|  |  |
| --- | --- |
| LAPERTOT Raphaël  LAFOURCADE Anthony  FAILLAUFAIX Thomas | PR-3001A  22/02/2019 |

Rapport  
Projet go-moku



# Introduction :

Ce projet a été réalisé par Raphaël LAPERTOT, Anthony LAFOURCADE et Thomas FAILLAUFAIX dans le cadre de l’unité PR-3001A « Résolution de problème et algorithme » de l’ESIEE Paris durant les mois de janvier et février 2019.

Le projet consiste à réaliser informatiquement un go-moku, ainsi qu’une intelligence artificielle (non-machine-learning) capable de jouer contre le joueur qui joue aussi.

*Mais enfin, qu’est-ce que le Go-moku ?*

Un go-moku est, pour faire court, un morpion où la grille n’est pas forcément 3 x 3, et le nombre de pions à aligner n’est pas forcément 3. Le joueur et l’ordinateur jouent tour par tour, l’un après l’autre, et le but est donc d’aligner un certain nombre de pions (*A*), diagonales comprises, dans une grille carrée (*N* x *N*).

Par exemple, voici une partie de go-moku, où la grille carrée est de dimension 4 x 4, et le nombre de pions à aligner est de 3. Ici, le joueur des croix (X) a gagné car il a aligné 3 pions.

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
| O | X | X | X |
|  | O |  |  |
| O | X |  |  |

Nous avons décidé de faire le projet en python pour avoir un haut niveau d’abstraction, en dépit des performances.

Notre programme est divisé en deux sous-fichiers :

* Le fichier fonction, qui comme son nom l’indique, regroupe toutes les fonctions du programme
* Le fichier play, qui permet de lancer le programme et de jouer contre l’ordinateur.

Nous avons mis à disposition toutes les versions de notre programme dans des dossiers pour permettre d’avoir une trace de l’avancement du code. Pour lancer le programme final, rendez-vous dans le dossier « Version finale » en ligne de commande, puis lancez « python play.py ».

# Fonction EvalPosition

Le but de la fonction **EvalPosition** est de retourner un score en fonction d’une situation donnée. Ce score varie entre -1000 et 1000, et donne une indication du joueur en tête. Un score positif indique que le joueur humain est en tête, tandis qu’un score négatif indique que l’ordinateur est en tête.

Principe de la fonction :

Le but de notre fonction est de construire une autre matrice des valeurs, où chaque case correspond à une case du plateau de jeu. Chaque case contient une liste de 4 valeurs, indiquant combien de pions de la même couleur que le pion de la case actuelle se trouvent en diagonale vers la gauche, au-dessus, en diagonale vers la droite, et à gauche, dans cet ordre. Sachant que si le pion n’a pas de pion de la même couleur, la case est initialisée à [1, 1, 1, 1].

Exemple :

|  |  |  |  |
| --- | --- | --- | --- |
| X | X | X | O |
| X | O | O | X |
| O |  |  | X |
|  | O | O |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 1 1  1 1 | 0 0  0 1 | 0 0  0 2 | 1 1  1 1 |
| 0 2  2 0 | 1 1  1 1 | 0 0  2 2 | 2 0  0 0 |
| 0 0  2 0 | 0 0  0 0 | 0 0  0 0 | 0 2  0 0 |
| 0 0  0 0 | 2 0  0 0 | 0 0  0 2 | 0 0  0 0 |

Création de la matrice :

Pour créer la matrice, nous avons fait deux boucles qui parcourent l’entièreté du plateau de jeu. Pour chaque case, nous regardons sa valeur (1 pour le joueur, -1 pour l’ordinateur, 0 pour une case vide), si c’est 0, nous pouvons passer à la case suivante, sinon, nous pouvons l’étudier plus. Il faut donc regarder la valeur des pions dans les 4 cases dont nous avons parlé précédemment, mais avant, il faut vérifier que nous ne sortons pas du plateau (cela peut être le cas si on chercher à regarder le pion au-dessus d’un pion sur la première ligne par exemple).

Une fois que l’on a trouvé une case à regarder, on regarde sa valeur, si c’est une valeur différente que celle du pion que l’on étudie, ou une case vide, on peut passer à la case suivante, par contre, si nous avons la même couleur, nous pouvons changer la valeur de la matrice des valeurs pour cette direction. Cette valeur sera simplement la valeur du pion adjacent dans cette direction incrémenté de 1.

Dans l’exemple ci-dessus, regardons la case en ligne 2 et colonne 3. En diagonale à gauche et au-dessus, nous avons des croix, donc nous mettons 0 pour les deux premières valeurs. Par contre, nous avons un rond en diagonale à droite, nous pouvons donc changer la troisième valeur. Pour cela, nous prenons la troisième valeur de ce rond, à savoir 1, et nous l’incrémentons de 1. Nous mettons donc 2 en troisième valeur. Il y a également un rond à gauche du pion étudié ; nous pouvons donc modifier la quatrième valeur. Ce pion a 1 pour quatrième valeur, donc nous mettrons 2 en quatrième valeur du pion étudié.

