Rapport de projet tuteuré

Puissance 4 dotée d'une IA

Membres :

Mark SOUS
Gaston LIONS
Yanis DE SOUSA ALVES
Fabien LE BEC

Tuteur :

Pierre VALARCHER

DUT INFO 2ème année

2019/2020

Sommaire

1. Introduction

1.1 Sujet

L'objectif du projet est de réaliser le jeu puissance 4 en y ajoutant une autre dimension, la profondeur. L'application propose à l'utilisateur d'affronter un autre joueur et différentes intelligences artificielles, à la fois en local et en réseau. En réseau signifie que deux joueurs peuvent jouer ensemble sur deux machines différentes.

Les règles de ce jeu sont celles du jeu classique "puissance 4", comme c'est un puissance 4 avec plusieurs profondeurs, de nouvelles règles sont disponibles. En plus des combinaisons gagnantes classiques telles que l'alignement de 4 jetons de la même couleur en horizontal, vertical ou diagonal, cette application permet de gagner en alignant 4 jetons en horizontal et en diagonal à travers les profondeurs.

Les dimensions de notre grille de jeu sont 5 lignes, 6 colonnes et 5 profondeurs. Nous avons volontairement choisi de réduire le nombre de colonnes et lignes par rapport au jeu classique afin que la complexité du jeu ne devienne pas trop grande suite à l'ajout d'une nouvelle dimension. Cette nouvelle dimension est de taille 5, taille qui nous semblait être un bon compromis entre offrir diverses situations de victoires (pour une taille 4, il n'y aurait qu'une seule façon d'aligner 4 jetons en profondeur) et ne pas faire exploser la complexité du jeu.

1.2 Pourquoi ce sujet ?

L'équipe projet avait tout d'abord pour objectif de réaliser une intelligence artificielle. Après discussion auprès d'un professeur, il s'est avéré qu'un jeu serait le meilleur environnement pour y inclure un système intelligent. Dans un second temps, le choix important à faire était le jeu, l'équipe a décidée de développer un jeu simple et connu de tous. Ceci permettrait alors de consacrer plus de temps à l'intelligence artificielle et à d'autres aspects du projet comme le réseau. L'idée d'un puissance 4 est alors apparu et a été rapidement accepté par l'ensemble des développeurs.

2. Méthode de travail

2.1 Organisation de l'équipe

L'équipe de ce projet est composée de quatre personnes :

- Fabien LE BEC
- Yanis DE SOUSA ALVES
- Gaston LIONS
- Mark SOUS

Le projet de ce puissance 4 se décompose en trois parties : l'interface graphique, le réseau et l'intelligence artificielle. Le groupe s'est alors séparé en trois puisque quatre personnes sur une seule partie aurait été contreproductif. Gaston s'est chargé de l'interface graphique, Mark du réseau, Yanis et Fabien de l'intelligence artificielle (figure 1).
Bien que le groupe se soit décomposé en trois, il est resté en étroite communication à chaque étape du projet et face aux divers problèmes rencontrés par chacun.

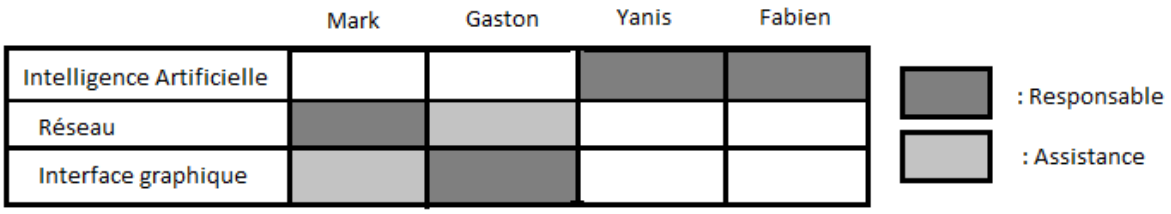

Figure 1 : *Tableau de répartition des tâches*

2.2 Diagrammes de Gantt

Voici le diagramme de Gantt, l'un des outils les plus efficaces pour représenter l'état d'avancement des différentes tâches qui constituent un projet, réalisé lors des premiers jours du projet (figure 2.1).

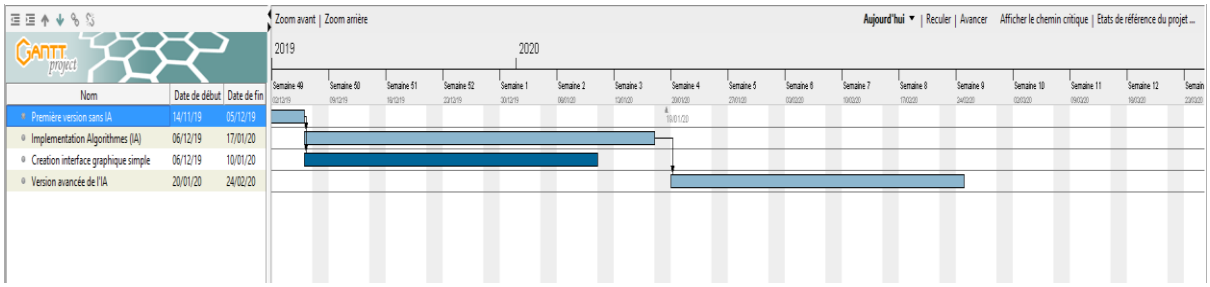

Figure 2.1 : *Diagramme de Gantt initial.*

Il y a des différences entre l'état d'avancement prévisionnel du projet et son avancement réel, voici son véritable avancement (figure 2.2)

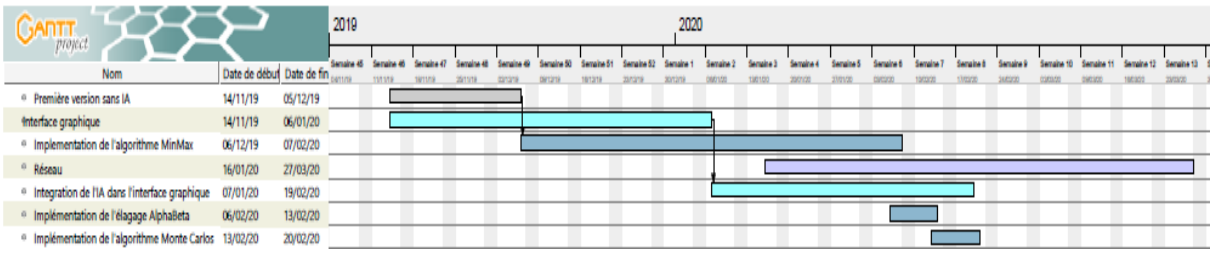

Figure 2.2 : *Diagramme de Gantt final.*

2.3 Réunion

Durant ce projet, l'équipe s'est rassemblée deux fois avec son tuteur Monsieur VALARCHER.
La première réunion s'est déroulée au début du projet pour établir correctement les différentes "fonctionnalités" de celui-ci comme l'interface graphique, le réseau et l'IA. C'est à cette réunion qu'il a été décidé de créer un puissance 4 avec des profondeurs puisque choisir une version classique avait été jugé, par le tuteur et à décision unanime, peu original (de nombreux projets de puissance 4 avec l'implémentation d'algorithmes existent déjà). Cette nouvelle dimension donne au joueur, de nouvelles opportunités stratégiques et représentait, pour l'équipe projet, un paramètre important à ne pas négliger.

Lors de la seconde réunion, après une présentation de l'avancement du projet, l'équipe projet a abordée plusieurs sujets. L'un d'eux était un problème concernant des variables partagées dans la partie réseau lors de la connexion de plusieurs clients, une solution a été trouvée. Le tuteur a proposé l'idée d'ajouter, au projet, une nouvelle vision du jeu pour l'utilisateur facilitant la visualisation des combinaisons possibles entre profondeurs.

2.4 Design pattern

Dans ce projet la structure du code a été pensé pour respecter le design pattern MVC : modèle, vue, contrôleur (figure 3).

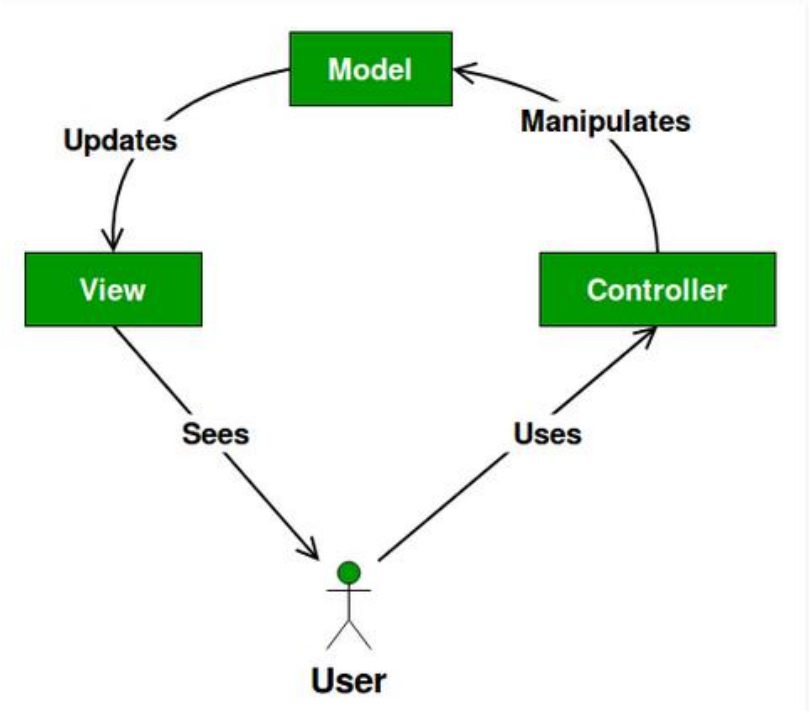

Figure 3 : *Schéma MVC.*

3. Interface graphique

L'interface graphique a été créée avec l'aide de la bibliothèque Swing.

3.1. Menus de l'application

Les menus de l'application ont été créés avec les classes héritant de JComponent comme JButton ou JLabel, puis de la classe GridBagConstraint qui est utilisée dans la disposition des éléments sur la fenêtre.

Figure X : *Menu principal*

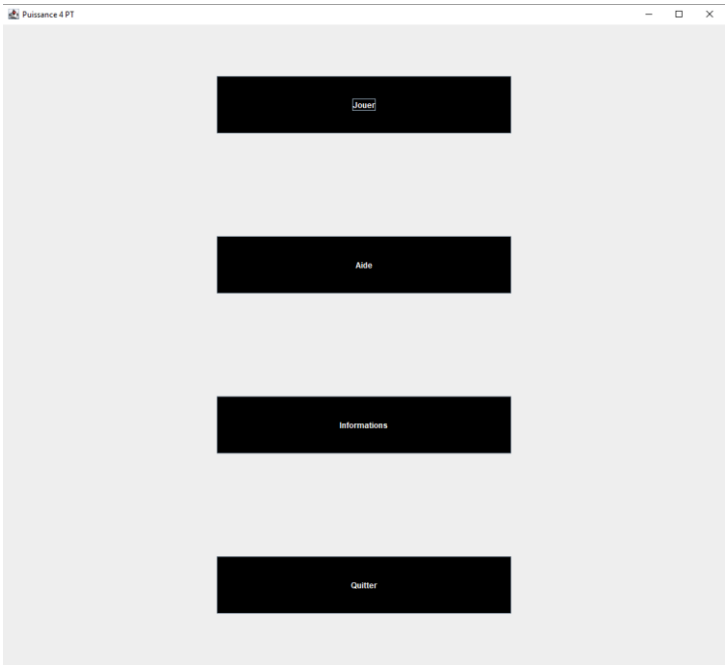

Figure 4 : *Menu principal.*

Au lancement de l'application, l'utilisateur se trouve sur le menu principal (figure X). L'utilisateur clique sur le bouton « Jouer », un sous-menu s'affiche, celui-ci permet de choisir si l'on souhaite jouer contre une IA, un autre joueur en réseau ou en local (figure X). S'il décide de cliquer sur « Aide », une nouvelle fenêtre va s'ouvrir affichant une indication en dessin sur les nouvelles possibilités de gagner une partie (figure X). S'il clique sur « Information », une nouvelle fenêtre se lance montrant les informations concernant les développeurs de l'application et de l'origine de celle-ci.

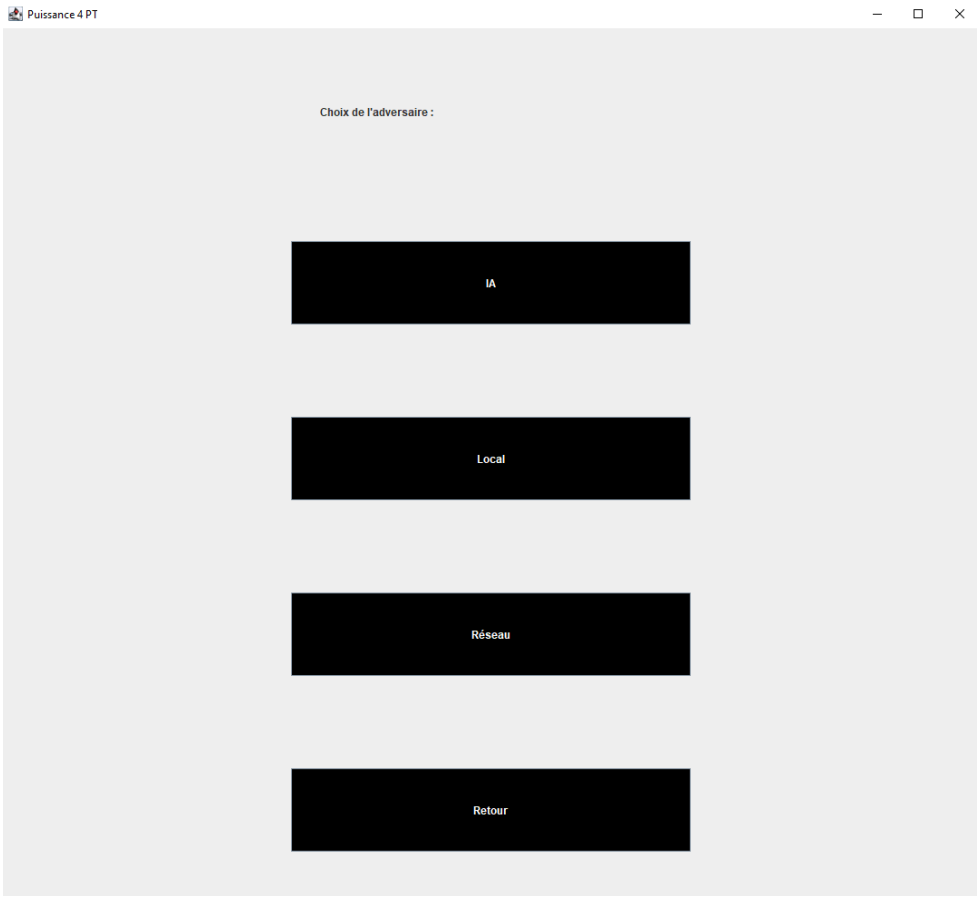

Figure 5 : *Menu jouer.*

Une fois arrivé sur ce menu vous devez donc choisir si vous désirez faire une partie contre une IA, ou une partie en réseau contre un autre joueur ou alors en local.

