Université de Caen

Projet Annuel

L3 Informatique

Bataille d'ia pour le jeu de Quarto

Auteurs :
Robin GALLIS
Pierre MAURAND
Nicolas AUBRY
Thomas FILLION

Professeur :
Mr Grégory BONNET

7 mars 2020

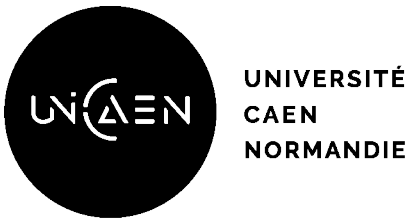

Table des matières

1 Introduction

Le projet annuel est indéniablement la meilleure façon de faire concorder les compétences techniques avec les compétences sociales acquises lors de nos 3 années de licence informatique. De plus, le projet annuel est une étape importante à réaliser afin de valider notre licence.
Pour ce faire, nous avions le choix entre plusieurs projets avec différents enseignants. Les projets allaient de la création d'un site de rencontres à l'étude des fractals, mais celui qui a le plus retenu notre attention est la **Bataille d'intelligence artificielle pour le jeu de QUARTO** dont l'intitulé est :

« Le but de ce projet est de développer un jeu de Quarto sur lequel une intelligence artificielle pourra jouer. Il s'agira dans un premier temps d'implanter le moteur du jeu (l'aspect graphique n'est en rien important.) et un algorithme naïf, type Minimax avec élagage Alpha-beta (Wikipédia min max). Dans un second temps, il s'agira d'implanter d'autres algorithmes, comme un MDT (A Minimax Algorithm faster than NegaScout) et un MCTS (Recherche arborescente Monte-Carlo) Et de permettre de choisir dynamiquement lequel sera exécuté. Une évaluation comparative des différents algorithmes sera alors demandée. »

D'après cet énoncé, on peut donc décomposer ce projet en 3 parties :

- La conception du modèle du jeu de QUARTO
- Le développement des différents algorithmes de résolution (MCTS, Minmax..etc).
- L'analyse des résultats en fonction de différents paramètres.

2 Objectifs du projet

2.1 Le Quarto

Le Quarto est un jeu de plateau stratégique dont le but est d'aligner 4 pions ayant des caractéristiques communes. Voici les pions dont le jeu est composé :

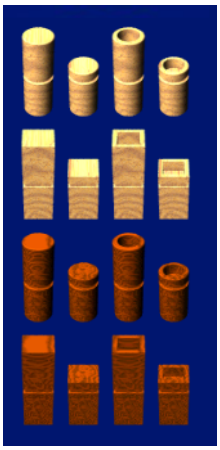

FIGURE 1 – Pièces composantes du jeu.

Sur cette image, vous pouvez remarquer que certaines pièces ont des caractéristiques communes (la taille, la couleur...etc).
Le jeu se déroule sur un plateau de 4x4 cases disposées comme suit :

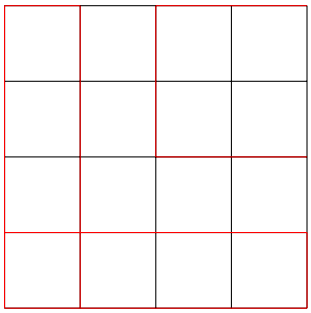

FIGURE 2 – Composition d'un plateau de Quarto.

À chaque tour, le joueur va placer la pièce que l'adversaire aura choisi pour lui et ensuite donner une pièce à l'adversaire.
De ce fait, au premier tour, le joueur qui va jouer en premier ne placera pas de pièce sur le terrain, il va simplement choisir la pièce que l'adversaire va devoir jouer.
Le gagnant sera donc le premier joueur qui placera 4 pièces ayant des caractéristiques communes soit en ligne soit en colonne soit en carré (voir en rouge sur la figure précédente).

2.2 Le Minmax

Avant de commencer nous tenons à indiquer que tous les détails algorithmiques concernant ces algorithmes seront expliqués ici : 4.3. Le Minmax est un algorithme permettant la résolution de jeux tels que le quarto.
Pour ce faire, le Minmax va aller explorer l'arbre du jeu.
Explorer l'arbre du jeu veux dire qu'il va tenter de jouer à chaque case disponible (sur une copie du jeu), puis pour chaque case testée on va regarder si on gagne, on perd ou le jeu n'est pas encore fini, dans le cas de cette dernière possibilité, on va essayer de jouer le tour d'après(toujours sur une copie du jeu) et ce jusqu'à ce que toutes les possibilités aient étés testées. En fonction de toutes ses possibilités, nous pourrons donc connaître le coup optimal. [1].
Cet algorithme va donc chercher à limiter (minimiser) la perte maximale [2]. En clair, cet algorithme va tenter au maximum de limiter les pertes tout en empêchant l'adversaire de minimiser les siennes.
L'algorithme Alphabeta fait la même chose que l'algorithme Minmax à la seule différence qu'il ne va pas construire la totalité des possibilités du jeu à chaque tour, au contraire, il ne va pas aller explorer les possibilités qui ne lui semblent pas prometteuses afin de grandement limiter le temps de calcul.

2.3 Le MCTS

De même que pour le Minmax, tous les détails algorithmiques seront donnés ici : 4.2.
Le MCTS [3] est un algorithme de recherche heuristique qui lui aussi permet de résoudre des jeux tels que le quarto.
Monte-Carlo tree search signifie que l'on va chercher la solution en s'aidant d'un arbre (tree). Cet algorithme possède une structure de données arborescente en mémoire [4].

1. La solution optimale est calculée grâce à une évaluation du jeu, nous y reviendrons après.
2. d'où le nom Minmax
3. Monte-Carlo tree search
4. il enregistre et stocke des informations réutilisables.

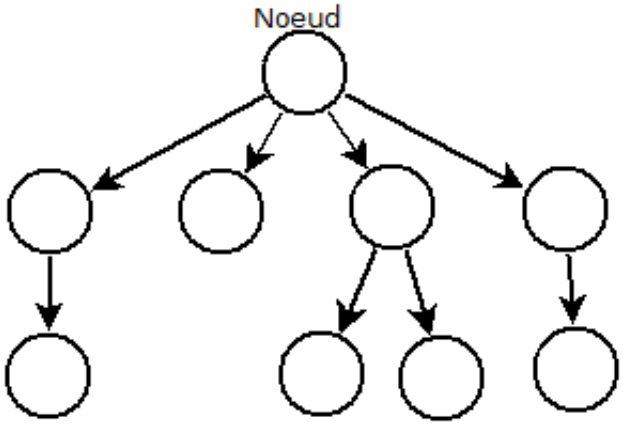

FIGURE 3 – Exemple d'arbre stocké par le MCTS

Chaque nœud possède 2 valeurs : le nombre de matchs effectués et le nombre de matchs gagnés. Le MCTS va donc créer un nœud pour chaque possibilité, puis va tenter une partie aléatoire (sur une copie du jeu) pour chaque nœud, si la partie aléatoire est gagnée, alors l'algorithme va faire remonter l'information en incrémentant le compteur de victoire du nœud, si elle est perdue, l'algorithme ne va pas modifier le compteur de victoire. Pour finir, l'algorithme va regarder quel nœud possède la meilleurs note [5] et jouer en fonction. La différence principale avec le Minmax/Alphabeta réside donc dans l'utilisation de l'aléatoire afin de jouer, un aléatoire qui va servir à orienter le jeu, mais étant donné que ça reste de l'aléatoire, il est probable que l'algorithme fasse, dans certains cas n'importe quoi, c'est d'ailleurs pour cela que nous pouvons produire plusieurs simulations sur chaque nœuds au lieu d'une, ce qui affinera le calcul .

