Algorithmique Avancée:
Compte rendu du projet

Équipe 25

Avril 2023

Résumé

Compte rendu du projet Algorithmique Avancée de l'année 2022-2023, implémentant différentes stratégies du jeu blobwars.

Introduction

Ci-dessous, résumé des optimisations et performances des différents algorithmes implémentés pour le jeu blobwars.

1 Minmax

1.1 Algorithme de base

Le premier algorithme que nous avons implémenté est la version simple du Minmax.
L'algorithme Minimax est une technique couramment utilisée dans les jeux à deux joueurs, tels que les échecs, le morpion ou le blobwars.

Le fonctionnement de l'algorithme est relativement simple. Il utilise une approche récursive pour explorer l'arbre de jeu complet, en évaluant chaque coup possible pour chaque joueur, jusqu'à atteindre un état final du jeu ou la profondeur visée si atteinte.
Chaque état final sera ensuite évalué par une fonction d'évaluation qui lui attribuera une valeur numérique indiquant le gain du joueur actif.

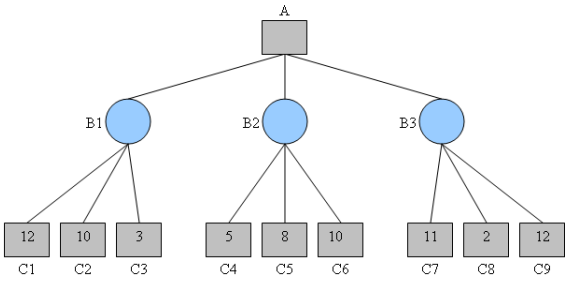

Figure 1 – Algorithme Minmax - Wikipedia

Pendant la recherche du meilleur coup, on va alterner les joueurs pour chaque niveau exploré. À chaque tour, le joueur actif cherchera à maximiser la valeur de son coup, tandis que l'adversaire cherchera à minimiser cette même valeur.
Finalement, à la racine de l'arbre on prendra le maximum des valeurs minimales de tous les coups possibles, pour maximiser le gain de notre joueur.
Le seul conflit avec le code, était que la fonction d'évaluation des états finals ne calculait pas le gain mais la perte, donc au lieu de maximiser les valeurs minimales on a minimisé les valeurs maximales.

1.2 Minmax Parallèle

Le problème principale avec l'algorithme Minmax, est qu'il faut explorer tout l'arbre des possibilités. Cela peut se passer relativement bien avec des jeux comme le morpion, qui n'ont pas beaucoup de coups possibles à calculer, mais l'algorithme devient rapidement laborieux pour des jeux aux possiblités élevées comme les échecs.
Pour le blobwars, nous n'arrivions à regarder uniquement 4 coups à l'avance. En effet, avec une profondeur supérieure ou égale à 5, le calcul était beaucoup trop complexe, d'où la nécessité de paralléliser.
Ici, à l'aide de Rayon, nous avons pu implémenter une version parallèle du Minmax, pour explorer l'arbre de jeu plus vite. Ceci est possible parce que, entre deux différents noeuds, il n'y a pas d'intersection ou de dépendance.
La parallélisation était donc rélativement simple une fois que nous avions compris comment utiliser Rayon, mais cela n'a pas amélioré la performance comme nous le verrons plus tard.

2 Alphabeta

2.1 Algorithme de base

L'algorithme Alpha-Bêta est une technique d'optimisation de l'algorithme Minimax, qui permet de réduire le nombre de nœuds explorés lors de la recherche de la meilleure décision à prendre.

L'algorithme Alpha-Bêta supprime les branches de l'arbre de recherche qui ne peuvent pas influencer la décision finale. Pour se faire, l'algorithme utilise deux valeurs appelées "alpha" et "bêta", qui représentent les bornes inférieures et supérieures des valeurs qu'un joueur peut obtenir à partir de l'état actuel du jeu.

