Rapport IN104 : Implémentation d'une IA pour jeux d'Echecs, Dames, Puissance4 et Abalone

Anthony AOUN
Adrien BENNATAN

20 mai 2020

Table des matières

Dans le cadre du cours d'IN104, nous avons eu l'occasion de nous familiariser avec l'implémentation d'une IA pour les jeux de puissance4 et de dames. Afin de réaliser ceci, nous avons soigneusement suivi le plan des quatre séances que nous ne détaillerons pas ci-dessous.

Ce rapport présentera notre travail personnel qui est venu compléter le travail fait en classe. Ceci-dit, nous ne détaillerons pas le fonctionnement de minimax ou d'alphabeta en supposant qu'il s'agit du minimum à faire pour ce cours.

Nous allons donc commencer par décrire comment on trouve limited time et research time dans minimaxTimeBrain. Ensuite, nous allons montrer ce que nous avons rapporté en plus à la fonction d'évaluation de puissance4, pour ensuite expliciter le fonctionnement de l'algorithme négascout que nous avons implémenté. Plus encore, nous avons ajouté une table de transposition à négascout pour rendre sa performance assez raisonnable. Étant donné que alphabeta + table de transposition et négascout + table de transposition sont très proches au niveau de temps de calcul, nous allons faire plusieurs tests sur ces deux algorithmes qui se battent pour la première place, pour enfin dresser un tableau qui compare la vitesse de tous les algorithmes que nous avons implémenté.

Dans ce qui suit nous allons utiliser Python munie de la libraire aiarena.

1 Gestion du temps

Pour pouvoir retourner un coup à jouer avant la fin du temps imparti, on utilise le module time et on en définit une instance à chaque entrée dans `minimax`. On s'en sert ensuite pour distribuer les `maxTime` des minimax appliqués aux enfants du noeud considéré. En définissant l'instance au début de la fonction `minimax`, on s'assure de prendre en compte toutes les actions, même celles qui ne sont pas liées aux minimax appliqués aux enfants.
Le temps de recherche `researchTime` est calculé lors de l'instanciation de la classe `MinimaxBrain` en générant un état initial d'une partie correspondant au jeu considéré et en calculant la moyenne du temps pris pour l'exécution de `get_children` sur cet état.

En pratique, bien qu'elle fonctionne sur le `minimax` de base, cette technique ne s'adapte pas correctement aux algorithmes du `minimax` plus élaborés, pour des raisons que nous n'avons pu comprendre. Dans ces derniers cas, l'algorithme n'utilise pas tout le temps dont il dispose. En dernier recours, on a donc introduit un coefficient `coeff` qui a pour but de gonfler artificiellement le temps `timeLimit` donné à chaque enfant. Le `coeff` est fonction affine de `time`, dont la formule a été ajustée empiriquement pour utiliser le plus de temps possible sans jamais dépasser le temps imparti. On s'est aussi assuré que cela fonctionnait sur nos 2 ordinateurs, car le `coeff` optimal doit dépendre de la machine sur laquelle tourne l'algorithme

2 Fonction d'évaluation de puissance4

En ce qui concerne les fonctions d'évaluations, nous avons décidé de garder celle du jeu de dames assez simple (renvoie la différence entre le nombre de pirèces de l'IA et celui de son adversaire) pour diminuer son temps de calcul et permettre à l'IA de descendre plus loin dans l'arbre de recherche.

Cependant, pour la fonction d'évaluation de puissance4, nous avons choisi d'améliorer sa performance même si cela veut dire aller moins loin dans l'arbre de recherche. Dans ce cas nous

favorisons la qualité de recherche plutôt que le nombre d'états atteints dans l'arbre.

Cette fonction d'évaluation couvre toutes les possibilités :

— Si l'IA est gagnante, la fonction renvoie 100

— Si l'IA est perdante, la fonction renvoie -100

— Si Le match est nul, la fonction renvoie 0

— Si la partie n'est pas encore terminée, la fonction renvoie la différence entre le nombre de 3 pièces blanches allignées et le nombre de 3 pièces noires allignées. Ceci pousse l'IA à maximiser ses chances de gagner plus tard.

3 Algorithme Négascout

Négascout est un algorithme qui peut être plus rapide qu'Alphabeta. Cet algorithme ne va jamais chercher à visiter un état qui aurait été éliminé par Alphabeta puisqu'il vérifie à chaque itération si $\alpha > \beta$, mais son avantage réside en éliminant plus de possibilités.

