IN104

DE CHARENTENAY Melchior, GASTAUD Rémi

Mai 2021

1 Généralités sur le développement du projet

1.1 Débogage et méthodes de test

Les tests pour les algorithmes Minimax étaient fournis avec le paquet, ainsi il a été très simple de vérifier s'ils marchaient, et de les déboguer. Ce fût un peu plus compliqué pour le cas des fonctions d'évaluation et des "Brain". Pour ces derniers, si le code s'exécutait sans erreur, la méthode de test était empirique : en effet, le paquet aiarena effaçant le terminal à chaque tour de jeu, il a fallu tester à la main en jouant contre l'IA que nous avions implémenté, afin de vérifier que ses coups étaient raisonnables (était-elle capable de battre un joueur jouant n'importe comment, était-elle capable de jouer un coup gagnant, ou empêcher un coup perdant ?)

1.2 Optimisation

Afin de savoir sur quoi se concentrer et quoi améliorer dans notre programme, nous avons utilisé un profiler, qui nous fournit le temps consacré à chaque fonction dans le programme.

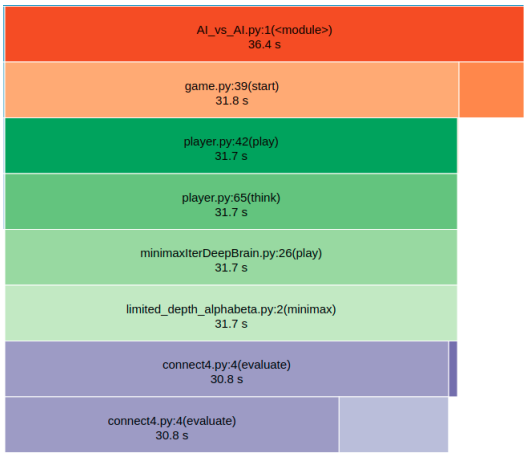

Figure 1: Répartition du temps de calcul pour un matchs entre deux IA

Ce qui apparaît clairement ici est que la fonction evaluate est celle nécessitant le plus de temps. Cela est cohérent: elle est appelé sur toutes les feuilles de l'arbre de jeu, et leur nombre croît exponentiellement avec la profondeur de l'arbre. Il sera donc important de faire non seulement une fonction pertinente, mais également une fonction efficace. Optimiser l'algorithme minimax risque d'être peu rentable étant donné qu'il est déjà bien optimisé avec la méthode alphabeta, et qu'il n'est responsable que d'une petite partie du temps total.

2 Implémentation des algorithmes

Peu de problèmes ont été rencontrés jusqu'à l'implémentation du principal algorithme : Minimax. Pour ce dernier, l'utilisation de la récursivité facilite grandement la tâche, même si un peu de reflexion a été nécessaire.

2.1 Elagage Alpha-Bêta

Dans le cas de l'élagage alpha-bêta, il est nécessaire de modifier la fonction en rajoutant un argument permettant de conserver une valeur particulière lors du parcours récursif. Python nous permet nous permet d'ajouter des arguments par défaut aux fonctions, afin que ceci soient sélectionnés si ils ne sont pas renseignés.

```
def minimax(node, maximize, get_children, evaluate, max_depth, parent_score = None):
```

Figure 2: Prototype de la fonction minimax avec elagage alpha beta

Dans notre cas la donnée à mémoriser est le minimum ou le maximum de la profondeur supérieure, identifiée ici comme "parent_score", par défaut à None,

c'est-à-dire qu'elle sera ignorée si on ne dispose pas de cette information. La fonction sans élagage passe les tests en à peu près 23 ms sur une configuration donnée, tandis qu'avec élagage, elle les passe en environ 18 ms. Le gain de temps est donc assez conséquent.

2.2 Exploration à profondeur maximale permise: minimaxiterdeepbrain

Pour cet algorithme, le concept était plutôt simple, la plus grosse difficulté a résidé dans l'utilisation du package alarm. Je n'ai pas trouvé la documentation officielle sur le package très claire d'où la difficulté initiale.
Une fois le package compris, il y a eu moins de problèmes. C'était important de penser à garder en mémoire le meilleur mouvement obtenu après une exploration à la dernière profondeur, car rendre le meilleur coup sans avoir explorer toute les possibilités n'a aucun sens.
Il est important de noter que bien que l'algorithme peut avoir l'air très peu efficace car il calcule à plusieurs profondeur qui ne servent à rien et on pourrait voir ça comme un gâchis de temps. Mais comme la complexité temporelle de l'algorithme minimax est exponentiel, le temps pour calculer tous les profondeurs précédentes est négligeable par rapport à celui nécessaire pour calculer la dernière. Donc l'économie de temps si on calculait directement à la dernière profondeur ne permet pas d'en calculer une nouvelle.

