Mini-Projet d'Intelligence Artificielle

CORENTIN BANIER and YOUSRI LAJNEF

Corentin Banier and Yousri Lajnef. 2023. Mini-Projet d'Intelligence Artificielle. 1, 1 (February 2023), 3 pages. **https://gitlab.univ-nantes.fr/E21B837U/hex-project**

Dans notre code, nous avons trois méthodes minimax : `minimax_strategy`, `minimaxAB_strategy` *et* `minimaxAB_bestChoice`. *Les deux premières méthodes sont les algorithmes classiques de l'implémentation de minimax et ou sans α et β. La dernière,* `minimaxAB_bestChoice` *est notre stratégie minimax qui comporte toutes les optimisations que nous avons pu mettre en place et qui sont décrites ci-dessous.*

1 DÉTAILS SUR L'IMPLÉMENTATION DU CODE

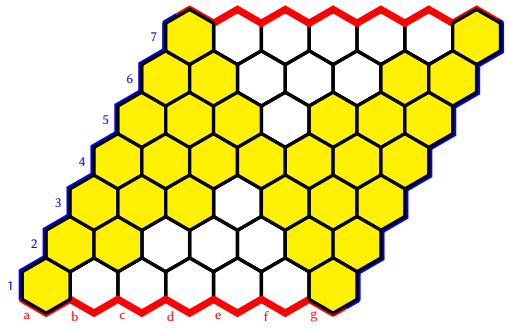
Par convention, tous les noeuds en jaune dans les figures symbolisent les noeuds qui sont explorés par l'algorithme.

1.1 Heuristiques

1.1.1 Exploration des noeuds sur une grille vide.

Fonction : `STRAT.first_move_choose`
Cette heuristique est appelée uniquement au début d'une partie, lorsque que le joueur ayant la stratégie `minimaxAB_bestChoice` débute le jeu.

Voici la configuration décrite lorsque le joueur qui commence est le `bleu` :



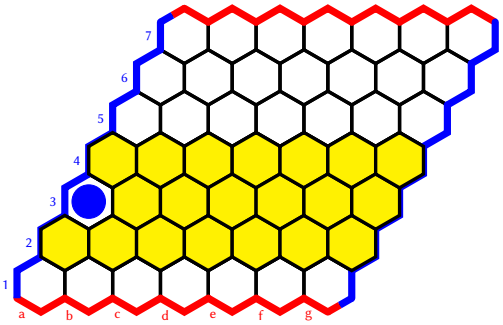
1.1.2 Exploration des noeuds sur une grille non vide.

Fonction : `Node.get_moves_to_explore`
Cette heuristique est appelée dès que l'on cherche à créer un étage de plus dans l'arbre de recherche.

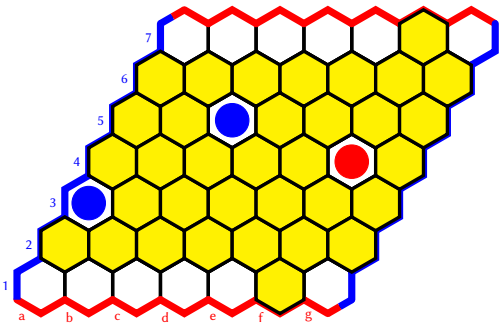
L'heuristique a pour but de réduire le nombre de noeuds à explorer. Le principe consiste à regarder l'ensemble des cases vides qui sont voisines de toutes les cases de l'adversaire, puis, pour toutes ces cases, on explore toutes les cases vides qui mène l'adversaire à ses deux bordures. De plus, parmi toutes les cases que le joueur jouant la stratégie possède, on va chercher à regarder toutes les cases vides qui tracent un chemin entre les deux bords qui lui appartiennent. L'heuristique prend du sens au début des parties, là où la combinatoire est très élevée.

Prenons des exemples :

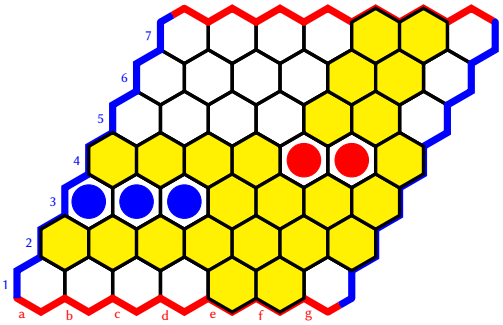
Voici la configuration décrite lorsque le joueur qui commence la partie est le `bleu` et que le joueur `rouge` suit la stratégie `minimaxAB_bestChoice` :



Si on avance un peu plus dans la partie, nous aurons la configuration suivante :



Dans la figure précédente, nous ne percevons pas énormément le gain vis-à-vis de l'exploration de tous les noeuds car les pions du joueur `bleu` sont éparpillés. Cependant, si l'on regarde une grille où les coups du joueur adverse sont rapprochés (ici, cela va simuler un comportement plus humain), le gain est significatif :



