ENSIMAG

Rapport de projet

Algorithmique Avancée

Pierre Lin
2A-ISI-G4

6 Mai 2024

Table des matières

Introduction

Dans ce rapport, nous allons présenter les différents algorithmes développés pour calculer les meilleurs coups possibles dans le jeu BlobWar, ainsi que leurs performances respectives.

1 Algorithmes Implémentés

Tous les algorithmes développés sont disponibles dans les fichiers **strategy.h / strategy.cc**. Ces algorithmes se basent sur des fonctions communes :

- **computeValidMoves()** qui donne la liste des coups possibles d'un joueur dans une position donnée.
- **estimateCurrentScore()** qui donne une estimation de la valeur d'une position pour un joueur. Cette fonction renvoie simplement la différence entre le nombre de blobs que l'on possède et le nombre de blobs que l'adversaire possède.
- **applyMove()** qui joue un coup dans une position donnée.
- **revertMove()** qui rétracte un coup dans un position donnée.

1.1 Algorithme Naïf

Cet algorithme correspond à la fonction **algoNaif()**.

L'algorithme joue le premier coup possible qu'il trouve. Il est très rapide, mais pas très fort...

1.2 Algorithme Glouton

Cet algorithme correspond à la fonction **algoGlouton()**.

L'algorithme calcule la liste de tous les coups possibles, et joue celui ayant la meilleur évaluation. Il est meilleur que l'algorithme naïf, mais n'est pas très fort non plus...

1.3 Algorithme Min-Max

Cet algorithme correspond à la fonction **algoMinMax()**.

Soit d une profondeur. L'algorithme évalue chaque coup possible en alternant parmi les deux joueurs pour chaque profondeur. Ceci donne un arbre de jeu de profondeur d, où l'on a calculé l'évaluation de la position pour toutes ses racines. Le coup choisi est alors celui correspondant à la branche de l'arbre où chacun des joueurs aura joué le coup ayant la meilleure évaluation pour lui.

Pour une profondeur $d \geq 2$, il bat l'algorithme glouton (même s'il est plus lent).
En revanche, si $d = 1$, alors cet algorithme est équivalent à l'algorithme glouton.

1.4 Algorithme Alpha-Bêta

Cet algorithme correspond à la fonction **algoAlphaBeta()**.

L'algorithme est une amélioration de l'algorithme Min-Max. Il utilise deux valeurs α et β correspondant aux bornes inférieures et supérieures des évaluations qu'un joueur peut obtenir dans une position. Cela permet de réduire le nombre de nœuds évalués en élaguant les branches qui ne contribueront pas à la décision finale.

Comme il est plus rapide que l'algorithme Min-Max, on peut choisir une profondeur d plus grande, ce qui permet à Alpha-Bêta de battre Min-Max.

1.4.1 Algorithme Alpha-Bêta Iterative Deepening (It-Deep)

Cet algorithme correspond à la fonction **algoAlphaBeta2Anytime()**.

L'algorithme utilise la fonction précédente **algoAlphaBeta()** en incrémentant la profondeur d jusqu'à un certain temps ou seuil. Le coup choisi sera celui que le dernier **algoAlphaBeta()** executée aura renvoyé (donc le d le plus grand).

Cet algorithme a été implémenté car dans le jeu BlobWar, une IA a 1 seconde pour calculer son coup quand c'est à son tour. L'algorithme fait donc des Alpha-Bêta de plus en plus complexe jusqu'à épuisement du temps donné.

En pratique, les algorithmes comme **algoMinMax()** ou **algoAlphaBeta()** sont très difficiles à utiliser car il est compliqué de choisir un d à l'avance qui permet d'utiliser le temps donné efficacement tout en veillant à ne pas le dépasser.
Par la suite, nous aurons donc toujours une version Iterative Deepening (It-Deep) de nos algorithmes qui sera celle utilisée en réalité.

1.5 Algorithme Alpha-Bêta Parallèle

Cet algorithme correspond aux fonctions **algoAlphaBetaPar()** et **algoAlphaBetaPar2Anytime()** (Version It-Deep).

L'API utilisée pour la parallélisation est **OpenMP**.

La parallélisation se fait au niveau de la boucle parcourant la liste des coups possibles en utilisant "**#pragma omp parallel for**".

De plus, pour ne pas avoir de problème lorsque plusieurs threads veulent écrire dans les variables partagées (alpha / bêta), on utilise "**#pragma omp critical**".

Enfin, dès qu'un thread trouve une coupure, il met un **flag** à vrai, et tous les autres threads sortent de la boucle quand ils voient ce **flag** à vrai. Il est donc possible d'explorer des nœuds qu'on n'aurait pas exploré en séquentiel.

Un problème avec la parallélisation est la gestion du tableau contenant la position du jeu. En effet, en séquentiel, on pouvait simplement :
– Jouer le coup.
– Évaluer récursivement la position.
– Rétracter le coup.
Ceci n'est pas possible en parallèle car les threads vont tous jouer leur coup en même temps, ce qui va mettre n'importe quoi dans le tableau. Une idée est alors de créer une copie de la position pour chaque thread avec la fonction **copyBiDiArray()**. Chaque thread aura donc son tableau qu'il pourra modifier comme il veut. De plus, cela permet de ne pas avoir à rétracter le coup joué, car on n'a plus besoin du tableau après, c'est un mal pour un bien...