Négascout effectue un parcours en largeur et calcule à chaque niveau le meilleur choix à faire. Cela est réalisé en supposant que le premier état visité est le meilleur mais reste à vérifier cette hypothèse par la suite. Cette hypothèse est vérifiée en parcourant les autres états avec ce qu'on appelle une fenêtre nulle (un scout nul) ; c'est à dire $\alpha = \beta$. Si l'hypothèse de base échoue, on passse au noeud suivant et ainsi de suite comme dans Alphabeta.

Normalement Négascout est plus efficace quand les états sont ordonnés du meilleur au pire ; ce qui n'est pas forcément le cas ici. C'est pour cette raison qu'on dit que Négascout peut être plus efficace qu'Alphabeta, et ceci si les meilleurs états sont par par chance au début de la liste des enfants de l'état actuel.

4 Addition d'une table de transposition à Négascout

La table de transposition permet de stocker les états (gameState) déja visités pour éviter de les revisiter étant donné que nous connaissons déja leur poids. Dans notre cas nous l'avons implémenté grâce à un dictionnaire et à chaque itération nous vérifions si l'état est déja dans cette table. Si oui nous réccupérons directement sa valeur (au lieu de visiter tous les sous-états de cet état) et continuons normalement, si non nous sommes dans le cas normal où il faut calculer récursivement le poids de cet état.

En ajoutant la table de transposition à Négascout (et naturellement à Alphabeta aussi), nous arrivons à une vitesse de calcul assez élevée qui sera détaillée ulterieurement en comparant la vitesse de tous les algorithmes.

5 Tests et comparaison des vitesses des différents algorithmes

5.1 Comparaison des moyennes de alphabeta_tt et negascout_tt

Nous faisons 20 tests á chacun des algorithmes : Alphabeta_tt et Negascout_tt. Chaque test consiste à lancer une partie AI_vs_AI puissance4 pour une profondeur égale à 5.

Après avoir fait ces tests et récupéré les durées des parties, nous représentons les résultats sur le graphe ci-dessous.

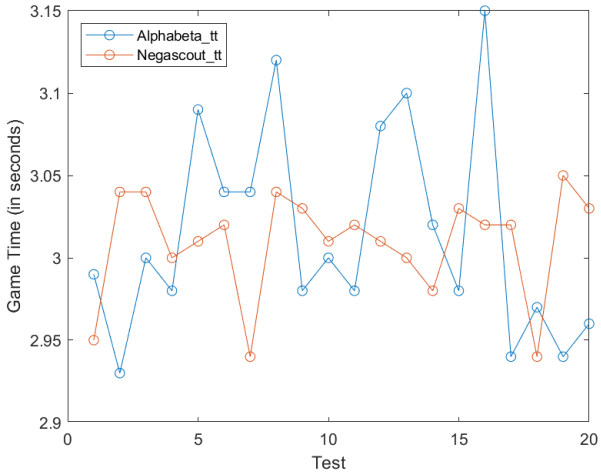

FIGURE 1: Graphe comparant le temps mis par chacune des 20 parties AI_vs_AI puissance4 pour les algorithmes Alphabeta_tt et Negascout_tt et cela pour une profondeur égale à 5

Nous calculons de même l'espérance de chaque algorithme et nous tirons les valurs suivantes :

$$E[Alphabeta\_tt] = 3.0145s$$

$$E[Negascout\_tt] = 3.0090s$$

Nous pouvons donc conclure qu'Alphabeta_tt est légèrement plus lent que Negascout_tt en moyenne. Nous califions donc Negascout_tt pour participer à la compétition du projet IN104.

5.2 Tableau de comparaison

Pour marquer l'évolution de nos algorithmes tout le long de ce projet, nous comparons leur performance dans le tableau ci-dessous :

Algorithme	Durée (s)
limited_depth	186.47
limited_depth_tt	48.36
limited_depth_alphabeta	6.09
limited_depth_negascout	5.80
limited_depth_alphabeta_tt	3.01
limited_depth_negascout_tt	3.00

TABLE 1: Comparaison du temps pris par une partie AI_vs_AI puissance4 à profondeur égale à 5

Nous remarquons un gain énorme en rapidité de calcul quand on passe du minimax normal á Alphabeta ou Négascout. Cependant il devient de plus en plus dur de réduire ce temps de

calcul lorsqu'on s'approche du temps optimal.

En ce qui concerne l'optimisation, nous étions très attentifs à toujours utiliser la méthode la moins coûteuse en temps de calcul (récursivité) et de ne pas appeler des fonctions quand ceci est non nécessaire.