Projet Blobwar - Algorithmique avancée

Alexandre Jeunot-Caire & Tanguy Poinson

Avril 2021

Sommaire

1 Introduction

L'objectif de ce projet est d'implémenter plusieurs algorithmes de prise de décision pour l'intelligence artificielle du jeu Blobwar. Le but est d'implémenter des algorithmes qui prennent la décision qui maximise les chances de victoire en un temps donné, à chaque tour. Nous avons d'abord implémenté un algorithme « naïf », qui fait le mouvement qui maximise le score du joueur au tour suivant. Par la suite, nous avons implémenté une série d'algorithmes dont le but est de maximiser le score du joueur dans plusieurs tours, tout en essayant de prévoir les mouvements du joueur adverse (on supposera que le joueur adverse est un bon joueur qui lui aussi cherchera toujours les meilleurs coups). Nous avons implémenté cette intelligence artificielle en Rust, un langage de bas niveau qui à la particularité de bien gérer l'execution des threads (instructions parallèles). Nous avons donc utilisé cette particularité pour implémenter des versions parallèles des algorithmes mis en place.

2 Algorithmes implémentés

2.1 Greedy

Cet algorithme cherche à maximiser le score du joueur au tour suivant. Il lui suffit donc d'évaluer le score de la grille après chaque coup possible, et de conserver le coup associé au meilleur score.

```
impl Strategy for Greedy {
    fn compute_next_move(&mut self, state: &Configuration) -> Option<Movement> {
        state.movements()
                .into_iter()
                .map(|coup| (coup, state.play(&coup).value()))
                .max_by_key(|&(_, val)| val)
                .map(|(res, _)| res)
    }
}
```

FIGURE 1 – Stratégie "Greedy"

Cet algorithme est relativement efficace, et clairement meilleur qu'un algorithme random, mais nos tests ont montré qu'il peut très facilement se faire "berner" par des algorithmes qui voient un peu plus loin que lui et tendront un piège pour mieux le détruire à plus long terme.

2.2 MinMax

2.2.1 Présentation de l'algorithme

L'algorithme **MinMax** cherche à donner au joueur le meilleur score possible dans n tours. Pour cela, il explore complètement l'arbre des coups possibles. La sélection est faite en considérant que le joueur adverse choisira lui aussi le meilleur coup possible (et donc le pire pour nous). Il s'agit donc d'une sucession de *max* (nos coups, où l'on joue du mieux possible) et de *min* (les coups de l'adversaire, qui joue au mieux pour lui et donc au pire pour nous).

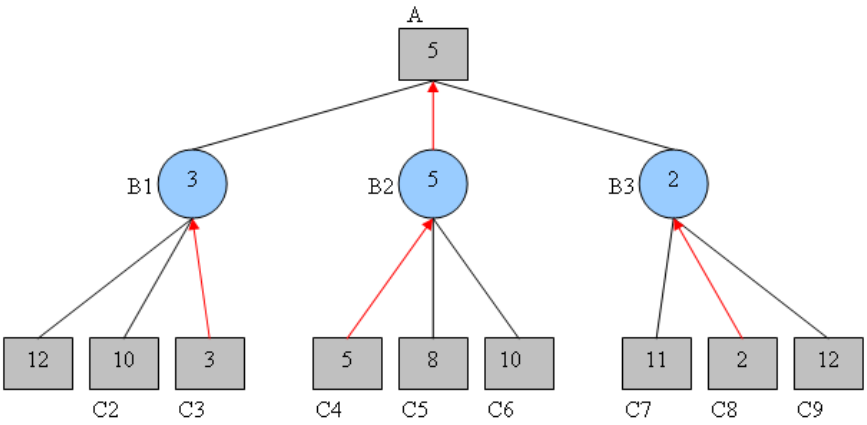

FIGURE 2 – Schéma représentant l'algorithme MinMax

2.2.2 Negamax

Nous avions d'abord implémenté une version classique de cet algorithme, avec à chaque fois un $if\ maximizing$ afin de savoir si nous devions renvoyer un max ou un min à un certain niveau. Toutefois, nos recherches nous ont amenés à nous intéresser à la version **Negamax** de l'algorithme.

