Algo Avancée - Rapport de projet

WILLEM Logan

Table des matières

1 Algorithmes implémentés

Cette section présente rapidement les trois algorithmes imposés pour la résolution du jeu de blobwar. Elle est de plus complétée par un 2 autres algorithmes (non fonctionnels...).

1.1 Algorithme Glouton

Cet algorithme est implémenté dans le fichier `src/strategy/greedy.rs`.

Comme son nom l'indique, il s'agit d'un algorithme glouton qui consiste donc à maximiser les gains à court terme en prenant les meilleures décisions possibles à chaque étape. Ici, il s'agit de maximiser le gain à chaque tour en choisissant le mouvement qui apporte le plus de points à court terme, sans prendre en compte les gains futurs ou les mouvements adverses. L'intérêt de cet algorithme reste minime de par la qualité médiocre des résultats qu'il fournit. Cet algorithme est volontairement non parallèle pour éviter des surcoûts inutiles.

1.2 Algorithme Minmax

Cet algorithme est implémenté dans le fichier `src/strategy/minmax.rs`.

Une fonction annexe est utilisée : `minmax_rec`. C'est une fonction récursive qui applique l'algorithme MinMax pour évaluer les états jusqu'à une profondeur donnée. Elle prend en entrée un `depth` qui correspond à cette profondeur, un booléen `player` qui indique si c'est au tour du joueur actuel de jouer (`true`) ou pas (`false`), ainsi que l'état courant du jeu `state`.

La méthode `compute_next_move` elle parallélise les appels à `minmax_rec` pour chaque mouvement possible dans l'état courant et associe à chaque mouvement le couple `(valeur, mouvement)`. La méthode `reduce_with` permet enfin de choisir le meilleur couple (donc celui avec la plus grande valeur). La méthode retourne le mouvement correspondant donc à la meilleure valeur trouvée.

Le parallélisme ne se fait qu'au niveau de l'appel à `minmax_rec` et non pas après dans la fonction même. En effet, la profondeur maximale envoyée à MinMax est de 4 si on veut avoir des résultats rapides. Cependant, j'ai remarqué empiriquement que le surcoût de parallélisation était plus élevé que l'évaluation en elle-même pour des profondeurs de 2/3. Ainsi paralléliser la fonction récursive n'aurait pas vraiment d'intérêt.

1.3 Algorithme Alphabeta

Cet algorithme est implémenté dans le fichier `src/strategy/alphabeta.rs`.

Il s'agit d'une extension de l'algorithme MinMax qui permet de réduire le nombre de nœuds évalués en éliminant les branches qui ne contribueront pas à la décision finale.

Tout comme pour MinMax, une fonction annexe est utilisée : `alpha_beta_rec`. Il s'agit également d'une fonction récursive qui implémente l'algorithme Alpha-

Beta. Elle prend en entrée la profondeur actuelle `depth`, les bornes `alpha` et `beta`, le booléen `player` et l'état actuel `state` du jeu.

La méthode `compute_next_move` utilise la même logique que pour MinMax en utilisant la parallélisation et la réduction des résultats obtenus après application de l'algorithme pour chaque mouvement.

Le parallélisme se fait ici aussi qu'à l'appel de la fonction récursive et non pas à l'intérieur de celle-ci à cause des dépendances sur les variables `alpha` et `beta`.

1.4 Algorithme Glouton randomisé

Cet algorithme est implémenté dans le fichier `src/strategy/greedy_rand.rs`.

J'ai voulu ici faire en sorte de choisir non pas toujours le premier mouvement ayant le meilleur gain mais plutôt choisir un mouvement aléatoire parmi tout ceux maximisant le gain à court terme. Malheureusement, cette approche ne donne pas particulièrement de plus-value. Tout comme pour l'algorithme glouton, l'implémentation du parallélisme n'est pas particulièrement pertinent.

1.5 Algorithme de Monte-Carlo

/!\ Cet algorithme ne fonctionne pas du tout, je n'ai pas réussi à tirer de lui ce que je désirais...

Cet algorithme est implémenté dans le fichier `src/strategy/montecarlo.rs`.

