ADAM Baptiste
ZHOU Sébastien
AFCHAIN Léa

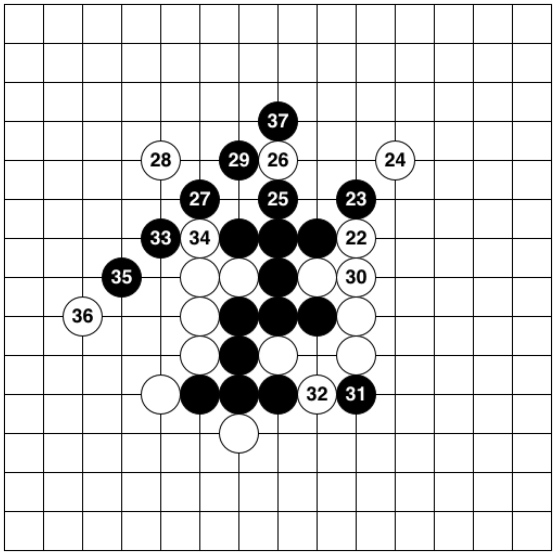
E3E

Atelier d'algorithmique – PR3001 – Projet Go-moku

Sommaire

Introduction

Lors de l'atelier d'Algorithmie, nous avons été amenés à réfléchir sur le Go-moku. Le Go-moku est un jeu de plateau à deux joueur souvent sur un plateau de 19 x 19. Les règles de ce jeu sont simples : le premier joueur qui obtient 5 pions consécutifs et alignés selon la longueur, la largeur ou les diagonales, gagne. Le Go-moku est plus dure que le morpion car nous utilisons un espace beaucoup plus grand et demande donc des stratégies plus complexes.



Pour notre projet, nous devons recréer ce jeu de plateau. Ce jeu doit être conçu pour deux joueurs dont une Intelligence Artificielle. Le but de ce projet est donc de créer une IA qui puisse prendre les meilleures décisions selon les situations et dans un délai raisonnable. Nous avons décidé de la coder en python pour la simplicité et la polyvalence de ce langage malgré la rapidité réduite.

Le programme doit être composé au minimum des fonctionnalités suivantes :

- une fonction initialisant la grille de jeu
- une fonction affichant la grille de jeu
- une fonction arrêtant le jeu s'il n'y a plus de cases libres
- une fonction pour saisir le coup du joueur sur la grille
- une fonction d'évaluation qui retourne un score proportionnel à l'avancement du joueur dans la partie

Une fois le projet plus avancé, il est possible d'améliorer la fonction d'évaluation en utilisant la stratégie du minimax. Cela sera détaillé plus tard.

A. Création du squelette

Les 3 premières versions du projet ont servi à créer le squelette global du programme. Ils utilisaient une fonction d'évaluation peu efficace mais rapide à programmer. Le programme a donc été séparé en trois fichiers distincts :

- **fonctions.py :** Ce fichier contient toutes les fonctions basiques indispensables au bon fonctionnement du jeu.
- **game_engine.py :** Ce fichier utilise les fonctions de *fonctions.py* pour faire tourner le jeu.
- **main.py :** Ce fichier initialise le *game engine*.

Lors du choix du design, il a été décidé d'appeler les colonnes par des lettres, et les lignes par des nombres. De cette façon, la première case du tableau est référencée par la dénomination A1 et la deuxième case de la première ligne par la dénomination B1. Ces lettres sont stockées dans la liste *alpha_utile*. Il a aussi été décidé qu'une case vide sera codée par un 0, une case où le joueur a joué par un 1 et une case où l'IA a joué par un -1.

I/ fonctions.py

La première fonction de ce fichier est *creer_tableau(N)*, elle crée un tableau à deux dimensions de taille N. La deuxième fonction, *afficher_tableau(...)*, affiche le tableau en utilisant un espace pour les cases vides, une croix pour le joueur et un rond pour l'IA.

