GRENOBLE INP Ensimag UGA

Ensimag ISI 2A
2023/2024

Loan GATIMEL
Julian COUX
Equipe 4

Projet Algo avancé :
BlobWar

Objectif du projet :

Réalisation d'un algorithme capable de jouer au jeu BlobWar. Cet algorithme doit être optimisé pour être le plus efficace possible et jouer son coup en moins de 1 seconde.

I - Algo glouton :

L'algorithme glouton consiste à parcourir naïvement tous les coups possibles et à choisir celui offrant le meilleur ratio.
Pour ce faire, nous ciblons un pion de notre couleur et parcourons tous ses déplacements potentiels. À chaque déplacement, nous calculons le nombre futur de pions de notre couleur ainsi que ceux de l'adversaire, ce qui nous donne un ratio associé à ce coup.
Si ce ratio est identifié comme étant le meilleur jusqu'à présent, nous sauvegardons le coup. Une fois que tous les pions ont été parcourus, nous renvoyons le coup offrant le meilleur ratio.

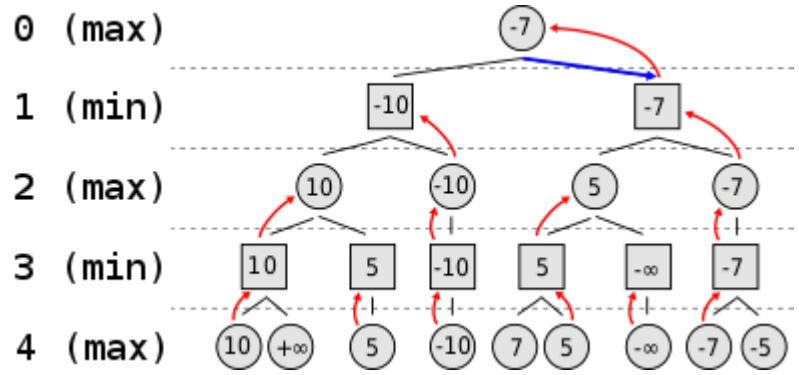
II - Idée Monte Carlo :

L'objectif de cet algorithme est de parcourir tous les déplacements possibles des "blobs". À chaque étape, un coup est sélectionné et appliqué et une boucle est entamée, répétée un certain nombre de fois (par exemple, 1000 fois). À l'intérieur de cette boucle, les coups sont appliqués de manière aléatoire successivement pour les joueurs bleu et rouge, jusqu'à ce que la partie soit terminée. Si le joueur devant jouer initialement remporte la partie, un compteur est incrémenté. À la sortie de la boucle, si ce compteur dépasse le compteur maximum trouvé précédemment, il est mis à jour. Après avoir parcouru tous les coups possibles, le choix optimal est déterminé en sélectionnant celui qui maximise le compteur. En résumé, cet algorithme consiste à choisir le coup offrant la meilleure probabilité de victoire après de nombreuses itérations de jeu aléatoire.

Nous n’avons pas implémenté cet algorithme pour 2 raisons, il n’était pas demandé et l’algorithme min-max est une amélioration de monte carlo.

III - Algo min max :

Description :



L'objectif premier de l'algorithme Minimax est de déterminer la stratégie optimale pour un joueur, en tenant compte de l'hypothèse que l'adversaire adopte également une stratégie optimale. Voici les étapes suivies par notre algorithme :

Dans un premier temps, on collecte tous les mouvements possibles pour le joueur en cours, puis on évalue chacun de ses mouvements. Par exemple, si le joueur actif est le rouge dans notre cas, on évalue les mouvements en se basant sur la différence entre le nombre de blobs rouges et bleus.

Ensuite, on explore les différentes possibilités de mouvements à partir de l'état initial du jeu, en construisant un arbre de recherche. Chaque nœud de cet arbre représente un état du jeu, tandis que les branches partant de chaque nœud représentent les différents mouvements possibles.

À partir des évaluations des feuilles de l'arbre, l'algo adopte une approche récursive pour attribuer des valeurs aux nœuds parents de l'arbre. Pour un nœud correspondant au joueur rouge, l'algorithme sélectionne le mouvement qui maximise son score. Pour un nœud correspondant au joueur bleu, il choisit le mouvement qui minimise le score du joueur. Ainsi, l'algorithme alterne entre les phases de "minimisation" et de "maximisation", d'où son nom "Min-max".

