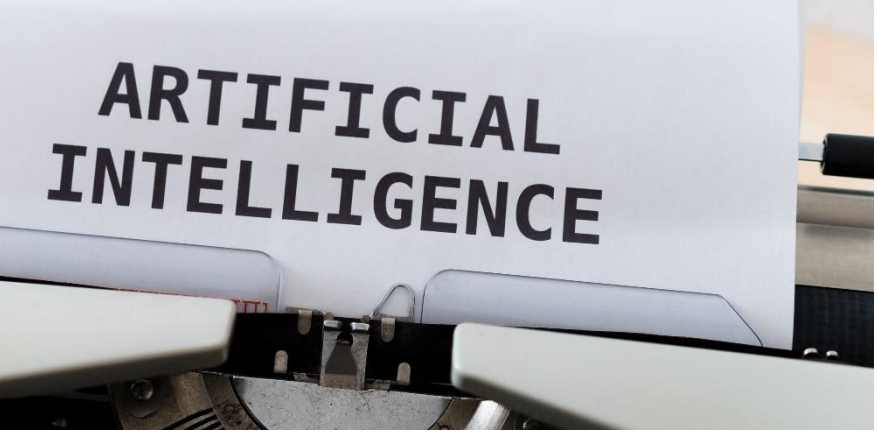

ESILV
ENGINEERING SCHOOL
DE VINCI PARIS

RAPPORT : IA BATTLE

« CHARBONIA »

2021

Data & Intelligence Artificielle

GABIS Asma — **PROFESSEUR TD**

RESPONSABLE MODULE

RODRIGUES Christophe

CHENDEB Safwan

AUTEURS

BAUMANN Lewis

AHMED Achirafi

BEN MAAMAR Mehdi

KAUFMAN Quentin

TD N

MAI 2021 — PROMO 2023

Table des matières

I Mise en contexte

Pour ce 6ème semestre à l'ESILV nous devions développer une intelligence artificielle capable de jouer au jeu du morpion. Ce projet fait partie du cours de Datascience et IA. Pour cela, nous devions nous baser sur l'algorithme du MinMax vu en TD durant l'année. Il était conseillé d'utiliser un élagage Alpha-Beta. Les dictionnaires de coups sont interdits. L'évaluation finale se présente sous forme d'un tournoi, où des IA jouent les unes contre les autres. Il y aura un tournoi final qui opposera les meilleures IA de chaque TD. En vérité, il ne s'agit pas du morpion classique, mais de l'une de ses variantes avec une matrice carrée de taille 12, avec une contrainte de 4 items à aligner. Ces deux derniers jeux sont des cas particuliers du jeu du m-n-k (généralisation du morpion).

II Choix du langage

Le premier choix fort que nous avons fait pour notre IA, que l'on nomme CharbonIA, est le langage de programmation. Nous avons préféré développer notre IA en C#. Le but est de gagner des affrontements contre d'autres IA. Beaucoup de points sont accordés à la vitesse de réponse de l'IA. Cela peut même devenir disqualifiant dans certains cas (réponse en plus de 10 sec). Une IA qui fait plus de calculs à la seconde a donc logiquement plus de chances de gagner. Le C# est un langage plus bas niveau que le Python, cela signifie qu'il est plus proche de la machine et qu'il effectue les calculs plus rapidement. En revanche, cela signifie généralement que le code est plus complexe à développer. Mais dans le cas présent, le traitement des matrices n'est pas beaucoup plus compliqué sur C# que sur Python. C'est donc avec le C# que nous avons développé CharbonIA.

III Le cœur de l'intelligence

Vous l'aurez probablement deviné, CharbonIA travaille uniquement avec des matrices représentant le plateau de jeu. C'est le format le plus intuitif et optimisé lorsqu'il s'agit de travailler sur un algorithme de morpion. Nous avons développé plusieurs fonctions. Certaines d'entre elles sont peu intéressantes (fonctions d'affichage etc.), et leur fonctionnement est plutôt intuitif. C'est pourquoi on ne s'attardera pas sur celles-ci.

