Rapport de création du jeu du Dame (English Draughts)

Sous la supervision de Mme Daoudi

Réalisé par : ELatmani Oumaima, Attaf kawtar,khaoula Bouheka

Table de matières

I. Définition du jeu :

Le jeu de dames est considéré comme un jeu compliqué avec 10^20 positions légales possibles rien que dans la version anglaise du jeu de dames (plateau 8*8) (beaucoup plus dans les dimensions supérieures).

Anglaise (plateau 8*8) (beaucoup plus pour les dimensions supérieures). Notre approche consiste à créer un agent

Agent informatique basé sur l'algorithme Minimax, avec des améliorations possibles, qui est l'état de l'art dans les jeux à un contre un.

Dans les jeux à un contre un. Nous commençons par implémenter un joueur minimax de base avec une heuristique d'évaluation de base, et

Fonction d'évaluation afin d'obtenir une meilleure approximation de la valeur de la position pour notre joueur informatique.

II. Les règles de jeu :

- La prise régulière n'est possible que dans la direction avant, sauf si la prise fait partie d'une prise en plusieurs étapes. sur le même coup, il est possible de capturer aussi en arrière.

- b. Lorsqu'un pion atteint la dernière ligne, il est couronné roi. Les rois peuvent se déplacer à la fois vers l'arrière et vers l'avant et ils peuvent également capturer en avant et en arrière.

Notre implémentation est générique et peut être facilement étendue à des plateaux de dimensions encore plus élevées qui préservent les propriétés du jeu actuel (les mêmes règles, les mêmes règles de jeu et les mêmes règles de jeu).

les mêmes règles, les 3 premières lignes de chaque côté contiennent des pièces).

III.L’algorithme minimax :

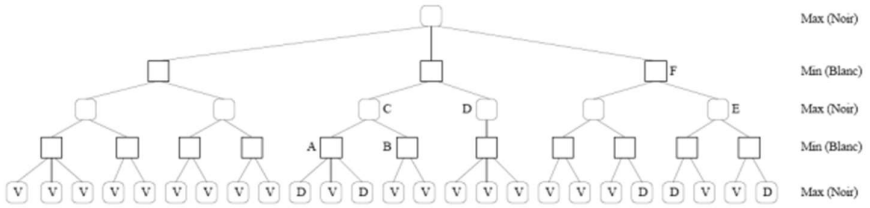
La famille des algorithmes Minimax permet la sélection d’un coup dans les jeux dits « à deux joueurs à somme nulle et information complète », famille dans laquelle se trouve la plupart des jeux de réflexion (Othello, échecs, morpion, puissance 4, Go, etc.) :

Somme nulle : les gains d’un joueur sont exactement l’opposé des gains de l’autre joueur (ces gains peuvent donc être négatifs).

Information complète : lors de sa prise de décision (i.e. du choix d’un coup à jouer dans le cas qui nous intéresse), chaque joueur connaît précisément :

- ses possibilités d’action,
- les possibilités d’action de son adversaire
- les gains résultants de ces actions,
- les motivations de son opposant.

L'algorithme MinMax est associé à un arbre de jeu.



Cet arbre est composé:

- d'un nœud racine représentant la configuration de jeu à analyser,
- de nœuds fils qui représentent chacun une configuration du jeu atteignable depuis le nœud racine,
- de branches qui symbolisent les coups possibles dans une configuration données,
- de feuilles qui représentent chacune une configuration finale du jeu.

A chaque feuille (configuration finale) est associée une valeur:

- positive en cas de victoire,
- 0 en cas de match nul,
- négative en cas de défaite.

Le principe de l'algorithme dissocie les nœuds où l'algorithme pourra jouer (nœuds MinMax) des nœuds où l'adversaire pourra jouer (nœuds adverses):

• nœuds MinMax: on associe à ces nœuds la plus grande valeur associée aux nœuds suivants,

• nœuds adverse:

on associe à ces nœuds la plus petite valeur associée aux nœuds suivants. En effectuant cette analyse jusqu'aux feuilles, on peut déterminer quel est le meilleur coup à jouer.

