

RAPPORT D'IA41

Force3



Maureen GRANDIDIER

Ziyi HUANG

Lucas BRUTON

27/12/2021

Sommaire

I) Introduction	2
II) Formalisation du problème	3
Règles du jeu	3
Implémentation du projet	4
III) Conception du jeu	5
A) Représentation d'un état	5
B) Implémentation du jeu	6
IV) Intelligence Artificielle	7
A) Méthode de résolution: algorithme min-max	7
B) Description variable d'état	8
C) Élagage alpha-beta	10
V) Demonstration	11
VI) Améliorations possibles du projet	12
VII) Conclusion	13

I) Introduction

Dans le cadre de l'UV IA41, nous avons eu l'occasion de réaliser un projet qui nécessite une IA. Notre choix du sujet s'est porté sur la réalisation d'une Intelligence Artificielle pour le jeu Force3.

Force3 est un jeu similaire au morpion. Deux joueurs cherchent à aligner leurs 3 pions ronds verticalement, horizontalement ou diagonalement.

L'objectif de ce projet est d'implémenter une intelligence artificielle capable, si possible, de battre un être humain sur des parties de jeu ou au moins d'être un adversaire correct.

Après mûre réflexion, nous avons décidé d'utiliser l'algorithme Min-Max avec ou sans l'élagage alpha-beta pour notre IA et d'utiliser Python comme langage de programmation.

Au cours de ce rapport, nous allons vous présenter les étapes de création du projet, la conception du jeu, le fonctionnement de l'intelligence artificielle, le comportement typique de l'IA, et pour finir, la liste des améliorations possibles au projet.

II) Formalisation du problème

Avant de débiter l'implémentation du code, nous voulions être certains d'être en accord en accord sur les règles et sur le fonctionnement du jeu. Nous avons donc longuement analysé le jeu.

Voici les règles sur lesquels nous nous sommes mis en accord:

A) Règles du jeu

Force3 est un jeu où deux joueurs s'affrontent pour aligner leurs 3 pions ronds que ce soit de manière horizontale, verticale ou diagonale. Les joueurs jouent sur un plateau carré de 9 cases qui contient 8 tuiles carrés et une case vide.

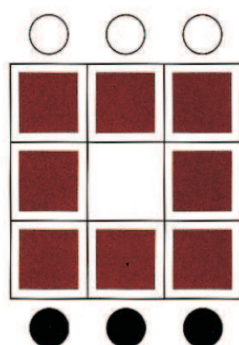


Diagramme 1 : Le tablier du jeu et l'installation des huit pions carrés au départ.

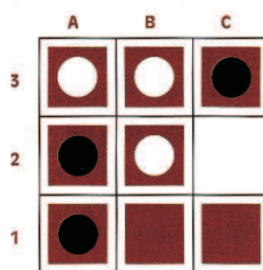


Diagramme 2 : Noir joue en poussant les deux carrés de la deuxième rangée et gagne la partie car il aligne ses trois pions en diagonale.

Fig. 1: Exemples possibles du plateau de jeu à un instant donné

Durant leur tour, les joueurs peuvent faire une action parmi les quatre suivante:

- Poser un pion rond sur une tuile inoccupée;
- Déplacer un pion rond sur une tuile inoccupée ;
- Déplacer une tuile par glissement de manière horizontale ou verticale (possible seulement si la case vide se trouve à côté de la tuile à déplacer)
- Si la case vide du plateau se trouve sur un bord, il est possible de déplacer deux carrés par glissement en un seul coup (nous avons considérés que le mouvement était possible seulement si les deux tuiles étaient alignées en direction de la case vide, comme le coup réalisé sur le diagramme 2 ci-dessus)

Si le joueur a déplacé une tuile ou deux, il est impossible au tour suivant de les déplacer dans le sens inverse pour remettre les tuiles dans leur position initiale.

Par contre, si un joueur fait glisser deux tuiles, au coup suivant son adversaire pourra déplacer l'une de ces tuiles puisque la position précédente ne sera pas reconstituée

B) Implémentation du projet

Après avoir établi les règles du jeu, nous nous sommes penchés sur la manière dont nous allions implémenter le projet. A partir de ce que nous avons vu en cours, nous avons décidé d'utiliser l'algorithme min-max. Il a été décidé que nous testerons notre code avec et sans élagage alpha-beta pour voir dans quel cas notre IA serait la plus performante.