Calcul du score :

Pour pouvoir calculer le score, nous avons besoin de deux informations supplémentaires : le nombre maximal de pions alignés par le joueur et par l’ordinateur. Pour cela, nous utilisons deux variables contenant le maximum de pions alignés pour chaque joueur, ainsi que deux variables contenant les coordonnées de ces maximums, et enfin deux variables contenant les directions de ces maximums. Toutes ces variables sont initialisées à 0.

A chaque fois que nous changeons une valeur dans la matrice des valeurs, selon la valeur du pion étudié, nous comparons la valeur mise dans la matrice avec le maximum du joueur correspondant. Si la valeur est supérieure, elle devient le nouveau maximum, et on garde ses coordonnées et sa direction. Si la valeur est égale au maximum, on rajoute les coordonnées de la valeur aux coordonnées des autres maximums. On rajoute également la direction aux autres directions.

Cette première version est très simple. Premièrement, on échelonne les valeurs entre -1000 et +1000, avec un pas de 1000/*A*. Par exemple, si on doit aligner 3 pions, le scoring va prendre les valeurs entre -1000, -750, -500, -250, 0, 250, 500, 750 et 1000.

Ensuite, on détermine le score de chaque joueur en comptant le nombre maximum de pions qu’il a réussi à aligner (pour l’instant, on ne considère pas le blocage, mais on le traitera), puis on le multiplie par ce pas de 1000/A. Par exemple, dans cette position (4 pions à aligner) :

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
| O | X | X | X |
|  | O |  |  |
| O | X |  |  |

Le joueur O a maximum deux pions d’alignés : il a donc un score de 2\*(1000/A) = 500

Le joueur X, quant à lui, a 3 pions d’alignés : il a donc un score de 3\*(1000/A) = 750

Enfin, pour avoir un score global de la position, on soustrait les deux scores. Dans notre projet, nous avons choisi de soustraire le score du joueur X (l’ordinateur) au score du joueur O (l’humain). Dans notre exemple, le score serait donc de 500 – 750 = -250. L’ordinateur est considéré comme ayant un avantage (ce qui ne sera plus le cas après l’ajout de notre fonction blocage !).

# 1ère Version : Version basique :

Maintenant que nous avons une fonction **EvalPosition**, nous pouvons commencer à créer notre programme de jeu. Mais avant nous avons besoin de quelques fonctions basiques. Tout d’abord **InitPosition**, qui va simplement créer le plateau de jeu, dans notre cas, elle va simplement créer un tableau de taille NxN rempli de 0.

Il faut également la fonction **AffichePosition**, dont le nom est assez explicite. Celle-ci va simplement parcourir le tableau de jeu et en afficher le contenu. Afin de rendre l’affichage plus agréable, nous remplaçons les -1 par « X », les 1 par « O » et les 0 par «. ».

Nous faisons ensuite la fonction **PlusDeCasesLibres**, qui va renvoyer 1 si le plateau n’est pas vide et 0 sinon. Pour cela, elle parcourt le tableau de jeu, et dès qu’elle trouve un 0 (⬄ case vide), elle renvoie 1. Si elle n’en trouve pas, elle renvoie 0 et donc le jeu est fini.

Enfin, nous avons besoin d’une fonction **SaisirCoupJoueur** pour placer notre coup. Elle nous demande d’écrire d’abord en quelle ligne, puis en quelle colonne souhaitons-nous placer notre pion. Pour que cela soit plus agréable, nous avons décidé que la première ligne serait la ligne numéro 1 et pas 0, pareil pour les colonnes. La fonction va regarder si les coordonnées correspondent bien à une case vide, sinon, elle va redemander de nouvelles coordonnées au joueur, jusqu’à avoir des coordonnées correctes, puis va placer le pion (donc mettre 1 aux coordonnées dans le tableau de jeu).

Il manque encore quelques fonctions nécessaires au déroulement de la partie, mais nous les verrons un petit peu plus tard.

Comportement de l’ordinateur :

Pour mettre en place une partie, nous devons déterminer comment doit se comporter l’ordinateur. Nous avons décidé que l’ordinateur chercherait soit à compléter sa plus grande ligne, soit à bloquer son adversaire. Pour cela, il va appeler **EvalPosition**, et récupérer toutes les informations. Le coup à jouer varie en fonction du signe du score ; si le score est négatif ou nul, l’ordinateur est en tête ou il y a égalité, il va donc essayer de compléter ses lignes maximales, par contre si le score est positif, le joueur est en tête : l’ordinateur va donc essayer de bloquer le joueur.

Si l’ordinateur n’a pas réussi à jouer (tout ses coups sont bloqués), il va alors ensuite placer le pion aléatoirement sur le plateau.

Les coordonnées récupérées vont nous indiquer à coté de quel pion jouer, et la direction récupérée va nous indiquer dans quelle direction jouer. Pour que l’ordinateur place le pion, nous avons besoin de plusieurs fonctions permettant de placer le pion dans chaque direction et pour le placer aléatoirement.

