Rapport Battle IA

Table des matières

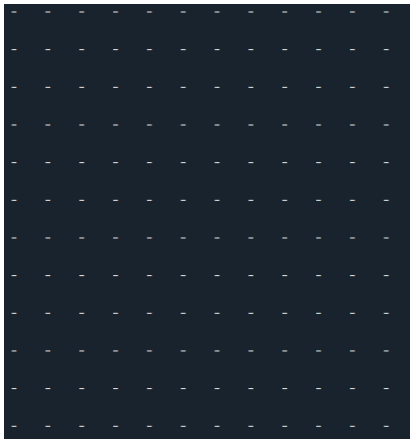
I. Intro :

L'objectif de ce projet est de modéliser une IA qui serait capable de jouer aux morpions dans un format spécial (plateau 12x12, alignement de 4 pour gagner) et d'expliquer les choix que nous avons fait suite à nos tests.

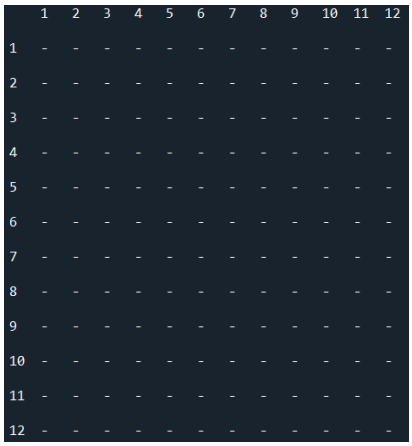
II. Les différents algorithmes:

1) Affichage :

Pour commencer, nous avions opté pour un affichage assez basique comme s'en suit :



Ensuite, nous avons opté pour un affichage plus ergonomique en accord avec les numérotations imposées par le sujet (1 à 12) :



Ainsi, il est plus facile de se repérer dans le plateau pour jouer les coups sans compter les colonnes et les lignes à chaque fois.

2)Coups Possibles :

Cet algorithme permet d'optimiser le choix des coups de l'IA. En effet, il établit une liste de positions vides voisines des positions pleines (occupées par le joueur et l'IA). En d'autres termes, jouer aléatoirement ou bien de manière trop éloignée des coups déjà joués empêche l'efficacité et la mise en place de stratégies. Afin d'éviter un parcours de tout le plateau inutilement, les coups sont enregistrés au cours de la partie afin que les coups possibles soient vérifié à côté de ces cases.

3)Eval et Score :

Score permet d'attribuer un score à des état de la partie donnés selon une série de coups définie. Toutes les possibilités de coups possèdent ainsi un score qui leur est propre. De cette manière, on peut facilement comparer les différentes séries de coups possibles et permettre à l'IA d'établir des stratégies selon ce score.

Par convention, plus une série de coups ou un état de la partie sont avantageux pour l'IA, plus le score est faible, et inversement pour le joueur.

Nous avons établi les scores de manière expérimentale, en notant les positions avantageuses et désavantageuses en attribuant des coefficients en conséquence comme ci-dessous. Un facteur '*(-1)' est appliqué à celles qui avantagent l'IA.

```
a = 20 #[x,x], [o,o]
b = 60 #[x,x,x], [o,o,o]
c = 41 #[x,o,o],[o,o,x],[o,x,x],[x,x,o]
d = 140 #[o,o,o,x],[o,o,x,o],[o,x,o,o],[x,o,o,o],[o,x,x,x],[x,o,x,x,x],[x,x,o,x],[x,x,x,o]
e = 50 #[-,x,x,x,-],[-,o,o,o,-]   (2*b ?)
f = 15 #[x,-,x], [o,-,o]
g = 41 #[o,x,o],[x,o,x]
```

En plus de cela, la fonction Eval attribue un score en donnant un score encore meilleur au coup amenant à la victoire.

4) MinMaxAlphaBeta :

Cet algorithme nous a été imposé. Il permet en effet de pondérer les séries de coups grâce à notre système de score jusqu'à une certaine profondeur. Une profondeur sur deux, l'algorithme va choisir la série de coup qui maximise le score (pour le joueur), puis le minimiser (pour l'IA).

Nous avons préféré le coder en récursif afin de gagner en complexité. De ce fait, le programme s'exécute de manière assez instantanée jusqu'à la profondeur 4. Le programme est trop lent pour une profondeur de 5 (19 secondes en moyenne) donc l'algorithme est optimisé pour une profondeur 4 car les coups joués sont les meilleurs en étant assez rapides (3 secondes en moyenne).

