Grenoble INP - Ensimag
Deuxième année filière ISI

Algorithme Avancée

Rapport Projet Blobwar

Author
Ryan Liing Pang Tee

23 avril 2021

Table des matières

1 Introduction

Ce rapport présente les différentes algorithmes implémentés dans le jeu *blobwar* ainsi que leur complexités et optimisations.

2 Algorithme Glouton (Greedy)

Greedy est un algorithme qui suit le principe de faire, étape par étape, un choix optimum local, dans l'espoir d'obtenir un résultat optimum global. Etant donné un certain état du jeu (`state`) on parcourt tous les mouvements possibles du joueur et on calcul, pour chacun d'entre eux, la valeur du jeu (`state.value()`), et on choisit celui qui donne le minimum, puisque la valeur indique le nombre de blobs qu'on a en moins par rapport à l'adversaire.

2.1 Variations

Il y a deux variations de cet algorithme. La différence vient du fait dans un état donné, il peut y avoir plusieurs choix optimums (de même valeur).

Greedy Simple C'est la fonction de base `Greedy()` où on choisit le première choix optimum.

Greedy Probabiliste Dans la fonction `RandomizedGreedy()`, on récupère tous les choix optimum (s'il y en a plusieurs) dans un vecteur et on en choisit un de manière aléatoire.

2.2 Performance

On fait 1000 test de l'algorithme `Greedy()` contre le `RandomizedGreedy()`. Le résultat est présenté dans le diagramme suivant. En général, la différence entre ces 2 algorithmes n'est pas très significative. Dans le côté `RandomizedGreedy()`, l'aspect probabiliste ne joue que si on avait plusieurs choix optimum, et plus ce dernier est nombreux plus on aura un meilleur résultat, ce qui n'est pas le cas ici. Si on avait une carte plus large (16x16 ou même 32x32 par exemple), on peut s'attendre à voir une plus grande amélioration de `RandomizedGreedy()`.

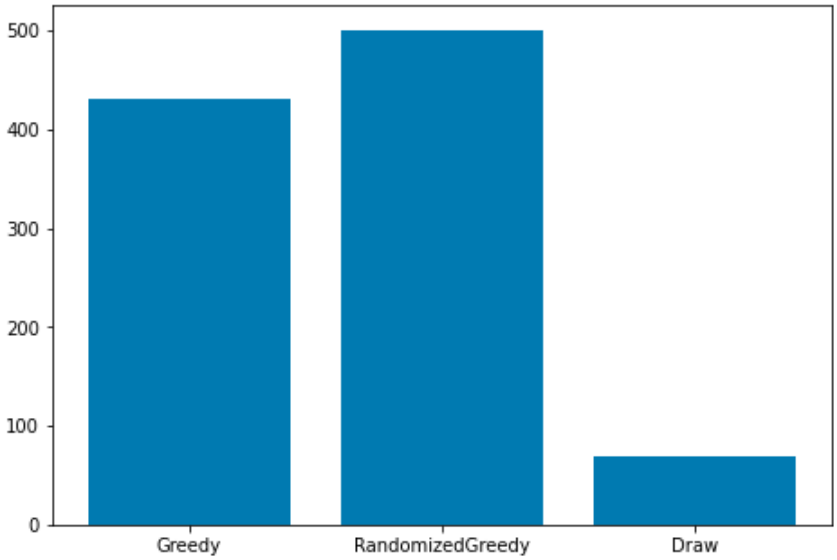

Figure 1 – 1000 Test de Greedy vs RandomizedGreedy

Dans la suite, on va utiliser `RandomizedGreedy()` pour faire la comparaison avec d'autres algorithmes.

3 Minimax

Dans `Minimax()`, on se distingue 2 parties : le *minimizer* (nous) et le *maximizer* (l'adversaire). Le *minimizer* cherche toujours à minimiser notre valeur du jeu, et le *maximizer*, quant à lui, cherche à maximiser cette même valeur. (Puisqu'on gagne si à la fin on a une valeur du jeu négative).

Le principe est le suivant : L'algorithme réalise une évaluation de la position courante, représentée par la racine de l'arbre de jeu, en partant des nœuds terminaux. Dans ce but l'ensemble des nœuds de l'arbre de jeu est divisé en deux classes : les nœuds de *Maximizer* sur les niveaux où ce joueur va choisir un coup et les nœuds *Minimizer* pour les niveaux de décision du joueur MIN (nous).