# Fonctions de placement :

L’ordinateur a besoin de fonctions pour pouvoir placer son pion pour chaque direction. Nous allons donc en créer 4 différentes. Chacune va prendre comme paramètre le plateau *P*, la taille du plateau *N*, le nombre de pions déjà alignés *max\_value* ainsi que les coordonnées du point ayant ce maximum *coord\_max*. Elles vont retourner le plateau *P* ainsi qu’une variable place qui va valoir 1 si un pion a été placé, et 0 sinon.

Dans chaque cas, on va regarder de chaque côté des pions déjà alignés, et on va regarder si la case est vide. Si oui, on place le pion dans une case libre, et on retourne le plateau *P* avec le pion en plus ainsi que *place* = 1, indiquant qu’il a bien été placé. Sinon on retourne le même plateau *P* et *place* = 0.

Il y a également un cas particulier, quand la valeur maximale est égale à 1. Dans ce cas le pion est seul, on va donc créer une fonction **Placer1** qui essayer de mettre un pion à coté de celui-ci, en testant chacune des quatre fonctions précédentes jusqu’à réussir à en placer un.

Il faut également une fonction de placement aléatoire, mais avant cela, nous devons connaitre toutes les coordonnées libres. Pour cela, nous avons créé une fonction **CoordCasesVides** qui va simplement parcourir tout le plateau de jeu et ajouter toutes les coordonnées des cases vides dans une liste. La fonction de placement aléatoire **PlacerAleat** va d’abord récupérer les coordonnées des cases vides, puis placer un pion dans une de ces cases au hasard.

# Déroulement d’une partie :

On commence par initialiser le plateau, puis on définit une variable *end* qui prend la valeur de **PlusDeCasesLibres**. Tant que *end* est différend de 0, il reste de la place sur le plateau, donc on peut jouer. Le joueur joue avec un appel à **SaisirCoupJoueur**, puis le plateau est affiché grâce à **AffichePosition**. C’est ensuite au tour de l’ordinateur, il récupère les informations du plateau avec **EvalPosition**. On regarde avant si le joueur n’a pas gagné en comparant le nombre de pions maximal aligné et le nombre de pions nécessaires pour gagner. Puis il garde les informations importantes en fonction du score (maximum, coordonnées, et directions, du joueur ou de l’ordinateur). Puis, si le maximum est 1, on appelle **Placer1** sur chaque coordonnée. Si on réussit à placer un pion, c’est maintenant au tour du joueur, sinon, on place un pion aléatoirement. Si le maximum est différent de 1, on essaie de compléter la ligne pour chaque coordonnée. Pour cela, on appelle la fonction de placement correspondant à la direction du maximum. Si on réussit à placer un pion, c’est maintenant au tour du joueur, sinon, on place un pion aléatoirement. On refait appel à **EvalPosition** pour regarder si l’ordinateur a gagné.

# 2ème version : Ajout d’un Pierre-Feuille-Ciseaux

Dans la version précédente, le premier joueur à jouer était prédéfini et ne changeait jamais, mais nous avions envie que le premier joueur change soit différent d’une partie à une autre.

Nous avons décidé d’implémenter un jeu de pierre-feuille-ciseaux entre le joueur et l’ordinateur avant de commencer le go-moku. Le gagnant du pierre-feuille-ciseaux commencera la partie. Le comportement de l’ordinateur sera très simple : il fera simplement un coup au hasard.

Modifications du programme play :

Il faut donc évidemment appeler la fonction **PierreFeuilleCiseaux** (que nous détaillerons par la suite). Nous l’appellerons au niveau de l’initialisation. Nous créons également une nouvelle variable, *idjoueur*, qui indiquera le joueur en train de jouer. Elle prendra la valeur 1 quand le joueur joue, et la valeur -1 quand l’ordinateur joue. Cette valeur alternera à la fin de chaque coup de chaque joueur.

Nous avions également remarqué un comportement de l’ordinateur assez problématique. Prenons cette situation pour exemple où il faut aligner 3 pions pour gagner, et l’ordinateur joue les X :

|  |  |  |  |
| --- | --- | --- | --- |
| O | X | X | O |
| O |  |  |  |
|  |  |  |  |
|  |  |  |  |

Dans cette situation, on remarque que le nombre maximal de pions alignés par chaque joueur est de 2. Le score total est donc nul. Dans cette situation, l’ordinateur va donc essayer de compléter sa ligne de 2, mais on remarque qu’elle est bloquée des deux côtés. Il va donc placer son pion aléatoirement sur le plateau, plutôt que de bloquer son adversaire.

Nous avons fait une modification lors du tour de l’ordinateur. Celui-ci va appeler **EvalPosition** et récupérer les informations relatives aux deux joueurs. Par contre, au lieu de ne garder les informations que d’un seul joueur (celui en tête), il va concaténer les informations concernant les deux joueurs, en mettant en premier les informations relatives au joueur en tête.

Maintenant, l’ordinateur, au moment de placer son coup, s’il est en tête va tenter de compléter sa ligne, puis tenter de bloquer son adversaire, puis placer au hasard. S’il perd, il va tenter de bloquer son adversaire, puis de compléter sa ligne, puis de placer au hasard.