Cette version, possible grâce au fait que Blobwar est un jeu à somme nulle et que chaque joueur a les mêmes possibilités et règles de calcul de score, n'apporte aucune différence quant aux résultats renvoyés, elle consiste simplement en une simplification basée sur l'observation que :

$$min(a, b) = -max(-a, -b)$$

Ainsi, nous avons pu éliminer les if pour finalement obtenir le code suivant :

```rust
fn negamax(profondeur: u8, state: Configuration) -> i8 {
    if profondeur == 0 {
        -state.value()
    } else {
        state.movements()
             .into_iter()
             .map(|coup| -negamax(profondeur - 1, state.play(&coup)))
             .max()
             .unwrap_or(-state.value())
    }
}
```

FIGURE 3 – Algorithme Negamax

2.2.3 Parallélisation

S'il n'était pas pertinent de paralléliser **Greedy** car le coût de création des threads ainsi que leur jointure est supérieur à un simple parcours de la map, la situation est ici bien différente. En effet, on observe trivialement que l'exploration de tous les coups possibles de l'arbre est exponentielle ($O(24^d)$ où d est la profondeur).

Ainsi, paralléliser cet arbre pourrait avoir un réel avantage. Pour cela, nous avons opté une parallélisation du premier niveau de MinMax uniquement, et de continuer en séquentiel. La partie "parallèle" s'effectue donc uniquement dans cette partie du code :

```rust
impl Strategy for MinMax {
    fn compute_next_move(&mut self, state: &Configuration) -> Option<Movement> {
        state.movements()
                .collect::<Vec<Movement>>()
                .par_iter()
                .map(|coup| (coup, -negamax(self.0 - 1, state.play(&coup))))
                .max_by_key(|&(_, val)| val)
                .map(|(res, _)| *res)
    }
}
```

FIGURE 4 – MinMax parallèle

2.2.4 Performances

Nous avons essayé de mesurer la performance des algorithmes que nous avons produits, notamment pour se rendre compte de l'efficacité de l'utilisation d'algorithmes parallèles pour la prise de décision. Nous avons donc fait s'affronter chaque algorithme contre lui-même pour une partie entière, mesuré le temps d'exécution de la partie, puis divisé ce temps résultant par le nombre de coups joués. Nous avons d'ailleurs remarqué que les algortithmes finissaient la partie en un nombre de coups plus important à mesure que la profondeur augmente, ce qui peut être signe que les stratégies devenaient de plus en plus fines.