Enfin, le parallélisme s'arrête à partir d'un certain seuil (nommé **currLimDepth** dans le programme) où l'on repasse en séquentiel. On fait cela car, pour des seuils assez bas, le coût de la création et de la gestion des threads devient plus important que le gain qu'ils apportent.

En faisant jouer la version parallèle contre le version séquentiel, nous avons pu remarquer que la version parallèle gagnait dans la majorité des cas. Nous allons donc garder le parallélisme pour nos prochaines améliorations.

1.6 Algorithme Alpha-Bêta Sort Parallèle

Cet algorithme correspond aux fonctions **algoAlphaBetaSortPar()** et **algoAlphaBetaSortParAnytime()** (Version It-Deep).

L'algorithme est une extension du précédent algorithme 1.5. Il trie la liste des coups possibles de la meilleure évaluation jusqu'à la pire pour le joueur courant. La fonction d'évaluation utilisée est **estimateCurrentScore()**. On trie cette liste dans l'espérance d'avoir des coupures le plus tôt possible.

Comme précédemment, en faisant jouer cet algorithme contre la version sans tri, la version avec tri gagnait dans la majorité des cas. Nous allons donc conserver le tri.

1.7 Algorithme Alpha-Bêta Sort Zobrist Parallèle

Cet algorithme correspond aux fonctions **algoAlphaBetaSortZobristPar()** et **algoAlphaBetaSortZobristParAnytime()** (Version It-Deep).

L'algorithme est une extension du précédent algorithme 1.6. Il va créer une table de hachage qui stocke des couples position/évaluation (clé/valeur). Pour calculer le hash d'une position, nous allons utiliser la fonction de hachage de Zobrist :
https://en.wikipedia.org/wiki/Zobrist_hashing.
Ceci permet d'avoir presque sûrement des hash unique pour chacune de nos positions.

En théorie, utiliser une telle table de hachage permettrait d'économiser du temps en n'ayant pas à recalculer récursivement l'évaluation des positions qu'on aurait déjà évaluées avant (notamment grâce à l'It-Deep).

Cependant, en pratique, cet algorithme était toujours plus lent que le précédent et n'arrivait pas à des profondeurs très élevées dans sa version It-Deep. On peut donc penser que calculer le hash pour toutes les positions à chaque fois n'est pas rentable par rapport au gain que la table de hachage permet. Néanmoins, cet algorithme pourrait peut-être être meilleur si on lui laissait plus de temps, ou s'il pourrait stocker sa table de hachage dans la RAM durant toute la partie (chose que je n'ai pas réussi à faire car on relance ./launchStrategy à chaque tour). On ne conservera donc pas la table de hachage Zobrist pour la suite.

1.8 Algorithme PVS (Principal Variation Search) Sort Parallèle

Cet algorithme correspond aux fonctions **algoPVSSortPar()** et **algoPVSSortParAnytime()** (Version It-Deep).

C'est une implémentation de l'algorithme Principal Variation Search (PVS) : https://en.wikipedia.org/wiki/Principal_variation_search.
Cet algorithme marche particulièrement bien dans le cas où les coups possibles sont bien triés. Pour trier ces coups, nous n'allons pas faire comme dans la partie 1.6, mais plutôt en utilisant le principe de "**killer move**". On va dire qu'un coup est un **killer move** si on fait une coupure grâce à lui. Dans ce cas, on va stocker ce coup dans un set à la profondeur où la coupure s'est faite (on aura un set pour chaque profondeur).
Ceci est particulièrement utile dans la version It-Deep, car on va pouvoir réutiliser les **killer move** de nos itérations précédentes pour trier la liste des coups possibles. En effet, on triera cette liste en mettant les **killer move** au début.

De plus, cet algorithme a été parallélisé en utilisant la même idée que dans la partie 1.5. Seulement, comme l'algorithme repose sur le fait que l'on explore en premier les nœuds où l'on joue un **killer move**, alors, on va d'abord explorer ces nœuds en séquentiel, puis, on explore les nœuds restants en parallèle.

Cet algorithme gagne dans la majorité des cas contre notre meilleur algorithme jusqu'à maintenant (1.6). On conservera donc celui-là.

Par ailleurs, il existe des versions non parallèle et non triée de cet algorithme qui sont respectivement **algoPVSSortAnytime()** et **algoPVSAnytime()**. Sans surprise, ils sont moins performants que **algoPVSSortParAnytime()**.

2 Évaluation des Algorithmes

Pour classer nos algorithmes, on va les faire s'affronter sur différentes configurations du plateau du jeu. On commencera par faire s'affronter les deux algorithmes les moins bons, puis, on gardera le vainqueur qui affrontera l'algorithme suivant etc...

Les plateaux du jeu choisis sont : Standard, Path, Rings, Irregular, Island, et Constrained.