Notre priorité une fois l'algorithme choisi, a été de créer un jeu fonctionnel avec deux joueurs humains et que nous aurions simplement à remplacer un des joueurs par l'IA. Cette partie du code a été plus difficile que ce que nous pensions à réaliser, nous ne nous rendions pas compte à l'époque du travail que cela représentait. Cependant, nous n'en étions pas assez loin dans nos cours à l'époque pour commencer directement la programmation de notre IA.

Pendant qu'un membre de notre groupe réalisait les bases du jeu, les autres membres ont pu réaliser des recherches sur le fonctionnement de l'algorithme min-max et ont pu ensuite réfléchir sur comment implémenter l'algorithme au projet.

Après avoir réalisé une première version du jeu, nous nous sommes rendu compte qu'il manquait des éléments importants pour une IA performante. Nous avons donc réalisé une seconde version du jeu auxquels nous avons ajouté une multitude d'éléments et d'améliorations (dont la création d'une classe Node pour les nœuds et EdgeAction pour les actions) que vous pourrez retrouver dans la partie "(III) A) Représentation d'un état".

Mademoiselle Huang ne parlant pas français, cela a été une difficulté supplémentaire dans notre projet pour pouvoir nous expliquer et être sûr que tout le monde avait compris la même chose. C'est d'ailleurs pour cela que notre code est rédigé et commenté en anglais. Mais nous nous sommes dit que c'était une expérience pour notre future vie professionnelle, car nous serons peut-être amenés à travailler avec des gens ne parlant pas notre langue maternelle. Par ailleurs maintenant, le développement de code se réalise majoritairement en anglais, même en France.

En résumé, la conception du projet s'est réalisée en 4 parties: réflexion sur le jeu et ses règles, réflexion sur l'IA, l'implémentation et amélioration du jeu, et pour finir l'implémentation de l'IA.

III) Conception du jeu

A) Représentation d'un état

Un état représente un plateau de jeu à un instant donné durant la partie. Chaque tour de jeu résulte par une action d'un joueur: poser ou déplacer un pion, déplacer une tuile voir deux si possible. Un pion peut être déplacé dans 8 directions. On ne peut pas poser un pion sur une case vide ou sur une tuile contenant déjà un pion. Une tuile possède seulement 4 déplacements.

Au départ, nous avons créé le plateau avec un système de liste sans utiliser de coordonnées. Nous proposons au joueur de choisir une tuile selon un numéro comme présenté ci-dessous:

1	2	3
4	5	6
7	8	9

L'ordre de ces numéros ne changeait jamais, c'était la tuile lorsqu'elle était déplacée, qui se mettait à jour et prenait la nouvelle valeur dans une variable nommée `idTile`. Mais nous nous sommes trouvés très vite limités dans nos fonctions et le développement de notre code.

Par ailleurs nous avons vu durant un TD que le jeu du taquin était simulé avec une matrice. Le plateau de Force3 ressemblant à celui du taquin, nous avons utilisé ce système de matrice plutôt que de liste en mettant en place un système de coordonnées avec des coordonnées `x, y` allant de 0 à 2.

Nous avons inclus ces coordonnées dans certaines classes objets comme celles des tuiles et des pions. Cela fut très utile pour bouger un pion par exemple, puisque avant le plateau interrogeait l'emplacement choisi pour savoir s'il contenait une tuile qui elle-même demandait si elle contenait ou pas un pion. Alors qu'après l'amélioration, le pion connaissait ses propres coordonnées, il suffisait de le récupérer grâce à son `id` et de lui indiquer où se déplacer.

D'ailleurs, stocker les pions dans la classe objet `Player` et récupérer ses données grâce à son `id` a été une amélioration importante dans la création de notre code. Sans cette amélioration, le code était moins performant lors de la création de notre IA. En effet, pour certaines fonctions d'évaluation comme `eval_pawn_between` qui demande au plateau de jeu si un des pions de l'IA se trouve entre deux pions de l'adversaire, cela signifiait qu'il fallait interroger le plateau case par case jusqu'à trouver un pion et devoir interroger ce pion pour savoir s'il appartenait au joueur ou pas grâce à sa couleur.

