Maxime BLANCHARD

Lingyi CHENG

Louis ROLLAND

Projet IA Teeko :

Professeur : Fabrice LAURI IA41 : Automne 2022

Table des matières

I – Sujet Teeko …………………………………………………………………………………………….. 1

1 – Choix du sujet ………………………………………………………………………………. 1

2 – Analyse générale du problème …………………………………………………….. 2

3 – Application en python ………..………………………………………………………… 2

II – Fonction éval / machines à états ……………………………………………………………. 4

1 – Description hiérarchie ………………………………………………………………….. 4

2 – Justification des choix …………………………………………………………………… 5

III – Problèmes rencontrés ……………………………………………………………………………. 6

1 – Fonction eval ..………………………………………………………………………………. 6

2 – Choix de la meilleure situation ……………………………………………………… 7

3 – Niveaux de difficulté ……………………………………………………………………… 7

4 – Réalisation UI ………………………………………………………………………………… 8

IV – Situations et analyse par le programme …………………………………………………. 9

1 – Situation 1 ……………………………………………………………………………………… 9

2 – Situation 2 ……………………………………………………………………………………. 10

V – Pistes d’améliorations ……………………………………………………………………………. 11

VI – Conclusion …………………………………………………………………………………………….. 12

---------------------------------------------------------------------------------------------------------

Annexe 1 : Résultats obtenus par le programme en jeu

Annexe 2 : next\_states et next\_states\_beginning

Annexe 3 : Implémentation de MinMax avec élagage alpha-bêta

I – Sujet Teeko :

Choix du Sujet -

Etant tous débutants en intelligence artificielle, nous avons d’abord décidé de choisir un sujet de difficulté novice, pour nous permettre d’avoir le temps de proposer un rendu à la hauteur de nos ambitions.

Le sujet Teeko nous a semblé être intéressant à traiter car il nous permettait d’implémenter l’algorithme MinMax avec élagage alpha-bêta, procédé abstrait que nous n’avions pas tout à fait saisi lors de sa présentation en cours. Néanmoins, cet algorithme avait piqué notre attention, et c’est d’une part pour cela que nous avons choisi ce sujet.

D’autre part, le Teeko nous a tout de suite paru être un jeu assez amusant, rappelant à la fois le « Puissance 4 » auquel nombre d’entre nous avions déjà joué, et créant une suite hypothétique à ce jeu où l’égalité n’arrive que trop souvent.

Cela paraît assez fou de se dire qu’il est possible de programmer une intelligence artificielle sur un jeu aussi stratégique que celui-ci, ce qui n’a fait qu’attiser notre intérêt pour ce sujet.

C’est pourquoi nous avons fait le choix d’interpréter ce sujet comme la création d’une intelligence artificielle capable de jouer une partie de Teeko contre un être humain, et supposément, de gagner la quasi-intégralité de ses matchs.

Analyse générale du problème –

Après avoir lu plusieurs fois le descriptif du sujet, nous avons déterminé la manière optimale de procéder pour venir à bout du projet.

Tout d’abord, nous allions créer une fonction qui prendrait un état de jeu en paramètres et qui permettrait de renvoyer l’ensemble des états fils de cet état. Le Teeko étant un jeu comportant deux phases (la première où chaque joueur pose tour à tour un jeton sur la grille, et la deuxième où chaque joueur déplace l’un après l’autre un de ses jetons), il faudra donc distinguer les deux cas dans notre fonction, que l’on a appelé next\_states.

Ensuite, il faudrait être capable de choisir le meilleur état fils pour une situation donnée. Nous avons donc créé une fonction d’évaluation qui attribue une « note » à chaque situation pour que le programme puisse choisir un unique coup à jouer, supposément le meilleur.

Finalement, il serait préférable que le programme trouve la meilleure situation rapidement et avec une certaine profondeur : on souhaiterait que l’IA décide du coup à jouer en examinant les conséquences de ses choix sur plusieurs tours à l’avance.

Application en python –

Nous avons d’abord considéré qu’un état serait caractérisé par un tuple composé du joueur auquel c’est le tour de jouer, ainsi que la grille de jeu sous la forme d’une matrice à deux dimensions de taille 5x5.