Il s'agit d'une implémentation de l'algorithme Monte-Carlo. Il simule un grand nombre de parties aléatoires (c'est à dire jusqu'à la fin du jeu avec l'utilisation de la méthode `game_over`) à partir de l'état actuel du jeu, en choisissant des mouvements aléatoires pour chaque joueur, puis en utilisant la valeur de chaque résultat pour déterminer le meilleur mouvement possible.

Contrairement à MinMax ou AlphaBeta, cet algorithme prend en entrée la variable `max_iterations` qui détermine le nombre de simulations qui seront effectuées pour chaque mouvement possible.

À chaque itération, on calcule le score final et on l'ajoute à la somme totale des scores pour le mouvement en question. À la fin de toutes les itérations, on divise la somme totale par le nombre d'itérations pour obtenir le score moyen. C'est par ce score moyen que le mouvement suivant sera sélectionné.

Ici aussi le parallélisme est utilisé.

2 Évaluation des performances

2.1 Méthode d'évaluation

Deux scripts ont été créés pour l'évaluation des performances :

— `temps_exec.sh` permettant de calculer le temps d'exécution moyen des algorithmes. Pour que l'algotithme marche, il faut veiller à n'avoir aucune erreur affichée par cargo. L'usage correct de cette commande est `temps_exec.sh <number>` avec `<number>` correspondant au nombre d'exécution de l'algorithme.

— `pourcentage_victoire.sh` permettant de calculer le taux de victoire moyen de la première stratégie sur la deuxième. L'usage correct de cette commande est `pourcentage_victoire.sh <number>` avec `<number>` correspondant au nombre d'exécution de l'algorithme.

Notons que pour les deux scripts, il faut changer en amont les algorithmes dans `main.rs`.

De plus, les profondeurs choisies pour les stratégies MinMax et AlphaBeta ont été fait de manière empirique : ces stratégies restaient efficaces en temps d'exécution pour une profondeur respectivement de 4. L'algorithme de Monte-Carlo quant à lui n'a pas été évalué puisqu'il ne fournit jamais de résultats satisfaisants.

2.2 Résultats

2.2.1 Temps moyen d'exécution

Ci-dessous est repertorié le temps moyen d'exécution pour chacune des stratégies. Pour chacune d'entre-elles, la stratégie adverse considérée est la même. Ainsi, le temps moyen repertorié est le temps pris par la même stratégie jouant deux fois et non pas le temps propre que prend l'algorithme. L'algorithme n'est pas présenté ici, choix qui s'explique facilement dans la sous-section suivante.

	Temps d'exécution moyen (en s.)	
Stratégie	Algo non parallèle	Algo parallèle
Greedy	0.103	/
GreedyRand	0.107	/
MinMax(4)	298	43.7
AlphaBeta(5)	80.0	44.6

TABLE 1 – Temps d'exécution moyen des différentes stratégies

On voit une nette amélioration du temps d'exécution que ce soit pour la

stratégie MinMax ou AlphaBeta entre l'algorithme parallèle et non parallèle. On voit également une amélioration de la profondeur pour ces deux stratégies pour un temps d'exécution relativement similaire en utilisant l'algorithme parallèle.

2.2.2 Ratio de victoires

	% victoire (contre)		
Stratégie	Greedy	MinMax(2)	MinMax(4)
Greedy	/	0%	0%
GreedyRand	46%	0%	0%
MinMax(4)	100%	100%	/
AlphaBeta(5)	100%	100%	100%
MonteCarlo(100)	0%	0%	0%

TABLE 2 – Pourcentage de victoire des stratégies en fonction de l'adversaire

Plusieurs remarques sur ce tableau :
- On remarque bien comme dit précédemment que les stratégies GreedyRand et MonteCarlo ne fournissent peu voire pas du tout de résultats satisfaisants, la stratégie Greedy étant déjà meilleure.
- On peut classer ces stratégies donc par ordre d'efficacité :
  MonteCarlo < GreedyRand <= Greedy < MinMax(4) < AlphaBeta(5)
- On remarque aussi que non seulement, comme vu dans la sous-section précédante, l'algorithme MinMax de profondeur 4 s'exécute aussi vite que l'AlphaBeta de profondeur 5, mais on voit maintenant en plus que celui-ci gagne à tout les cas. Donc cette stratégie est celle à considérer.