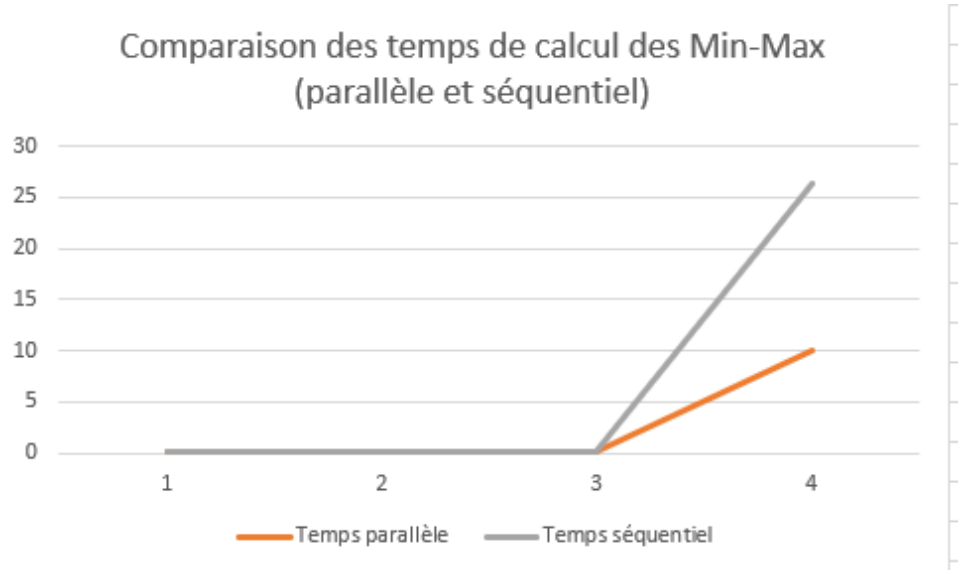

FIGURE 5 – Comparaison du temps de calcul des Min-Max séquentiel et parallèle en fonction de la profondeur

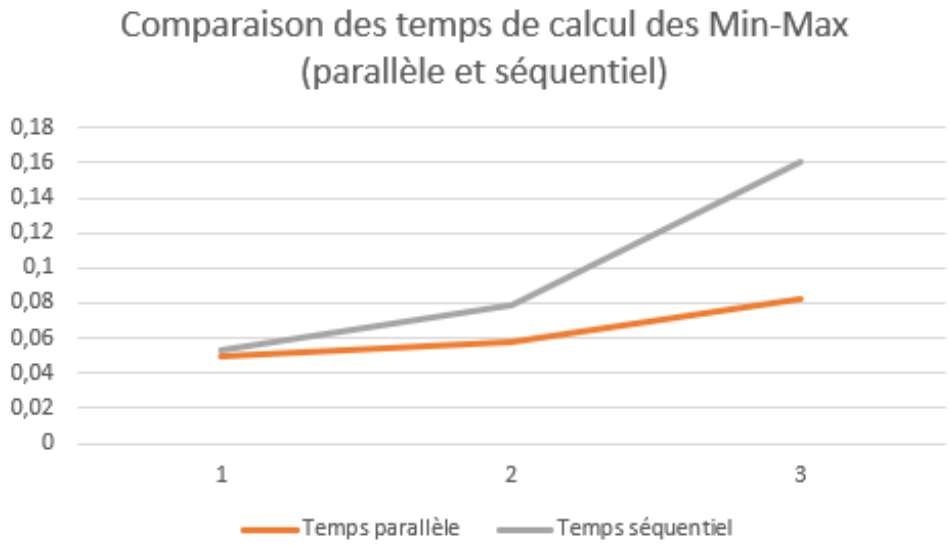

FIGURE 6 – Comparaison du temps de calcul des Min-Max séquentiel et parallèle en fonction de la profondeur (pour $N_{max} = 3$)

2.2.5 Pistes d'amélioration

La principale amélioration de l'algorithme de **MinMax** est l'élagage **AlphaBeta** que nous avons implémenté et sur lequel nous reviendrons plus en détail ultérieurement.

Dans notre algorithme final, nous n'avons parallélisé que le premier niveau de MinMax. On peut toutefois envisager d'autres possibilités, comme paralléliser chaque niveau. Nous avons testé cette solution, qui ne s'est pas avéré probante. Une meilleure solution serait peut-être de fixer une limite permettant de décider si l'on doit travailler en parallèle ou en séquentiel ($\frac{profondeur}{2}$ par exemple).

2.3 Alpha-Beta

2.3.1 Présentation de l'algorithme

L'algorithme **AlphaBeta** est une amélioration de **MinMax**. Si le principe initial est le même, il repose toutefois sur des mécanismes de **Branch and Bound** dans l'optique d'éviter d'explorer certaines branches dont on sait qu'elles ne conviendront pas. En réduisant la fenêtre $[alpha, beta]$ au fil de l'exécution, on peut couper de plus en plus de branches.

Nous avons expérimenté différentes stratégies et optimisations que nous allons à présent expliquer.