2.4 Le MDT

N'ayant pas réussi à l'implémenter, nous ne pourrons pas nous attarder sur les détails algorithmiques de cet algorithme. L'algorithme MDT [6] est un algorithme de classification et de prédiction permettant de prévoir des informations grâce à des calculs sur des bases de données.
MDT signifiant : Microsoft Decision Trees, cet algorithme, inventé par microsoft, va tenter de prendre une décision en prévoyant ce que va faire l'adversaire. Pour ce faire, tout comme le MCTS, cet algorithme utilise une structure de données arborescente. L'algorithme ajoute un nœud au modèle chaque fois qu'une colonne d'entrée en corrélation significative avec la colonne prédictive est détectée. En plus clair, cet algorithme va regarder comment joue l'adversaire(dans le cas général) et va ajouter un nœud quand il pensera pouvoir prédire un coup. On va donc indiquer à l'algorithme comment doit se

5. La note est le rapport nombre de victoire sur nombre de matchs.
6. Microsoft Decision Trees

finir le jeu pour qu'il gagne et lui va essayer de prévoir les coups de l'adversaire afin de jouer et arriver à ce résultat.

2.5 Notre implémentation

Le projet annuel ayant été écourté nous n'avons pas pu arriver à faire tout les algorithmes demandés, cependant nous avons tout de même réussi à créer 2 versions du Minmax ainsi que leurs élagages AlphaBeta correspondants. De plus, nous avons mené à terme la mise en place de l'algorithme MCTS, le tout reposant évidemment sur une version du Quarto totalement fonctionnelle et fidèle au jeu initial.
Voici donc un résumé ce qui a été fait :

Eléments	Demandés	Faits
Jeu du Quarto	X	X
Minmax	X	X
Alphabeta	X	X
MCTS	X	X
MDT	X	
Minmax V2		X
Alphabeta V2		X

TABLE 1 – Liste des éléments produits et demandés

3 Organisation du projet

3.1 Répartition du travail

Le temps étant compté, nous avons donc développé le modèle du jeu en commun avec le deuxième groupe de Mr Bonnet. Ensuite, sous les conseils de notre enseignant, nous nous sommes séparé les expérimentations à faire avec le deuxième groupe. De notre côté, nous devions faire les expérimentations concernant le joueur 1, tandis que le deuxième groupe ferait celles pour le joueur 2 [7]. Les deux groupes étant désormais séparés, de notre côté nous nous sommes attelés à la conception de l'algorithme MCTS. Cet algorithme étant composé de 4 parties, nous avons chacun développé l'une des parties. Robin s'est occupé de la sélection, Thomas développa la partie rétro-propagation, Pierre à développé la simulation des matchs et pour finir Nicolas a produit la partie expansion.
Après la dernière réunion avec notre professeur référent, nous nous sommes encore une

7. En effet les algorithmes ne vont pas joueur de la même façon s'ils sont le joueur 1 ou le joueur 2

fois séparés en 4, Pierre et Thomas ont développés les algorithmes Minmax et Alphabeta tandis que Nicolas s'occupait de la maintenance du code et que Robin produisait les expérimentations. Enfin, nous avons tous contribué à la création du rapport. Voici un diagramme de gant résumant tout cela :

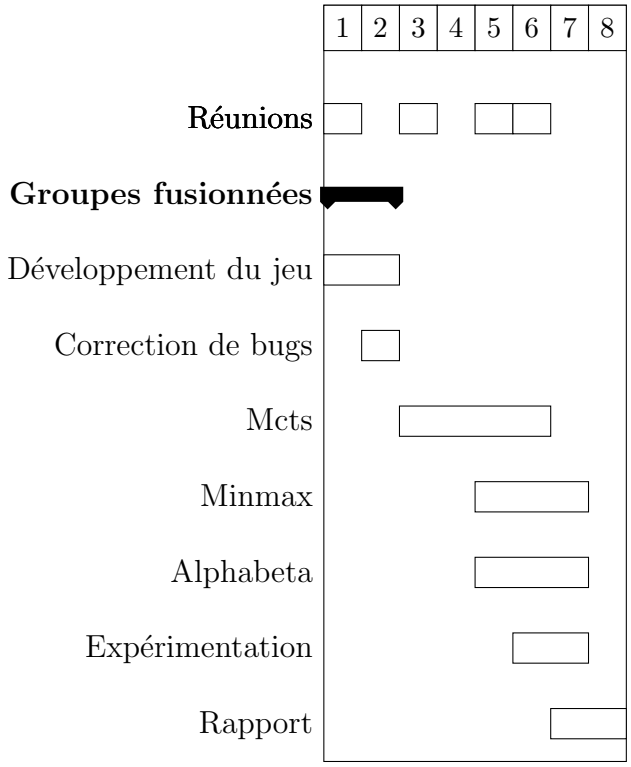

FIGURE 4 – Diagramme de gant de notre projet en semaines

3.2 Architecture

Concernant l'architecture de notre programme, il est composé de 5 packages qui sont :

— **game :** Package contenant le modèle du jeu Quarto ainsi que le modèle du plateau (que nous avons séparés afin de pouvoir gérer les plateaux sans appliquer les règles du jeu pour le MCTS notamment).

— **mcts :** Package contenant les 4 instructions (rétro-propagation, expansion, sélection et simulation) qui composent un algorithme MCTS. Ce package sera utilisé par le MctsPlayer (voir plus loin) afin de jouer.

— **packPlayer :** Package contenant la totalité des joueurs que nous avons crées. Il a un joueur aléatoire, un joueur humain, un joueur MCTS, deux versions du Minmax et enfin 2 versions de l'Alphabeta.

— **token :** Ce package ne contient que la représentation des pièces du jeu car comme dit précédemment une pièce peut être noire ou blanche, ronde ou carrée, grande ou petite.

— **util :** Ce package contient toutes les classes utilitaires permettant entre autres de modéliser l'arbre ou de modéliser un coup du jeu.