- Dans le premier cas deux nouveaux sous-menus vont s'afficher afin de préciser l'algorithme souhaité (figure 7) ainsi que le niveau de l'algorithme (figure 8), plus il est élevé plus il sera dur à battre.
- Dans le deuxième cas, une JOptionPane va s'afficher sur laquelle il faut préciser l'adresse du serveur auquel on souhaite se connecter. Si cela échoue un message d'erreur apparais, sinon une fenêtre s'ouvre en attendant que le deuxième joueur se connecte au même serveur. Une fois les deux joueurs connectés, la partie se lance.
- Dans le troisième cas, la partie se lance.

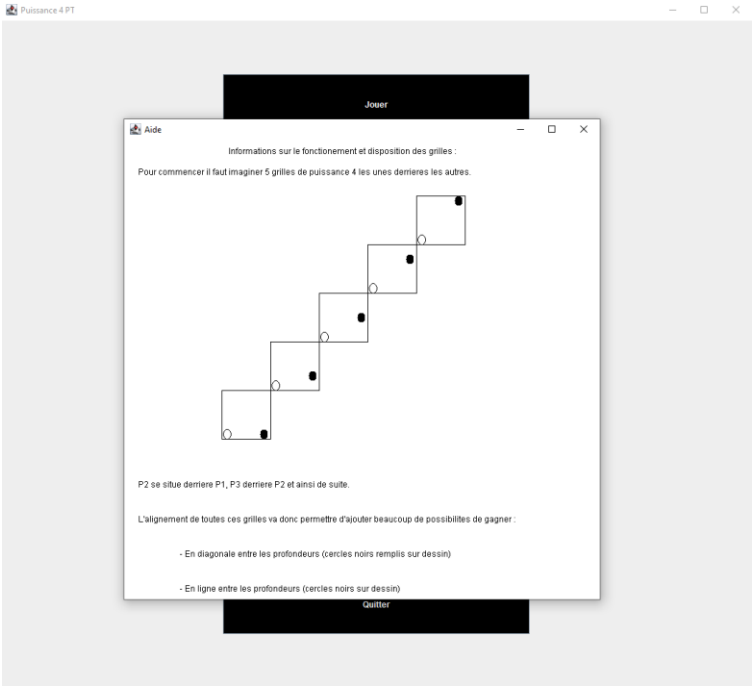

Figure 6 : *Fenêtre Aide.*

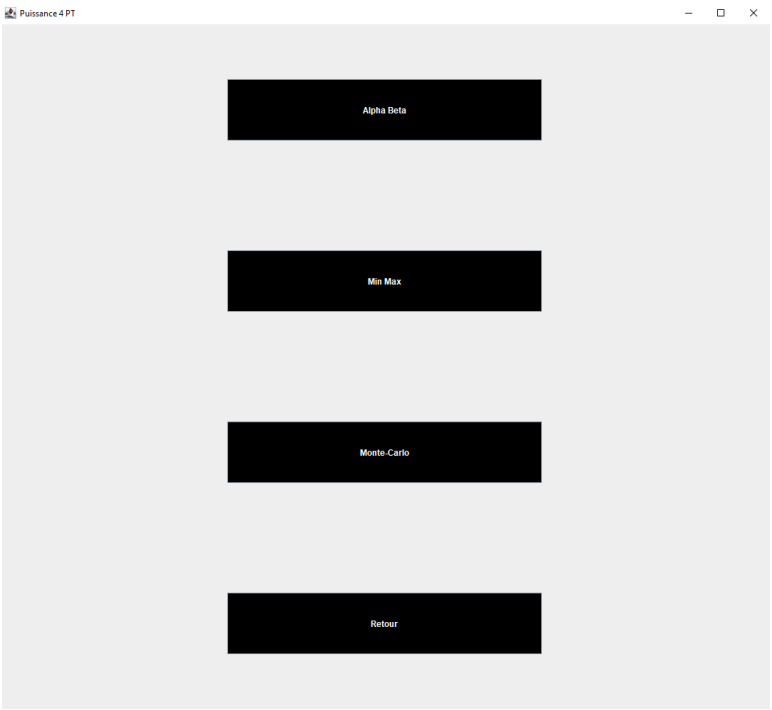

Figure 7 : *Menu Algorithme.*

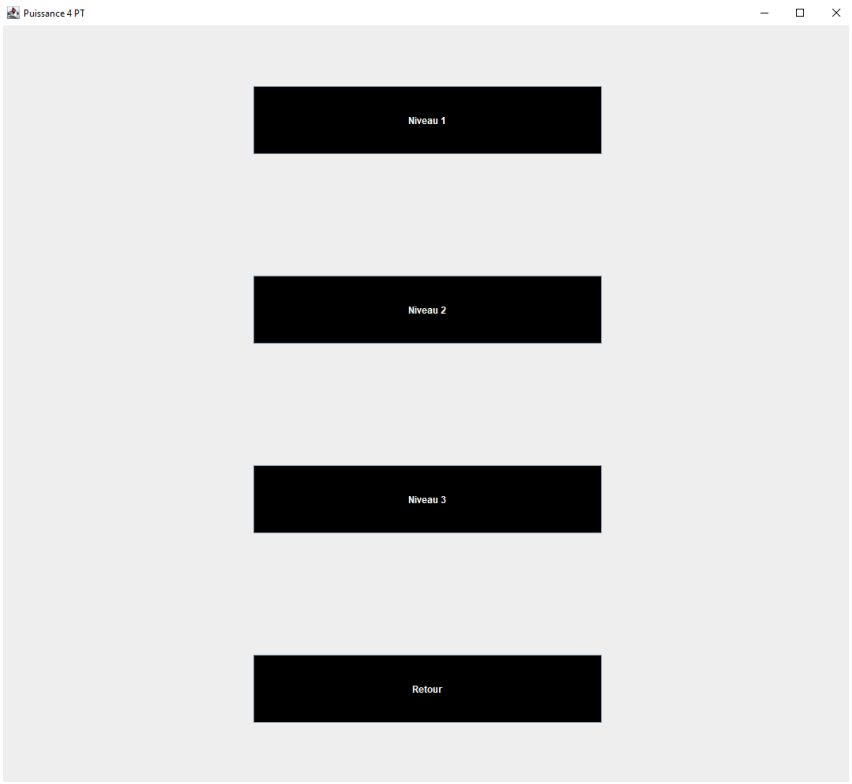

Figure 8 : *Menu Niveau.*

3.2. Illustration d'une partie

3.2.1 Structure du code

L'interface graphique d'une partie a été réalisée entièrement grâce à la méthode « paintComponent » qui permet de dessiner des formes géométriques, images, textes, dans un élément. Tous ces dessins sont faits dans la classe DessinGrille qui comme son nom l'indique s'occupe de dessiner les différentes grilles et leurs cercles. L'application respecte le modèle MVC (Modèle,Vue,Controller), DessinGrille est donc considérée comme une « vue » et a donc besoin d'une ou plusieurs classes «model » qui vont stocker toutes les données dont elle aura besoin pour gérer ses dessins. On retrouve donc la classe Partie qui va permettre de connaître le nombre de tours effectué dans la partie, s'il existe un gagnant ou non, à qui appartient le tour actuel, les différents joueurs. Depuis cette classe, on peut aussi accéder à la classe Grille qui permet de connaître chaque case d'une grille… Elle va aussi faire appel à la classe InfoGrille qui est utilisée pour calculer les positions et tailles des éléments qui vont être dessinés dans DessinGrille. InfoGrille possède les méthodes permettant de gérer l'animation d'un jeton qui tombe dans la grille, l'affichage du jeton de prévisualisation au-dessus de la colonne où la souris de l'utilisateur se trouve puis plusieurs méthodes gérant la responsivité du dessin.

3.2.2 Animation de la chute d'un jeton

Cet événement est appelé lorsque l'utilisateur clique sur une colonne d'une grille où il a la possibilité de jouer (voir la classe ActionSourisEnJeu). Pour faire cette animation, il a fallu initialiser une position Y pour un jeton, puis de la baisser petit à petit afin de le faire tomber jusqu'à la case où l'utilisateur a décidé de jouer. Pour réaliser ceci, il a fallu créer un Thread afin de pouvoir réaliser le calcul baissant la position du jeton simultanément de son affichage.

3.2.3 Présentation en images

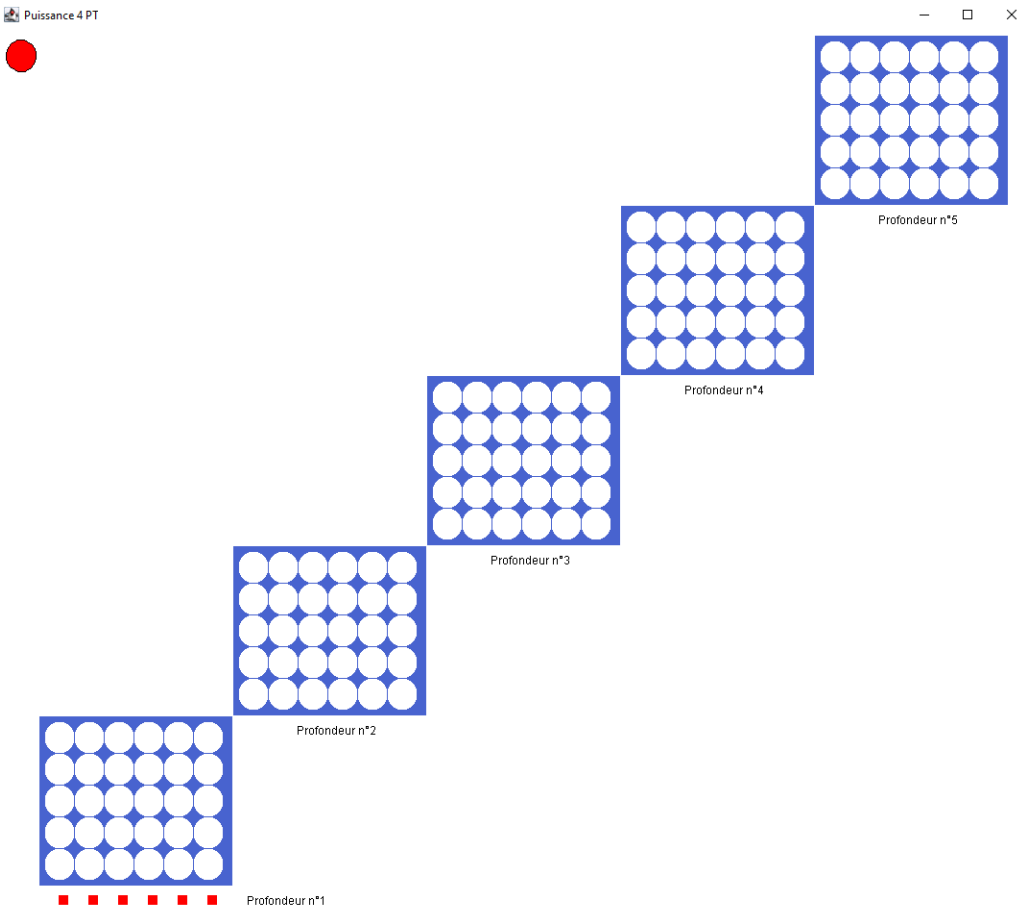

Figure 9 : *Image partie.*

- Les différentes profondeurs sont placées en diagonales afin de représenter l'effet de profondeur.
- Les cercles blancs à l'intérieur représentent une case (deviennent jaune ou rouge si un joueur à jouer dans celle-ci).
- Le dernier jeton joué apparaît avec des bordures plus foncées.
- Le cercle rouge en haut à gauche de la fenêtre représente à qui est le tour actuel.
- Les petits rectangles en bas de la profondeur n°1 permettent d'avoir une vue différente et facilitant la visualisation des combinaisons possibles entre profondeurs (figure 10).

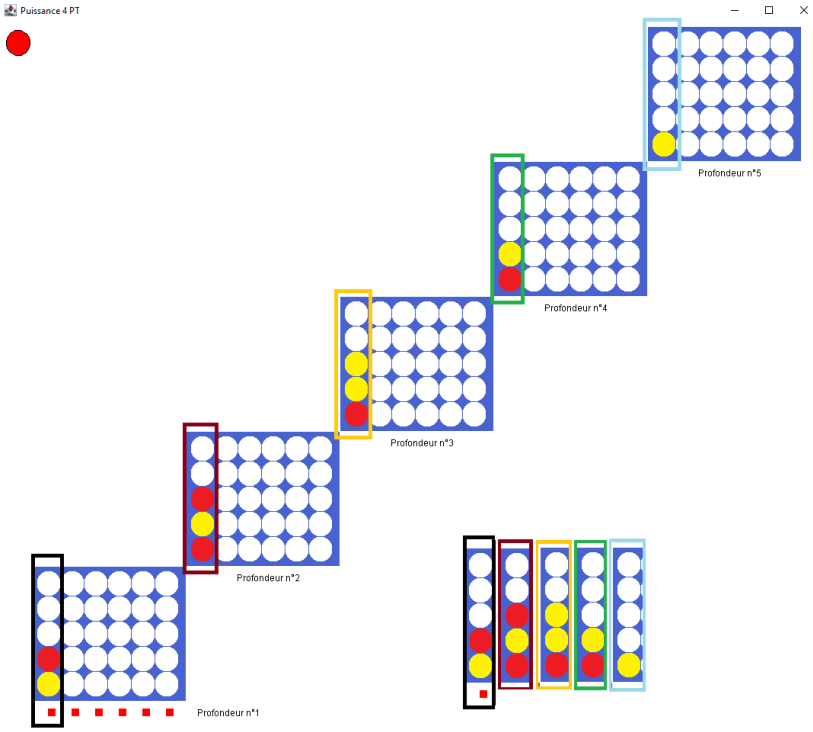

Figure 10 : *Montage explication vue.*

Si l'on clique sur le rectangle de la 1ère colonne, l'affichage va donc regrouper la première colonne de chaque profondeur afin de créer une grille en les mettant les unes à côtés des autres (figure 11). L'image de dessus montre cet événement, cette image est un montage

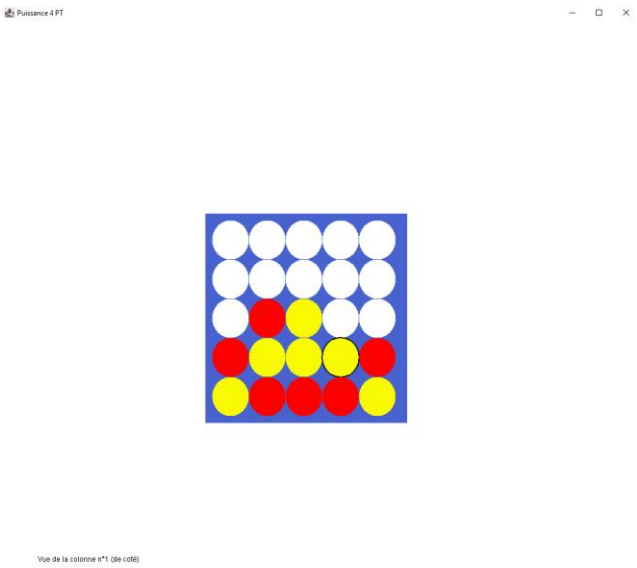

Figure 11 : *Image 2nd vue.*

L'utilisateur ne peut pas jouer depuis cet affichage. Il lui suffit de cliquer n'importe où dans la fenêtre afin de revenir à la vue principale de la partie.