Comme il n'est pas toujours possible d'explorer l'arbre jusqu'aux feuilles, il est fréquent d'avoir recourt à des fonctions d'évaluations qui retournent une estimation d'une partie inachevée.

NegaMax Minimiser une valeur revient à maximiser l'opposée de cette valeur. En s'appuyant sur cette constatation, il est possible d'écrire une version plus compacte de l'algorithme:

IV. L'implémentation de notre jeu :

Lors de la création de notre jeu on a suivi un processus comme suit :

- La création des boards et les pièces.
- Poser les constant qu'on va utiliser plus fréquemment
- Après rendre notre code dynamique
- Puis supprimer le code de l'opposée 'humain vs humain '
- Enfin l'implémentation du l'algorithme du minimax.

la bibliothèques centrale dans notre code c'est pygame.

Notre code comporte 6 fichiers.

Main : c'est la fonction principe qui fait l'appel a tout le code et il suffit de l'exécuté pour avoir la sortie finale

Constantes : est un fichier dans laquelle on stocke notre constants.

Algorithme : qui définit notre algorithme minimax

Game : qui formalise les règles de jeu et définir les évènements.

Board : rassemble le code de création de la fenêtre affichée lors du ''run '' du code et après son modification par le capture d'une pièce.

Piece : c'est le fichier qui porte le code de création de nôtres pièces et leurs emplacements.

```python
import pygame
from checkers.constants import WIDTH, HEIGHT, SQUARE_SIZE, RED, WHITE
from checkers.game import Game
from minimax.algorithm import minimax

FPS = 60

WIN = pygame.display.set_mode((WIDTH, HEIGHT))
pygame.display.set_caption('Checkers')

def get_row_col_from_mouse(pos):
    x, y = pos
    row = y // SQUARE_SIZE
    col = x // SQUARE_SIZE
    return row, col

def main():
    run = True
    clock = pygame.time.Clock()
    game = Game(WIN)

    while run:
        clock.tick(FPS)
```

Tout d'abord on importe la bibliothèque pygame et on l'initialise par pygame.init().

Après on importe le fichier du constants qui porte la largeur de la fenêtre et sa hauteur

Après avoir l'importer on les affectés à l'attribut du pygame qui responsable sur la création et l'affichage de la fenêtre préliminaire pyagme.display.set_mode()

. Puis on fixe la vitesse d'exécution du jeu en 60 pour *qu*'il soit compatible avec touts les pc (bas performance que ls hauts) fps=60.

Pygame.display.set_caption() c'est l'attribut qui nous affiche le nom du jeu en haut de la fenêtre .

Après on définit une fonction main() qui fixe la vitesse clock.tick(fps) d'exécution du jeu ainsi que les règles du jeu .

```python
import pygame
from .constants import BLACK, ROWS, RED, SQUARE_SIZE, COLS, WHITE
from .piece import Piece

class Board:
    def __init__(self):
        self.board = []
        self.red_left = self.white_left = 12
        self.red_kings = self.white_kings = 0
        self.create_board()

    def draw_squares(self, win):
        win.fill(BLACK)
        for row in range(ROWS):
            for col in range(row % 2, COLS, 2):
                pygame.draw.rect(win, RED, (row*SQUARE_SIZE, col *SQUARE_SIZE, SQUARE_SIZE, SQUARE_SIZE))
```

Dans le fichier Board on a importé le pieces, les constants et bien évidemment la bibliothèque pygame afin de l'utiliser plus tard dans notre code.

