Résolution d’un Puissance 4 grâce à l’algorithme MinMax avec élagage alpha-bêta

Le puissance 4 est un jeu sur lequel j’ai énormément de temps de jeu, j’ai donc été très intéressé par le sujet de l’IA de puissance 4 et ai énormément travaillé et fait de recherches dessus.

Mon rendu est constitué de 3 fichiers python : 2 fichiers pour les classes Arbre et Puissance ainsi que le fichier program pour lancer le programme

Fichier program.py :

Pas grand-chose à décrire dans le fichier program.py, on créer les différents joueurs en fonction de s’ils sont joueurs ou IA, on peut également choisir si on joue dans une puissance 4 à 7 ou 12 colonnes.

Le temps est calculé dans le fichier program.py également. Ce temps n’est pas que la pour l’observation : on peut définir une durée max et une durée min pour l’IA, si elle met un temps de calcul qui n’est pas dans l’intervalle de ces deux durées, elle réduit ou augmente son nombre de coups calculés. On peut également choisir le nombre de coups initial à calculer.

Puissance3.py :

Fichier dans lequel se trouve la classe Puissance : le 3 dans le nom du fichier est là car il y avait puissance2.py et puissance1.py, j’ai codé plusieurs manières de stocker la grille en cherchant la plus efficace. Après des heures de code et de recherche j’ai découvert une personne qui avait fait une IA imbattable de puissance 4 avec MinMax codée en C++. Je m’en suis inspiré (lien : <http://blog.gamesolver.org/>)

La personne a fait certains choix que je n’ai pas repris (comme l’utilisation de l’algorithme Negamax, une variante de MinMax qui permet de ne pas différencier joueur actuel et joueur adverse) mais une m’a marqué par sa pertinence en termes de rapidité : le bitboard : Une grille de puissance 4 peut être vu comme une matrice 6\*7 ( 6\*12 dans notre cas) qui prend pour valeur 0, 1 ou 2 : 0 si vide, 1 si on a un pion, 2 si l’adverse a un pion. Mettons de côté la différence entre pion adverse et pion à nous. On a alors une matrice 6\*7 qui prend pour valeur 0 ou 1. On peut donc assimiler cette matrice à un nombre de 42 bits. Ce qui en termes de stockage sera grandement plus efficace.

Comme je me suis inspiré du site pour le bitboard, beaucoup de décision ne viennent pas de moi, je n’ai fait que comprendre et traduire en python.

On va stocker les bits dans l’ordre des colonnes, on aura ainsi besoin d’un bit supplémentaire pour chaque colonne pour marquer la séparation de colonne :

Si on représente en tableau on a chaque bit à l’emplacement:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 7 | 14 | 21 | 28 | 35 | 42 | 49 |
| 6 | 13 | 20 | 27 | 34 | 41 | 48 |
| 5 | 12 | 19 | 26 | 33 | 40 | 47 |
| 4 | 11 | 18 | 25 | 32 | 39 | 46 |
| 3 | 10 | 17 | 24 | 31 | 38 | 45 |
| 2 | 9 | 16 | 23 | 30 | 37 | 44 |
| 1 | 8 | 15 | 22 | 29 | 36 | 43 |

Représentons la grille de puissance 4 avec des 0 en supposant qu’on a joué 2 fois dans la colonne 1 et une fois dans la colonne 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 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 |

Cette grille a donc pour valeur : 2^0 + 2^1 + 2^7 = 131

On a ainsi stocké très efficacement l’ensemble des jetons. Il faut maintenant différencier nos jetons et les jetons de l’adversaire :

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

La valeur de la grille pour le joueur « 1 » est 2^0 + 2^7 = 129, pour le joueur « 2 » : 2^1 = 2

On aurait ainsi stocké la grille en 3 valeurs de 49 bits. Cependant on peut être plus malin que ça et stocker en seulement deux valeurs

Définissons le « mask » la grille sans prendre en compte à qui sont les jetons et position la grille juste pour le joueur « 1 »

mask = 131 en base 10 = 1100001 en base binaire  
position = 129 en base 10 = 1000001 en base binaire  
On peut remarque qu’on faisant l’opération binaire « Xor » ou « ou exclusif » entre mask et position, on trouve 0100000 en base binaire = 2 en base 10 soit la position du joueur « 2 »  
De cette manière on peut sauvegarder l’ensemble d’une grille grâce à position et mask.

En mémoire c’est efficace mais ce n’est pas là l’intérêt principal du bitboard. Le bitboard va nous permettre de calculer de manière extrêmement rapide s’il y a un alignement de 4 jetons. Normalement avec une matrice 6\*7 on aurait dû regarder dans un rayon de 3 cases tous les autours d’un jeton et voir s’il y a un alignement et faire ça pour chaque case de la grille. On doit faire une triple boucle for pour chaque case de notre matrice donc une quintuple boucle for au total, ce qui en temps de calcul est extrêmement long. Le bitboard va nous permettre de transformer ces 5 boucles en 8 lignes de calcul. En effectuant des opérations binaires, on regarde très rapidement les alignements. Il suffit de deux lignes de calcul pour regarder un alignement dans une direction (donc 8 pour chaque direction : horizontal, vertical, diagonale 1 et 2).

Voir directement dans le code : l’explication est plus lisible :  
Explication rapide pour le vertical (les autres fonctionnent pareil mais sont plus longs à expliquer)  
Supposons qu'on a un alignement : on aura dans le bitboard une chaine de 0 et 1 du type ...01011110...