3.2.4 Déroulement

1. L'utilisateur clique sur la colonne du dessin où il souhaite placer son jeton.
2. La méthode « mouseClick » de la classe ActionSourisEnJeu est appelée. On récupère la position X et Y du clic puis on calcule la colonne dans laquelle le jeton doit être ajouté.
3. Une fois celle-ci récupérée la méthode « jouer » de ControllerJeu est appelée, dans un premier temps elle va tenter d'insérer le jeton dans la grille à l'aide de la méthode « inserer » de la classe Grille, celle-ci retourne 0 si la colonne est pleine ou 1 si le jeton a bien été inséré. Une fois cette étape passée, la méthode « jouer » appelle « qqunGagne » de Grille qui va vérifier si le jeton ajouté a permis de gagner ou non. Si ce n'est pas le cas « nouveauTour » de Partie est appelé, qui donne la main au second joueur, et « jouer » retourne 1 signifiant que la partie continue. Mais s'il existe un gagnant « jouer » retourne 2 et va assigner le vainqueur à l'aide de la méthode « setVainqueur » de Partie.
4. La combinaison gagnante apparaît en vert et le nombre de coups nécessaires pour gagner ainsi qu'un bouton pour revenir au menu principal s'affichent (figure 12).

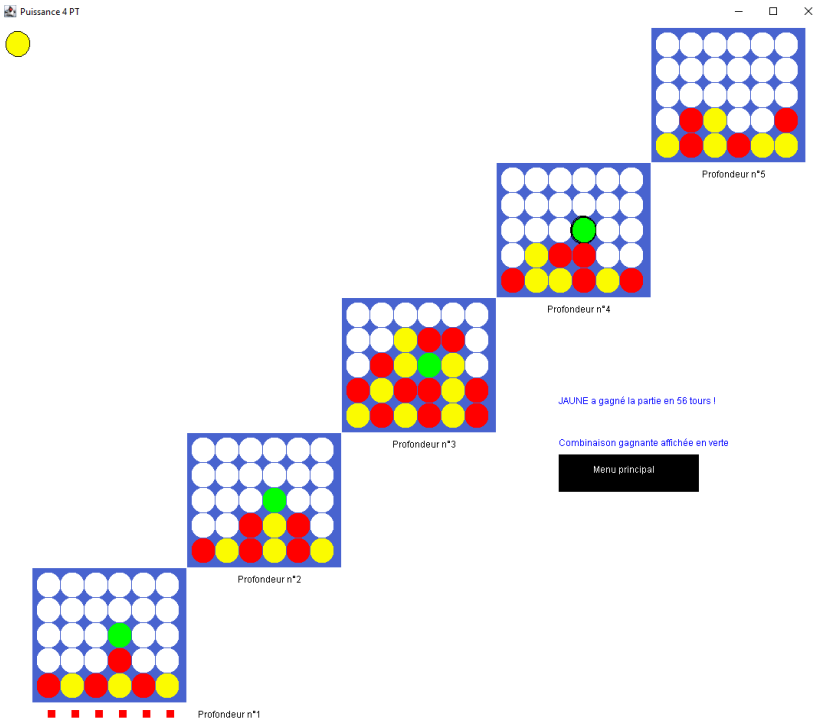

Figure 12 : *Image fin de partie.*

3.3 Diagrammes

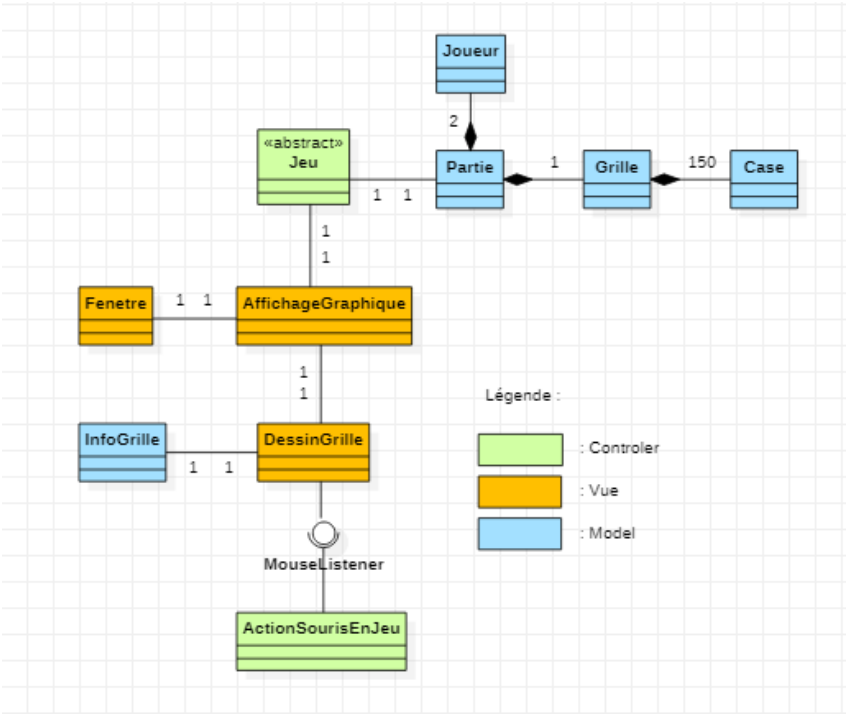

Figure 13 : *Diagramme de classes pour l'interface graphique.*

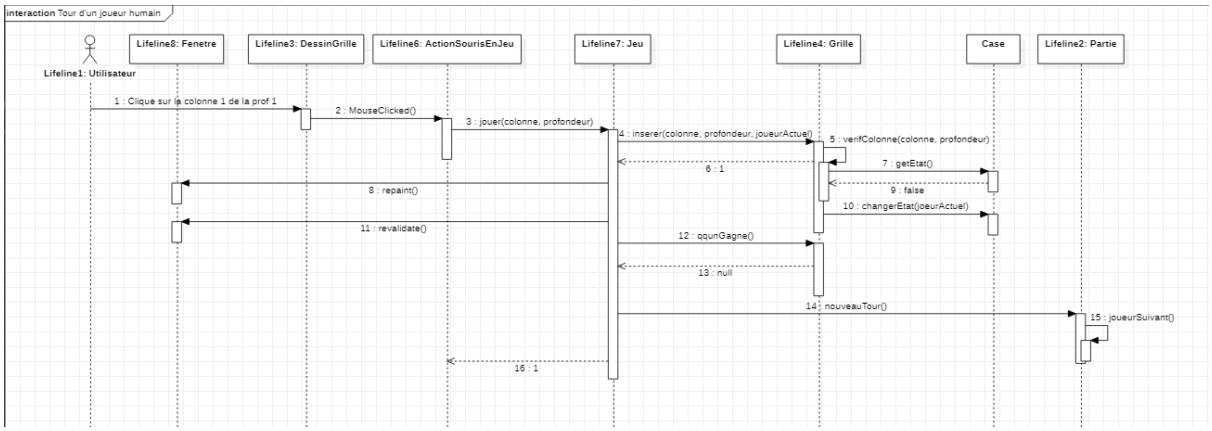

Figure 14 : *Diagramme de séquence du tour de jeu d'un joueur Humain.*

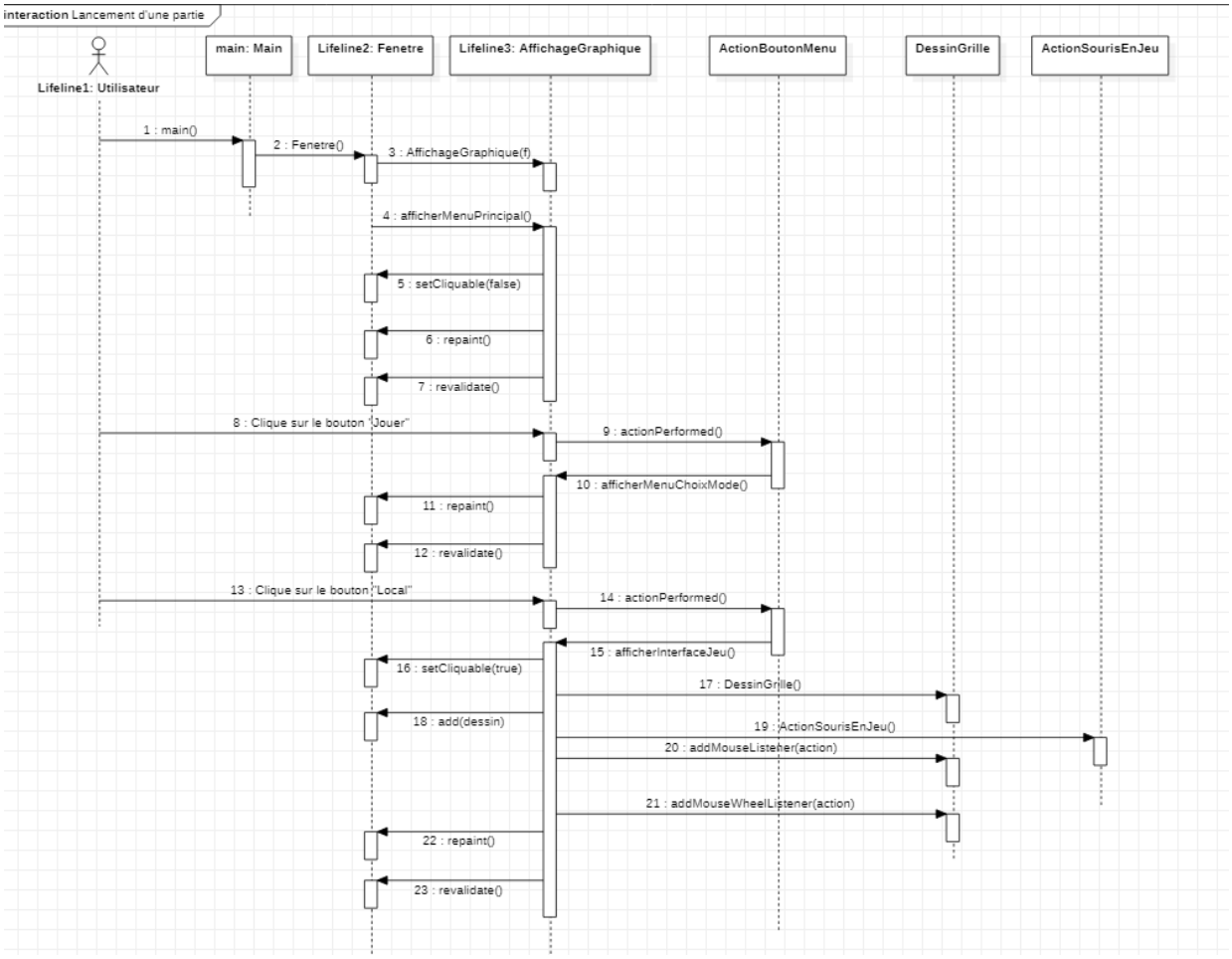

Figure 15 : *Diagramme de séquence du lancement d'une partie.*

3.4 Améliorations envisageables

- Donner la possibilité à l'utilisateur de pouvoir jouer depuis la 2nd vue (figure X).
- Retravailler la disposition et taille des différentes grilles afin d'améliorer l'effet de profondeur.
- Faire une interface graphique 3D

3.5 Difficultés rencontrées

- Gérer la responsivité tout en gardant un affichage propre et visible.
- Comprendre le problème de multitâches lors de l'animation du jeton qui tombe.
- L'interface graphique a été dans un premier temps créée à part des classes comportant toutes les variables sur la partie actuelle et de la grille. La mise en commun des deux a donc pris beaucoup de temps mais a permis de modifier, simplifier ou ajouter des variables, idées ou méthodes.

- Un changement de structure des classes a été fait en fin de projet et a donc demandé de revoir chaque appel de méthode, nommage de classe dans les différents fichiers de l'interface graphique et du jeu en lui-même.

4. Réseau

4.1 Les applications réseaux

Depuis les débuts d'internet, le nombre d'applications réseau a considérablement augmenté. Elles ont beaucoup évolué et apparaissent sous plusieurs formes :

- Messagerie électronique,
- Accès aux terminaux distants,
- Transfert de fichiers,
- Jeu en ligne …

Ces applications sont basées sur la communication entre deux entités : le client et le serveur.

4.2 Principe du client/serveur

Ce principe repose sur une communication d'égal à égal entre les applications. Cette communication est réalisée par dialogue entre deux processus :

- Un processus Client
- Un processus Serveur

Ces processus ne sont pas sur la même machine, ils ne sont donc pas identiques. Ils forment un système coopératif se traduisant par un échange de données.
Le client est à l'origine de l'échange, c'est lui qui l'initie en envoyant une requête au serveur.
Le serveur, quant à lui, est toujours à l'écoute d'une requête provenant d'un client éventuel.

4.3 Serveur

Un programme serveur tourne en permanence, attendant des requêtes. Il peut répondre à plusieurs clients en même temps. Le serveur nécessite une machine :

- Robuste ;
- Rapide ;
- Qui fonctionne 24h/24.

Pour les gros serveurs (c'est-à-dire les serveurs qui doivent recevoir des requêtes d'un très grand nombre de clients), la présence d'administrateurs systèmes et réseau est obligatoire.
Quelques exemples de serveurs :

- Serveur de fichiers (NFS, SMB) ;

- Serveur d'impression (lpd, CUPS) ;
- Serveur de calcul ;
- Serveur d'applications ;
- Serveur de bases de données ;
- Serveur de temps ;
- Serveur de noms (annuaire des services).

4.4 Client

Tout d'abord, le client désigne la machine sur lequel est exécuté le logiciel client (applications, logiciels, sites…).
Comme dit précédemment, le client envoie des requêtes au serveur, il attend donc une réponse à celle-ci.
Il existe 3 types de clients :

- le client léger ;
- le client lourd ;
- le client riche.

Le client léger est une application où le traitement des requêtes du client est entièrement réalisé par le serveur : le client recevra une réponse prête qu'il pourra directement utiliser comme ressource.
Le client lourd est une application où le traitement des requêtes du client est partagé entre le serveur et le client : le client recevra une réponse qu'il devra exploiter avant de l'utiliser comme ressource.
Le client riche est un client léger plus avancé : le traitement des requêtes du client est majoritairement effectué par le serveur, le client reçoit directement des réponses semi-utilisable qu'il finalisera. Le client riche permet de mettre en œuvre des fonctionnalités comparable à celle d'un client lourd.