Tout d'abord on a crié un classe Board qui initialise le nombre des pièces de chaque joueuret le roi par 0 (car une pièce n'est devenu un roi si seulement si elle est passée à l'autre coté de règne du l'adversaire) en utilisant la constructeur __init__(self)

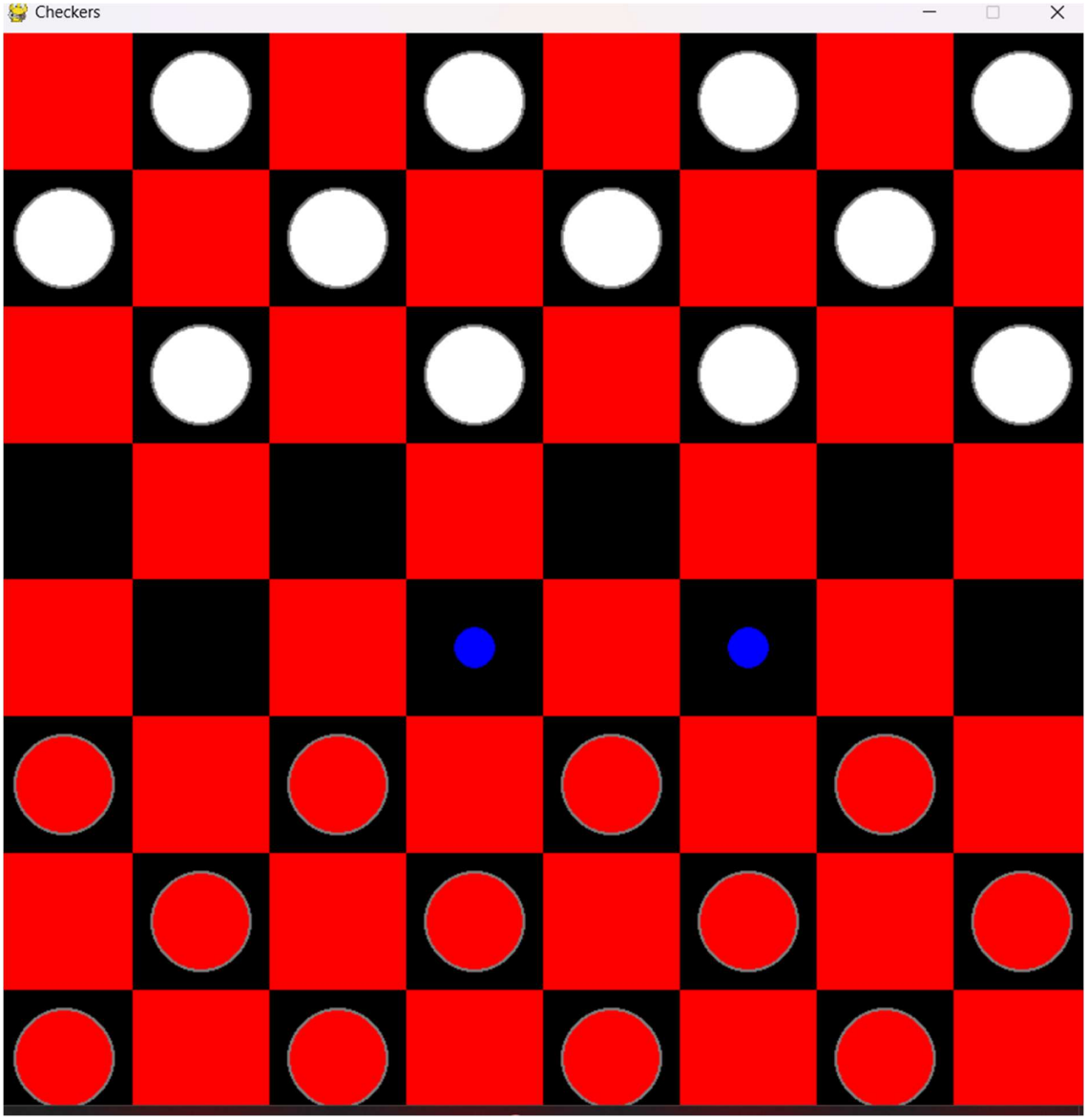
- On a définit La fonction draw_squares() , qui comme objectif de tracer les petites carrées les rouges et les noir.
- Tout d'abord on remplit la fenêtre en noire puis on a fait une loop pour faires des petites carreaux en rouge (pour qu'ils soit croisées avec les carreaux noirs) win.fill(black) et pyagme.draw.rect()

Dans la création des pièces on a postuler le padding (la marge entre la pièce et la case) et Outline (son contour) pour les exploiter dans la calcule du rayon de la pièce.