Par ailleurs, les joueurs possèdent chacun un numéro et une couleur qui ont été déclarés dans une classe `CONSTANTE` car ils ne changent pas durant la partie.

Ainsi, un état correspond à l'état du plateau de jeu selon un joueur et selon l'emplacement de des tuiles et des pions présents sur le plateau.

B) Implémentation du jeu

En nous appuyant sur les règles de jeu présentées au cours de la partie “I) Formalisation du problème”, nous avons réalisé les classes suivantes pour le jeu:

- **Player:** Cette classe représente un joueur. Un joueur est initié avec la couleur noire ou blanche et peut placer un total de 3 pions ronds sur le plateau. Lorsqu’il place un pion, cet objet pion est ajouté au tableau pawns dans sa fonction d’initialisation, ce qui est extrêmement pratique pour récupérer le nombre de pions que possède le joueur et leurs données.
- **CircularPawn:** Cette classe représente un pion rond. Elle contient des coordonnées et un id pour pouvoir différencier les pions.
- **SquarePawn:** Cette classe représente une tuile carrée. Les objets de cette classe seront utilisés pour contenir les pions ronds posés sur le plateau. Ils doivent pouvoir effectuer des glissements de une ou deux cases.
- **Tile:** Cette classe représente une case du plateau. 9 objets de cette classe seront créés par le plateau et 8 de ces cases posséderont un pion carré à un instant donné.
- **Board:** Cette classe représente le plateau. C’est à travers cette classe que les joueurs pourront interagir avec le jeu.

Voici le diagramme des classes décrites précédemment:

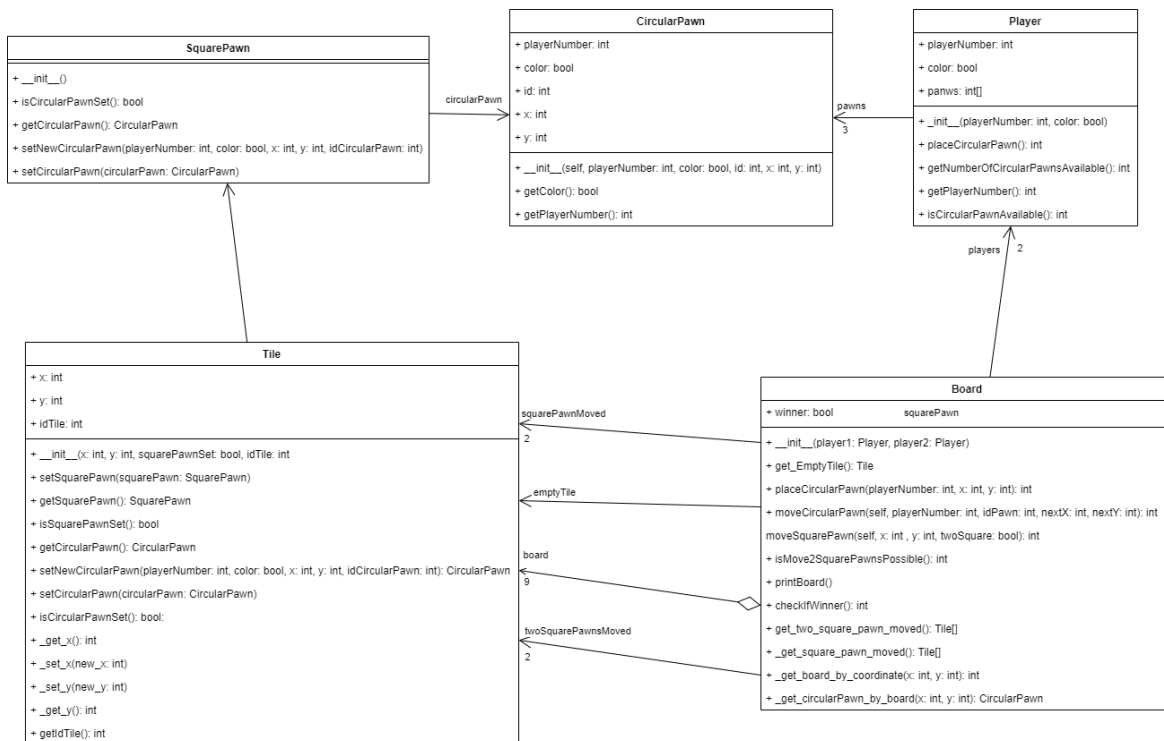


Fig. 2: Diagramme des classes du jeu

IV) Intelligence Artificielle

A) Méthode de résolution: algorithme min-max

L'**algorithme min-max** est un algorithme qui s'applique en général assez bien pour des jeux à deux joueurs comme le Force3 par exemple. Cet algorithme consiste à minimiser la perte maximum (c'est-à-dire lorsqu'on envisage le pire des cas).