4.5 Modèle client/serveur

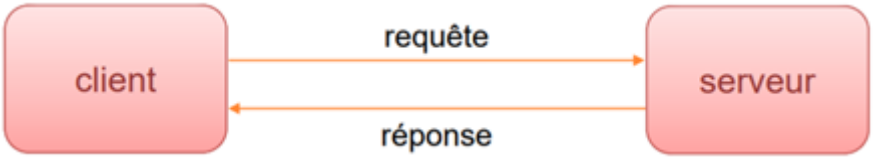

Figure 16 : *schéma de communication entre le client et le serveur.*

Ici, le client envoie une requête au serveur : le client demande l'exécution d'un service.
Le serveur réalise ce service et renvoie la réponse (figure 16).

La communication entre le client et le serveur se fait par messages (chaîne de caractères) (figure 17).

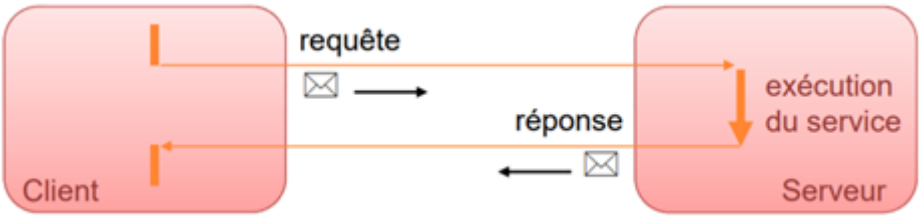

Figure 17 : *détails du schéma de communication entre un client et un serveur.*

Les requêtes peuvent être traitées selon plusieurs modes par le serveur : le mode itératif et le mode concurrent.
Pour le mode itératif : le serveur traite les requêtes les unes après les autres.

Pour le mode concurrent : le serveur crée des processus léger (threads) pour répondre aux clients. Ainsi, chaque processus léger peut gérer le client qui lui est attaché et le serveur peut continuer à recevoir les clients.

4.6 Application au Puissance 4

Pour la partie réseau du puissance 4 le principe est le même : il y a un serveur qui attend la connexion d'un ou plusieurs joueurs (figure 18).

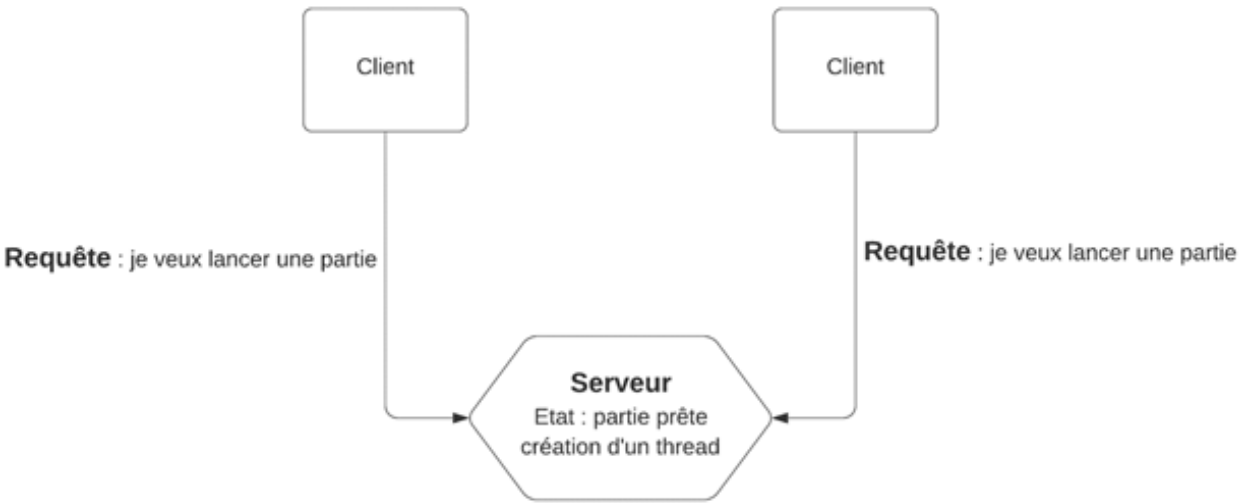

Figure 18 : *schéma montrant la mise en relation entre les clients et le serveur.*

Si deux joueurs ont demandé au serveur de jouer alors le serveur va créer un processus léger qui va s'occuper de la partie. Ainsi le serveur peut continuer de recevoir des clients (figure 19).

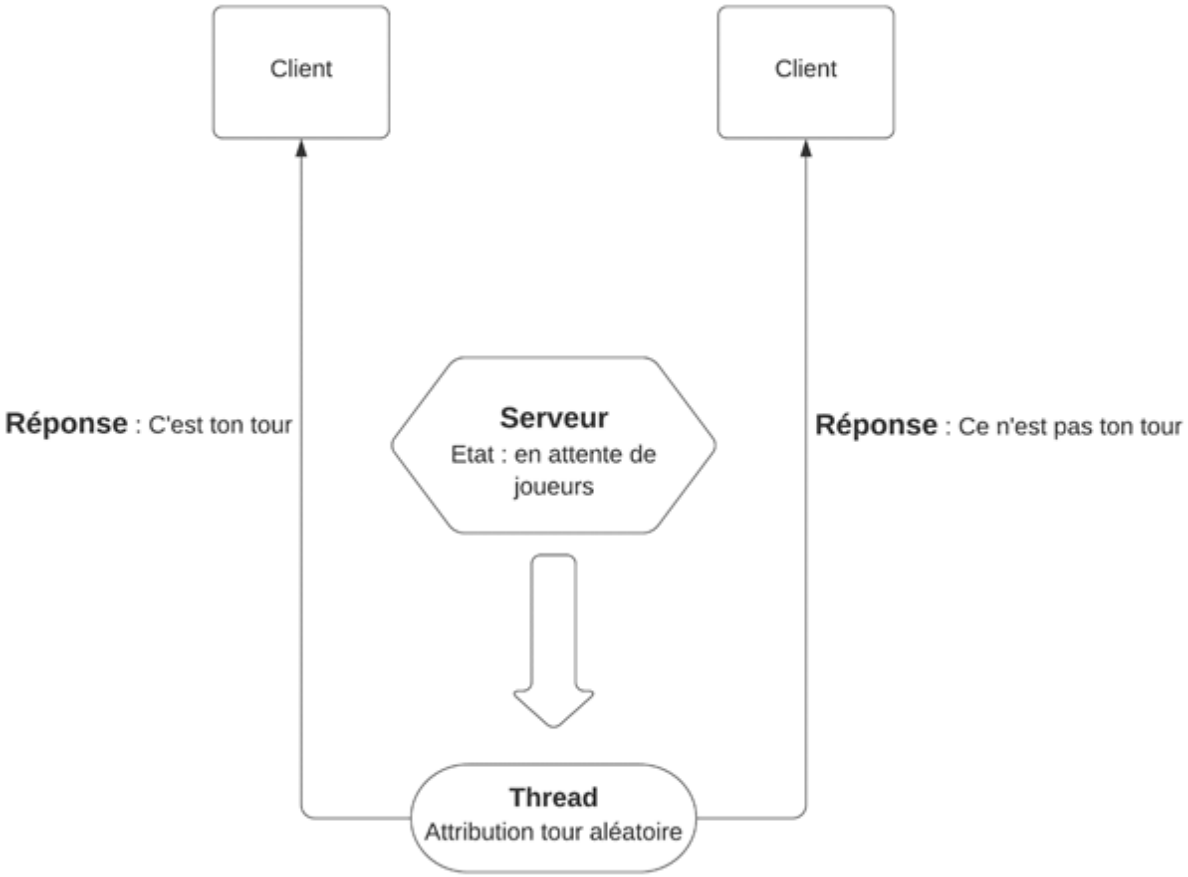

Figure 19 : *Création d'un thread pour une partie*

Lorsque le serveur a attaché un thread et deux clients, la partie peut commencer (figure 20).

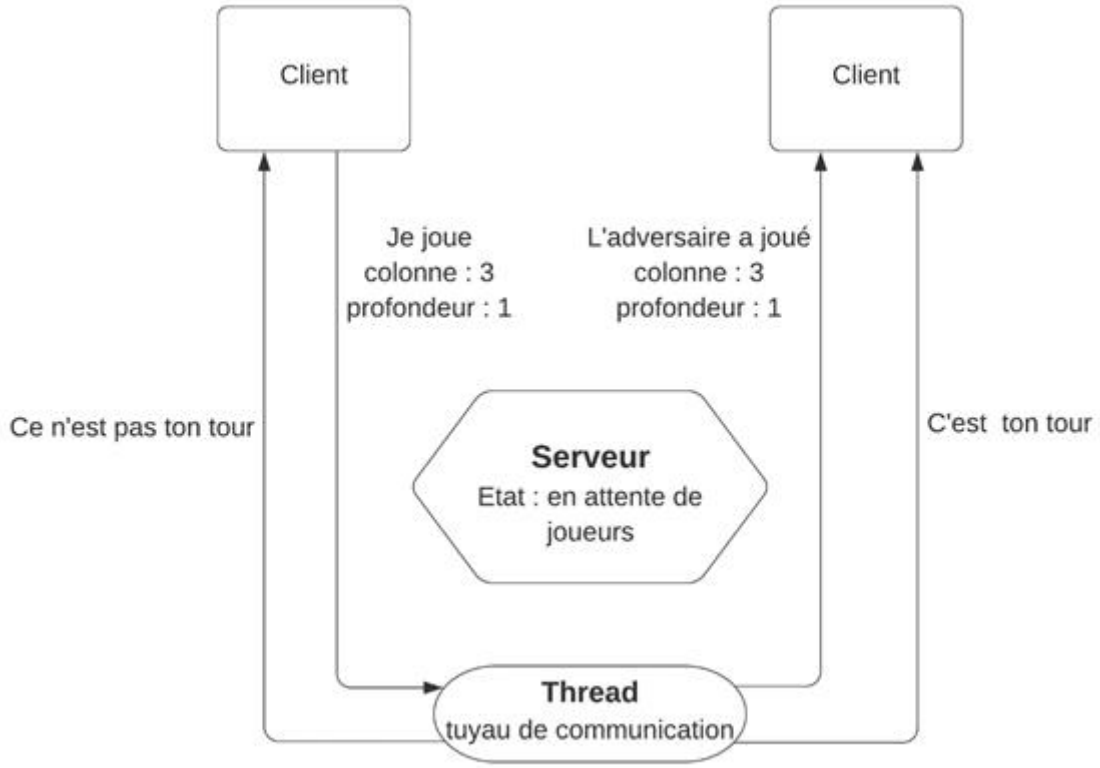

Figure 20 : *Déroulement d'une partie*

Ici, le thread ne sert que de tuyau de communication, il transmet des messages d'un client vers l'autre et s'occupe de gérer le tour de chaque joueur (figure 20 et 21).

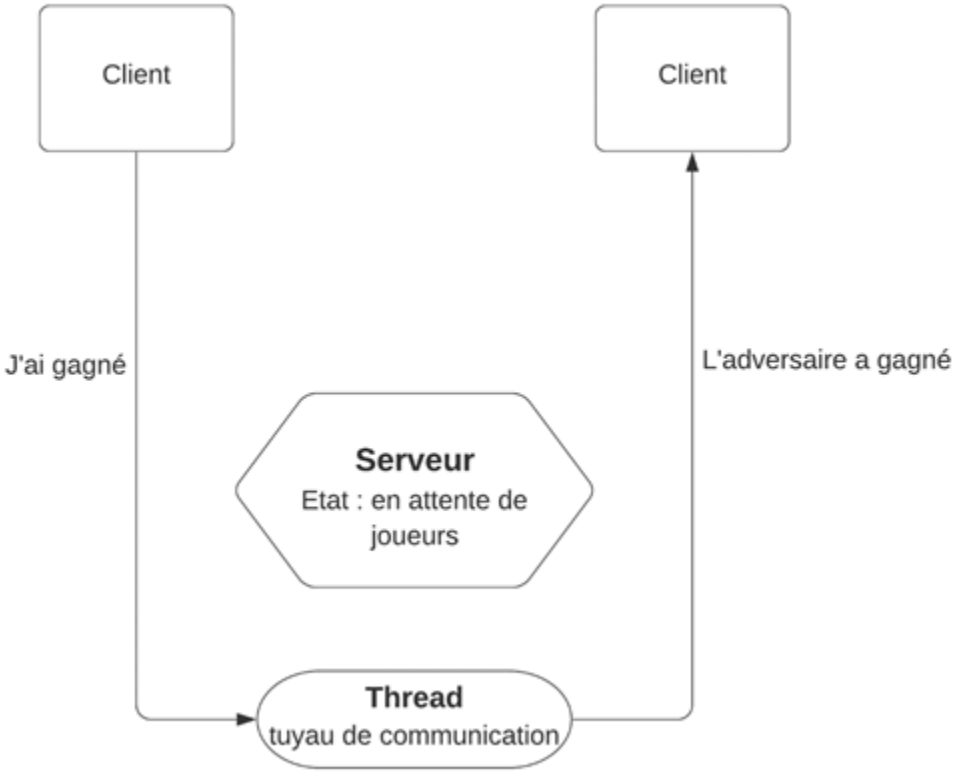

Figure 21 : *Fin d'une partie*

4.7 Problèmes rencontrés

Tout d'abord, le problème le plus important qui a été rencontré pendant la réalisation de ce projet est de créer un serveur en mode concurrent : l'attachement d'un thread à deux clients posait des problèmes de variables partagées.
Lorsque les données des deux clients sont stockées dans le serveur (pour pouvoir les donner au thread et leur répondre) et que d'autres clients arrivent, le thread confondait les anciens clients avec les nouveaux.
Cette difficulté a été résolue en créant de nouvelles variables pour stocker les données des nouveaux clients.
Une autre difficulté qui a été rencontrée est l'encodage et le décodage des messages. Suite à cela, une classe Information a été créée. Son but est de décoder et encoder les messages lors de la réception et de l'envoi des requêtes.

4.8 Améliorations possibles

Une amélioration possible aurait été de créer un compte lors du lancement de l'application pour créer un classement en ligne des meilleurs joueurs. Une autre idée d'amélioration serait de mettre tous les algorithmes de l'intelligence artificielle dans le serveur.

Ainsi si l'ordinateur sur lequel est lancé le jeu n'est pas puissant, l'intelligence artificielle prendrait moins de temps à jouer sur un ordinateur avec une plus grande capacité de calculs.

5. Intelligence artificielle

L'intelligence artificielle, souvent abrégée par le sigle IA, est un terme créé par *John McCarthy* en 1956. Elle est définie par l'un de ses créateurs, *Marvin Lee Minsky,* comme : *"Construction de programmes informatiques qui s'adonnent à des tâches qui sont, pour l'instant, accomplies de façon plus satisfaisantes par des êtres humains car elles demandent des processus mentaux de haut niveau tels que l'apprentissage perceptuel, l'organisation de la mémoire et le raisonnement critique."*. Pour le Larousse, il s'agit d'un *"ensemble de théories et de techniques mises en œuvre en vue de réaliser des machines capables de simuler l'intelligence humaine."*.
Le manque de consensus scientifique, mais aussi les vastes domaines d'application rendent la définition de l'intelligence artificielle complexe.
Depuis quelques années, on associe souvent l'intelligence artificielle aux capacités d'apprentissage (machine learning, deep learning). Néanmoins, le domaine de l'IA n'a pas toujours considéré l'apprentissage comme essentiel à l'intelligence, c'est donc pour cela que nous considérerons ici nos algorithmes comme des intelligences artificielles.