Maintenant que nous avons notre grille et que nous sommes capable de l'afficher, il faut pouvoir jouer dedans. C'est le travail de la fonction *jouer(tab, N, ligne, colonne, joueur)*. Elle écrit dans la case correspondante à la ligne et la colonne passées en paramètre la valeur *joueur* (qui vaut donc 1 ou -1 en fonction du joueur concerné). Elle retourne 0 si l'action s'est bien passée ou -1 si l'action n'est pas possible (tentative de jouer sur une case déjà jouée ou en dehors de la grille).

Il a été décidé que le joueur pouvait faire plusieurs actions différentes lors d'une partie. La fonction *help()* affiche donc la liste de toutes les actions possibles et leur utilité. Parmis ces actions, il y a :

- help : pour afficher la liste des actions possibles
- quit : pour quitter une partie en cours
- print : pour afficher la grille
- [jouer dans une case] : explicite comment jouer dans une case

Avec l'avancement du projet, des actions de débogage ont été ajoutées :

- test : affiche la valeur de score du joueur et de l'IA
- pr[case] : permet de connaître la valeur de l'évaluation statique de la case demandée

Enfin, la dernière fonction de ce fichier à ce stade est *eval_position(...)*. Elle ne sera pas détaillée ici puisque la fonction utilisée n'a pas pour vocation de rester jusqu'à la fin. Néanmoins, il est important de savoir que le score renvoyé est entre 0 et 1000 et que -1 est renvoyé si il n'y a plus de cases libres.

II/ game_engine.py

Ce fichier possède une constante globale ALPHABET, c'est une chaîne de caractère avec toutes les lettres de A à Z. Elle est utilisée pour créer *alpha_utile* lors de l'initialisation.

Les deux premières fonctions de ce fichier sont le noyau du fonctionnement du programme. *initialisation(N)* crée *alpha_utile* en gardant les N premières lettres de ALPHABET. Elle crée un tableau de taille N en appelant la fonction correspondante de *fonction.py*. Elle définit Nb, le nombre de pions à aligner dans la grille pour gagner. Enfin, elle lance la fonction *game_engine(...)*.

La fonction *game_engine(...)* commence par décider aléatoirement qui du joueur ou de l'IA commence. Si l'IA commence, celle-ci joue, puis nous rentrons dans une boucle tant que où chaque itération correspond à un tour de jeu (le joueur joue puis l'IA joue). Grâce à un input, la fonction obtient l'action du joueur et commence à la traiter. Cette action peut être égale à toutes les actions détaillées ci-dessus et l'action correspondante est effectuée. Si la saisie ne correspond à rien, un message redirigeant vers la fonction *help()* est affiché.

Dans le cas où le joueur tente de placer un pion, plusieurs critères sont à respecter. La saisie doit être au maximum composé de 3 caractères (une lettre et entre 1 et 2 chiffres pour l'éventualité où la grille est suffisamment grande). Une fois avoir tenté de jouer aux coordonnées spécifiées, un message d'erreur s'affiche si ca n'a pas été possible. Si la tentative s'est bien passée, il faut vérifier si le joueur a maintenant gagné grâce au score renvoyé par *eval_position(...)*, si ce n'est pas le cas, il faut vérifier s'il y a encore des cases libres sur la grille toujours grâce au score renvoyé (si il vaut -1).

Après tout cela, il faut maintenant faire jouer l'IA grâce à la fonction *jouer_ia(...)* qui sera détaillée juste en dessous. Les mêmes vérifications que pour le joueur sont de mise (si il a gagné et si la grille est pleine).

La fonction *jouer_ia(...)* fait jouer l'IA dans la case la plus intéressante. Elle évalue d'abord la position du joueur et de l'IA. Si les deux valeurs retournées sont à 0 cela indique que toutes les cases sont vides, l'IA joue vers le centre du plateau. Ensuite cette fonction permet de bloquer un coup très dangereux notamment de bloquer 4 pions alignées donc d'empêcher la victoire du joueur, et pour repérer la dangerosité du joueur, il faut regarder la valeur retournée par le *eval_position()* faite précédemment.

