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 : Skypiea_v5

Principe

Notre agent est très proche de celui de type `skypiea`. C'est-à-dire, un Minimax Alpha-Beta prunning pur.

Choix du type d'algorithme

Nous avons choisi ce type d'algorithme car il présentait les meilleurs résultats entre nos agents.

Heuristique

Nous nous sommes concentrés sur l'heuristique décrite dans la partie Alpha-Beta Pruning, après avoir réalisé un benchmark pour les magiques numbers 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`

Implémentation

Le code de notre agent est disponible ici

Performances

Pour étudier les performances, nous allons étudier

- D'une part, les performances Abyss :

Agents	Notes	Elo	Total matches	Victoires	Défaites	W/L	Nombre de diversités	NbDivercités/Game	Points marqués	Points concédés	Marqués/Concédés
Skypiea v5	Alpha Beta Pruning	1282	13	10	3	3,333333333	30	2,307692308	259	201	1,288557214
Impel Down 25000	MCTS puis AB	937	9	6	3	2	22	2,444444444	200	147	1,360544218
Wano		588	15	2	13	0,153846154	11	0,7333333333	213	340	0,6264705882
Impel Down 15000	MCTS puis AB	1006	1	1	0	#DIV/0!	5	5	29	16	1,8125
Water Seven		971	1721	903	818	1,10391198	1972	1,145845439	29541	28239	1,046106449
Sabondy		1108	226	142	84	1,69047619	291	1,287610619	4224	3718	1,136094675
Skypiea v2.2	Alpha Beta Pruning	1165	25	17	8	2,125	26	1,04	447	386	1,158031088
Skypiea v2	Alpha Beta Pruning	1275	489	350	139	2,517985612	636	1,300613497	9085	7472	1,215872591
Little Garden		713	707	337	370	0,910810811	488	0,690240453	11860	11468	1,034182072
Skypia		960	5	2	3	0,666666667	0	0	91	88	1,034090909

Nous observons que la plupart de nos agents

- D'autre part, les performances entres nos agents :

Pistes d'améliorations

Conclusion

Annexes

Code de l'agent final : 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 pièces 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
        )
```

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

```
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-être 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 modifié 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 pièces 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
```