Après avoir parcouru tout l'arbre de jeu jusqu'à une certaine profondeur prédéfinie (par exemple, 3 ou 4 pour nous), l'algorithme prend une décision basée sur le meilleur mouvement trouvé jusqu'à cette profondeur.

Problème et Solution:

Au cours de ce projet, notre premier défi est survenu avec l'algorithme Min-max. Cependant, la façon dont nous avons abordé ce problème a finalement été très bénéfique pour l'ensemble du projet.

Le problème était en fait assez simple à résoudre, mais au début, nous avons eu du mal à le déboguer efficacement, ce qui nous a empêchés d'identifier rapidement sa source. Le problème était lié à la manière dont nous récupérions les mouvements possibles à chaque niveau de profondeur de l'algorithme. Au lieu de tenir compte du plateau de jeu modifié à chaque niveau, nous récupérions les mouvements possibles par rapport au plateau initial, ce qui était faux.

Nous avons réalisé que quelque chose n'allait pas lorsque nous avons constaté que l'algorithme Min-max était moins efficace que l'algorithme glouton, même s'il était censé avoir une portée plus étendue. Pour résoudre ce problème, nous avons ajusté le temps entre deux coups imposés et utilisé la fonction getchar() pour examiner en détail chaque coup. De plus, nous avons mis en place une fonction prettyPrint pour afficher les différents états du jeu à n’importe quel niveau de profondeur.

Cette approche nous a permis de corriger le problème de manière relativement simple. De plus, cela nous a aidé à résoudre les autres problèmes rencontrés par la suite, car nous avions trouvé une méthode de débogage efficace.

IV - Élagage alpha beta :

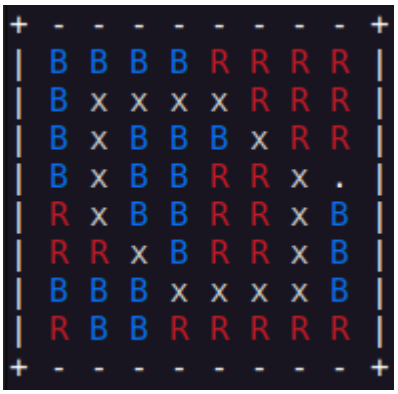
Description :

Afin de réaliser l'algorithme d'élagage alpha beta, nous avons simplement repris l'algorithme minimax, en y ajoutant des variables ***alpha*** et ***beta***, qui permettent, en fonction de leur valeur, de supprimer certaines branches de l'arbre, qui par définition n'améliorent pas la solution. Cette méthode permet de ne plus explorer les branches inutiles de notre arbre et ainsi gagner un temps conséquent.

Lors du développement de alphaBeta, nous avons rencontré plusieurs problèmes qui nous ont poussé à debug notre algo en détail.

Problèmes :

Problème de se téléporter au lieu de se dupliquer pour finir la partie.
Une fois que nos algorithmes miniMax et alphaBeta marchaient assez, bien. Nous avons effectué de nombreux tests qui nous ont fait remonter une erreur commune aux deux algorithmes. Illustrons par un exemple :



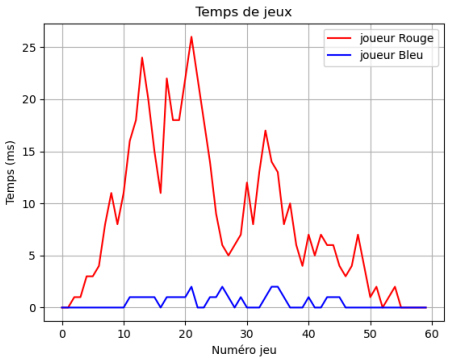
Dans cette configuration de jeu, si c'était par exemple au joueur Bleu de jouer, on voudrait naturellement qu'il se duplique dès son premier coup pour remplir le tableau et finir la partie. Mais l'erreur récurrente que nous avons observé est qu'il préférait prendre un pion plus loin et se téléporter. Etant donné que les deux joueurs faisaient cela, la fin de partie tournait en rond.
Le problème est qu'on impose une profondeur de 3 par exemple, et si le pion se duplique, ses fils n'auront plus aucune possibilité de jeux donc on ne calcule pas de meilleur coup. Dans ce cas, on renvoie le mouvement par défaut : de 0,0 à 0,0 et avec un ratio de +/- l'infini. Donc dans certain cas, c'était le mouvement qu'on renvoyait, mouvement qui n'est pas possible donc la partie se terminait.
Il fallait alors prendre en compte le fait qu'un joueur n'avait plus aucun coup de possible et arrêter le parcours de l'arbre.