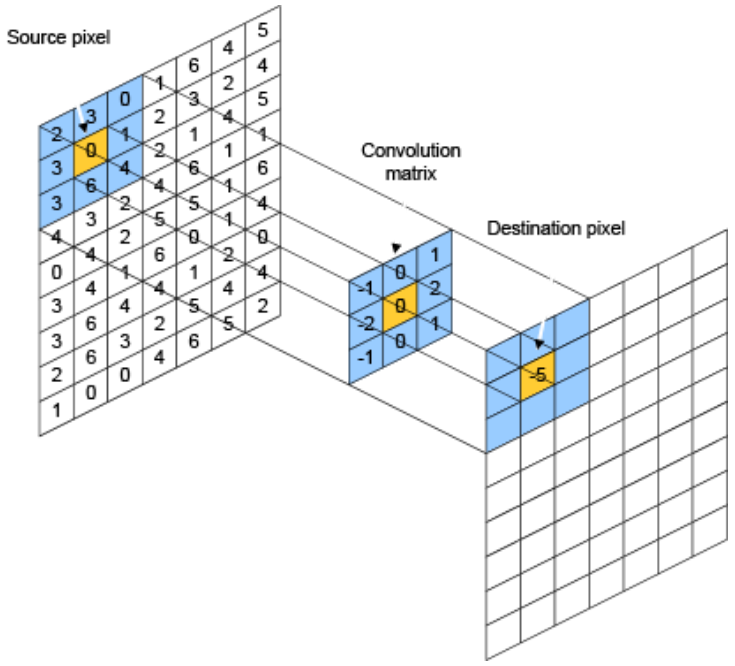
Finalement, la fonction *win(joueur)* affiche un message de victoire ou de défaite en fonction de la valeur *joueur* passée en paramètre.

III/ main.py

Ce fichier initialise la valeur N puis appelle la fonction *initialisation(N)* de *game_engine.py*.

B. Amélioration de eval_position(...)

Pour notre projet Go-moku, le filtre de Sobel a été utilisé. Il s'agit d'une technique de filtrage local utilisé en traitement d'images pour la détection de contours, obtenue par convolution. Une convolution consiste à calculer le résultat de chaque pixel en faisant une somme pondérée de son voisinage, en fonction d'une matrice de convolution. Dans l'exemple ci-dessous, la valeur du pixel de destination est obtenue en faisant la somme des produits des voisins avec leur correspondant dans la matrice de convolution.



source : information et support visuel récupéré sur le site de Benjamin Raynal, schooding.com

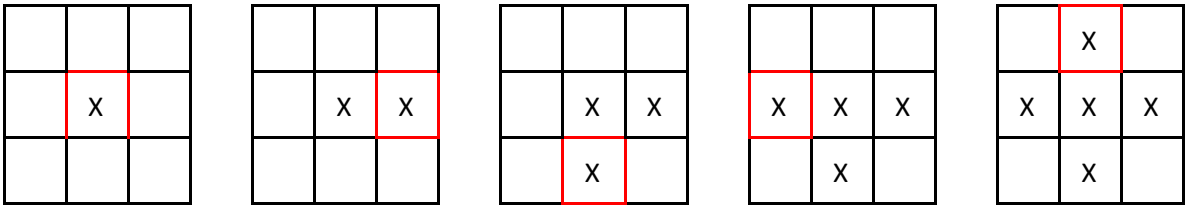
Nous avons ensuite décidé de séparer le programme *eval_position(...)* en deux parties : *eval_position(...)* qui regarde l'état de la partie et *eval_action(...)* qui renvoie les coordonnées du point le plus intéressant à jouer pour l'IA. Cela a permis de pouvoir utiliser la fonction *eval_position(...)* pour le le joueur afin de déterminer si celui-ci a gagné après avoir jouer sans faire tous les calculs nécessaires pour savoir où joue l'IA.