On organisera les données sous la forme d'un tableau, avec les noms des plateaux sur les lignes, et les noms des algorithmes sur les colonnes. Pour chaque case, on mettra :
– O pour l'algorithme gagnant.
– Une case vide pour l'algorithme perdant.
– // sur les deux cases en cas d'égalité.
– ∞ sur les deux cases en cas de répétition infinie (quand la partie ne se termine pas et que les algorithmes répètent la même suite de coups).

Enfin, sur les colonnes, le nom de l'algorithme sera coloré en bleu ou en rouge selon la couleur qu'il était durant la partie.

2.1 algoNaif() – algoGlouton()

Commençons par faire combattre nos deux algorithmes les plus faibles.

	Naïf	Glouton
Standard		O
Path		O
Rings		O
Irregular		O
Island		O
Constrained		O

	Naïf	Glouton
Standard		O
Path		O
Rings		O
Irregular		O
Island		O
Constrained	∞	∞

Taux de victoire **Naïf** : 0%.
Taux de victoire **Glouton** : 92% –> Vainqueur.
Taux d'égalité/répétition : 8%.

On garde donc **algoGlouton()** pour le combat suivant.

2.2 algoGlouton() – algoMinMax2Anytime()

	Glouton	MinMax
Standard		O
Path		O
Rings		O
Irregular		O
Island		O
Constrained		O

	Glouton	MinMax
Standard		O
Path		O
Rings		O
Irregular		O
Island		O
Constrained		O

Taux de victoire **Glouton** : 0%.
Taux de victoire **MinMax** : 100% –> Vainqueur.
Taux d'égalité/répétition : 0%.

2.3 algoMinMax2Anytime() – algoAlphaBeta2Anytime()

	MinMax	Alpha-Bêta
Standard		O
Path	O	
Rings		O
Irregular		O
Island		O
Constrained	∞	∞

	MinMax	Alpha-Bêta
Standard		O
Path	O	
Rings		O
Irregular	∞	∞
Island		O
Constrained	∞	∞

Taux de victoire **MinMax** : 17%.
Taux de victoire **Alpha-Bêta** : 58% –> Vainqueur.
Taux d'égalité/répétition : 25%.

2.4 algoAlphaBeta2Anytime() – algoAlphaBetaPar2Anytime()

	Alpha-Bêta	Alpha-Bêta Parallèle
Standard		O
Path		O
Rings		O
Irregular		O
Island	//	//
Constrained		O

	Alpha-Bêta	Alpha-Bêta Parallèle
Standard		O
Path		O
Rings		O
Irregular		O
Island	//	//
Constrained	O	

Taux de victoire **Alpha-Bêta** : 8%.
Taux de victoire **Alpha-Bêta Parallèle** : 75% –> Vainqueur.
Taux d'égalité/répétition : 17%.

2.5 algoAlphaBetaPar2Anytime() – algoAlphaBetaSortParAnytime()

	Alpha-Bêta Parallèle	Alpha-Bêta Sort Parallèle
Standard		O
Path	O	
Rings	O	
Irregular		O
Island	//	//
Constrained	∞	∞

	Alpha-Bêta Parallèle	Alpha-Bêta Sort Parallèle
Standard		O
Path		O
Rings		O
Irregular		O
Island		O
Constrained	∞	∞

Taux de victoire **Alpha-Bêta Parallèle** : 17%.
Taux de victoire **Alpha-Bêta Sort Parallèle** : 58% –> Vainqueur.
Taux d'égalité/répétition : 25%.

2.6 algoAlphaBetaSortParAnytime() – algoAlphaBetaSortZobristParAnytime()

	Alpha-Bêta Sort Parallèle	Alpha-Bêta Sort Zobrist Parallèle
Standard	O	
Path	O	
Rings	O	
Irregular		O
Island	O	
Constrained	O	

	Alpha-Bêta Sort Parallèle	Alpha-Bêta Sort Zobrist Parallèle
Standard	O	
Path		O
Rings	O	
Irregular	O	
Island	O	
Constrained	∞	∞

Taux de victoire **Alpha-Bêta Sort Parallèle** : 75% –> Vainqueur.
Taux de victoire **Alpha-Bêta Sort Zobrist Parallèle** : 17%.
Taux d'égalité/répétition : 8%.

2.7 algoAlphaBetaSortParAnytime() – algoPVSSortParAnytime()

	Alpha-Bêta Sort Parallèle	PVS Sort Parallèle
Standard		O
Path	O	
Rings		O
Irregular	O	
Island		O
Constrained	O	

	Alpha-Bêta Sort Parallèle	PVS Sort Parallèle
Standard		O
Path		O
Rings	//	//
Irregular		O
Island		O
Constrained	O	

Taux de victoire **Alpha-Bêta Sort Parallèle** : 34%.
Taux de victoire **PVS Sort Parallèle** : 58% –> Vainqueur.
Taux d'égalité/répétition : 8%.

En conclusion, **PVS Sort Parallèle** semble être l'algorithme le plus performant dans la majorité des cas, on utilisera donc celui-là.