Fonction PierreFeuilleCiseaux :

Cette fonction va prendre pour paramètre le plateau de jeu *P*, et la taille de ce plateau *N*. Elle renvoie le tableau *P*, ainsi que le la valeur de *idjoueur*.

Elle va d’abord demander au joueur d’entrer son coup, puis l’ordinateur va choisir un coup au hasard. Elle va ensuite comparer ces deux valeurs, si elles sont égales, alors elle demande au joueur d’entrer un coup à nouveau, et l’ordinateur choisit un nouveau coup au hasard, puis on recompare. En cas de nouvelle égalité, on recommence jusqu’à avoir des valeurs différentes. Une fois qu’on a une situation gagnante pour le joueur, on retourne le plateau *P*, et la valeur 1 pour *idjoueur*. Sinon, l’ordinateur a gagné, dans un premier cas, nous renvoyions le plateau *P* et la valeur -1 pour *idjoueur*, mais dans ce cas, l’ordinateur jouait toujours le même coup. Nous avons donc choisi que l’ordinateur jouerait un coup au hasard. Nous avons donc créé une fonction **OrdiCommence**, qui prend pour paramètres *P*, et *N*, et qui place simplement un pion à une position au hasard sur le plateau.

Donc dans le cas où l’ordinateur gagne le pierre-feuille-ciseaux, on appelle la fonction **OrdiCommence**, et on renvoie le tableau *P* modifié et la valeur 1 pour *idjoueur*.

# 3ème version : 1ères optimisations :

Détections de blocages :

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | X |
|  |  |  |  |
|  | X |  |  |
| O |  |  |  |

Maintenant que nous avions un programme qui fonctionnait correctement, nous avons voulu l’optimiser un peu. Nous avons voulu dans un premier temps détecter les blocages. Pour cela, étudions plusieurs exemples où il faut toujours trois pions à aligner (soit *A* = 3) et où l’ordinateur joue les X :

|  |  |  |  |
| --- | --- | --- | --- |
|  | O |  |  |
|  | X |  |  |
|  | X |  |  |
|  | O |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
| O | X | X |  |
|  |  |  |  |
|  |  |  |  |

Ces situations représentent trois exemples de cas où nous pouvons étudier une situation de blocage ou pas pour les X.

* Dans la première situation, on remarque que l’on a deux X alignés, mais qu’il y a un O de chaque côté, dans cette situation les deux X sont bloqués, on peut donc passer leur valeur verticale à 0 dans la matrice des valeurs. Les deux O sont également bloqués par un X, on peut donc également passer leur valeur à 0.
* Dans la deuxième situation, il y a également deux X alignés. En regardant à gauche, on est bloqué par un O, mais à droite, on a une case libre, où il est possible de mettre un X. Cette ligne n’est donc pas bloquée
* Dans la troisième situation, il n’y pas de pions alignés. Regardons le X en bas à gauche, et tentons de voir si sa diagonale / est bloquée. Pour cela, regardons en bas à gauche : c’est un O, donc ce n’est pas la peine de continuer dans cette direction. Ensuite, regardons en haut à droite, c’est un trou, on n’est pas bloqué donc on peut continuer dans la même direction, où on a un X, donc on n’est pas bloqué non plus. On arrive au bout du tableau, et on remarque qu’il est possible de construire une diagonale de 3, donc les deux X ne sont pas bloqués.

Dans ces trois exemples, nous avons de procédé de la même façon, nous sommes partis d’un pion dont au moins une de ses valeurs dans la matrice des valeurs est non nulle. Nous avons ensuite regardé les cases adjacentes de chaque côté dans une direction ayant une valeur non nulle. Dès que nous arrivions à une case ayant un pion adverse, nous pouvions arrêter de regarder de ce côté car il n’a plus d’intérêt, et si nous arrivions au bout du tableau, nous pouvions nous arrêter. Dans les autres cas, nous comptions le nombre de trous, et le nombre de pions de notre couleur.

Une fois ces lignes étudiées, nous faisions la somme du nombre de trous et du nombre de pions de la bonne couleur, et si ce nombre est supérieur ou égal à *A*, alors la ligne n’est pas bloquée, sinon, elle est bloquée.

Dans le cas d’une ligne bloquée, il n’est plus intéressant de regarder les pions de la couleur étudiée dans la direction de la ligne. On peut donc passer la valeur de cette direction à 0 pour ces pions dans la matrice des valeurs.

Il y a également une remarque : il n’est pas nécessaire de regarder tous les pions d’un côté du pion étudié, jusqu’à arriver à un pion adverse ou au bout du tableau. Prenons cette situation où *A* = 3 :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | X |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  | X |  |  |
|  |  | O |  |  |

Regardons le X du bas, et testons s’il est bloqué verticalement. Il est bloqué vers le bas, donc on regarde vers le haut ; certes on n’est pas bloqué, mais il n’est pas nécessaire de monter jusqu’à l’autre X, car on ne pourra pas faire de ligne de 3 contenants ces deux pions. Donc on peut en déduire qu’il n’est pas nécessaire de regarder l’entièreté de la ligne de chaque côté du pion, mais simplement les *A* première cases.

