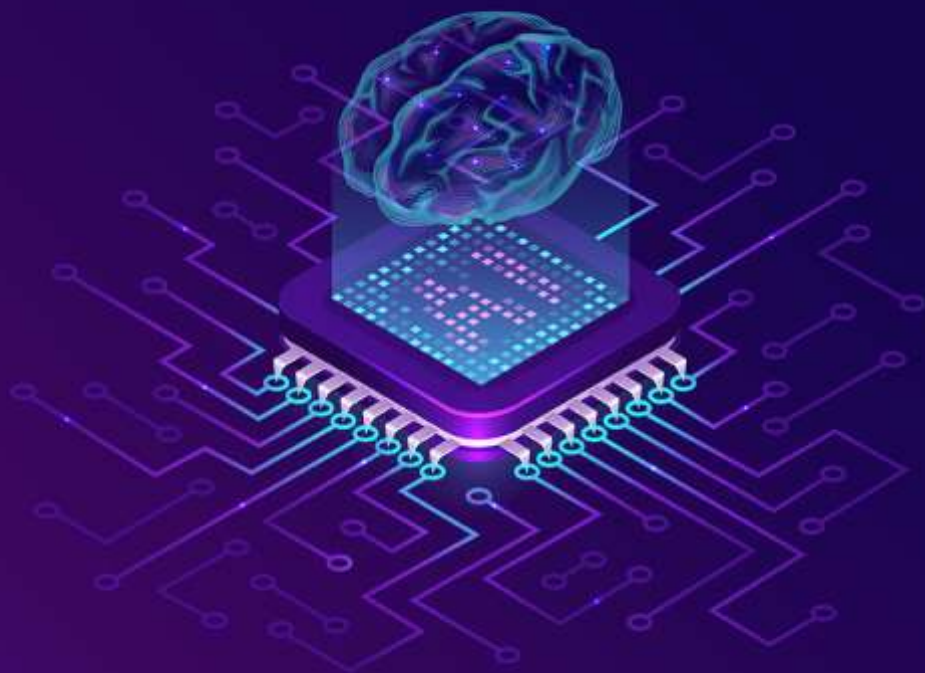


ERASLAN Hakan
GAYGUSUZ Osman
GOUDAL Victor
MAURER Gilles



UNIVERSITÉ DE TECHNOLOGIE DE BELFORT-MONTBÉLIARD

PROJET TEEKO

IA41 – A2022



Responsable de l'UV : Fabrice LAURI



Table des matières

PROJET TEEKO	1
IA41 – A2022	1
1. Introduction et objectifs du projet	3
2. Présentation et règles du jeu	4
2.1 Début de partie	4
2.2 Déplacements des pions	5
3. Modélisation du projet	6
3.1 Formalisation du projet	6
3.2 Ajout d'une partie graphique	7
3.2.1 Main menu	7
3.2.2 Game	8
3.2.3 Game Over	9
4. Analyse et principe de résolution	10
4.1 Détection des positions gagnantes	10
4.2 Algorithme MinMax et AlphaBeta Pruning	10
4.3 Heuristique	12
4.4 BFS	13
4.5 Apprentissage	14
5. Exemple de cas et exécution	15
6. Améliorations possibles et perspectives d'ouvertures	16
6.1. Optimisation de notre heuristique basé les parties précédentes	16
6.1.1. Récupération et sauvegarde des données des parties jouées	16
6.1.2. Utilisation de ces données pour améliorer l'heuristique	17
6.1.3. Utilisation de ces données pour d'autres raisons	17
6.2. Utilisation du Deep Reinforcement Learning : AlphaZero	18
7. Problèmes rencontrés et solutions trouvées	19
8. Conclusion	21

1. Introduction et objectifs du projet

Afin de mettre en pratique les principes abordés et les connaissances acquises durant ce semestre dans le cadre de l'unité de valeur IA41 (Intelligence artificielle : concepts fondamentaux et langages dédiés), nous avons eu à réaliser un projet parmi une multitude de sujets proposés. Le sujet que nous avons choisi de traiter est celui du Teeko¹.

Peu connu de nos jours, Teeko est un jeu de stratégie à somme nulle ce qui signifie que la somme des gains et des pertes de tous les joueurs est égale à 0. Par conséquent le gain de l'un constitue obligatoirement une perte pour l'autre. Ce jeu a été finalisé durant les années 1960 par le magicien John Scarne. Deux joueurs s'affrontent en duel autour d'un plateau, chaque joueur connaissant ses possibilités d'action ainsi que celles de son adversaire.

Plusieurs objectifs ont été définis pour mener à bien ce projet :

- Arborer un travail de groupe et une coordination entre chacun des membres de ce groupe.
- Découvrir les particularités du langage python et mettre en œuvre les connaissances pour la programmation.
- Mettre en œuvre un des algorithmes étudiés durant l'UV : l'algorithme Min-Max et la version optimisée Alpha-Beta Pruning, ainsi que le BFS que nous étudierons plus en détails dans la suite de ce rapport.
- Implémenter une intelligence artificielle qui sera en mesure de s'opposer à un humain ou contre un ordinateur sur le jeu Teeko. Ceci implique pour l'IA de faire le meilleur choix de coups.

Afin de répondre aux objectifs susmentionnés, nous verrons premièrement, le principe et les règles du jeu. Ensuite, notre réflexion se dirigera sur la modélisation du problème. Enfin nous étudierons les principes de résolutions utilisés. Par ailleurs, une situation d'exécution en conditions réelles sera illustrée avec des exemples d'application et nous terminerons par une conclusion.

2. Présentation et règles du jeu

2.1 Début de partie

Teeko est un jeu où deux joueurs s'affrontent en duel autour d'un plateau de taille 5x5 qui est le suivant :

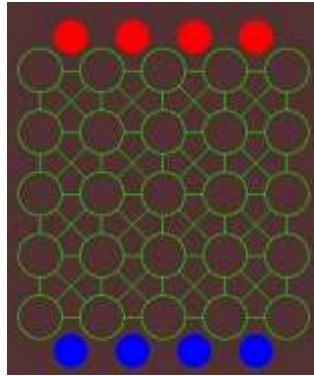


Figure 1

Chaque joueur dispose de 4 pions qui ici sont représentés avec des pions de couleur rouge pour le joueur 1 et de couleur bleu pour le joueur 2. La partie débute avec chaque joueur plaçant un pion à tour de rôle, l'objectif étant d'obtenir une position gagnante dès le début de la partie.