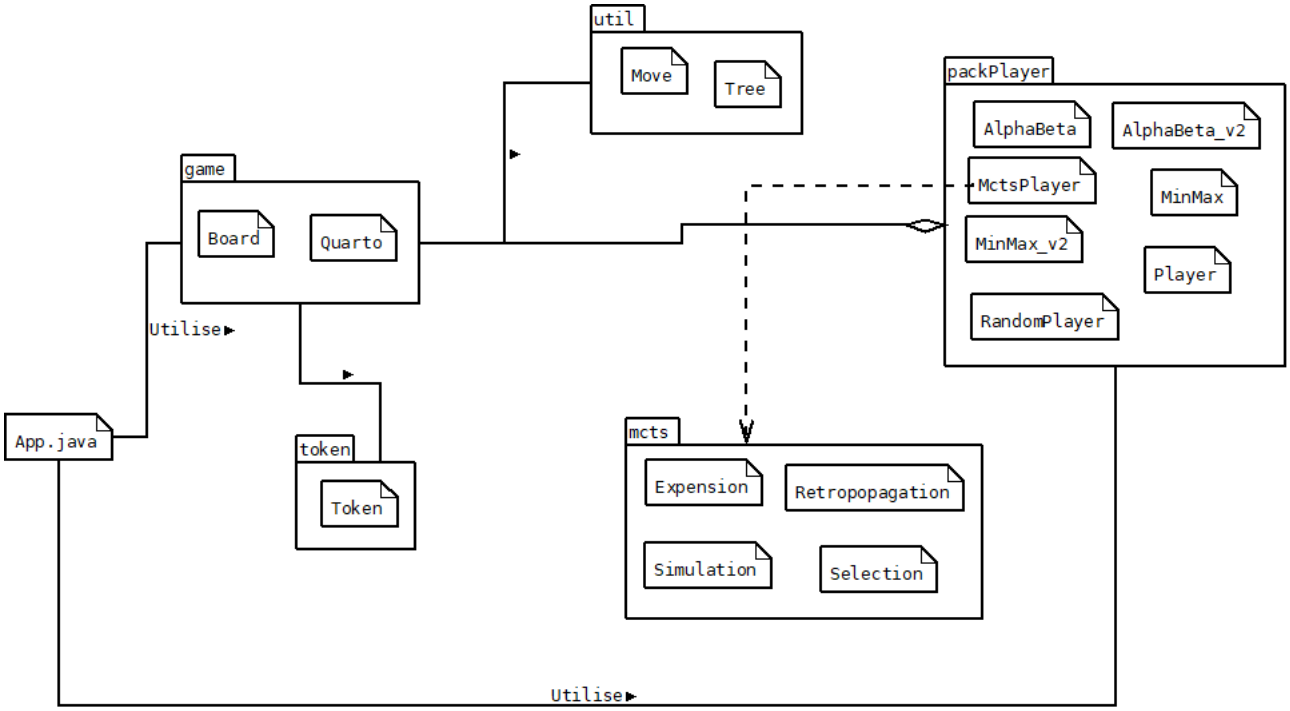

FIGURE 5 – Diagramme de nos packages

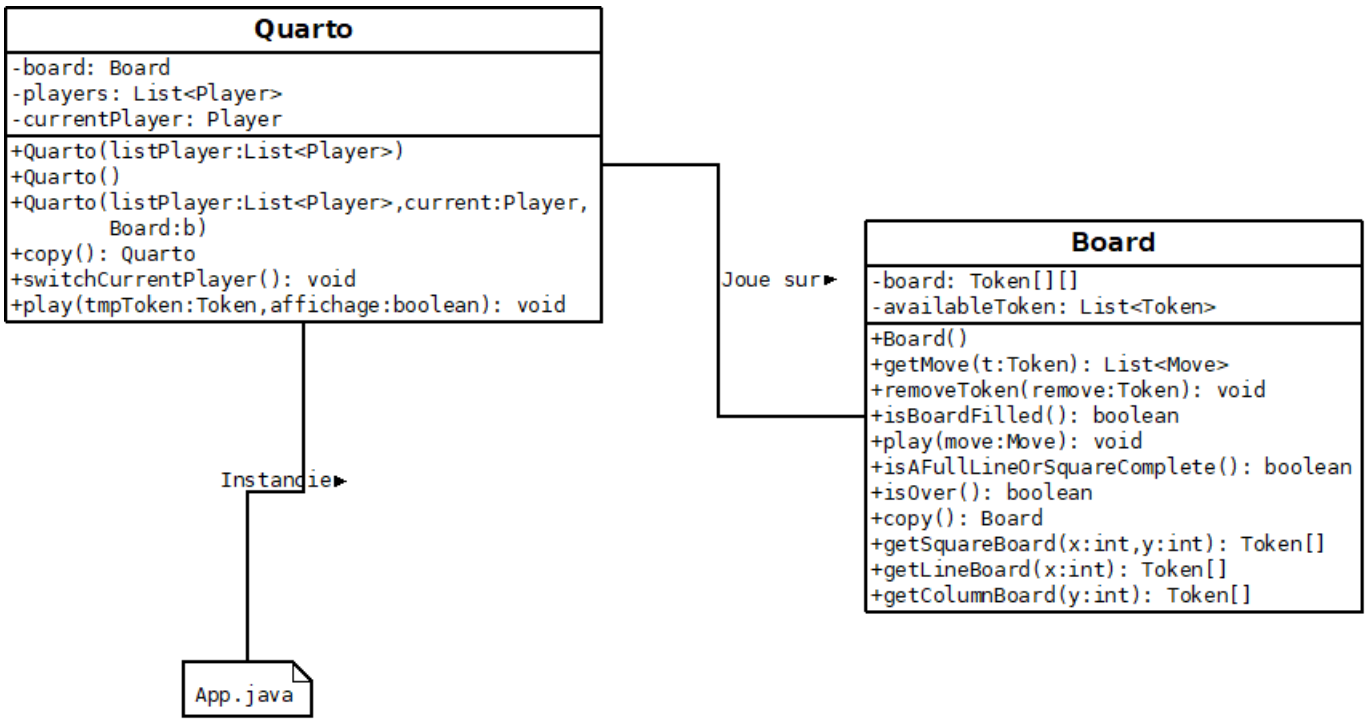

FIGURE 6 – Diagramme de classe du package game

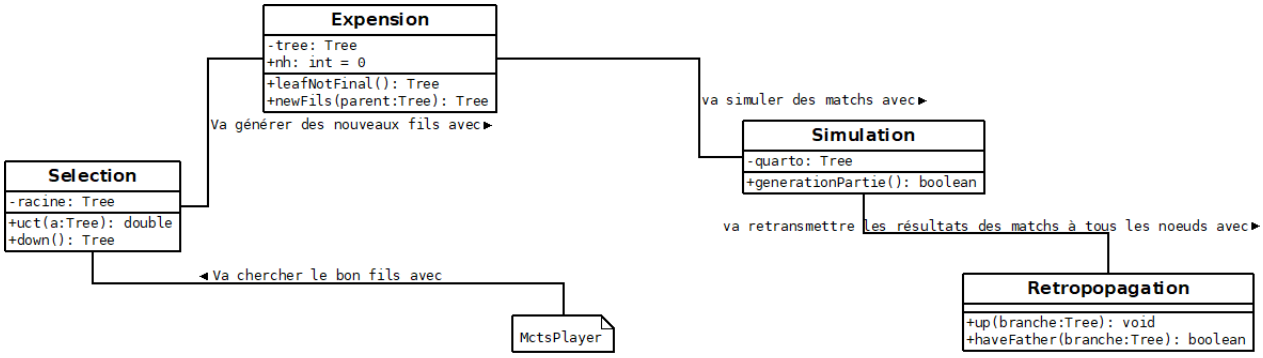

FIGURE 7 – Diagramme de classe du package MCTS

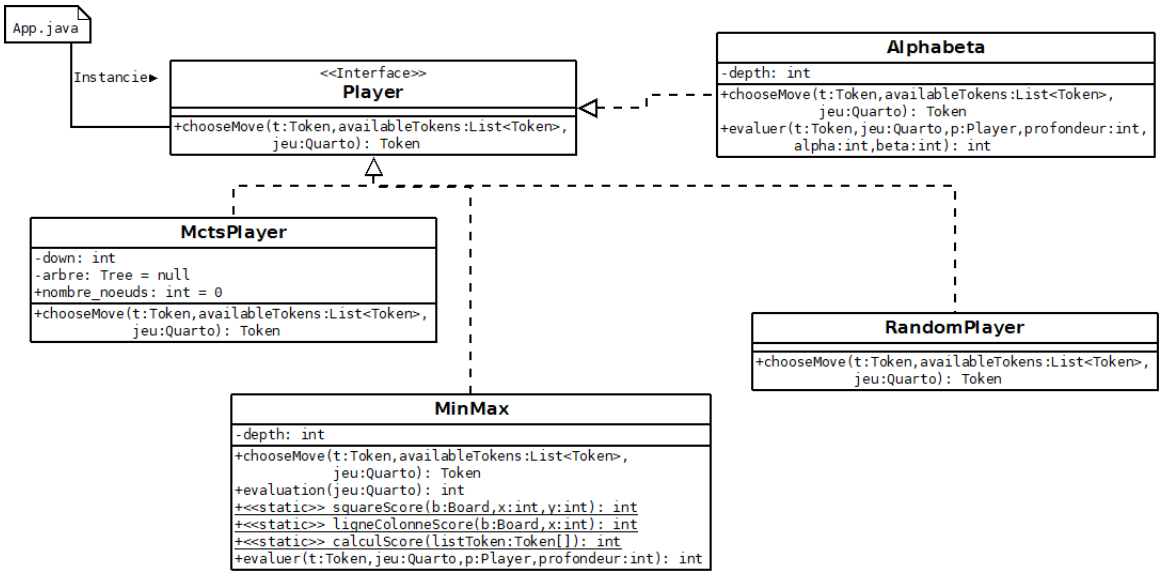

FIGURE 8 – Diagramme de classe du package packPlayer

4 Éléments techniques

4.1 Le modèle du jeu

Comme énoncé précédemment, notre jeu se compose d'un modèle (la classe Quarto) qui va se jouer sur un plateau (la classe Board).
Cela permet d'avoir un plateau qui peut convenir pour d'autres jeux ou un jeu qui pourrais jouer sur un autre plateau si dans le futur nous avions d'autres expérimentations à mener.
La classe Quarto
Le Quarto est composé des joueurs ainsi que du plateau, mais ce qui fait sa spécificité, c'est que dans notre code il n'y a pas de classe qui va jouer le jeu, pour des soucis d'optimisation, c'est le quarto qui possède une méthode "play" qui va jouer et finir une partie en fonction des joueurs et du plateau qu'il possède.
De plus, cette méthode possède un paramètre "affichage" qui va nous permettre de choisir si l'on souhaite afficher ou non le jeu, car afficher le jeu va évidemment augmenter les temps de calcul, mais permet de corriger les éventuels bugs.
La classe Board
Cette classe représente simplement le plateau de jeu (un plateau carré en 4x4 cases) et permet d'obtenir une copie du jeu afin de faire jouer nos IA dessus par exemple. Cette classe permet aussi de savoir si le jeu est fini ou non et qui gagne.