I/ Eval_action(...)

eval_action(...) retourne les coordonnées de la case où l'IA va jouer. Dans une première version le programme utilise un filtre de Sobel avec des coefficients dépendants de la distance de la case avec le point observé, sur la ligne, la colonne et les diagonales passant par le point observé. La matrice utilisée est la suivante :

2	0	0	0	2	0	0	0	2
0	3	0	0	3	0	0	3	0
0	0	4	0	4	0	4	0	0
0	0	0	5	5	5	0	0	0
2	3	4	5	0	5	4	3	2
0	0	0	5	5	5	0	0	0
0	0	4	0	4	0	4	0	0
0	3	0	0	3	0	0	3	0
2	0	0	0	2	0	0	0	2

Cette première matrice permettait au IA de jouer de façon plus logique comparé aux versions précédentes. Cependant le calcul restait très mauvais, l'IA jouait "en croix" :



Les cases en diagonales directs ayant un plus grand coefficient qu'une case alignée à 2 cases de distance, l'IA décidais donc de jouer dans une case comme l'étape 3 ci-dessus. La première matrice a donc été séparée en quatre filtres de Sobel pour chaque direction. Les matrices sont maintenant les suivantes :

0	0	0	0	2	0	0	0	0
0	0	0	0	3	0	0	0	0
0	0	0	0	4	0	0	0	0
0	0	0	0	5	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	5	0	0	0	0
0	0	0	0	4	0	0	0	0
0	0	0	0	3	0	0	0	0
0	0	0	0	2	0	0	0	0

0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
2	3	4	5	0	5	4	3	2
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0

2	0	0	0	0	0	0	0	0
0	3	0	0	0	0	0	0	0
0	0	4	0	0	0	0	0	0
0	0	0	5	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	5	0	0	0
0	0	0	0	0	0	4	0	0
0	0	0	0	0	0	0	3	0
0	0	0	0	0	0	0	0	2

0	0	0	0	0	0	0	0	2
0	0	0	0	0	0	0	3	0
0	0	0	0	0	0	4	0	0
0	0	0	0	0	5	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	5	0	0	0	0	0
0	0	4	0	0	0	0	0	0
0	3	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0

Avec ces nouveaux filtres, nous calculons 4 valeurs correspondant aux 4 directions. Chaque pion étant représenté par 1 ou -1, la valeur la plus grande représente le meilleur coup pour le joueur et la valeur la plus petite représente le meilleur coup pour l'IA. En choisissant le maximum de la valeur absolue de ces valeurs, le coup joué sert soit à tenter de gagner, soit à bloquer le joueur.

En parallèle de la recherche de la plus grande valeur, les coordonnées de la case la plus intéressante rencontrée jusque-là sont sauvegardées. La fonction retourne donc la valeur maximale et les coordonnées de la case correspondante.

II/ Eval_position(...)

eval_position(...) fonctionne sur le même principe que *eval_action(...)*, mais au lieu d'utiliser des matrices coefficientées donnant une importance à certaines cases, elle utilise des matrices dont la direction regardée est remplie de 1.

0	0	0	0	1	0	0	0	0
0	0	0	0	1	0	0	0	0
0	0	0	0	1	0	0	0	0
0	0	0	0	1	0	0	0	0
0	0	0	0	1	0	0	0	0
0	0	0	0	1	0	0	0	0
0	0	0	0	1	0	0	0	0
0	0	0	0	1	0	0	0	0
0	0	0	0	1	0	0	0	0

0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0

1	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0
0	0	0	0	1	0	0	0	0
0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	1

0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	1	0	0
0	0	0	0	0	1	0	0	0
0	0	0	0	1	0	0	0	0
0	0	0	1	0	0	0	0	0
0	0	1	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0

Le but est simplement de compter le nombre de pions autour de la case regardée (le pions de la dite case est inclue dans le comptage). De cette façon, le nombre le plus grand correspond à la case avec le plus de pions du joueur dans les alentours alors que le minimum correspond à la case avec le plus de pions du l'IA. Pour cette case, il faut mémoriser la direction dans laquelle sont situés les pions ainsi que les coordonnées.

Durant le parcours de la grille, nous en profitons pour compter les case vides, cela permet de savoir si la grille est pleine. Si c'est le cas, la fonction renvoie directement -1, ce n'est pas la peine d'aller plus loins dans le traitement.