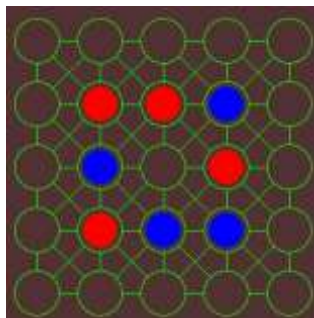
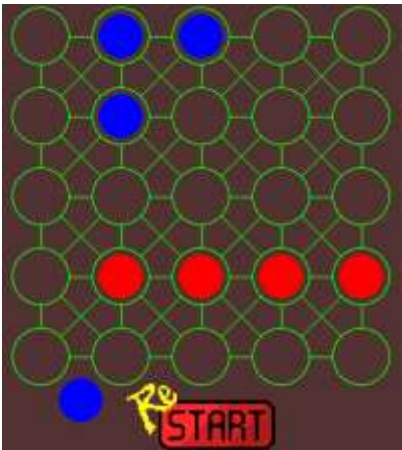
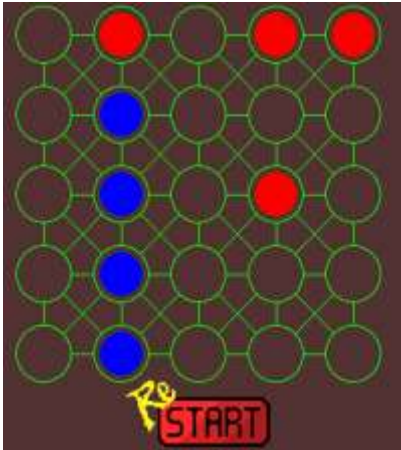
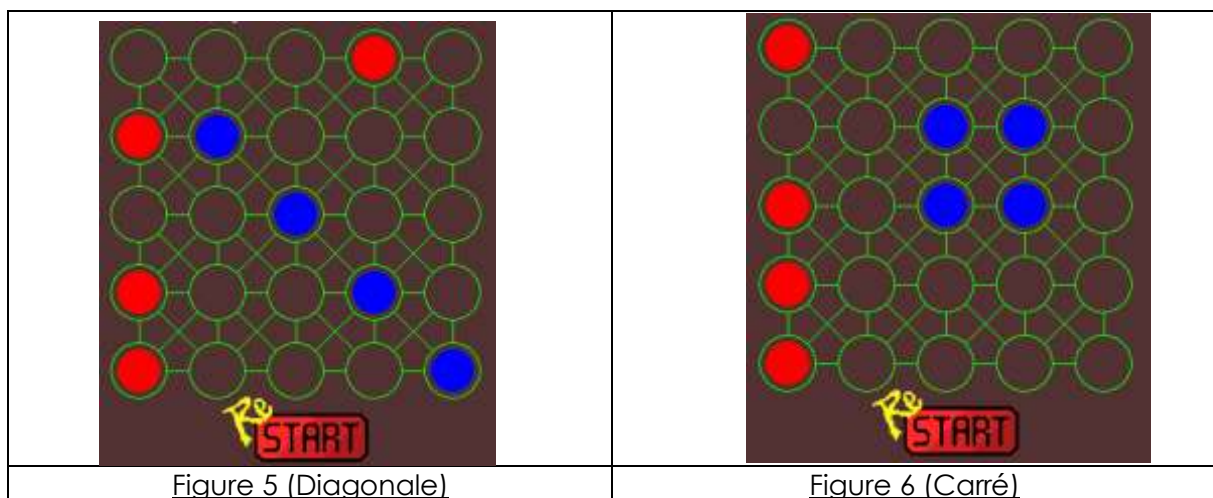


Figure 2 (exemple de début de partie)

Il existe 4 configurations gagnantes qui peuvent être formées n'importe où sur le plateau qui sont les suivantes :

	
Figure 3 (Horizontale)	Figure 4 (Verticale)



2.2 Déplacements des pions

Une fois tous les pions placés en début de partie et à défaut d'une victoire par l'un des deux joueurs, chacun doit déplacer à tour de rôle un de ses pions sur le plateau dans une des cases adjacentes au pion déplacé, le but étant toujours de réussir à former une configuration gagnante. Les déplacements ne sont possibles que si la case de destination est vide. Voici un exemple sur la figure 1.7 illustrant les déplacements autorisés avec des cercles non remplis pour le pion rouge.

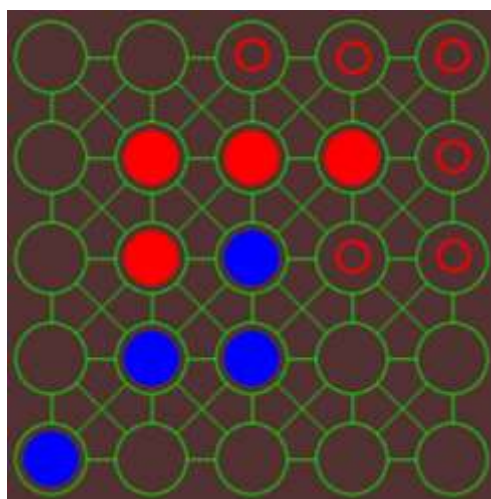


Figure 7 : Exemple de situation avec les déplacements autorisés pour le pion rouge

3. Modélisation du projet

3.1 Formalisation du projet

La spécification du jeu Teeko peut se représenter par le diagramme suivant :

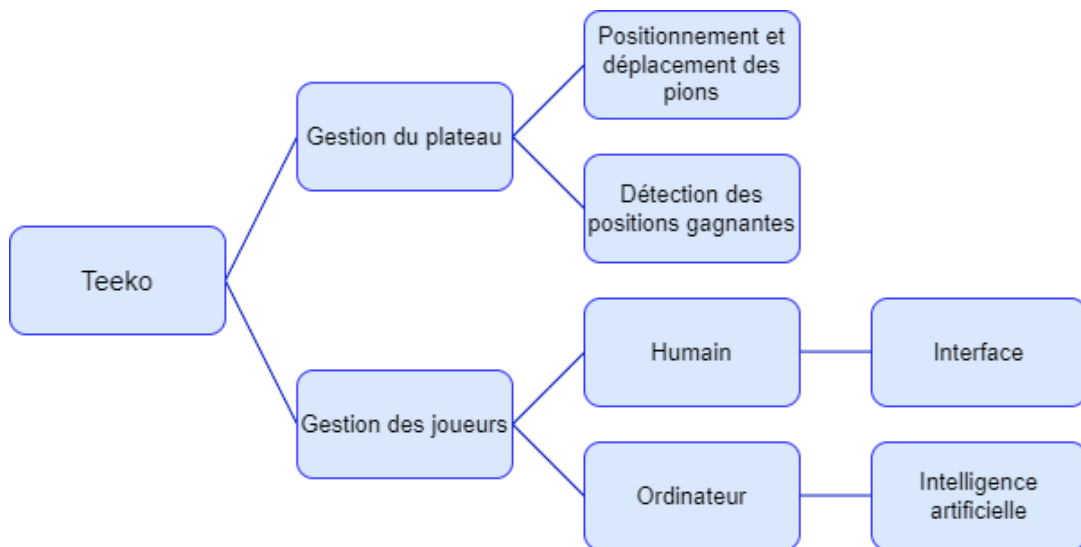


Figure 8 : Formalisation du projet

La gestion du plateau et la gestion des joueurs sont les deux éléments clés du jeu Teeko.

