POLYTECHNIQUE MONTRÉAL

UNIVERSITÉ D'INGÉNIERIE

Département de génie informatique et génie logiciel

INF8215
Intelligence artificielle : méthodes et algorithmes

Projet - Agent Intelligent pour le jeu Quoridor
Équipe Challonge: PouletDansCage

Soumis auprès de Quentin Cappart (Ph.D.)
par **William Balea (1904905) et Jean-Michel Lasnier (1905682)**
Automne 2021 (5 décembre 2021)

Pour les lecteurs de ce rapport, prenez note que lorsque nous faisons référence au code dans une capture d'écran, la ligne ou section du code en question sera spécifié de cette façon:

[xx - xx] "Dans ces lignes, nous voyons que..."

1-Mise en contexte/spécification

Dans le cadre du cours INF8215, nous avons implémenté une intelligence artificielle capable de jouer au jeu Quoridor. Le but a été de mettre en pratique les techniques apprises en classe. Le jeu se joue à deux joueurs, sur une planche de 9x9 cases, tour par tour. Les pions sont placés sur l'un des côtés de la planche. Pour gagner une partie, un joueur qui commence d'un côté de la planche doit se rendre au côté opposé. Pour empiéter le chemin du joueur, l'adversaire peut mettre jusqu'à 10 murs horizontaux ou verticaux d'une longueur de deux cases. Nous allons ici démontrer notre approche choisie afin de créer un adversaire virtuel écrit en langage Python et qui a une contrainte de temps de 5 minutes pour faire ses réflexions. Nous allons détailler le fonctionnement de notre agent, nos choix de conceptions ainsi que les limites de notre AI.

2-Méthodologie/choix de la méthode (AlphaBeta)

Pour notre agent intelligent, nous avons décidé d'utiliser la méthode d'élagage AlphaBeta. Nous avons décidé d'y aller selon cette voie parce que c'était une technique qui nous était familière et nous avons beaucoup d'exemples sur quoi nous baser. De plus, à cause de la similarité du jeu Quorridor avec les échecs, nous pensons qu'AlphaBeta serait une approche judicieuse pour entreprendre ce projet. En effet, cette stratégie de recherche est employée par le fameux logiciel Stockfish qui est un des meilleurs logiciels aux échecs.

Nous savions que plusieurs améliorations à la méthode de base peuvent être implémentés afin d'obtenir un agent de plus en plus robuste. En effet, l'ordre dans lequel les actions sont évaluées permet de pruner l'arbre de recherche afin d'améliorer la vitesse du calcul des mouvements. On peut ajouter à cela une table de transposition pour éviter de faire des calculs redondants. Nous pouvons même ajouter une profondeur maximale de recherche et intégrer des heuristiques afin de diriger la recherche. Bref, la méthode AlphaBeta offre plusieurs possibilités qui nous semblaient intéressantes à expérimenter afin d'obtenir l'agent intelligent idéal.

Voici les grandes lignes sur comment l'algorithme AlphaBeta fonctionne. Tout d'abord, il faut savoir qu'AlphaBeta se base sur la recherche Minimax. Considérons le cas suivant:

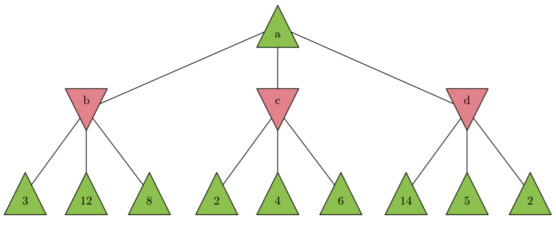


Figure 1: Représentation d'un arbre de recherche de l'algorithme minimax

Partons du joueur en **a** (triangle vert) qui liste toutes ses possibilités. Pour chacun des choix de mouvements, on obtient un état de la planche **b, c** et **d** (triangles rouges). De ces états, l'adversaire listera toutes ses possibilités de mouvements. On descend alors comme cela dans l'arbre en alternant entre les mouvements du joueurs et celui de l'adversaire. En atteignant les états finaux, nous pouvons calculer le score de la partie. Ensuite, pour propager le score vers le premier état et pour choisir la meilleure action à effectuer, les états de l'ennemi (triangles rouges) tenteront de choisir l'action qui minimise le score tandis que les états du joueur (triangles verts) tenteront de choisir l'action qui maximise le score. Dans notre

