INF8215 – Intelligence artificielle: méthodes et algorithmes

Automne 2021

Projet - Agent intelligent pour le jeu Quoridor

Groupe 05 - Équipe de *lampe14* sur Challonge

1960266 – Yanis Toubal

1947497 – Yuhan Li

Soumis à : Quentin Cappart

Dimanche 5 décembre 2021

1. Méthodologie

1.1 Fonctionnement de l'agent

Notre agent se base principalement sur l'algorithme *alpha-beta pruning*, vu lors du Module 2 du cours, pour prendre ses décisions d'actions. Par contre, notre agent n'emploie pas la stratégie *alpha-beta pruning* tout au long de la partie. Au début d'une partie Quoridor, il est codé de sorte à bouger le pion du joueur selon sa plus petite distance par rapport à la ligne d'arrivée. Cette distance est calculée à l'aide de l'implémentation de l'algorithme A* fournit. C'est lorsque la partie atteint un certain nombre de tours et de temps (chiffres déterminés expérimentalement) qu'on va commencer à considérer les placements de murs. Cette décision (placer un mur ou bouger le pion) est déterminée par notre méthode *h_alphabeta_search*, qui est basée sur l'algorithme *alpha-beta pruning*. Ensuite, arrivé à la fin de la partie et sous certaines conditions, notre agent va utiliser encore une fois la même stratégie qu'en début de partie. Dans les sections à suivre, nous allons voir plus en détails cet algorithme ainsi que les démarches entreprises pour concevoir notre agent.

1.2 Choix de conception

Notre choix pour l'algorithme alpha-beta pruning s'explique par le fait qu'il s'agissait de l'algorithme avec lequel nous étions le plus familier et c'est également celui qui nous semblait le plus intuitif à réaliser. Aussi, au vu de la limite de temps de 5 minutes, nous avons écarté l'algorithme d'arbre de recherche de Monte-Carlo puisque nous avons jugé que ce court laps de temps n'est pas suffisant pour générer assez de simulations pour avoir un bon agent.

1.2.1 Alpha-beta pruning

Lorsqu'on exécute l'algorithme du alpha-beta pruning, nous faisons appel à *h_alphabeta_search*, une méthode qui retourne la meilleure action selon les possibilités d'échanges avec l'adversaire. Ces possibilités d'échanges peuvent être vues comme un arbre, où chaque niveau représente les actions possibles d'un joueur. C'est justement en utilisant de façon récursive les fonctions de comparaison *max_value* et *min_value* que nous allons parcourir ces niveaux d'actions dans ledit arbre. Plus encore, les actions que ces deux fonctions analysent sont toutes filtrées au préalable par notre méthode *get_action*, expliquée en détails à la section 1.2.4 Spécificités de notre agent.

Bien entendu, par souci de temps et de performance, il faut arrêter de considérer à un certain point les possibilités d'actions. C'est ce que nous faisons avec notre méthode *cut_off*, sachant qu'on applique l'heuristique Minimax qui va évaluer la qualité de l'état courant pour notre argent. Dans notre cas, nous avons posé une profondeur par défaut de 25 pour le *cutoff*, et cette méthode est appelée à chaque niveau de l'arbre, soit autant dans *max_value* que dans *min_value*. Si on arrive au niveau de la profondeur en question ou bien si l'exploration de l'arbre atteint 5 secondes, la fonction *cut_off* fait effet en appelant une heuristique (section 1.2.2 Heuristique) qui va évaluer le score de l'état actuel. Toutefois, la profondeur par défaut est diminuée à 2 lorsque le temps restant est en dessous de 100 secondes ou lorsque nous avons effectué moins de 7 actions. Cela permet d'effectuer des mouvements qui sont plus droit au but et d'accélérer l'exécution de l'algorithme *alpha-beta pruning*.

1.2.2 Heuristique

Comme mentionnée à la section précédente, l'heuristique retourne un score qui reflète l'état courant du jeu. La formule de ce score est plutôt simple puisque le jeu en lui-même n'est pas très complexe. Tout d'abord, on considère la différence entre le nombre de pas avant de gagner de l'adversaire et le nôtre, donné par la méthode *get_score*, dans lequel on applique un facteur multiplicatif de 50 afin de donner un poids élevé à cette mesure. Ensuite, nous avons voulu considérer le nombre de murs restant de notre