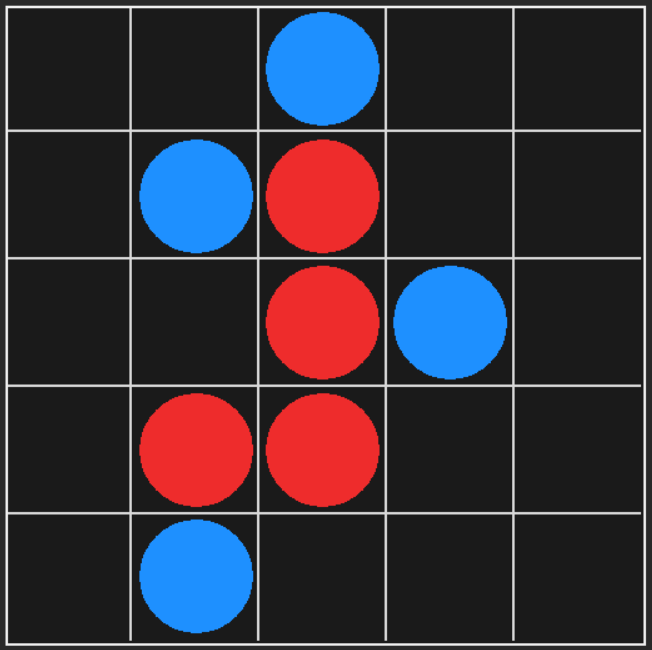
Ensuite nous avons créé la fonction next\_states qui renverra l’ensemble des plateaux de jeu où le joueur auquel c’est le tour de jouer pose un nouveau jeton, ou bien déplace un de ses jetons là où il le peut, selon la phase de jeu. Le résultat est sous forme d’une liste d’états.

Voir les fonctions python de manière concrète dans l’Annexe 2.

Pour attribuer une note à un état, nous avons ensuite construit une fonction qui prend en paramètre un état, et qui renvoie un score qu’elle attribue à cet état en fonction de différents paramètres arbitrairement choisis :

* Le nombre de jetons sur la ligne verticale la plus complète
* Le nombre de jetons sur la diagonale la plus complète
* Le nombre de pièces dans le « carré » le plus complet
* Le nombre de jetons dans la ligne horizontale la plus complète
* La distance moyenne des jetons par rapport au centre du plateau

Par exemple, dans le cas ci-dessous, la ligne horizontale la plus complète pour le joueur rouge (encadrée en jaune) contient 2 jetons, et la ligne verticale la plus complète (en vert) comprend 3 jetons.



Compter les jetons sur les lignes, diagonales et carrés permet de savoir à quel point les deux joueurs sont proches d’atteindre une position victorieuse.

Cette fonction prend le score associé aux jetons posés par l’IA, et soustrait le score correspondant aux jetons du joueur à ce dernier.

Ainsi, dans le cas où le joueur est en position avantageuse, le score renvoyé sera négatif. Dans le cas contraire, le score associé à la position de jeu sera positif.

Une image contenant texte

Description générée automatiquement

*Score négatif car le joueur (-1 représenté par des -) a l’avantage*

Ensuite, nous avons fait appel à l’algorithme MinMax accompagné de l’élagage alpha-bêta vu en cours (voir Annexe 3 pour notre implémentation en Python).

Le rôle de cet algorithme est de parcourir le plus efficacement possible l’arbre des états fils d’une situation de jeu, dans le but de trouver la situation la plus avantageuse pour l’IA en vérifiant plusieurs tours à l’avance les conséquences d’un coup.

II – Fonction éval :

Description hiérarchie -

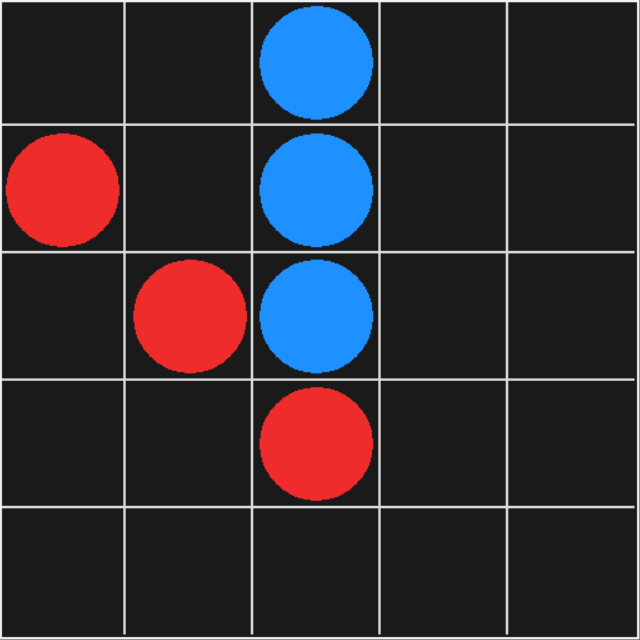
Nous avons décidé que les différents paramètres de la fonction d’évaluation seraient coefficientés de la manière suivante :