exemple, parmi les scores finaux de l'arbre, **b** choisira l'action qui minimise le score à 3, **c** choisira celle qui donne un score de 2 et **d** celle qui donne aussi 2. Ensuite, **a** choisira l'action qui maximise le score parmi **b, c,** et **d** : c'est l'action menant à l'état **b** avec un score de 3.

Puisque le nombre d'actions possibles est grand, l'arbre de recherche peut être très grand et la recherche de la bonne action peut s'éterniser. C'est là qu'intervient l'algorithme AlphaBeta qui permet de couper des branches de recherches inutiles qui n'ont pas d'impact sur les valeurs minimax et ainsi augmenter l'efficacité. Cela est fait en maintenant deux variables : **alpha** qui retient le score minimum que le joueur (triangle vert) va obtenir et **beta** le score maximum que le joueur peut obtenir. Si le score d'un nœud rouge est inférieur à alpha, on exclut les autres possibilités d'actions de ce nœud de l'arbre. Si le score d'un nœud vert est supérieur à beta, on exclut les autres possibilités d'actions de ce nœud de l'arbre.

De plus, nous pouvons limiter le temps de recherche en limitant la profondeur maximale de la recherche dans l'arbre. Alors, au lieu de prendre le score final de la partie, nous pouvons instaurer des heuristiques afin de pouvoir quantifier les avantages en jeu d'un joueur par rapport à l'autre : qui est plus proche de l'arrivée? Qui possède le plus de murs restants? Etc.

3-Caractérisation du code

Le code de notre joueur est divisé en 5 sections: la fonction de jeu Play, l'algorithme AlphaBeta, l'heuristique utilisé, une fonction pour réduire le branchement et quelques fonctions utilitaires. Dans cette partie du rapport, chaque section sera décrite plus en détails

FONCTION PLAY

La fonction play est appelée à chaque tour du joueur. C'est donc ici que nous déclarons les variables initiales importantes, prenons des décisions sur l'état du jeu et appelons notre algorithme Minimax-Alphabeta.

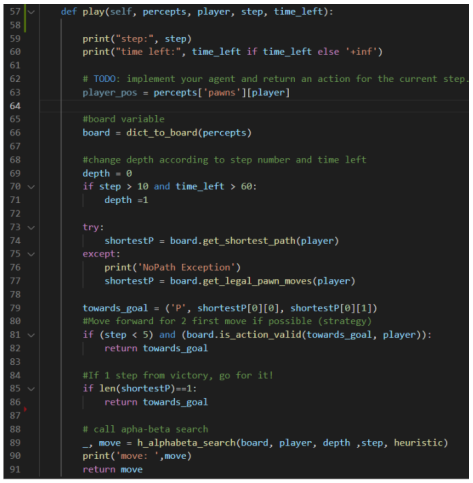


Figure 2: Code de la fonction Play()

[69-71] Tout d'abord, nous changeons la profondeur de recherche dans l'arbre selon le nombre de tours (step) et le temps restant . Puisqu'en début de partie les décisions sont moins critiques, nous laissons une profondeur de 0 lors des 10 premiers tours. Ensuite, une profondeur de 1 est utilisée jusqu'à ce qu'il reste 60 secondes au joueur. Cela lui permet de prendre de meilleures décisions en milieu de partie mais assure de ne pas perdre la partie par manque de temps.