agent par rapport à celui de son adversaire. Au final, en voulant donner une importance élevée à ce critère, nous avons augmenté leur différence de mur à une puissance de 2. Cela est un oubli maladroit de notre part, sachant que la valeur sera toujours positive. Il fallait donc garder un facteur multiplicatif comme nous l'avons fait initialement. Aussi, on considère l'état actuel de notre joueur et le nombre de murs. 100 points de pénalité sont attribués dans le cas où on a un score négatif et qu'on n'a plus de mur. 100 points en bonus sont attribués dans le cas où on a un score positif et que l'adversaire n'a plus de mur. Un autre oubli s'est glissé à ce stade du code, car nous avons tenu en compte le nombre de murs du jeu en entier plutôt que de l'état courant. Enfin, en tout dernier, on applique une grosse pénalité dans le cas où on perd la partie et un gros bonus dans le cas où on a gagné. Si le score final est positif, cela veut dire qu'il s'agit d'un état avantageux pour nous et, dans le cas où le score final est négatif, cela veut dire qu'il s'agit d'un état désavantageux (l'adversaire est avantagé).

```
90      def heuristic():
91          def estimate_score(game:Board , state: Board, player):
92              opponent = (player + 1) % 2
93              try:
94                  # Difference between lengths of my shortest path and of my opponent
95                  my_score = 50*state.get_score(player)
96              except NoPath:
97                  print("NO PATH estimate_score")
98
99              # Consider the remaining walls of each player
100             wall_comparison = (game.nb_walls[player]) - (game.nb_walls[opponent])
101             my_score += pow(wall_comparison, 2)
102
103             # If no walls left and player lost
104             if game.nb_walls[player] == 0 and my_score < 0:
105                 my_score -= 100
106             if game.nb_walls[opponent] == 0 and my_score > 0:
107                 my_score += 100
108
109             # Consider if our agents wins or loses
110             if state.pawns[player][0] == state.goals[player]: my_score += 1000
111             elif state.pawns[opponent][0] == state.goals[opponent]: my_score -= 1000
112             return my_score
113
114         return estimate_score
```

Figure 1 . Fonction *heuristic* qui calcule le score courant

1.2.3 Gestion du temps

Nous avons essentiellement déterminé l'allocation du temps de manière expérimentale. En effet, c'est en observant le temps écoulé lors de chaque partie que nous avons pu ajuster nos paramètres de façon optimale. Nous avons considéré le temps restant à deux endroits dans le code, soit dans la fonction *play* et dans la fonction *cutoff*. Pour la fonction *play*, lorsqu'il reste en bas de 45 secondes, nous arrêtons d'utiliser la stratégie *alpha-beta pruning* pour utiliser à la place la stratégie de déplacement dans le chemin le plus court. Cette décision s'explique par le fait que nous considérons qu'il ne reste plus assez de temps pour utiliser *alpha-beta pruning* et que la meilleure chance de gagner à ce moment là est de rapidement se diriger vers le camp ennemi en empruntant le chemin le plus court. Pour la fonction *cutoff*, nous considérons un *depth* (profondeur) de 2 lorsqu'il reste moins de 100 secondes. Ce choix s'explique par le fait qu'à ce stade, il est plus rentable de ne plus aller davantage en profondeur dans l'arbre afin de sauver du temps. Il s'agit là d'un compromis entre la qualité de la stratégie *alpha-beta pruning* et du coût en termes de temps. De cette sorte, l'agent semble réagir plus rapidement et plus directement aux coups de son adversaire. En effet, lors des tests avec un temps plus petit, on ne profitait pas assez de la recherche alpha-beta, et lors des tests avec un temps supérieur, l'agent semblait faire des actions illogiques.

1.2.4 Spécificités de notre agent

Nous considérons qu'il y a deux principales spécificités qui caractérisent notre argent. Tout d'abord, nous avons le filtrage des actions et ensuite nous avons le changement de stratégie de façon dynamique. Pour ce qui est du filtrage, nous avons filtré ses actions avec une heuristique qui ne garde que les placements de mur ayant une distance de Manhattan d'au plus 3 de l'adversaire ainsi que les placements de mur qui sont sur le chemin le plus court que peut emprunter l'adversaire. Cela a pour but de considérer les placements de mur à proximité de l'adversaire ou qui peuvent bloquer le chemin que peut emprunter l'adversaire. De cette sorte, on réduit considérablement le nombre d'action possible que la stratégie *alpha-beta pruning* va examiner. En retour, cela fait en sorte que l'agent joue de meilleures actions à chaque tour.