Maintenant que nous savons où se situe le plus grand nombre de pions du joueur qui nous intéresse, nous devons vérifier si ils sont côte à côte, et surtout, combien sont côte à côte. Pour ce faire, nous regardons chaque pions dans l'ordre en commençant par le premier. Tant que ce sont les pions du joueur qui nous intéresse, nous les comptons. Lorsque nous rencontrons une case vide ou un pion de l'opposant, nous gardons en mémoire le nombre de pions alignés si c'est le plus grand que nous avons rencontré jusque-là.

Après avoir parcouru la totalité des pions, nous connaissons donc le nombre de pions alignés. Il est temps de renvoyer le score adéquat. Si un score de 1000 représente 5 pions alignés, il suffit de renvoyer une valeur proportionnelle à ce score en fonction du nombre de points alignés.

Actuellement, notre programme fonctionne correctement sur des grilles dont le côté est compris entre 6 (voulant aligner 5 pions, c'est le minimum possible) et 26 (désignant les colonnes par des lettres, c'est le maximum que nous pouvons faire sans utiliser des stratagèmes bien compliqués pour simplement nommer des colonnes).

C. Anticipation des coups

L'IA n'ayant pas les fonctions cognitives pour "penser" il lui ait impossible d'anticiper les coups en n'utilisant que la fonction d'évaluation. Pour combler ce manque, nous pouvons utiliser l'algorithme *minimax*.

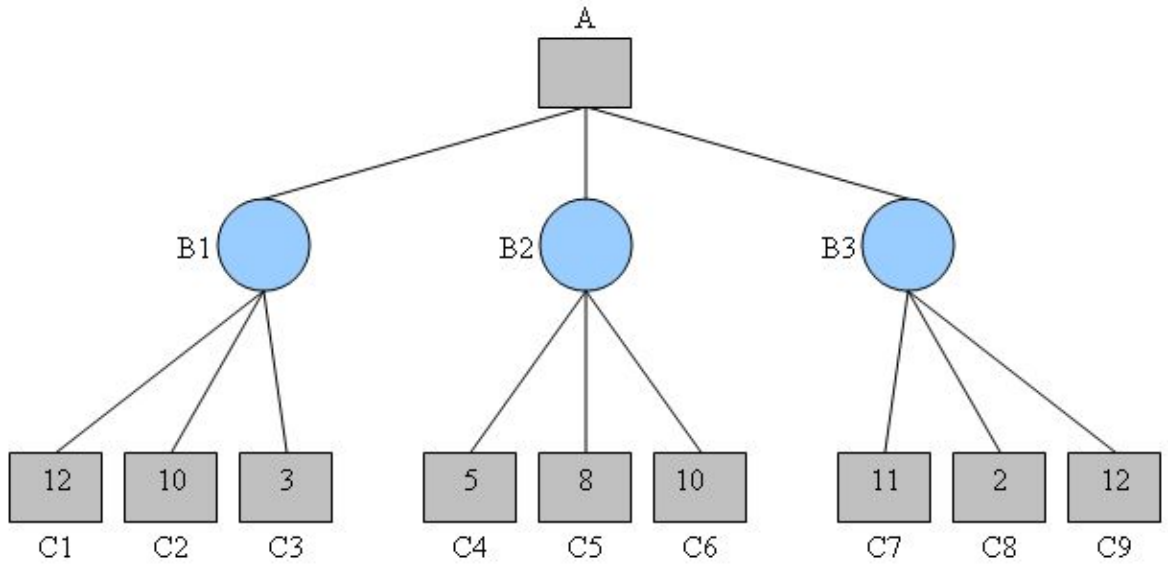
I/ Minimax

L'algorithme *minimax* crée un arbre correspondant à tous les coups possibles basé sur la situation de départ. Plus nous descendons dans l'arbre, plus nous regardons dans le futur. La racine de l'arbre représente donc la position des joueurs au moment présent du jeu. Cet algorithme repose sur le principe suivant : le joueur cherchera toujours à maximiser son gain alors que l'opposant cherchera toujours à minimiser le gain du joueur.