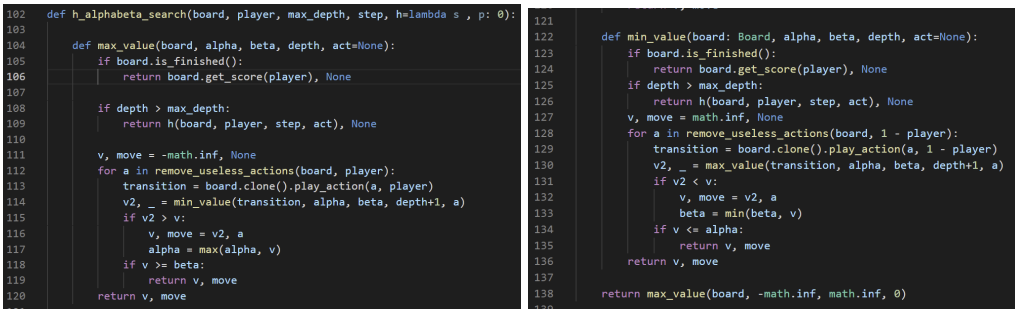
[73-79] Ensuite, cette section est utilisée pour forcer le joueur à avancer par en avant dans certaines situations bien précise. La variable 'towards_goal' est simplement le 1er mouvement à faire dans le 'shortest_path'. Par contre, il est à noter que la fonction 'get_shortest_path' retourne parfois des mouvements non-valides par erreur causant une défaite. Pour contrer ce problème, nous utilisons un 'try-except' par sécurité et dans le cas d'une erreur, nous prenons simplement le premier élément de la liste des déplacements légaux retournée par 'get_legal_pawn_moves()'.

[81-82] La première situation où 'towards_goal' est utilisée est en début de partie. On oblige notre joueur à avancer par en avant pour les 2 premiers mouvements lorsque possible. C'est la stratégie de début de partie que nous avons choisie permettant à l'agent de gagner du terrain tôt. De plus, le forcer à avancer ici sans appeler AlphaBeta sauve du temps de calcul pour le reste de la partie.

[86-85] La deuxième situation est lorsque le joueur est à 1 pas de la victoire. Pour des raisons inconnues, notre joueur était incapable de terminer une partie lorsqu'il lui restait 1 seul pas à faire. Le forcer à avancer a réglé notre problème.

[89] Finalement on peut trouver l'appel de l'algorithme 'h_alphabeta_search'. Le board de jeu, le joueur, la profondeur et le step sont passés en paramètres.

FONCTION H_ALPHABETA_SEARCH(...)



```python
def h_alphabeta_search(board, player, max_depth, step, h=lambda s , p: 0):

    def max_value(board, alpha, beta, depth, act=None):
        if board.is_finished():
            return board.get_score(player), None

        if depth > max_depth:
            return h(board, player, step, act), None

        v, move = -math.inf, None
        for a in remove_useless_actions(board, player):
            transition = board.clone().play_action(a, player)
            v2, _ = min_value(transition, alpha, beta, depth+1, a)
            if v2 > v:
                v, move = v2, a
                alpha = max(alpha, v)
            if v >= beta:
                return v, move
        return v, move
```

```python

    def min_value(board: Board, alpha, beta, depth, act=None):
        if board.is_finished():
            return board.get_score(player), None
        if depth > max_depth:
            return h(board, player, step, act), None
        v, move = math.inf, None
        for a in remove_useless_actions(board, 1 - player):
            transition = board.clone().play_action(a, 1 - player)
            v2, _ = max_value(transition, alpha, beta, depth+1, a)
            if v2 < v:
                v, move = v2, a
                beta = min(beta, v)
            if v <= alpha:
                return v, move
        return v, move

    return max_value(board, -math.inf, math.inf, 0)
```

Figure 3: Code de la fonction h_alphabeta_search()

Dans cette section, on retrouve le cœur de l'algorithme AlphaBeta. Il est important de mentionner que ce code est grandement inspiré du laboratoire 2. Alphabeta a 2 fonctions principales: ‘max_value()’ et ‘min_value()’. Ces fonctions sont très similaires et ont comme rôle d’explorer une séquence d’actions à l’aide d’une heuristique et ensuite de sélectionner l’action maximisant ou minimisant le score (dépendamment de si on se trouve dans un max_value ou min_value). Puisque ces deux fonctions sont exactement pareilles à l’exception de ce qui a été expliqué précédemment, seule la fonction ‘max_value’ sera décrite plus en détail.

[105-106] Permet d’arrêter l’exploration de l’arbre une fois qu’on a trouvé un état vainqueur.

