Rapport INF8175 : Skypiea

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

Table des matières

1 Introduction

Dans ce rapport, nous allons présenter le cheminement aboutissant à notre agent final pour le projet du cours Intelligence artificielle : méthodes et algorithmes, centré cette année sur le jeu de société Divercité.

2 Cheminement

Dans un premier temps, un algorithme *minimax* a été mis en place, un *alpha-bêta pruning* y a été ensuite ajouté. Enfin, la possibilité d'un Monte-Carlo Tree Search (MCTS) a été également explorée.

Les codes Python du *minimax* et de l'*alpha-bêta pruning* sont disponibles en annexe.

— ***Minimax* :** Cette implémentation est disponible sous les noms d'agents : `Water_seven` et `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 entre les deux agents réside dans l'heuristique choisie :

  — L'heuristique de `little_garden` est la suivante : `player_score - opponent_score`. On cherche à maximiser l'écart de points en notre faveur. Cependant, cette heuristique n'est pas assez précise, elle peut favoriser des divercités pour l'adversaire.

  — La seconde, celle de `Water_seven`, tient compte des observations faites lors de nos sessions de jeu. 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, on pénalise le choix de jouer une cité au fil de la partie et on favorise le choix 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 donc un paramètre qui tient compte de l'ensemble des coups disponibles (ressources ou cités).

— ***Alpha-bêta pruning* :** Cette implémentation est disponible dans les agents : `Enies_Lobby` et `skypiea_vX`. L'avantage de l'*alpha-bêta pruning* réside dans le gain de temps obtenu grâce à l'élagage. Ce temps supplémentaire est réinvesti au profit d'une recherche plus profonde. **Attention :** On remarque qu'une grande profondeur couplée à une mauvaise heuristique propage de mauvais résultats, ce qui entraine 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.* Ainsi, le **premier coup est joué de manière aléatoire parmi les coups jouant des cités**, ce qui permet de gagner 100 à 200 s de jeu :

Listing 1 – Code pour le premier coup

```
# Pr  -choisir une action si on joue en premier
if current_state.get_step() < 2:
    possible_actions = current_state.get_possible_light_actions()
    city_actions = [action for action in possible_actions
                    if action.data["piece"] in ['RC', 'GC', 'BC', 'YC']]
    return random.choice(city_actions)
```

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

Listing 2 – Gestion de la profondeur

```
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 d'ajustement de la profondeur abouti à `skypiea_v2`, l'agent le plus performant de la génération *alpha-bêta pruning*. La profondeur est gérée finalement de la manière suivante :

Listing 3 – Gestion finale de la profondeur

```
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 résultats donnés par notre agent basé uniquement sur un MCTS ne sont pas assez satisfaisants pour mériter une présentation complète du code, il ne bat aucun des agents présentés plus haut. Cependant, nous avons développé un agent combinant *alpha-bêta pruning* et MCTS de la manière suivante :

  — *Sur les 30% du début de jeu, jouer avec un MCTS moins optimal qu'un alpha-bêta pruning mais très rapide.*

  — Puis, un *alpha-bêta pruning de profondeur progressive de type 5 / 7.*

L'idée première était d'obtenir rapidement des premiers coups, cohérents, voirs bons pour libérer du temps afin de réaliser une recherche plus profonde dans un deuxième temps. Dans le cas où le MCTS ne propose que des coups incohérents et mauvais, l'effet pourrait être compensé par la deuxième phase, qui engendrait déjà à ce stade des résultats probants.

L'ajustement final des paramètres n'a pas pu être réalisé 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). Le code de notre agent hybride est disponible en annexe sous le nom d'`Impel Down`.

Nous avons donc choisi de perfectionner pour ce projet `skypiea_v2`, un agent *minimax alpha-bêta pruning* pur.

3 Notre agent : Skypiea_v5

3.1 Heuristique

La base de l'heuristique Skypiea_v5 est celle décrite dans la partie *Alpha-bêta Pruning*. Après avoir réalisé des tests pour affiner les coefficients de pénalisation, nous obtenons l'heuristique suivante : `player_score - opponent_score + (1 - 24 * state.step / 40) * nb_cite + (1 + 27 * state.step / 40) * nb_ressource`

3.2 Implémentation

Le code de notre agent est disponible en annexe.

