IA For Decision Making - Tron Game (Minimax algorithme)

Hafsa BOUGHEMZA

Ryan CASSISI

Lucas SALI–ORLIANGE

Encadré par Monsieur Giovanni CHIERCHIA

April 2024

Résumé

Ce rapport contient les informations relatives au projet de **DSIA 4301B - IA for Decision Making**, réalisé au sein d'ESIEE Paris.

Mots-clefs: IA, Decision, Making, Tron, Game, Minimax, Algorithme, ESIEE, E4

Table des matières

Table des figures

Introduction

L'objectif de ce projet est d'exploiter les connaissances acquises au sein de l'unité "**DSIA 4301B - IA for Decision Making**", afin d'implémenter un algorithme au sein d'un jeu comme "*Flappy Bird*", "*Tron*", "*Blackjack*", ...

Présentation du jeu

Le jeu sélectionné est "**Tron Game**" à **2** joueurs. Le jeu se déroule sur une grille de jeu, encadrée par des murs, où les deux joueurs se déplacent à moto selon le folklore du jeu. Ces dernières au cours de leurs déplacements, laissent une traînée qui se matérialise en mur. Ainsi, l'objectif de chaque joueur, est de faire rentrer en collision son adversaire et sa traînée tout en évitant la traînée ennemie.

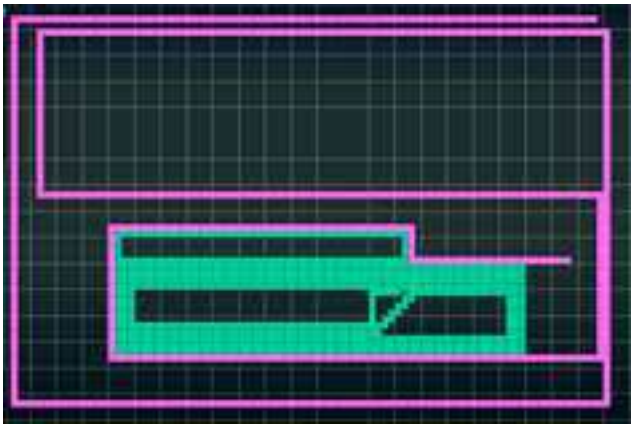


FIGURE 1 – Extrait du jeu Tron - Source : [igrynadvoih.ru]

Règles du jeu

Les règles sont les suivantes :

Les déplacements des joueurs

Les joueurs peuvent se déplacer horizontalement et verticalement sur la grille en excluant des mouvements diagonaux. Il est donc impossible de traverser des obstacles de manière horizontale. Il n'est pas possible de se déplacer dans un autre joueur.

Les obstacles

Les obstacles au sein du jeu sont multiples. Il y a dans un premier temps les murs à la création de la carte (essentiellement les bordures, mais il est possible d'en ajouter au milieu de la grille de jeu), les murs formés par les trajectoires du joueur 1, ainsi que ceux formées par le joueur 2.

Conditions de fin de partie

Avec deux joueurs dans la partie, 3 issues sont possibles. La première consiste à ce que le joueur 1 rentre en collision avec un obstacle, tandis que le second continue de rouler sans rencontrer d'obstacle. Ainsi, le joueur 2 sera déclaré vainqueur. La seconde issue est l'inverse déclarant ainsi le joueur 1 vainqueur. Enfin, dans le cas où les deux joueurs rentrent simultanément en collision avec un obstacle, l'égalité est déclarée.

Modélisation

Instances du jeu

Au sein du jeu, on comptera ainsi **3 instances majeures**. La première sera une instance "**Game**" qui contiendra les données relatives au jeu. On pourra notamment y compter la grille de jeu, incluant les obstacles, mais aussi les joueurs, et enfin le statut du gagnant de la partie.

```
class Game:
    """
    A class for managing the Game of the Tron game.

    Attributes:
        width (int): The width of the game grid.
        height (int): The height of the game grid.
        grid (np.ndarray): A numpy array representing the game grid.
        player_1 (Player): The first player of the game.
        player_2 (Player): The second player of the game.
        winner (int): The id of the winning player, or None if the game is
         ongoing.
    """

    def __init__(self, width: int, height: int, player_1: Player,
                 player_2: Player) -> None:
        """
        :param width: to define the width of the grid
        :param height: to define the height of the grid
        :param player_1: play the game as player 1
        :param player_2: play the game as player 2
        """
        self.width, self.height = width, height
        self.player_1 = player_1
        self.player_2 = player_2
        self.grid = self.init_grid(self.width, self.height, player_1, player_2)
        self.winner = None
```