4.2 Le MCTS

Comme dit précédemment, le MCTS est composé de 4 parties distinctes :
-La sélection de l'arbre
-L'expansion de l'arbre
-La simulation des parties aléatoires
-La rétro-propagation des résultats

Les arbres :
On en parle depuis le début du rapport, le MCTS utilise des arbres pour pouvoir stocker les résultats des simulations qu'il effectue et ainsi pouvoir faire remonter les notes de chaque nœud. Pour bien comprendre la suite voici quelques définitions :
Un arbre est un ensemble de nœuds possédant une relation parent-fils.
Le père d'un nœud est le nœud relié à celui-ci directement au-dessus.
Le fils d'un nœud est le nœud relié à celui-ci directement au-dessous.
Si un nœud ne possède pas de fils alors s'est une feuille
La racine de l'arbre est le nœud qui est situé le plus haut dans l'arbre.
Un nœud est étendu s'il ne peut plus avoir de fils. (car le jeu serait fini par exemple)

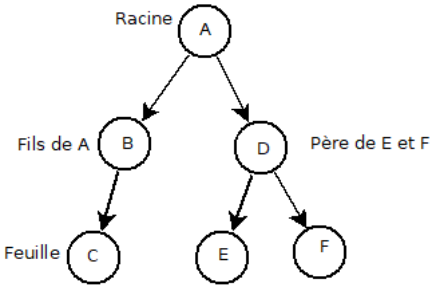

FIGURE 9 – Résumé de la définition d'un arbre

La sélection :

Pour commencer, le MCTS va être obligé de générer tous les coups possibles après celui de la racine. Par exemple après le premier coup, l'arbre MCTS aura donc 15^2 fils (15 cases * 15 pièces : ensemble des coups disponibles) qui représenteront toutes les positions possibles. Ensuite, il va choisir vers quel nœud (quel enfant) aller. Pour ce faire, il existe une formule mathématique nommée "UCT" qui va permettre d'avoir un compromis entre aller explorer un nœud qui est prometteur (qui a un bon ratio victoire/match) et un nœud qui est un peu moins prometteur afin de laisser une chance à l'aléatoire des tirages. Voici cette formule :

$$\frac{nombre\_victoire}{nombre\_test} + \sqrt{2} * \sqrt{\frac{ln(nombre\_test\_du\_pere)}{nombre\_test}}$$

FIGURE 10 – formule de L'uct

Ensuite, nous avons une méthode permettant de descendre vers tous les premiers fils afin de leur attribuer une valeur calculée grâce à l'uct. En voici l'algorithme :

Algorithme 1 : Évaluation des fils d'un nœud avec une formule UCT

Entrées : Vide

Sortie : Un arbre contenant les nœuds évalués avec l'uct

```
1  Tree passage ← this.racine
2  double init
3  tant que true faire
4      si passage est étendu ou que son Board est fini alors
5          retourner passage
6      fin
7      Tree tmp ← null
8      init = 0
9      pour un Tree fils parcourant la liste des fils de passage faire
10         double result ← this.uct(fils)
11         si result est supérieur ou égal à init alors
12             init ← result
13             tmp ← fils
14         fin
15     fin
16     passage ← tmp
17 fin
```

L'expansion :

Notre classe Expansion va nous permettre de savoir si une feuille est totalement étendue ainsi que de créer un nouvel enfant qui n'existe pas encore.

Pour savoir si une feuille est totalement étendue, on va compter ses fils et regarder s'ils sont équivalents au (nombre de pièces encore possible) au carré. S'il n'est pas étendu, la méthode va alors appeler la méthode "newFils" qui va nous permettre de créer un enfant aléatoire parmi ceux qui ne sont pas encore crées. De plus, ce nouvel enfant crée donnera une pièce aléatoire lors de la simulation des matchs par le MCTS.

La Simulation :

La Simulation ne possède qu'une méthode qui va permettre, comme annoncée précédemment, de simuler une partie aléatoire à partie du nœud dans lequel la méthode est appelée. De plus grâce à cette méthode, l'on va savoir qui est le gagnant.

Voici donc l'algorithme de cette méthode :

Algorithme 2 : Simulation d'une partie aléatoire

Entrées : Vide

Sortie : Un booleen permettant de savoir si on gagne ou non

```
1  Quarto copy ← this.quarto.getBoard().copy()
2  List < Player > players ← newArrayList < Player > ()
3  RandomPlayer random ← newRandomPlayer()
4  players.add(random)
5  players.add(newRandomPlayer())
6  Quarto q ← newQuarto(players, random, copy.getBoard())
7  Tree tmp ← quarto
8  q.play(this.quarto.getPiece(), false)
9  si q.getWinner().toString().equals(random.toString()) alors
10 |   retourner true
11 fin
12 retourner false
```

On remarque donc que la partie aléatoire est produite, car deux joueurs aléatoires jouent dessus, ce n'est donc pas le quarto qui va générer la partie aléatoire. De plus, le MCTS va posséder ce qu'on appelle le budget. Le budget est le nombre de simulations aléatoires que nous pourrons produire à chaque nœuds, donc plus le budget sera grand, plus l'expertise de l'algorithme sera précise(plus la note attribuée risque d'amener vers une victoire) mais plus l'algorithme sera lent.

La Rétro-propagation :

La rétro-propagation ne marche que si le nœud a un père, donc pour vérifier cela, nous avons crées une méthode permettant de vérifier cela. Ensuite, la deuxième méthode permet de faire propager le résultat du match aléatoire sur les nœuds supérieurs jusqu'à la racine (seul nœud qui est censé ne pas posséder de père).

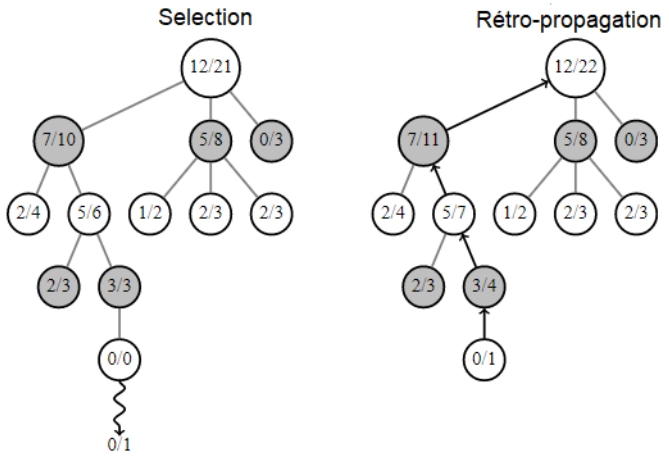

FIGURE 11 – Démonstration du fonctionnement de la rétro-propagation

Sur ce schéma, le nombre à gauche dans chaque nœud est le nombre de victoires et le nombre de matchs est le nombre de droite. On remarque donc qu'après la simulation (qui est une défaite), la rétro-propagation va étendre le résultat à tous les nœuds au-dessus jusqu'à la racine.

Le Joueur MCTS :

Pour commencer, le joueur MCTS va regarder si le coup qui lui est donnée est null(il ne peut être null que s'il est le premier joueur.), et si c'est le cas, il va donner une pièce aléatoirement à l'adversaire. Ensuite, il va voir si l'arbre qu'il possède est null, si c'est le cas, il va créer la racine initiale afin de pouvoir commencer à expérimenter. Sinon, on va couper l'arbre au niveau du nœud ou on jouera. Ceci va permettre de ne pas saturer la mémoire car en plus de faire cela, on va supprimer les références au nœud et reset le garbage collector de java.