L'algorithme du MinMax compare des « branches » pour sélectionner celle qui donnera le meilleur résultat final. Chaque branche représente une instance de jeu, ou autrement dit, elles représentent les différentes façons possibles dont pourraient se dérouler la partie. Ce sont des futurs différents, c'est à nous de voir lequel nous fait gagner. On connait tous la phrase « il avait un coup d'avance ! », c'est une expression que l'on utilise régulièrement. C'est amusant, car elle décrit parfaitement le fonctionnement de l'algorithme. On regarde à un certain nombre de coups à l'avance dans l'espoir de trouver un coup victorieux (ou du moins qui nous avantagera). Voir dans le futur c'est bien beau, mais dans ce cas cela signifierait que le premier qui joue a gagné (car il aura littéralement un coup d'avance). En réalité, le nombre de calculs à effectuer est si élevé qu'il nous est impossible de parcourir toutes les branches

jusqu'à leur fin. Dans un temps raisonnable, on ne peut donc voir que quelques coups à l'avance. Eh oui, voir dans le futur coûte cher en ressources !

Pour en revenir à notre problème de morpion, chaque nœud d'une branche correspond à une matrice. Dans l'algorithme du MinMax, on attribue à chaque matrice une valeur pour savoir si elle nous avantage ou non. Le seul problème est : Comment déterminer cette valeur ?

Cela nous amène donc à la fonction de fitness.

A. Fitness

Dans notre code, la fonction de fitness s'appelle CheckAround. Cette fonction est censée attribuer une valeur numérique à une matrice selon l'avantage que cette dernière donne. Cela semble simple, mais c'est en réalité plus compliqué qu'il n'y parait. On pourrait commencer simplement et se dire que si on voit 2 jetons alignés on renvoie 1, et 0 sinon. On peut aussi renvoyer -1 si c'était le tour de l'adversaire. L'idée est de maximiser notre gain et de minimiser celui de l'adversaire. On va donc de manière alternative, prendre le Min puis le Max des nœuds des branches. C'est ce que nous avons fait au début, mais le temps de calcul était très élevé pour un résultat approximatif. Après réflexion, nous avons décidé de rendre le score plus « continu ». Plutôt que de n'attribuer que 3 valeurs différentes, nous pourrions prendre en compte le nombre de jetons alignés. Plus on aligne de jetons, plus on est proche de la victoire (ou de la défaite si c'est le tour de l'adversaire). On peut donc donner une valeur plus élevée à XXX qu'a XX. Nous avons donc implémenté cela. Les résultats étaient meilleurs, mais on ne gagne pas encore régulièrement. En effet si notre ligne ressemble à cela :

XX0X

On voit que le jeton adverse au milieu nous empêche de terminer la ligne. Le jeton adverse ne partira jamais, nous ne pourrons donc jamais terminer la ligne. Cela nous amène à une autre amélioration de l'heuristique : prendre en compte les blocages adverses en soustrayant des points.

Une dernière amélioration possible est de prendre en compte la distance entre deux jetons. Il est plus simple de bloquer des jetons séparés par un espace que 2 jetons collés l'un à côté de l'autre respectivement : XO X ou OXXO remarquez que dans le premier cas, 1 jeton suffit de bloquer le coup alors que dans le deuxième il en faut 2. On peut donc parler d'une notion de distance entre des jetons. Plus un jeton est éloigné d'un autre jeton, moins il a de points. Plus il est proche, plus il a de points.

Il est intéressant de voir comment nous sommes partis d'une heuristique plutôt incomplète, et que nous l'avons améliorée. Dans sa version actuelle, l'heuristique cherche à jouer le meilleur coup. C'est-à-dire un coup qui nous avantage en désavantageant l'autre.

Notre fonction fitness parcourt toutes les cases de notre matrice de jeu. Pour chaque case, elle vérifie les conditions que nous avons implémentés ci-dessus. Elle est donc efficace mais lourde.

B. Optimisations

Nous venons de faire une belle heuristique qui peut déjà retourner de bons résultats. Mais c'est encore grandement améliorable !

1. Table de transposition

Comme nous l'avons dit plus tôt, la quantité de calculs est un enjeu clé dans ce projet. Notre fonction fitness est performante mais lourde, chaque calcul de valeur de matrice est coûteux en temps et en ressource. D'autant plus qu'entre les branches, on retrouve plusieurs fois les même coups joués (ils ne sont juste pas dans la même branche). Néanmoins, on les recalcule. Il serait pratique de pouvoir stocker en mémoire la valeur des coups déjà calculés ! C'est ce que nous avons fait. Dans notre fonction MinMax, à chaque calcul de coup, nous stockons ce dernier dans un dictionnaire. De sorte qu'à chaque itération, on vérifie si la valeur du coup a déjà été calculée. Si ce n'est pas le cas, on le calcule grâce à la fitness puis on le stocke dans le dictionnaire. Le dictionnaire permet de stocker des éléments identifiés par des clés. C'est un outil très puissant qui peut retrouver très rapidement des éléments car en utilisant le principe du hashage. Une question reste en suspens : Comment créer une clé unique qui correspond à une matrice ?