Dans notre code, cet algorithme amène l'ordinateur à passer en revue toutes les possibilités pour un nombre limité de coups et à leur assigner une valeur qui prend en compte les bénéfices pour le joueur et pour son adversaire. Le meilleur choix est alors celui qui minimise les pertes du joueur tout en supposant que l'adversaire cherche au contraire à les maximiser. En effet, le jeu est à somme nulle, la somme des gains et des pertes de tous les joueurs est égale à 0. Cela signifie donc que le gain de l'un constitue obligatoirement une perte pour l'autre.

Dans notre cas, un nœud représente un état du jeu et nous définissons une relation d'ordre qui permet de choisir un nœud plutôt qu'un autre. Cette relation d'ordre est définie par la fonction evaluation.

```
def MaxValue(node, depth, player, other_player):  
    if (node.board.checkIfWinner() != -1 or depth == 0):  
        node.value = evaluation(player, other_player, node.board)
```

Fig. 3: Fonction MaxValue

```
def evaluation(player_max: Player = None, player_min: Player = None, board: Board = None):  
    if (board.checkIfWinner() == player_max.playerNumber):  
        return 500  
    if (board.checkIfWinner() == player_min.playerNumber):  
        return -500  
    else:  
        score = eval_position_pawn(player_max)  
        score += eval_nb_pawn(player_max, player_min, board)  
        score += eval_pawn_between(player_max, board)  
        score += eval_opposition_edge_pawn(player_max, board)  
        score += eval_pawn_alignment(player_max)  
        return score
```

Fig. 4: Fonction evaluation

On appelle la fonction evaluation à partir de l'état du plateau d'un nœud en particulier. Si l'IA gagne avec ce nœud, la valeur 500 est renvoyée. Si son adversaire gagne par contre, la valeur -500 est retournée. Si il n'y a aucune victoire des deux côtés, on additionne les valeurs rapportées par les différentes fonctions d'évaluation.

Les différentes fonctions d'évaluation prennent en compte:

- La position du pion sur le plateau (plus avantageux s'il se trouve sur un coin ou en particulier au milieu car cela permet de créer plus facilement un alignement de 3 pions. On a respectivement 3 possibilités d'alignement avec un coin (horizontal, vertical, une diagonale) et 4 avec la case du milieu (horizontal, verticale, et deux diagonales).
- Le nombre de pions: si un joueur a plus de pions que son adversaire, la situation est plus avantageuse pour lui et inversement.
- Si le joueur empêche son adversaire de gagner en mettant un pion après l'alignement de deux pions adverse ou entre deux pions adverse.
- Si le joueur effectue un alignement avec deux de ses pions (même en ayant un espace entre, par exemple s'il a un pion en [0][0] et un autre en [0][2])

B) Description variable d'état

Pour appeler l'algorithme MinMaxPL on utilise un objet Node. Cet objet possède une profondeur qui correspond à la profondeur de l'arbre. Il possède également l'état et les données des deux joueurs, une sauvegarde de l'action effectuée durant un nœud particulier, la valeur de cette action, l'état du plateau et surtout, ce nœud contient les enfants qu'il engendre grâce à la fonction `createChild()`.

La taille de l'arbre du Force3 étant assez grande, il faut pouvoir évaluer des nœuds non-terminaux. C'est pour cela que nous utilisons MinMax avec une profondeur limitée. Cette profondeur limitée permet d'étendre l'arbre de jeu jusqu'à une profondeur N à partir du nœud courant. Nous avons choisi une profondeur de 3 pour de simples raisons de temps de calcul. En effet, voici le temps d'exécution en secondes de l'IA pour une profondeur de 3.