5.1 MiniMax

L'algorithme minimax (aussi appelé algorithme MinMax) est un algorithme basé sur le théorème du Minimax conçu en 1928 par le mathématicien John von Neumann. Il est utilisé dans la "Théorie des jeux combinatoires" c'est-à-dire un jeu à deux joueurs à somme nulle et information complète (Puissance 4, évidemment, mais aussi morpion, échecs, Go, etc) :

- **somme nulle :** la somme des gains et des pertes de tous les joueurs est égale à 0 (les gains d'un joueur sont exactement l'opposé des gains de l'autre joueur) ;

- **information complète :** lors de sa prise de décision (c'est-à-dire lorsqu'il choisit son prochain coup), chaque joueur connaît précisément :

  – ses possibilités d'action ;
  – les possibilités d'action de son opposant ;
  – les gains résultants de ces actions.

Le but de MinMax est de trouver le coup optimal (c'est-à-dire maximiser sa chance de gagner) pour un joueur en considérant que le joueur adverse joue toujours de façon optimale. A chaque coup, les joueurs tentent de maximiser leur gain minimum et donc de minimiser le gain maximal de leur adversaire (somme nulle).

Dans cet algorithme, les joueurs sont appelés *maximiseur* (*Max*) et *minimiseur* (*Min*). On appelle ***état*** du jeu une configuration du jeu, ***état initial*** un état où aucun joueur n'a joué et ***état final*** un état mettant fin au jeu (victoire, défaite, égalité).
Pour déterminer les gains, on définit une fonction d'*évaluation* (aussi appelée fonction *heuristique*) qui attribue une note à un état du jeu. Cette note peut être interpréter comme la valeur que l'on peut espérer gagner en jouant tel coup.

5.1.1 Fonctions d'évaluation

Le développement d'une fonction d'évaluation dans un jeu où la complexité (nombre d'état possibles) est élevée est nécessaire, car procéder à des calculs jusqu'à atteindre un état final, où il est alors très facile de connaître les gains d'un joueur, est trop coûteux. Il s'agit souvent de la partie la plus complexe à définir car elle est empirique (pour une bonne fonction d'évaluation, il faut une bonne connaissance du jeu).

a) Evaluation calculant le nombre de combinaisons

La première heuristique implémentée consiste à créer un tableau de mêmes dimensions que la grille du jeu, où chaque case contient le nombre d'alignements gagnants possibles si l'on place un pion à cet endroit.
En partie, l'IA choisira de jouer dans la case comportant la meilleure note parmi celles où il est possible de jouer.
Cette heuristique ne défend pas en partie, son seul but est d'augmenter le nombre d'alignements qu'elle peut faire.

Cette évaluation est représentée, dans le projet, par une classe, celle-ci possède plusieurs fonctions. Chaque fonction, vérifie, pour une direction et pour chaque case de la grille, le nombre de fois que celle-ci peut permettre d'effectuer un alignement de 4 jetons.
Comme dit précédemment, la classe créer un tableau de la taille de la grille avec les résultats obtenus.

Prenons une seule profondeur de la grille de ce puissance 4, notre fonction d'évaluation lui affectera ces notes (figure 22).

5	6	7	9	6	5
6	8	10	12	9	7
7	10	13	15	12	9
7	10	13	15	12	9
6	8	10	12	9	7

Figure 22 : *grille des nombres de combinaisons par case*

Le joueur ici correspond à la couleur rouge et l'IA au jaune. Si le joueur choisit de jouer à la ligne 1, colonne 3, l'IA va observer toutes les cases où elle peut actuellement jouer et elle choisira la position ayant la meilleure note c'est à dire la position à la ligne 2, colonne 3 avec la note de 13, comme illustré ci-dessous (figure 23.1).

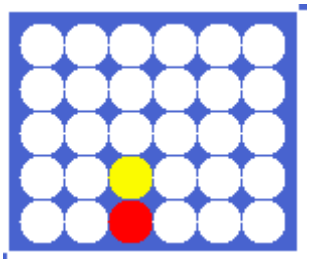

Figure 23.1 : *représentation d'une grille après deux coups*

Si l'utilisateur souhaite jouer à la ligne 3 toujours sur la colonne 3. L'IA ne va pas continuer à joueur en hauteur puisque la note de la case du dessus est de 10 mais elle va détecter qu'à la position ligne 1, colonne 4, le score est de 12 et c'est donc dans cette case qu'elle jouera (figure 23.2)

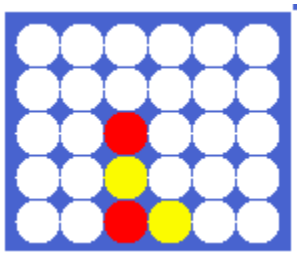

Figure 23.2 : *représentation d'une grille après quatre coups*

Cette présentation illustre bien que cette évaluation ne permet pas à l'IA d'être offensive ou défensive. Dans un cas où le joueur humain réussit à aligner 3 jetons, si la case lui permettant de constituer un alignement de 4 jetons et donc de gagner ne possède pas pour l'IA le meilleur score alors elle jouera à une autre position ce qui l'amènera à perdre. Ceci peut aussi se produire dans le cas d'une possibilité de victoire pour l'IA, qui pourrait choisir une case quelconque au détriment de celle qui lui garantirait la victoire.

L'équipe projet a réussi à développer cette heuristique, mais dans ce puissance 4 celle-ci n'a pas grand intérêt puisqu'elle est prévisible et qu'elle n'attaque pas et ne défend pas volontairement. Elle n'a donc pas été prises en compte dans le menu et lors de l'élaboration des statistiques. Cependant, une autre évaluation plus efficace a été programmée.

b) Evaluation comptant le nombre d'alignements

La seconde heuristique que l'on a implémentée consiste à établir un certain score total à la partie, ce score est déterminé par les différentes combinaisons (alignements) que possède le joueur. En effet, s'il a réussi à aligner deux jetons alors il obtiendra un certain nombre de points, s'il en a trois alignés, ce nombre de points augmente, dans le cas d'un alignement de quatre jetons le nombre de point attribué au score est encore plus élevé. Dans cette évaluation, un alignement de 2 jetons correspond à 1 point, de trois 10 et quatre à 1000.

Prenons une partie quelconque et arrêtons-nous pour comprendre quel score cette situation possède-t-elle (figure 24).

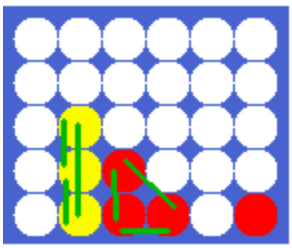

Figure 24 : *différents alignements au cours d'une partie*

En vert, sont indiquées les combinaisons rapportant des points au score global. Il y a 5 combinaisons de deux jetons valant 1 point chacune et une combinaison de trois jetons valant 10 points.
En additionnant tous ces points, on arrive à un score de 15.

5.1.2 Recherche exhaustive

Dans un premier temps il faut un établir un arbre de jeu dans lequel MinMax va effectuer sa recherche. Un arbre de jeu est composé d'une racine (nœud initial), d'un ou plusieurs nœuds internes et d'une ou plusieurs feuilles (nœuds terminaux) :

- **nœud :** un nœud est une structure qui peut contenir une valeur ou représenter une structure de données distinctes ;
- **racine :** le nœud qui se trouve au sommet de l'arbre, le principal ancêtre ;
- **feuille :** un nœud sans enfants ;
- **nœud interne :** un nœud avec au moins un enfant, la racine peut être un nœud interne ;

- **branche :** une connexion entre un nœud et un autre ;

- **profondeur :** la distance (le nombre de branche) entre un nœud et la racine, la profondeur maximale de l'arbre correspond à la plus grande distance entre une feuille et la racine ;

- **facteur de branchement :** nombre d'enfants à chaque nœud interne. Si cette valeur n'est pas uniforme, un facteur de branchement moyen peut être calculé.

Dans le cas de notre jeu, qui est une grille de taille 5x6x5, le facteur de branchement est de 30 (nombre de colonnes*nombre de profondeurs = 6*5) c'est-à-dire que à chaque tour, un joueur peut jouer de 30 manières différentes. En réalité, ce nombre n'est valable que durant les premiers tours d'une partie car une colonne peut vite devenir pleine, diminuant ainsi le facteur de branchement de 1. Nous ne pouvons pas vous fournir un facteur de branchement moyen car cela nécessiterait, pour qu'il soit fiable, un grand nombre de données de parties dites "réalistes", chose que nous n'avons pas. Néanmoins, nous pouvons supposer qu'au vu de la complexité de la grille et le grand nombre de possibilités de victoires, une partie se finit bien avant qu'une grille soit pleine et donc que le facteur de branchement moyen ne doit pas être très éloigné de 30.

Un arbre de jeu peut être représenté tel que sur la figure 25.1 (un faible facteur de branchement, contrairement à celui de notre jeu, a été choisi afin de ne pas complexifier inutilement l'explication). Chaque nœud représente un état du jeu et les branches représentent les coups que peuvent réaliser Max ou Min à partir de cette position (dans le cas du Puissance 4, le nombre de coups possibles est restreint si une ou plusieurs colonnes sont pleines). Les feuilles correspondent, dans une recherche exhaustive, à un état final. Des valeurs V ou D pour victoire ou défaite, du point de vue de Max, sont associées aux feuilles. Ces valeurs rendent compte de l'issue d'une séquence de coups depuis la racine.

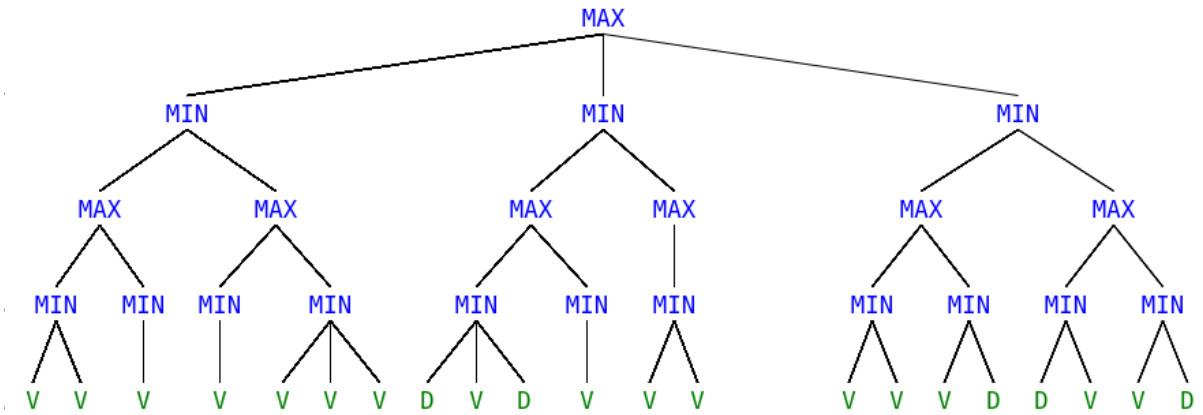

Figure 25.1 : *Arbre de recherche MinMax développé jusqu'à un état final. Un nœud MAX correspond au cas où le joueur essaie de maximiser son gain tandis qu'un nœud MIN correspond au cas contraire.*

Cet arbre montre que le joueur Max remporte la partie car il peut jouer 2 coups où il est sûr de gagner peu importe les répliques (coups) du joueur Min. Pour en arriver à une telle conclusion nous allons analyser le fonctionnement de l'algorithme MinMax sur cet arbre de jeu.
Pour étudier la stratégie de recherche MinMax il faut d'abord comprend le rôle de Max et de Min dans cet arbre. Au départ (la racine), le joueur Max a la possibilité de jouer un coup parmi 3 possibles. Il va donc choisir un coup (un nœud) qui va lui permettre d'augmenter ses chances de gagner, on appelle cela **maximiser** ses gains. A l'inverse, pour chaque réplique du joueur Min, ce dernier va chercher à minimiser les chances de victoires de Max c'est-à-dire **minimiser** les gains de Max (le jeu étant à somme nulle, le joueur Min, par la même occasion, maximise ses gains). Max essayera donc de maximiser le minimum des gains qu'il peut avoir (Min essayera toujours de réduire les gains de Max).

Passons à l'analyse :

- Premier coup : le premier coup (correspondant à la première branche de la racine) mène, à la suite d'une série de coups, à des états finaux qui correspondent tous à une victoire du point de vue du joueur Max : ce coup (nœud) mène donc à une victoire comme illustré dans la figure 26.2.

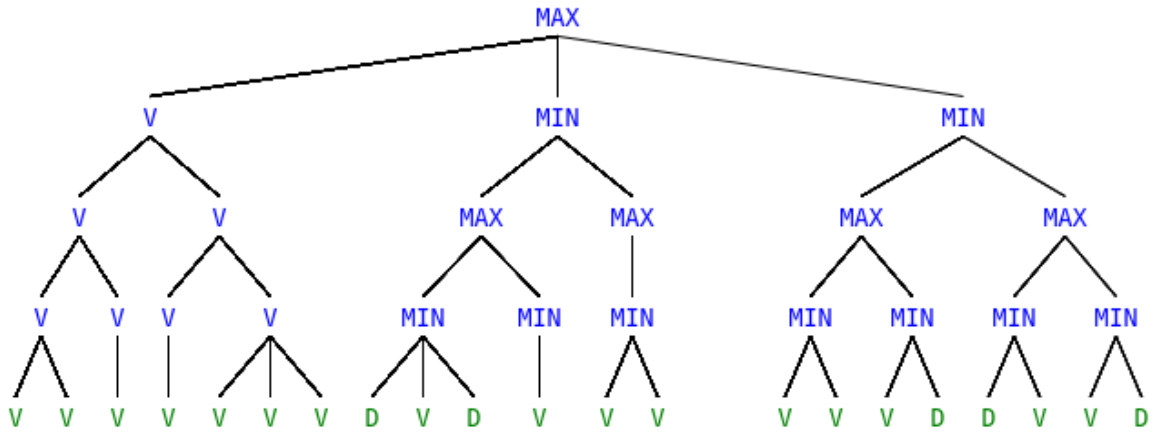

Figure 26.2 : *Application de l'algorithme MinMax pour le premier coup.*

- Second coup : pour la seconde branche, la situation sur laquelle va aboutir ce coup est plus difficile à éclaircir car, deux des feuilles, le premier et le troisième fils de C, correspondent à des défaites pour le joueur Max. Min a la possibilité sur le nœud C de choisir l'un de ces deux coups, son objectif étant de minimiser les chances de gagner pour Max, il ne se privera pas et choisira un de ces deux coups. Max considérera donc que s'il arrive au nœud C (en jouant le premier coup lorsqu'il se trouve au nœud A), la partie sera perdue. On considère alors le nœud C comme une position de défaite tandis que le nœud D sera considéré comme une position de victoire car son seul fils mène à un état final correspondant à une victoire (figure 26.3). Il en va de même pour le nœud E dont les fils mènent à des victoires (figure 26.3).