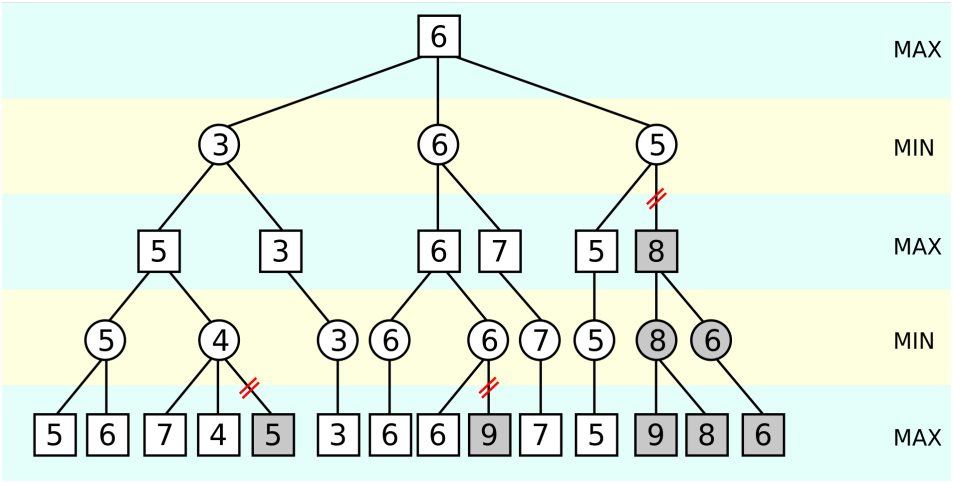

FIGURE 7 – Schéma représentant l'algorithme AlphaBeta

2.3.2 Algorithmes séquentiels

AlphaBeta classique Voici notre implémentation la plus basique d'AlphaBeta. Comme vous pouvez le constater, elle repose sur les mêmes principes que le **Negamax**.

```rust
fn alphabeta(profondeur: u8, mut alpha: i8, beta: i8, state: Configuration)
            -> (Option<Movement>, i8) {
    if profondeur == 0 || state.movements().peekable().peek().is_none() {
        (None, -state.value())
    } else {
        let mut best_move = None;
        let mut best_val = -127;

        for coup in state.movements() {
            let val = -alphabeta(profondeur - 1, -beta,
                                 -alpha, state.play(&coup)).1;
            if val > best_val {
                best_val = val;
                best_move = Some(coup);
                if best_val > alpha {
                    alpha = best_val;
                    if alpha >= beta {
                        break; // Problème pr paralléliser
                    }
                }
            }
        }
        (best_move, best_val)
    }
}
```

FIGURE 8 – Stratégie "AlphaBeta" classique

AlphaBeta Sorted Nous nous sommes intéressés par la suite à la possibilité de trier les coups dans l'itérateur *state.movements*. En effet, trier ne serait-ce que par un $alphabeta(profondeur - 2)$ (ou même un simple $state.value()$) ne coûte pas si cher, et pourrait potentiellement permettre d'avoir une estimation de ce qui pourrait être un bon coup, ce qui permettrait de couper plus rapidement. Cependant, les performances de notre implémentation laissaient à désirer, ce qui nous a conduits à tenter d'autres approches.

```
    ...
if profondeur >= 4 {
    let mut mouvements_ordonnes = state.movements().collect::<Vec<Movement>>();
    mouvements_ordonnes.sort_by_key(|&coup| -alphabeta(2, alpha, beta,
            state.play(&coup)).1);
    for coup in mouvements_ordonnes {
    ...
```

FIGURE 9 – Extrait de notre stratégie "Sorted"

AlphaBeta Aspiration Search La fenêtre d'aspiration initialise *alpha* et *beta* à des valeurs plus proches du score actuelle (nous avons eu des résultats acceptables pour $alpha = state.value() - 30, beta = state.value() + 30$) en espérant que la bonne valeur soit dedans. Cela permet de couper plus de branches. Toutefois, si la "bonne" valeur est hors de cette fenêtre, il faut alors relancer AlphaBeta, soit avec une fenêtre légèrement plus grande, soit (et c'est ce que nous avons fait) avec $[-127, 127]$.

