IA PyRat : Livrable Final

Explication de la stratégie :

La stratégie du joueur dans notre algorithme est d'aller à chaque fois vers la pièce de fromage la plus proche et cela en utilisant deux fonctions : order_by_nearest et find_closest_piece_of_cheese.

En effet, la fonction order_by_nearest réordonne les éléments d'un dictionnaire donné par l'ordre croissant de ses valeurs comme le montre son code ici :

```
def order_by_nearest(dictionary):
    reverse={j:i for i,j in dictionary.items()}
    a=list(dictionary.values())
    a.sort()
    return {reverse[i]:i for i in a}
```

Ainsi, dans la function find_closest_piece_of_cheese, on n'a qu'à extraire le premier élément du dictionnaire que nous fournira Djikstra et retourner le chemin qu'il faut emprunter :

```
def find_closest_piece_of_cheese(graph,current_position,remaining):

    distances,routing_table=Dijkstra(graph,current_position)
    weight={i:distances[i] for i in remaining}
    order=order_by_nearest(weight)
    closest_piece_of_cheese=list(order.keys())[0]
    chemin=find_route(routing_table,current_position,closest_piece_of_cheese)
    return closest_piece_of_cheese,chemin
```

On fait cette étape à chaque fois que le rat n'est pas en mouvement, chose qu'on peut savoir avec la variable booléenne `moving` dans la fonction turn qui nous permet de trouver un chemin qui nous mène à la pièce de fromage la plus proche dans la liste `listcheese=list(set(pieces)-set(eaten_pieces))` des pièces restantes dans le labyrinthe.

Mais dans notre stratégie en fait, on prend en compte ce que fait l'adversaire par le biais de deux compteurs : `is_matched` et `is_following_me` ainsi que des variables booléennes `test` et `testing[-1]==eaten_pieces[-1]`.

En effet, `test` signale dans turn si notre adversaire a attrapé une pièce de fromage on teste également si c'est la pièce vers laquelle je me dirige à travers `testing[-1]==eaten_pieces[-1]`. Dans le cas échéant, j'abandonne ce chemin et j'emprunte un nouveau (en utilisant Dijkstra) qui me mènera à la pièce la plus proche de ma position actuelle.

J'applique également cette alternative dans le cas où l'adversaire est devant moi d'une ou deux cases (`opponent_location in [path_to_new_target[1],path_to_new_target[2]]`). En effet, si l'adversaire demeure devant moi `is_following_me` fois (dans notre code c'est 3 fois maximum), j'abandonne cette pièce et je pars à la recherche d'une nouvelle et si il y parvient avant que je fasse(`len(path_to_new_target)<3`), `is_following_me` reçoit 3 automatiquement .

Finalement, je teste aussi si on est sur la même case par le biais du compteur `is_matched`. Si mon score est plus grand que l'adversaire, je ne fais rien, mais dans le cas contraire, et si on reste collés plus que 3 fois, je change d'objectif immédiatement tout en supposant que cette pièce a été mangée à travers la syntaxe `listcheese.remove(tempted)`.