- On initialise notre variable dans le constructeur pour l'utiliser plus tard.
- On définit la fonction calc_pos() qui détermine les coordonnées de la pièce.
- Puis on a définit la fonction draw(self,win) qui dessine les pièces sur la fenêtre , on définit le rayon de notre pièce pour qu'elle soit compatible avec la case,et on a utilisé l'attribut de pygame pygame.draw.circle() qui dessine les cercle dans les cases selon leur cordonnées .
- Si la pièce est roi on y dessine à l'intérieur un couronne win.blit().

- On a dessiné un cercle avec la couleur gris plus grand que l'autre qui il s'en vas disposer.

  Comme le screen ci-dessous montre.



- La fonction make_ king() définit la pièce est ce qu'elle est roi (en l'affectant true)
- La fonction move() définit une méthode move au sein d'une classe. Elle prend deux paramètres, row et col, et met à jour les attributs row et col de l'objet avec les valeurs fournies. Après la mise à jour de la position, elle appelle ensuite la méthode calc_pos, vraisemblablement pour recalculer les coordonnées x et y en fonction des nouvelles valeurs de row et col.
- la méthode __repr__ retourne la chaîne de caractères résultante de la conversion de l'attribut color de l'objet en une chaîne à l'aide de str(self.color).

```
def create_board(self):# on initilise  l'emplacement  des pieces des joeurs
    for row in range(ROWS):
        self.board.append([])
        for col in range(COLS):
            if col % 2 == ((row +  1) % 2):
                if row < 3:
                    self.board[row].append(Piece(row, col, WHITE))
                elif row > 4:
                    self.board[row].append(Piece(row, col, RED))
                else:
                    self.board[row].append(0)
            else:
                self.board[row].append(0)

def draw(self, win):#on affiche le dessin
    self.draw_squares(win)
    for row in range(ROWS):
        for col in range(COLS):
            piece = self.board[row][col]
            if piece != 0:
                piece.draw(win)
```

Après on initialise l'emplacement des pièces, en haut les 3 premiers lignes consacré pour les pièces blancs et les 3 dernières lignes de notre fenêtre consacrée à les pièces rouge.

La fonction draw ()est la fonction qui dessine toute la fenêtre avec ces pièces et tout.

```
def get_all_pieces(self, color):
    pieces = []
    for row in self.board:
        for piece in row:
            if piece != 0 and piece.color == color:
                pieces.append(piece)
    return pieces
```

elle prend un paramètre color pour filtrer les pièces en fonction de leur couleur. La méthode parcourt les lignes du plateau, puis les pièces de chaque ligne. Si une pièce n'est pas égale à 0 (en supposant que 0 représente un espace vide sur le plateau) et que sa couleur correspond à la couleur spécifiée, la pièce est ajoutée à la liste des pièces.

`for row in self.board:` : Cela parcourt chaque ligne du plateau.

`for piece in row:` : Cela parcourt chaque pièce dans la ligne actuelle.

`if piece != 0 and piece.color == color:` : Cette condition vérifie si la pièce actuelle n'est pas égale à 0 (pas un espace vide) et si sa couleur correspond à la couleur spécifiée.