4 Performances

4.1 Performances de jeu

Ces statistiques ont été obtenues à partir d'Abyss. On y présente tous les agents qui ont réalisé au moins un match.

Table 1: Performances des agents

Agents	**Notes**	**Elo**	**Total matches**	**W/L**	**Divercités/ Game**	**Marqués/ Concédés**	**Points marqués/- Game**	**Points concédés/- Game**
Skypiea v5	Alpha Beta Pruning	1282	13	3.33	2.31	1.29	19.92	15.46
Impel Down 25000	MCTS puis AB	937	9	2.00	2.44	1.36	22.22	16.33
Impel Down 15000	MCTS puis AB	1006	1	-	5.00	1.81	29.00	16.00
Water Seven	-	971	1721	1.10	1.15	1.05	17.17	16.41
Skypiea v2.2	Alpha Beta Pruning	1165	25	2.13	1.04	1.16	17.88	15.44
Skypiea v2	Alpha Beta Pruning	1275	489	2.52	1.30	1.22	18.58	15.28
Little Garden	-	713	707	0.91	0.69	1.03	16.78	16.22
Skypia	-	960	5	0.67	0.00	1.03	18.20	17.60

Remarques : Le nombre de matches joués par un agent est assez inhomogène, les conclusions suivantes en prennent compte et sont à nuancer.

Le tableau met en évidence la compétitivité des différents agents dans l'environnement Abyss. Voici un commentaire détaillé des résultats :

- **Agent Skypiea_v5 (*Alpha-bêta pruning*)**
  - Performance globale : Avec un Elo de 1282, l'agent Skypiea_v5 domine la majorité des parties, affichant un ratio victoires/défaites de 3,33 (pour seulement 13 matches cependant)
  - Divercités : Il obtient en moyenne 2,3 divercités par partie, témoignant d'une capacité à exploiter les ressources efficacement, bien que cela soit perfectible.
  - Ratio points marqués/concédés : Le ratio de 1,29 montre une capacité à marquer plus de points que l'adversaire, reflet d'une bonne gestion des ressources et de l'heuristique employée.
- **Agent Impel Down (MCTS puis *Alpha-bêta pruning*)**
  - Deux versions : Les versions avec 15 000 et 25 000 simulations aléatoires affichent des performances variables. La version 25 000 obtient un Elo de 937, mais l'agent peine à être décisif (ratio W/L de 2) et est davantage soumis au risque de *time out*. (Il est actuellement inactif pour cette même raison).

— Divercités : Le score moyen de divercités par partie (2,44) est légèrement meilleur que Skypiea_v5, suggérant que l'exploration initiale en MCTS réussit à choisir de meilleurs coups en début de partie que les agents de type Skypiea.

— **Agents intermédiaires (Skypiea_v2)**

— Ces deux agents de type *Alpha-bêta pruning* purs montrent des résultats stables sur un grand nombre de matches, obtenant respectivement 1275 et 1108 d'Elo.

— Ces résultats confirment qu'un agent basé *alpha-bêta pruning* représente une base solide pour obtenir des résultats probants rapidement et facilement. Cela conforte le choix d'avoir poursuivi l'affinage des agents de type Skypiea.

4.2 Nombre d'états visités et profondeur dynamique

Ci-dessous, un graphique du nombres d'états évalués par Skypiea_v5 en fonction de l'avancée de la partie.