On peut donc construire une fonction blocage, qui prend pour paramètres le plateau de jeu *P*, sa taille *N*, le nombre de pions à aligner pour gagner *A*, et la matrice de valeurs *M*.

Cette fonction va donc parcourir toutes les cases de la matrice *M* et appliquer l’algorithme décrit précédemment, et renvoyer la matrice *M* modifiée (ou pas).

Diagonales inutiles :

Prenons un exemple où *A* = 4 :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  | X | O |  |  |
|  |  | X | O |  |

Dans cette situation, on remarque qu’il est inutile pour les X de compléter la diagonale, car ils ne pourront en aligner que 3. En revanche, les O peuvent compléter la diagonale car il est possible d’en aligner 4. On remarque qu’il existe pour chaque taille de plateau et pour chaque A des diagonales pour lesquelles il sera impossible de gagner.

Afin de rendre l’ordinateur plus intelligent, il ne faudrait pas qu’il essaie de compléter une diagonale, si celle-ci est une diagonale où il sera impossible de gagner. Il faut maintenant trouver un moyen efficace de trouver les coordonnées de ces diagonales.

Diagonales \ :

Pour trouver un moyen efficace de trouver les diagonales impossibles, nous avons tracé des tableaux de différentes tailles et nous avons trouvé à la main les diagonales impossibles pour différentes valeurs de *A*. Notre objectif était de trouver une ou des propriétés sur les coordonnées des diagonales en fonction de *N* et *A*. Faisons quelques exemples, où les diagonales valides sont en rouge :

N = 4 et A = 4 N = 4 et A = 3

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Coordonnées : [0 ;0] Coordonnées : [0 ;0], [0 ;1]

[1 ;1] [1 ;0], [1 ;1], [1 ;2]

[2 ;2] [2 ;1], [2 ;2], [2 ;3]

[3 ;3] [3 ;2], [3 ;3]

N = 5 et A = 5 N = 5 et A = 3

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

Coordonnées : [0 ;0] Coordonnées : [0 ;0], [0 ;1], [0 ;2]

[1 ;1] [1 ;0], [1 ;1], [1 ;2], [1 ;3]

[2 ;2] [2 ;0], [2 ;1], [2 ;2], [2 ;3], [2 ;4]

[3 ;3] [3 ;1], [3 ;2], [3 ;3], [3 ;4]

[4 ;4] [4 ;2], [4 ;3], [4 ; 4]

En faisant ces exemples, nous avons remarqué une propriété : la différence entre les deux coordonnées (en valeur absolue) est toujours inférieure ou égale à *N*-*A*. En exploitant cette propriété, nous avons pu construire une fonction **CoordDiagGD**, qui va simplement déterminer les coordonnées des diagonales autorisées et les renvoyer.

Diagonales / :

Pour ces diagonales, nous avons tenté de trouver une propriété de la même façon que pour les autres diagonales. Mais nous n’avons pas réussi à trouver une propriété permettant de trouver les coordonnées. En revanche, nous avons remarqué qu’il était assez simple de passer d’une diagonale \ à une diagonale / pour les mêmes valeurs de *N* et de *A*. Exemple :

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Coordonnées : [0 ;0], [0 ;1] Coordonnées : [0 ;2], [0 ;3]

[1 ;0], [1 ;1], [1 ;2] [1 ;1], [1 ;2], [1 ;3]

[2 ;1], [2 ;2], [2 ;3] [2 ;0], [2 ;1], [2 ;2]

[3 ;2], [3 ;3] [3 ;0], [3 ;1]

On remarque que le *x* n’a pas changé, par contre le *y* a changé. Pour déterminer *y*, nous faisons simplement *N*-1-*y* (de la première matrice) modulo *N*.

Nous pouvons donc faire une fonction **CoordDiagDG** qui va simplement appeler **CoordDiagGD**, puis faire le calcul précédent sur les coordonnées obtenue avec la première fonction.

Utilisation des fonctions :

Nous allons utiliser ces fonctions dans **EvalPosition**. Nous récupérons les coordonnées des deux types de diagonale au début de la fonction. Puis à la fin, après avoir calculé la matrice *M* et avant de déterminer les blocages, on regarde toutes les cases de *M*, et si les coordonnées de la case ne font pas partie des coordonnées de la diagonales \, alors la première valeur de la matrice des valeurs (valeur relative à la diagonale \) passe à 0. Si les coordonnées de la case ne font pas partie des coordonnées de la diagonales /, alors la troisième valeur de la matrice des valeurs (valeur relative à la diagonale /) passe à 0.

Calcul des nouveaux maximums :

Dans les précédentes versions, nous déterminions la valeur du maximum de pions alignés pour chaque joueur (ainsi que d’autres informations sur ce pion maximal) au moment de calculer la matrice des valeurs *M*. Mais maintenant, les valeurs de la matrice *M* sont susceptibles d’être modifiées, ce qui pourrait fausser les valeurs précédemment calculées. Il faut donc changer le moment où nous déterminons les maximums.

Cette détermination se fera donc après avoir déterminé les blocages. Nous allons donc parcourir la matrice *M*, regarder à qui appartient le pion dans la case, et comparer la valeur dans chaque direction au maximum du joueur approprié. Nous récupèrerons et stockerons les informations de la même façon que dans la dernière version.