Prenons comme exemple un état où on a 2 choix possibles pendant ce tour : **Duplicate** et de valeur -1 et **Jump** de valeur 2. Pour simplification, supposons également que chacun de ces choix ont aussi 2 choix (noeuds fils). Cet exemple est présente dans la Figure 2.

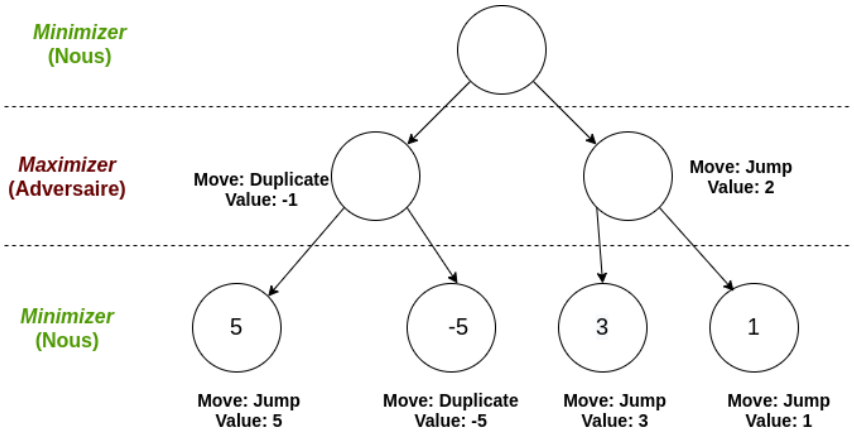

Figure 2 – Exemple de l'arbre de jeu avec profondeur = 2

L'algorithme Greedy a tendance à choisir le noeud **Duplicate** puisque le choix a une meilleure valeur du jeu (-1 < 2). Or si on suppose que notre adversaire (le *Maximizer*) va toujours choisir le choix optimal pour lui (càd de maximiser notre score), pendant le tour suivant il va choisir **Jump** de valeur 5.

L'algorithme `Minimax`, quand à lui, va parcourir récursivement tous les choix possibles jusqu'à la limite de profondeur (dans l'exemple le profondeur est 2), et choisir le noeud qui lui permet d'avoir une meilleure valeur finale. Dans ce cas, c'est le noeud **Move**, et on voit dans la figure suivante que c'est effectivement le choix optimal.

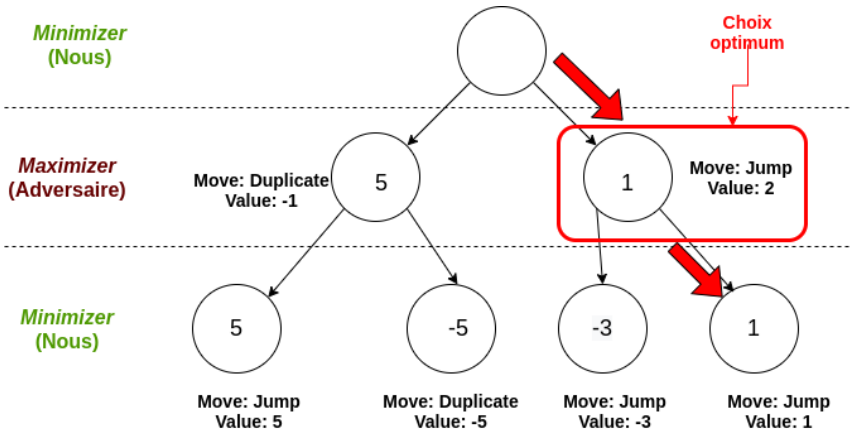

Figure 3 – Le choix optimum est celui de Jump

3.1 Performance

Pour la performance, L'algorithme `MiniMax` avec seulement un profondeur de 2 est largement supérieure à `RandomizedGreedy`. Sur un test de 500 jeux, `MinMax` a gagné 486 fois, 8 victoires pour `RandomizedGreedy` et 6 égalité.(Figure 4).