`pieces.append(piece)` : Si la condition est vraie, la pièce est ajoutée à la liste des pièces.

```
    def evaluate(self):
        return self.white_left - self.red_left + (self.white_kings * 0.5 - self.red_kings * 0.5)

    def get_all_pieces(self, color):
```

Cette méthode semble calculer un score d'évaluation simple pour un état de plateau dans un jeu.

```
    def move(self, piece, row, col):
        self.board[piece.row][piece.col], self.board[row][col] = self.board[row][col], self.board[piece.row][piece.col]
        piece.move(row, col)

        if row == ROWS - 1 or row == 0:
            piece.make_king()
            if piece.color == WHITE:
                self.white_kings += 1
            else:
                self.red_kings += 1
```

self.board[piece.row][piece.col], self.board[row][col] = self.board[row][col], self.board[piece.row][piece.col] : Cette ligne échange les positions de la pièce actuelle (piece) avec la nouvelle position spécifiée par les coordonnées (row, col) dans le tableau self.board. Cela représente le déplacement de la pièce.

piece.move(row, col) : Cette ligne met à jour les coordonnées de la pièce (piece) avec les nouvelles coordonnées (row, col).

Ensuite, le code vérifie si la pièce a atteint l'une des lignes extrêmes du plateau, c'est-à-dire si row est égal à ROWS - 1 ou row est égal à 0. Si c'est le cas, la pièce est promue en tant que roi avec la méthode make_king().

Si la couleur de la pièce est blanche (piece.color == WHITE), alors le compteur de rois blancs (self.white_kings) est augmenté de 1. Sinon, si la couleur de la pièce est rouge, le compteur de rois rouges (self.red_kings) est augmenté de 1.

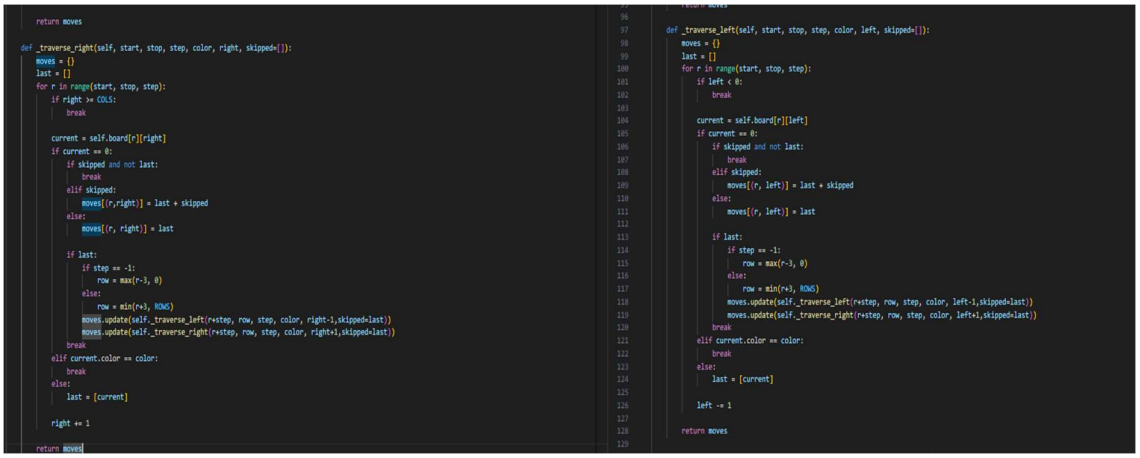
En résumé, cette méthode effectue le déplacement d'une pièce sur le plateau, met à jour les informations de position de la pièce, vérifie si la pièce atteint une ligne extrême, et si c'est le cas, la promeut en roi, mettant à jour également le compteur de rois correspondant à la couleur de la pièce.

```python
def remove(self, pieces):
    for piece in pieces:
        self.board[piece.row][piece.col] = 0
        if piece != 0:
            if piece.color == RED:
                self.red_left -= 1
            else:
                self.white_left -= 1

def winner(self):
    if self.red_left <= 0:
        return WHITE
    elif self.white_left <= 0:
        return RED

    return None

def get_valid_moves(self, piece):
    moves = {}
    left = piece.col - 1
    right = piece.col + 1
    row = piece.row

    if piece.color == RED or piece.king:
        moves.update(self._traverse_left(row -1, max(row-3, -1), -1, piece.color, left))
        moves.update(self._traverse_right(row -1, max(row-3, -1), -1, piece.color, right))
    if piece.color == WHITE or piece.king:
        moves.update(self._traverse_left(row +1, min(row+3, ROWS), 1, piece.color, left))
        moves.update(self._traverse_right(row +1, min(row+3, ROWS), 1, piece.color, right))

    return moves
```

remove(self, pieces) : Cette méthode prend une liste de pièces en argument et les retire du plateau en mettant à jour les coordonnées correspondantes dans le tableau self.board. Si la pièce est de couleur rouge, le compteur self.red_left est décrémenté, sinon, si la pièce est de couleur blanche, le compteur self.white_left est décrémenté.