[108-109] Cette condition assure d’arrêter l’exploration de l’arbre à une certaine profondeur maximale. Ce mécanisme est utilisé pour une question de performance. Plus l’arbre est exploré en profondeur, plus l’agent voit des scénarios d’avance. Par contre, le temps de calcul augmente exponentiellement selon la profondeur. Dans notre cas, nous utilisons une profondeur de 0 pour les 10 premiers steps et une fois qu’il reste moins de 60 secondes à la partie. Une profondeur de 1 est utilisée dans les autres cas. Lorsque la profondeur maximale est atteinte, on appelle l’heuristique pour évaluer l’état de la planche de jeu..

[111-119] Chaque action possible est évaluée et on peut apercevoir l’appel récursif de ‘min_value’.

[112] Pour permettre à l’algo d’atteindre une profondeur de 1 avec un temps raisonnable, la fonction ‘remove_useless_actions()’ est utilisée pour réduire le branchement. Cette méthode retourne un sous-ensemble des actions possibles et sera détaillée plus loin dans ce rapport. Les variables ‘alpha’, ‘beta’ sont utilisées pour la logique du pruning servant à réduire le branchement. Ici encore, ce mécanisme a pour unique but de réduire le branchement de l’arbre exploré et donc le temps de calcul à chaque appel de l’algo. Cela ne permet pas de prendre de meilleures décisions.

HEURISTIQUE

Le rôle d’une fonction heuristique est de retourner un score décrivant l’état actuel du plateau de jeu. Lorsque plusieurs actions sont évaluées, nous obtenons donc un score pour chacune de ces actions. Ceci permet à l’algorithme AlphaBeta de sélectionner l’action maximisant ou minimisant les gains du joueur.

```python
def heuristic(board: Board, player, step, act):
    #[sh_path_player, sh_path_opponent, wall_player, wall_ennemy]
    percentage = [2, 1.8, 0.5, 0.5, 0.4]
    percentage_player_plusproche = [2.2, 1.5, 0.7, 0.5]
    percentage_ennemy_plusproche = [1.7, 1.5, 0.4, 0.3]

    score = board.get_score(player)
    if score >0:
        percentage = percentage_player_plusproche
    elif score< 0:
        percentage = percentage_ennemy_plusproche

    try:
        sh_path_player = -len(board.get_shortest_path(player))
        sh_path_opponent = len(board.get_shortest_path(1 - player))
    except:
        print('NoPathError')
        sh_path_player = manhattan(board.pawns[player], board.goals[player])
        sh_path_opponent = manhattan(board.pawns[1-player], board.goals[1-player])

    wall_player = board.nb_walls[player]
    wall_ennemy = -board.nb_walls[1 - player]

    return (sh_path_player * percentage[0] + sh_path_opponent * percentage[1]
            + wall_player * percentage[2] + wall_ennemy * percentage[3])
```

Figure 4: Code de la fonction heuristique()

[156-168] Le score est calculé à l’aide de 4 éléments: la longueur du chemin le plus court de notre joueur, la longueur du chemin le plus court de l’adversaire, le nombre de murs de notre joueur et le nombre de murs de l’adversaire. Il nous semblait effectivement pertinent d’utiliser les chemins les plus courts car c’est un bon indicateur de qui a l’avantage de la partie à tout moment. Un mur qui rallonge le parcours de l'adversaire de 4 steps aura beaucoup plus d’impact que simplement avancer notre joueur de 1 case par exemple.

En plus, nous considérons le nombre de murs de chaque joueur puisque ceux-ci ont un grand impact sur le déroulement du jeu. On veut donc s'assurer que notre joueur ne gaspille pas de murs inutiles et l'inciter à garder l'avantage numérique des murs par rapport à son adversaire.
[146-148] Chaque élément du score est multiplié par un pourcentage, ce qui nous permet de changer dynamiquement les priorités de notre agent selon la situation du jeu.
[150-154] Nous avons établi 3 différentes répartitions des pourcentages en fonction de la valeur retournée par 'get_score()'. L'idée est que si notre joueur est plus proche de son but que l'adversaire, on veut favoriser les déplacements. Par contre, lorsque l'adversaire est plus proche de son but, alors on veut encourager notre agent à placer des murs. On met donc plus en valeur la valeur du plus court chemin de l'adversaire et réduisons l'importance de garder nos murs. En d'autres mots, cela permet un comportement plus aggressif/offensif et un plus défensif.
Les valeurs de pourcentage ont été déterminées en expérimentant au cours de plusieurs parties par essais et erreurs.