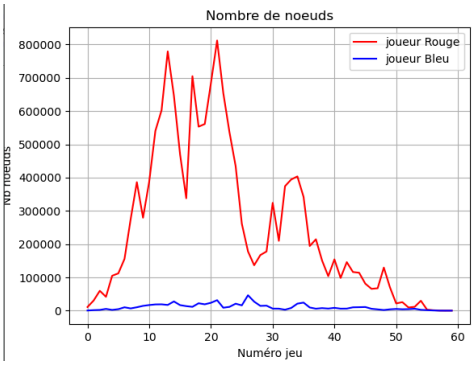
```
if(valid_moves.empty()){    //si il n'y a aucun mouvement possible
    MinimaxResult result{nbRouge - nbBleu, movement(0, 0, 0, 0), true};
    return result;
}
```

Ce que nous avons fait pour corriger ce bug, c'est que nous avons simplement testé si arrivé à un état, il n'y a plus aucun coup à jouer, il renvoie simplement un mouvement null avec le ratio actuel : nbRouge - nbBleu.

Après une première version fonctionnelle de cet algorithme, nous avons pu observer une différence de performance incroyable entre l'algorithme alpheBeta et minimax :

Rouge : minimax
Bleu : alphaBeta
Map : pleine
Profondeur : 3





V - Optimisation de alphaBeta :

Au départ, nous avons mesuré la performance de nos algorithmes minimax et alpha-beta en évaluant leur temps de calcul et leur fiabilité lors de jeux contre eux-mêmes. Cependant, pour affiner l'optimisation de l'algorithme alpha-beta, nous avons cherché des méthodes plus précises de comparaison.

Nous avons conservé le principe de faire jouer les différentes versions des algorithmes les unes contre les autres. Pour chaque coup, nous avons mesuré le temps nécessaire à l'algorithme pour trouver la meilleure solution ainsi que le nombre de nœuds explorés dans l'arbre de jeu. Cela nous a permis d'évaluer l'efficacité de l'optimisation tant en termes de temps de calcul que de réduction du nombre de nœuds explorés.

Idée d'optimisation :

La première optimisation de l'algorithme alpha-beta consistait à ordonner les coups possibles du joueur en fonction de leur ratio, présumant que les coups initialement forts seraient également les meilleurs par la suite. Cette approche a en effet montré une réduction significative du nombre de nœuds explorés. Cependant, le temps de calcul n'a pas été proportionnellement réduit en raison du coût de tri du tableau des coups, même en utilisant des méthodes de tri efficaces telles que le quicksort ou le tri par fusion.
Nous avons donc opté pour un tri beaucoup plus simple mais rapide, qui ne nécessite aucun parcours préalable du tableau.
Une première version (V1) consiste à prendre en compte uniquement le meilleur ratio, de manière à placer en tête le max et de mettre de reste au fur et à mesure en queue de tableau. Une insertion en tête coûte **n** avec les vecteurs, ce qui peut rapidement être coûteux. C'est pour cela que cet algorithme est intéressant, il fait le moins de calculs possible.

```
if(ratio > maxRatio){
    maxRatio = ratio;
    ratios_and_moves.insert( position: ratios_and_moves.begin(), x: make_pair( &: ratio, &: mv));
    continue;
}
else{
    ratios_and_moves.push_back(make_pair(ratio, mv));
}
```

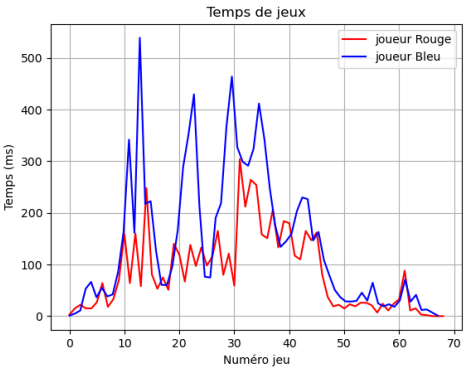
Une seconde version (V2) place les nouveaux ratios en tête ou en fin de tableau en se basant sur des informations telles que le ratio moyen, le ratio maximum, le ratio minimum, ainsi que les distances entre les ratios et ces valeurs de référence. Bien que moins précis, ce tri permet en moyenne de placer les meilleurs coups en tête du tableau, améliorant ainsi l'efficacité de l'algorithme alpha-beta.