La gestion du plateau est séparée en deux sous-parties comme le montre le diagramme ci-dessus. La première partie se concentre sur l'administration "physique", c'est-à-dire la pose des pions au début de la partie suivi du déplacement de ces pions. La seconde composante de la gestion du plateau est un système permettant d'identifier les positions gagnantes pour chaque joueur. À tout moment de la partie, ce système doit être capable de reconnaître une position gagnante peu importe le joueur et la configuration gagnante formée.

La deuxième grande section à gérer concerne les joueurs, qu'il s'agisse de joueurs "humains" ou "Ordinateur". Pour le joueur "humain", le jeu doit disposer à la fois d'une interface pour le joueur afin qu'il puisse interagir avec celui-ci, ainsi qu'un affichage permettant de suivre le déroulement de la partie et les déplacements de l'adversaire. De plus, le joueur contrôlé par l'ordinateur doit être suffisamment intelligent pour comprendre le jeu, analyser le plateau et jouer une partie de Teeko.

Enfin, il doit bien entendu être également possible de suivre une partie entre deux joueurs "artificiels" et visualiser le déroulement sur le plateau.

3.2 Ajout d'une partie graphique

Dans un premier temps, nous sommes partis sur un affichage dans le terminal de l'IDE Python afin d'avoir un jeu qui soit jouable.

Pour simplifier la vision du jeu, nous avons ensuite remplacé l'affichage dans le terminal par un affichage sur une fenêtre graphique à part grâce à la librairie Pygame utilisée dans la classe Graphics du sous-fichier Graphics.py. Cette librairie permet la création de jeux en temps réels et d'obtenir une interface de jeu qui convient à nos besoins.

Il y a 3 interfaces qui se succèdent : le menu principal (Main Menu), le plateau de jeu où s'affrontent les joueurs et la fenêtre de fin de partie qui marque la fin de la partie (Game Over).

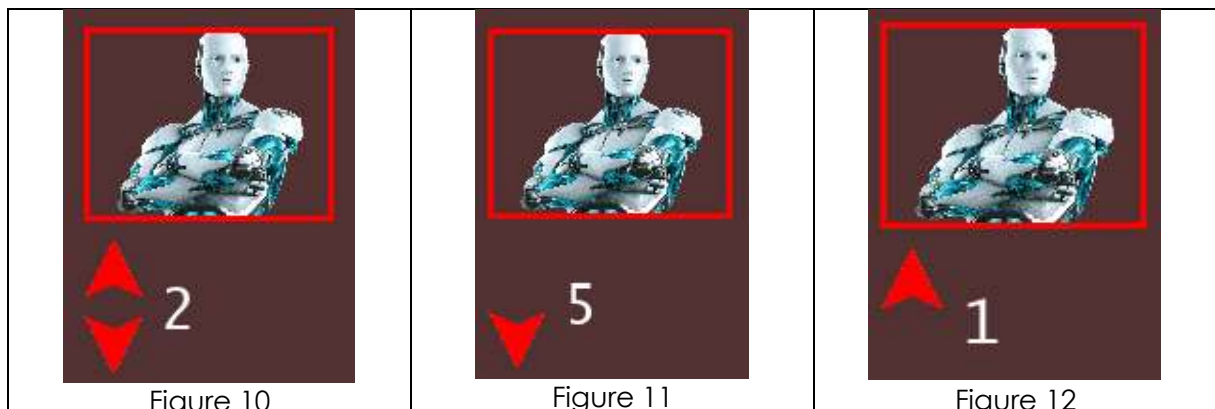
3.2.1 Main menu

Le Main Menu permet de choisir le mode de jeu, on peut donc premièrement choisir si chaque joueur est un humain ou une IA, les trois combinaisons suivantes étant possibles : joueur contre joueur, joueur contre bot, bot contre bot.



Figure 9 : Main Menu

De plus, lorsque le joueur est une IA (l'ordinateur), on peut choisir des paramètres avancés comme la difficulté du bot entre 1 et 4 (le chiffre correspond directement à la profondeur du MinMax).



Le mode Bot vs Bot propose deux options supplémentaires :

- Endless Loop : qui permet de relancer une nouvelle partie automatiquement lorsque celle-ci se termine. Ce mode est très utilisé un deep reinforcement learning pour l'entraînement d'un bot et le lancement de parties en boucle.
- Random FM : Random First Move oblige le bot à jouer un coup aléatoire. Cette utilisation sera expliquée plus en détail dans la partie 4.5. Apprentissage.



Figure 13 : Options du Bot vs Bot

3.2.2 Game

Cette étape est composée de 2 parties :

- La première phase consiste au placement des pièces de chaque joueur, à tour de rôle en commençant par le joueur rouge, il suffit de cliquer sur les cases souhaitées pour placer directement le pion sur le plateau.
- La deuxième phase après le placement des pièces, si aucune position gagnante n'a été formée lors de la phase 1, est le déplacement des pièces. Chaque joueur sélectionne un pion pour le déplacer dans les cases adjacentes au pion choisi. Les déplacements possibles sont affichés :

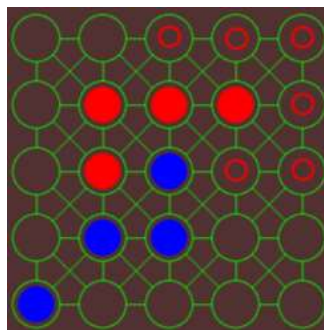


Figure 14 : déplacements possibles

3.2.3 Game Over

Lorsque la partie est terminée, le gagnant est affiché et un bouton restart apparaît. Si le mode Endless Loop a été sélectionné dans le cas d'un affrontement Bot contre Bot, la partie se relance automatiquement.

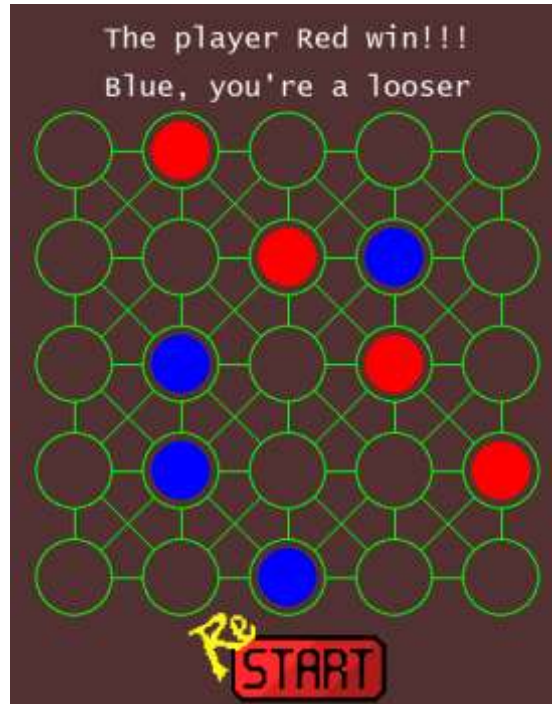


Figure 15 : fin de partie

De plus, dans le terminal sont affichées les statistiques des bots lorsqu'il y en a :

```
Stats player 1 :  
Minimax stats :  
Average time per move was : 0.11316098888892157  
Average iter per move was : 1947  
Stats player 2 :  
Minimax stats :  
Average time per move was : 0.8354790499997762  
Average iter per move was : 14405
```

Figure 16 : statistiques de fin pour les bots

Pour chaque bot est affiché le temps moyen pour calculer un déplacement, cela dépend bien évidemment de la profondeur du MinMax choisi dans le menu principal.