            CAS ALIGNEMENT :                  |           CAS PAS D'ALIGNEMENT :

        pos      : ...01011110...             |       pos      : ...011011100...

        pos >> 1 :  ...01011110...            |       pos >> 1 :  ...011011100...

        donc m   :  ...00011100...            |       donc m   :  ...010011000...

        On regarde ensuite m et m >> 2 :     |       On regarde ensuite m et m >> 2 :

        m        : ...00011100...             |       m        : ...010011000...

        m >> 2   :   ...00011100...           |       m >> 2   :   ...010011000...

        Donc on a:   ...00010000...           |       Donc on a:   ...0000000...

        On return alors True                  |       On return alors False

J’ai aussi repris une méthode pour détecter les alignements de 3 jetons qui permettrons un alignement de 4 mais je ne fais pas d’exemple car ça serait encore plus long que celui-ci.

Arbre.py :

Fichier dans lequel se trouve la classe Arbre, c’est ici qu’on fait tourner MinMax et l’élagage. Maintenant qu’on a un bitboard fonctionnel, on va pouvoir faire notre MinMax.

J’avais à la base codée la classe Arbre avec une instance de la classe par nœud de l’arbre. Mais pour avoir plus de visibilité en débugage et une meilleure performance j’ai changé pour une seule instance dans laquelle on stock toutes les valeurs (ou poids) des nœuds dans un dictionnaire.

Lors de la création de l’instance on lance la fonction récursive qui va créer les enfants du nœud initiale. J’ai codé d’abord MinMax sans élagage pour être sûr d’avoir un bon fonctionnement : Mon MinMax fonctionne sans heuristique, ils cherchent à un certain nombre de coups soit il gagne : le poids vaut alors le nombre de jetons restant, soit le joueur adverse gagne le poids vaut alors le nombre de jeton restant mais en négatif, soit il ne va pas assez loin pour savoir alors le poids vaut 0.

J’ai ensuite rajouté l’élagage simple : en fonction de si on est en élagage alpha ou beta, on se compare à ses « oncle » et on regarde si on a intérêt à continuer les recherches. Cependant l’élagage peut aller encore plus loin : on peut se comparer à ses grands grands oncles, puis à ses grands grands grands grands oncles s’ils existent etc. En faisant cet élagage, la vitesse d’execution c’est énormément amélioré

J’ai ensuite compris que pour que l’élagage soit réellement efficace, il faut ordonner l’ordre dans lequel on regarde les nœuds. Dans le Puissance 4 en 7\*6, la colonne 4 est souvent la meilleure pour jouer et les colonnes 1 et 7 les pires. On trouvera souvent le meilleur scénario en allant vers ce choix. L’élagage qui s’en suit est alors très puissant.

J’ai mis deux étapes de tri : j’ai d’abord une matrice qui renvoie à chaque position le nombre de fois qu’on peut faire un alignement différent à cette case, on regardera d’abord en fonction de ça. Ensuite on refait un tri pour qu’on regarde en priorité les colonnes où on crée des alignements de 3 jetons, car souvent cela nous aidera dans les prochains coups. J’ai dit plus haut qu’il n’y avait pas d’heuristique, ce qui n’est pas tout à fait vrai. En début de partie on ne calcul jamais assez de coup pour tout prévoir (on ne calcul par 42 coups) on va donc remonter pour toutes les colonnes un poids de 0. On va donc jouer dans la colonne qui sera en première étudiée. Le tri est donc une forme d’heuristique.

Une fois cela fait j’ai accéléré le processus en libérant des nœuds : dans l’élagage on se compare à tous ses oncles, dans le choix du poids parmi ses enfants, on regarde chaque enfant. Pour aller plus vite j’ai créé une méthode qui à chaque enfant créée, le compare à son frère sauvegardé. On gardera le meilleur des deux frères (en fonction de si on est en min ou en max). De cette manière un parent n’aura qu’un seul enfant et le choix du poids parmi ses enfants sera alors très rapide. Également pour l’élagage, chaque nœud n’aura qu’un seul oncle, qu’un seul grand grand oncle etc.

J’ai après ajouter une mémoire : grâce au bitboard, on ne fait pas de différence entre deux grilles en fonction de l’ordre dans lesquels les jetons ont été posés. Si on joue « 1 » , « 2 », « 3 » puis « 4 » ou « 3 » , « 2 », « 1 » puis « 4 », on va se retrouver avec exactement les mêmes grilles et donc on calculerait deux fois la même chose. On va donc enregistrer (à chaque tour) les différents couples (mask ; position) en clé d’un dictionnaire avec la valeur associée en value. De cette manière on évite de calculer énormément de branches.

Enfin la dernière modification et l’amélioration de l’élagage : mettons-nous dans un cas où on élague si on a une valeur inférieure à notre oncle. Jusqu’à maintenant supposons que notre oncle à une valeur de 0 et qu’on a une valeur de 1, on ne devrait pas élaguer car un de nos frère pourrait trouver une valeur de 2 ou plus. Or de la manière dont nous avons calculé notre poids : si on a une valeur de 1, on est sûr de gagner (du moins à ce niveau de l’arbre). Un poids de 2 nous permettrait juste de gagner en moins de coups.

Notre objectif est uniquement de gagner, pas de gagner avec le moins de coups. On peut donc élaguer dès ce moment-là et gagner pleins de temps de calcul.

Conclusion :

J’ai beaucoup aimé le projet