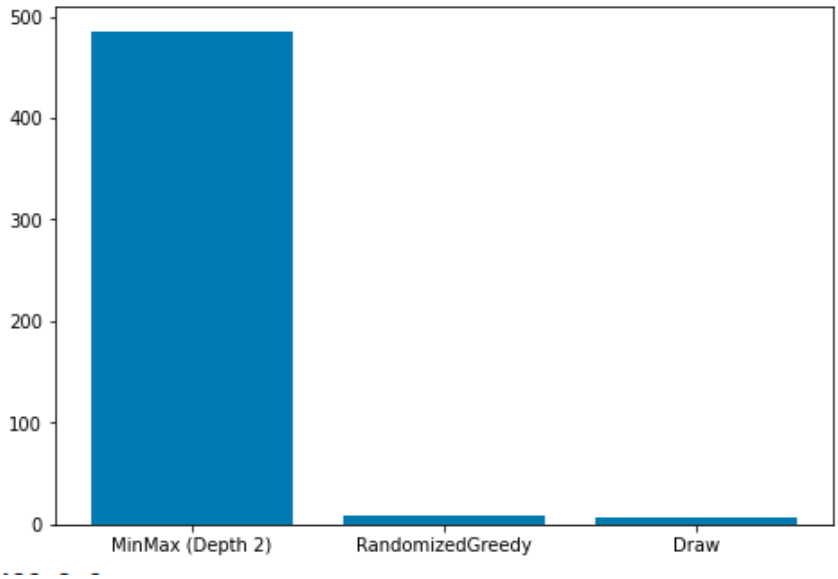

Figure 4 – 500 Test de MinMax depth 2 vs RandomizedGreedy

Un point négative de cet algorithme est son temps d'exécution. Comme on parcourt tous les résultats possibles pour chaque noeud, dans le pire cas on aura :

$$W = O(c^p) \tag{1}$$

où c est le maximum nombre de choix possible et p le profondeur de l'arbre. Donc plus le profondeur est élevé plus le temps d'exécution est important :

Profondeur	Temps d'exécution en moyenne de `compute_next_move` (en ms)
2	1.5969640681818185
3	150.4038493043478
4	2243.137100769231

On va améliorer cet algorithme avec une méthode d'élagage de l'arbre, présenté dans la partie suivante.

4 Alpha-bêta

L'algorithme d'élagage `Alpha-Beta` est une version amélioré de `Minimax`. Il conduit à une économie de temps de calcul et mémoire, en renonçant à l'évaluation des sous arbres dès que leur valeur devient inintéressante pour le calcul de la valeur associée aux nœuds père de ces sous arbres. Prenons un exemple similaire à celui de section de `Minimax` (Arbre de jeu avec profondeur = 2, chaque noeud a 2 choix possibles, cf Figure 5). Ici on suppose que l'algorithme a parcouru tous sauf le dernier noeud.

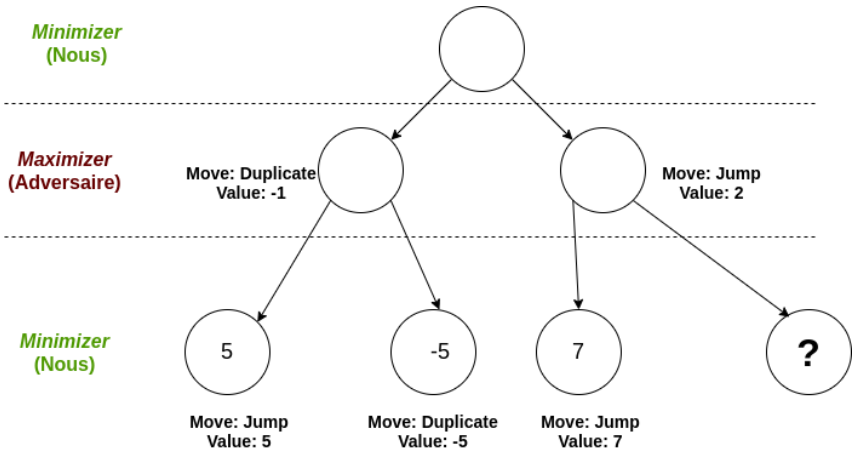

Figure 5 – Exemple de l'arbre de jeu avec profondeur = 2. Il reste un dernier noeud à explorer.

a jusqu'à présent (5) avec celui qu'on vient de trouver (7). Comme l'adversaire (*Maximizer*) va toujours choisir celui le plus grand, on est certain qu'on ne pourra pas avoir une valeur meilleure (moins) que 7. On ignore donc le noeud final et choisit le noeud de gauche.