Temps d'execution	Temps d'execution	Temps d'execution
0.8642368316650391	1.3204188346862793	1.306291103363037

Fig. 5: Temps d'exécution pour une profondeur de 3

Pour des actions similaires, voici un temps d'exécution pour une profondeur de 4.

Temps d'execution	Temps d'execution	Temps d'execution
14.101321697235107	34.833699226379395	19.426950454711914

Fig. 6: Temps d'exécution pour une profondeur de 4

Ces temps étant beaucoup trop longs pour une expérience de jeu agréable, nous nous sommes donc décidés sur 3.

Un exemple de création d'arbre et de noeud avec cette profondeur après que j'ai placé un pion en [0][0]:

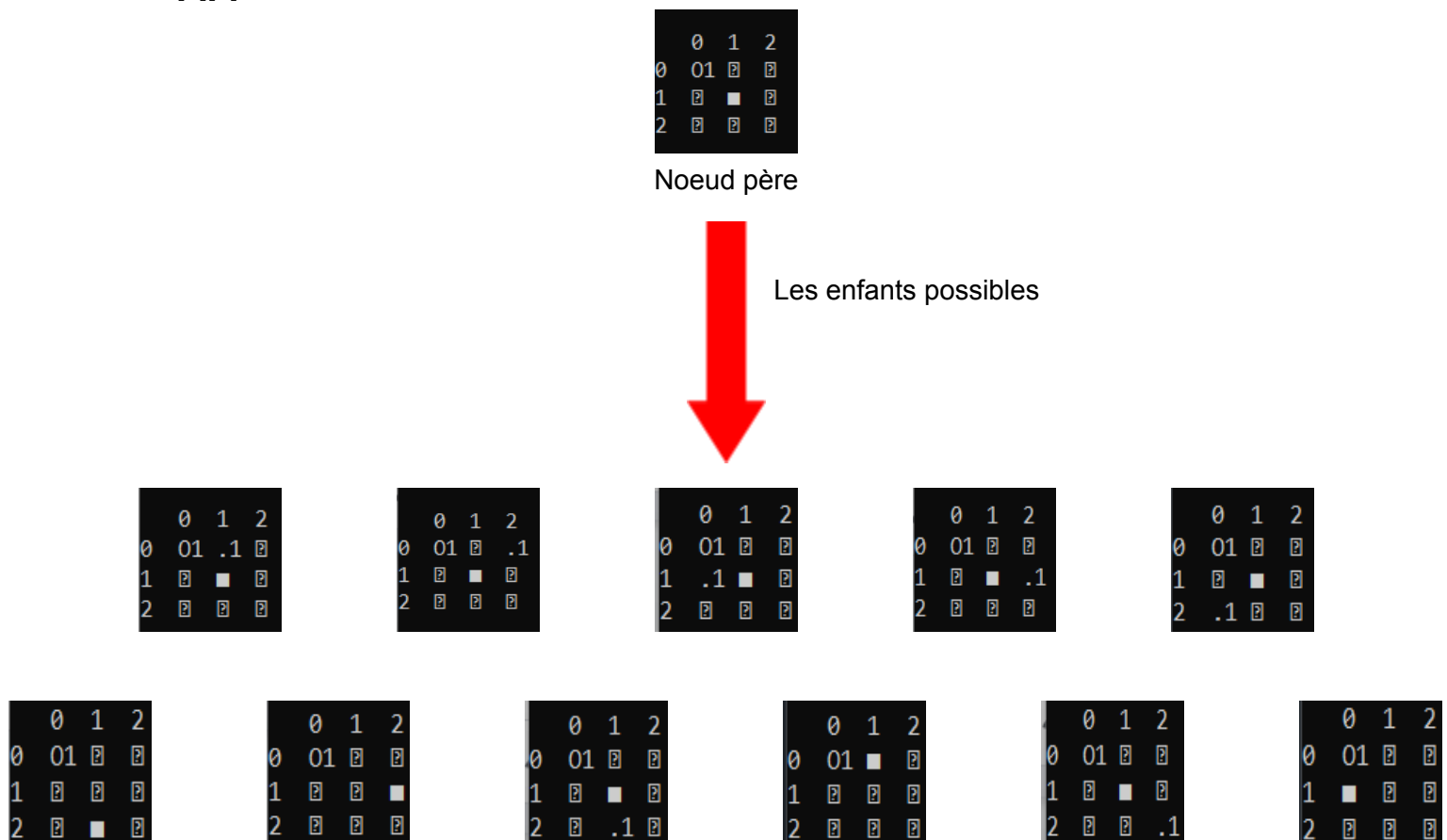


Fig. 7 : Exemple d'arbre

Ensuite, cet algorithme MinMaxPI renvoie une valeur stockée dans v . v représente la valeur de la meilleure action possible. Si parmi les nœuds développés, une des actions a le même score que v et qu'elle ne fait pas gagner l'adversaire, on l'ajoute aux actions possibles. S'il en existe plusieurs, on choisira ensuite au hasard une de ces actions possibles. Le noeud contient dans ses attributs un objet action qui contient le numéro de l'action à effectuer, l'id du pion (utile si l'action consiste à bouger un pion), ainsi que les coordonnées de l'action.