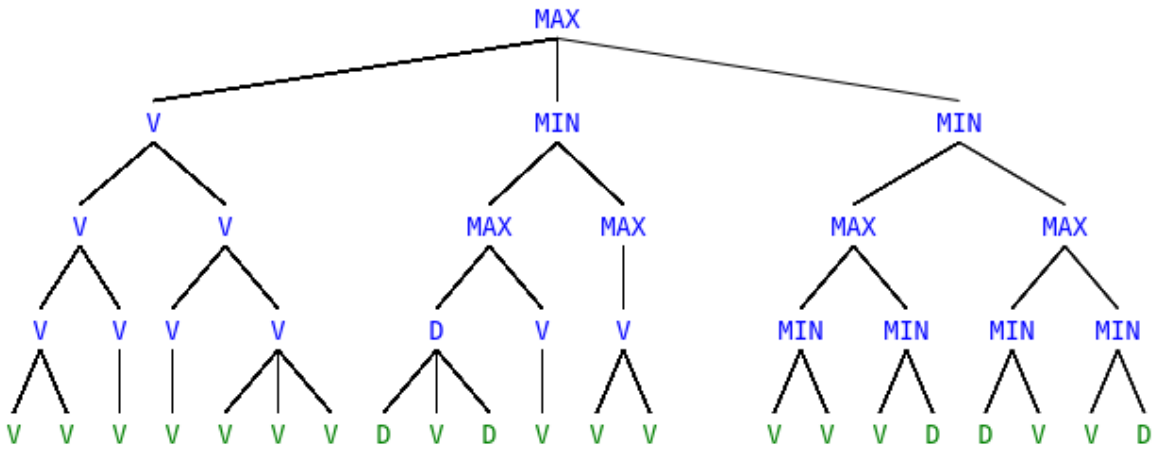

Figure 26.3 : *Application de l'algorithme MinMax pour les nœuds C, D et E.*

Ainsi, dans le nœud A, Max peut choisir entre deux coups (le nœud C et le nœud D), l'un le menant à une défaite (nœud C) et l'autre à une victoire (nœud D) ; puisqu'il essaye de maximiser ses chances de gagner, il choisira donc le second coup (nœud D) (figure 26.4). Dans le cas du nœud B, Max ne peut choisir qu'un seul coup (nœud E), dans le cas présent, il mène à une victoire (figure 26.4). Le second coup que peut réaliser Max est donc nécessairement gagnant (s'il ne commet pas d'erreur dans le nœud A) car Min n'a le choix dans le nœud Z qu'entre deux coups, les deux étant gagnants il ne pourra pas minimiser les chances de victoires de Max (figure 26.4).

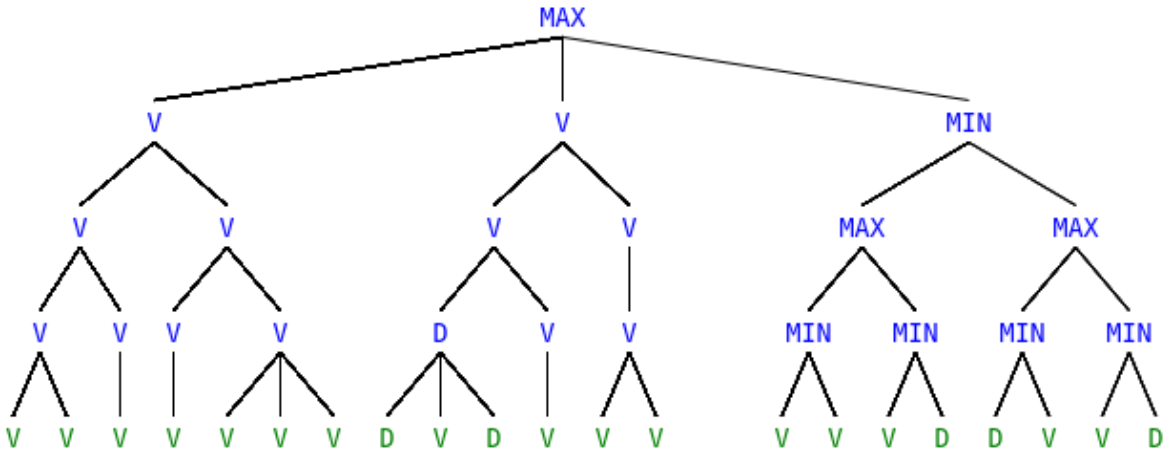

Figure 26.4 : *Application de l'algorithme MinMax pour les nœuds A, B et Z.*

- Troisième coup : si l'on applique le même raisonnement que précédemment pour le troisième coup on se rend compte que le nœud F mène à une victoire (son premier fils mène nécessairement à une victoire et comme le nœud F correspond au joueur Max, c'est ce fils qui sera choisi) tandis que le nœud G mène à une défaite (ses deux fils mènent à des défaites). De ce fait, le troisième coup mènera à une défaite : Min n'hésitera pas à choisir le nœud G pour minimiser les chances de gagner de Max (figure 26.5).

Ainsi, la position dans laquelle se trouve le joueur Max est très favorable car il dispose de deux possibilités de victoires en jouant soit le premier, soit le second coup (figure 26.5).

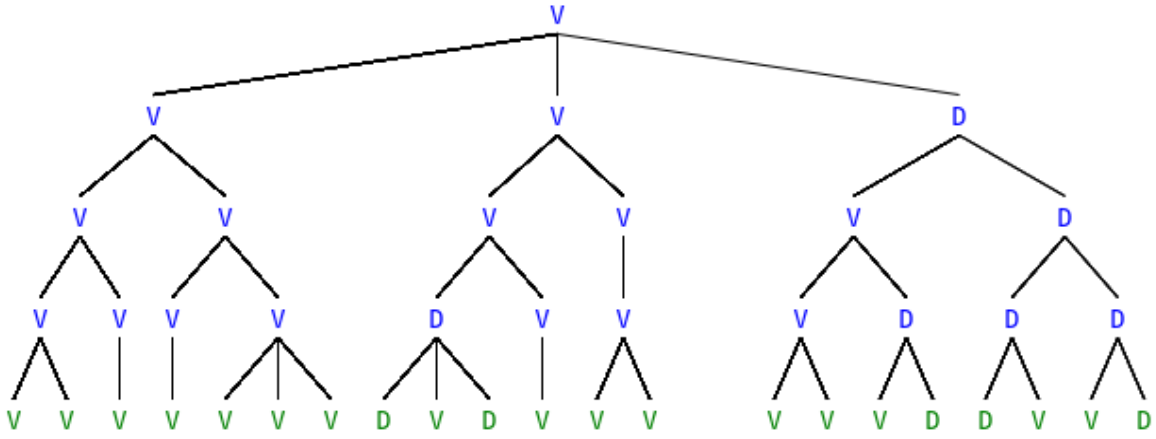

Figure 26.5 : *Application de l'algorithme MinMax pour tout l'arbre de jeu.*

5.1.3 Recherche non-exhaustive

La situation de recherche décrite dans la dernière partie ne peut exister que si la complexité du jeu est faible (le morpion par exemple), ce qui permet de calculer tous les coups possibles depuis un état initial en un temps raisonnable, ou lorsqu'elle est mise en œuvre à partir d'un état qui s'approche d'un état final. Dans notre cas, cela n'est pas possible, il s'agit donc d'une recherche non-exhaustive ou nous allons devoir utiliser une profondeur d'arbre prédéfinie (aussi appelée profondeur de recherche) et une fonction d'évaluation comme décrit dans une partie antérieure.
L'arbre de jeu n'est alors développé que jusqu'à une certaine profondeur (pour l'arbre des figures 1.x, la profondeur est de 4), plus la profondeur de recherche prédéfinie sera grande, plus le temps de calcul d'un coup sera long : pour un facteur de branchement moyen de 30, le nombre de nœuds généré sera environ de 30^p, p étant la profondeur de recherche. En contrepartie, l'algorithme choisira un meilleur coup par rapport à une profondeur de recherche inférieure. Les feuilles de l'arbre ne représentent plus des états finaux, ainsi, il n'est plus possible de déterminer si une feuille est gagnante ou perdante comme dans le cas précédent, il faut donc faire appel à une fonction d'évaluation qui va attribuer une note à chaque feuille de l'arbre de jeu (figure 27.1). Néanmoins, l'algorithme MinMax procédera de la même façon que dans une recherche exhaustive : Min cherchera toujours à minimiser les chances de victoire de Max en choisissant, cette fois, le nœud ayant la note la plus faible (et non un nœud qualifié de défaite) et inversement pour Max qui cherchera à augmenter ses chances de victoire en prenant le nœud ayant la note la plus élevée (figure 27.2).

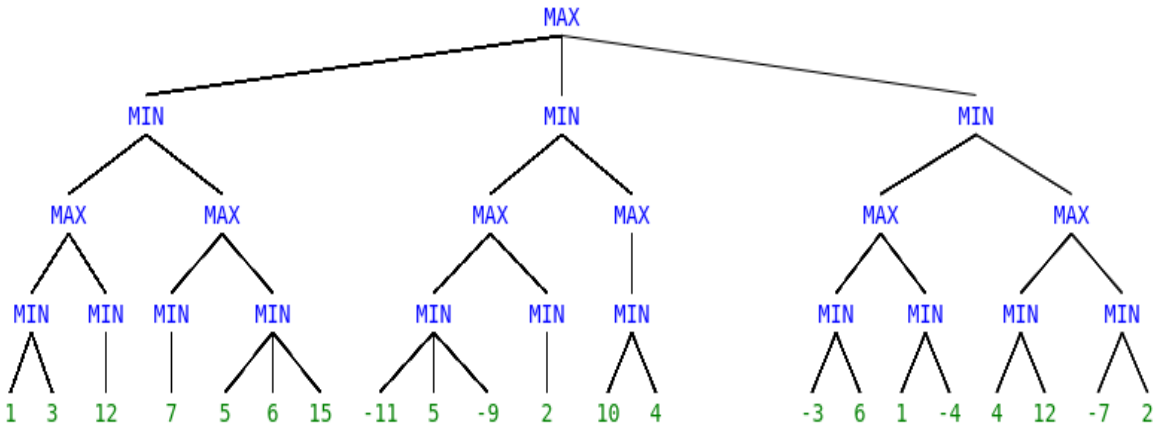

Figure 27.1 : *Arbre de jeu contenant, aux feuilles, les notes de la fonction d'évaluation.*

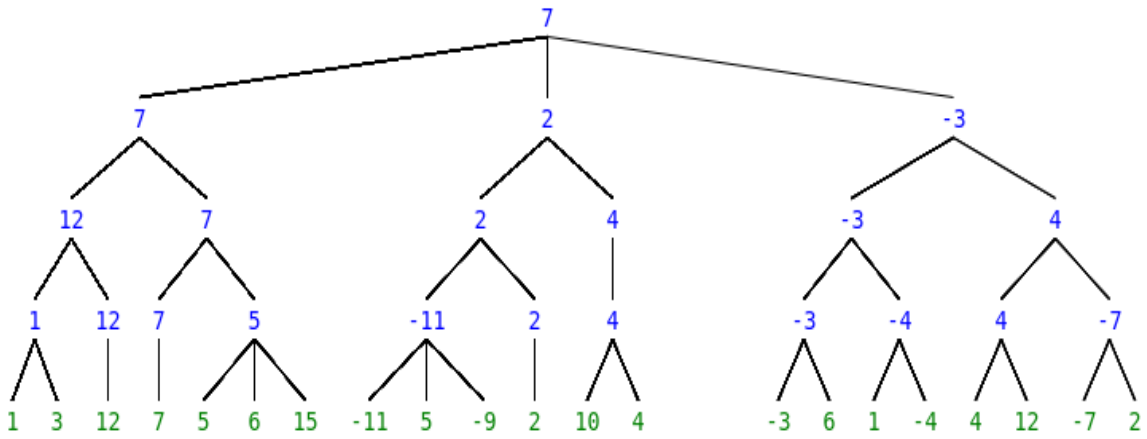

Figure 27.2 : *Application de l'algorithme MinMax à un arbre de jeu dont les feuilles représentent des notes.*

5.1.4 Pseudo-code

Après avoir analysé le fonctionnement de l'algorithme MinMax en détails, un pseudo-code est décrit dans la figure 28. On remarque vite la distinction entre un nœud Max et un nœud Min : l'un cherche la plus grande valeur entre celle actuelle et celles des profondeurs inférieures tandis que l'autre cherche la valeur la plus faible. Lorsqu'on arrive aux feuilles (profondeur 0), on utilise la fonction d'évaluation pour associer une valeur aux feuilles. Si l'on est attentif, on peut remarquer que la fonction *max* est l'opposé de la fonction *min* (jeu à somme nulle) c'est-à-dire que *max(a,b) = -min(-a,-b)*, le code de MinMax peut donc être simplifié devant ainsi l'algorithme Negamax. Dans ce projet, le code implémenté est celui de MinMax. On remarque aussi l'aspect récursif de la fonction afin de faire remonter les notes jusqu'à la racine.

```
fonction minimax(nœud, profondeur, joueurMAX) est
si profondeur = 0 ou nœud est une feuille alors
      renvoyer la note de la fonction d'évaluation pour nœud
si joueurMAX alors
      note := −∞
      pour chaque fils de nœud faire
            note := max(note, minimax(fils, profondeur − 1, FAUX))
      renvoyer note
autre (* joueurMIN *)
      note := +∞
      pour chaque fils de nœud faire
            note := min(note, minimax(fils, profondeur − 1, VRAI))
      renvoyer note
```

Figure 28 : *Pseudo-code de l'algorithme MinMax.*

5.2 Elagage AlphaBeta

Le problème majeur de MinMax est son temps d'exécution du fait que les informations (les notes) ne circulent que dans un seul sens : des feuilles vers la racine de l'arbre de jeu. Pour qu'il puisse faire transiter les notes des feuilles vers la racine il est nécessaire d'avoir explorer l'arbre de jeu jusqu'aux feuilles et d'avoir attribué une note à chaque feuille.

5.2.1 Principe

Le but de l'élagage AlphaBeta (qui est une amélioration de MinMax), aussi appelé algorithme Alpha-Beta, est justement d'éviter les explorations vers des feuilles qui s'avèrent être inutiles. Pour y parvenir, l'algorithme va utiliser un processus d'exploration dit *"depth first"* (*profondeur d'abord*) où, avant d'explorer un frère d'un nœud, les fils sont développés. Pour épauler ce processus d'exploration, l'information sera utilisée en la remontant des feuilles vers les nœuds internes mais aussi en la redescendant vers d'autres nœuds.

5.2.2 Explications

Concrètement, le principe d'AlphaBeta est de tenir à jour tout au long de l'exploration deux valeurs : α et β. La première valeur correspond à la note minimum que le joueur Max est assuré d'obtenir depuis un nœud (coup) tandis que la seconde valeur correspond à la note maximale que le joueur Min est assuré d'obtenir depuis un nœud. Si la note maximale que le joueur Min est assuré d'avoir, c'est-à-dire β, est inférieure à la note minimum que le joueur Max est assuré

d'obtenir, c'est-à-dire α, alors le joueur Max ne doit pas considérer l'exploration du nœud sur lequel il se trouve car il ne sera jamais atteint dans la partie. On appelle cela une coupure.
On peut illustrer cela avec un exemple de la vie réelle. Supposons que vous êtes en train de jouer à une partie de puissance 4 et c'est votre tour. Vous avez trouvé un coup qui augmentera vos chances de gagner. Vous continuez à chercher s'il n'existe pas un coup encore mieux que celui que vous avez trouvé. Vous en trouvé un mieux mais vous vous rendez compte que ce dernier offre une possibilité à l'adversaire d'aligner 4 jetons, vous allez donc arrêter de réfléchir aux autres possibilités de ce coup car vous savez que le joueur adverse peut gagner. Dans notre cas, l'algorithme arrêtera d'explorer.
Pour bien comprendre son fonctionnement, une analyse d'un arbre de jeu est nécessaire (figure 29).