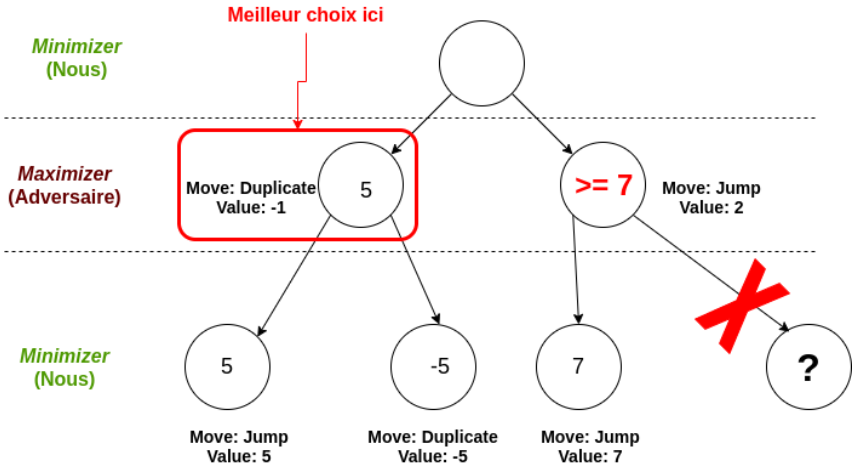

Figure 6 – Pas la peine d'explorer le noeud final (élagage)

4.1 Performance

Dans le pire des cas, il n'y a pas d'élagage possible, donc le travail est identique à `Minmax`. Dans le meilleur des cas, on explore un seul noeud à chaque niveaux (càd on choisit toujours le meilleur noeud en premier), on a :

$$W = O(b^{\frac{d}{2}}) \tag{2}$$

La mesure du temps d'exécution montre une amélioration très importante :

Profondeur	Temps d'éxecution en moyenne de la fonction `compute_next_move` (en ms)	
	`Minimax`	`Alpha-Beta`
2	1.5969640681818185	0.19881555319148939
3	150.4038493043478	2.3855502727272726
4	2243.137100769231	18.62468974

4.2 Optimisation avec Memoisation

Comme il est possible d'atteindre une même position par plusieurs chemins différents, on pourrait stocker les positions et leur valeurs dans un table de hachage. On introduit donc une nouvelle fonction `alphabeta_memo` similaire à celui d'avant, mais qui en plus prend en paramètre un table de hachage. Le table prend comme clé une chaîne de caractères représentant l'état du jeu sérialisé, et la valeur est un tuple (valeur, choix optimal).

Plus le profondeur augment, plus l'amélioration est significative :

Profondeur	Temps d'éxecution en moyenne de la fonction `compute_next_move` (en ms)	
	`AlphaBeta`	`AlphaBeta_Memo`
2	0.19881555319148939	0.1848736274509804
3	2.3855502727272726	3.6706067068965496
4	18.62468974	20.30303963043478
5	618.8734534693878	546.7893422173913
6	3522.7713463703712	1378.5317213636363
7	61705.06743333333	39640.71084285714

Plus l'arbre du jeu est grand, plus c'est probable d'atteindre un état déjà rencontré.

4.3 Optimisation avec tri

Implémenté par la fonction `alphabeta_memo_sort`. L'idée est que dans un niveau donné, au lieu de parcourir tous les noeuds fils, on va explorer en priorité ceux avec des meilleures valeurs, puisque l'un de ces derniers sont plus probables d'être le choix optimal. Même si ce n'est pas toujours le cas, comme on a pu voir dans les exemples précédentes (Figure 3), la plupart du temps, on a moins de chance de tomber sur le choix optimal en explorant un noeud avec une valeur du jeu locale plus grande qu'un noeud avec une valeur locale plus petite (donc meilleure). Pour cela, on va trier les noeuds selon leurs valeurs du jeu décroissante (de plus petit au plus grand).

5 Conclusion

Pour le jeu *blobwar*, l'algorithme `Alphabeta` est sans doute le meilleur algorithme qu'on peut avoir en terme de précision de la prédiction et aussi le temps d'éxecution. Il existe d'autre possibilités qui sont tous aussi intéressant et qui dans certains cas s'avèrent être plus rapide que l'élagage alpha-beta, par exemple la recherche arborescente Monte-Carlo ou bien le NegaScout (Principal Variation Search).