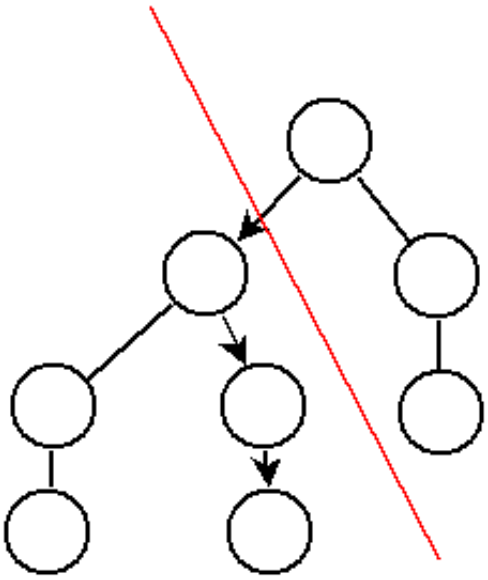

FIGURE 12 – Exemple de coupure de l'arbre par le Mcts

Ici, par un choix de la sélection du MCTS, il va aller vers le nœud à gauche, de ce fait, ou va couper le reste de l'arbre (tout ce qui est à droite du trait rouge) afin de permettre à java de ne pas avoir d'erreurs de mémoire.
Ensuite, le joueur MCTS va simplement appeler l'une après l'autre les classes décrites précédemment dans l'ordre normal : sélection, expansion, simulation et rétro-propagation. Sachant que la sélection attribue une valeur "uct" à chaque nœud, le joueur MCTS va aller vers la valeur la plus grande, et si plusieurs nœuds ont la même valeur (ce qui arrive souvent), on va sélectionner l'un de ces nœuds de manière aléatoire permettant de ne pas avoir toujours les mêmes parties jouées.

4.3 MinMax et Alphabeta

Le Minmax :

Pour commencer voici l'algorithme du Minmax sous sa forme de pseudo-code :

```
function minimax(node, depth, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, minimax(child, depth − 1, FALSE))
        return value
    else (* minimizing player *)
        value := +∞
        for each child of node do
            value := min(value, minimax(child, depth − 1, TRUE))
        return value
```

FIGURE 13 – Pseudo code Minmax

Comme dit précédemment, le joueur Minmax va utiliser une fonction d'évaluation afin de désigner le meilleur coup et ainsi pouvoir jouer en conséquence. Dans notre cas, nous avons fait 2 algorithmes Minmax, le premier algorithme va donner un score à chaque formation de pions permettant la victoire (soit une ligne, soit une colonne ou soit un carré comme dit au début du rapport). De part la nature de l'algorithme Minmax, nous sommes obligés de donner une profondeur à notre Minmax car sinon la construction de tout les coups possibles à chaque tour saturerai totalement notre mémoire et ne pourrait pas se produire sur nos ordinateurs. Dans la deuxième version du Minmax, cette fois nous allons toujours prendre le score donné comme précédemment, mais ce score ne sera attribué que si chaque pion de la formation partage au moins un point en commun. Notre deuxième est donc plus informé que le premier et sera donc plus efficace, mais par contre, il sera plus lent, car il prend plus de choses en compte. Voici donc les deux fonctions d'évaluation des deux algorithmes, Minmax et Minmax_v2 :

Algorithme 3 : ÉVALUATION DU MINMAX

Entrées : Le jeu de Quarto en cours

Sortie : Un entier représentant la note de chaque partie évaluées

```
1  int score ← 0
2  Board board ← jeu.getBoard()
3  pour int i ← 0; i < 3; i + + faire
4  |   pour int j ← 0; j < 3; j + + faire
5  |   |   score+ = squareScore(board, i, j)
6  |   fin
7  fin
8  pour int i ← 0; i < 4; i + + faire
9  |   score+ = squareScore(board, i, j)
10 fin
11 retourner score
```

Pour cet algorithme, on va donc utiliser une fonction nommée "CalculScore" qui va calculer sur le plateau le score de la ligne et de la colonne demandée, ce score sera équivalent aux nombre de pièce sur la ligne ou la colonne. Cette fonction fera de même pour les placements en carré. Une partie qui aura donc beaucoup de lignes/colonnes/carrés presque remplies aura donc un meilleur score qu'une autre en ayant moins.
L'algorithme d'évaluation pour le Minmax_v2 est le même, sauf que pour le Minmax_v2, on utilise une nouvelle fonction se nommant "pointEnCommun" qui va permettre de savoir si les éléments que on lui donne on a point commun, on lui donnera donc les pièces contenues sur les lignes, carrés ou colonnes. Voici l'algorithme :

Algorithme 4 : MÉTHODE POINTENCOMMUN

Entrées : La liste des pièces dont on veux savoir s'ils possèdent un point commun

Sortie : Un booléen indiquant si les pièces partagent un point en commun ou non

```
1  List<Token> pris ← newArrayList < Token > ()
2  pour un Token t parmis la liste des pièces faire
3      si t n'est pas null alors
4          si la taille de la liste pris est 0 alors
5              ajouter t à la liste pris
6          fin
7          sinon
8              pour Un Token t2 parmis la liste pris faire
9                  si t n'a pas de points communs avec t2 alors
10                     retourner false
11                 fin
12             fin
13             ajouter t à la liste pris
14         fin
15     fin
16 fin
17 retourner true
```

C'est donc cette méthode qui nous permet de mieux évaluer, et ainsi de gagner plus vite.

L'Alphabeta :

Pour commencer voici l'algorithme Alphabeta en pseudo code

```
fonction alphabeta(nœud, α, β) /* α est toujours inférieur à β */
   si nœud est une feuille alors
       retourner la valeur de nœud
   sinon si nœud est de type Min alors
           v = +∞
           pour tout fils de nœud faire
               v = min(v, alphabeta(fils, α, β))
               si α ≥ v alors  /* coupure alpha */
                   retourner v
               β = Min(β, v)
    sinon
           v = -∞
           pour tout fils de nœud faire
               v = max(v, alphabeta(fils, α, β))
               si v ≥ β alors /* coupure beta */
                   retourner v
               α = Max(α, v)
    retourner v
```

FIGURE 14 – Pseudo code Alphabeta

Le principe de l'algorithme Alphabeta est d'optimiser l'algorithme Minmax. Nous n'allons pas optimiser les fonctions d'évaluation, mais plutôt tenter de ne pas explorer des nœuds qui ne semblent pas bons. En diminuant le nombre de nœuds, on va donc améliorer considérablement la vitesse de calcul, mais on va risquer de se retrouver dans un minimum local et donc globalement l'Alphabeta sera moins efficace qu'un Minmax, mais beaucoup plus rapide. Nous avons donc crées une version d'Alphabeta pour le premier algorithme Minmax et une deuxième version pour le second algorithme.

5 Expérimentations et Usages

5.1 Les expérimentations

Afin de tester nos algorithmes nous les avons soumis à plusieurs expérimentations. Les résultats principaux de ces tests seront l'augmentation du temps de prise de décision de l'algorithme MCTS ainsi que sont taux de victoires augmentant en fonction du budget attribué, le taux de victoires du Minmax contre le Minmax_v2 et la comparaison du nombre de nœuds explorés par les Minmax face aux Alphabeta.

5.1.1 Le taux de victoires du MCTS et son nombre de nœuds

Pour commencer, les premiers tests comparaient le joueur MCTS et le joueur random :
-Comparaison du taux de victoire du MCTS en fonction du budget.
-Comparaison du nombre de nœuds générés par le MCTS en fonction du budget.