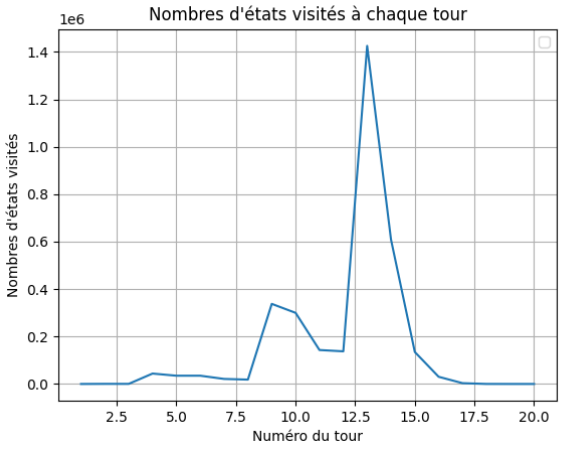

FIGURE 1 – États visités en fonction de l'avancée de la partie

Sur le graphique des états visités, nous observons bien les différentes profondeurs de l'arbre du *minimax*. De plus, notre stratégie qui vise à avoir un milieu de partie très profond est rendu possible grâce à un début de partie très peu profond. Voici le nombre d'états visités au fil de la partie : `[0, 452, 402, 43888, 34809, 34825, 21231, 18401, 337614, 299769, 143075, 137567, 1425242, 608967, 134754, 30259, 3387, 169, 25, 3]`

— Le premier coup est aléatoire donc 0

— La phase de recherche en profondeur 2 : `[452, 402]`

— La phase de recherche en profondeur 4 : `[43888, 34809, 34825, 21231, 18401]`

— La phase de recherche en profondeur 5 : `[337614, 299769, 143075, 137567]`

— La phase de recherche en profondeur 7 : `[1425242, 608967, 134754, 30259, 3387, 169, 25, 3]`

5 Limites

Outre des *time out* dès le coup 1 que nous ne comprenons pas, voici les pistes d'amélioration que nous avons identifiées :

— **L'heuristique :** Notre heuristique ne tient pas compte de **la présence de divercités** ou d'**opportunités de divercités** pour l'agent ou l'agent adverse. En effet, l'agent ne sait à aucun moment combien de ses pièces ou des pièces adverses sont encore capables de réaliser une divercité, ni si elles en sont éloignées. C'est pourquoi, nous pouvons juger une situation plus favorable alors qu'elle nous empêche de réaliser une divercité ou rapproche l'adversaire d'en faire une. C'est d'ailleurs ce que l'on peut observer dans l'étude des performances, nos agents réalisent en moyenne moins de 1,5 divercité par partie.

— **La connaissance des pièces adverses :** Dans Divercité, les joueurs ont connaissance des pièces adverses. Par exemple, savoir que l'adversaire n'a plus de ressources rouges permet de savoir qu'il ne pourra plus faire de divercité. S'il n'a plus de cité, les places de cité disponibles nous reviennent. Il n'y aurait donc plus besoin de les pénaliser, il faudrait même construire des divercités autour des emplacements de cités vides.

— **Le temps de calcul :** L'agent présenté utilise presque l'intégralité du temps disponible (il reste 35 secondes à la fin), sans pourtant présenter des résultats excellents. Cela suggère qu'il est possible d'élaguer davantage d'états, toujours dans l'optique d'augmenter la profondeur de recherche.

6 Avantages

— **L'implémentation :** L'algorithme *minimax* avec *alpha-bêta pruning* n'est pas difficile à implémenter, ce qui facilite le développement et permet d'avoir une phase de tests beaucoup plus longue permettant d'ajuster l'heuristique par rapport à un algorithme plus complexe à mettre en place.

— **Les performances :** Le ratio entre la performance et la difficulté d'implémentation est excellent pour Skypiea_v5 et les autres agents de sa famille. A titre de comparaison, les agents hybrides avec un MCTS sont prometteurs mais nécessitent l'ajustement complexe de plusieurs autres paramètres (nombre de simulations totales, choix entre simulations aléatoires ou guidées par une heuristique, choix du moment de passage à l'*alpha-bêta pruning*) sans garantie d'amélioration substantive.

7 Pistes d'améliorations

— **L'heuristique :** L'heuristique actuelle peut-être enrichie avec des facteurs comme le contrôle des zones stratégiques, la proximité de réalisation des divercités, et les blocages potentiels pour l'adversaire. Cela permet une prise de décisions plus stratégiques en fonction du contexte du jeu, surtout lors de phases plus critiques.