De plus le nombre moyen d'états parcourus pour déterminer le meilleur coup est également affiché. On remarque bien dans la figure ci-dessus qu'un algorithme de profondeur 3 calcule beaucoup plus de positions qu'un algorithme de profondeur 2.

4. Analyse et principe de résolution

4.1 Détection des positions gagnantes

La fonction *check_victory* vérifie si un joueur a gagné dans une partie donnée. La fonction prend en entrée un objet *game* et un entier *team_number* qui représente le numéro du joueur.

La fonction commence par trouver le premier pion en haut à gauche du joueur qui vient de poser ou déplacer un pion dans la grille en utilisant la fonction *find_first_pion*. Ensuite, la fonction vérifie si le joueur a gagné en utilisant différentes méthodes :

- *check_horizontal* vérifie si le joueur a gagné en alignant 4 pions horizontalement à partir du premier pion trouvé.
- *check_vertical* vérifie si le joueur a gagné en alignant 4 pions verticalement à partir du premier pion trouvé.
- *check_diagonal_right* vérifie si le joueur a gagné en alignant 4 pions en diagonale vers la droite à partir du premier pion trouvé.
- *check_diagonal_left* vérifie si le joueur a gagné en alignant 4 pions en diagonale vers la gauche à partir du premier pion trouvé.
- *check_square* vérifie si le joueur a gagné en formant un carré de 2x2 avec des pions à partir du premier pion trouvé.

Ces différentes fonctions permettent bien de vérifier toutes les positions de victoires. Si l'une de ces méthodes renvoie True, le joueur qui vient de terminer son tour gagne la partie. Sinon, toutes les méthodes renvoient False et la partie continue avec le joueur suivant.

4.2 Algorithme MinMax et AlphaBeta Pruning

Lors de la conception de l'IA Teeko, nous avons cherché un algorithme adapté à ce type de problème. Celui-ci doit permettre à l'IA de jouer le meilleur coup possible. Le jeu Teeko remplit les conditions nécessaires pour appliquer l'algorithme MinMax, c'était également l'occasion pour nous d'appliquer cet algorithme vu en classe. En effet, le jeu Teeko est un :

- Jeu à deux participants
- Jeu à somme nulle : la somme des gains et des pertes de tous les joueurs est égale à 0
- Jeu à informations complètes et précises

Le fonctionnement de cet algorithme est assez facile à comprendre. Tous les coups potentiels du joueur actuel seront testés. Il testera ensuite tous les mouvements potentiels

que l'autre joueur peut effectuer pour chacun de ces mouvements, et ainsi de suite, jusqu'à ce qu'il ait testé tous les mouvements en allant à une certaine profondeur définie lors de l'initialisation. Tous ces tests peuvent être représentés sous la forme d'un arbre, où chaque nœud représente une configuration du plateau Teeko et chaque arête indique le coup nécessaire pour y parvenir.

Une fois tout en fin d'arbre, l'algorithme doit attribuer une note à chaque configuration. Par exemple, -100 si le joueur perd, ou +100 s'il gagne. Ensuite, par back propagation, la fonction MinMax remonte dans l'arbre jusqu'au move de départ et à chaque nœud, si c'est un coup du joueur, on prend le score maximal entre tous les scores des différents fils et inversement pour les coups des adversaires. D'où le nom MinMax qui alterne entre la valeur maximale avec *maxValue* et la valeur minimale avec *minValue*.

Le point essentiel du MinMax et qui détermine l'efficacité de l'algorithme, c'est bien évidemment la profondeur (depth). Cette dernière permet de limiter l'algorithme de voir trop loin (trop anticiper). En effet, voir plus loin est important pour rendre un bot plus fort mais pour un jeu comme teeko, il y a trop de variantes à calculer pour anticiper jusqu'à la victoire ou la défaite. Le temps de calcul est donc le problème et la limite majeure. Le nombre de coups à calculer est d'environ 10^8 , ce qui est bien plus faible qu'aux échecs où l'on l'estime à 10^{120} , mais reste néanmoins toujours trop élevé avec nos ordinateurs et dans un temps raisonnable inférieur à 1 minute. Donc, par rapport au jeu, un changement de la profondeur influencera le coup choisi.

Afin de pouvoir examiner plus profondément l'arbre de test et augmenter la rapidité de l'IA, nous avons amélioré l'algorithme MinMax en utilisant l'élagage AlphaBeta. Cette optimisation permet d'éviter l'exploration d'un nœud lorsqu'il est évident que la recherche ne permettra pas d'obtenir un meilleur résultat.

Voici un exemple sur le schéma ci-dessous (Figure 17). Dans le sous-arbre de droite, on cherche le minimum de ses branches. L'exploration de la première nous renvoie la valeur 5. Or, le nœud racine est un nœud max, il va donc prendre la valeur maximale entre 6 et la valeur du sous-arbre de droite dont la valeur est 5 pour le moment. De plus cette valeur sera, à la fin, inférieure ou égale à 5 étant un nœud min. Par conséquent la valeur qui sera sélectionné sera forcément 6 et il est inutile d'explorer les branches grisées.

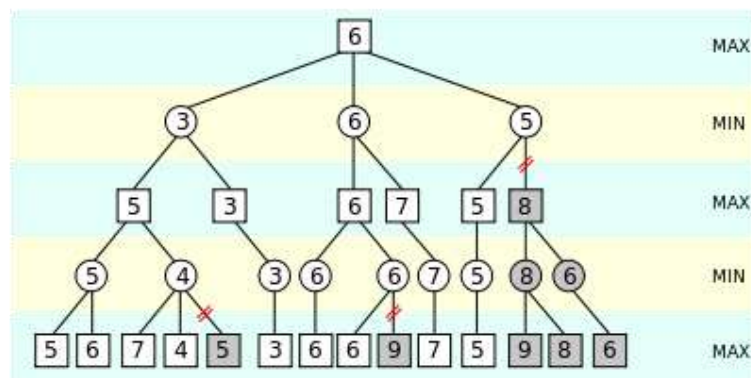


Figure 17 : exemple AlphaBeta

Finalement, nous avons choisi de créer 4 niveaux de difficultés pour l'IA. Tous les niveaux utilisent la version avec MinMax avec de l'AlphaBeta et de l'heuristique que nous abordons ci-dessous avec des profondeurs différentes selon le niveau.