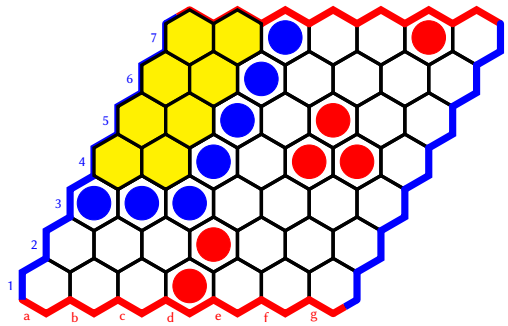
Ici, avec une profondeur $depth = 4$, et en étant dans la pire configuration possible, c'est-à-dire celle où l'algorithme doit descendre à la profondeur $depth = 0$, on a $24 \times 23 \times 22 \times 21 = 255024$ noeuds qui sont explorés. Toutefois, dans une configuration normale, c'est à dire en faisant appel à `node.untried_move`, nous aurions parcouru l'ensemble des noeuds disponibles, c'est-à-dire

$44 \times 43 \times 42 \times 41 = 3258024$ noeuds, soit un facteur 13 fois plus important.

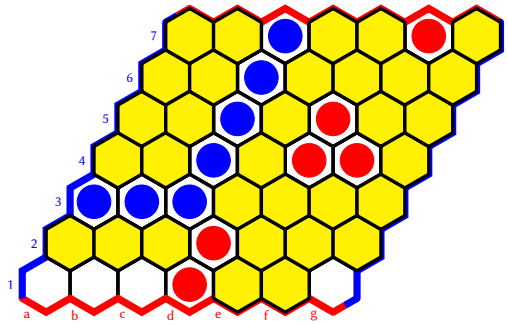
1.1.3 *Suppression des noeuds en embuscade - non implémenté.*

Dans certaines configurations du jeu, il y a des noeuds qui ne peuvent pas mener à la victoire, ce sont des impasses. Nous avons essayé de mettre en place cette heuristique afin de supprimer ces noeuds de l'exploration, mais sans succès.

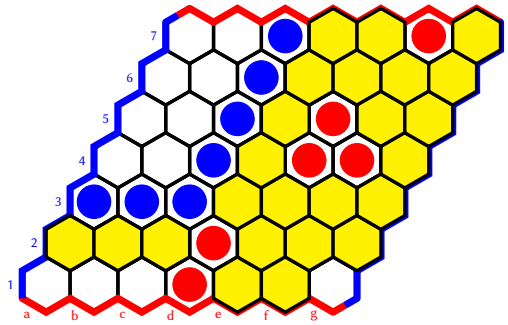
Par exemple, les noeuds `jaunes` ci-dessous sont des noeuds qui ne peuvent pas mener à la victoire pour le joueur `rouge`. De plus, ils sont aussi inutiles pour le joueur `bleu` car celui-ci à déjà un chemin provenant d'un de ces bords.



Le code, dans l'état et dans la configuration précédente va explorer les noeuds suivants :



Avec l'implémentation décrite précédemment, nous aurions eu l'exploration suivante :



Ainsi, avec une profondeur $depth = 4$ et en étant dans la pire configuration possible, c'est-à-dire celle où l'algorithme doit descendre à la profondeur $depth = 0$, actuellement, $32 \times 31 \times 30 \times 29 = 863040$ noeuds sont explorés tandis qu'avec l'heuristique décrite précédemment, nous aurions parcouru $24 \times 23 \times 22 \times 21 = 255024$ noeuds, soit un facteur d'environ 3.5 fois moins important.

1.1.4 *Optimisation des coups joués.*

Fonction : `STRAT.choose_best_move`

Avec une stratégie de type `minimax`, avec ou sans α, β, nous pouvons nous retrouver dans des situations où un choix arbitraire devra être fait (`random`).
Nous avons voulu optimiser ce choix, en prenant en compte les choses suivantes :

- Les valeurs de `minimax` évidemment, sinon on change le principe propre à l'algorithme.
- La longueur du chemin qui mène à une victoire. En effet, que cela soit pour gagner ou bien pour contrer l'adversaire, la meilleur façon de la faire est de le faire le plus rapidement possible. Ainsi, si nous avons plusieurs chemins gagnants, nous veillons à choisir celui le plus court.
- La profondeur de l'arbre à laquelle nous avons une solution viable. Ce paramètre vient compléter le précédent, c'est-à-dire qu'ici aussi, cela s'applique s'il l'on veut gagner ou bien contrer son adversaire.
  Dans tous les algorithmes `minimax`, nous avons fait le choix de définir une profondeur, i.e. $depth$ =?, puis de décrémenter cette valeur à chaque appel récursif jusqu'à ce que $depth = 0$. Ce qui fait que dès lors que nous trouvons une succession de noeud gagnante, nous faisons aussi remonter la valeur de la profondeur afin de finalement savoir en combien de coup, ce chemin est réalisable. L'optimisation consiste donc à maximiser la valeur $depth$ que tous les appels récursifs font remonter. Plus cette valeur sera élevée, plus nous devrons jouer le coup rapidement.

  Par exemple, si mon adversaire gagne en jouant en $(0, 4)$ avec $depth = 3$ Eet que moi, je peux gagner si joue en profondeur $(3, 5)$ avec $depth = 2$, cela veut dire qu'en réalité, pour gagner, je devrais joueur un tour de plus. Par conséquent, il vaut mieux contrer l'adversaire au risque que la partie se termine.