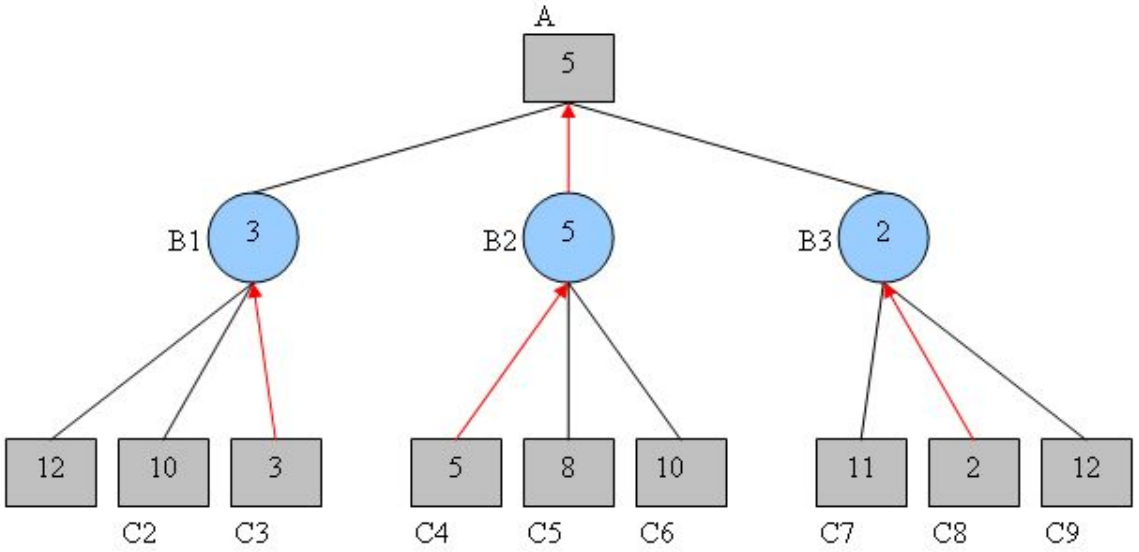
Cela ce traduit donc par :

- le joueur prends le maximum des possibilités qui s'offrent à lui
- l'opposant prends le minimum des possibilités qui s'offrent à lui

Ci-dessous un exemple :



Dans le schéma ci-dessus, les nœuds gris représentent les nœuds joueurs et les bleus les nœuds opposants. Pour déterminer la valeur du nœud A, on choisit la valeur maximum de l'ensemble des nœuds B. Il faut donc déterminer les valeurs des nœuds B qui reçoivent chacun la valeur minimum stockée dans leurs fils. Les nœuds C sont des feuilles, leur valeur peut donc être calculée par la fonction *eval_action(...)*.



Dans cet exemple, le noeud A prends donc la valeur 5.

Dans le meilleur des mondes, il faudrait continuer jusqu'à une situation de victoire pour le joueur. Mais cela est beaucoup trop gourmand en ressources et générer toutes ces possibilités prend beaucoup trop de temps : nous devons donc nous limiter à une certaine profondeur de l'arbre.

Voici le pseudo code sur lequel nous nous somme basé :

```
function minimax(node, depth, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value = −∞
        for each child of node do
            value = max(value, minimax(child, depth −1, FALSE))
        return value
    else (* minimizing player *)
        value = +∞
        for each child of node do
            value = min(value, minimax(child, depth −1, TRUE))
        return value
```

---

source : https://fr.wikipedia.org/wiki/Algorithme_minimax#Principe

Nous avons dû créer une fonction supplémentaire : *calcul_all_childs(tab, N, joueur)* qui calcule toutes les grilles possibles résultant de la grille *tab* passée en paramètre. Le paramètre *joueur* correspond au joueur jouant dans cette simulation.

Voulant garder en mémoire les coordonnées du point dans lequel nous voulons jouer, notre fonction *minimax(...)* ne renvoie pas seulement la valeur comme dans le pseudo code, elle renvoie aussi lesdites coordonnées. Par conséquent, nous avons dû créer des fonctions min et max permettant de ne pas perdre l'association des coordonnées avec la valeur.

maximum(value1, value2) et *minimum(value1, value2)* où value1 et value2 sont des tuples de la forme (value, coord) comparent donc les deux valeurs des tuples passés en paramètre et renvoient le couple correspondant.