winner(self) : Cette méthode vérifie s'il y a un gagnant dans le jeu. Si le nombre de pièces rouges restantes (self.red_left) est inférieur ou égal à zéro, alors les pièces blanches (WHITE) ont gagné. Si le nombre de pièces blanches restantes (self.white_left) est inférieur ou égal à zéro, alors les pièces rouges (RED) ont gagné. Sinon, la méthode renvoie None, indiquant qu'il n'y a pas encore de gagnant.

get_valid_moves(self, piece) : Cette méthode génère et retourne un dictionnaire de mouvements valides pour une pièce donnée. Elle utilise des méthodes de traversée gauche et droite (_traverse_left et _traverse_right) pour explorer les mouvements possibles en fonction de la couleur et du statut de roi de la pièce. Les mouvements valides sont stockés dans le dictionnaire moves.



```
    return moves

def _traverse_right(self, start, stop, step, color, right, skipped=[]):
    moves = {}
    last = []
    for r in range(start, stop, step):
        if right >= COLS:
            break

        current = self.board[r][right]
        if current == 0:
            if skipped and not last:
                break
            elif skipped:
                moves[(r,right)] = last + skipped
            else:
                moves[(r, right)] = last

            if last:
                if step == -1:
                    row = max(r-3, 0)
                else:
                    row = min(r+3, ROWS)
                moves.update(self._traverse_left(r+step, row, step, color, right-1,skipped=last))
                moves.update(self._traverse_right(r+step, row, step, color, right+1,skipped=last))
            break
        elif current.color == color:
            break
        else:
            last = [current]

        right += 1

    return moves
```

```
def _traverse_left(self, start, stop, step, color, left, skipped=[]):
    moves = {}
    last = []
    for r in range(start, stop, step):
        if left < 0:
            break

        current = self.board[r][left]
        if current == 0:
            if skipped and not last:
                break
            elif skipped:
                moves[(r, left)] = last + skipped
            else:
                moves[(r, left)] = last

            if last:
                if step == -1:
                    row = max(r-3, 0)
                else:
                    row = min(r+3, ROWS)
                moves.update(self._traverse_left(r+step, row, step, color, left-1,skipped=last))
                moves.update(self._traverse_right(r+step, row, step, color, left+1,skipped=last))
            break
        elif current.color == color:
            break
        else:
            last = [current]

        left -= 1

    return moves
```

la méthode get_valid_moves pour explorer les mouvements possibles d'une pièce dans différentes directions (gauche et droite).

_traverse_left : Cette méthode explore les mouvements possibles vers la gauche pour une pièce donnée. Les paramètres incluent start (ligne de départ), stop (ligne d'arrêt), step (pas de déplacement, généralement -1 ou 1), color (couleur de la pièce), left (colonne de départ à gauche), et skipped (liste des pièces sautées). Elle retourne un dictionnaire de mouvements valides.

_traverse_right : Cette méthode explore les mouvements possibles vers la droite pour une pièce donnée. Les paramètres sont similaires à _traverse_left, avec l'ajout de right (colonne de départ à droite). Elle retourne également un dictionnaire de mouvements valides.

Les deux méthodes utilisent une approche de traversée pour explorer les cases du plateau dans une direction spécifique. Elles tiennent compte des pièces présentes sur le chemin, ajoutant celles-ci à la liste last. Lorsqu'une case vide est rencontrée, les mouvements valides sont ajoutés au dictionnaire moves. En cas de pièce ennemie rencontrée, la méthode s'arrête dans cette direction.