Voici les résultats de ces premières expérimentations :

Budget	Taux de victoire MCTS vs Random
50	54,44%
100	56,48%
200	69,6%
400	74,66%
800	77,5%
1600	82,66%
3200	81,62%
6400	81,66%

TABLE 2 – Résultats de la première expérimentation

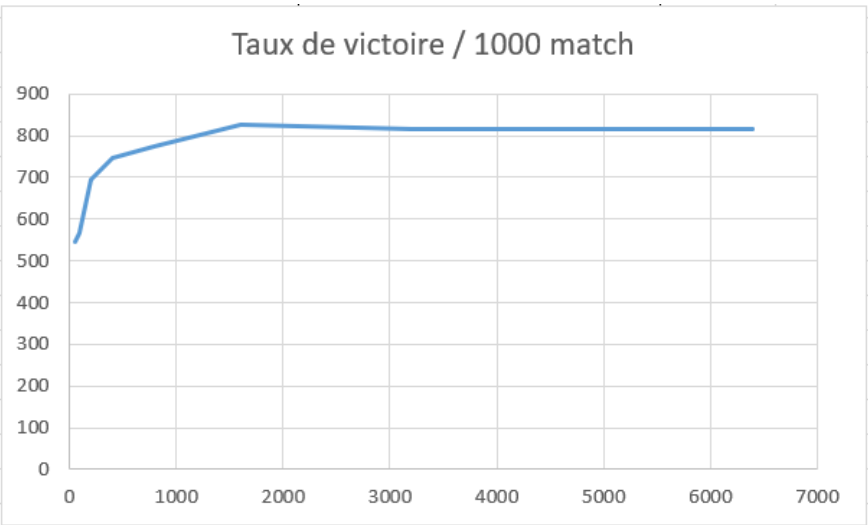

FIGURE 15 – Graphique recensant les résultats

On remarque donc que le taux de victoires stagne après les 1600 simulations, donc on peut dire que notre algorithme ne pourra pas beaucoup dépasser ce taux de 82% de victoires contre un joueur aléatoire.
La deuxième expérimentation concerne le nombre de nœuds crées par le MCTS en fonction du budget :

Budget	Nombre de nœuds moyens
50	232
100	450,2
200	832,4
400	1518,4
800	2779,4
1600	5236,6
3200	10000,3
6400	18883

TABLE 3 – Résultats de la deuxième expérimentation

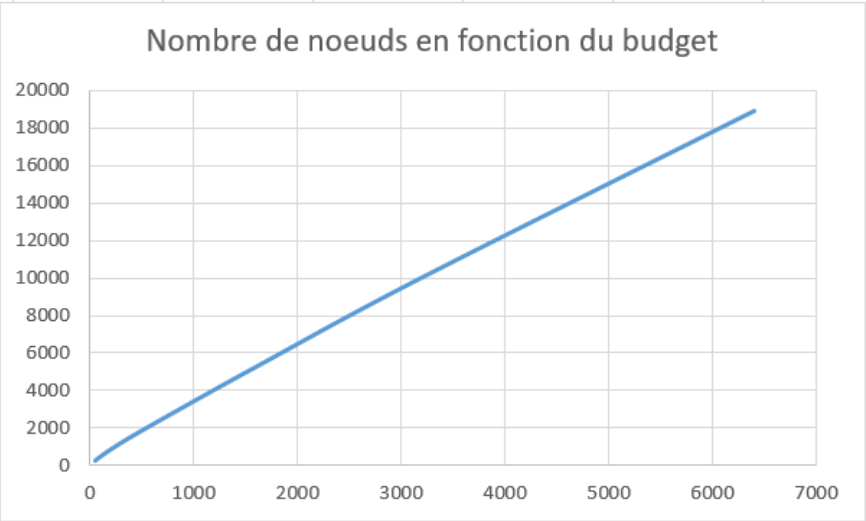

FIGURE 16 – Graphique recensant les résultats

On remarque que l'évolution que nombre de nœuds augmente de manière linéaire. Cela est dû au fait que l'on pas encore rencontré le point où l'arbre est totalement construit. Par soucis de temps, nous ne pouvons pas calculer plus loin que 6 400 simulations, mais au moins nous savons qu'il y aura donc plus de 180 000 nœuds dans l'arbre.
Pour finir, nous avons calculés le temps que prend le programme à finir en fonction du budget.

Budget	Temps
50	16,321s
100	19,9078s
200	31,8166s
400	41,4736s
800	58,3996s
1600	95,5874s
3200	175,8414s
6400	375,32s

TABLE 4 – Temps de fonctionnement du MCTS

Tout les calculs pour le MCTS ont étés réalisés sur une moyenne sur 1000 matchs

5.1.2 Expérimentations sur le Minmax et le Minmax_v2

Maintenant, on va regarder le temps d'exécution, le nombre de nœuds explorés et le nombre de victoires du Minmax ainsi que du Minmax_v2 contre un joueur random. La notion de budget n'ayant pas de sens pour des algorithmes Minmax ou Alphabeta, ici nous allons utiliser la notion de profondeur. La profondeur étant le nombre de coup d'avance que l'algorithme pourra tester. Plus il y aura de coups d'avance, plus les chances de gagner augmenteront mais l'algorithme sera plus lent. Voici ces résultats :

Profondeur	Taux de victoires Minmax vs random	Taux de victoire Minmax_v2 vs random
1	100%	100%
2	100%	100%
3	100%	100%

TABLE 5 – Résultats des expérimentations des Minmax sur 100 matchs

On remarque que la profondeur, contre un random ne sert pas, le taux de victoires stagne dès la profondeur 1, cela est causé par la taille du jeu qui est petite et donc le jeu peux se résoudre de peu de manières. Le taux de victoires n'est pas obligatoirement de 100% (même si dans nos expérimentations, c'est le cas) car il se peut que le joueur aléatoire joue parfaitement, mais nous avons, par manque de temps fait trop peu de matchs pour le voir. Voici maintenant la deuxième expérimentation concernant le nombre de nœuds crées par le Minmax et le Minmax_v2 :

Profondeur	Nombre de nœuds moyens Minmax	Nombre de nœuds moyens Minmax_v2
1	34 991,3676 nœuds	4 802,5 nœuds
2	4 567 902,2 nœuds	461 086,9 nœuds
3	1 029 254 053 nœuds	728 536 267,5 nœuds

TABLE 6 – Résultats pour le nombre de nœuds explorés par matchs en moyenne

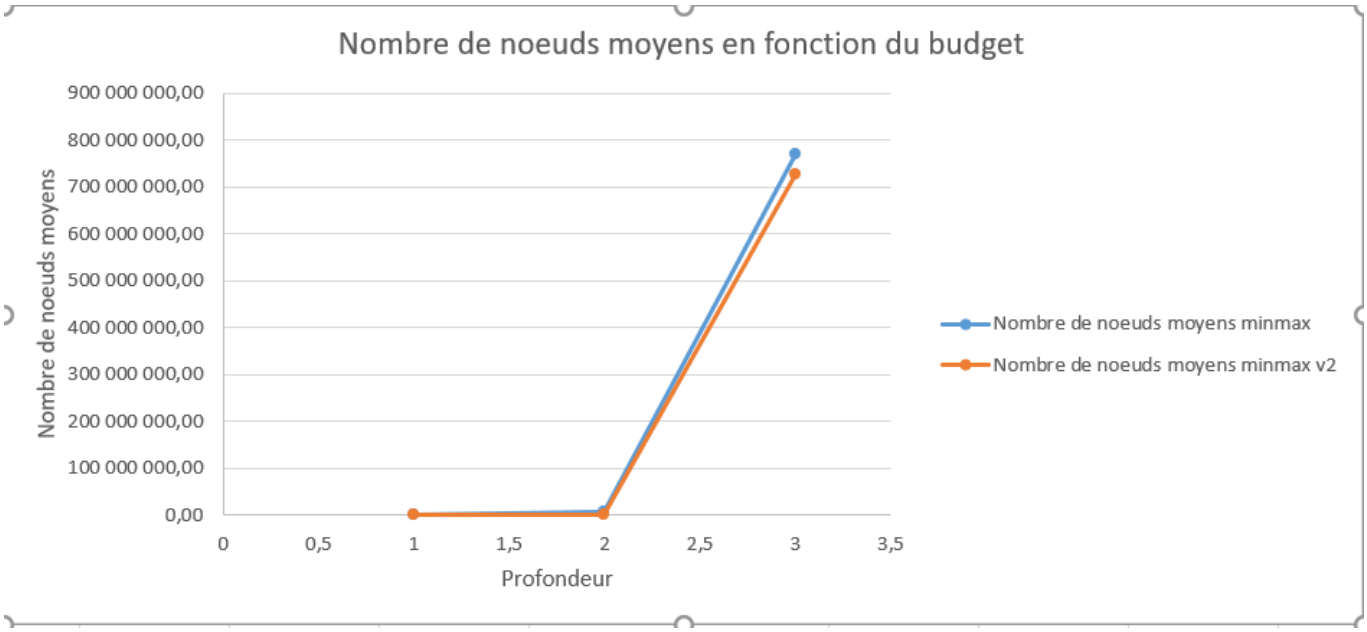

FIGURE 17 – Graphique recensant les résultats

Pour commencer on remarque que le Minmax_v2 parcours moins de nœuds ce qui est logique car ici on calcule le nombre total de nœuds parcourus dans le match, or ce nouveau Minmax va gagner plus vite, donc explorer moins de nœuds au total mais le même nombre environ à chaque coup(évidemment c'est une moyenne donc il y aura des petites différences). Voici donc le nombre de nœuds moyens parcourus par coup en moyenne :

