```python
# Kono.py

import copy
import random as rd
import matplotlib.pyplot as plt
import time

## JEU KONO ##
test=[['X','X','X','R'],['X','R','R','N'],['R','R','X','X'],
['N','X','N','N']]


def test_plateau(i: int, j: int) -> bool:
    """
    Vérifie si les coordonnées (i, j) sont dans les limites
d'un plateau 4x4.

    Args:
        i (int): L'indice de ligne.
        j (int): L'indice de colonne.

    Ret:
        bool: True si (i, j) est dans le plateau, False sinon.
    """
    return 0 <= i < 4 and 0 <= j < 4

def recherche_pion(L: list[list[str]]) ->
tuple[list[tuple[int, int]], list[tuple[int, int]]]:
    """
    Recherche les positions des pions rouges ('R') et noirs
('N') dans un plateau de jeu 4x4.

    Args:
        L (list[list[str]]): Une matrice 4x4 contenant des
éléments 'R', 'N' ou 'X'.

    Ret:
        tuple[list[tuple[int, int]], list[tuple[int, int]]]:
        - Une liste des coordonnées (ligne, colonne) des pions
rouges ('R').
        - Une liste des coordonnées (ligne, colonne) des pions
noirs ('N').
    """
    MR= []  # Liste des coordonnées des pions rouges
    MN= []  # Liste des coordonnées des pions noirs

    for i in range(4):
        for j in range(4):
            if L[i][j] == "R":
                MR.append((i, j))
            elif L[i][j] == "N":
                MN.append((i, j))

    return MR, MN

```

```
047|
048| def evaluation_partie(L: list[list[str]]) -> int:
049|     """
050|     Évalue l'état actuel de la partie en comptant la
différence entre le nombre de pions rouges ('R')
051|     et le nombre de pions noirs ('N') sur le plateau.
052|
053|     Args:
054|         L (list[list[str]]): Une matrice 4x4 contenant des
éléments 'R', 'N' ou 'X'.
055|
056|     Ret:
057|         int: La différence entre le nombre de pions rouges et
le nombre de pions noirs (len(MR) - len(MN)).
058|     """
059|     MR, MN = recherche_pion(L)
060|     return len(MR) - len(MN)
061|
062|
063|
064|
065| def couleur_adversaire(couleur: str) -> str:
066|     """
067|     Retourne la couleur de l'adversaire.
068|
069|     Args:
070|         couleur (str): La couleur du joueur actuel ('R' ou
'N').
071|
072|     Ret:
073|         str: La couleur de l'adversaire ('N' si l'entrée est
'R', sinon 'R').
074|     """
075|     if couleur == "R":
076|         return "N"
077|     return "R"
078|
079| def deplacement_possible(L: list[list[str]], pos: tuple[int,
int]) -> list[tuple[int, int]]:
080|     """
081|     Détermine les déplacements possibles pour un pion donné
sur un plateau 4x4.
082|
083|     Args:
084|         L (list[list[str]]): La matrice représentant le
plateau (4x4) avec 'R', 'N' et 'X'.
085|         pos (tuple[int, int]): La position actuelle du pion
sous la forme (ligne, colonne).
086|
087|     Ret:
088|         list[tuple[int, int]]: Une liste des positions
accessibles pour le pion, sous forme de tuples (ligne, colonne).
089|     """
090|     i, j = pos
091|     pos_possible = []
```

```
092|     couleur = L[i][j]
093|     couleur_adv = couleur_adversaire(couleur)
094|     if couleur=="X":
095|         return []
096|
097|     for k in [-1, +1]:
098|         if test_plateau(i + k, j) and L[i + k][j] == "X":
099|             pos_possible.append((i + k, j))
100|         if test_plateau(i, j + k) and L[i][j + k] == "X":
101|             pos_possible.append((i, j + k))
102|         if test_plateau(i + k * 2, j) and L[i + k][j] ==
couleur and L[i + k * 2][j] == couleur_adv:
103|             pos_possible.append((i + k * 2, j))
104|         if test_plateau(i, j + k * 2) and L[i][j + k] ==
couleur and L[i][j + k * 2] == couleur_adv:
105|             pos_possible.append((i, j + k * 2))
106|
107|     return pos_possible
108|
109| def deplacement(L: list[list[str]], pos_dep: tuple[int, int],
pos_arriv: tuple[int, int]) -> list[list[str]]:
110|     """
111|     Effectue le déplacement d'un pion sur le plateau.
112|
113|     Args:
114|         L (list[list[str]]): La matrice représentant le
plateau (4x4) avec 'R', 'N' et 'X'.
115|         pos_dep (tuple[int, int]): La position initiale du
pion sous la forme (ligne, colonne).
116|         pos_arriv (tuple[int, int]): La position finale du
pion sous la forme (ligne, colonne).
117|
118|     Ret:
119|         list[list[str]]: Le plateau mis à jour après le
déplacement.
120|     """
121|     L[pos_arriv[0]][pos_arriv[1]] = L[pos_dep[0]][pos_dep[1]]
122|     L[pos_dep[0]][pos_dep[1]] = "X"
123|     return L
124|
125|
126|
127|
128|
129|
130|
131|
132|
133|
134|
135|
136| def arbre_iter(etiq: list[list[str]], controle: str,
nb_branches: int) -> tuple[
137|     list[int], list[list[int]], list[list[list[str]]],
list[str], list[list[int]]]:
```

```
    """
    Construit un arbre des états possibles d'une partie en explorant les déplacements possibles.

