# Développement d'une intelligence artificielle pour un jeu à 2 joueurs

Sylvain BESNARD, Romain LE BORGNE,
Fabien BRAULT, Baptiste BIGNON

Encadreur : Christian RAYMOND

**Résumé**

Dans un monde où la puissance de calcul des ordinateurs ne cesse de s'accroître, les jeux deviennent de plus en plus perfectionnés et réalistes. Cette croissance de puissance a des conséquences sur la qualité des intelligences artificielles. En effet, cette augmentation permet d'exécuter les algorithmes de plus en plus rapidement et de pouvoir anticiper plus de coups. Tout ceci permet aux joueurs d'avoir l'impression de jouer contre un autre être humain, plutôt que contre une machine. Par exemple, le champion du monde d'échecs Garry Kasparov a été battu en 1997 par Deep Blue, un superordinateur spécialisé pour ce jeu.

## 1 Introduction

L'intelligence artificielle dans les jeux à 2 joueurs sans facteur aléatoire est le thème central de notre projet. Après avoir présenté son déroulement au cours de l'année, nous allons expliquer nos choix sur le jeu et le langage de programmation. Nous allons également présenter les algorithmes qui nous ont permis de réaliser cette intelligence artificielle et leur implémentation ainsi que les optimisations pour rendre le jeu plus rapide.

## 2 Présentation du projet

L'objectif de ce projet était de développer un jeu à 2 joueurs en y incorporant une intelligence artificielle. Ce projet s'est décomposé en 2 parties :

- La première partie consistait en l'étude bibliographique de l'algorithme permettant de réaliser cette intelligence artificielle : l'algorithme MinMax. Nous avons aussi étudier la coupure Alpha-Beta permettant d'accélérer cet algorithme. Nous nous sommes également mis d'accord sur le jeu que nous allions développer et sur le langage de programmation que nous allions utiliser.
- La deuxième partie de ce projet a été la réalisation pratique de ce jeu et l'application des algorithmes cités précédemment.

Nous avons choisi d'implémenter le jeu de dames parmi plusieurs jeux qui convenaient à cet algorithme. En effet, des jeux comme le morpion nous semblaient sans intérêt car les coups sont trop restreints. Le choix s'est fait entre les échecs et les dames, mais ce dernier nous était plus connu et ses règles sont plus simples.

Concernant le langage de programmation, nous avons choisi le C++, d'une part pour nous permettre d'apprendre ou d'approfondir ce langage, et d'autre part pour

bénéficier de la librairie Qt, une bibliothèque multiplateforme pour créer des interfaces graphiques utilisateur conçue pour être utilisée en C++, et d'Open MP, une interface de programmation pour le calcul parallèle, pour pouvoir créer des threads aisément.

# 3 Conception du support - le jeu de dames

## 3.1 Le modèle MVC

Notre projet suit le modèle MVC qui permet de séparer la partie modèle de données de la partie interface utilisateur, avec un contrôleur permettant de faire le lien entre ces 2 parties :

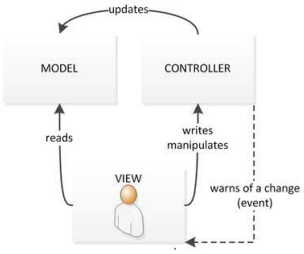


FIGURE 1 – Schéma modèle MVC (Model View Controller)

- Modèle :
  Il s'agit des classes où sont stockées les données du jeu. Classes Game, Checkerboard et Player (voir partie II - Le jeu)
- Vue :
  Il s'agit de l'interface graphique visible par l'utilisateur (voir partie III - le graphisme).
- Contrôleur :
  Il s'agit des classes faisant le lien entre l'interface graphique (l'utilisateur) et le jeu en lui même (voir partie III - Le graphisme).

## 3.2 Le jeu

La structure de notre code concernant le fonctionnement du jeu en lui-même n'utilise que les classes Game, Checkerboard et Player selon le schéma suivant :

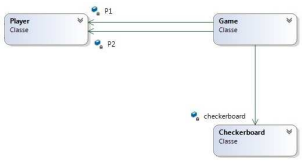


FIGURE 2 – Diagramme de classe

Au cours de l'exécution, le jeu comporte une seule instance de classe Game auquel est associée une instance de classe Checkerboard et deux instances de classes Player.

Les méthodes de la classe Game coordonnent le déroulement du jeu, celles de la classe Player exécutent les actions propres à chaque joueur tandis que celles de la classe Checkerboard exécutent les requêtes demandées sur le plateau de jeu.