* Le nombre de jetons alignés à l’horizontale, la verticale et en carré ont un poids de 1
* Le nombre de jetons alignés en diagonale ont un poids de 2
* La distance moyenne des jetons par rapport au centre donne un malus de poids 2

Mathématiquement, cela donne :

Score = 1 x (nombre max de jetons à l’horizontale)^2 + 1 x (nombre max de jetons à la verticale)^2 + 1 x (nombre max de jetons formant un carré)^2 + 2 x (nombre max de jetons alignés en diagonale)^2 – 2 x distance moyenne des jetons par rapport au centre

(Le nombre maximum de jetons alignés dans chaque position est mis au carré pour réellement valoriser un alignement de 3 à celui de 2, et de 2 à 1)



Ainsi, sur la grille de jeu suivante, le score attribué au joueur bleu sera le suivant : Score = 1 x 1 + 1 x 3 + 1 x 1 + 2 x 1 – 2 x ((0 + 1 + 2) / 3) = 11

Le score total attribué à la situation sera égal au score attribué à la position des jetons du joueur bleu auquel on soustraira celui attribué à la position des jetons du joueur rouge.

Dans la situation illustrée ci-dessus, le score total sera donc calculé de la manière suivante :

Score = Score\_bleu – Score\_rouge

* Score = 11 - (1 x 1 + 1 x 1 + 1 x 1 + 2 x 9 – 2 x ((1 + 1 + ) / 3)

= 11 – (21 – 2.82)

= -7.18

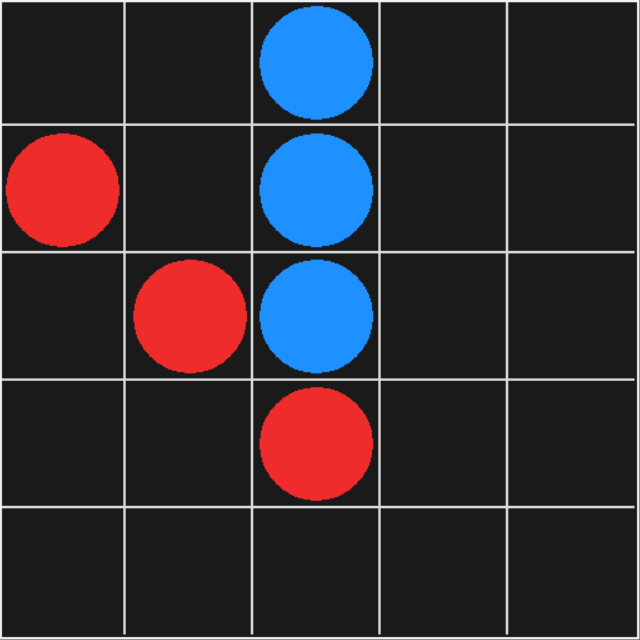
Justification des choix -

Nous avons arbitrairement choisi les coefficients évoqués précédemment en fonction de ce que nous avons pu tirer des quelques parties jouées entre les membres de l’équipe.

Pour les paramètres de la fonction déterminés en premier (nombre de jetons alignés à l’horizontale, à la verticale, en position carré), nous avons pensé qu’un poids de 1 serait une bonne référence pour être à même de choisir les poids des autres paramètres de la fonction.

Par exemple, le nombrer de jetons alignés en diagonale. Nous avons jugé bon de favoriser ce paramètre car nous estimons (après examen de certaines parties) qu’une position où le joueur possède plusieurs jetons en diagonale est plus avantageuse qu’une autre.

La distance moyenne des jetons par rapport au centre de la grille nous semblait également un élément de jeu important à prendre en compte pour pouvoir gagner une partie : plus les jetons d’un joueur étaient regroupés vers le centre, plus ce dernier tendait à avoir l’avantage.



Ainsi, dans la situation montrée précédemment, on a pu voir que la situation du joueur rouge était considérée comme meilleure que celle du joueur bleu (car le score total obtenu du point de vue du jouer bleu est négatif). Cela vient en majorité du fait qu’un alignement de plusieurs jetons en diagonale est considéré plus bénéfique qu’un autre alignement du même nombre de jetons.

III – Problèmes rencontrés :

Fonction eval -

Beaucoup de problèmes ont été rencontrés lors de la création et de la réalisation en python de cette fonction.