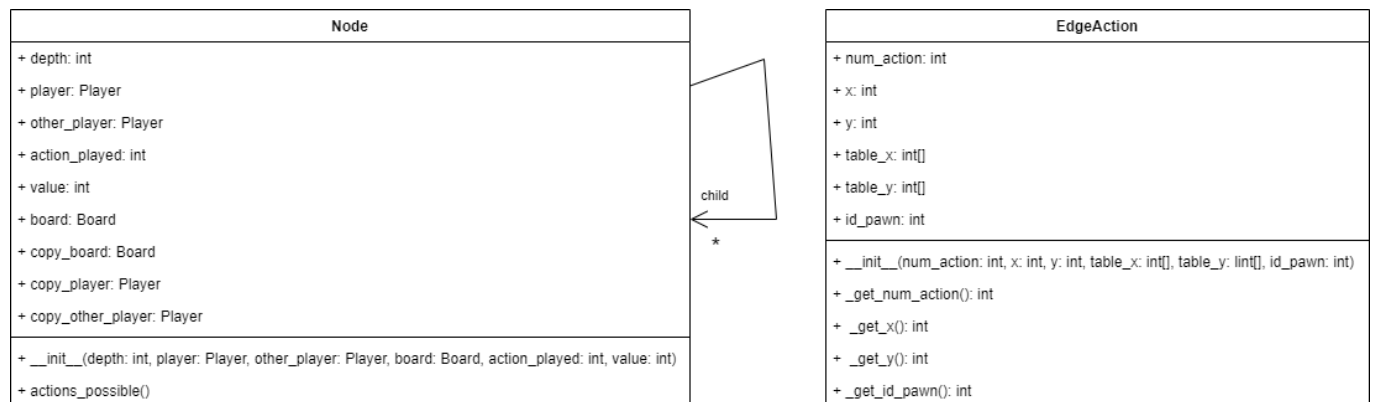


Fig. 8: Diagramme de classe de Node et EdgeAction

C) Élagage alpha-beta

Nous n'avons pas remarqué de différences avec ou sans l'élagage alpha-beta.

Nous savons que cet élagage permet d'arrêter de générer les successeurs d'un nœud dès qu'il est évident que ce nœud ne sera pas choisi, grâce à la conservation d'une valeur alpha et beta. L'élagage Alpha-Beta tient à jour ces deux variables qui contiennent respectivement à chaque moment du développement de l'arbre la valeur de son meilleur successeur trouvé jusqu'à présent et la valeur de son pire successeur trouvé jusqu'à présent.

L'élagage Alpha-Beta empêche toute possibilité de stratégie et de vision sur le long terme étant donné que toutes les actions sont indépendantes les unes des autres.

Nous pensons que notre IA n'est pas assez complexe pour voir une réelle différence avec ou sans élagage Alpha-Beta. Les deux versions sont présentes dans notre code.

V) Demonstration

Pour jouer, il faut choisir les actions que l'on souhaite réaliser grâce au clavier. Le plateau est réimprimé à chaque tour.

Un message de félicitation s'affiche et le jeu s'arrête lorsqu'un des joueurs gagne.