### 3.3 Le graphisme

Comme nous l'avions décidé au début du projet, pour réaliser l'interface graphique, nous avons utilisé le framework Qt pour son efficacité mais aussi pour sa popularité qui permet de trouver de l'aide plus facilement sur internet en cas de besoin. Afin de faciliter son utilisation nous avons réalisé l'interface sous Qt creator et à l'aide de Qt Designer (qui permet de placer les éléments de base de la fenêtre visuellement).

Nous avons choisi de créer une interface simple : la partie gauche de la fenêtre est consacrée au plateau du jeu, tandis que celle de droite permet de choisir les différents réglages du jeu (joueurs humains ou IA, taille du plateau, utilisation ou non de la coupure alpha-beta ou du multithreading pour l'IA). Cela permet de commencer à jouer facilement et rapidement.

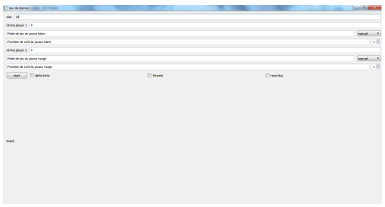

FIGURE 3 – Jeu de dames

## 4 L'intelligence artificielle

### 4.1 Les algorithmes MinMax et NegaMax

L'algorithme MinMax est applicable à des jeux à deux joueurs sans hasard où chaque joueur joue alternativement. Le déroulement du jeu peut être représenté par un arbre :

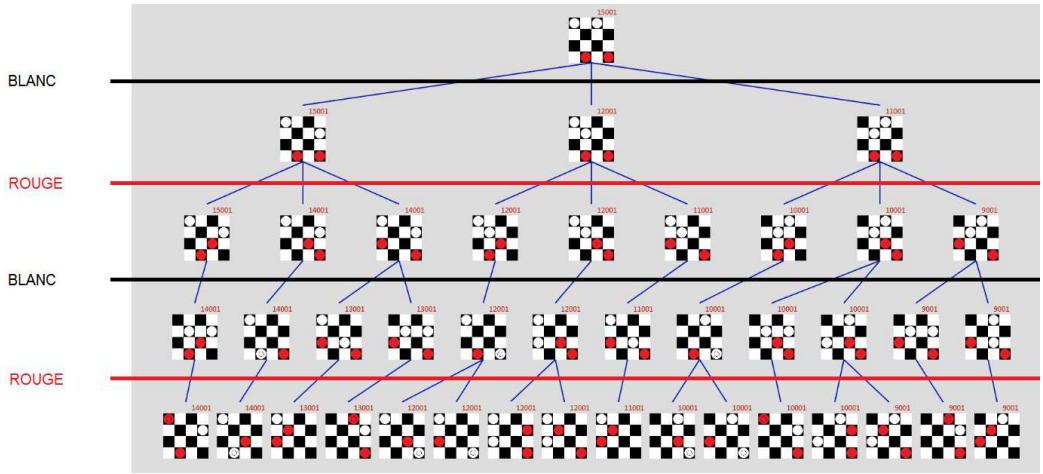


FIGURE 4 – Arbre avec MinMax, sans valeur et joueurs

La racine de l'arbre est la situation initiale. Chaque profondeur correspond à un tour de jeu, alternativement au joueur 1 et au joueur 2. Chaque nœud est une situation potentielle. Chaque arc représente le mouvement ayant permis de passer d'une situation à une autre. À chaque feuille est calculée la fonction de coût, la fonction permettant de calculer l'avantage pour un des joueurs, associée à la situation.