Profondeur	Nombre de nœuds moyens Minmax	Nombre de nœuds moyens Minmax_v2
1	40,8777 nœuds	37,548 nœuds
2	4 954,3407 nœuds	4 610,869 nœuds
3	114 361 561,444 nœuds	112 595 230 nœuds

TABLE 7 – Résultats pour le nombre de nœuds explorés par matchs en moyenne

On remarque que le nombre de nœuds explorés augmente considérablement alors que

la profondeur n'augmente que de 1 et cela va se ressentir dans les temps d'exécution :

Profondeur	Temps d'éxécution Minmax	Temps d'éxécution Minmax_v2
1	0,1265s	0,2125s
2	2,6715s	8,2465s
3	251,23s	831,5205s

TABLE 8 – résultats pour les temps d'exécution par matchs en moyenne

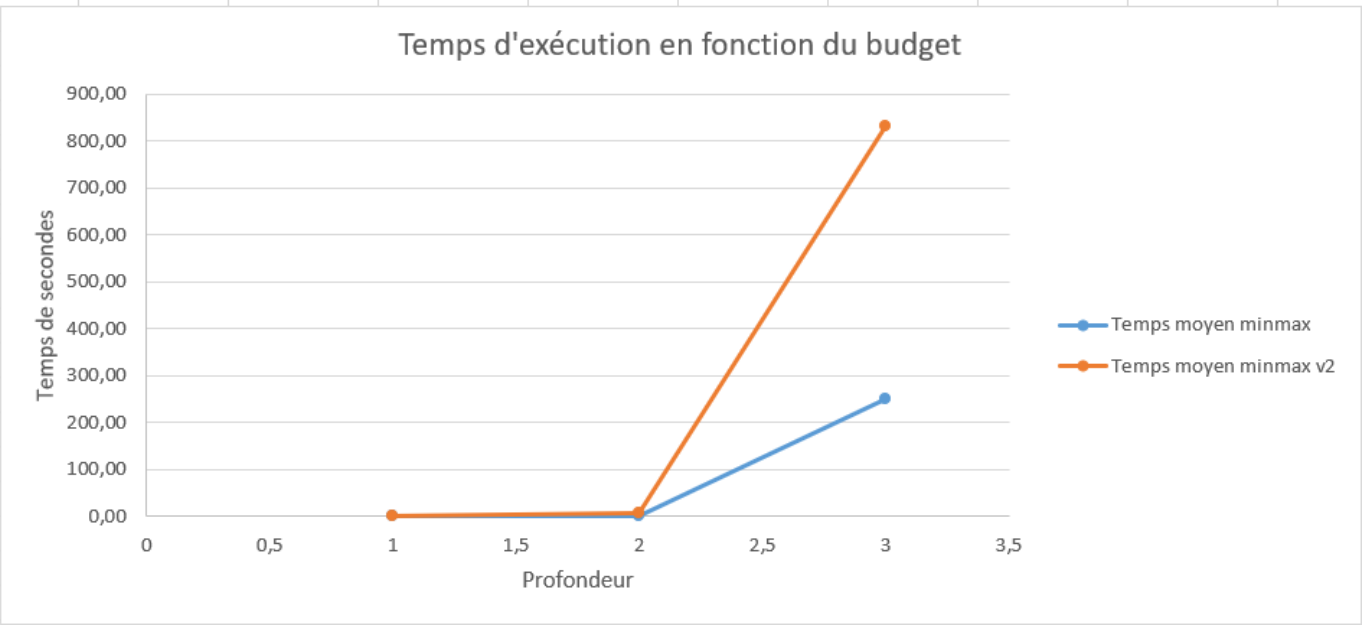

FIGURE 18 – Graphique recensant les résultats

Ici, on voit que le deuxième Minmax est minimum 2 fois plus long pour finir son exécution, donc le fait d'informer plus l'algorithme le rend bien plus long. Voici pour finir les résultats des matchs du Minmax contre le Minmaxv2 :

Profondeur	Taux de victoires du Minmax_v2 vs Minmax en étant le joueur 1	Taux de victoires du Minmax_v2 vs Minmax en étant le joueur 2
1	70%	66%

TABLE 9 – Combat du Minmax v2 contre le Minmax

Le Minmaxv2, est donc bien plus fort que le Minmax malgré une vitesse moindre. Aussi après calculs, nous avons remarqués que le Minmax gagne en 11 tours en moyenne alors que le Minmax_v2 gagne en 7 tours en moyenne.

5.1.3 Expérimentations sur l'Alphabeta et l'Alphabeta_v2

On va maintenant reproduire les 3 précédentes expérimentations sur les 2 algorithmes Alphabeta :

Profondeur	Taux de victoires Alphabeta vs random	Taux de victoires Alphabetav2 vs random
1	100%	100%
2	100%	100%
3	100%	100%

TABLE 10 – Résultats des expérimentations des Alphabeta sur 100 matchs

Tout comme pour les Minmax, le taux de victoires reste constant, car qui signifie que l'algorithme marche toujours bien. Voici maintenant la deuxième expérimentation concernant le nombre de nœuds crées par l'Alphabeta et l'Alphabeta_v2 :

Profondeur	Nombre de nœuds Alphabeta	Nombre de nœuds Alphabeta v2
1	2818.95 nœuds	3026.0 nœuds
2	71036.41 nœuds	77216.17 nœuds
3	16 022 749,8 nœuds	836360,02 nœuds

TABLE 11 – Résultats pour le nombre de nœuds explorés par matchs en moyenne

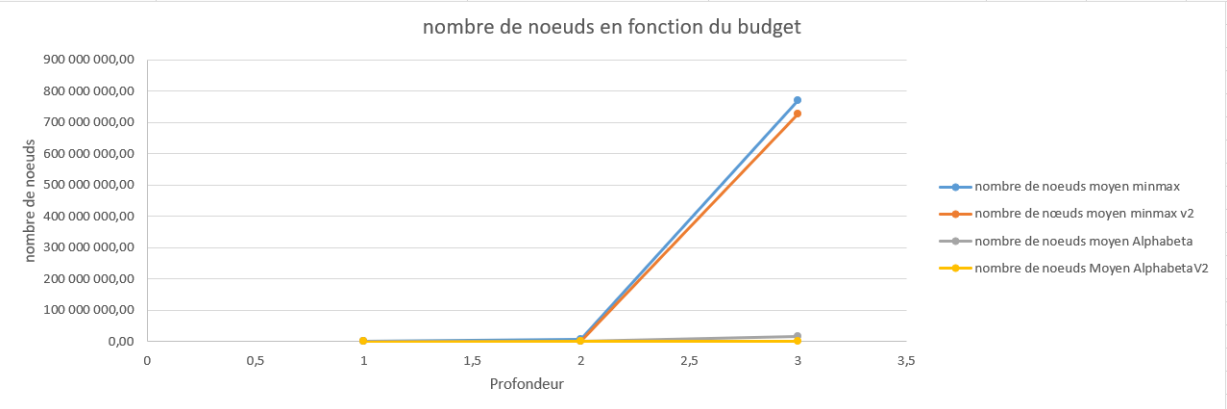

FIGURE 19 – Graphique recensant les résultats

On va voir maintenant le nombre de nœuds par coup en moyenne :