# Version 4 : Nouvelle optimisation :

Il reste à prendre en compte une situation pour que l’ordinateur joue mieux. Le type de situation est le suivant :

*N* = 4 et *A* = 4

|  |  |  |  |
| --- | --- | --- | --- |
|  | X | O |  |
|  |  |  |  |
|  |  | O |  |
|  |  |  |  |

Cette situation est très avantageuse pour les O, mais l’ordinateur ne le remarquera pas, et s’il joue les X, il ne tentera pas de jouer entre les O. Il faut donc trouver un moyen de repérer ce genre de ligne, et de les bloquer ou de les compléter selon les situations.

Construction d’une nouvelle matrice :

Afin de repérer et de mémoriser ces lignes, nous avons décidé de construire une nouvelle matrice *Mt*, qui aura la même structure que la matrice *M*, sauf qu’elle ne comptera pas le nombre de pions adjacents, mais le nombre de pions dans cette direction même s’il y a des trous.

La construction de cette matrice se fera dans **EvalPosition**, après avoir déterminé les blocages. Nous allons donc parcourir notre plateau de jeu *P*, et pour chaque pion, nous allons regarder les cases dans la diagonale \ à gauche, en haut, dans la diagonale / à droite, et enfin à gauche. Comme pour détecter les blocages, il n’est pas nécessaire de regarder toute la ligne dans cette direction, mais seulement les *A* premières cases. On regarde donc le contenu de chaque case dans chaque direction (si on peut regarder *A* cases sans sortir du plateau). On utilise une variable *cptcouleur* qui va compter le nombre de pions de la même couleur que le pion étudié et une variable *cpttrou*, qui va compter le nombre de trous. Si on trouve une case vide, on incrémente *cpttrou* et on continue, si on trouve une case de la même couleur, on augmente *cptcouleur*, et on continue, et si on trouve un pion de l’adversaire, on arrête de parcourir la ligne, et on met 0 dans *Mt* pour ce pion pour la direction traitée.

Il faut maintenant remplir la matrice *Mt* quand on n’a pas rencontré de pion adverse. Pour cela, on met simplement la valeur de *cptcouleur* dans *Mt* pour la direction traitée. Mais il faut faire attention à certaines situations, par exemple :

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
| O | X |  |  |
| X |  |  |  |

*N* = 4 et *A* = 4

Dans cette situation, on remarque que les X ont une diagonale entamée. Elle sera indiquée dans la matrice *M*, par contre en calculant la matrice *Mt*, on va partir du *X* en bas à gauche, et monter de 4 cases selon la diagonale /. On ne va pas trouver de pions adverses, on va trouver deux pions de la bonne couleur, et deux trous, et on va donc vouloir mettre 2 dans la matrice *Mt*, mais on remarque qu’il s’agit de la même valeur que dans la matrice *M*. Il n’est donc pas très intéressant d’écrire cette information à nouveau, d’autant plus qu’il s’agit d’un début de ligne sans trou, ce qui n’aurait pas eu de sens dans la matrice *Mt*.

Il faut donc vérifier si le nombre de pions de la bonne couleur dans une direction est égal à la valeur de *M* pour ce pion dans cette direction. Si ces valeurs sont égales, il est inutile de l’écrire à nouveau dans *Mt*, on met donc 0, sinon, on met la valeur de *cptcouleur* dans *Mt*.

Détermination des nouveaux maximums :

Maintenant que nous avons deux matrices de valeur, le maximum peut se trouver dans une des deux matrices (ou dans les deux). Donc maintenant, au moment de déterminer les maximums, il faut parcourir les matrices *M* et *Mt* en même temps, et comparer le maximum approprié aux valeurs des deux matrices. La détermination des autres informations se fera de la même façon.

Cependant, nous verrons dans la prochaine partie que nous ne remplissons pas de la même façon une ligne « normale » (ligne dont les valeurs sont dans la matrice *M*), et une ligne « à trou » (ligne dont les valeurs sont dans la matrice *Mt*). Il faut donc indiquer si la ligne maximum est une ligne normale ou une ligne à trou. Pour cela, nous créons deux nouvelles listes *type\_ligne\_joueur* et *type\_ligne\_ordi*, qui vont contenir le type de ligne pour chaque maximum de chaque joueur. Une ligne normale correspondra à une valeur « *n* » et une ligne à trou correspondra à une valeur « *t* ».

Remplissage d’une ligne à trou :

Maintenant, quand l’ordinateur joue, il faut qu’il regarde quel est le type de ligne maximum, pour savoir comment remplir cette ligne. Il sait donc déjà comment remplir une ligne normale, mais pas encore une ligne à trou. Pour cela nous avons créé 4 nouvelles fonctions (une pour chaque direction), qui prendrons pour paramètre le plateau de jeu *P*, la taille du plateau *N*, le nombre de pions à aligner pour gagner *A* et les coordonnées du pion maximum, *coord\_max*, et va retourner le plateau modifié et la valeur de *place* déterminant si le pion a bien pu être placé.

