COMPTE RENDU

SAE 1.02 - Python 2

Université d'Orléans Département Informatique
BÉATRIX Quentin, LOUVET Damien, LUISIN Nathan

I) Contexte

Dans le contexte d'une SAE au cours d'un BUT informatique, les étudiants ont eu pour but de créer une IA capable de jouer à un jeu de type "Splatoon". Pour ce faire, nous avons été mis en groupe pour implémenter l'IA, pour mettre à l'épreuve nos connaissances en Python, notre logique et notre travail d'équipe.

Nous allons tout d'abord vous parler des règles du jeu, ensuite nous enchaînerons avec la stratégie de notre IA, puis des principaux algorithmes de cette IA et enfin nous verrons l'état de notre travail, c'est-à-dire ce que nous avons réussi à faire ou pas.

II) Les règles du jeu

Le jeu se passe sur une carte de 14×14 cases, cette carte est composée de murs et de zones libres. Chaque joueur contrôle une IA qui correspond a une couleur et dispose d'une réserve de peinture limitée.

L'objectif est de peindre le plus de surface possible de sa couleur pour obtenir un score le plus élevé possible par rapport à celui de l'adversaire. Le score dépend principalement du nombre de cases contrôlées sur la carte. Le jeu fonctionne au tour par tour et à chaque tour l'IA doit faire quelque chose ou une action ou deux, sinon elle peut être déconnectée pour cause de timeout et sans prendre trop de temps pour réfléchir, sinon on est aussi timeout.

La peinture consomme de la réserve, ce qui oblige l'IA à gérer sa peinture pour éviter de se retrouver à court. Il y a aussi des objets qui apparaissent aléatoirement sur la carte pendant la partie un peu comme des "bonus", comme des pistolets qui permettent de peindre sur les murs ou encore les bidons qui permettent de recharger sa réserve, etc.

La stratégie du joueur repose donc à la fois sur l'optimisation des déplacements, la gestion de la peinture, l'utilisation des objets et le contrôle du territoire, tout en empêchant l'expansion de l'adversaire.

III) Notre stratégie

Nous avons dû établir une stratégie pour avoir une Intelligence Artificielle la plus optimisée possible. Nous avons donc pensé à une méthode pour que notre IA soit efficace. Pour commencer, si on n'a pas beaucoup dans notre réserve de peinture, on va se focaliser sur la récolte de bidons.

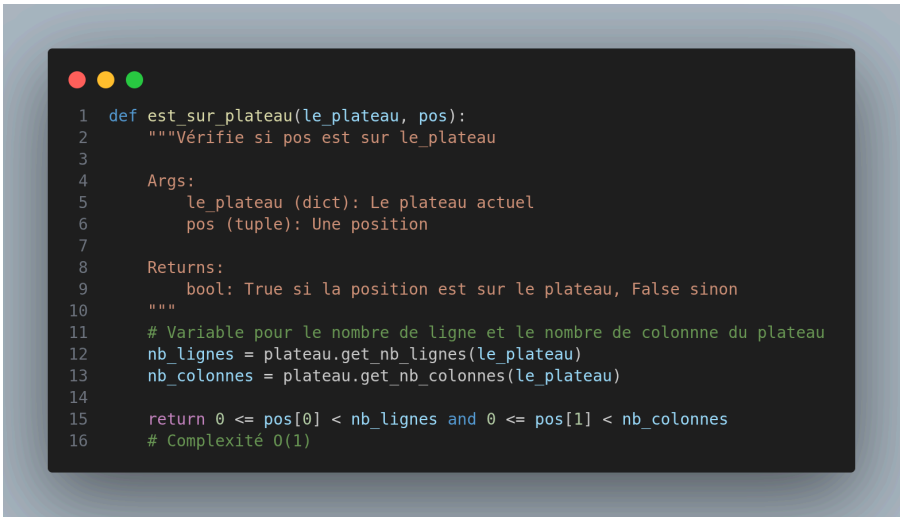
Mais s'il n'y en a pas alors on va essayer de trouver notre surface peinte pour aller dessus et récupérer de la peinture en réserve. Ensuite, on regarde si on a un objet, si c'est le cas on vérifie que c'est un pistolet. Si on a un pistolet alors au lieu d'éviter les murs, là on va se focaliser à tirer sur les murs.

On a voulu implémenter pour la bombe un tir optimal mais nous n'avons pas eu le temps. Après ça, si on n'a pas d'objet, alors on vérifie s'il y a un objet sur la carte, en focalisant surtout les pistolets. Si c'est le cas, alors on s'y dirige. S'il n'y a pas d'objet, il y a deux possibilités, soit c'est le début de la partie, on cherche à s'étendre.

Soit il y a eu plus de 50 tours qui sont passés, donc on va focaliser la personne qui a le plus de points. Si aucune de ces conditions n'est validée, alors on peint juste autour de nous les cases qui ne sont pas de notre couleur.