Profondeur	Nombre de nœuds Alphabeta	Nombre de nœuds Alphabeta v2
1	3,3672 nœuds	3,381 nœuds
2	66,0140 nœuds	67,4704 nœuds
3	27 781,64 nœuds	25 453,474 nœuds

TABLE 12 – Résultats pour le nombre de nœuds explorés par matchs en moyenne

On voit bien que le nombre de nœuds pour les Alphabeta sont quasiment identiques, ce qui est aussi logique que pour les Minmax.
Le nombre de nœuds est bien moindre par rapport aux Minmax, et cela va se voir sur les temps d'exécution :

Profondeur	Temps d'exécution Alphabeta	Temps d'exécution Alphabeta_v2
1	0,0478s	0,1210s
2	0,2172s	0,7367s
3	6,2053s	15,8407s

TABLE 13 – résultats pour les temps d'exécution par matchs en moyenne

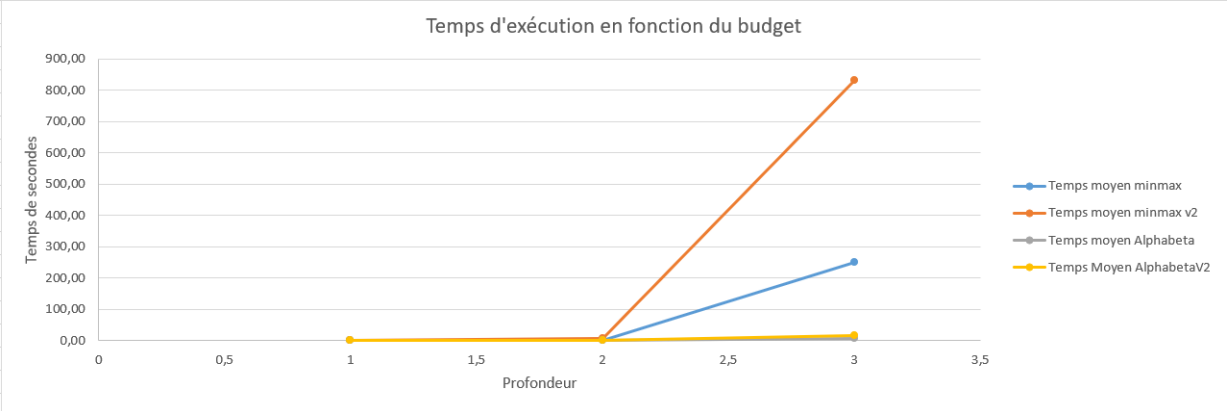

FIGURE 20 – Graphique recensant les résultats

5.1.4 MCTS contre l'alphabeta

Pour finir, on va comparer le MCTS avec l'Alphabeta et faisant varier sa profondeur et en laissant l'Alphabeta à une profondeur de 1. Voici les résultats des taux de victoire :

Budget	Taux de victoire Mcts vs Alphabeta
50	0%
100	0%
200	0%
400	0%
800	0%
1600	0%
3200	0%
6400	0%
12800	3%
25600	6%

TABLE 14 – Confrontation du MCTS et de l'Alphabeta

La victoire de l'Alphabeta est écrasante contre le MCTS, mais cela est logique, pour commencer, le MCTS deviens fort lorsque son arbre commence à être enrichi, or le but de l'Alphabeta est de gagner rapidement, de ce fait, le MCTS n'aura pas le temps de construire son arbre et se fera battre rapidement. De plus, le jeu est trop petit pour que l'algorithme MCTS soit à son plein potentiel, sur un jeu de dame ou de Go, la victoire du MCTS sur l'Alphabeta sera déjà beaucoup plus probable. D'ailleurs, on remarque que lorsque on augmente assez le budget du MCTS, on commence à avoir des victoires (certes en petit nombre), c'est logique, car le nombre de nœuds explorés commence à se rapprocher du nombre de nœuds explorés par l'Alphabeta.

Après toutes ces expérimentations, on peut tirer des conclusions :
-Le Minmax/Alphabeta gagnera qu'importe son adversaire, car sur un jeu aussi petit, il peut construire son arbre de toujours avoir la solution optimale.
-Le MCTS, est beaucoup plus rapide que le Minmax/Alphabeta ce qui est logique, car il conserve son arbre en mémoire, ne fera que chercher dedans et simuler aléatoirement des parties (l'aléatoire étant plus rapide que l'orienté).
-Un Minmax plus informé gagnera contre un moins informé, mais sera moins rapide.

5.2 Utilisation du programme

Notre programme étant fait uniquement pour produire des expérimentations, il ne possède pas d'interface graphique. Néanmoins, pour l'utiliser, nous avons créé une petit script bash permettant de s'y retrouver facilement. Pour démarrer notre programme, il faut donc démarrer ce script, celui-ci vous demandera si vous voulez générer le temps d'exécution ou le comptage des victoire et des nœuds, si vous voulez le temps, il suffit d'écrire true, sinon écrivez n'importe quoi. Ensuite, le script vous demandera d'inscrire

les arguments. Voici comment les renseigner : <nom_joueur1> <arguments_joueur1> <nom_joueur2> <arguments_joueur2> <nombre_de_matchs>
Les noms des joueurs sont : minmax, minmaxv2, alphabeta, alphabetav2, mcts, random. De plus, ces arguments sont modulables, car si vous n'indiquez pas le nombre de matchs, celui-ci sera fixé à 1000 automatiquement. Aussi, si vous avez un joueur random, évidemment il ne possède pas d'argument donc il ne faut rien mettre, il n'y aurait donc plus 5 arguments, mais 4 (ou 3 s'il y a 2 random). Enfin, vous ne pouvez pas jouer avec un random en joueur 1 si le joueur 2 n'est pas aussi un random, car notre groupe devait réaliser les expérimentations sur les algorithmes en tant que joueur 1, or le random ne nécessite pas d'expérimentations spécifiques. Pour finir, les arguments des joueurs IA sont soit leur profondeur pour les Minxmax et Alphabeta (si vous dépassez les 3 de profondeur ça deviens vraiment long à exécuter) soit le budget pour le MCTS (au-delà de 12 800 simulations ça deviens aussi très long). Tous les résultats seront dans un fichier nommés test.txt juste à côté de ce script.

6 Conclusion

Au vu du projet que nous avons crées, il semble évident que des améliorations sont possibles. Par exemple, nous aurions pu tenter de créer le MDT initialement demandé. Aussi, nous aurions aimé ajouter plus de paramètres permettant des expérimentations plus fines. Néanmoins, nous avons su faire preuve d'organisation et de travail d'équipe afin de fournir un programme complet sans pour autant accabler l'un des membres de notre groupe sous une charge de travail trop lourde. Mais la chose la plus importante que ce projet nous a appris, c'est le fonctionnement d'autres algorithmes de recherche et de résolution que les Minmax et Alphabeta appris il y a maintenant plus de 1 an. Un projet qui a su nous pousser dans nos retranchements afin de réfléchir à des questions non pas informatiques ni de programmation, mais des questions de réflexion purs et de choix d'algorithmes (par exemple pour la fonction d'évaluation du Minmax, comment évaluer un coup en sachant que l'adversaire va nous déstabiliser ?).

7 Références

1. Monte-carlo tree search wikipédia [en ligne] Disponible sur :
   https://fr.wikipedia.org/wiki/Recherche_arborescente_Monte-Carlo
2. The mdt algorithm [en ligne] Disponible sur :
   https://arxiv.org/abs/0904.0217
3. Algorithme MDT (Microsoft Decision Trees) [en ligne] Disponible sur :
   https://docs.microsoft.com/fr-fr/analysis-services/data-mining/microsoft-decision-trees-algorithm?view=asallproducts-allversions
4. Algorithme Minmax [en ligne] Disponible sur :
   https://fr.wikipedia.org/wiki/Algorithme_minimax
5. Elagage Alphabeta [en ligne] Disponible sur :
   https://fr.wikipedia.org/wiki/élagage Alpha Bêta
6. QUARTO CLASSIC [en ligne] Disponible sur :
   https://www.gigamic.com/jeu/quarto
7. An Artificial Intelligence for the Board Game 'Quarto !' in Java [en ligne] Disponible sur :
   http://suendermann.com/su/pdf/pppj2013.pdf