Chaque fonction va partir des coordonnées max et regarder les *A*-1 cases dans une direction, et dès qu’elle trouve une case vide, elle la remplit avec un pion de l’ordinateur.

# Version 5 : MinMax :

Nous allons maintenant implémenter un algorithme **MinMax**, permettant de prédire des futures positions (à un degré donné) pour évaluer la position actuelle, et donner le prochain meilleur coup, de façon un peu bourrine (mais on l’améliorera avec un élagage alpha-beta).

Brièvement, **MinMax** retourne le prochain coup de façon à minimiser la meilleure position de l’adversaire du joueur courant.

On crée donc deux fonctions :

* **minimax**, qui va s’appeler récursivement en diminuant son degré (sa profondeur) actuel jusqu’à atteindre le degré voulu, où il va utiliser **EvalPosition**. Cette fonction prend en paramètres :
  + La dimension de notre gomoku : *N*
  + La position actuelle dont on veut évaluer la position : *P*
  + Le nombre de pions à aligner : *A*
  + Le degré (la profondeur) actuel : on a déjà prédit combien de coups par rapport à la position à évaluer ? : *degre\_actuel*
  + Le degré voulu : combien de coups on veut prédire à l’avance : *degre\_voulu*
  + Quel joueur joue à ce tour ? : *joueur*

Cette fonction va retourner le score de la position qu’on lui a donné, ainsi que les coordonnées du coup optimal

* **MinMax**, qui va effectuer l’appel principal. Cette fonction a les mêmes paramètres que minimax à l’exception de *degre\_actuel*.

Remarque : Utiliser **MinMax** à un degré 0 revient à faire simplement appel à **EvalPosition** sur la position courante.

Si le degré voulu est strictement positif, **MinMax** vérifie si le degré voulu est supérieur au nombre de cases vides, puis appelle minimax (avec un *degré\_actuel* à 0), qui procède comme ci-décrit.

D’abord, on détermine toutes les prochaines positions possibles :

* Chaque case vide est un potentiel prochain coup à jouer

🡺 On récupère toutes les cases vides

* Pour chacune de ces cases vides
  + On crée une nouvelle position où le joueur courant (humain ou ordinateur) a joué dans cette case vide
  + On sauvegarde cette nouvelle position

De cette manière, on récupère toutes les prochaines positions.

Il nous faut maintenant évaluer chacune de ces positions. C’est ici qu’on fait apparaitre l’aspect récursif de **minimax** :

* On initialise le *score optimal* à –inf. si c’est le joueur qui joue, +inf. si c’est l’ordinateur, et le *coup optimal* à Null
* On diminue le degré actuel de 1
* Si le degré devient égal au degré voulu, pour chaque prochaine position *Next\_P*
  + On fait simplement **EvalPosition** sur cette position *Next\_P* pour récupérer son score
  + Sinon on fait **minimax** sur cette position pour récupérer le score de *Next\_P*, avec les mêmes paramètres (degré a déjà été diminué de 1), sauf *joueur* où on spécifie cette fois l’autre joueur, et *P* qui devient la prochaine position *Next\_P*
  + Pour chaque prochaine position, on vérifie si son score est meilleur (= inférieur si c’est l’ordi qui va jouer, =supérieur si c’est l’humain) que le *score optimal* (le meilleur score parmi les scores des positions qu’on a déjà parcouru)
    - Si c’est le cas, ce nouveau score devient le *score optimal*, et le *coup optimal* devient le coup joué pour arriver à la prochaine position
* Enfin, on retourne le *score optimal* et le *coup optimal* précédemment calculé

Et le tour est joué ! Faisons maintenant un exemple de déroulement sur une partie simple : avec un degré voulu de 2

|  |  |  |
| --- | --- | --- |
| X | O |  |
| O |  |  |
| X |  |  |

Ici, on admet que c’est à l’ordinateur (X) de jouer.

Les 5 coups possibles suivants sont les suivants :

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| X | O | X |  | X | O |  |  | X | O |  |  | X | O |  |  | X | O |  |
| O |  |  |  | O | X |  |  | O |  | X |  | O |  |  |  | O |  |  |
| X |  |  |  | X |  |  |  | X |  |  |  | X | X |  |  | X |  | X |

Ici, minimax va calculer le score de chacune des prochaines positions (les *sous-positions* où l’humain a joué un nouveau coup) de chacune de ces positions avec **EvalPosition**, et donc évaluer chacune de ces positions ci-dessus en gardant le score où l’humain est le plus avantagé (🡺 le score maximum) parmi les scores des *sous-positions*.

Enfin, on refait la même chose pour évaluer la position de base, sauf que cette fois on garde le score où l’ordinateur est le plus avantagé (🡺 le score minimum) parmi les scores des positions calculés. Au final, dans cet exemple, le 2ème cas est le meilleur car il aura deux lignes potentielles à compléter, et aura donc un score minimum (optimal pour l’ordinateur).