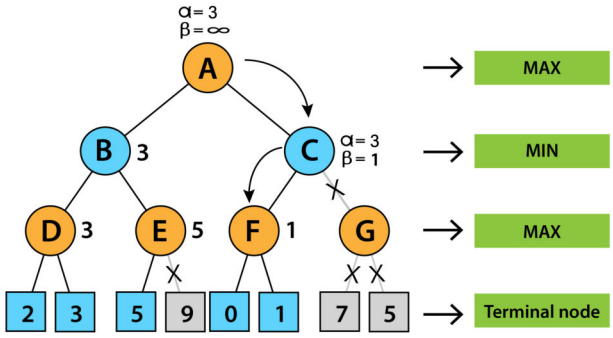

Figure 2 – Algorithme Alpha-Bêta en IA - MyGreatLearning

Similairement à l'algorithme Minmax, l'algorithme Alpha-Bêta parcours l'arbre de jeu récursivement. Lorsque la valeur d'alpha dépasse la valeur de bêta d'un nœud, cela signifie que la branche de l'arbre de recherche sous ce nœud ne peut pas influencer la décision finale, car elle mène à une situation moins favorable pour le joueur. On dit donc que cette branche est élaguée, et on ne l'explorera donc pas.

2.2 Alphabeta Table

La mise en place de table de transposition est une technique d'optimisation de l'algorithme Alpha-Bêta qui permet de réduire le nombre de nœuds explorés lors de la recherche de la meilleure décision à prendre en utilisant une mémoire cache.

On stock les évaluations des positions dans une table de transposition, de manière à pouvoir les réutiliser ultérieurement si elles sont rencontrées à nouveau lors de l'exploration de l'arbre de recherche.
Dans le cas d'un jeu à deux joueurs comme le notre, on utilisera pour la hashmap une fonction de hashage de zobrist, méthode courante pour identifier de manière unique chaque position sur le plateau de jeu. Vous avez donc implémenté la fonction *zobristkey* pour générer la clé correspondante à chaque position sur le plateau de jeu, en utilisant une table de hachage Zobrist.
La table de hachage Zobrist est initialisée avec des valeurs aléatoires, ce qui garantit que chaque position du plateau de jeu aura une clé unique et que deux positions identiques auront des clés différentes.

Lors de l'exploration de l'arbre de recherche, l'algorithme Alpha-Bêta stocke dans la table de transposition les positions évaluées avec une valeur d'alpha et de bêta. Si la même position est rencontrée plus tard lors de l'exploration de l'arbre, l'algorithme peut utiliser les valeurs d'alpha et de bêta stockées pour couper prématurément la recherche si la valeur stockée est suffisamment précise.

2.3 Alphabeta Pass

Une autre optimisation de l'algorithme Alpha-Bêta était d'implémenter le Null Move Pruning. Cette technique consiste à effectuer une recherche de coup "nul" (c'est-à-dire sans jouer de coup) à certains niveaux de l'arbre de recherche.

Si un joueur est dans une position très favorable, il devrait être capable de renoncer à un coup sans que cela ne lui cause trop de préjudice. Ainsi, en effectuant une recherche de coup nul, on simule que le joueur courant ne joue pas de coup, et que l'autre joueur joue deux coups consécutifs à la place.

Si le joueur courant est dans une position encore favorable après avoir passé son tour, alors on peut assumer que sa position était excellente, et les coups qui l'ont amené à cette position le sont tout autant. On peut alors couper prématurément la recherche en utilisant l'évaluation de la position actuelle.

Cependant, en utilisant cette technique, il est possible de louper un coup encore meilleur, et cela peut donc avoir un impact sur le résultat de la partie. Il s'agira donc de tester plusieurs valeurs arbitraires pour décider quand une position est très favorable (par exemple, le joueur gagne avec 9 blobs d'avance), auquel cas on testera le null move pruning.