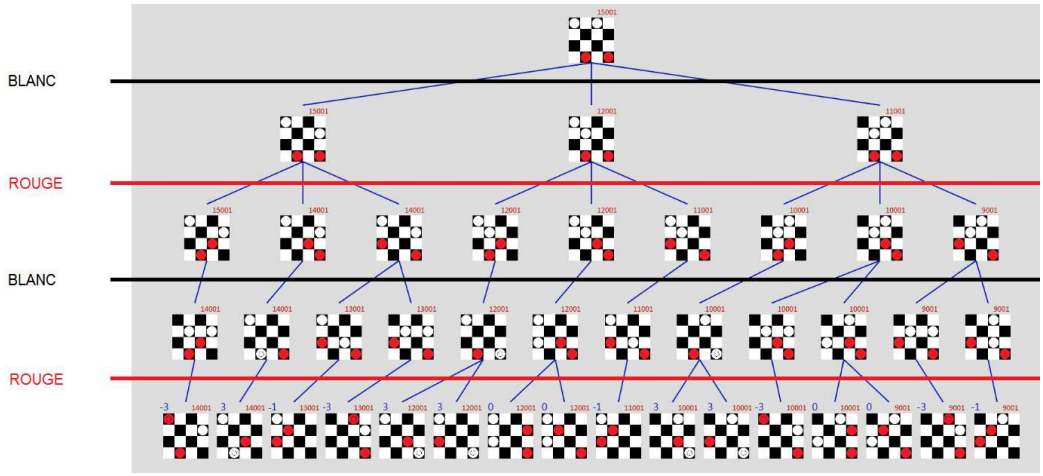


FIGURE 5 – Arbre avec MinMax, avec valeur sur les feuilles et joueurs

La valeur associée à une situation finale (une feuille) est ensuite remontée jusqu'à la racine en supposant que l'on choisisse la solution qui nous avantage le plus et que l'adversaire choisisse la solution qui nous avantage le moins.

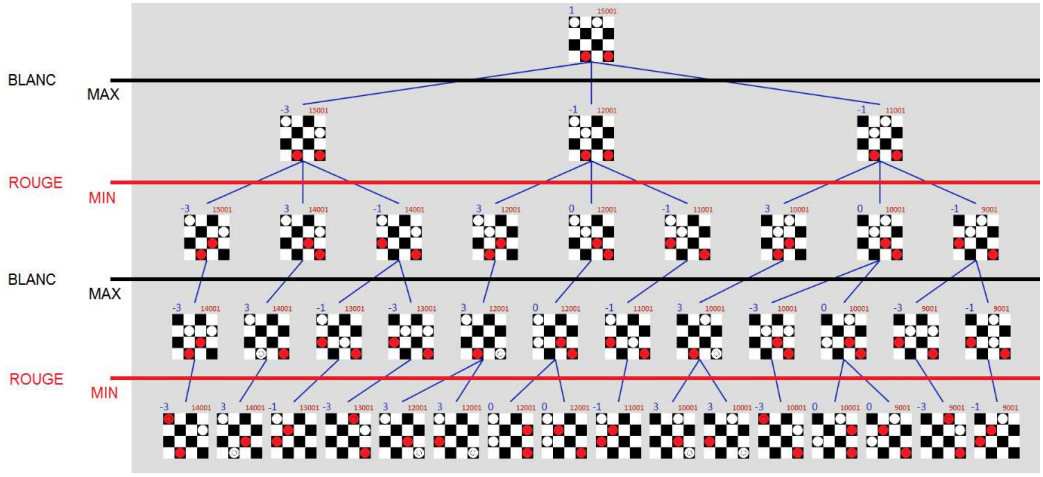


FIGURE 6 – Arbre avec MinMax, avec toute les valeurs, les joueurs et les choix MinMax

La valeur arrivée à la racine est la meilleure situation possible. Le premier arc du chemin allant de la racine à la feuille la plus avantageuse (ayant été remontée jusqu'à la racine) est le mouvement à réaliser.

L'évaluation se faisant selon les mêmes critères pour les 2 joueurs, la valeur trouvée pour un même damier selon que l'on se considère comme un joueur est égal à l'opposé de celle trouvée si l'on se considérait comme l'autre joueur.

Exemple :

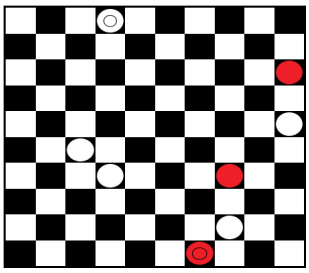

FIGURE 7 – Damier correspondant à la fonction de coût

Fonction de coût pour les blancs = X = 2

Fonction de coût pour les noirs = -X = -2

Selon l'algorithme MinMax et si l'on est le joueur blanc, si on simule l'action du joueur noir dans cette situation, alors on prendra l'action qui nous avantage le moins. En supposant que le joueur puisse comparer avec une situation de valeur Y, il va choisir : Min(2,Y)

L'algorithme NegaMax remplace la valeur de la fonction de coût par son opposé dans le cas où il s'agit d'une simulation d'un tour de jeu de l'adversaire. La valeur de la fonction est donc à chaque fois en fonction du joueur pour qui c'est le tour de jouer.

L'algorithme NegaMax ne va donc plus choisir alternativement la situation la plus avantageuse puis la moins avantageuse pour nous. Il va toujours choisir la solution la plus avantageuse pour le joueur à qui c'est le tour de jouer.