Pour grandement améliorer notre algorithme, nous avons dû implémenter une heuristique qui permet d'évaluer la qualité d'une position de jeu. Elle est donc utilisée pour améliorer les performances de l'algorithme MinMax en lui permettant de donner une estimation plus précise de la valeur d'une position. En utilisant une heuristique, l'algorithme MinMax peut éviter de parcourir l'intégralité de l'arbre de jeu et se concentrer sur les branches les plus prometteuses, ce qui peut significativement améliorer ses performances.

4.3 Heuristique

Cette partie définit la fonction d'évaluation qui est un point crucial du projet puisque la pertinence des résultats dépend de celle-ci. Cette fonction permet d'évaluer une position afin de déterminer lequel des deux joueurs a l'avantage. Cette partie est définie dans notre projet par la fonction *eval*.

La fonction *eval* évalue la qualité d'un état donné de la grille de jeu. Elle prend en entrée la grille de jeu *gridC*, la profondeur de l'état dans l'arbre de recherche *depth*, une liste d'états précédemment visités *listState*, et un entier *player* qui représente le joueur qui doit jouer à l'état actuel.

La fonction commence par vérifier si l'équipe actuelle a gagné ou si l'adversaire a gagné en utilisant la fonction *check_victory*. Si le joueur actuel a gagné, la fonction renvoie $100 + depth$, ce qui signifie que l'état actuel est très favorable pour le joueur actuel. Si l'adversaire a gagné, la fonction renvoie $-100 - depth$, ce qui signifie que l'état actuel est très défavorable pour le joueur actuel.

Si aucune des deux équipes n'a gagné, la fonction vérifie si l'état actuel a déjà été visité en utilisant la fonction *is_redundance*. Si c'est le cas, la fonction renvoie $-50 - depth$, ce qui signifie que l'état actuel est défavorable car il a déjà été visité et qu'il n'apporte donc pas de nouvelles informations. Cela évite de tomber dans une boucle infinie lorsqu'on lance une partie opposant deux Ordinateurs.

Si aucune des trois conditions précédentes n'est vraie, la fonction calcule la qualité de l'état actuel en utilisant plusieurs critères :

- **spacing_value** évalue l'espacement des pions des 2 joueurs. Pour chaque joueur, la fonction récupère les coordonnées des 2 pions les plus éloignés et calcule l'aire entre ces 2 pions donc plus les pions sont éloignés plus l'aire sera grand et inversement. Et plus les pions sont éloignés les uns des autres, moins il y a de chance d'aligner les pions. Donc, pour finir, la fonction ajoute au score l'aire formée par les pions de l'adversaire (plus éloignés = plus grande aire = meilleur position = ajoute l'aire de l'adversaire au score), et enlève l'aire formée par nos pions (plus éloignés = plus mauvaise position = enlève l'aire au score). La valeur de *spacing_value* peut varier entre -21 et 21. Par exemple, si 2 pions

de l'adversaire sont aux extrémités, l'aire est alors de $5 \times 5 = 25$ et si les nôtres forment un petit carré de 2 par 2 alors l'aire est de $2 \times 2 = 4$ donc $25 - 4 = 21$. Ce score n'est pas réellement possible car si un joueur forme le plus petit carré alors l'heuristique aura déjà retourné la victoire du joueur

- **center_value** évalue la position de chaque pion des deux joueurs par rapport au centre de la grille. Plus les pions du joueur actuel sont proches du centre, plus la fonction ajoute du score et inversement pour les pions de l'adversaire. La valeur de *center_value* peut varier entre -13 et 13.
- **delta** évalue la chance du joueur actuel de gagner en diagonale. Cette fonction permet de compenser la première fonction car lorsque les pions sont en diagonale, le score retourné par *spacing_value* est très faible alors que le joueur peut être près d'une victoire. Si le joueur actuel a déjà placé 4 pions sur la grille, delta vérifie si ces pions sont disposés de manière à permettre une victoire en diagonale. Si c'est le cas, delta prend une valeur positive, sinon, elle prend une valeur négative. La valeur de delta peut varier entre -7 et 7.

Enfin, la fonction calcule la valeur de l'état actuel en additionnant les valeurs de *spacing_value*, *center_value* et *delta*, et renvoie ce total en tant que score. Plus le score est élevé, meilleur est l'état actuel pour le joueur actuel.

4.4 BFS

Le BFS (Breadth-First-Search) permet de parcourir un graphe en largeur.

Dans notre dernière version de l'algorithme, nous n'utilisons plus le BFS car nous l'avons en quelques sortes remplacé par l'utilisation de l'heuristique dans le MinMax. En effet, avant la connaissance du principe d'heuristique, nous avons déjà fait un MinMax qui n'utilisait donc pas d'heuristique. Le problème était que lorsque le MinMax ne trouvait ni de victoire ni de défaite, il renvoyait un score de 0 pour tous les coups, le bot jouait donc le premier coup qui est le coup par défaut.

Nous avons eu alors l'idée, dans le cas où le MinMax ne trouve pas de meilleur coup, de créer un algorithme complémentaire qui est ni plus ni moins qu'un BFS et qui détermine pour le joueur en question, le coup qui mène à la victoire avec le plus petit nombre de déplacements possibles. Nous avons eu cette idée car finalement, c'est un peu le raisonnement humain de trouver le coup qui nous rapproche de la victoire sans réellement calculer ou anticiper tous les contres de l'adversaire.

Nous allons maintenant voir les raisons qui nous ont poussé à définitivement le remplacer par l'ajout d'une heuristique dans le BFS, car comparé à un MinMax sans heuristique, il était bien plus performant et permet d'éviter de jouer des coups aléatoires.

- Le BFS ne prend donc pas en compte les coups de l'adversaire car il considère juste les pions adversaires comme des murs ou des obstacles or durant tous les déplacements, les pions de l'adversaires sont immobiles, ce qui n'est pas réaliste.
- Lorsque la victoire est impossible, nous avons une boucle infinie. Dans la figure ci-dessous, on remarque que le joueur rouge ne peut pas gagner donc le BFS ne peut pas trouver la victoire la plus rapide, on a alors une boucle infinie.

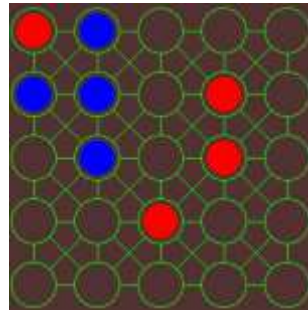


Figure 18

- Dans certain cas, l'algorithme prenait vraiment beaucoup de temps avec l'ajout d'un élagage. Dans le cas ci-dessous, le joueur rouge est à 9 coups de la victoire la plus rapide. Cependant à partir de 7 coups, l'ordinateur commence à prendre plusieurs minutes et sachant que c'est exponentiel, pour certains coups, le temps d'attente était bien trop long.

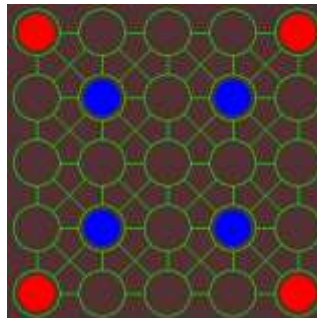


Figure 19

Pour ces raisons nous avons abandonné cette méthode et finalement l'heuristique est bien plus performante.