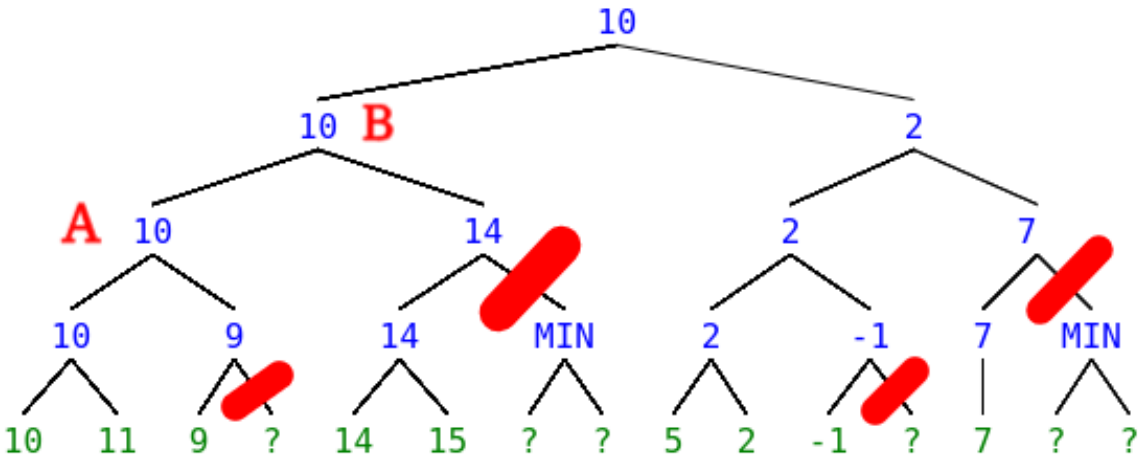

Figure 29 : *Arbre de jeu contenant des branches non explorées, les traits rouges représentent des coupures alpha ou beta.*

La première coupure intervient au niveau de la branche reliant la quatrième feuille car le développement du premier fils du nœud A permet de remonter l'information que $\alpha \geq 10$ (le nœud A est un nœud Max, la valeur qu'il choisira sera au moins 10). Ainsi, cette information est propagée dans le second fils (qui est un nœud Min) du nœud A où l'on trouve que la valeur qu'il choisira sera au plus de 9 c'est-à-dire que $\beta \leq 9$. Comme $\beta < \alpha$ on procède à une coupure de type alpha : Min ne devra pas considérer l'exploration du nœud.
Pour la seconde coupure, le second fils (qui est un nœud Max) du nœud B nous informe qu'il lui est possible de jouer un coup lui rapportant au moins 14 (α) tandis que le nœud B (Min) nous informe qu'il lui est possible de jouer un coup donc la valeur maximale est 10 (β). Comme $\alpha > \beta$ on procède à une coupure de type beta : Max ne devra pas considérer l'exploration du nœud.
A noter que le coup renvoyé par MinMax et AlphaBeta est le même si la profondeur de recherche est la même. La seule différence se situe au niveau du nombre nœuds explorés qui est inférieur pour AlphaBeta. On estime le nombre de nœuds explorés par AlphaBeta de l'ordre de la racine carrée du nombre de nœuds explorés par MinMax, cela se traduira aussi au niveau du temps d'exécution : si MinMax met, par exemple, 25 secondes à calculer le prochain coup, AlphaBeta ne mettra que 5 secondes.

5.2.3 Pseudo-code

Comme pour l'algorithme MinMax, voici un pseudo-code pour l'algorithme AlphaBeta (figure 30). On remarque la présence des conditions de coupures par rapport à l'algorithme MinMax.

```
fonction alphabeta(nœud, profondeur, α, β, joueurMAX) est
      si profondeur = 0 ou nœud est une feuille alors
            renvoyer la note de la fonction d'évaluation pour nœud
      si joueurMAX alors
            note := −∞
            pour chaque fils de nœud faire
                        note := max(note, alphabeta(fils, profondeur −
1, α, β, FAUX))
                        α := max(α, note)
                        si α ≥ β alors
                        interruption (* coupure β *)
            renvoyer note
      autre
            note := +∞
            pour chaque fils de nœud faire
                        note := min(note, alphabeta(fils, profondeur −
1, α, β, VRAI))
                        β := min(β, note)
                        si α ≥ β alors
                  interruption (* coupure α *)
            renvoyer note
```

Figure 30 : *Pseudo-code de l'algorithme AlphaBeta.*

5.3 Monte-Carlo

Monte-Carlo désigne une famille de méthodes algorithmiques visant à calculer une valeur numérique approchée en utilisant des procédés aléatoires. Le nom de ces méthodes, qui fait allusion aux jeux de hasard pratiqués au casino de Monte-Carlo, a été inventé en 1947 par *Nicholas Metropolis* et publié pour la première en 1949 dans un article coécrit avec *Stanislaw Ulam*.
Dans ce projet, les procédés aléatoires représentent la simulation (avec des coups au hasard) d'une partie jusqu'à un état final. Un grand nombre de simulation permet ainsi de comptabiliser la proportion de parties gagnantes/perdantes remplaçant ainsi la note qu'aurait attribuée une fonction d'évaluation à un état du jeu par une estimation statistique.
Cette famille d'algorithmes permet donc de se passer d'une fonction d'évaluation qui, pour être bonne, nécessite un gros travail de réflexion comme pour les fonctions d'évaluation de ce projet qui, même après un travail conséquent d'élaboration, sont loin d'être parfaites.
Deux méthodes ont été étudiées : Monte-Carlo biaisé et Monte-Carlo Tree Search (MCTS). Néanmoins, seule la première a eu le temps d'être implémentée dans le projet mais nous aborderons les deux dans cette partie.

5.3.1 Monte-Carlo biaisé

L'algorithme Monte-Carlo biaisé reprend simplement l'idée de la famille d'algorithmes Monte-Carlo : réaliser des simulations aléatoires pour estimer la valeur d'un état de jeu.
Il réalise une série de simulations pour chaque coup possible depuis un état de jeu et attribue une note à chaque coup. Elle représente la proportion de victoires/défaites des simulations réalisées depuis cet état. Plus ce nombre est élevée, plus la probabilité de gagner (en jouant le coup associé à cette note) est grande. L'algorithme choisi donc le coup ayant la plus grande valeur (la plus grande probabilité de victoire) parmi les coups possibles (figure 31).

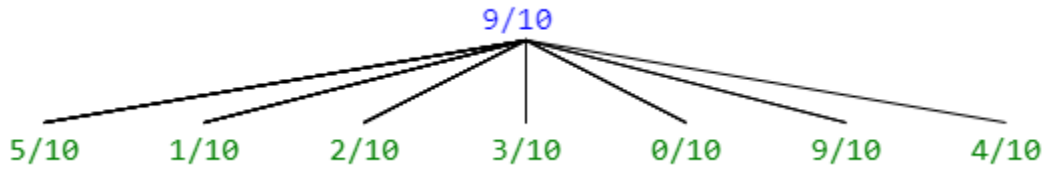

Figure 31 : *Application de l'algorithme Monte-Carlo biaisé à un arbre de jeu.*

Cet algorithme, contrairement aux autres vus précédemment, ne possède pas de conditions d'interruption, il faut donc lui en imposer une qui sera le temps (cela peut aussi être le nombre de simulations). Ici, le choix a été de répartir équitablement le temps de simulation entre chaque coup possible. Par exemple, si on définit un temps de simulation de 2 secondes et qu'il y a 4 coups possibles depuis une certaine position, chaque coup effectuera *n* simulations d'une durée

totale d'une demi-seconde. En ce sens, l'efficacité de l'algorithme pour un temps donné dépend principalement de la puissance de la machine sur laquelle il s'exécute.
Un pseudo-code pour l'algorithme Monte-Carlo biaisé est proposé dans la figure 32.

```
fonction montecarlo(noeud, temps, joueur) est
      meilleurRatio := -1
      pour chaque fils de noeud faire
            victoire := 0
            défaite := 0
            tant que temps n'est pas écoulé faire
                  résultat := simulerPartie()
                  si résultat = joueur alors
                        victoire++
                  autre
                        défaite++
            ratio := victoire/(victoire+défaite)
            si ratio ≥ meilleurRatio alors
                  meilleurRatio := ratio
      renvoyer meilleurRatio
```

Figure 32 : *Pseudo-code de l'algorithme Monte-Carlo biaisé avec pour condition d'interruption un temps donné.*

5.3.2 Monte-Carlo Tree Search

L'algorithme de recherche arborescente Monte-Carlo (MCTS), contrairement à l'algorithme Monte-Carlo biaisé, va explorer l'arbre de jeu. Chaque feuille de cet arbre sera soit un état final, soit un état où aucune simulation n'a encore été lancée. Pour chaque nœud on stocke deux nombres : le nombre de simulations gagnantes et le nombre total de simulations.
L'algorithme se décompose en quatre étapes qui seront répétées tant que l'algorithme sera exécuté.

a) Sélection

Depuis la racine, on sélectionne successivement des nœuds fils jusqu'à atteindre une feuille. Cette sélection est guidée par un compromis entre exploitation c'est-à-dire aller vers un nœud fils qui a été prouvé comme prometteur (ratio simulations gagnantes/nombre total de simulations grand) et exploration c'est-à-dire aller vers un autre nœud fils, qui a l'air moins prometteur mais pourrait le devenir.
La formule la plus utilisée pour cette sélection est la formule UCT. Elle recommande de choisir le nœud pour lequel l'expression $\frac{w}{n} + c * \sqrt{\frac{\ln N}{n}}$ a la valeur maximale. Dans cette formule :

- w est le nombre de parties gagnés pour le nœud considéré ;
- n est le nombre de parties simulées pour le nœud considéré ;
- N est le nombre de parties simulées pour le père du nœud considéré ;
- c est le paramètre d'exploration (généralement il est égal à $\sqrt{2}$) ;

A noter que plus le paramètre d'exploration est grand, plus la sélection se tournera vers le côté exploration. Une valeur de $\sqrt{2}$ pour *c* représente un bon compromis entre exploitation et exploration.
Dans la figure 33.1, la feuille ayant la valeur maximale en appliquant la formule UCT est la A.

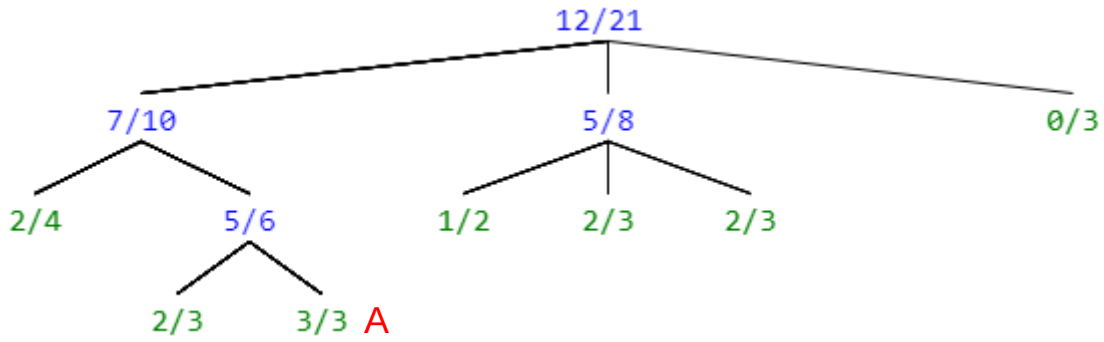

Figure 33.1 : *Application de l'algorithme MCTS à un arbre de jeu. La feuille choisie par la formule UCT est la A.*

b) Expansion

Si la feuille choisie n'est pas un état final, alors on crée un nœud fils qui aura comme nombre de simulations gagnantes zéro et comme nombre total de simulations zéro (figure 33.2).

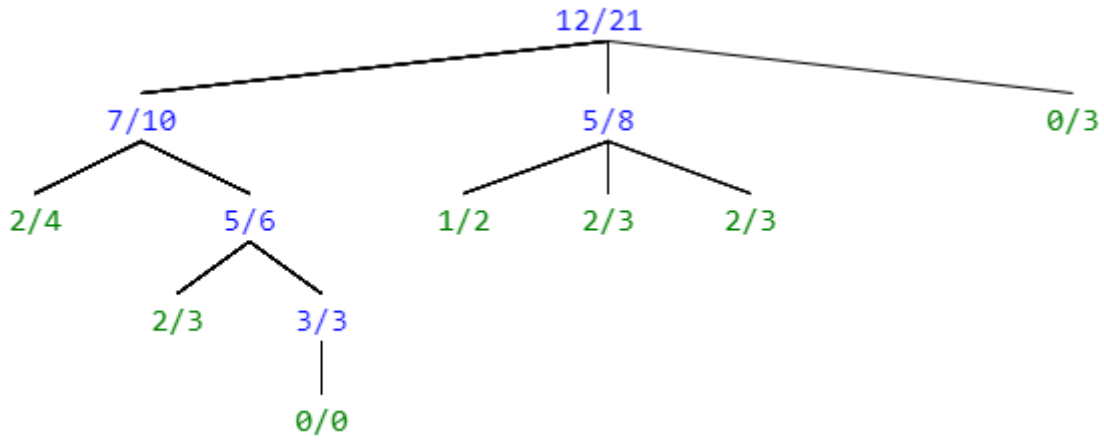

Figure 33.2 : *Phase d'expansion, création d'un nœud.*

c) Simulation

On simule ensuite une partie au hasard avec pour configuration l'état de jeu du nœud fils (figure 33.3).