— **La réduction du temps de calcul :**

  — On pourrait **stocker les évaluations des états déjà visités** pour éviter de recalculer des scénarios redondants. L'utilisation d'une table de transposition permet d'accélérer les recherches, notamment dans les configurations de jeu symétriques. Pour se faire, on peut utiliser des hashcodes d'états pour les identifier efficacement ; tout en tenant compte des symétries du plateau pour réduire de manière significative les répétitions de calcul.

  — Une stratégie de ***Late Move Reduction*** peut également être mise en place. Elle consiste à réduire la profondeur de recherche sur les coups explorés tardivement dans une branche, en supposant que les premiers coups sont généralement les meilleurs.

  — Une autre piste d'amélioration est de **gérer le temps de manière adaptive en fonction du temps restant**. Il s'agit de limiter la profondeur ou d'interrompre des branches si le temps alloué pour un coup est proche de l'expiration.

— **Tenir compte de l'historique de nos parties :** Via une exploration bayésienne des actions, on pourrait ajuster la probabilité de sélection des actions en fonction de leur performance dans des simulations précédentes.

8 Conclusion

Ainsi, notre équipe a développé un agent pour Divercité de type *alpha-bêta pruning* plutôt performant, fruit d'un processus itératif formé par trois générations d'agents.

L'heuristique employée met l'accent non seulement sur le score de la partie, mais prend également en compte des stratégies développées lors des sessions de jeu réelles en favorisant les coups impliquant des cités et en pénalisant les coups impliquant des ressources en *early game*.

L'heuristique et le type d'algorithme choisis ayant obtenu des résultats probants très tôt, le principal enjeu et défi de ce projet aura donc été d'augmenter la profondeur de recherche dans l'algorithme afin d'améliorer ces résultats, tout en tenant compte de la contrainte de temps.

Plusieurs approches ont été employées pour y répondre, dont une profondeur dynamique en fonction de l'avancée de la partie, pour favoriser la recherche d'états favorables en milieu de jeu, identifié comme le moment décisif d'une partie.

L'idée d'un agent hybride MCTS-*alpha-bêta pruning*, qui apporte une réponse différente à ce problème, n'a pas été retenue. Elle demeure cependant une approche prometteuse mais plus complexe, nécessitant davantage d'investissement pour l'ajuster, l'affiner et l'améliorer.

Finalement, à travers ce projet, nous avons mis en pratique plusieurs algorithmes de recherche adversariale dans un cadre applicatif stimulant et ludique. Cette expérience mettant à profit nos capacités analytiques a nécessité la mise en place d'une réelle démarche de recherche collaborative, enrichissant ainsi notre bagage scientifique et nos compétences interpersonnelles.

A Annexes

A.1 Code de l'agent final : Skypiea_v5

Listing 4 – Code de l'agent Skypiea_v5

```
import random
from functools import lru_cache
from seahorse.game.action import Action
from seahorse.game.game_state import GameState
from player_divercite import PlayerDivercite

class MyPlayer(PlayerDivercite):
    def __init__(self, piece_type: str, name: str = "AlphaBetaOptimized"):
        super().__init__(piece_type, name)

    def compute_action(self, current_state: GameState, **kwargs) -> Action:
        if current_state.get_step() < 2:
            possible_actions = current_state.get_possible_light_actions()
            city_actions = [action for action in possible_actions
                            if action.data["piece"] in ['RC', 'GC', 'BC', 'YC']]
            return random.choice(city_actions)

        # Calcul de la profondeur en fonction des pi ces restantes
        depth = self.calculate_depth(current_state)
        _, best_action = self.alpha_beta_minimax(current_state, depth,
                                                 float('-inf'), float('inf'), True)
        return best_action

    def calculate_depth(self, state: GameState) -> int:
        players = state.players
        dic_player_pieces = state.players_pieces_left
        pieces = ['RC', 'RR', 'GC', 'GR', 'BC', 'BR', 'YC', 'YR']
        total_pieces = sum(dic_player_pieces[p.get_id()][p_type]
                           for p in players for p_type in pieces)

        if total_pieces >= 35:
            return 2
        elif total_pieces >= 24:
            return 4
        elif total_pieces >= 16:
            return 5
        else:
            return 7

    def alpha_beta_minimax(self, state: GameState, depth: int,
                           alpha: float, beta: float, maximizing_player: bool):
        if depth == 0 or state.is_done():
            return self.evaluate_state_cached(state), None

        actions = state.get_possible_light_actions()
        if len(actions) > 5:
            actions = sorted(actions,
                             key=lambda a: self.evaluate_state_cached(state.
                                 apply_action(a)),
                             reverse=maximizing_player)

        best_action = None
        if maximizing_player:
            max_eval = float('-inf')
            for action in actions:
                next_state = state.apply_action(action)
                eval, _ = self.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')
            for action in actions:
                next_state = state.apply_action(action)
                eval, _ = self.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

    @lru_cache(maxsize=5000)
    def evaluate_state_cached(self, state: GameState) -> float:
        return self.evaluate_state(state)

    def evaluate_state(self, state: GameState) -> float:
        player_id = self.get_id()
        player_score = state.scores[player_id]
        opponent_score = sum(score for pid, score in state.scores.items()
                             if pid != player_id)

        dic_pieces = state.players_pieces_left[player_id]
        nb_cite = sum(dic_pieces[c] for c in ['RC', 'GC', 'BC', 'YC'])
        nb_ressource = sum(dic_pieces[r] for r in ['RR', 'GR', 'BR', 'YR'])

        return (
            player_score - opponent_score
            + (1 - 24 * state.step / 40) * nb_cite
            + (1 + 24 * state.step / 40) * nb_ressource
        )
```