Les résultats que nous avons obtenu avec cet algorithme étaient bons. Il était légèrement plus rapide que la plupart des autres, cependant nous avons observé qu'il avait tendance à un peu moins bien jouer. Cela nous a conduit à ne pas l'utiliser dans la version finale de notre programme. Nous nous en sommes cependant inspirés pour créer notre algorithme champion (dans les séquentiels) que nous allons à présent vous présenter.

AlphaBeta PVS Cet algorithme est le meilleur que nous ayions réalisé. L'idée est que beaucoup de coups explorés sont trop mauvais (ou trop bons) pour être acceptables. Cela consiste à faire un alpha-beta "normal" sur le premier coup possible, puis à tester les autres avec une fenêtre réduite. La fenêtre, beaucoup plus étroite, permet de couper bien plus facilement.

Si l'on fait un coup qui fait évoluer alpha, on refait une recherche afin de trouver la valeur exacte du score.

```
fn alphabeta_pvs(profondeur: u8, mut alpha: i8, mut beta: i8, state: Configuration)
        -> (Option<Movement>, i8) {
    if profondeur == 0 || state.movements().peekable().peek().is_none() {
        (None, -state.value())
    } else {
        let mut best_move = None;
        let mut best_val = - 127;
        for (i, coup) in state.movements().enumerate() {
            let mut score;
            if i == 0 {
                score = -alphabeta_pvs(profondeur - 1, -beta, -alpha,
                            state.play(&coup)).1;
            } else {
                score = -alphabeta_pvs(profondeur - 1, -alpha - 1, -alpha,
                            state.play(&coup)).1;
                if alpha < score && score < beta {
                    score = -alphabeta_pvs(profondeur - 1, -beta, -score,
                            state.play(&coup)).1;
                }
            }
            if score > best_val {
                best_val = score;
                best_move = Some(coup);
                if best_val > alpha {
                    alpha = best_val;
                    if alpha >= beta {
                        break;
                    }
                }
            }
        }
        (best_move, alpha)
    }
}
```

FIGURE 10 – AlphaBeta PVS

2.3.3 Parallélisation

La parallélisation d'AlphaBeta est bien plus complexe que celle de Min-Max. Tout d'abord, ce qui fait toute la puissance d'AlphaBeta est la capacité de *break*, de ne pas continuer à explorer les branches lorsqu'il sait qu'elles ne sont pas bonnes. Nous l'avons vu en cours, les *break* ne sont pas triviaux

à gérer.

En outre, un autre avantage d'**AlphaBeta** est la propagation des valeurs d'alpha et de beta, qui permet de couper de plus en plus de branches à mesure que l'on avance. En parallélisant, à moins de trouver des algorithmes particulièrement pointus, il semble que l'on perde en capacité à couper des branches, puisque chaque thread ignore que les autres ont potentiellement réduit l'écart entre alpha et beta.

Nous avons essayé plusieurs variantes, avec des résultats plus ou moins probants.

Parallélisation du premier niveau La première idée que nous avons eue consistait à ne limiter la parallélisation qu'au premier niveau de l'arbre. L'objectif ici était double :

— d'une part, cet algorithme serait relativement facile proche de ce que nous avons fait pour MinMax. Par conséquent, il pourrait être plus facile à adapter.

— d'autre part, nous avons vite estimé qu'il était plus efficace de mieux couper des branches que de paralléliser davantage. Un compromis devait être fait entre propagation des états d'alpha et beta *vs* de l'exploration en parallèle, et celui-ci nous semblait adapté.