FIGURE 2 – Présentation de la classe "**Game**"

Les **2 autres instances** sont finalement les **joueurs** représentés au sein de la classe "**Player**". Ces dernières ont pour attributs, entre autres, leurs coordonnées "*(x, y)*", si une intelligence artificielle doit leur être assignée, son score, sa couleur, etc...

```
class Player:
    """
    Manage all the features related to the Player.

    Attributes:
        x (int): The position of the player along the x-axis.
        y (int): The position of the player along the y-axis.
        ai (bool): Indicates whether the player is an AI or not.
        dead (bool): Indicates whether the player is dead or alive.
        score (int): The score of the player, used for decision-making.
        number (int) : Value of the player.
        color (str) : Color of the player.
        wall_color (str) : Color of the wall from the player.
    """

    def __init__(self, x: int, y: int, number: int, color: str, wall_color: str,
                 score=0, ai: bool = True):
        """
        Initialize the Player object
        :param x: position of the player in abs
        :param y: position of the player in ord
        :param number: value of the player
        :param color: color of the player
        :param wall_color: wall color of the payer
        :param score: score of the player to make decision
        :param ai: indicates whether the player is an AI or not
        """
        self.x = x
        self.y = y
        self.ai = ai
        self.dead = False
        self.score = score
        self.number = number
        self.color = color
        self.wall_color = wall_color
```

FIGURE 3 – Présentation de la classe "**Player**"

Les états du jeu

Le "**minimax algorithm**" requiert la création de **nouveaux états** afin de prendre sa décision. Ainsi, une **gestion des états** sera expliquée au sein de l'algorithme.

‘

Pour ce qui est du jeu global, pour chaque situation (**états actuels** du joueur 1, du joueur 2, de la grille), l'état est dans un premier temps affiché (on y verra notamment les joueurs, les obstacles, le vainqueur s'il est décidé). Dans un second temps, nous appliquons l'algorithme afin de faire ressortir, selon lui, la meilleure décision pour chacun des joueurs. A l'issu de la compilation de l'algorithme. Puis, nous mettons à jour les instances à travers la création de nouveaux obstacles, une mise à jour des scores ainsi que de la position et des statuts (vivant ou mort) des joueurs. Enfin, nous bouclons de nouveau, afin de générer de nouveaux choix pour les joueurs.

Les actions du jeu

Comme expliqué précédemment, les **actions** sont horizontales ou verticales. Ainsi, en fonction du "*string*" passé en paramètre, une fonction permet de déplacer le joueur en mettant à jour sa position "*(x, y)*". Ce mouvement prendra effet à la prochaine mise à jour grâce à la méthode du "**Player**" intitulé "*apply_move*" :

```
def apply_move(self, picked_move: tuple[int, int]) -> None:
    """
    Apply a movement to the Player
    :picked_move: tuple of integers (i, j) representing the new move, where:
            - i represents the new position along the x-axis.
            - j represents the new position along the y-axis.
    """
    self.x += picked_move[0]
    self.y += picked_move[1]
```

FIGURE 4 – Fonction de la classe "**Player**" : apply_move()

Les coûts et récompenses du jeu

Afin d'obtenir la meilleure décision conformément aux capacités de l'algorithme, il est nécessaire d'attribuer un score à chaque décision possible. Pour cela, un système de coûts et récompenses a été pensé pour pouvoir valoriser les meilleures décisions, et pénalisant avec des coûts les moins bonnes décisions. La fonction suivante réalise les coûts et les récompenses associés aux actions potentielles :

Les récompenses

Une première raison de valoriser le joueur serait naturellement lorsque ce dernier **élimine son adversaire**. Une seconde option pour valorisant le joueur est beaucoup plus complexe. Dans l'hypothèse où les deux joueurs viennent à former un mur séparant ces derniers, il est impératif de valoriser celui qui **s'enferme dans une zone avec une surface plus importante**.