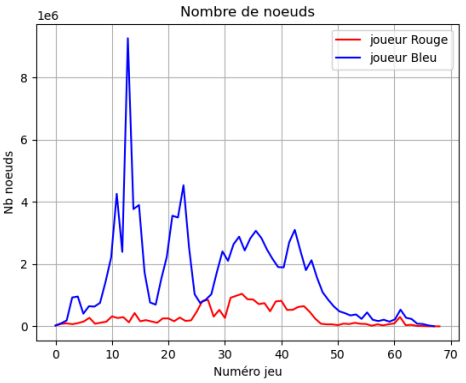
```
int moyenne = ratioMoyen / nbRatioCalc;
int distMax = abs( x: ratio - maxRatio);
int distMoy = abs( x: ratio - moyenne);
int distMin = abs( x: ratio - minRatio);
//on pourrait encore découper en plusieurs sections pour être précis
if(distMoy < distMax && distMoy < distMin){
    int milieu = ratios_and_moves.size() / 2;
    ratios_and_moves.insert(ratios_and_moves.begin() + milieu, make_pair(ratio, mv));
}
else if(distMin < distMax){
    ratios_and_moves.push_back(make_pair(ratio, mv)); // Sauvegarde du ratio et du mouvement dans le tableau
}
else{
    ratios_and_moves.insert(ratios_and_moves.begin() + 1, make_pair(ratio, mv));
}
```

Comparaison des algorithmes :

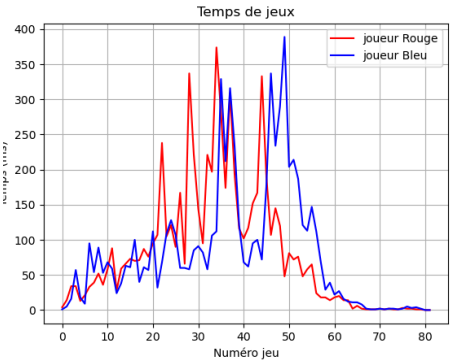
Afin de déterminer quelle optimisation sera la meilleure lors du tournoi, nous avons effectué une série de tests qui se basent sur le score finale, le temps de calcul et le nombre de nœuds parcourus.
Nous effectuons les tests sur la carte sans aucun trous car c'est sur celle-ci que il y a le plus grand nombre de calculs, ainsi les différences de performance seront plus visibles.

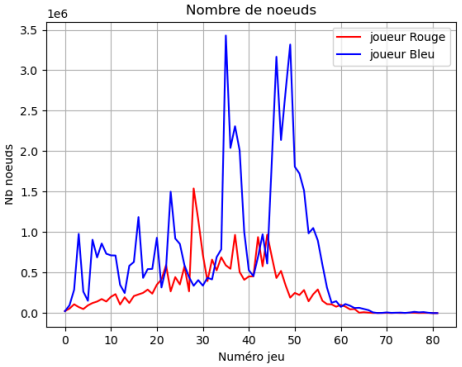
Rouge : alphaBetaOpti_V1
Bleu : alphaBeta
Map : pleine
Profondeur : 5
Victoire :
Rouge, 35 à 29





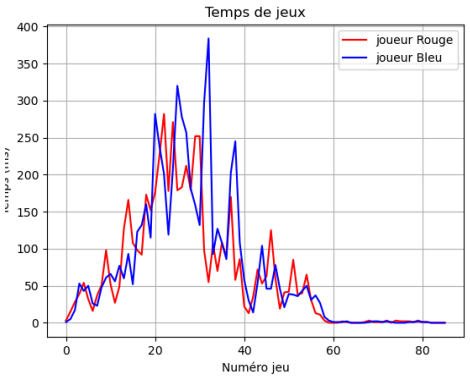
Rouge : alphaBetaOpti_V2
Bleu : alphaBeta
Map : pleine
Profondeur : 5
Victoire :
Rouge, 36 à 28