    Args:
        etiq (list[list[str]]): Le plateau initial sous forme de matrice 4x4 avec 'R', 'N' et 'X'.
        controle (str): La couleur du joueur actuel ('R' ou 'N').
        nb_branches (int): Le nombre maximum de branches à explorer pour chaque état.

    Ret:
        tuple:
            - S (list[int]): Liste des indices des sommets.
            - A (list[list[int]]): Liste des arêtes sous forme de couples (parent, enfant).
            - E (list[list[list[str]]]): Liste des états du plateau à chaque nœud.
            - C (list[str]): Liste des couleurs du joueur associé à chaque état.
            - listeBR (list[list[int]]): Liste des sommets classés par profondeur.
    """
    S = [0]  # Indices des sommets
    A = []  # Arêtes de l'arbre
    E = [etiq]  # États du plateau
    C = [controle]  # Couleurs contrôlant le sommet
    BR = [0]  # Profondeur des sommets
    listeBR = [[] for _ in range(nb_branches + 1)]  # Liste des sommets par profondeur
    listeBR[0].append(0)
    liste_attente = [0]
    indice = 0

    while indice < len(liste_attente):
        if BR[indice] < nb_branches:
            ind_couleur = (C[indice] == "N") * 1  # 0 si "R", 1 si "N"
            pos_pions = recherche_pion(E[indice])[ind_couleur]
            for pos in pos_pions:
                for dep in deplacement_possible(E[indice], pos):
                    plateau = copy.deepcopy(E[indice])
                    plateau = deplacement(plateau, pos, dep)
                    indice_fils = len(S)
                    S.append(indice_fils)
                    A.append([indice, indice_fils])
                    E.append(plateau)
                    C.append(couleur_adversaire(C[indice]))
                    liste_attente.append(indice_fils)
                    BR.append(BR[indice] + 1)
                    listeBR[BR[indice] + 1].append(indice_fils)
        indice += 1
```

```
181|
182|     return S, A, E, C, listeBR
183|
184|
185| import random as rd
186|
187| def choix(etiq: list[list[str]], controle: str) ->
list[list[str]]:
188|     """
189|     Détermine le meilleur coup possible pour le joueur donné
en utilisant un arbre de recherche de profondeur 6.
190|
191|     Args :
192|     - etiq : liste de listes représentant l'état actuel du
plateau.
193|     - controle : chaîne de caractères ("R" ou "N")
représentant le joueur actuel.
194|
195|     Ret :
196|     - Un nouvel état du plateau après avoir choisi le coup
optimal selon l'algorithme Minimax.
197|     """
198|     S, A, E, C, listeBR = arbre_iter(etiq, controle, 6)
199|     s = len(S)
200|
201|     # Initialisation des listes pour stocker les enfants et
parents de chaque sommet
202|     enfants = [[] for _ in range(s)]
203|     parents = [[] for _ in range(s)]
204|
205|     # Construire les listes parents-enfants
206|     for ar in A:
207|         enfants[ar[0]].append(ar[1])
208|         parents[ar[1]].append(ar[0])
209|
210|     # Initialisation des évaluations des sommets
211|     evaluations = [1000 for _ in S]
212|
213|     # Parcours inverse de l'arbre pour évaluer les sommets
214|     for k in range(len(listeBR)):
215|         for sommet in listeBR[-1 - k]:
216|             if len(enfants[sommet]) == 0:
217|                 evaluations[sommet] =
evaluation_partie(E[sommet])
218|             else:
219|                 if C[sommet] == "R":
220|                     evaluations[sommet] = max(evaluations[i]
for i in enfants[sommet])
221|                 else:
222|                     evaluations[sommet] = min(evaluations[i]
for i in enfants[sommet])
223|
224|     # Sélection du coup optimal
225|     eval = evaluations[0]
226|     choix_possibles = [enf for enf in enfants[0] if
```

```
evaluations[enf] == eval]
227|
228|     return E[rd.choice(choix_possibles)]
229|
230|
231|
232|
233| def victory(etiq: list[list[str]], controle: str) -> str:
234|     """
235|     Détermine si la partie est terminée et renvoie le gagnant.
236|
237|     Args :
238|     - etiq : liste de listes représentant l'état actuel du
plateau.
239|     - controle : chaîne de caractères ("R" ou "N")
représentant le joueur actuel.
240|
241|     Ret :
242|     - "R" si les rouges gagnent.
243|     - "N" si les noirs gagnent.
244|     - "X" si la partie continue.
245|     """
246|
247|     # Vérifier s'il reste des pions sur le plateau
248|     MR, MN = recherche_pion(etiq)
249|     if len(MR) == 0:
250|         return "N"
251|     if len(MN) == 0:
252|         return "R"
253|
254|     # Vérifier s'il reste des déplacements possibles
255|     TR, TN = True, True
256|     for pos_R in MR:
257|         if deplacement_possible(etiq, pos_R) != []:
258|             TR = False
259|             break
260|     for pos_N in MN:
261|         if deplacement_possible(etiq, pos_N) != []:
262|             TN = False
263|             break
264|
265|     if controle == "R" and TR:
266|         return "N"
267|     if controle == "N" and TN:
268|         return "R"
269|
270|     return "X"
```