Les coûts

Ces derniers sont plus nombreux. Par opposition aux récompenses, un coût est imposé lorsque le joueur **subit une d'élimination**. Dans un second temps, on pénalise aussi le joueur s'il **réalise une égalité**. En effet, nous cherchons à valoriser la victoire et non l'égalité ou la défaite. Enfin, nous pénalisons les joueurs qui **s'enferme dans une zone moins importante que son adversaire**.

```
def evaluate_board(self, grid: np.array, player_1: Player,
                   player_2: Player) -> int:
    """
    Evaluate the board bases on the player_1 (if player 2 loses, points are
    increasing, while decreasing when player 2 wins)
    :param grid: board of the game to evaluate
    :param player_1: player evaluated
    :param player_2: player to define the scoring for player_1
    :return: score_board of the game for player_1
    """
    score_board = 0

    # If the opponent has won
    if player_1.dead and not player_2.dead:
        score_board -= 100
        # Lose even more points if the player has been killed by the
        # other player
        if self.got_killed_by_other_player(player_1):
            score_board -= 500
        # If the player has won
    elif not player_1.dead and player_2.dead:
        score_board += 100
        # Win even more points if the player killed the other player
        if self.got_killed_by_other_player(player_2):
            score_board += 500
    # In case of a draw (both dead)
    if player_1.dead and player_2.dead:
        score_board -= 50
```

FIGURE 5 – Fonction de la classe "**Game**" : evaluate_board()