Nous avons aussi décidé que la variable *depth* serait initialisée dans *main.py* pour une facilité de modification.

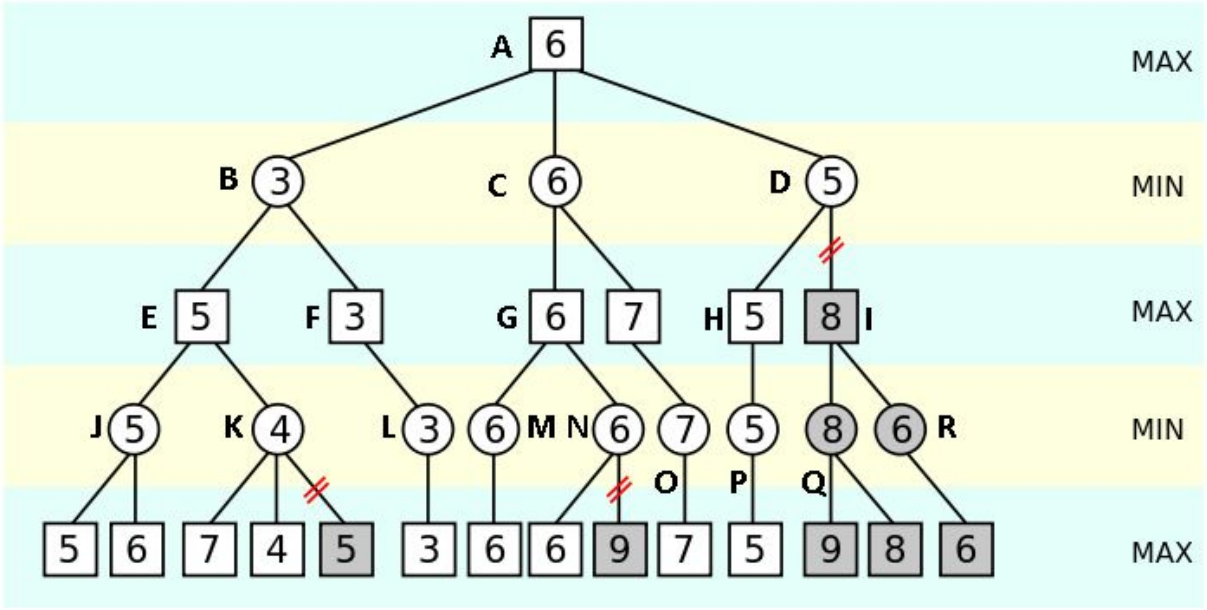
Comme mentionné plus haut, cette procédure est extrêmement gourmande en ressources, et même en se limitant à une profondeur de 2, le temps d'exécution est long, chaque coup de l'IA dure plusieurs secondes pour un plateau de côté 8, à savoir deux fois plus petit qu'avant l'implémentation de *minimax(...)*. Cela peut monter à quelques minutes voir quelques heures selon la taille du plateau. Pour remédier à cela, nous devons implémenter la stratégie de l'élagage alpha-beta.

II/ Élagage alpha-beta

L'élagage vient s'ajouter à l'algorithme *minimax(...)* afin de l'optimiser. En effet l'élagage alpha-beta n'évalue pas des nœuds dont on est sûr que leur qualité sera inférieure à un nœud déjà évalué. Pour un joueur, cela veut dire qu'un noeud dont la valeur est plus petite que la valeur la plus grande trouvée jusque-là ne sera pas regardée. Pour l'opposant, cela signifie que tout noeud dont la valeur est plus grande que la plus petite valeur trouvée jusque-là ne sera pas regardée.

Grâce à ça nous pouvons réduire considérablement le temps de calcul : ainsi nous pourrons effectuer des recherches beaucoup plus rapidement voir même beaucoup plus loin dans l'arbre de jeu.