En effet, cette fonction est à la clé de la réussite de notre programme : il fallait donc trouver les paramètres optimaux pour cette dernière.

Très vite, nous avons pensé à 4 premiers paramètres :

* Si 4 jetons étaient alignés en :
  + Diagonal
  + Horizontal
  + Vertical
  + carré

que nous avons ensuite affinés en « nombre de jetons alignés » dans les mêmes dispositions pour chaque joueur, de manière à obtenir des scores plus diversifiés et adaptés à toutes les situations de jeu.

Pour programmer ceci en python, de nombreuses difficultés ont été rencontrées, mais le temps et la détermination nous ont permis d’en venir à bout.

Cependant, nous nous sommes vite rendu compte de l’insuffisance de ces simples paramètres. Nous avons donc commencé à réfléchir à quels paramètres pourraient avoir du sens sur un plateau de Teeko.

C’est alors que nous est venu l’idée de prendre en compte la distance moyenne des jetons par rapport au centre : en effet, plus les jetons sont proches du centre de plateau, plus la chance d’en aligner 4 devient grande.

Choix de la meilleure situation -

Cela reste un des problèmes qui nous aura pris le plus de temps à résoudre tout au long de la réalisation de ce projet.

En effet, malgré la fonctionnalité de score attribué à chaque situation et une implémentation assez fidèle à l’algorithme MinMax d’origine, l’intelligence artificielle choisissait de jouer certains coups qu’un être humain moyen pourrait qualifier de stupide.

Après quelques recherches, nous avons pu déterminer que le problème venait en fait de notre implémentation de l’algorithme minmax, qui ne considérait pas les états finaux comme des « feuilles » dans l’arbre de recherche, et continuait à rechercher les positions suivantes. Ainsi, l’IA ignorait les états victorieux, et choisissait en général de continuer à jouer même lorsqu’il lui suffisait de déplacer un jeton pour gagner.

Un autre problème auquel nous avons dû remédier est que la fonction d’évaluation renvoyait le même score pour tous les états victorieux. Ainsi, il arrivait que l’IA choisisse un coup qui lui permette potentiellement de gagner dans plusieurs tours, plutôt qu’un coup qui lui assurerait une victoire immédiate. Nous avons résolu ce problème en ajoutant au score des états finaux une valeur qui diminue avec la profondeur, pour que l’IA favorise les victoires les plus proches.

Niveaux de difficulté -

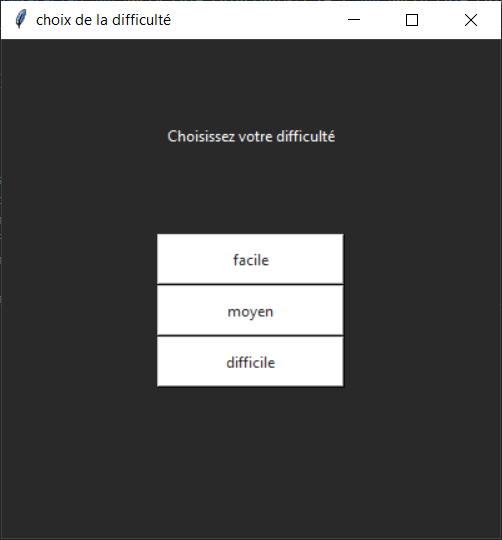
La nature de ce problème n’est pas réellement d’ordre technique, mais plus intellectuel. Lorsque nous nous sommes rendu compte de la nécessité de créer plusieurs niveaux de difficulté, nous n’étions pas fortement inspirés quant à la manière de procéder.

Nous avons donc décidé de simplement ajouter un coefficient dans le calcul du score des situations, et de changer la profondeur utilisée avec l’algorithme MinMax en fonction du niveau de difficulté choisi. Par exemple, le niveau facile impose une profondeur de 1, ce qui fait que le programme ne prendra pas compte des conséquences de son coup.

Réalisation UI -

Autant plusieurs d’entre nous étaient déjà assez familiers avec Python, aucun n’avait eu l’occasion de travailler sur une interface utilisateur dans ce langage auparavant. Il a donc fallu apprendre le fonctionnement de la librairie Tkinter assez rapidement, puis faire la connexion entre le programme de l’IA déjà faite et l’affichage, ce qui nous a posé quelques soucis.

Ensuite, il a fallu mettre en place les différents niveaux de difficulté et la fenêtre qui va avec :



Puis réaliser le lien entre la difficulté choisie par l’utilisateur et le changement de comportement de l’IA en jeu.