Le premier algorithme que nous avons réalisé était celui-ci :

```
fn alphabeta_par(profondeur: u8, state: Configuration)
        -> Option<Movement> {
    if profondeur == 0 || state.movements().peekable().peek().is_none() {
        None
    } else {
        state.movements()
                .collect::<Vec<Movement>>()
                .par_iter()
                .map(|coup| (coup, -alphabeta(profondeur - 1, -127, 127,
                            state.play(&coup)).1))
                .max_by_key(|&(_, val)| val)
                .map(|(res, _)| *res)
    }
}
```

FIGURE 11 – Premier algorithme AlphaBeta parallèle

On remarquera que dans cet exemple, c'est *alphabeta* qui est appelé à

partir de $profondeur - 1$, mais en pratique n'importe lequel des algorithmes séquentiels pouvait être utilisé. En particulier, nous avons toujours utilisé *alphabeta_pvs* une fois que nous nous sommes rendus compte de ses performances.

Parallélisation plus "Rust" S'il y a quelque chose qui nous a frappé au cours de ce module, c'est la façon très particulière de programmer en Rust, différente des autres langages que nous avons vus jusqu'ici, avec beaucoup d'itérateurs et de fold ou map au lieu des traditionnels for.

Tout comme nous l'avons fait pour **Greedy** et **MinMax**, nous avons donc essayé de réécrire notre programme. Nous avions de plus l'espoir que réussir - pourquoi pas - à faire remonter certaines valeurs de alpha et beta si le nombre coups était grand au point de faire attendre quelques threads.

Pour cela, nous nous sommes tournés vers la méthode **try_fold** afin d'essayer d'utiliser l'enum "Result" permettant d'interrompre avec un **Err**. On fait ensuite un reduce pour renvoyer le meilleur résultat.
Voici le code que nous avons produit :

```rust
fn alphabeta_par_pvs2(profondeur: u8, mut alpha: i8, beta: i8, state: Configuration)
        -> (Option<Movement>, i8) {
    if profondeur == 0 || state.movements().peekable().peek().is_none() {
        (None, -state.value())
    } else {
        let (coup, val) = state.movements()
             .collect::<Vec<Movement>>()
             .par_iter()
             .try_fold(|| -> (Option<Movement>, i8) { (None, -127 as i8) },
                       |(best_move, best_val), coup| {
                let mut alpha2 = alpha;
                let vala = -alphabeta_pvs(profondeur - 1, -beta, -alpha2,
                            state.play(&coup)).1;
                let mut tmp_bc = best_move;
                let mut tmp_bv = best_val;
                if vala > best_val {
                    tmp_bv = vala;
                    tmp_bc = Some(*coup);
                    if tmp_bv > alpha {
                        alpha2 = tmp_bv;
                        if alpha2 >= beta {
                            return Err((tmp_bc, tmp_bv));
                        }
                    }
                    return Ok((tmp_bc, tmp_bv));
                } else {
                    return Ok((best_move, best_val));
                }
             })
             .reduce(|| -> Result<(Option<Movement>, i8), (Option<Movement>, i8)>
                    { Ok((None, 0)) }, |a, b| {
                        let (coup1, val1) = a.unwrap();
                        let (coup2, val2) = b.unwrap();
                        if coup1.is_none() {
                            return b;
                        } else if coup2.is_none() {
                            return a;
                        }
                        if val1 > val2 { a } else { b }
                    })
             .unwrap_or((None, -state.value()));
        (coup, val)
    }
}
```

FIGURE 12 – Algorithme parallele PVS final

C'est cet algorithme que nous avons utilisé lors du tournoi. Toutefois, nous ne sommes pas familiers avec certaines de ces structures, et il se peut que le coup renvoyé soit le dernier "**Ok**" au lieu du premier "**Err**". Cela mériterait davantage de tests.

Parallélisation sur n niveaux Nous avons également implémenté deux autres types d'algorithmes. Le premier consiste en une parallélisation sur deux niveaux (contre un), et le deuxième est une parallélisation totale (chaque niveau).

Ces algorithmes, nommés **alphabeta_par_pvs_double_depth_par** et **alphabeta_par_infinite** se sont toutefois montrés très mauvais. La raison que nous en avons déduite est que, comme nous l'évoquions, il est bien plus important de couper des branches que de paralléliser lorsque l'on veut optimiser alphabeta. Nous en sommes donc restés au compromis du premier niveau d'amélioration.