5) Action et Resul :

Ces algorithmes permettent de choisir un coup avec Action, puis de jouer ce coup choisi avec Resul. Nous avons opté pour différencier ces fonctions pour le joueur et l'IA afin de sécuriser les saisies. Concernant le joueur, la saisie est sécurisée des numéros de colonnes et de lignes qui respectent l'affichage. Pour ce qui est de l'IA, tant qu'un coup est impossible, l'IA choisit un autre coup.

6) Winner et Win :

Winner permet de vérifier si un joueur a gagné à l'aide d'un booléen.

Win permet de savoir qui a gagné en retournant 1 pour le joueur, -1 pour l'IA et 0 pour un match nul.

Pour optimiser la complexité de ces fonctions, on utilise un return dès lors que la condition est respectée.

7) Cases vides et TerminalTest :

Cases vides permet de vérifier si le plateau est plein pour traiter le cas du match nul

En ce qui concerne la victoire, les fonctions Win et Winner nous sont utiles

A l'aide de celle-ci, TerminalTest permet d'arrêter la partie en cas de victoire ou de match nul.

8) Partie et Copie plateau :

Copie plateau permet de copier en profondeur l'état du plateau afin que l'IA simule ses coups lorsqu'il exécute MinMaxAlphaBeta sans que cela n'ait d'incidence sur la partie en cours avec le joueur.

Nous avons optimisé l'ergonomie de l'algorithme qui lance la partie.

```
L'IA joue les X
Le joueur joue les O

Saisir qui joue en 1er : saisir 'O' pour Joueur | 'X' pur l'IA :
```

Par convention, l'IA joue les 'X' et le joueur les 'O'.

La saisie du joueur est sécurisée.

Nous avons aussi pensé à afficher un compteur de tours.

Le tour de l'IA se lance après qu'on appuie sur 'Entrée'.

A la fin de la partie, un message s'affiche pour déclarer le vainqueur de la partie ou bien le match nul.

III. Idées d'optimisations et tests :

1) Tri des coups possibles :

Pour améliorer la rapidité de notre algorithme MinMaxAlphaBeta, nous avions pensé à quelques idées pour éviter d'explorer les possibilités de coups trop inutiles comme le **tri en**

profondeur croissant selon le score pour le tour de l'IA (Min), décroissant pour le tour du joueur (Max).

De plus, nous avions pensé à une **parallélisation** : pour chaque profondeur, les séries de coups sont explorées et leurs scores pondérés simultanément.

Par ailleurs, nous avons aussi pensé à **trier les coups possibles selon leur nombre de voisins** afin d'éviter à l'IA de contrer des coups trop inoffensifs du joueur.

Cependant, lors de nos tests, nous n'avions pas relever de différence notoire en ce qui concerne le tri et la parallélisation. Pour ce qui est du tri des coups possibles selon les voisins, cela entachait considérablement la notion de stratégie donc nous avons abandonné cette idée.

2)Simulation :

Nous avons pensé à un algorithme de Simulation qui pour la prise de décision du coup à jouer pour l'IA, **simule un nombre de parties définies jusqu'à la fin. Les coups joués par l'IA durant chaque partie sont ainsi enregistrées puis trié selon la longueur de la série de coups. Après quoi nous retournons le 1er coup de la série qui amène à la victoire en le moins de coups possibles.**

Après de nombreux tests, nous avons conclu que les coups joués étaient certes d'une qualité optimale, mais l'IA mettait un temps fou à jouer. A titre d'exemple, le meilleur ratio qualité du coup / temps mis à jouer étaient pour 6 simulations avec une profondeur de 3 (16 secondes). Cependant, cela restait moins qualitatif que l'algorithme sans Simulations qui tournait plus rapidement pour une meilleure profondeur.

IV. Conclusion :

Finalement, nous avons choisi un algorithme qui effectue simultanément MinMax et AlphaBeta (MinMaxAlphaBeta) avec une liste de coups possibles calculée à partie des voisins des coups déjà joué. Nous avons optimisé les algorithmes autant que possibles en évitant un maximum d'opérations inutiles, même les plus élémentaires. Le meilleur compromis entre **qualité des coups** et **temps d'exécution** que nous avons trouvé s'effectue pour une profondeur de **4**. En général, les remarques que nous avons faites sont :

- Les premiers coups sont très rapides (1 seconde en moyenne)
- Si l'adversaire est bon, l'IA joue bien et assez rapidement (4 secondes en moyenne)
- Si l'adversaire est mauvais, l'IA met un peu plus de temps à jouer car elle est optimisée pour les meilleurs coups (7 secondes en moyenne)
- Si l'IA possède trop d'options, que ce soit de victoire ou de contre, elle met beaucoup plus de temps (9 secondes en moyenne avec des anomalies dont le temps le plus long enregistré qui est de 32 secondes)