3 Benchmark

3.1 Criterion

Pour tester la performance des différents algorithmes implémentés pour le jeu blobwars, nous avons utilisé la bibliothèque Criterion.rs pour effectuer des benchmarks. Nous avons créé une fonction *benchmarkpergroup* qui prend en paramètre un groupe de benchmark, un nom pour l'algorithme à tester et la profondeur maximale à tester. Cette fonction parcourt toutes les profondeurs et effectue un benchmark pour chaque profondeur. Ce benchmark correspond à une partie de l'algorithme à tester, contre le MinMax de profondeur 3. Nous mesurons alors le temps que la partie se finisse, ce qui n'est pas parfait car nous devrions plutôt mesurer la performance pour trouver un coût, car si la partie s'éternise la mesure est faussée.

Nous avons ensuite groupé les benchmarks ensemble pour chaque algorithme, et mis chaque groupe ensemble pour pouvoir tracer dans un même graphique les performances des différents algorithmes.

3.2 Résultats

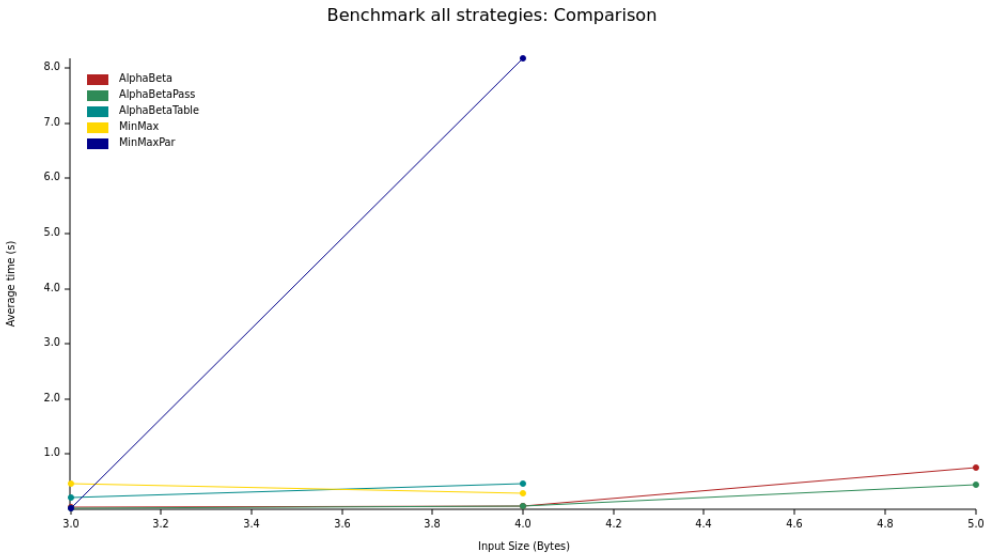

Figure 3 – Execution time of a game according to the depth

Le premier résultat visible est la très mauvaise performance de MinMax Parallèle. Grâce au benchmark, il est facile de voir que notre implémentation comporte des erreurs. Nous décidons donc de ne pas afficher cet algorithme dans le graphique pour mieux distinguer les différences de performance des autres algorithmes. De plus, nous augmentons la profondeur maximale, en sacrifiant le nombre d'échantillons (passant de 100 à 20).

On obtient alors :