4.5 Apprentissage

Après avoir implémenté correctement l'algorithme MinMax avec l'heuristique, le bot était capable de jouer de manière très performante mais nous nous sommes demandé s'il était possible, à l'instar des échecs où chaque phase de jeu a son algorithme, d'optimiser la partie de posage des pions.

L'idée était donc d'implémenter un bot qui apprend lui-même de ses précédentes parties et qui chercherait uniquement à trouver le premier emplacement le plus optimal. Nous avons donc créé une grille de stats *first_pion_stat* qui représente la grille de jeu et à chaque victoire on ajoute 1 à la valeur de l'emplacement du premier pion de cette partie.

Nous avons ensuite créé une fonction qui met le premier pion de manière aléatoire et une fonction *endless* qui permet de faire jouer 2 bots l'un contre l'autre en boucle, comme vu précédemment. Le but était donc de faire jouer 2 bots l'un contre l'autre et de regarder sur un grand nombre de parties quels étaient les emplacements de départ qui menait le plus souvent à la victoire.

Tout cela a donc bien été implémenté, mais dans la réalité cette méthode possède de nombreuses limites qui font que même si elle est intéressante théoriquement, dans les faits elle n'apporte pas grand-chose.

Déjà l'heuristique ayant été améliorée pour prendre en compte la position des pions au centre de la grille, le bot n'aurait pas vraiment besoin de cette grille de statistiques pour choisir efficacement son premier emplacement de pion.

Ensuite, comme on modifie également la grille de statistiques lorsque le pion n'a pas été choisi de manière aléatoire, si on lance beaucoup de parties *normales* les 2 emplacements qui ont la plus grande valeur vont être choisis à chaque fois (selon si le bot commence ou pas) et donc c'est uniquement ces 2 emplacements qui vont être augmenté creusant ainsi l'écart. Cela pourrait amener à penser que ces 2 emplacements sont bien meilleurs que les autres alors que c'est juste qu'un plus grand nombre de parties ont été lancées depuis eux.

Cette grille de statistiques est donc à prendre avec des pincettes mais nous avons choisi de la garder car elle représente notre essai plus ou moins fructueux de faire apprendre notre bot tout seul.

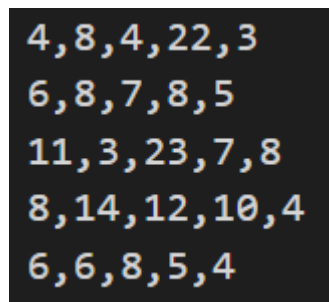
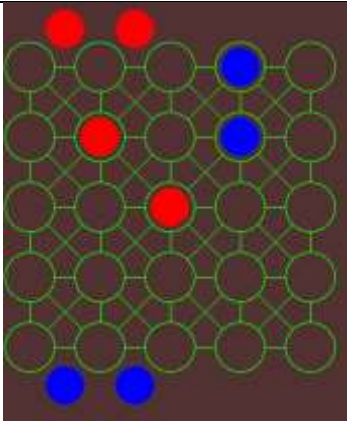
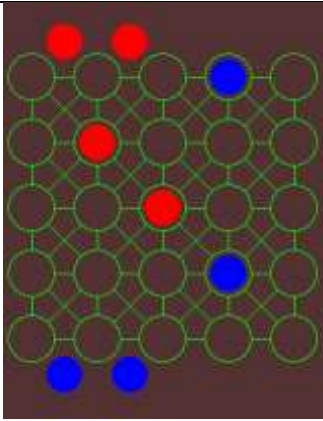


Figure 20 : grille de statistiques

Voilà donc la grille de statistiques au moment où nous écrivons ce rapport, on peut voir que le centre et l'emplacement en haut à droite dépassent de loin les autres mais comme expliqué c'est car ce sont les 2 emplacements qui avaient la plus grande valeur quand nous avons lancé de nombreuses parties sans premier pion aléatoire. L'emplacement du centre est effectivement le meilleur, celui en haut à gauche n'est peut-être pas aussi bon que ça.

5. Exemple de cas et exécution

Prenons l'exemple suivant, Le joueur rouge est le premier à commencer et c'est au tour de l'IA (en bleu). Comparons le choix que doit faire le joueur bleu suivant le niveau de l'IA.

Si l'IA du joueur bleu est d'un niveau inférieur à 3, l'IA n'aura pas assez de profondeur pour pouvoir contrer la victoire du joueur rouge en bloquant la diagonale correctement.	Si l'IA regarde plus profondément, il verra qu'il peut empêcher le joueur rouge de gagner. Il jouera donc ce coup pour contrer la diagonale.
	
Figure 21	Figure 22

6. Améliorations possibles et perspectives d'ouvertures

6.1. Optimisation de notre heuristique basé les parties précédentes

Dans l'optique de comparer les différentes versions de nos algorithmes et dernièrement d'optimiser notre heuristique, nous avons eu l'idée de sauvegarder toutes les parties jouées dans 2 fichiers .csv.

6.1.1. Récupération et sauvegarde des données des parties jouées

Dans le premier `gameStats.csv` sont sauvegardés les différentes données des parties comme : "Date, Nombre de rounds, Gagnant" et ensuite pour chaque joueur, s'il est un humain ou un bot, et si c'est un bot, on sauvegarde la profondeur, le temps d'exécution moyen et le nombre de nœuds parcourus dans le graph en moyenne. Ce premier fichier `gameStats.csv` est affiché ci-dessous :

```
resources > gameStats.csv
1 Date,round,winerP1,pP1,pP2,Depth P1,DepthP2,AvgTimeBEGP1,AvgIterBEGP1,AvgTimeBEGP2,AvgIterBEGP2,AvgTimeMBGP1,AvgIterMBGP1,AvgTimeMBGP2,AvgIterMBGP2
2 22/11/2022,97,False,False,False,2,3,,,,,0.09129095714279711,1545.1020400101265,0.0221502200001111,11621.387755102041
3 22/11/2022,35,False,False,False,2,3,,,,,0.037427966666655266,729.6666666666666,0.2525412055554423,4848.722222222223
4 22/11/2022,0,False,False,False,2,3,,,,,0.111280790090884,2287.8,0.0716918790090884,17858.6
5 22/11/2022,11,False,False,False,2,3,,,,,0.10136248333381566,2196.0,0.070930566667894,17789.666666666666
6 22/11/2022,15,False,False,False,2,3,,,,,0.07485391349974782,1544.625,0.5117140000001428,10536.625
```

Figure 23 : Aperçu du fichier `gameStats.csv`

Dans le second fichier `gameMoves.csv` sont sauvegardés tous les coups joués pour chaque partie c'est à dire : l'Identifiant Game dans `gameStats`, le Round du coup, le joueur qui effectue le coup, et le coup joué. Ci-dessous un aperçu du tableau `gameMoves.csv`.

