Rapport INF8175 : Skypiea

Equipe Challonge : **Orque des Tranchées** *Stephen Cohen - 2412336; Alicia Shao - 2409849*

Introduction

Dans ce rapport, nous allons présenter notre cheminement vers notre agent final dans le cadre du cours d'Intelligence Artificielle INF8175 centré cette année autour du jeu de société **Divercité**.

Cheminement

Notre réflexion a d'abord débuté par l'implémentation de l'algorithme de Minimax, puis d'un Minimax alpha-bêta prunning et enfin de Monte Carlo Tree Search (MCTS). Dans la suite, le code du *minimax* et de l'*alpha-bêta prunning* sont disponibles en annexe.

- **Minimax :** Cet implémentation est disponible sous les noms d'agents : `Water_seven`, `little_garden` sur Abyss. Dans les deux cas, *la profondeur de recherche est fixée à 3* pour **respecter la contrainte de temps imposée**. Avec ce paramètre, il reste environ 100 s à la fin de la partie. La différence réside dans l'heuristique choisie :
    - La première, celle de `little_garden` est la suivante : `player_score - opponent_score`. On cherche à maximiser l'écart de points en notre faveur. Cependant on se rend vite compte que cette heuristique n'est pas assez précise; elle peut favoriser des divercités pour l'adversaire.
    - La seconde, celle de `Water_seven`, tient compte de nos observations : nous avons remarqué que placer des cités en fin de partie est défavorable alors que les ressources permettent de finir des divercités ou de bloquer celles de l'adversaire. Ainsi nous avons cherché à pénaliser le choix de jouer une cité au fil de la partie, et de la favoriser celle de ressources. Nous aboutissons à l'heuristique suivante : `player_score - opponent_score + (1 - 2 * state.step/40) * nb_cite + (1 + 4 * state.step/40) * nb_ressource`. **En plus de l'écart entre les points, nous rajoutons un paramètre qui tient compte du roaster de coups disponibles (ressources ou cités).**
- **Alpha-Beta prunning :** Cette implémentation est disponible dans les agents : `Enies_Lobby`, `skypiea_vX`. L'avantage de l'alpha-Beta prunning réside dans l'élagage des états. Ce gain de temps est réinvesti au profit d'une recherche plus profonde. **Attention :** Nous avons remarqué qu'une grande profondeur avec une mauvaise heuristique propageait de mauvais résultats ce qui entrainent une dégradation significative des résultats de l'agent (notamment avec `skypiea` avec l'heuristique classique)
    - **La profondeur est dynamique en fonction de l'avancée de la partie**. En effet, une *recherche profonde en début de partie représente un investissement de temps peu rentable pour un coup peu décisif*. C'est pour cela que le **premier coup est joué de manière aléatoire** ce qui permet de gagner 100 à 200s de jeu :

```
# Pré-choisir une action si on joue en premier
    if current_state.get_step() < 2:
        possible_actions = current_state.get_possible_light_actions()
        return random.choice(list(possible_actions))
```

Puis, la profondeur est gérée comme suit afin de conserver environ 60 s en fin de partie :

```
if nb_pieces_1 + nb_pieces_2 >= 35:
            depth = 4
        elif nb_pieces_1 + nb_pieces_2 >= 12:
            depth = 5
        else:
            depth = 6

        _, best_action = alpha_beta_minimax(current_state, depth, float('-
inf'), float('inf'), True)
```

Cette étape abouti à `skypiea_v2`, l'agent le plus performant de la génération alpha-beta pruning La profondeur est gérée de la manière suivante :

```
if nb_pieces_1 + nb_pieces_2 >= 12:
                depth = 4
            else:
                depth = 6
```

et l'heuristique est celle-ci : `player_score - opponent_score + (1 - 2 * state.step/40) * nb_cite + (1 + 4 * state.step/40) * nb_ressource`

- **MCTS :** Les resultats donnés par notre agent basé uniquement sur un MCTS ne sont pas assez performant pour mériter une présentation complète du code, il ne bat aucun des agents présentés plus haut. Cependant, nous souhaitions développer un agent combinant alpha-beta pruning et MCTS de la manière suivante: :
    - *Sur les 30% du début de jeu, jouer avec un MCTS pas forcément très efficient mais très rapide* (avec un compromis tout de même pour ne pas subir une avance trop conséquente de l'adversaire)
    - Puis un *Alpha-prunning de profondeur progressive de type 5 / 7*.

      Cependant, nous n'avons pas pu finaliser l'ajustement des paramètres avant le début du tournoi. Nos agents étaient soit **trop lents** (mais assez compétitifs, sans battre nos agents précédents) ou **trop peu efficaces** (mais rapides). Ce qui nous faisait perdre contre nos propres agents sans MCTS. Le code de notre agent qui réalise cela se trouve en annexe sous le nom d'`Impel Down`.