Ci-dessous un exemple :



Le noeud K met à jour sa valeur à 4 puisqu'il recherche la plus petite valeur possible. Or le noeud E cherche la plus grande valeur possible et est déjà sauvegardé à 5, le noeud K lui est donc inutile et on ne le regarde pas plus loin.

source : https://fr.wikipedia.org/wiki/%C3%89lagage_alpha-b%C3%AAta

Voici le pseudo code sur lequel nous nous sommes basé :

```
fonction alphabeta(nœud, α, β) /* α est toujours inférieur à β */
    si nœud est une feuille alors
        retourner la valeur de nœud
    sinon si nœud est de type Min alors
            v = +∞
            pour tout fils de nœud faire
                v = min(v, alphabeta(fils, α, β))
                si α ≥ v alors  /* coupure alpha */
                    retourner v
                β = Min(β, v)
     sinon
            v = -∞
            pour tout fils de nœud faire
                v = max(v, alphabeta(fils, α, β))
                si v ≥ β alors /* coupure beta */
                    retourner v
                α = Max(α, v)
     retourner v
```

Puisque l'élagage alpha-beta n'est simplement qu'une amélioration du minimax, il est possible de simplement améliorer la fonction *minimax(...)* existante. En terme de pseudo-code, cela donne :

```
function minimax(node, depth, maximizingPlayer, α, β) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value = -∞
        for each child of node do
            value = max(value, minimax(child, depth-1, FALSE, α, β))
            if value ≥ β do /* coupure beta */
                return value
            α = max(α, value)
    else (* minimizing player *)
        value = +∞
        for each child of node do
            value = min(value, minimax(child, depth-1, TRUE, α, β))
            if α ≥ value do /* coupure alpha */
                return value
            β = min(β, value)
    return value
```

Dans la fonction *minimax(...)*, nous avons donc ajouté les arguments alpha et bêta et les parties en couleurs dans la dernière section de pseudo code.

Initialement, au premier nœud de l'arbre, les valeurs de alpha et de bêta sont respectivement - l'infini et + l'infini. Mais dans notre programme nous avons décidé de remplacer cela par de grande valeur qui ne sont jamais atteinte lors du calcul de la valeur des cases.

Maintenant que l'élagage alpha-bêta est implémenté, le programme a retrouvé un temps de calcul plus que raisonnable. Il est de quasiment instantané sur un grille de côté 8 avec une profondeur de 2 si bien que nous pouvons agrandir la grille jusqu'à 11 de côté tout en gardant un temps de calcul acceptable. Une grille à 12 de côté reste jouable, mais chaque actions de l'IA nécessite un temps relativement élevée.

Conclusion

Nous sommes partis d'un programme basique que nous avons amélioré petit à petit. Nous avons eu l'occasion de mettre en pratique des compétences que nous avions acquis au cours des années précédentes, notamment pour le filtre de Sobel. Nous avons aussi découvert des algorithmes essentiels pour l'optimisation et la création de jeux de ce type au travers du minimax et de l'élagage alpha-beta.

En tout dernier lieu, nous avons ajouté la possibilité de choisir la difficulté de l'IA. Il y a maintenant deux modes de jeu possibles, une IA simple sur un plateau de côté 19, ou une IA plus intelligente sur un plateau plus restreint.

Malgré tout, notre réalisation est encore loin du véritable jeu de Go-moku. Pour pouvoir jouer sur un plateau de 19 par 19, il faudrait d'abord encore améliorer la vitesse de traitement de l'IA. Une première solution pour cela serait de coder en C plutôt qu'en python. Par la même occasion, si le traitement devient plus rapide, cela donne la possibilité d'augmenter la profondeur du minimax pour anticiper plus de coups. Une autre amélioration possible serait d'améliorer l'interface graphique afin de pouvoir cliquer sur la case où le joueur veut jouer plutôt que de devoir rentrer ses coordonnées. Enfin, nous avons fait le choix de décider qui commence la partie au hasard, mais nous aurions pu laisser ce choix au joueur.

Ce projet était grandement tourné vers l'optimisation d'algorithme, chose qui est un enjeu majeur de le monde professionnel. Cela nous a donc permis de comprendre l'importance et l'omniprésence de ce problème dans toutes activités. Finalement, une expérience intéressante à faire serait de confronter notre IA à elle-même ou à une autre IA au cours de plusieurs parties.