2.3 Fonctions d'évaluations

2.3.1 Fonction d'évaluation du puissance 4

Un bon algorithme minimax repose sur une fonction d'évaluation pertinente. Pour le puissance 4, celle choisie est assez intuitive, puisque nous allons évaluer tous les quadruplets de la grille de la manière suivante :

$$score_{grille} = \sum score_{quadruplet}$$

$$score_{quadruplet} = \begin{cases} 0 & \text{s'il y a des pions de couleurs differentes} \\ 10^{n-1} & \text{s'il n'y a que des pions blancs, avec n le nombre de pions} \\ -10^{n-1} & \text{s'il n'y a que des pions noirs} \end{cases}$$

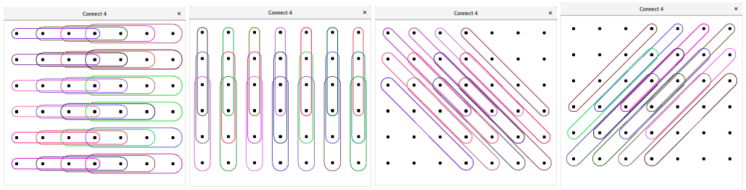

Figure 3: Quadruplets possibles au puissance 4

La figure ci-dessus montre tous les quadruplets possibles au puissance 4. Notre méthode va calculer 2 fois chacun d'eux car elle explore toutes les cellules et passe donc par les deux extrémités de tous les quadruplets. Ça n'influe pas sur les décisions de l'algorithme car tous les scores sont simplement multipliés par 2, et donc les comparaisons de sont pas affectées. Par contre le temps de calcul n'est pas optimal. Un essai a été effectué avec une fonction qui stocke les couples $(cellule, direction)$ qui sont inutiles à visiter, mais le parcours de cette liste rajoute en fait bien plus de temps de calcul : en effet la complexité passe de $O(n)$ à $O(n^2)$ où n est le nombre de cellules de la grille.

2.3.2 Fonction d'évaluation du jeu de dames

Après l'implémentation du premier algorithme d'évaluation où on se contentait de compter le nombre de pièce, nous avons cherché à créer un algorithme plus compliqué qui prenait en compte la position de chaque pièce sur l'échiquier lors de l'évaluation. Nous avons évalué chaque pièce de la manière suivante:

	Piece blanche	Piece noire
Pion normal	100	-100
Roi	200	-200
Pion sur le bord	5	-5
Pour un pion en (i,j)	i	nRows-i
Pour un roi en (i,j)	-i	i-nRows

Avec nRows la taille de l'échiquier

Ce type d'évaluation permet de pousser les pions à avancer et surtout pousse les rois à reculer. En effet en fin de partie, lorsque les règles ne permettent pas au rois de voler, il ne voit pas assez de coup en avance pour aller vers des pions ennemis et font donc des aller-retour ce qui provoque souvent des égalités.

3 Conclusions et pistes d'amélioration

Pour conclure, notre algorithme est plutôt efficace, il arrive à battre des joueurs avec de l'expérience au puissance 4 et aux dames. Pour autant il reste beaucoup d'amélioration à faire. Les plus notables et celles que nous n'avons pas eu le temps de faire sont:

- Les fonctions d'améliorations: lors de nos recherches nous avons découvert sur internet qu'il existe beaucoup de manière d'améliorer nos fonctions d'évaluations, surtout aux dames car le jeu est plus compliqué. On pourrait par exemple faire de nombreux essais afin d'obtenir le meilleur score à donner au roi qui ne veut pas forcément deux fois plus qu'un pion.
- L'optimisation temporelle: il est possible d'optimiser les algorithmes d'évaluations et de recherche de solution. Pour des jeux avec des arbre à solution simples comme les dames ou le puissance 4, la plus grande optimisation doit

se faire sur les algorithmes d'évaluation car d'après le profiler c'est aux qui prennent le plus de temps. Pour autant éviter de calculer les situations doublons lorsque l'on parcourt l'arbre est une piste d'amélioration de l'algorithme de recherche qui pourrait faire gagner beaucoup de temps si on arrive à le mettre en place.