Ensuite, pour ce qui est du changement dynamique de stratégie, lors des débuts de partie on adopte une stratégie qu'on pourrait considérer gloutonne (greedy), telle qu'expliquée précédemment. Par la suite, on utilise la stratégie *alpha-beta pruning* pour la majorité de la partie avec également des changements dynamiques dans les paramètres selon des facteurs comme le nombre de murs ou encore le temps restant. Enfin, vers la fin de la partie, on s'oriente encore une fois vers une approche gloutonne afin de conclure la partie rapidement. Ce changement de stratégie fait en sorte qu'on s'adapte à la partie et on prend en compte les forces et faiblesse, en particulier en matière de temps, pour chacune des stratégies ainsi que du jeu en général.

2. Résultat et Évolution de l'agent

Tableau 1: Score des parties entre l'agent glouton et les diversions versions de notre agent

	Glouton (greedy)	Version 1	Version 2
Version 1	3-2	-	-
Version 2	5-0	3-2	-
Version finale	5-0	5-0	5-0

La première version de notre agent était assez basique. Nous avions uniquement implémenté le commencement d'une partie *hardcodée* ainsi que l'algorithme alpha-beta pruning, sans aucun filtrage et avec un manque de test exhaustif. Bien que cet agent arrivait parfois à gagner contre l'agent *greedy*, il possédait de nombreux défauts. D'abord, un des problèmes est qu'il avait tendance à manquer de temps. Le temps de 5 minutes s'écoulait sans que l'agent gagne, ce qui causait une défaite automatique. Cela est dû en partie au manque de détail dans l'implémentation de nos fonctions *cut_off*, ainsi qu'au manque de filtrage des actions considérées par notre algorithme *alpha-beta*. Ces lacunes ont également engendrées un problème au niveau des murs. En effet, l'agent semblait poser de façon assez aléatoire, sans aucune logique apparente, ses murs. Enfin, un dernier défaut majeur serait que l'agent était coincé dans une boucle infinie. Parfois, à un certain point de la partie lorsqu'il n'y avait plus de murs à placer, le pion effectuait des mouvements va et vient en continue. Malgré tout, cet agent arrivait à battre glouton pratiquement tout le temps lorsqu'il était du côté bleu, mais, lorsqu'il était du côté rouge, il avait tendance à perdre.

Pour la deuxième version, le changement majeur que nous avons apporté est une fonction qui filtre les actions possibles avant de les faire analyser par l'algorithme *alpha-beta pruning*. Cet ajout a pour but de

limiter le nombre d'actions à explorer en ne gardant que les meilleures actions. Pour ce faire, nous avons choisi de filtrer les actions de placement de murs puisqu'il y avait beaucoup de choix possibles pour ce type d'action. Nous avons pu observer une amélioration importante au niveau des placements des murs par l'algorithme ce qui s'est concrétisé en un pourcentage de victoire plus élevé. Par contre, un des grands défauts qui a persisté est celui dans lequel le pion de l'agent restait coincé dans une boucle infinie comme expliqué dans la version 1. On peut donc observer que cette version d'agent arrive très facilement à battre l'algorithme glouton. Par contre, au vu du défaut qui a persisté, la version 2 avait du mal contre la version 1.

Pour la troisième et dernière version de l'agent, mis à part certains ajustements dans les paramètres, nous avons détaillé l'heuristique, nous avons aussi perfectionné le filtrage des actions et nous avons réglé le bogue qui amène le pion dans une boucle infinie. Pour ce qui est de l'heuristique, nous avons considéré d'autres paramètres comme le nombre de murs de chaque joueur et si l'état courant est une victoire ou une défaite. Cela a pour but d'améliorer l'estimation d'un état et ainsi de mieux choisir une bonne action avec la stratégie alpha-beta pruning. Pour ce qui est du filtrage des actions, en plus de la distance de manhattan par rapport à l'adversaire, nous ajoutant aussi considéré placer des murs en périphérie du chemin le plus court que peut emprunter l'adversaire pour gagner. Cela a amélioré davantage le placement des murs. Enfin pour régler la principale cause du bogue avec le pion piégé dans une boucle infinie, nous utilisons le chemin le plus court pour gagner au lieu d'utiliser alpha-beta pruning lorsqu'il ne nous reste plus de murs. Ces trois principaux changements ont fait en sorte de régler tous les problèmes des versions précédentes ce qui fait en sorte que l'on gagne sans trop de difficulté contre le glouton ainsi que les versions 1 et 2.