REMOVE_USELESS_ACTIONS()

Cette méthode est utilisée pour réduire le branchement de l'arbre et ainsi permettre d'aller chercher une profondeur maximale dans l'arbre avec des performances tout de même acceptables. Les mouvements intéressants conservés sont les déplacements du joueur, les murs autour de l'adversaire ainsi que les murs autour de notre joueur. Initialement, on gardait aussi les murs autour de ceux déjà posés, mais l'idée a été abandonnée pour la remise du code puisque cela allongeait beaucoup de temps de calcul et ça n'avait pas un grand impact sur le talent de notre joueur.

```python
def remove_useless_actions(board: Board, player):
    actions = board.get_actions(player)
    ennemy_pos = board.pawns[1-player]
    my_pos = board.pawns[player]
    walls = board.horiz_walls + board.verti_walls

    good_action=[]
    for a in actions:
        a_pos = (a[1],a[2])
        #keep moving actions
        if a[0]=='P':
            good_action.append(a)
        #keep walls around ennemy
        elif walls_around_ennemy(a_pos, ennemy_pos):
            good_action.append(a)
        #keep walls around my_player
        elif walls_around_player(a_pos, my_pos):
            good_action.append(a)
        # #keep walls close to other walls
        # for w in walls:
        #     if manhattan(a_pos, w)<1:
        #         good_action.append(a)

    return good_action
```

Figure 5: Code de la fonction remove_useless_actions()

FONCTIONS UTILITAIRES

```python
#======================UTILS======================#
def manhattan(pos1, pos2):
    return abs(pos1[0] - pos2[0]) + abs(pos1[1] - pos2[1])

def walls_around_ennemy(action, player):
    #is wall next to player
    return (-2 <= (action[0] - player[0]) <=1) and (-2 <= (action[1]-player[1]) <= 1)
def walls_around_player(action, player):
    #is wall next to player
    return (-2 <= (action[0] - player[0]) <=1) and (-2 <= (action[1]-player[1]) <= 1)
```

Figure 6: Code des fonctions utilitaires

Quelques fonctions supplémentaires ont été utilisées.
[195-196] D'abord, la distance manhattan est calculée dans l'heuristique lorsque 'get_shortest_path()' retourne une erreur. Elle était aussi utilisée pour garder les murs proches de ceux déjà posés dans 'remove_useless_actions()'.
[198-203] Ensuite, 'walls_around_ennemy()' et 'walls_around_player()' sont utilisés pour conserver les murs autour des joueurs dans un rayon prédéterminé dans 'remove_useless_actions()'. Initialement, un rayon plus grand était utilisé autour de l'adversaire mais ce rayon a été changé et les distances sont les mêmes pour les 2 joueurs.

4-Difficultés, avenues explorés et améliorations

Cette section sera utilisée pour expliquer certaines difficultés rencontrées au cours du développement du projet ainsi que les avenues explorées.

HEURISTIQUE

Un gros problème que nous avons rencontré était que notre joueur commençait tout d'un coup à prendre des mauvaises décisions et reculait dans la mauvaise direction, alors qu'il avait l'avantage et allait gagner. Ce problème nous a hanté pendant plusieurs jours, et la cause

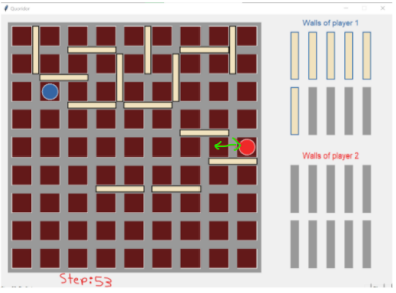


Figure 7: Board représentant l'erreur de l'heuristique obtenue

était le fait que nous mettions un plus grand poids à la longueur du ‘shortest_path’ de l'ennemi qu’à notre propre ‘shortest_path’ dans certaines situations. Après avoir investigué en affichant dans le terminal les détails des scores calculés, nous avons conclu qu’il fallait toujours mettre un poids plus élevé au chemin le plus court de notre joueur plutôt que de l’adversaire. Dans l’image ici, le joueur allait de gauche à droite sans arrêt plutôt que d’avancer vers le haut (son but).