Algorithme Minimax

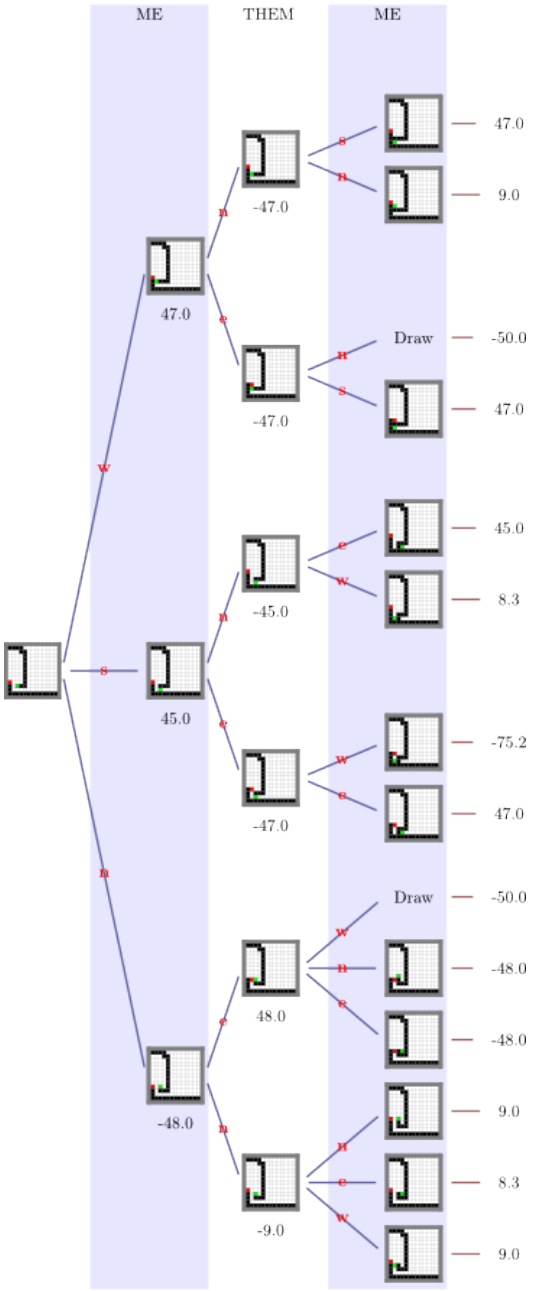


FIGURE 6 – Algorithme Minimax

Présentation du principe de l'algorithme

L'algorithme a pour objectif de maximiser le score du joueur cible, et de minimiser celui de l'adversaire d'où son nom "**minimax**". L'algorithme dépend de la profondeur qu'on lui attribue. A la première profondeur, on cherche à maximiser le score de notre joueur cible. Pour cela, on explore toutes les possibilités conformément à la position initiale. Puis à la seconde profondeur, on cherchera à minimiser celle de notre adversaire en testant toutes les possibilités de l'adversaire par rapport à toutes les possibilités de notre joueur cible. On réitérera ce cycle conformément à la profondeur indiquée. Ainsi, la fonction exécutant l'algorithme est dite "*récursive*", où la maximisation est un paramètre booléen dont la valeur alterne à chaque itération.

Cas de base

Le cas de base de notre récursion se produit lorsque la profondeur est à "**0**" ou bien **qu'un des deux jours au minimum est mort**. Dans le cas échéant, nous pouvons retourner la valeur du score avec une absence de mouvement, caractérisée par le tuple "*(0, 0)*".

Cas de la maximisation

En cas de maximisation, nous créons une variable **score** initialisée à "**-**∞" afin que tout nouveau score puisse être sélectionné lorsque l'on souhaitera prendre le score maximale entre la valeur trouvée et le score actuel. Par la suite, nous déterminons les actions possibles au vu de la **grille** passée en paramètre, et donc de ses obstacles, ainsi que des**positions des joueurs 1 et 2**. Ensuite, chaque action explorera son propre arbre, où elle exécutera de manière récursive l'algorithme sur sa nouvelle grille, et avec les nouvelles positions des joueurs. Une fois la profondeur finale de l'arbre atteinte, pour chaque niveau de profondeur, c'est le score maximale qui sera remonté jusqu'à la situation initiale (grâce au cas de base). Ainsi, l'action ayant le meilleur score pour une profondeur "**k**" sera sélectionnée. Afin de diminuer le temps de compilation, un système d' "**alpha-beta pruning**" intervient afin de ne pas explorer les branches peu prometteuses (scores inférieurs à ceux déjà obtenus). Ainsi, le temps de compilation est diminué, et la profondeur croissante complexifiant atténuée, permettant d'allouer une profondeur plus importante. Ainsi, nous retournerons la valeur du score ainsi que l'action associée.

Cas de la maximisation

Pour le cas de la minimisation, la réflexion est similaire. Certains points divergent néanmoins, à l'instar de la variable **score**, sa valeur vaut désormais "**+**∞" comme nous cherchons à la minimiser. On prendra donc la valeur minimale entre le score actuelle et le score remontée au sein de chaque situation. En revanche, un point qui ne change pas est le "*return*" de la fonction qui donc le même que lors de la maximisation.

Listing 1 – Fonction de la classe "**Game**" : minimax()

```
def minimax(self, depth: int, maximizing_player: Player,
            minimizing_player: Player, alpha: float, beta: float,
            maximizing_player_1=True) \
        -> tuple[int, tuple[int, int]]:
    """
    Use the minimax algorithm to explore a tree and select the best output
    :param depth: of the tree for the minimax algorithm
    :param maximizing_player: player to maximize the score
    :param minimizing_player: player to minimize the score
    :param maximizing_player_1: maximize the score of the player 1
    :param alpha: the best score for the maximizing player
    :param beta: the best score for the minimizing player
    :return: tuple of integer and tuple of integers (a, (i, j)) representing
     the best move, where:
            - a represent the score_board of the current state
            - i represents the new position along the x-axis
            - j represents the new position along the y-axis
    """
    # We check if the players should be alive or not
    self.check_alive(maximizing_player)
    self.check_alive(minimizing_player)
    # As we use this function recursively, we need to check the end game
    # condition. It will change 'winner' attribute if the game is over
    self.check_end_game(maximizing_player, minimizing_player)

    # We define the base case when the max depth is reached/game is over
    if depth == 0 or self.winner is not None:
        return self.evaluate_board(self.grid, maximizing_player,
                                   minimizing_player), (0, 0)

    best_move = None

    if maximizing_player_1:
        # As we maximize, we start from -inf
        best_score = float('-inf')

        # We retrieve all the possible moves
        possible_moves = self.get_allowed_moves(maximizing_player)

        # We explore the nodes from the current state
        for move in possible_moves:
        # We assign a new position to the maximizing_player
            maximizing_player.apply_move(move)

            score, _ = self.minimax(depth - 1, maximizing_player,
                                    minimizing_player,
                                    alpha, beta,
                                    not maximizing_player_1)
```