3. Discussion sur l’agent final

3.1 Avantages

Tout d'abord, notre agent respecte la grande majorité du temps (>95%) la limite de 5 minutes d’une partie de jeu. En effet, nous avons pris beaucoup de précautions afin que l'agent prenne en compte le temps restant. De nombreux tests ont été effectués afin de trouver les meilleurs moments pour commencer la recherche *alpha-beta* ou pour activer le *cutoff*. De ce fait, nous sommes assurés que notre agent n’aura pratiquement jamais de défaite dû au temps écoulé. Ensuite, malgré le court laps de temps alloué à la recherche dans alpha-beta pruning, l'agent retourne la majorité du temps une action qui semble logique et bonne. Plus encore, ce choix d’actions se fait dans un délai assez raisonnable et optimal. Ces avantages sont causés grâce à la bonne implémentation de l’algorithme alpha-beta. Bien entendu, il y a tout de même place à amélioration dans la version de notre agent actuel. Cela serait assez facile à faire, sachant que le code est assez bien structuré et commenté pour des modifications ou des ajouts postérieures.

3.2 Limites

Nous avons remarqué que, dans de rares cas, notre agent jouait des actions assez mauvaises dans des situations très serrées ce qui le pénalise grandement dans la partie et faisait en sorte qu'il perdait la partie à cause de cela. Après beaucoup de travail, nous avons réussi à diminuer l'occurrence de ces situations sans toutefois résoudre entièrement le problème.

Plus encore, malgré les précautions prises dans le code, il arrive encore, dans moins de 5% des parties, que l'agent soit à court de crédit et finisse par perdre la partie dû au manque de temps. Nous avons gardé un œil attentif sur ce problème au cours des nombreux tests et au final nous avons jugés qu'il ne s'agissait pas d'un souci majeur au vu de sa rare occurrence.

3.3 Tentative d'amélioration

Nous avons tenté d'implémenter une table de transposition ou en d'autres mots une mémoire. À travers, les notions apprises nous savions que ce mécanisme permettrait d'accélérer grandement l'exploration de l'arbre en évitant de réévaluer plusieurs fois le même chemin. Après quelques recherches, nous avons décidé d'utiliser le *Zobrist Hashing* (Aradhy, 2019) qui semblait être très approprié pour un jeu comme Quoridor pour identifier un état avec un hash unique. Nous avons donc généré trois tableaux, soit un tableau pour la position des pions, un tableau pour la position des murs horizontaux et un tableau pour la position des murs verticaux que nous avons remplis de nombres aléatoires allant jusqu'à 2^{64} afin d'éviter les collisions de hash. Plus encore, nous avons deux *maps* qui vont contenir respectivement les meilleurs min actions et les meilleurs max actions d'un état représenté par le *hash*. Dans les deux dernières images de la Figure 2, on peut observer la façon dont on a appliqué la mémoire pour la partie max du alpha-beta pruning. Par contre, lors de l'utilisation de ce mécanisme, nous avons eu quelques bogues avec l'agent notamment un où l'agent restait coincé dans une loupe infinie. Notre manque d'expérience avec ce mécanisme ainsi que le manque de temps nous a empêché, au final, de garder ce code.

```
state_table_pawn = [[random.randint(0, 2**64) for _ in range(10)] for _ in
range(10)]
state_table_horizontal_wall = [[random.randint(0, 2**64) for _ in range(10)
] for _ in range(10)]
state_table_vertical_wall = [[random.randint(0, 2**64) for _ in range(10) ]
for _ in range(10)]
min_values_map = {}
max_values_map = {}

def calculate_hash(self, board):
    hash = 0
    for (x,y) in board.pawns:
        hash ^= self.state_table_pawn[x][y]

    for (x,y) in board.horiz_walls:
        hash ^= self.state_table_horizontal_wall[x][y]

    for (x,y) in board.verti_walls:
        hash ^= self.state_table_vertical_wall[x][y]

    return hash
```

```
def max_value(state: Board, alpha, beta, depth):
    hash = self.calculate_hash(state)
    if hash in self.max_values_map:
        return self.max_values_map[hash]
```

```
    if v >= beta:
        self.max_values_map[hash] = v_star, m_star
        return v_star,m_star
self.max_values_map[hash] = v_star, m_star
return v_star,m_star
```

Figure 2. Tentative d'implémentation d'une mémoire

Références

Aradhy, A. (25 juillet 2019). *Minimax Algorithm in Game Theory | Set 5 (Zobrist Hashing),* GeeksforGeeks.https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-5-zobrist-hashing/