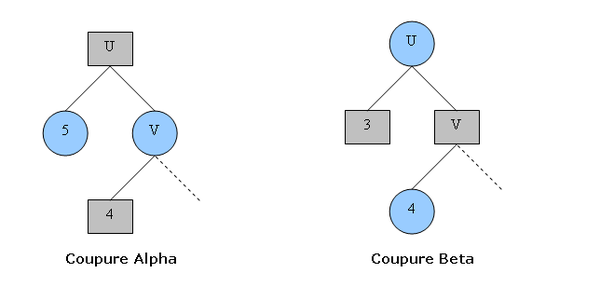
# Version finale : Elagage alpha-beta :

MinMax permet d’anticiper tous les prochains coups, ce qui peut très vite devenir long. L’idée de cette implémentation est de retirer les parcours inutiles, d’éliminer les coups qu’on sait pas optimaux, de façon à accélérer l’algorithme. Il exploite le fait qu’un coup est forcément favorable (ou nul) au joueur qui l’a joué.

Pour cela, on ajoute deux paramètres à **minimax** : *alpha* et *beta*, qui représentent respectivement le *score* optimal du degré précédent (minimum pour *alpha*, maximum pour *beta*). On n’utilise qu’un des deux à la fois, dépendant de si le nœud courant est un nœud max (tour de l’humain : dans ce cas c’est *beta* qui est utilisé) ou si c’est un nœud min (tour de l’ordinateur : dans ce cas on utilise alpha).

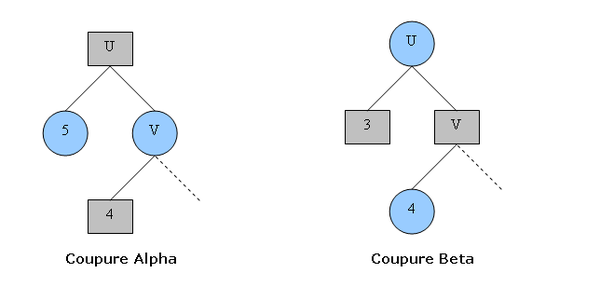
L’algorithme est le suivant :

* Si on est au tour du joueur,
  + On fait comme ce que faisait minimax
  + Durant le parcours des sous-positions, à la fin du parcours de la sous-position, on vérifie si notre *score optimal* (maximum) est supérieur à *beta*, le minimum de la position mère.
    - Si c’est le cas, ça veut dire que le score de la position courante (sous position de la position mère) sera forcément supérieur (car on cherche le max 🡺 il ne peut qu’augmenter) au *score optimal* de la position mère, qui cherche à être minimum. On arrête donc le parcours des sous-positions en retournant le *score optimal* actuel ainsi que le coup à jouer pour arriver dans cette position.



(Source : [https://fr.wikipedia.org/wiki/Élagage\_alpha-bêta](https://fr.wikipedia.org/wiki/%C3%89lagage_alpha-bêta) )

* + - Par exemple, ici, l’humain est en gris (et cherche à maximiser son score en U). Comme son *score optimal* ici vaut 5, et que le score de la prochaine position V est d’au plus 4 (minimisant le score pour l’ordinateur), on ne calcule pas le reste des sous-positions de V.
* Si on est au tour de l’ordinateur,
  + On fait comme ce que faisait minimax
  + Durant le parcours des sous-positions, à la fin du parcours de la sous-position, on vérifie si notre *score optimal* (minimum) est inférieur à *alpha*, le minimum de la position mère.
    - Si c’est le cas, ça veut dire que le score de la position courante (sous position de la position mère) sera forcément inférieur (car on cherche le min 🡺 il ne peut que diminuer) au *score optimal* de la position mère, qui cherche à être maximum. On arrête donc le parcours des sous-positions en retournant le *score optimal* actuel ainsi que le coup à jouer pour arriver dans cette position.



(Source : [https://fr.wikipedia.org/wiki/Élagage\_alpha-bêta](https://fr.wikipedia.org/wiki/%C3%89lagage_alpha-bêta) )

* + - Par exemple, ici, l’ordinateur est en bleu (et cherche à minimiser son score en U). Comme son *score optimal* ici vaut 3, et que le score de la prochaine position V est d’au moins 4 (maximisant le score pour l’humain), on ne calcule pas le reste des sous-positions de V.

# Conclusion :

Au final, nous avons un programme qui fonctionne assez bien. L’ordinateur arrive à jouer et à gagner beaucoup de parties. Il existe cependant quelques situations où l’ordinateur ne joue pas de manière optimale, et où il peut ne pas remarquer une situation très favorable (un pion restant pour gagner par exemple) pour un joueur. Malheureusement, nous n’avons pas le temps d’étudier plus précisément d’où vient ce problème et de le corriger.

De plus, l’ordinateur peut mettre un certain temps à jouer, notamment sur un grand plateau ou avec un degré plus élevé. Cela pourrait s’améliorer en optimisant un peu plus les fonctions d’évaluation, minmax et d’élagage. Cette lenteur vient également en partie du langage python. Nous aurions gagné en vitesse en utilisant un autre langage.

Au final, nous avons beaucoup apprécié cette unité, et la réalisation de ce projet. Nous avons pu développer nos capacité en algorithmique et en programmation à travers un projet intéressant et original.