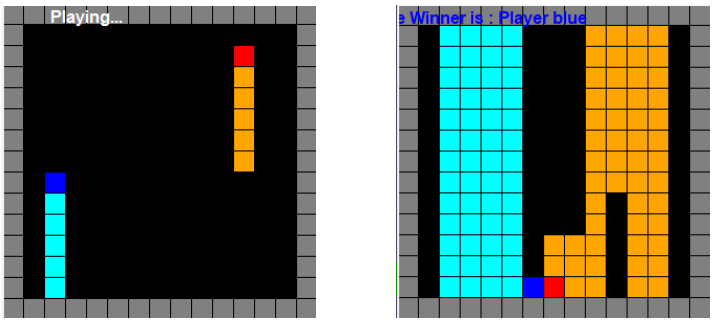
Listing 2 – Fonction de la classe "**Game**" : minimax() (suite)

```
            # We assign the oldest position to the maximizing_player to
            # explore the other branch
            maximizing_player.undo_move(move)

            # In case the score is superior to the best one we keep it
            if score > best_score:
                best_score = score
                best_move = move

            # Update alpha
            alpha = max(alpha, score)
            # Prune the branch if beta <= alpha
            if beta <= alpha:
                break

    # In the case we need to minimize the score of the minimizing_player
    else:
        # As we minimize, we start from inf
        best_score = float('inf')

        # We retrieve all the possible moves
        possible_moves = self.get_allowed_moves(minimizing_player)

        # We explore the nodes from the current state
        for move in possible_moves:
            # We assign a new position to the minimizing_player
            minimizing_player.apply_move(move)

            # We generate the score for this new branch
            score, _ = self.minimax(depth - 1,
                                    maximizing_player,
                                    minimizing_player,
                                    alpha, beta,
                                    not maximizing_player_1)

            # We assign the oldest position to the minimizing_player to
            # explore the other branch
            minimizing_player.undo_move(move)

            # In case the score is inferior to the best one we keep it
            if score < best_score:
                best_score = score
                best_move = move

            # Update beta
            beta = min(beta, score)
            # Prune the branch if beta <= alpha
            if beta <= alpha:
                break

    return best_score, best_move
```

Résultats

Présentation des résultats

Pour ce qui est des résultats, notre première constatation est que le résultat n'est pas optimal. En effet, certaines prises de décisions peuvent être qualifiées de mauvaises. Il arrive encore trop souvent que certaines décisions que nous prendrions instinctivement afin de confronter notre adversaire ne survienne que trop tard dans la partie. Comme on peut le voir ci dessous, les joueurs ne pouvant se détecter et scorer par manque de profondeur ne cherchent pas à se rencontrer :



(a) Situation initiale (b) Situation Finale

FIGURE 7 – Comparaison de situations

Dans ces situations, on observe clairement que chaque joueur mène sa propre existence sans prendre excessivement compte des mouvements de l'adversaire alors qu'un humain chercherait à coincer son adversaire.

L'approche de cet algorithme est intéressante, mais ne porte ses fruits que trop tard par rapport à d'autres algorithmes. Ainsi, il est possible qu'en ne prenant en compte que trop peu de profondeur (car extrêmement gourmand en cas de grande profondeur même avec de l'alpha-beta pruning), notre IA se retrouve piégé par des IA plus malignes ou même des joueurs humains.

Pistes d'amélioration

Afin d'améliorer notre IA, il faudrait donc se pencher sur de nouvelles conditions de scoring afin de complexifier ses décisions, et peut-être, prendre de meilleures décisions à l'avenir. Néanmoins, cet algorithme reste gourmand en matière de ressource de compilation et ne saurait dépasser une certaine profondeur