La méthode est simple, on prend chaque élément de la matrice dans l'ordre et on les met dans un string. Nous avons donc étudié le meilleur type à définir pour les clés du dictionnaire. Au début, nous pensions que le int était le type le plus efficace à traiter pour la machine. Cependant, il y a 144 situations de jeux différentes. Le choix d'un int aurait donc impliqué d'avoir des nombres avec 144 chiffres (12 *12 le nombre de cases du morpion). Nous avons donc tenté d'utiliser une librairie permettant de manipuler une classe BigInt, équivalente au int classique du C#, mais permettant d'augmenter drastiquement le nombre de valeurs.

Ceci étant, nous avons finalement décidé de laisser de côté cette dernière approche, car le traitement des informations était beaucoup trop lourd. Nous sommes finalement restés sur l'idée d'utiliser des strings avec 144 caractères comme des clés pour chercher les matrices dans un dictionnaire.

2. CheckBad

Le rôle de CheckBad est de régir le comportement de l'IA lorsqu'elle est sûre de perdre. Grâce à cette fonction, elle joue le tout pour le tout à ce moment. L'objectif est d'attaquer en jouant le meilleur coup possible dans les pires situations. De cette façon, on profite de la moindre erreur de l'IA adverse lorsqu'elle est sur le point de gagner.

Le code de CheckBad se construit de la même manière que CheckAround. Nous avons exactement la même structure mais avec des paramètres différents. Ces derniers ont été ajustés de manière à privilégier les coups offensifs en cas de situation difficile pour CharbonIA.

La fonction est appelée à chaque niveau de profondeur, mais sa valeur est uniquement utilisée lorsque le score indique une défaite. À première vue, il semble que CheckBad ralentisse le programme en alourdissant les calculs. Pour pallier ce problème, on utilise la classe Thread de C# qui nous permet d'effectuer la tâche de calculer le moins pire coup en parallèle des autres calculs. C'est ce que l'on appelle le multi-threading

3. Elagage Alpha-Beta

Un élagage Alpha-Beta a été implémenté dans l'algorithme du MinMax. Celui-ci a été étudié en TD. Nous ajoutons deux paramètres alpha et beta au MinMax. Le paramètre alpha prend en premier la valeur int minimale que C# peut donner, et beta la valeur maximale. L'objectif de cet élagage est de diminuer le nombre de branches de l'arbre du MinMax à parcourir. Ainsi, on vérifie à chaque branche que alpha soit supérieur ou égal à beta, alpha prenant à chaque fois des valeurs supérieures à lui. Si alpha est supérieur ou égal à beta, on peut sortir de la boucle, car cela signifie que l'on aura trouvé une valeur supérieure à l'ancienne valeur maximale.

4. Autres améliorations

Il y a aussi quelques améliorations qui sont très simples mais permettent de gagner beaucoup de temps de calcul. Une optimisation simple est de faire un tri des meilleurs résultats de la première génération. Souvent, une branche qui commence par un bon coup à de meilleures chances d'être une bonne branche et de mener à une victoire. Ce simple tri a pour effet d'orienter les calculs des futures générations vers des branches qui ont déjà un résultat satisfaisant. À cela, il est possible d'ajouter une limite de temps au-delà de laquelle on arrête les calculs. Ce dernier point sera discuté dans la partie suivante.

Une autre amélioration qui nous a permis de grandement réduire le temps de calcul, est la méthode : EstAMoinsDe2Cases. Comme son nom l'indique, elle vérifie que le coup qu'on veut jouer se trouve à moins de 2 cases d'un autre coup joué (allié ou adverse). Cela se base sur une règle qu'on pourrait formuler comme : « On joue là où il y a du jeu ». De cette façon, on diminue le nombre de calculs pour retrouver ou calculer la valeur d'un coup. Sans cette optimisation, on doit générer au plus 144 matrices différentes à chaque génération. Avec cela on réduit les calculs à une dizaine de matrices seulement.