3 Réflexions, axes d'amélioration

3.1 Saut vs duplication

Tout d'abord, il pourrait être intéressant de tester des algorithmes favorisant des **Duplications** au lieu des **Jump** (et réciproquement). Nous pensons en effet que cela aurait une influence. En effet, lors du tournoi, nous avons constaté que notre algorithme avait tendance à sauter lorsque nous nous attendions à dupliquer. Toutefois, il est difficile de considérer qu'il s'agit là d'une erreur tactique ; à de telles profondeurs il est impossible pour l'être humain de comprendre la subtilité de telles variations sur le long terme de la partie.

3.2 Jouer "méchant"

On pourrait essayer d'isoler les pions adverses avec des murs de blobs dans l'optique de le pousser à passer son tour et ainsi conserver un avantage matériel. D'une part, certains groupes ont oublié d'interdire le "**None**" lorsqu'un coup est possible, d'autre part, le simple fait de passer son tour peut avoir des conséquences dramatiques, comme nous l'avons testé manuellement sur certaines parties de test.

3.3 Envisager d'autres fonctions de score

La fonction de score que nous avons est efficace, mais somme toute basique. On pourrait considérer que d'autres paramètres affectent la qualité de son jeu, comme par exemple la répartition des pions sur la carte : sont-ils

agrégés en une boule compacte ou éparpillés au quatre coins du plateau ?

En effet, laisser un trou peut parfois permettre à l'adversaire de pénétrer sa défense et de rapidement conquérir tout un côté du terrain. De plus, un pion sur un côté du terrain a moins de possibilités de déplacement qu'un pion situé en plein milieu, on pourrait imaginer qu'il a donc moins de valeur.

3.4 Transposition table / Refutation table

Un grand classique, que nous avons testé brièvement mais pas conservé consiste à stocker en mémoire les valeurs des grilles à certaines profondeurs. Une amélioration pourrait également être de transposer la grille, car de nombreuses grilles sont en réalité des symétries les unes des autres.

En outre, pour des algorithmes tels que ceux utilisant l'**aspiration search** ou le **PVS**, une table de hashage pourrait être très puissante car elle éviterait à de très nombreuses valeurs d'être recalculées.

Nous avions abandonné l'idée vers le début d'alphabeta, car nous nous étions rendu compte que très peu de grilles revenaient, et que le coût d'écriture et de recherche dans la HashMap était trop important par rapport aux bénéfices. Ce n'est que lorsque nous avons implémenté **PVS** que l'idée nous est revenue, mais il était trop tard pour en faire une implémentation très propre et efficace.

3.5 Améliorer PVS avec un tri

Notre algorithme PVS n'utilise pas de tri et calcule donc le premier coup qui n'est peut-être pas l'un des meilleurs. En combinant notre algorithme **sorted** (basé sur la valeur de la grille avec state.value, ou même $alphabeta(2)$ voire $alphabeta(n-2)$ qui est négligeable par rapport à $alphabeta(n)$, les coupes seraient encore plus impressionnantes et notre algorithme bien plus rapide.

3.6 Changer l'iterative deepening

3.6.1 Rendre l'iterative deepening plus tardif

Lors du tournoi, nous avons laissé l'iterative deepening partir de 1, parce que nous ne savions pas s'il était autorisé de le modifier (et nous avons considéré que très peu de monde le ferait, donc que ce ne serait pas fair-play de notre part). Toutefois on pourrait gagner du temps en commençant à 3, par exemple. Cette stratégie est risquée, car si l'on dépasse la limite des deux secondes on ne joue pas du tout alors que l'on aurait pu jouer "plutôt bien".

3.6.2 Memoizer les valeurs de l'iterative deepening

Au lieu de faire un tri comme je l'évoquais en sous-section 3.5 (cliquez), puisque PVS ne traite différemment que le premier élément, on pourrait simplement lui faire traiter celui renvoyé par l'iterative deepening à une profondeur $n-1$, déjà calculé, afin de gagner beaucoup de coupes.