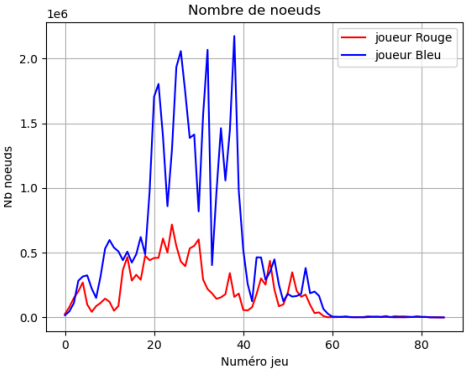




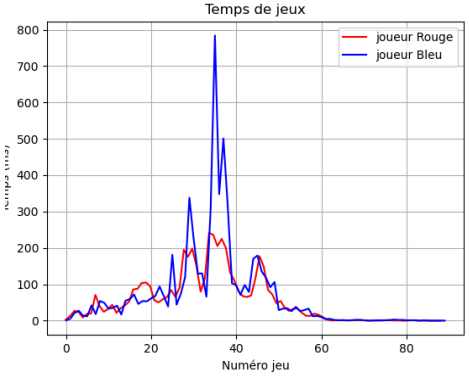
On peut d'abord voir que la V1 de l'optimisation est légèrement plus rapide que la V2. Mais étant donné que le déroulé de la partie n'est pas tout à fait le même, il est difficile de comparer.
Pour les comparer efficacement, on fait s'affronter la V1 avec la V2 sur la map pleine.

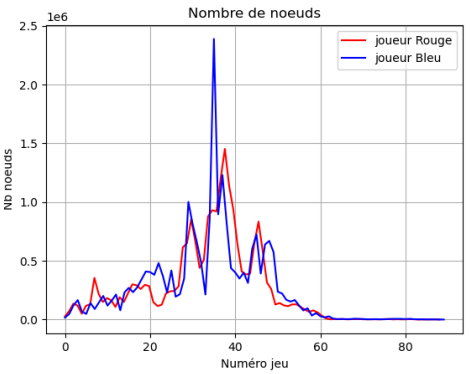
Rouge : alphaBetaOpti_V1
Bleu : alphaBetaOpti_V2
Map : pleine
Profondeur : 5
Victoire :
Rouge, 41 à 23





Rouge : alphaBetaOpti_V2
Bleu : alphaBetaOpti_V1
Map : pleine
Profondeur : 5
Victoire :
Bleu, 39 à 25





On peut voir que de manière générale, l'optimisation V1 est sensiblement plus lente que la V2, ce qui s'explique par un nombre de nœuds parcourus plus élevé en moyenne. Cependant, bien que celui-ci soit plus lent, il bat la V2 dans les deux configurations de jeux.
En théorie, la seconde optimisation (V2) est meilleure car le tri est plus précis donc permet d'élaguer plus l'arbre. Mais en pratique, ce tri plus précis ne fait pas gagner en performance de jeu, au contraire.
Ce qui nous poussera à prendre, le jour du tournoi, la version V1 plutôt que la V2, afin de mettre toutes les chances de notre côté avec l'algorithme qui performe le mieux sur le plateau de jeux.
Avec les contraintes de temps du jeu, nous ne pouvons pas dépasser une profondeur de 5 sur la map pleine. Sur certaines map avec des trous, la profondeur 6 est assez rapide mais nous privilégierons une profondeur de 5 quelque soit la map.

VI - Algo min-max parralèle :

Nous avons choisi d'aborder d'abord la parallélisation de l'algorithme Minimax avant de nous pencher sur l'algorithme Alpha-Bêta en parallèle. Cette décision a été motivée à la fois par le désir de nous familiariser avec la parallélisation et parce que nous avons rapidement compris comment procéder en écrivant l'algorithme Minimax de manière séquentielle. De plus, nous avons préféré commencer par Minimax car sa parallélisation impliquait l'utilisation d'une boucle parallèle sur les coups, suivie de la récupération du maximum de chaque coup, ce qui ressemble beaucoup à la technique de réduction que nous avons étudiée en cours et en travaux dirigés. En revanche, Alpha-Bêta semblait plutôt correspondre à un modèle MapReduce car il impliquait une dépendance entre alpha et bêta.

Malheureusement, nous avons rencontré des difficultés à utiliser correctement les outils de parallélisation, même si nous avions une idée claire du code à mettre en œuvre.

VII - Conclusion :

En résumé, notre projet sur l'optimisation des algorithmes pour BlobWar nous a permis de s'intéresser à la prise de décision dans les jeux stratégiques. De l'algorithme glouton à l'élagage alpha-beta, nous avons exploré diverses approches pour améliorer les performances de notre programme.

Nous avons surmonté des défis techniques, tels que la correction des erreurs dans l'algorithme Minimax et l'introduction de l'élagage alpha-beta pour accélérer le processus de recherche. Bien que la parallélisation de Minimax ait posé des difficultés, elle nous a permis de mieux comprendre les défis de la programmation concurrente.

En fin de compte, ce projet était une bonne expérience dans le développement d'algorithmes pour les jeux stratégiques, cela a aussi renforcé nos compétences en résolution de problèmes et en optimisation. Ces connaissances seront certainement bénéfiques pour la suite.