusieurs séances de Tp. Une fois de plus le problème

semblait incompréhensible car nous l'avions correctement implémenté mais le souci était chaque fois le premier joueur qui jouait gagnait quel que soit le budget attribué. Une fois de plus est venu à la rescousse notre chargé de Tp qui après avoir revu notre code nous a faits savoir que dans notre implementation les deux joueurs jouaient le même arbre de recherche et ainsi ça pouvait arriver qu'un joueur qui n'explore pas un noeud déjà visité par son adversaire.

Pour résoudre le problème notre chargé de Tp nous a exposé plusieurs idées, toutes éventuellement intéressantes mais la plupart des idées allaient nous pousser à refaire le modèle ce qui nous semblait un peu tard, il y avait pas beaucoup de temps pour terminer le projet et on s'est tourné vers une option qui était nous sauvait la vie. Cette option était de faire une copie profonde de notre state initial et de le partager aux deux joueurs et après se mettre à jour plus tard le deuxième joueur qui jouera au deuxième tour. Qui fut une première car jusque-là nous n'avions jamais considéré cette éventualité.

La liste de problèmes rencontrés durant ce projet étant extrêmement longue nous nous limiterons à ces deux qui furent le plus problématique.

7 Conclusion

7.1 Avis Général

Ce projet était très intéressant. Nous ne connaissions pas le jeu de hex auparavant et c'était intéressant de se pencher, sur les règles de ce jeu et de le recréer afin de pouvoir y jouer autrement que sous forme physique.
Ce projet nous aura permis de beaucoup apprendre et de nous améliorer.

7.1.1 Eléments à améliorer

Bien que tout ce qui a été demandé dans le sujet a été rempli, pas mal de choses restent quand même à améliorer mais faute de temps et du nombre de projets ont fini en ce fin semestre tout ce que l'ont souhaité n'a pas pu être accompli. Par exemple;

- L'implémentation de l'optimisation RAVE(Rapid Action Value Estimation).
- Amélioreration de l'interface graphique, car l'interface finale est très basique.
- Rajouter d'autres fonctionnalités comme pouvoir jouer par soi-même.
- faire des tests comparant différents types d'algorithmes.

References

[1]

[2] codingame, recherche arborescente monte-carlo. https://www.codingame.com/playgrounds/57060/mcts-tic-tac-toe-playground.

[3] Mcts. https://www.cs.cmu.edu/~katef/DeepRLFall2018/MCTS_katef.pdf.

[4] Wikipedia, hex. /https://fr.wikipedia.org/wiki/Hex/.

[5] Wikipedia, recherche arborescente monte-carlo. https://fr.wikipedia.org/wiki/Recherche_arborescente_Monte-Carlo.

[6] Tristan Cazenave and Abdallah Saffidine. Utilisation de la recherche arborescente monte-carlo au hex. *Rev. d'Intelligence Artif.*, 23(2-3):183–202, 2009.

[7] Red Blob Games. Hexagonal grids. https://www.redblobgames.com/grids/hexagons/.

8 Annexes

8.1 Code informatique

voici le code de la classe MonteCarloTreeSearch :

```
home > traore > jeu-de-hex-et-bandits-manchots > HexGames > src > mcts > MonteCarloTreeSearch.java > MonteCarloTreeSearch > select(State)
27      *
28      */
29     public State select(State state) {
30         State currentNode = state;
31         while (true) {
32             if (currentNode.isOver(currentNode.getCurrentPlayer())) {
33                 return currentNode;
34             }
35             if (currentNode.getChildren().isEmpty()) {
36                 expand(currentNode);
37                 if (currentNode.getChildren().isEmpty()) {
38                     return currentNode;
39                 }
40                 return currentNode.getChildren().get(0);
41             } else {
42                 for (State child : currentNode.getChildren()) {
43                     child.setUCTValue(test);
44                 }
45
46                 Collections.sort(currentNode.getChildren());
47                 currentNode = currentNode.getChildren().get(0);
48                 if (currentNode.getNumVisits() == 0) {
49                     return currentNode;
50                 } else {
51                     return select(currentNode);
52                 }
53             }
54         }
55
56     }
57     /*
58     *@param State s
```

```
home > traore > jeu-de-hex-et-bandits-manchots > HexGames > src > mcts > MonteCarloTreeSearch.java > MonteCarloTreeSearch > select(State)
59     * create node tree
60      */
61     public void expand(State state) {
62         Set<Move> allMovePossibles = state.getMove();
63         for (Move m : allMovePossibles) {
64             State child = state.play(m);
65             child.setCurrentPlayer(state.getCurrentPlayer() == 1 ? 2 : 1);
66             state.setChildren(child);
67
68
69         }
70     }
71
72     /*
73     *@param State s
74     * @return winner of simulation
75      */
76     public int simulation(State s) {
77         State newState = s.deepCopy();
78         if (newState.isOver(newState.getCurrentPlayer())) {
79             return newState.getCurrentPlayer();
80         }
81         while (true) {
82             Move m = newState.getRandomMove(newState.getMove());
83             if (m == null) {
84                 return newState.getCurrentPlayer();
85             }
86             newState = newState.play(m);
87             if (newState.isOver(newState.getCurrentPlayer()) == true) {
88                 return newState.getCurrentPlayer();
89             }
90             newState.setCurrentPlayer((newState.getCurrentPlayer() == 1) ? 2 : 1);
```

```
me > traore > jeu-de-hex-et-bandits-manchots > HexGames > src > mcts > MonteCarloTreeSearch.java > MonteCarloTreeSearch > select(State)
96     *@param State s
97     * @param int won
98     * update all spanning tree
99      */
100    public void backpropagateRolloutResult(State s, int won) { // backpropagate
101
102        State current = s;
103        while (current != null) {
104            current.setNumVisits(current.getNumVisits() + 1);
105            if (rootNode.getCurrentPlayer() == won) {
106                current.setVictories(current.getVictories() + 1);
107            } else current.setLosses(current.getLosses() + 1);
108            current = current.getParent();
109        }
110    }
111    /*
112    *@param the number iteration
113    * given bestMove
114     */
115    public void findBestMove(int numIterations) {
116
117        for (int i = 0; i < numIterations; i++) {
118
119            State nodePromise = select(rootNode); // selection / expansion phase
120            int won = simulation(nodePromise); // rollout phase
121            backpropagateRolloutResult(nodePromise, won); // backpropagation phase
122
123        }
124
125        double victories = 0;
126        for (State child : rootNode.getChildren()) {
127            if (child.getVictories() >= victories) {
128                bestMove = child;
129                movePlay = bestMove.geLastMove();
130                victories = child.getVictories();
131
132            }
133        }
134
```