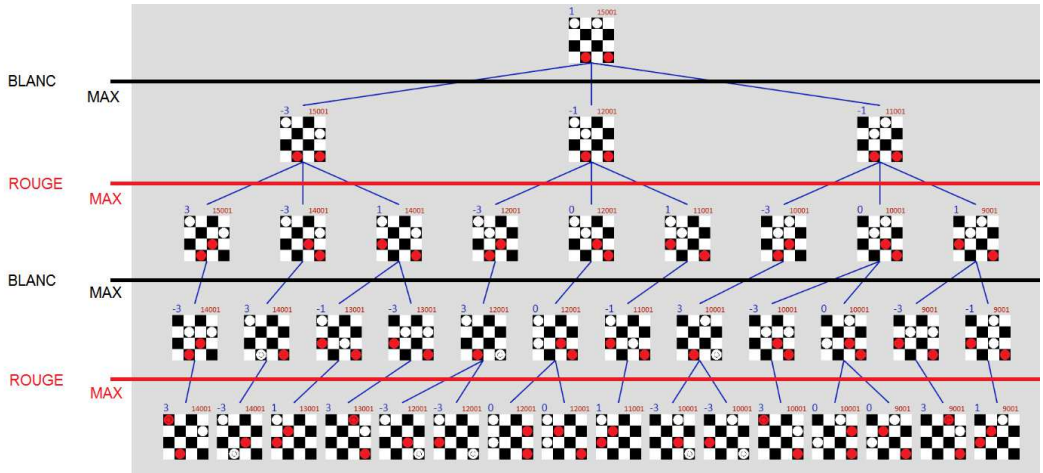


FIGURE 8 – Arbre avec NegaMax, avec toutes les valeurs, les joueurs et les choix MaxMax

L'algorithme NegaMax est donc une variante de l'algorithme MinMax permettant de toujours choisir le maximum au lieu d'alterner entre minimum et maximum. Il a pour seul but de simplifier l'implémentation.

### 4.1.1 Implémentation de l'algorithme

Voici le pseudo code de l'algorithme NegaMax :

```
function negamax(node, depth, color) {
if depth = 0 or node is a terminal node
     return color * the heuristic value of node
bestValue := -infini
foreach child of node
     val := -negamax(child, depth - 1, -color)
     bestValue := max( bestValue, val )
return bestValue
}

//Initial call for Player A's root node
rootNodeValue := negamax( rootNode, depth, 1)

//Initial call for Player B's root node
rootNodeValue := -negamax( rootNode, depth, -1)
```

Notre algorithme est donc une application de l'algorithme Negamax à notre jeu de dames :

- Node = Il s'agit des nœuds de l'arbre = les damiers.
- Depth = il s'agit de la profondeur restant à parcourir.
- Color = il s'agit du joueur à qui c'est le tour de jeu = blanc/noir = 1/-1.
- Node is a terminal node = lorsqu'un nœud est une feuille de l'arbre = lorsque la profondeur maximale a été atteinte ou que le damier admet un gagnant ou une égalité.
- The heuristic value of node = la valeur de la fonction de coût.
- BestValue = la valeur de fonction de coût maximal actuellement trouvée parmi les nœuds fils.
- Foreach child of node = pour chaque fils du nœud = pour chaque mouvement possible par le joueur ayant le trait sur le damier fils..
- val := -negamax(child, depth - 1, -color) = appel récursif de la fonction NegaMax pour chaque fils et inversement de la valeur de retour pour adapter la valeur au fait que le nœud actuelle est un joueur différent.
- bestValue := max( bestValue, val ) = sélection de la valeur maximale.
- return bestValue = on retourne la valeur maximale.

## 4.2 Les optimisations : la coupure alpha-beta et les threads

Après avoir fini le jeu de Dames en mode automatique, nous avons essayé d'optimiser le jeu pour que le temps d'exécution soit le plus rapide possible. Le premier moyen possible était d'améliorer l'algorithme MinMax en utilisant la méthode de la coupure alpha-beta. Cette méthode consiste à ne pas consulter certains nœuds lorsque l'on sait que le nœud père ne sera pas le coup optimal. Cette méthode nous a permis de passer d'une profondeur maximale de 4 à une profondeur maximale 6

dans la recherche du meilleur coup. Voici un diagramme du temps gagné en utilisant cette méthode :

| Nombre de coups à calculer | Sans alpha-beta | Avec alpha-beta |
|---|---|---|
| 1 | 140 ms | 140 ms |
| 2 | 537 ms | 536 ms |
| 3 | 3 853 ms | 3 562 ms |
| 4 | 18 641 ms | 15 203 ms |

Pour cette méthode, nous avons repris la méthode NegaMax à laquelle nous avons rajouté un paramètre (la valeur maximale actuelle au niveau du nœud père). Il faut noter que la coupure alpha-beta utilise quand même la fonction NegaMax pour tous les nœuds fils gauches.