A.2 Code du minimax

Listing 5 – Code de l'implémentation Minimax

```
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
```

```
23
24 if current_state.get_step() < 2:
25     possible_actions = current_state.get_possible_light_actions()
26     return random.choice(list(possible_actions))
27 else:
28     # Ajustement de la profondeur en fonction du nombre de pi ces restantes
29     players = current_state.players
30     players_id = [p.get_id() for p in players]
31     dic_player_pieces = current_state.players_pieces_left
32     dic_pieces_1 = dic_player_pieces[players_id[0]]
33     dic_pieces_2 = dic_player_pieces[players_id[1]]
34     pieces = ['RC', 'RR', 'GC', 'GR', 'BC', 'BR', 'YC', 'YR']
35     nb_pieces_1 = sum(dic_pieces_1[p] for p in pieces)
36     nb_pieces_2 = sum(dic_pieces_2[p] for p in pieces)
37
38     _ , best_action = minimax(current_state, 3, True)
39
40     return best_action
```

A.3 Code de l'alpha-beta pruning

Listing 6 – Code de l'implémentation Alpha-Beta Pruning

```
1  def compute_action(self, current_state: GameState, **kwargs) -> Action:
2      def alpha_beta_minimax(state: GameState, depth: int,
3                             alpha: float, beta: float,
4                             maximizing_player: bool) -> float:
5          if depth == 0 or state.is_done():
6              return self.evaluate_state(state), None
7
8          if maximizing_player:
9              max_eval = float('-inf')
10             best_action = None
11             actions = state.get_possible_light_actions()
12
13             if len(actions) > 5:
14                 actions = sorted(actions,
15                                  key=lambda a: self.evaluate_state(state.apply_action(a
                                        )),
16                                  reverse=True)
17
18             for action in actions:
19                 next_state = state.apply_action(action)
20                 eval, _ = alpha_beta_minimax(next_state, depth - 1,
21                                              alpha, beta, False)
22                 if eval > max_eval:
23                     max_eval = eval
24                     best_action = action
25                 alpha = max(alpha, eval)
26                 if beta <= alpha:
27                     break
28             return max_eval, best_action
29         else:
30             min_eval = float('inf')
31             best_action = None
32             actions = state.get_possible_light_actions()
33
34             if len(actions) > 5:
35                 actions = sorted(actions,
36                                  key=lambda a: self.evaluate_state(state.apply_action(a
                                        )))
37
38             for action in actions:
39                 next_state = state.apply_action(action)
40                 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

    if current_state.get_step() < 2:
        possible_actions = current_state.get_possible_light_actions()
        return random.choice(list(possible_actions))
    else:
        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 = sum(dic_pieces_1[p] for p in pieces)
        nb_pieces_2 = sum(dic_pieces_2[p] for p in pieces)

        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
```

