TP1 Compte Rendu IA

Johan Gras

2)

En plus des structures nécessaires aux jeux et à l'IA j'ai implémenté une structure de données pour « débugger » les algorithmes. J'ai ainsi mis en place une liste chainé qui permet alors de remonter depuis les nœuds inférieurs l'ensemble de coups qui à amener minmax/alphabeta à jouer le coup actuel. Cela m'a donc permis d'analyser et debugger plus facilement l'heuristique.

3)

Pour choisir une fonction d'évaluation correct j'ai dû réaliser plusieurs essais en passant par des mixte de maximisations de nombres de pions par rapport à l'adversaire ou minimisation de la distance par rapport à la ligne ennemie.

Les techniques tenant en plus compte de la valeur des unités ne sont pas facile à mettre en place et pas nécessairement très indicatives.

Au final mon heuristique finale tiens compte de la distance moyennes des pions du joueur jusqu'à l'arrivé (moins celle de l'advairsaire) : plus les pions sont avancés dans le camp adverse plus l'heuristique va renvoyer une valeur élevé et vice-versa. Cette technique à plutôt bien marché, par rapport à sa variante ne tenant compte que du pion le plus avancé, elle permet de faire avancer les pions « en groupe ». Aussi elle permet de tenir compte d'une certaine manière du nombre de pions restant : si un des pions en avant se fais manger, un recul de la moyenne aura lieu et l'heuristique en tiendra compte. Par contre il est vrai que si des pions à l'arrière se font manger l'effet inverse aura lieu mais ce genre de situation est nettement plus rare.

Si un des joueur à gagner, l'heuristique renverra un score maximal/minimal tout en tenant compte de la profondeur actuel (c'est mieux de gagner au tour d'après que dans 5 tours).

Bien sur de nombreuses améliorations de l'heuristiques peuvent être faites comme : tenir compte de l'avancé du jeu, de la valeur des pions, de leurs disposition spatiale, de pattern, bibliothèque d'ouverture/fermeture,...

4)

Contrairement à la version min-max, la version alpha-beta m'a posé bien plus de problème pratique. Elle ne se comportait pas comme sa version classique, le jeu ne finissais pas dans la même position finale. C'est à ce moment-là que j'ai construit une liste chainé à la main pour se souvenir des coups choisis.

Pour corriger le problème j'ai par exemple du utiliser un autre infini que les constantes INT_MIN et INT_MAX (de limit.h) car lorsqu'on les inverse on ne tombe pas sur les valeurs algébrique attendu (INT_MIN étant plus petit d'une valeur).

J'ai aussi du initialiser alpha beta avec des valeurs légèrement plus grande/petite que les valeurs maximum/minimum renvoyé par l'heuristique car si on prend une situation ou un joueur est sur de perdre (tous les coups qu'il peut joueur lui amène à une situation de perdant) alors l'algorithme ne va jamais sélectionner de coup car la condition valeur > alpha ne sera jamais rempli. Mais si alpha est initialisé avec une valeur inférieur alors un coup légal sera quand même joué.

Enfin j'ai tenu compte de la profondeur dans l'heuristique lors d'une situation gagnante/perdante. Car ce qu'il se produisait sinon c'est que un joueur pouvais se trouver dans une situation gagnante à coup sûr (ex : à une ligne de l'arrivé, pas de risque de se faire manger et de perdre) de ce fait toutes les valeurs renvoyés par l'heuristique étant maximales. Un coup au « hasard » étant alors joué, de ce fait il était fort possible que le joueur ne joue pas le coup gagnant en « le retardant ».

5.1)

On peut tenter de faire une approche approximative du nombre d'opération moyenne pour un coup quelconque.

Pour l'estimation on négligera les opérations pour créer l'arbre alpha/beta ou min-max, on évaluera seulement la complexité au niveau des nœuds.

Ensuite on remarquera pour évalue un nœud que la complexité la plus importante se trouve dans la double boucle qui parcours toutes les cases du jeu à quelques conditions, affectations et lectures près.