Nous voulions que lorsqu’un bouton soit pressé, 2 actions s’opèrent :

* La difficulté du jeu soit changée
* La fenêtre soit fermée et laisse place au jeu

Cependant, cela n’est pas possible avec les boutons tkinter. Nous avons donc fait en sorte que l’appui sur un bouton appelle une fonction qui réalise ces 2 actions l’une après l’autre :

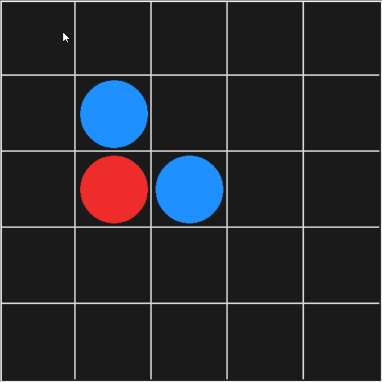


Une image contenant texte

Description générée automatiquement

IV – Situations et analyse par le programme :

Situation 1 –



Dans cette situation, c’est à l’IA (en rouge) de jouer. Il s’agit là d’une situation compliquée à mettre en place, car il faut faire comprendre à l’IA que la situation la plus avantageuse n’est pas celle avec le plus haut score sur le coup d’après.

En effet, aligner 2 jetons rouges dans cette situation signerait la fin de la partie, car le joueur bleu pourrait alors compléter une diagonale de 3 jetons. C’est alors que la notion de profondeur dans l’algorithme MinMax intervient.

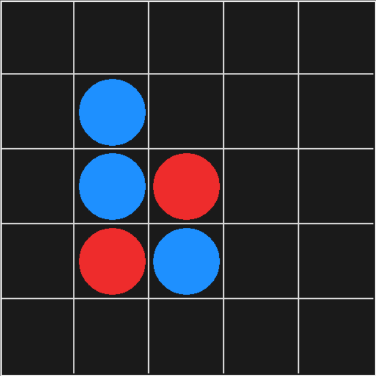


*Notion de profondeur dans le code Python de l’algorithme MinMax*

Nous avons fait en sorte que le programme ne se limite pas à faire un choix pour le coup suivant, mais également à prendre en compte les conséquences que pourraient avoir ce coup dans le temps. Ici, nous évaluerons une situation jusqu’à 4 coups à l’avance.

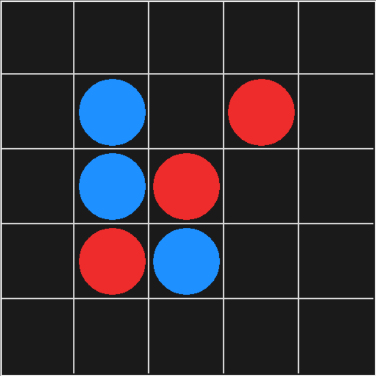
Dans ce projet, nous avons appliqué la notion de profondeur de manière récursive dans la fonction MinMax (le cas d’arrêt étant lorsque la profondeur atteint 0).

Situation 2 –



Pour un humain, le choix à faire paraît évident. Il faut que le joueur rouge (ou l’IA) joue de manière à se réserver 2 possibilités de victoire. Ainsi, le joueur bleu ne pourra en contrer qu’une et perdra inévitablement la partie.

*Le coup en question :*



En effet, jouer ce coup offre deux possibilités de victoires différentes. De plus, il aboutit forcément à la défaite du joueur adverse, c’est donc le meilleur coup à jouer.

Nous ne pouvons que supposer le raisonnement de l’IA dans une situation pareille, mais nous pensons que similairement à la situation précédente, la profondeur dans l’algorithme MinMax fait que l’IA prend conscience de la supériorité de ce coup par rapport aux autres sur le long terme, ce qui fera qu’elle choisira ce coup là après examen de l’ensemble des coups qu’il est possible de jouer.

V – Pistes d’améliorations :

Malgré le temps passé à réaliser ce projet, nous pensons qu’il est encore sujet à critique, et pourrait être amélioré sur de nombreux points.

Par exemple, l’UI n’est pas aussi belle que nous l’aurions voulu. Ou encore, le code n’est pas assez optimisé ce qui fait que nous devons nous limiter à une certaine profondeur dans l’algorithme MinMax pour que l’IA soit capable de choisir un coup à jouer en un temps raisonnable. Après coup, nous pensons que nous aurions dû utiliser le multithreading ou le multiprocessing pour réduire le temps de calcul.