2 MÉTRIQUES

Nous avons implémenté différentes métriques que nous avons jugées utiles et nous permettant de comparer plusieurs approches de stratégie entre elles. Leurs méthodes d'affichage sont implémentées dans la classe `Tournament` de `tournament.py`.

2.1 Nombre de tours joués

On a dans un premier temps un compteur du nombre de tours initialisé dans la fonction `__init__()` de la classe `Game` et incrémenté à chaque tour dans la fonction `run_turn()` de la même classe. Cette valeur est ensuite retournée dans la fonction `get_game_info()` appelée dans la fonction `single_game()` de la classe `Tournament` qu'il nous reste simplement à afficher.

2.2 Nombre de coups joués

Cette même valeur du nombre de tours joués nous permet ensuite de déterminer et d'afficher le nombre de coups joués par chaque joueur en fonction du joueur qui commence.

2.3 Nombre de parties remportées / Taux de victoire

On crée un dictionnaire `win_count` dans la fonction `championship()` qui va être utilisé pour comptabiliser le nombre de victoires de chaque joueur en utilisant la clé 1 pour le joueur `noir` et la clé 2 pour le joueur `blanc`. Pour chaque tour de la boucle `for`, la méthode `self.single_game()` est appelée pour déterminer le gagnant et incrémenter le compte correspondant dans `win_count`. On affiche ensuite ce nombre de parties gagnées et le taux de victoire en pourcentage associés à chaque joueur.

2.4 Temps moyen que met chaque joueur pour jouer un coup

On crée le dictionnaire `play_move_time` dans le fichier `strategy.py` avec deux clés : 1 et 2 associées à des listes vides. Lorsque la stratégie est exécutée dans la fonction `start()`, on enregistre le temps écoulé dans la liste correspondante au sein du dictionnaire en utilisant la clé associée au joueur qui démarre. Puis, on importe ce dictionnaire dans le fichier `tournament.py`. Enfin, on affiche dans la fonction `championship()` le temps moyen que met chaque joueur pour jouer un coup en sommant tous les temps enregistrés pour chaque joueur à chaque coup dans la liste et en le divisant par la taille de cette dernière. Le résultat est affiché en millisecondes pour gagner en précision et en information sur les comparaisons entre nos différentes approches.

3 RÉSULTATS

Voici les notations pour le tableau suivant :

- OPTI = `minimax_bestChoice` (notre stratégie)
- AB = `minimaxAB_strategy`
- M = `minimax_strategy`
- R = `random_strategy`

Pour les parties où la `random_strategy` est utilisé, nous avons fait nos tests sur un échantillon de 30 parties ($N\_GAMES = 30$). Puis pour toutes les autres, les résultats obtenus sont le résultat de 10 parties jouées ($N\_GAMES = 10$).

Strat 1	Strat 2	N	Gagnant	% victoire	Temps moyen
AB	OPTI	6	2	60%	1 : 40m 28s / 2 : 23m 44s
R	OPTI	6	2	100%	1 : 1.7ms / 29m : 30s
R	OPTI	5	2	100%	1 : 0.5ms / 15m : 18s

Table 1. Statistiques des confrontations stratégiques lorsque $depth = 4$

Strat 1	Strat 2	N	Gagnant	% victoire	Temps moyen
AB	OPTI	7	2	80%	1 : 23m 7s / 2 : 19m 40s
R	OPTI	6	2	100%	1 : 0.26ms / 4m : 9s
M	OPTI	4	2	100%	1 : 1m 6s / 2 : 7s
M	AB	4	2	60%	1 : 1m 7s / 2 : 10s
AB	OPTI	4	2	60%	1 : 10s / 2 : 6s

Table 2. Statistiques des confrontations stratégiques lorsque $depth = 3$

Pour conclure, nous pouvons dire que nos optimisations sur `minimax_bestChoice` sont performantes, notamment en terme de temps d'exécution. Le taux de victoire entre la stratégie `minimaxAB_strategy` et `minimax_bestChoice` est en faveur de notre stratégie mais n'est pas significatif lorsque le facteur N qui définit la taille du plateau est petit. En revanche, pour un plateau dont la taille est plus importante, on constate un taux de victoire nettement plus important.
Pour se rendre compte de la performance de l'algorithme, nous pouvons essayer de jouer contre ce dernier puis le comparer à une autre stratégie. Si vous le faites, vous verrez que la décision de celui-ci est assez précise. Jouer une partie dont la dimension est de 7×7 avec une profondeur maximale de 3 est aisément faisable dans un laps de temps assez court (ou en 6×6 pour $depth = 4$ pour un temps d'exécution environ similaire mais une précision plus importante).

Malheureusement, une des conséquences de la méthode utilisée est que lorsque l'algorithme estime qu'il est sur le point de perdre, il choisit une stratégie aléatoire parmi toutes les options disponibles. Cela signifie que lorsque l'algorithme réalise qu'il a perdu, il ne joue plus "intelligemment".