Pour optimiser le temps d'exécution, nous avons aussi paralléliser les threads afin que le programme analyse plusieurs nœuds en même temps. Cette amélioration a été permise par l'utilisation d'Open MP, qui a été conçu afin de faciliter la mise ne place de calcul parallèle. Voici un diagramme du temps gagné grâce à cette méthode :

| Nombre de coups à calculer | Sans threads | Avec threads |
|---|---|---|
| 3 | 3 853 ms | 3 286 ms |
| 4 | 18 641 ms | 9 164 ms |
| 3 | 107 144 ms | 33 942 ms |
| 4 | 879 187 ms | 179 286 ms |

Nous avons aussi essayé de représenter graphiquement l'exécution de l'algorithme. Ainsi, cela nous permettait de mieux vérifier que la méthode Alpha-Beta fonctionnait (en voyant si toutes les positions étaient analysées ou non). Ce travail se sépare en 2 étapes :

- La première a été de créer un fichier texte contenant la taille du plateau et, pour chaque nœud, le contenu du damier, le temps d'exécution, sa valeur, son nombre de nœud fils et le nombre de noeud analysé.
- La deuxième est de créer un reporting graphique, codé en javascript via Canvas, et permettant de mieux visualiser les situations.

## 4.3 La fonction de coût

Une fonction de coût associée au damier à un moment précis de la partie, définie la situation comme bonne ou mauvaise pour l'un des deux joueurs. Cette définition se fait sous forme numérique, plus la valeur est élevée, plus la situation est avantageuse.

Dans un premier temps, nous avons défini une fonction de coût comparant le nombre de pions du joueur avec le joueur adversaire en donnant une plus grande importance aux dames :

Exemple :

- Valeur d'un pion = 1
- Valeur d'une dame = 3

Si l'on se considère comme le joueur blanc, nous avons 4 pions et 1 dame alors que les noirs ont 2 pions et 1 dame.

La valeur de la fonction de coût est donc :

Fonction de coût = (Nombre de pions blancs * valeur d'un pion + nombre de dames blanches * valeur d'une dame) - (nombre de pions noirs * valeur d'un pion + nombre de dames noires * valeur d'une dame)

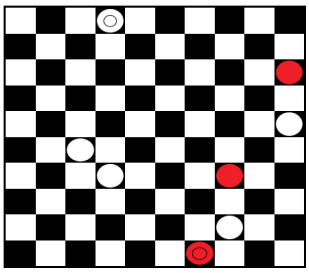

FIGURE 9 – Exemple de damier

= 4*1 + 1*3 - (2*1 + 1*3)
= 2

La valeur étant positive la situation est avantageuse, donc plus la valeur est élevée, plus la situation est avantageuse.

Pour améliorer l'algorithme, nous avons aussi essayé de perfectionner la fonction de coût. En effet, la fonction de départ ne regardait que le nombre de pions de chaque joueur. Cependant, la position des pions ont aussi leur importance. Donc nous avons essayé de prendre cela en compte dans le jugement de la position. Au final, nous avons réalisé plusieurs fonctions de coût que nous avons fait se rencontrer plusieurs fois pour voir la meilleure d'entre elles. Voici un tableau des confrontations :

| Confrontations | 1 | 2 | 3 | 5 |
|---|---|---|---|---|
| 1 | 1/2 | 0 | 1 | 1 |
| 2 | 1 | 1/2 | 1 | 1 |
| 3 | 0 | 0 | 0 | 1/2 |
| 5 | 0 | 0 | 1 | 0 |

Dans ce tableau :

- 0 correspond à une défaite du joueur en ligne
- 1/2 correspond à un match nul
- 1 correspond à une victoire du joueur en ligne

## 5 Conclusion

Nous avons donc atteint notre objectif premier qui était de créer une IA pour le jeu de Dames. Cet algorithme peut calculer jusqu'à 6 coups en avance.

On peut cependant noter que certaines améliorations pourraient rendre l'IA encore plus rapide. Par exemple, on pourrait rajouter une fonction qui, avant d'utiliser la fonction de coût, vérifie si ce nœud n'existe pas autre part dans l'arbre. On éviterait ainsi de refaire l'étude de tous ses nœuds fils.

Dans le cadre de l'étude pratique, nous avons donc établi des fondations idéales pour poursuivre l'expérience dans deux directions différentes, mais néanmoins complémentaires que sont la modélisation de jeux et l'Intelligence Artificielle.