A.4 Code Impel Down

Listing 7 – Code de l'agent Impel Down

```
import random
from seahorse.game.action import Action
from seahorse.game.game_state import GameState
from player_divercite import PlayerDivercite
import math

class TreeNode:
    def __init__(self, state : GameState, max_root_children = -1, parent=None):
        self.state = state
        self.parent = parent
        self.max_root_children = max_root_children
        self.children = {}
        self.visits = 0
        self.value = 0.0

    def is_fully_expanded(self):
        """ Check if all possible actions have been expanded. """
        if (self.parent == None and self.max_root_children > -1) :
            return len(self.children) == self.max_root_children
        else:
            return len(self.children) == len(self.state.get_possible_light_actions())

    def uct_value(self, exploration_constant=math.sqrt(2)):
        """ Calculate the UCT value for this node. """
        if self.visits == 0:
            return float('inf')  # Ensure unvisited nodes are prioritized
        exploitation = self.value / self.visits
```

```
        exploration = exploration_constant * math.sqrt(math.log(self.parent.visits) / self.visits)
        return exploitation + exploration

    def best_child(self, exploration_constant=math.sqrt(2)):
        """ Select the child with the highest UCT value. """
        return max(self.children.values(), key=lambda child: child.uct_value(exploration_constant))

    def expand(self):
        """ Expand by adding a child for an untried action. """
        actions = self.state.get_possible_light_actions()
        untried_actions = [a for a in actions if a not in self.children]
        action = random.choice(untried_actions)
        next_state = self.state.apply_action(action)
        child_node = TreeNode(next_state, parent=self)
        self.children[action] = child_node
        return child_node

    def update(self, outcome):
        """ Update node statistics on backpropagation. """
        self.visits += 1
        self.value += outcome

    def select(self):
        """ Traverse the tree using UCT until reaching a leaf node. """
        node = self
        while not node.isLeaf() and node.is_fully_expanded():
            node = node.best_child()
        return node

    def isLeaf(self):
        """ Check if this node is a leaf (has no children). """
        return len(self.children) == 0

class MyPlayer(PlayerDivercite):
    """
    Player class for Divercite game that uses the Minimax algorithm with alpha-beta pruning and MCTS for the first 10 moves.
    """

    def __init__(self, piece_type: str, name: str = "AlphaBetaPlayer"):
        super().__init__(piece_type, name)

    def mcts(self, state: GameState, simulations: int = 1000) -> Action:      ### Attention que 1000 simulations peut- tre  pas assez
        """ Perform MCTS to determine the best action. """
        action_counts = {action: 0 for action in state.get_possible_light_actions()}
        action_values = {action: 0 for action in state.get_possible_light_actions()}

        for _ in range(simulations):
            # Convert possible actions to a list
            possible_actions_list = list(action_counts.keys())
            action = random.choice(possible_actions_list)
            next_state = state.apply_action(action)

            # Simulate the game to completion from the next state
            while not next_state.is_done():
                possible_actions = next_state.get_possible_light_actions()
                # Convert possible actions to a list
                possible_actions_list = list(possible_actions)
                random_action = random.choice(possible_actions_list)
                next_state = next_state.apply_action(random_action)

            # Use the evaluation function to determine the outcome of the simulation
            outcome = self.evaluate_state(next_state)
```

```
            action_counts[action] += 1
            action_values[action] += outcome

        # Calculate average values and choose the best action
        best_action = max(action_values, key=lambda a: action_values[a] /
            action_counts[a])
        return best_action

    def simpleSimulation(self, node):
        current_state = node.state
        while not current_state.is_done():
            possible_actions = list(current_state.get_possible_light_actions())
            action = random.choice(possible_actions)
            current_state = current_state.apply_action(action)
        return self.evaluate_state(current_state)

    def heuristicsSimulation(self, node):
        current_state = node.state
        while not current_state.is_done():
            possible_actions = list(current_state.get_possible_light_actions())

            # Evaluate each possible next state
            action_scores = []
            for action in possible_actions:
                next_state = current_state.apply_action(action)
                score = self.evaluate_state(next_state)
                action_scores.append((action, score))

            # Calculate the total score for normalization
            total_score = sum(score for _, score in action_scores)

            if total_score > 0:
            # Weighted random choice based on normalized probabilities
                probabilities = [score / total_score for _, score in action_scores]
                action = random.choices([a for a, _ in action_scores], weights=
                    probabilities, k=1)[0]
            else:
                # Fallback to uniform random choice if all scores are zero
                action = random.choice(possible_actions)

            # Apply the chosen action
            current_state = current_state.apply_action(action)

        return self.evaluate_state(current_state)

    def mcts_taylorsVersion(self, state : GameState, simple, max_root_children = -1,
        simulation = 1000):
        treePaine = TreeNode(state, max_root_children)
        if treePaine.parent == None and max_root_children > 0:
            actions = state.get_possible_light_actions()
            actions = sorted(actions, key=lambda a: self.evaluate_state(state.
                apply_action(a)), reverse=True)[:max_root_children]
            treePaine.children = {action: TreeNode(state.apply_action(action), parent
                =treePaine) for action in actions}
        for _ in range(simulation):
            print(f"\rMCTS Iteration: {_ + 1}/{simulation}, root children: {len(
                treePaine.children)}", end='', flush=True)
            if _ == simulation - 1:
                print("\n")
            #Select
            node = treePaine.select()

            # 2. Expansion
            if not node.state.is_done() and not node.is_fully_expanded():
                node = node.expand()