Un autre problème avec notre application est que l’interface s’immobilise lorsque l’algorithme MinMax parcourt les états fils. Cela pourrait être résolu en effectuant le parcours d’arbre sur un thread différent de celui qui gère l’interface, bien que cela semble compliqué en utilisant la librairie Tkinter pour l’interface.

De plus, nous aurions apprécié développer des niveaux de difficulté distincts plutôt que simplement changer les coefficients des paramètres de la fonction d’évaluation et la profondeur de recherche de l’algo MinMax (pour obtenir des intelligences artificielles moins malines). Ce qui fait une amélioration envisageable à apporter à notre programme.

Enfin, l’IA n’est pas imbattable comme nous l’aurions souhaité au commencement du projet. Nous pensons que nous n’avons pas fourni assez de paramètres à la fonction d’évaluation pour lui permettre d’évaluer une situation de manière optimale.

Alors, une amélioration que nous pourrions apporter serait de rajouter certains paramètres qui pourraient être utiles à l’évaluation d’un état (par exemple la distance entre les jetons, le nombre d’étapes avant une victoire possible, …).

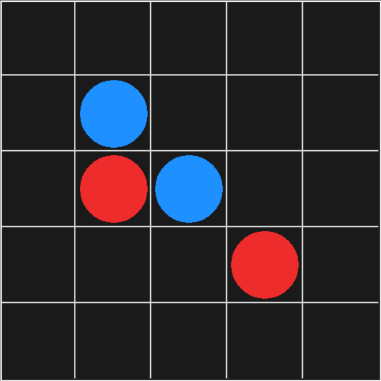
VI – Conclusion :

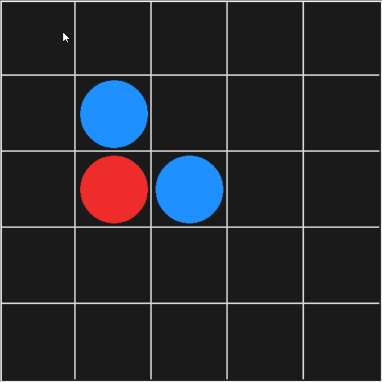
Réaliser une intelligence artificielle capable de trouver le meilleur coup à jouer dans un jeu stratégique semblait irréalisable en début de semestre, et pourtant le travail que nous avons réalisé se rapproche de cet objectif.

Je pense parler pour tout le groupe en disant que ce projet a été très enrichissant. Il nous a permis de mettre en pratique les notions théoriques vues en cours et, en tant que novices, de nous initier aux bases de l’intelligence artificielle.

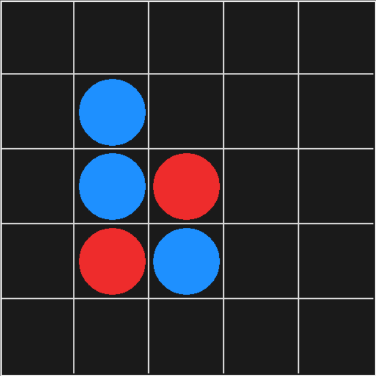
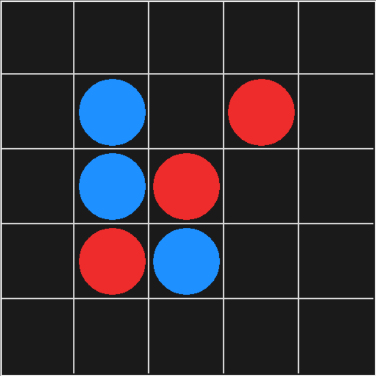
Malgré de multiples échecs et nombreuses pistes d’améliorations encore à explorer, nous restons fiers du travail accompli, et remercions les professeurs d’IA41 pour nous avoir permis de réaliser un tel projet.

Annexe 1 : Résultats obtenus par le programme en jeu



 -

Ceci est un coup remarquable de la part de l’IA, car un jeton posé par le joueur à cet endroit aurait inévitablement entraîné la victoire du joueur, 2 coups plus tard.



L’IA joue un coup qui lui permet de mettre le joueur en échec : non seulement elle aligne 3 jetons, mais elle les aligne de la façon la plus intelligente.

Annexe 2 : fonction de génération des états suivants

next\_states (1ère phase)

Une image contenant texte

Description générée automatiquement

next\_states (2ème phase)

Une image contenant texte, moniteur, argent, capture d’écran

Description générée automatiquement

Annexe 3 : implémentation de MinMax avec élagage alpha-beta

Une image contenant texte

Description générée automatiquement