IV) Les principaux algorithmes

Nous avons implémenté plusieurs types d'algorithmes. Il y a un algorithme de vérification, "est_sur_plateau" nous sert à savoir si une position est bien sur le plateau. Très utile pour ne pas faire crasher notre IA.



```python
def est_sur_plateau(le_plateau, pos):
    """Vérifie si pos est sur le_plateau

    Args:
        le_plateau (dict): Le plateau actuel
        pos (tuple): Une position

    Returns:
        bool: True si la position est sur le plateau, False sinon
    """
    # Variable pour le nombre de ligne et le nombre de colonnne du plateau
    nb_lignes = plateau.get_nb_lignes(le_plateau)
    nb_colonnes = plateau.get_nb_colonnes(le_plateau)

    return 0 <= pos[0] < nb_lignes and 0 <= pos[1] < nb_colonnes
    # Complexité O(1)
```

COMPTE RENDU

Il y a aussi un algorithme pour tirer de façon optimale, “tir_opti”, cet algorithme sert à regarder si là où on se déplace la case est de notre couleur. Si c’est le cas, alors on ne va pas tirer et gâcher de l’encre.

Cependant si ce n’est pas le cas alors on tire directement sur la case pour ne pas perdre de la peinture bêtement et pour agrandir notre surface. On a juste vérifié sur une seule case car nous n’avons pas spécialement eu le temps de modifier l’algorithme pour vérifier 2, 3 ou plus de cases selon notre réserve.

```python
def tir_opti(direction, notre_position, ma_couleur, le_plateau):
    """Permet de donner une direction pour tirer de façon optimal

    Args:
        direction (str): La direction dans laquelle on va
        notre_position (tuple): Notre position actuelle
        ma_couleur (str): Notre couleur
        le_plateau (dict): Le plateau actuel

    Returns:
        str: La direction de la peinture
    """
    # Initialisation des variables
    peinture = 'X'
    ligne, colonne = plateau.INC_DIRECTION[direction]
    nouvelle_pos = (notre_position[0] + ligne, notre_position[1] + colonne)

    # On regarde si la nouvelle position est sur le plateau et si la couleur est différente de la notre
    if est_sur_plateau(le_plateau, nouvelle_pos) and case.get_couleur(plateau.get_case(le_plateau, nouvelle_pos)) != ma_couleur:
        peinture = direction
    return peinture
    # Complexité O(1)
```

Enfin, il y a nos algorithmes de direction / recherche, qui sont le cœur du projet. Ils nous servent à se déplacer vers là où on souhaite, c’est-à-dire se diriger vers un objet, vers notre couleur ou vers la couleur ennemie. Ce sont les fonctions “trouver_direction_objet”, “trouver_direction_bidon”, “trouver_direction_recharge”, “trouver_direction_cible” et “trouver_direction_autre_couleur”. On avait pensé à factoriser toutes ces fonctions qui sont juste des répétitions avec un tout petit changement mais nous n’avons pas eu le temps de la faire. Donc malheureusement, il y a beaucoup de lignes qu’on aurait pu éviter.

```python
def trouver_direction_objet(le_plateau, pos_depart):
    """Permet de trouver le plus court chemin vers objet à partir de pos_depart

    Args:
        le_plateau (dict): Le plateau actuel
        pos_depart (tuple): La position de départ

    Returns:
        tuple: Un tuple de str, int représentant la direction à prendre et l'objet vers lequel on va
    """
```

Nous avons aussi la fonction “directions_murs” qui sert surtout pour le pistolet, cette fonction nous renvoie les directions vers où se trouvent des murs (juste sur les 4 cases voisines à notre position). Pour cette fonction, on aurait aussi pu regarder sur plusieurs cases mais il n’y avait pas forcément le temps.

```python
def directions_murs(le_plateau,pos):
    """Permet de savoir la direction où il y a des murs

    Args:
        le_plateau (dict): Le plateau actuel
        pos (tuple): La position de départ

    Returns:
        str: Les directions où il y a des murs
    """
    # Initialisation des variables
    direction_possibles = ""
    ligne, colonne = pos

    # Pour chaque direction
    for direction, (dirl, dirc) in plateau.INC_DIRECTION.items():

        if direction != 'X': # Si ce n'est pas le X (ne pas bouger)

            # On prend la nouvelle position
            nouvelle_pos = (ligne + dirl, colonne + dirc)

            # On vérifie si elle est dans le plateau pour récupérer la case
            if est_sur_plateau(le_plateau, nouvelle_pos):
                la_case = plateau.get_case(le_plateau, nouvelle_pos)

                if case.est_mur(la_case):
                    direction_possibles+=direction

    return direction_possibles
```

V) L’état de notre travail

Notre Intelligence Artificielle actuelle est plutôt bonne et optimisée. Nous avons pu implémenter toutes les fonctions de base que nous voulions. Notre Intelligence Artificielle se comporte comme nous le souhaitons.

Avec plus de temps, nous aurions pu faire des améliorations, comme par exemple ajouter une distance maximum pour la recherche d’objet. Mais aussi regarder plus d’une case pour tirer de façon encore plus optimale.