A ce stage, nous avons choisi parmi nos agents de perfectionner `skypiea_v2` (donc un algorithme de type minimax avec alpha-prunning pur)

Notre agent

Principe

Choix du type d'algorithme

Heuristique

Implémentation

Performances

Pistes d'améliorations

Conclusion

Annexes

Code du minimax

Voici le code utilisé pour le minimax pur :

```
def minimax(state: GameState, depth: int, maximizing_player: bool) -> float:
            if depth == 0 or state.is_done():
                return self.evaluate_state(state), None

            if maximizing_player:
                max_eval = float('-inf')
                for action in state.get_possible_light_actions():
                    next_state = state.apply_action(action)
                    eval, _ = minimax(next_state, depth - 1, False)
                    if eval > max_eval :
                        max_eval = eval
                        best_action = action
                return max_eval, best_action
            else:
                min_eval = float('inf')
                for action in state.get_possible_light_actions():
                    next_state = state.apply_action(action)
                    eval, _  = minimax(next_state, depth - 1, True)
                    if eval < min_eval:
                        min_eval = eval
                        best_action = action
                return min_eval, best_action

        if current_state.get_step() < 2:
            possible_actions = current_state.get_possible_light_actions()
            return random.choice(list(possible_actions))
        else :
            # Ajustement de la profondeur en fonction du nombre de pièces
restantes
            players = current_state.players
            players_id = [p.get_id() for p in players]
            dic_player_pieces = current_state.players_pieces_left
            dic_pieces_1 = dic_player_pieces[players_id[0]]
            dic_pieces_2 = dic_player_pieces[players_id[1]]
            pieces = ['RC', 'RR', 'GC', 'GR', 'BC', 'BR', 'YC', 'YR']
            nb_pieces_1, nb_pieces_2 = sum(dic_pieces_1[p] for p in pieces),
sum(dic_pieces_2[p] for p in pieces)

            # Modifier la profondeur en fonction du nombre de pièces restantes
            '''
            if nb_pieces_1 + nb_pieces_2 >= 22:
                depth = 3
            elif nb_pieces_1 + nb_pieces_2 >= 12:
                depth = 4
```

```
            else:
                depth = 5
            best_action = None
            # best_value = float('-inf')
            '''
            _ , best_action = minimax(current_state, 3, True)  ###################
Ici pour changer la profondeur et mettre à True car on veut maximiser

            return best_action
```

Code de l'alpha-beta prunning

```
def compute_action(self, current_state: GameState, **kwargs) -> Action:
        """
        Use the minimax algorithm with alpha-beta pruning to choose the best
action.
        """
        def alpha_beta_minimax(state: GameState, depth: int, alpha: float, beta:
float, maximizing_player: bool) -> float:
            if depth == 0 or state.is_done():
                return self.evaluate_state(state), None

            if maximizing_player:
                max_eval = float('-inf')
                best_action = None
                actions = state.get_possible_light_actions()

                # Ne trie que si le nombre d'actions est assez grand
                if len(actions) > 5:
                    actions = sorted(actions, key=lambda a:
self.evaluate_state(state.apply_action(a)), reverse=True)

                for action in actions:
                    next_state = state.apply_action(action)
                    eval, _ = alpha_beta_minimax(next_state, depth - 1, alpha,
beta, False)
                    if eval > max_eval:
                        max_eval = eval
                        best_action = action
                    alpha = max(alpha, eval)
                    if beta <= alpha:
                        break
                return max_eval, best_action
            else:
                min_eval = float('inf')
                best_action = None
                actions = state.get_possible_light_actions()

                if len(actions) > 5:
                    actions = sorted(actions, key=lambda a:
self.evaluate_state(state.apply_action(a)))
```

```
                for action in actions:
                    next_state = state.apply_action(action)
                    eval, _ = alpha_beta_minimax(next_state, depth - 1, alpha,
beta, True)
                    if eval < min_eval:
                        min_eval = eval
                        best_action = action
                    beta = min(beta, eval)
                    if beta <= alpha:
                        break
                return min_eval, best_action

        # Pré-choisir une action si on joue en premier
        if current_state.get_step() < 2:
            possible_actions = current_state.get_possible_light_actions()
            return random.choice(list(possible_actions))

        else:
        # Ajustement de la profondeur en fonction du nombre de pièces restantes
            players = current_state.players
            players_id = [p.get_id() for p in players]
            dic_player_pieces = current_state.players_pieces_left
            dic_pieces_1 = dic_player_pieces[players_id[0]]
            dic_pieces_2 = dic_player_pieces[players_id[1]]
            pieces = ['RC', 'RR', 'GC', 'GR', 'BC', 'BR', 'YC', 'YR']
            nb_pieces_1, nb_pieces_2 = sum(dic_pieces_1[p] for p in pieces),
sum(dic_pieces_2[p] for p in pieces)

            # Modifier la profondeur en fonction du nombre de pièces restantes

            # Fonctionne plus rapidement
            '''
            if nb_pieces_1 + nb_pieces_2 >= 35:
                depth = 3
            elif nb_pieces_1 + nb_pieces_2 >= 12:
                depth = 4
            else:
                depth = 6
            '''

            if nb_pieces_1 + nb_pieces_2 >= 35:
                depth = 2
            elif nb_pieces_1 + nb_pieces_2 >= 12:
                depth = 5
            else:
                depth = 7

            _, best_action = alpha_beta_minimax(current_state, depth, float('-
inf'), float('inf'), True)
            return best_action
```

Code Impel Down