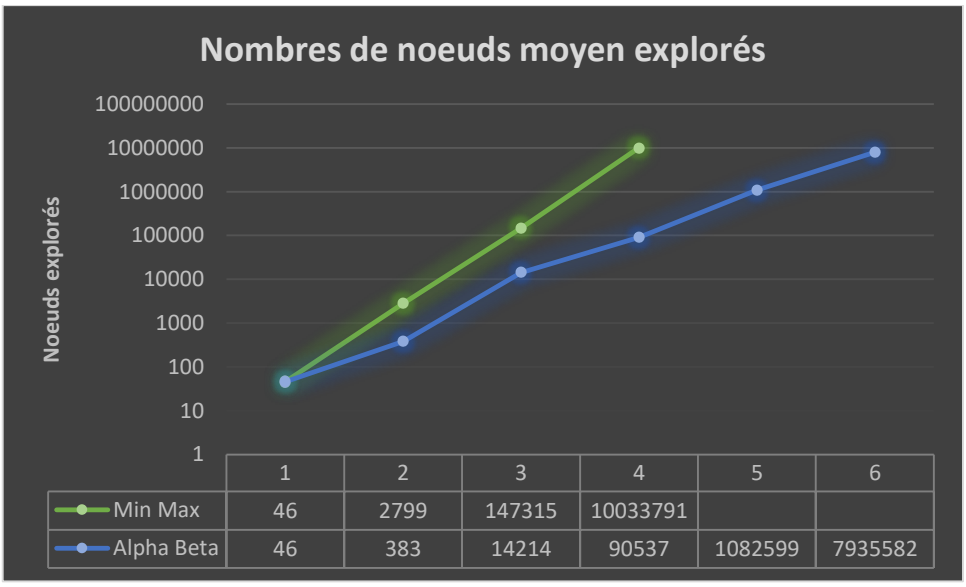
Mais on peut retenir que la complexité en nombres d'opérations d'un nœud à pour ordre de grandeur O(n) avec n le nombre de cases (ici 100).

On pourrait compter exactement le nombre d'instructions et leurs types mais cela n'aurais pas grand intérêt.

Ensuite dans le cas de min-max on a une complexité théorique en terme d'analyse de nœuds de $O(b^d)$ avec b le facteur de branchement moyen et d la profondeur d'analyse. D étant une constante définis et b étant variable selon l'évolution du jeu (il peut être calculé ainsi : nombres de pions du joueur * nombre de degré de liberté moyen du joueur). Ainsi pour avoir le nombre d'opérations moyen pour évaluer un coup on calcul l'ordre de grandeur de nœuds évalués (b^d) fois le nombre d'opérations par évaluation de nœuds (100 * constante d'opérations).

Pour alpha-beta on à comme complexité dans le pire des cas celle de min-max (soit $O(b^d)$ ou au mieux $O(\sqrt{b^d})$.

5.2)



Voici le graphique avec l'évolution du nombre de nœuds moyens explorés par alphabeta et minmax en fonction de la profondeur. Pour y voir clairement l'échelle est logarithmique. Ces valeurs ont étaient calculés en fessant la moyenne sur une partie entière d'IA vs IA.

On remarque très clairement que pour la même profondeur alpha beta évalue beaucoup moins de nœuds et plus la profondeur est grande plus l'écart se creuse (profondeur 4, facteur 100 de différence).

Nous avons plus voir plus haut que min-max à une complexité $O(b^p)$ et alpha-beta à la même complexité dans le cas où il ne fais aucune coupure et $O(\sqrt{b^d})$ si le maximum de coupure est réalisé.

Le nombre moyen de nœuds explorés étant calculé sur une exécution entière d'une partie IA vs IA. L'exécution d'alpha beta est strictement égale à min max en termes de résultat et de l'ordre d'exécution des branches (excepté celles coupés), les algorithmes étant complètement déterministe (ils produisent la même exécution à chaque lancement), alors on peut ainsi en conclure que l'exécution de minimax ou alphabeta à profondeur p aura exactement le même facteur de branchement moyen b lors d'une exécution (pour alpha beta on parle bien facteur de branchement sans coupure).

Ainsi on peut se rendre compte en analysant le gain expérimental de alphabeta sur n'importe quel profondeur qu'il est effectivement bien supérieur à celui de minmax mais aussi loin du gain maximum (racine carré).

Pour que notre programme devienne plus performant à profondeur constante on pourrait soit améliorer la performance de l'algorithme de recherche en effectuant une recherche à profondeur 1 puis 2, 3, 4,… jusqu'à la profondeur voulu et en réordonnant à chaque fois les coups dans l'ordre qui est le plus susceptible de produire des coupures ; cette amélioration est au final pas si couteuse que sa car seule la dernière recherche est très couteuse, de plus on utilise souvent en parallèle une table de transposition qui se souviens des nœuds déjà calculés et de ce fait booste encore les performances.

On peut aussi utiliser des techniques « d'aspiration » où la fenêtre de recherche alpha beta est réduite, « null-window search » poussant la technique à l'extrême en initialisant alpha = beta.

On peut aussi apporter des améliorations au niveau de l'heuristique : tenir compte de la valeur des pions, de leurs positions, de patterns,… Mais il faut en général faire beaucoup de tests.

On peut aussi mettre en place des réseaux de neurones, des bibliothèques d'ouvertures et fermetures…