5. Création d'un timer

L'une de nos optimisations consiste à utiliser un timer qui limite le temps de calcul dans la fonction MinMax. En effet, les contraintes du projet nous forcent à avoir une IA qui puisse non seulement gagner, mais qui joue aussi le plus rapidement possible. Or, il n'est pas possible d'optimiser au maximum les prédictions du MinMax en jouant en moins d'une demi-seconde. Ainsi, nous avons décidé de placer un timer qui force l'algorithme à retourner le meilleur coup trouvé au bout d'une certaine limite de temps.

Les consignes que nous avons retenues nous impose de ne pas dépasser la barre des 10 secondes, auquel cas cela serait considéré comme une défaite pour l'IA. Nous avons donc effectué plusieurs tests pour trouver la meilleure limite de temps. Celle-ci doit être la plus petite possible, tout en permettant à l'IA de trouver les meilleurs coups possibles. Après plusieurs essais, nous avons finalement définit la limite de temps à 3 secondes. Ainsi, tant que le temps de calcul est inférieur à 3 secondes, le MinMax continue de chercher le meilleur coup. Et si la limite de temps est atteinte alors que l'algorithme n'a pas eu le temps de trouver le meilleur coup possible, le meilleur coup trouvé est retourné.

IV MinMax (Association du score pour chaque coup)

A. Association du score à une matrice

Comme dit plus tôt, l'algorithme du MinMax compare des branches pour en tirer celle qui donnera le meilleur résultat final. Pour cela, il nous faut associer des scores pour chaque branche (ici chaque matrice) afin que CharbonIA puisse choisir le meilleur coup à faire. Nous trions les meilleurs éléments.

On peut noter que 10000000 et -10000000 sont des scores de victoire ou de défaite. Si l'algorithme ne détecte ni l'une ni l'autre avec Win condition, alors on calcule les scores associés aux différentes possibilités via CheckAround. Celui-ci associera à chaque matrice possible un score entre 10000000 et -10000000.

B. Choix de la profondeur

En ce qui concerne le choix de la profondeur, celle-ci a évoluée au fur et à mesure de la conception de notre IA. Au début du projet, CharbonIA pouvait mettre beaucoup de temps à jouer en regardant à 3 coups à l'avance. Mais après amélioration de notre heuristique, ainsi que des entraînements contre des adversaires humains ou IA, nous avons pu nous permettre de regarder plus de coups à l'avance en moins de temps (avant 1 coup était déjà un calcul élevé). Nous avons donc maximisé le nombre de coups d'avance qu'elle pouvait voir en moins de 10 secondes (délai imposé par l'énoncé).

Une dernière amélioration a été de moduler le niveau de profondeur du MinMax. Nous avions retenu un niveau de profondeur 4. L'IA adoptera une tactique différente en fonction de celui qui commence. Si l'IA commence, elle jouera a un niveau de profondeur 4 pendant les 2 premiers coups, avant de clôturer la partie en profondeur 2. Si l'adversaire commence, l'IA jouera directement en profondeur 2 jusqu'à la fin de la partie. Le niveau de profondeur 4 nous permet de jouer des coups qui nous assurent la victoire, tandis que le niveau 2 a des performances légèrement moins bonnes mais permet d'avoir des résultats plus rapidement. Ainsi, ces combinaisons nous assurent la victoire lorsque l'IA joue en premier, et nous permettent d'obtenir le point de rapidité dans le cas contraire.

V Conclusion et Critiques

Finalement, nous avons conçu une IA qui arrive à réagir à un maximum de situations. En effet, nous avons trouvé après un grand nombre de tests, et de quelques affrontements face à d'autres IA, qu'on ne peut pas gagner face à CharbonIA dès qu'elle commence à jouer. De plus, elle semble aussi bien réagir lorsque l'IA adverse commence à jouer.

Il existe une stratégie de jeu à appliquer lorsque l'on joue en premier. Elle permet de gagner à coup sûr quelle que soit la réponse adverse. Ainsi, si le premier joueur s'oriente vers la bonne tactique, le second ne peut pas gagner. L'heuristique que nous avons développé a permis à l'IA d'orienter ses actions vers la tactique citée précédemment. Nous ne percevons pas bien la raison de ce comportement. Seule l'expérience et les essais nous ont permis d'arriver à ces résultats.