```

```
            outcome = 0
            if simple:
                outcome = self.simpleSimulation(node)
            else:
                outcome = self.heuristicsSimulation(node)

            # Backpropagate
            while node:
                node.update(outcome)
                node = node.parent

        # Choose the action leading to the best child
        best_action = max(treePaine.children.items(), key=lambda item: item[1].visits)[0]
        return best_action

    def alpha_beta_minimax(self, 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, _ = self.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  # Coupure
            return max_eval, best_action  # Return value and 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, _ = self.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  # Coupure
            return min_eval, best_action  # Return value and best action

    def greedy(self, state):
        possible_actions = state.generate_possible_heavy_actions()
        best_action = next(possible_actions)
```

```
        best_score = best_action.get_next_game_state().scores[self.get_id()]
        for action in possible_actions:
            state = action.get_next_game_state()
            score = state.scores[self.get_id()]
            if score > best_score:
                best_action = action
        return best_action

    def compute_action(self, current_state: GameState, **kwargs) -> Action:
        """
        Compute action using MCTS for the first 10 moves, then alpha-beta pruning.
        """

        if current_state.get_step() < 2:
            return self.greedy(current_state)
        # Utiliser MCTS pour les 10 premiers coups
        if current_state.get_step() < 10:
### Attention j'ai modifi  ta version ici
            #return self.mcts_taylorsVersion(current_state, True, 10, 20000)
            return self.mcts_taylorsVersion(current_state, True, 10, 15000)
        # Pour les coups suivants, utiliser alpha-beta
        else:
            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)

            # Ajuster la profondeur en fonction du nombre de pi ces restantes
            if nb_pieces_1 + nb_pieces_2 >= 12:
                depth = 4
            else:
                depth = 6

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

    def evaluate_state(self, state: GameState) -> float:
        """
        Evaluate the game state and return a heuristic value.
        """
        players = state.players
        players_id = [p.get_id() for p in players]
        player_id = self.get_id()

        player_score = state.scores[self.get_id()]
        opponent_score = state.scores[players_id[0]] if players_id[0] != player_id
            else state.scores[players_id[1]]

        dic_player_pieces = state.players_pieces_left
        dic_pieces_1 = dic_player_pieces[player_id]
        cite = ['RC', 'GC', 'BC', 'YC']
        ressource = ['RR', 'GR', 'BR', 'YR']
        nb_cite, nb_ressource = sum(dic_pieces_1[c] for c in cite), sum(dic_pieces_1[
            r] for r in ressource)

        return player_score - opponent_score + (1 - 4 * state.step / 40) * nb_cite +
            (1 + 4 * state.step / 40) * nb_ressource
```