En règle général l'IA va:

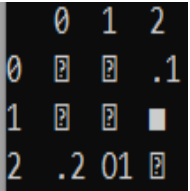
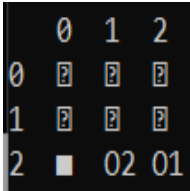
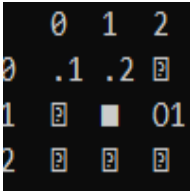
S'interposer si l'adversaire a deux pions alignés ou deux pions sur une même ligne/colonne/diagonalle et donc possibilité de gagner:

Ligne	Diagonale	Colonne
 <pre> 0 1 2 0 . 2 01 1 0 0 0 2 . 1 0 </pre>	 <pre> 0 1 2 0 01 0 0 1 . 1 02 . 2 2 0 0 . 3 </pre>	 <pre> 0 1 2 0 01 0 0 1 02 0 0 2 . 1 0 0 </pre>

Privilégier le fait de gagner à n'importe quelle autre action (ici par exemple, elle aurait pu vouloir s'interposer pour éviter que l'adversaire gagne mais privilégie sa propre victoire)

 <pre> 0 1 2 0 . 2 01 1 . 3 0 0 2 0 . 1 02 </pre>	→	 <pre> 0 1 2 0 . 2 01 1 . 3 0 0 2 . 1 0 02 </pre>
--------------------------------------------------------------------------------------------------------------------------------------	---	---------------------------------------------------------------------------------------------------------------------------------------

Va essayer de placer ses pions sur la même colonne ou ligne ou diagonale.

 <pre> 0 1 2 0 0 0 . 1 1 0 0 0 0 2 . 2 01 0 </pre>	 <pre> 0 1 2 0 0 0 0 1 0 0 0 2 0 02 01 </pre>	 <pre> 0 1 2 0 . 1 . 2 0 1 0 0 0 01 2 0 0 0 </pre>
---------------------------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------

VI) Améliorations possibles du projet

Après avoir terminé de programmer le projet, voici des améliorations possibles au projet auxquels nous avons pensé:

- Amélioration de l'IA:
 - Faire en sorte que l'IA utilise toutes les actions possibles d'un joueur classique: malgré que nous ayons implémenté la possibilité à l'IA d'effectuer le déplacement de deux tuiles en un seul coup, celle-ci n'utilise jamais cette action sans que nous ne comprenions réellement pourquoi.
 - Améliorer la fonction d'évaluation d'un état:
 - Nous savons qu'il reste encore beaucoup de choses à prendre en compte lorsque nous évaluons une action jouée. Par exemple un cas qui est revenu assez souvent: Si un joueur a ses pions en [0][0] et [0][1] et que l'IA met la case vide en [0][2], elle considère que le joueur ne peut pas gagner. Et si ce joueur met un pion en [1][2], elle ne voit pas de danger puisque techniquement, le joueur n'a pas de possibilité de remporter le tour en plaçant son pion ici. Cependant, le tour d'après, il suffit au joueur de déplacer la tuile contenant son pion en [0][2] pour qu'il remporte la partie. L'IA ne trouve pas de solution pour contrer ce coup car elle ne peut pas poser un de ces pions sur la case vide pour contrer son adversaire et la seule tuile qu'elle pourrait mettre sur le chemin des deux pions alignés ferait gagner son adversaire.

```

O O [ ]
■ ■ O
■ ■ ■

```

Fig. 9: Situation problématique pour l'IA

- Implémentation d'une interface graphique ;
- Améliorer le temps d'exécution du programme en écrivant du code plus performant.

VII) Conclusion

Ce projet a été très formateur. Il nous a permis de mettre en œuvre une multitude de notions vu en cours comme l'algorithme Min-Max.

De plus, le projet nous a donné l'occasion de prendre beaucoup de décisions pour l'implémentation du jeu et de l'IA, de réfléchir à ce qu'il fallait prendre en compte ou non dans les décisions de notre IA, ce qui pouvait l'impacter, ce que nous devions prioriser comme actions...

Malheureusement, nous nous sommes concentrés sur le fait d'avoir une IA la plus performante possible et par manque de temps, nous n'avons pas eu le temps de créer une interface graphique pour le jeu. Même si le rendu sur console offre une expérience de jeu correcte, nous regrettons cette mauvaise gestion du temps.

Malgré cela, nous sommes globalement satisfaits de notre compte-rendu pour le projet.

Pour finir, la réalisation d'une IA a été passionnante en plus d'être pédagogique et nous a permis de nous rendre compte de la difficulté à implémenter une intelligence artificielle dans un jeu.