PERFORMANCE

Un autre limitation est le depth maximal de l’algo. La profondeur maximale de notre algorithme était de 0 si nous gardions toutes les actions possibles. La méthode pour garder seulement un sous-ensemble des actions nous a permis d’aller chercher une profondeur de 1 au moins (donc de faire un max-min-max).

Une autre façon d’améliorer la performance de notre algo aurait été d’ajouter une cache. Une cache permet de sauvegarder l’action posée pour un certain état du jeu. On peut ensuite, lors de parties subséquentes, aller chercher rapidement l’action à poser plutôt que de recalculer l’AlphaBeta si on a déjà vu cet état. Nous avons commencé l’implémentation d’une cache mais avons décidé de ne pas l’inclure au joueur final pour plusieurs raisons. Tout d’abord, la cache fonctionnait juste pour le joueur bleu et nous avons manqué de temps pour le faire fonctionner des 2 côtés. De plus, notre heuristique n’était pas prête et si nous avions utilisé la cache, nous aurions sauvegardé des actions non-optimales. Avec plus de temps, nous aurions terminé la cache et nous aurions pu jouer plusieurs parties pour peupler son répertoire. Voici une capture d’écran de la cache commencé:

```
86          # Tentative de cache
87          minimal_state = (player, board.pawns, board.horiz_walls, board.verti_walls)
88          # cache array of (minimal_state, move)
89          cache = pickle.load(open("cache.p", "rb"))
90          for i in cache:
91              if i[0] == minimal_state:
92                  print("found move in cache!")
93                  if board.is_action_valid(i[1], player):
94                      return i[1]
```

Figure 8 : Tentative d’implémenter une cache avec la librairie Pickle

Une autre avenue que nous aurions pu explorer est de placer dans un ordre spécifique la liste des actions à évaluer. Le pruning est beaucoup plus efficace si l’action la plus intéressante est évaluée en premier. Il aurait donc été intéressant de tenter d’organiser la liste des actions utiles en plaçant d’abord les actions généralement intéressantes. Par exemple: déplacement par en avant, les murs proches de l'adversaire, etc.

MONTE CARLO TREE SEARCH

Enfin, nous avons aussi expérimenté en changeant complètement d’algorithme et en implémentant la méthode de Monte Carlo Tree Search (MCTS). L’avantage de cette approche est qu'il est possible d’explorer plus profondément l’arbre de recherche et que l’algorithme ne dépend pas autant de la qualité de l’heuristique implémentée. Nous voulions voir si MCTS et sa simulation aléatoire allait être plus efficace que notre AlphaBeta implémenté. Malheureusement, notre tentative d’implémentation a échoué et, après avoir programmé toutes les étapes, MCTS ne nous donnait pas les résultats souhaités. L’agent semblait faire n’importe quoi malgré le nombre d’itérations de simulations fixé à 6. De plus, au-delà de ce nombre, les décisions de MCTS prenaient trop de temps et nous ne voyons plus les avantages de continuer avec cette méthode.

5-Conclusion

En conclusion, l’algorithme AlphaBeta que nous avons implémenté comprend plusieurs éléments intéressants dont une heuristique dynamique qui change selon l’état du jeu, un filtre pour éliminer les actions moins intéressantes et un mécanisme pour changer la profondeur de l’arbre exploré selon le nombre de tours joués et le temps restant. Tout cela nous a permis d’obtenir un agent performant capable de vaincre de coriaces adversaires tel que greedy_player ainsi que des humains expérimentés au jeu de Quoridor.