3.6.3 Garder en mémoire les coups déjà calculés

Si l'on pouvait avoir une **HashMap** que l'on garderait d'un coup à l'autre, il pourrait alors être très avantageux de noter les scores calculés dans le passer et de simplement les actualiser pour la profondeur courante.

3.7 Autres algorithmes

Au cours de nos recherches, nous nous sommes intéressés à des implémentations qui pourraient concurrencer ou s'avérer meilleures qu'alpha-beta. Deux en particulier nous paraissaient prometteuses : la première est le **Monte Carlo Tree Search** (MCTS). En effet, cela nous aurait permis de renforcer nos connaissances sur les algorithmes probabilistes en nous faisant étudier un algorithme utilisé très fréquemment en intelligence artificielle (et en particulier pour les jeux tels que Blobwar).

Le second type d'algorithme que nous aurions aimé implémenter est le **Deep-Q Learning** (reinforcment learning). En effet, s'il est déjà très impressionnant de voir que nos algorithmes jouent mieux que nous, au point que nous ne comprenons pas leurs coups lorsque la profondeur devient supérieure à 2, il est à mes yeux encore plus impressionnant d'imaginer qu'un algorithme puisse "apprendre" par lui-même en expérimentant et en jouant contre lui-même.

Nous pensons donc, comme je vous le disais sur Riot, poursuivre ce projet durant l'été afin d'essayer de mieux comprendre ces algorithmes en les implémentant nous-même.

4 Choix de l'algorithme pour la compétition

Nous avons choisi de prendre l'algorithme Alpha-Beta PVS par 2, car il était notre algorithme le plus rapide. De plus, durant les affrontements entre les différents Alpha-Beta, c'est celui-qui sortait le plus souvent vainqueur.

5 Tests

Dès le début du projet, nous avons remarqué que prévoir le mouvement choisi par les algorithmes que nous allions créer était une tâche très compliquée. En effet, il faut être capable de voir la totalité des coups possibles, n coups à l'avance pour pouvoir espérer prévoir la sortie de l'algorithme. Face aux contraintes de temps auquelles nous avons dû faire face, nous avons décidé de ne pas chercher à faire de tests exaustifs de nos différents algortihmes, pour plutôt essayer de faire s'affronter nos algorithmes entre eux, afin de ne garder que ceux qui gagnent le plus souvent. Cela à été fait notamment grâce au module evil, que nous avons créé dans le but de faire s'affronter deux itérations d'un même algorithme. Cette manière de produire les tests était plus "artisanale", mais elle nous a permis de passer plus de temps sur la production d'algorithmes, tout en gardant un oeil sur la qualité de ceux-ci.

6 Conclusion

Ce projet nous a permis de mieux comprendre comment pour faire pour trouver le meilleur algorithme possible pour un problème donné, notamment via la recherche d'information sur internet et l'expérimentation individuelle. Cela nous a aussi permis de développer nos capacités en rust, étant donné que c'était le premier projet d'envergure dans ce langage. Nous aurions toutefois préféré avoir un peu plus de temps (ou avoir eu le projet à faire en début de semestre), car les contraintes d'emploi du temps nous ont empêché d'aller au fond des choses avec ce projet.

Nous avons eu l'occasion de lire beaucoup de littérature sur ces sujets, comme ceci ou encore cela.

J'aimerais (Alexandre) en outre remercier :
— **Michael Bleuez** pour ses conseils méthodologiques et sur les réflexions apportées sur ce projet et, dans un cadre plus général, sur l'algorithmique avancée.
— **Guillaume Ricard** qui m'a appris l'existence du try_fold, me permettant de développer notre seconde version des algorithmes parallèles.
— **Frédéric Wagner** pour sa disponibilité et sa bienveillance, qui m'a prodigué de conseils précieux au cours de ce projet ainsi que d'autres non liés à l'Ensimag.