Index	Game	Round	Player	Move
2	1	1	"	"
3	1	2	"	"
4	2	1	"	"
5	2	2	"	"
6	3	1	"	"

Figure 24 : Aperçu du tableau gameMoves.csv

6.1.2. Utilisation de ces données pour améliorer l'heuristique

Dernièrement, nous nous sommes concentrés sur l'amélioration de l'heuristique. Sachant que cette partie consiste à mieux évaluer une position et que l'amélioration se fait par tâtonnement, nous aurions pu savoir si les ajouts des différents critères *spacing_value*, *center_value* et *delta* étaient pertinents. Par exemple nous aurions pu comparer les résultats de chaque version de l'heuristique et regarder si les nouveaux critères améliorent l'estimation ou au contraire rentrent en conflit avec d'autres critères qui s'annulent donc. De plus, nous aurions pu faire varier les récompenses de chaque critère et regarder quels critères sont plus importants que d'autres et donc qui influent plus ou moins sur le score final que d'autres critères. Par manque de temps, nous n'avons pas pu implémenter cette fonction.

6.1.3. Utilisation de ces données pour d'autres raisons

Grâce à toutes ces données sur les parties précédentes, nous aurions pu implémenter beaucoup d'améliorations :

- Possibilité au joueur de pouvoir regarder l'historique de ses parties pour potentiellement les analyser plus en détail et donc pouvoir s'améliorer. Cependant, dans le contexte de devoir se concentrer principalement sur l'intelligence artificielle, cette amélioration n'était pas indispensable.
- Dans une autre branche GitHub, sauvegarde_18_11, nous avons sauvegardé toutes les versions de notre bot, nous aurions donc pu analyser toutes ces versions et donc par la même occasion les comparer, ce qui nous aurait permis de faire une analyse plus poussée de chaque élément de notre MinMax comme par exemple, l'efficacité d'augmenter la profondeur, le temps gagné par un élagage alpha-beta, les performances gagnées (ou perdues ?) par l'heuristique par rapport à l'algorithme BFS dans le cas où le MinMax ne trouve ni de victoire ni de défaite, etc... Cependant, dans le but de créer un bot qui joue au Teeko et non de comparer les performances des différents algorithmes, nous n'avons pas développé ce point et nous avons créé une version du code *main*, qui possède seulement le bot le plus performant.

6.2. Utilisation du Deep Reinforcement Learning : AlphaZero

Pour créer une intelligence artificielle capable de jouer au Teeko, plusieurs méthodes sont possibles. Dans notre rapport nous avons en quelques sortes utilisé la force brute, c'est-à-dire calculer un maximum de coup à l'avance. Cependant pour des jeux dont le nombre de variantes à calculer jusqu'à la fin est de l'ordre de 10^{12} , la force brute n'est pas la méthode la plus optimisée.

Une autre méthode consiste à faire apprendre à un réseau de neurones quel est le coup à jouer en fonction d'un état donné. Pour un jeu où 2 joueurs s'affrontent et à somme nulle, sachant que le jeu est assez complexe pour qu'on ne puisse pas calculer toutes les variantes, l'algorithme le plus optimisé aujourd'hui est **AlphaZero**.

L'algorithme AlphaZero est un algorithme d'apprentissage par renforcement développé par DeepMind, une division de recherche d'Alphabet Inc. Il est conçu pour apprendre et améliorer ses compétences en jouant à des jeux contre lui-même et est capable de réaliser des performances surhumaines dans certains jeux comme les échecs ou le Go.

L'algorithme repose sur un mélange de réseaux de neurones profonds et de l'algorithme de recherche arborescente de Monte Carlo (MCTS). Le réseau de neurones est entraîné à prédire le meilleur coup dans une position donnée en utilisant une variante de l'approche d'apprentissage supervisé connue sous le nom d'apprentissage par différence temporelle. L'algorithme MCTS est utilisé pour parcourir les coups possibles et évaluer leur probabilité de mener à une victoire.

On peut facilement le représenter sur Tic-tac-toe où l'on voit l'algorithme analyser de nombreuses variantes ce qui forme un arbre. Ci-dessous N représente le nombre de fois qu'est passé par cette état et W est le nombre de fois que l'algorithme a gagné en prenant cette voie.

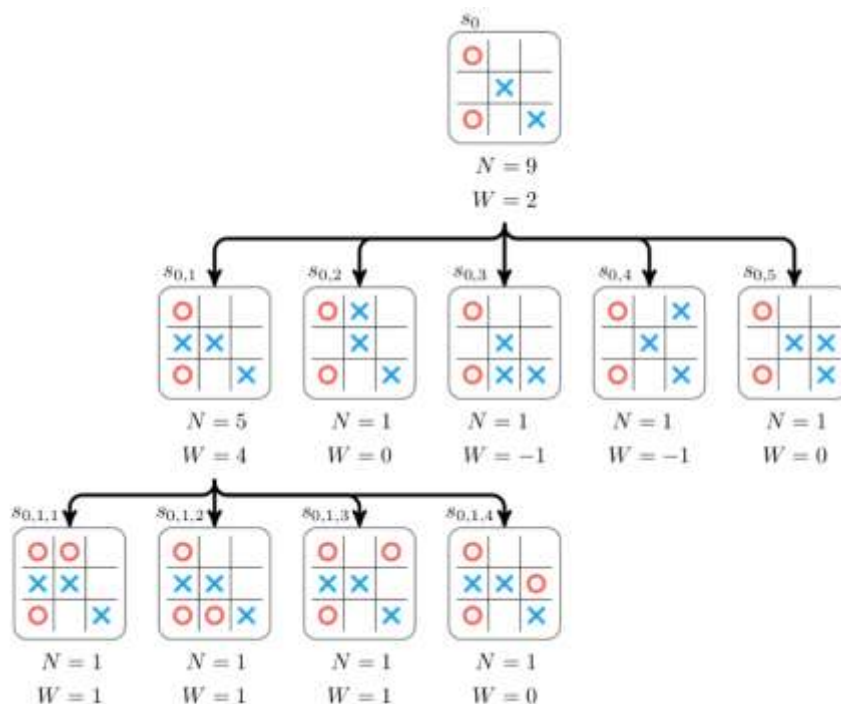


Figure 25

Pour en revenir au projet, une version simplifiée du AlphaZero peut être réalisée sur de simples ordinateurs. En effet, il est possible de créer un algorithme jouant à Puissance 4, un jeu finalement similaire donc nous pouvons aussi le faire sur Teeko. Cependant, avec le temps impartis, et partant de 0 dans le monde des réseaux de neurones, nous n'avons pas réussi à l'implémenter à notre Teeko.

7. Problèmes rencontrés et solutions trouvées

Durant la réalisation de l'intelligence artificielle, nous avons rencontrés plusieurs problèmes.