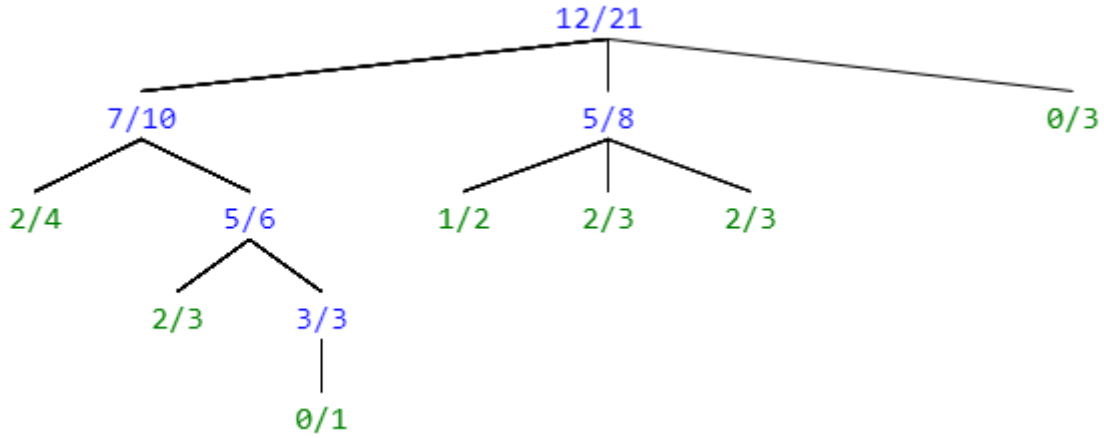

Figure 33.3 : *Simulation d'une partie dans le nœud précédemment créé. Dans cet exemple, la simulation mène à une défaite.*

d) Rétro propagation

On propage ensuite l'information du résultat de la partie jusqu'à la racine c'est-à-dire que le nombre total de simulations de chaque nœud entre le nœud ayant réalisé la simulation et la racine sera incrémenté d'un et le nombre de simulations gagnantes sera incrémenté d'un si l'état final de la partie est donné gagnant (figure 33.4).

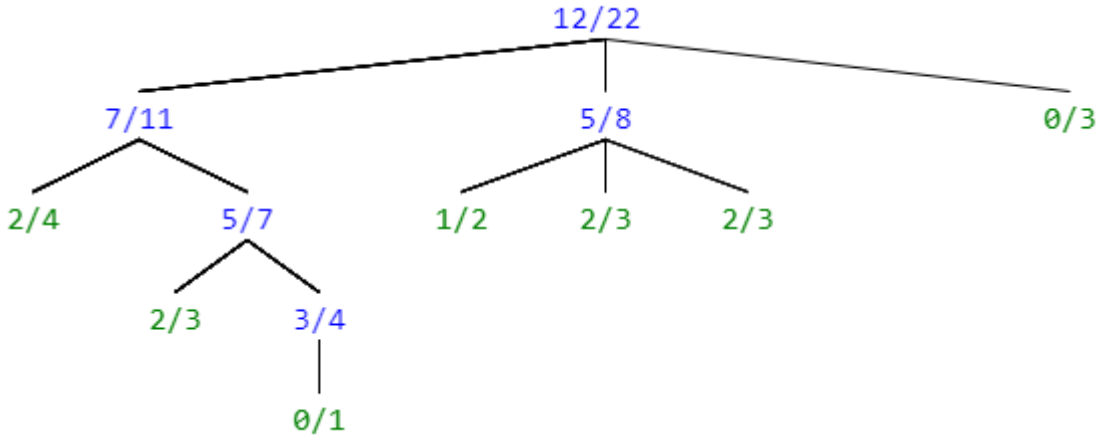

Figure 33.4 : *Simulation d'une partie dans le nœud précédemment créé. Dans cet exemple, la simulation mène à une défaite.*

5.4 Résultats expérimentaux

Pour se faire une idée plus précise des performances des algorithmes, plusieurs tableaux contenant diverses statistiques ont été réalisés.

5.4.1 Protocole

Ces statistiques ont été faites en simulant un nombre défini de partie entre chaque algorithme et en utilisant des configurations différentes pour chaque. Le nombre de partie simulé pour chaque confrontation a été fixé à 1000. Cela nous a semblé être un bon compromis entre temps de calcul et fiabilité des résultats (échantillons suffisamment représentatif) car après avoir simulé 10 000 parties pour une confrontation, la différence du résultat par rapport à 1000 parties était de l'ordre de moins de 5%.
Les simulations ont toutes été réalisées sur la même machine afin de ne pas fausser les statistiques concernant l'algorithme Monte-Carlo (l'efficacité de l'algorithme dépend de la puissance de la machine lorsque la condition d'arrêt est un temps fixé).
Cette machine est équipée d'un processeur AMD Ryzen 5 2600 : 6 cœurs physiques (cadencé à 3.4 Ghz) et 12 cœurs logiques (threads).
Pour profiter au maximum de ce nombre de cœurs, le programme de simulation est parallélisé : le nombre de simulations qui s'exécutent simultanément (ici, 12) correspond au nombre de cœurs logiques que possède la machine sur laquelle s'exécute la simulation.
A noter qu'il a fallu 4 jours de calcul pour établir ces statistiques (30 000 simulations ont été réalisées).

5.4.2 Résultats

Les statistiques concernent les algorithmes AlphaBeta et Monte-Carlo biaisé, chacun dans trois configurations différentes. Pour AlphaBeta le paramètre changeant est la profondeur de recherche tandis que pour Monte-Carlo biaisé le paramètre qui diffère est le temps (en millisecondes) de simulation pour calculer le prochain coup. Cela donne un tableau à double entrées représentant un pourcentage de victoires (figure 34), les paramètres sont précisés entre parenthèses, *Joueur 1* correspond au joueur (algorithme) jouant en premier, *Joueur 2* en deuxième.

Joueur 1 \ Joueur 2	αβ(4)	αβ(3)	αβ(2)	MC(2600)	MC(1300)	MC(600)
AlphaBeta(4)		82,30%	88,30%	57,40%	75,10%	82,90%
AlphaBeta(3)	28,40%		75,90%	32,30%	52,70%	63,80%
AlphaBeta(2)	32,50%	39%		40,30%	54,20%	77,60%
Monte-Carlo(2600)	60,20%	88,40%	84,20%		78,60%	89,80%
Monte-Carlo(1300)	44,70%	74,30%	70,20%	52%		78,80%
Monte-Carlo(600)	25,60%	53,90%	45,50%	27,20%	41,60%	

Figure 34 : *Tableau statistiques contenant le pourcentage de victoire entre le Joueur 1 (premier à jouer) et le Joueur 2 après avoir simulé 1000 parties.*

Le paramètre le plus élevé pour l'algorithme Monte-Carlo biaisé a été fixé à 2,6 secondes car le temps de calcul pour un coup de l'algorithme AlphaBeta à une profondeur de recherche de 4 variait de 1 à 5 secondes (voir plus en début de partie).
La première observation que l'on peut tirer est que, plus on augmente le temps de simulation pour l'algorithme Monte-Carlo, plus il est performant. Ce n'est pas surprenant car cela augmente le nombre de simulations qu'il réalise.
Quant à l'algorithme AlphaBeta, l'augmentation de la profondeur de recherche augmente bien le pourcentage de victoire sauf pour la profondeur de recherche 3. Là où on s'attend à ce que le pourcentage de victoire contre diverses adversaires se situe entre celui d'une profondeur 2 et celui d'une profondeur 4, on a à la place un pourcentage de victoire légèrement inférieur à celui d'une profondeur de recherche inférieur. Les simulations ayant été réalisées peu de temps avant la remise du rapport, nous n'avons pas eu le temps d'investiguer. Néanmoins, on peut supposer que ce problème se produit sur une profondeur de recherche impair où les feuilles dans l'arbre de jeu correspondent au joueur adverse (Min), la fonction d'évaluation est peut-être la cause de ce dysfonctionnement.
De ce tableau on peut tirer un second tableau mettant en avant les performances de chaque algorithme dans des configurations différentes avec leur pourcentage de victoire moyen en jouant en premier et en jouant en deuxième (figure 35).

	Joueur 1	Joueur 2	% de victoire moyen
AlphaBeta(2)	48,72%	27,18%	37,95%
AlphaBeta(3)	50,62%	32,42%	41,52%
AlphaBeta(4)	77,20%	61,72%	69,46%
Monte-Carlo(600)	38,76%	21,42%	30,09%
Monte-Carlo(1300)	64,00%	39,56%	51,78%
Monte-Carlo(2600)	80,24%	58,16%	69,20%
Moyenne	59,92%	40,08%	

Figure 35 : *Tableau statistiques contenant le pourcentage de victoire moyen d'un algorithme lorsqu'il joue en premier (Joueur 1) et lorsqu'il joue en deuxième (Joueur 2).*

La dernière ligne de ce tableau nous indique le pourcentage de victoire moyen lorsqu'un joueur commence en premier et lorsqu'il commence en deuxième. On remarque donc que le fait de jouer en premier ou en deuxième a une influence significative sur les chances de gagner la partie pour un algorithme. Il a, en moyenne, 20% de chances en plus de gagner la partie s'il joue en premier que s'il joue en deuxième.
Dernière observation que nous pouvons tirer est la performance de l'algorithme AlphaBeta(4) et celle de l'algorithme Monte-Carlo(2600), ils ont le même pourcentage de victoire. C'est une surprise car l'algorithme Monte-Carlo biaisé ne se base que sur des statistiques provenant de simulations de parties aléatoires. Cela confirme la difficulté que représente l'élaboration d'une fonction d'évaluation performante pour l'algorithme AlphaBeta.

5.5 Améliorations

Pour l'algorithme MinMax, nous avons implémenté une amélioration qui est l'algorithme AlphaBeta. Ce dernier pourrait être plus performant en améliorant la fonction d'évaluation mais aussi dans le tri des coups, c'est-à-dire l'ordonnancement des branches lors de la génération de l'arbre de jeu. Une autre amélioration pourrait être la mise en place d'un système d'exploration parallélisé (explorer plusieurs branches en même temps) qui pourrait s'appliquer à l'algorithme AlphaBeta mais aussi aux algorithmes de la famille Monte-Carlo.
Enfin, l'implémentation de l'algorithme MTD(f) (qui semble être un des algorithmes ayant comme base MinMax les plus efficace) pourrait être intéressante.

6. Conclusion

Voici un tableau général des connaissances acquises au cours de notre formation en DUT Informatique et qui nous ont été utiles dans la mise en œuvre de ce projet (figure 36).

Notions vues	Mise en pratique
Socket	Mise en place d'un système de partie en réseau
Thread	- Gestion des différentes connexions provenant des clients vers un serveur (mode concurrent) ; - Rafraîchissement en temps réel de l'interface graphique ; - Exécution de simulations en même temps pour élaborer les statistiques.
Programmation événementielle	Menu et jeu interactif

Figure 36 : *Tableau comparatif.*

6.1 Conclusion personnelle

6.1.1 Mark SOUS

Ce projet a été une très bonne expérience pour moi. C'était enrichissant, instructif et formateur : je sais par exemple, que la gestion du temps est un critère déterminant. Nombreux sont les groupes qui s'y prennent trop tard et notre projet n'a malheureusement pas échappé à la règle. Le dossier de projet aurait pu être commencé plus tôt afin de consacrer plus de temps aux finalisations. Le choix définitif du sujet a aussi été long. Ceci amène une première chose dont je suis maintenant convaincu, il faut s'activer dès les premières semaines sinon ce temps risque de nous manquer à la fin. Je voudrais aussi mentionner que la cohésion du groupe est un facteur déterminant à la réussite d'un projet. Je pense que nous nous sommes bien entendu, même s'il y avait quelques avis qui différait. Ce projet m'a, enfin, permis d'approfondir mes connaissances dans le domaine de l'intelligence artificielle et du réseau. Nous avons réalisé

plusieurs TP en réseaux et ce projet m'a permis de mettre en application toutes les compétences acquises. De même pour l'intelligence artificielle, ce terme était très vague avant la confection de ce projet, il est désormais bien plus clair même s'il y a certains aspects de celle-ci qui restent à découvrir.

6.1.2 Gaston LIONS

Depuis petit, je me suis toujours demandé comment fonctionnait l'IA contre qui je jouais dans un jeu vidéo. Comment fait-elle pour anticiper les actions ? Jouer toute seule ? La réalisation de ce projet a été l'opportunité d'avoir des réponses à certaines de mes questions ainsi que de découvrir des bases sur la création d'une IA.
Durant ce projet, j'ai principalement travaillé sur l'interface de l'application, j'aime particulièrement ajouter une touche de créativité dans un projet, malheureusement je n'ai pas eu suffisamment de temps pour rendre l'interface des menus à mon goût afin d'éviter d'en avoir des basiques (comme c'est le cas actuellement). La partie la plus intéressante a été d'imaginer et de mettre en place l'affichage d'une partie de notre application, comme l'animation lorsqu'un jeton est placé dans la grille, la 2nd vue créer, le fait d'afficher un jeton au-dessus de la colonne dans laquelle la souris de l'utilisateur se trouve.
L'idée de faire un puissance 4 ne me plaisait pas trop au départ, j'aurais aimé que l'on travaille sur un autre jeu, mais moins connu afin d'apporter de l'originalité dans notre choix.

6.1.3 Yanis DE SOUSA ALVES

Au début de ce projet, je dois avouer que j'étais réticent face au terme "Intelligence artificielle", n'étant pas l'un des meilleurs élèves en mathématiques, je me demandais comment je pourrais contribuer à la conception d'une de ces intelligences. Finalement, j'ai tout de même réussi à collaborer avec mes camarades afin de fournir différentes heuristiques. J'ai trouvé intéressant d'approfondir la partie réseau vu en cours en y amenant un objectif concret qui est celui de jouer au même jeu qu'une autre personne en temps réel. J'aurais aimé assister davantage Mark dans cette partie du projet car j'ai souvent rencontré cet aspect de "multijoueur" et j'étais curieux de savoir comment cela fonctionnait vraiment. Dans l'ensemble, je peux dire que ce projet s'est bien déroulé puisque nous avons atteint tous nos objectifs, bien-sûr, avec plus de temps ou une meilleure gestion du temps, nous aurions pu améliorer quelques aspects de notre réalisation.

6.1.4 Fabien LE BEC

Ce projet m'a particulièrement plu. J'ai été très intéressé dans la recherche et le développement des différentes intelligences artificielles, peut être bien plus que je l'aurais imaginé au départ. Cela a été une grande satisfaction de mettre au point des algorithmes et de voir que, petit à petit, ces derniers commençaient à avoir un niveau se reprochant de celui d'un humain dans un jeu tel que le Puissance 4. N'éprouvant pas une attirance particulière concernant le développement d'Interface Homme-machine, cela ne m'a pas dérangé de ne pas travailler sur le développement de cette dernière. Néanmoins, j'aurais aimé travailler un peu plus sur la partie réseau : ma seule intervention sur cette partie a été sur un problème nécessitant la mise en place de processus légers. Un autre regret concerne la gestion du temps, la partie de recherche concernant les différents algorithmes qui pouvaient être implémentés a prit bien trop de temps et nous avons commencé l'implémentation un peu tard ce qui nous a empêché d'implémenter tous les algorithmes que nous avions jugés pertinents. Dans l'ensemble, ce projet, qui est le plus gros sur lequel j'ai eu l'occasion de travailler, a été une expérience très enrichissante.

6.2 Conclusion générale

Au terme de ce projet, nous avons atteint nos différents objectifs qui étaient de créer un puissance 4 qui diffère légèrement de celui que tout le monde connait avec la présence de différentes intelligences artificielles contre laquelle le joueur pourrait jouer, et en lui donnant la possibilité de jouer contre une autre personne en réseau.
Nous avons compris qu'il est important d'évaluer les connaissances de chacun pour répartir au mieux le travail, et de faire un point sur nos tâches régulièrement. Notre organisation était plutôt bonne. Il y avait une bonne cohésion de groupe. Nous avons appris à gérer un projet et à faire face aux difficultés ensemble.