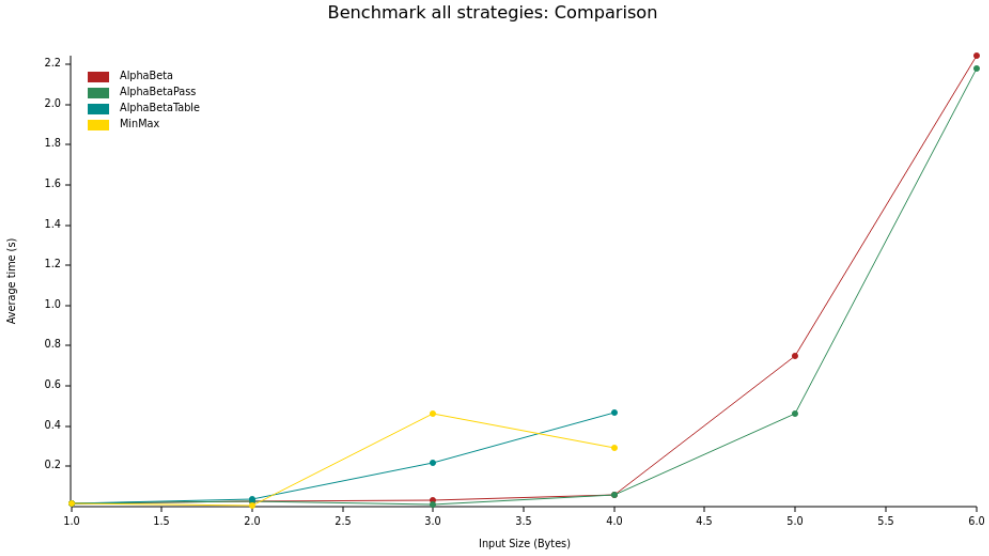

Figure 4 – Execution time of a game according to the depth, without MinMaxPar

Nous remarquons tout d'abord que l'algorithme MinMax est le moins performant. Sa performance s'améliore en passant d'une profondeur 3 à 4, ce qui peut sembler étonnant. Cependant,

il faut se rappeler que nous mesurons le temps d'une partie. Ici donc, avec une profondeur de 4, l'algorithme gagnera en beaucoup moins de coup, ce qui contrebalancera totalement la perte de performance due à la profondeur accrue.

Par ailleurs, l'algorithme AlphaBeta avec une table de transposition est peu performant, et il a donc fallu se limiter à une profondeur de 4. Nous pouvons donc dire qu'il n'y a que très peu de collisions, et donc nous ne gagnons quasi jamais de temps à ne pas recalculer une position déjà dans la table. De plus, pour presque chaque coup, nous devons créer un hash de la position et le stocker dans la table, ce qui augmente donc considérablement les performances.

Il reste donc les algorithmes AlphaBeta et AlphaBetaPass, correspondant à l'optimisation null move pruning. Il est d'abord notable que ces deux algorithmes, comme avec la table de transposition, ont des performances exponentielles en profondeur, car tous basés sur AlphaBeta.

Cependant, on peut noter une faible amélioration des performances pour AlphaBeta avec null move pruning. Nous avons, après tests, que pour optimiser le taux de victoire, il fallait essayer le null move pruning à partir de la profondeur $maxdepth - 3$, car avant on risquait trop souvent de louper un meilleur coup. Nous avions mis comme conditions supplémentaires que le joueur devait être en position très favorable, c'est à dire avec une avance de 7 blobs. Si ces deux conditions étaient réunies, on pouvait essayer le null move, et si après les deux coups successifs de l'adversaire, le joueur gagnait toujours de 7 blobs (score inchangé), alors on arrêtait la recherche et on choisissait le coup considéré.

Si on diminue la restriction de ces deux critères, avec une avance de 4 blobs seulement, et une avance après les deux coups successifs de l'adversaire de 2 blobs, on augmente les cas de null move pruning et donc la performance. Cependant, cela entraîne aussi une perte considérable de qualité de jeu, l'algorithme pouvant alors perdre contre un AlphaBeta de profondeur 4, chose impensable avec les restrictions fortes sur les critères.

4 Optimisations souhaitées

Il aurait été aussi possible de paralléliser Alpha-Beta, en ne parallélisant uniquement le premier niveau de noeuds.
La difficulté réside dans le fait que les valeurs de alpha et bêta sont dépendantes des calculs précedants. Une utilisation des mutex sur alpha et bêta était envisageable, pour qu'ils ne soient pas modifiés concurrement.

Conclusion

En conclusion, plusieurs versions de Minmax et d'Alpha-Bêta ont été implémentées. La seule optimisation intéressante était le null move pruning, qui avec de bonnes valeurs arbitraires de paramétrage, augmentait significativement le taux de victoire. De plus, une implémentation d'algorithme complexe en parallèle n'est pas chose facile, et semble souvent moins performant que l'algorithme non-paralléliser, du aux nombreuses optimisations efficaces du compilateur de Rust.