- Le problème qui nous a suivi tout au long du projet est le **temps d'exécution** du MinMax qui est finalement "la" limite à notre problème. Grâce aux différentes améliorations du MinMax, nous avons pu gagner en temps d'exécution. Une autre solution, plus complexe à réaliser aurait été d'utiliser aussi le GPU pour gagner en puissance de calcul.
- Ensuite le second problème a été la **redondance**. En effet lorsque notre algorithme a commencé à être plus performant et à ne plus utiliser de fonctions aléatoires, lorsque nous faisons du bot vs bot, les deux bots se retrouvaient très souvent dans une boucle infinie où l'un attaque et switch entre 2 positions et l'autre qui essaye de défendre et qui du coup switch aussi entre 2 positions comme on peut le voir sur cette ancienne version où il y a une boucle infinie de 4 coups.

X\Y	0	1	2	3	4
0	0	0	2	0	2
1	0	0	1	1	0
2	0	0	0	1	0
3	0	1	2	0	0
4	0	0	0	2	0

X\Y	0	1	2	3	4
0	0	0	2	0	2
1	0	0	1	1	0
2	0	0	2	1	0
3	0	1	0	0	0
4	0	0	0	2	0

X\Y	0	1	2	3	4
0	0	0	2	1	2
1	0	0	0	1	0
2	0	0	2	1	0
3	0	1	0	0	0
4	0	0	0	2	0

X\Y	0	1	2	3	4
0	0	0	2	1	2
1	0	0	0	1	0
2	0	0	0	1	0
3	0	1	2	0	0
4	0	0	0	2	0

Figure 26

Sachant que nous sauvegardions toutes les positions de la partie et que tout est sauvegardé dans une liste, nous avons simplement créé une fonction *is_redondance* qui récupère la liste des positions, et si la position est déjà dans la liste alors elle renvoie True. Si cette fonction détecte une redondance, la fonction d'éval (heuristique) qui évalue la position renvoie directement un score très négatif, ce qui incite le MinMax à ne pas considérer cette position comme viable.

- Un autre problème est apparu lorsque nous avons implémenté la partie graphique. En effet, le programme n'est pas exécuté de la même manière. Lorsque nous exécutons dans le terminal, la globalité du programme n'était lancée qu'une seule fois alors qu'en utilisant une interface graphique, l'entièreté du programme doit être exécuté à chaque frame sinon Pygame ou Tkinter plante. Donc à chaque frame, le programme doit tout réexécuter et savoir à quel moment de la partie il se trouve. Nous avons donc implémenté cette fonction :


```

def run(self):
    running = True #Variable qui permet de quitter la partie / le jeu

    while running:

        # Check for user input
        running = self.graphics.checkInput()

        # Call appropriate game function based on game state
        if self.earlygame:
            self.earlyGame()

        elif self.middlegame:
            self.play()

        elif self.gameover:
            self.gameOver()

        else:
            self.mainMenu()

        # Update the screen
        self.graphics.updateScreen()

```

Figure 27

Par exemple, comme problème concret, lorsque le joueur sélectionne un pion à bouger, il ne va pas le déplacer dans le même frame car en effet, l'algorithme met en pause le programme en attendant la réponse du joueur. Ici, il continue le déroulement même s'il n'a pas eu de réponse. Il a donc fallu retravailler le déroulement du programme pour qu'il ne plante pas lorsqu'il n'a pas reçu de valeur.

Ci-dessous la fonction qui permet le déplacement d'un pion par un humain. On remarque de nombreuses conditions *if* qui permettent de savoir où en est le programme car tant qu'il n'a pas reçu d'informations sur le pion à déplacer et où le déplacer, l'algorithme doit rester dans cette partie du programme et ne rien faire mise à part vérifier les inputs de l'utilisateur.

```

def move_pion(self, team_number):
    cancel = None
    newCoordinate = None

    if self.graphics.firstCoordinate:
        self.graphics.oldCoordinate = self.graphics.detectCercle(team_number)
    else:
        newCoordinate = self.graphics.detectCercle(0)
        if newCoordinate == None:
            cancel = self.graphics.detectCercle(team_number)

    if self.graphics.oldCoordinate is not None:
        self.graphics.firstCoordinate = False
        self.graphics.draw_circle(team_number, (abscisse[self.graphics.oldCoordinate[1]], ordonnee[self.graphics.oldCoordinate[0]], 5)
        list = list_all_possibles_moves(self.gridC, team_number)
        for move in list:
            if move[0] == self.graphics.oldCoordinate:
                self.graphics.draw_empty_circle(team_number, (abscisse[move[1][1]], ordonnee[move[1][0]], -10)

    if newCoordinate is not None:
        list = list_all_possibles_moves(self.gridC, team_number)
        move = [self.graphics.oldCoordinate, newCoordinate]
        self.graphics.firstCoordinate = True
        self.graphics.oldCoordinate = None
        if move in list:
            return move

    if cancel is not None and cancel == self.graphics.oldCoordinate:
        self.graphics.firstCoordinate = True
        self.graphics.oldCoordinate = None

```

Figure 28

- Le dernier problème est apparu lorsque nous avons implémenté l'heuristique. En effet, au début l'heuristique était seulement composée du critère *spacing_value* et comme on l'a déjà cité dans la partie sur les critères, ce critère renvoie une valeur positive dans le cas où les pions sont regroupés. Cependant dans l'exemple ci-dessous, le joueur rouge est proche de la victoire mais il aura le score minimum accordé par le critère.

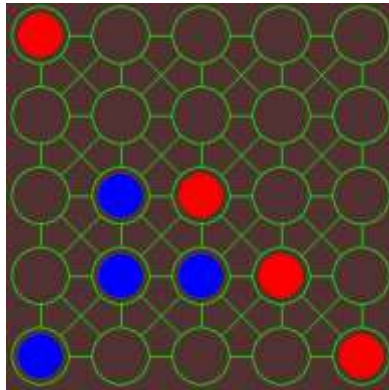


Figure 29

Ainsi nous avons dû créer une autre fonction *delta* qui compense la fonction *spacing_value* comme vu précédemment.

8. Conclusion

En somme, ce projet nous a permis de mettre en place une IA de manière ludique, en appliquant directement les connaissances abordées au cours de cette UV : à savoir l'algorithme MinMax et son optimisation AlphaBeta ainsi que la recherche d'une fonction d'évaluation.

Ce projet nous a également permis de découvrir des nouvelles techniques d'intelligence artificielle et de les appliquer à notre problème. Il nous a également permis de prendre conscience des difficultés liées au travail collectif et ceci à plusieurs niveaux.

- Gestion du temps et des emplois du temps de chacun pour réussir à se retrouver régulièrement pour faire un point sur l'avancement du projet et ainsi établir une bonne coordination.
- Gestion des capacités de chacun et répartition des tâches.

Cette organisation s'étant faite à travers divers outils (Word, Drive, Github, Discord) et à travers des réunions chaque semaine.

Ainsi nous estimons avoir rempli les objectifs demandés par le sujet bien qu'il puisse exister des améliorations possibles.

Merci pour votre lecture.