Annexe :

```
import heapq

MOVE_DOWN = 'D'
MOVE_LEFT = 'L'
MOVE_RIGHT = 'R'
MOVE_UP = 'U'

pieces=[]
moves=[]
eaten_pieces=[]
moving=False
meta_graph = {}
best_paths = {}
testing={}
path_to_new_target=[]
is_following_me=0
is_matched=0
consider_as_eaten=[]
tempted=tuple()

def create_structure():
    return []
    #Create a minheap

def empty(structure):
    return structure == []

def add_or_replace (structure, element) :
    heapq.heappush(structure, element) #element = (key(vertex), value(dista
nce from initial vertex))
    # Add an element to the minheap

def remove (structure) :
    #Before executing the function, we check wether our structure is an emp
ty list or not
    assert structure != []
    return heapq.heappop(structure)
    # remove an element from the minheap

def Dijkstra (graph,start_vertex) :

    # Initialize structure with initial vertex, at distance 0, with no pred
ecessor
    queuing_structure = create_structure()
    add_or_replace(queuing_structure, (0, start_vertex, None))
```

```
    # Initialize routing (main difference with course is we also store the length of paths stored with explored vertices)
    explored_vertices = {}
    routing_table = {}

    # Loop until all vertices have been explored
    while len(queuing_structure) > 0 :

        # Pop next vetex to visit
        (distance_to_current_vertex, current_vertex, parent) = remove(queuing_structure)
        if current_vertex not in explored_vertices :

            # Store route to it
            explored_vertices[current_vertex] = distance_to_current_vertex
            routing_table[current_vertex] = parent

            # Add its its neighbors to the structure for later consideration
            for neighbor in graph[current_vertex] :
                if neighbor not in explored_vertices :
                    distance_to_neighbor = distance_to_current_vertex + graph[current_vertex][neighbor]
                    add_or_replace(queuing_structure, (distance_to_neighbor, neighbor, current_vertex))

    # Return shortest paths and routing table
    return explored_vertices, routing_table

def find_route (routing_table, source_location, target_location) :

    # Return a sequence of locations from source to target using provided routing table
    route = [target_location]
    while route[0] != source_location :
        route.insert(0, routing_table[route[0]])
    return route

def order_by_nearest(dictionary):
    reverse={j:i for i,j in dictionary.items()}
    a=list(dictionary.values())
    a.sort()
    return {reverse[i]:i for i in a}
def updatepieces (metaGraph,location):
    test=False
    if location in metaGraph :
        eaten_pieces.append(location)
        test=True
    return eaten_pieces,test
```

```python
def build_meta_graph (mazeMap, pieces_of_cheese):

    all_locations = pieces_of_cheese
    metaGraph = {}
    bestPaths  = {}

    i = len(all_locations)-1
    while i >= 0:

        explored_vertices,routing_table = Dijkstra(mazeMap,all_locations[i])

        j = 0
        while j < i:

            if all_locations[i] not in bestPaths :
                bestPaths[all_locations[i]] = {}
                metaGraph[all_locations[i]] = {}

            if all_locations[j] not in bestPaths :
                bestPaths[all_locations[j]] = {}
                metaGraph[all_locations[j]] = {}

            if not metaGraph[all_locations[j]].get(all_locations[i], False):
                path = find_route(routing_table, all_locations[i], all_locations[j])
                distance = explored_vertices[all_locations[j]]

                metaGraph[all_locations[i]][all_locations[j]] = distance
                bestPaths[all_locations[i]][all_locations[j]] = path

                metaGraph[all_locations[j]][all_locations[i]] = distance
                bestPaths[all_locations[j]][all_locations[i]] = path[::-1]

            j += 1

        i -= 1
    metaGraph={i:order_by_nearest(metaGraph[i]) for i in metaGraph.keys()}

    return metaGraph, bestPaths
```

```python
def move_from_locations_step (source_location, target_location) :
    difference = (target_location[0] -
 source_location[0], target_location[1] - source_location[1])
    if difference == (0, -1) :
        return MOVE_DOWN
    elif difference == (0, 1) :
        return MOVE_UP
    elif difference == (1, 0) :
        return MOVE_RIGHT
    elif difference == (-1, 0) :
        return MOVE_LEFT
    else :
        raise Exception("Impossible move")

def find_closest_piece_of_cheese(graph,current_position,remaining):
    distances,routing_table=Dijkstra(graph,current_position)
    weight={i:distances[i] for i in remaining}
    order=order_by_nearest(weight)
    closest_piece_of_cheese=list(order.keys())[0]
    chemin=find_route(routing_table,current_position,closest_piece_of_chees
e)
    return closest_piece_of_cheese,chemin

def move_from_locations(locations):
    if len(locations)<2:
        pass
    else:
        move_to_apply=move_from_locations_step(locations[0],locations[1])
        locations.pop(0)
        return move_to_apply


def preprocessing (maze_map, maze_width, maze_height, player_location, oppo
nent_location, pieces_of_cheese, time_allowed) :
    global meta_graph, best_paths,pieces
    pieces=pieces_of_cheese[:]
    meta_graph,best_paths=build_meta_graph (maze_map, pieces_of_cheese)
```

```
def turn (maze_map, maze_width, maze_height, player_location, opponent_loca
tion, player_score, opponent_score, pieces_of_cheese, time_allowed) :
    global meta_graph,eaten_pieces,testing,path_to_new_target,pieces,is_fol
lowing_me
    global moving
    global tempted,is_matched
    # Si l'ennemi mange une pièce de fromage et que je n'y suis pas, on la
compte
    eaten_pieces,test = updatepieces (meta_graph,opponent_location)

    # Si une pièce de fromage a été mangée, on la compte
    eaten_pieces,_ = updatepieces (meta_graph,player_location)
    listcheese=list(set(pieces)-set(eaten_pieces))

    if moving:
        if (not path_to_new_target) or (test and testing[-
1]==eaten_pieces[-1]) or (is_following_me==3) :
                is_following_me=0
                moving = False
    if not moving:
        if is_matched==3 and opponent_score>player_score:
            is_matched=0
            listcheese.remove(tempted)
            new_target,path_to_new_target=find_closest_piece_of_cheese(maze
_map,player_location,listcheese)
            moving=True
            testing=path_to_new_target
            path_to_new_target.pop(0)
        else:
            new_target,path_to_new_target=find_closest_piece_of_cheese(maze
_map,player_location,listcheese)
            tempted=new_target
            testing=path_to_new_target
            path_to_new_target.pop(0)
            moving = True
            if len(path_to_new_target)>=3:
                if opponent_location in [path_to_new_target[1],path_to_new_
target[2]]:
                    is_following_me+=1
            else:
                is_following_me=3
            if opponent_location==player_location:
                is_matched+=1
    next_location = path_to_new_target.pop(0)
    UDRL=move_from_locations_step (player_location, next_location)

    return UDRL
```