

SAE 1.02

Création d'une IA pour Diamants



Compte rendu réalisé par:
Armand Clouzeau Info1 B
Rayan Ben Tanfous Info1 B

Description générale du projet :

Ce projet consistait à développer une IA pour jouer au jeu Diamants que nous avons déjà développé durant la SAE 1.01. Pour à terme faire un tournoi d'IA contre les IA des autres élèves de la promo.

Pour faire cette IA, nous avions à notre disposition un fichier pdf et un dossier avec les différents code. Le fichier PDF nous présentait le projet avec son objectif général, comment lancer une partie, comment fonctionne tours... Ce fichier était en quelque sorte un README. Quant à lui, le dossier contenait 4 IA (IA_à_completer, IA_aleatoire, IA_téméraire, IA_trouillarde), un fichier main.py qui sert à lancer le jeu et un fichier moteur_diamant.py qui est le code du jeu que nous utiliserons.

Nos réflexions, nos analyses et nos idées :

Comme nous devions travailler sur un code qui n'était pas le nôtre, nous avons consacré beaucoup de notre temps à la lecture de ce dernier.

Lors de notre lecture du fichier moteur_diamants.py, nous avons remarqué la présence de class et de self. Ces termes nous étaient inconnus, nous avons donc fait des recherches pour essayer de les comprendre. Nos recherches ne nous ont pas permis de comprendre clairement le fonctionnement des class et de self.

Malgré notre incompréhension de ces notions, nous avons commencé la lecture des différentes IA pour ne pas perdre de temps. Mais lorsque nous avons compris que nous ne pouvions pas commencer notre IA tant que nous n'avons pas compris le fonctionnement de class et self, nous avons repris nos recherches et nous avons discuté avec d'autres groupes qui avaient les mêmes problèmes de compréhension que nous. Grâce à ces discussions, nous avons commencé à comprendre le fonctionnement de ces nouvelles notions. Mais c'est grâce aux cours de Mr. Auger que nous avons vraiment compris clairement le fonctionnement et l'utilité ces deux notions.

Après avoir compris comment utiliser les concepts de "class" et "self", il était nettement plus facile de comprendre le fonctionnement du moteur_diamant et des IA. Nous avons passé toute la première semaine à nous familiariser avec le code.

A la fin de la première semaine, après avoir analyser le fichier PDF et les différents codes, nous avions assimilé les noms de variables choisies par le professeur, où et comment étaient stocké les données. Nous avons porté une attention toute particulière à la class Match et à la fonction tour_de_jeu car nous les utilisons pour faire fonctionner les IA.

Tout ce temps passé à comprendre le fonctionnement du code nous a permis d'être efficace dans la création de notre IA.

Nous avons pensé à utiliser la méthode split sur la class match pour récupérer le nombre de joueurs et à faire la même chose sur la fonction tour pour avoir la carte qui a été piochée et l'état des joueurs. En fonction de la carte piochée, nous la stockons dans des endroits

précis pour ensuite que notre IA puisse jouer en fonction des cartes qui ont déjà été piochées .

Nous avons également compris que pour faire fonctionner une IA, il faut initier des variables dans la fonction `__init__` pour pouvoir stocker des valeurs du jeu pour que l'IA puisse les traiter et prendre certaines décisions dans la fonction `action`.

Nous souhaitions avoir une IA avec beaucoup de conditions pour qu'elle puisse filtrer au maximum et qu'elle fasse les meilleurs choix possible pour la victoire.

Nous avons fait plusieurs versions de notre IA. Nous avons tout d'abord voulu commencer par une IA très basique pour se familiariser avec le fonctionnement et ensuite nous l'avons fait évoluer pour la rendre la plus optimal possible.

Première version de notre IA :

```
class IA_Diamant:  
    def __init__(self, match : str):  
        """génère l'objet de la classe IA_Diamant  
  
        Args:  
            match (str): descriptif de la partie  
        """  
        self.historique_carte = []  
        self.reste = 0  
        self.nb_joueur = match.split("|")[1]  
        self.piège = {"P1" : 3, "P2" : 3, "P3" : 3, "P4" : 3, "P5" : 3}  
        self.nb_relique = 0  
        self.nb_piège = 0  
        self.part = 0  
  
        print("IA SAE reçoit match = '" + match + "'")
```

Nous avons dans un premier temps identifié les données que nous pensions importantes et on a cherché un moyen de récupérer les données efficacement.

Ces différentes données ont été initialisées dans la fonction `__init__`.

Nous avons choisi d'initialiser les données ci dessous :

- un historique de carte dans lequel sera rajoutée la carte qui vient d'être pioché.
- Le reste des rubis
- le nombre de joueur
- les différents pièges et leurs quantités
- le nombre de relique et de piège en jeu
- la part de rubis de chacun

```

27     def table(self, tour : str):
28         """
29             Recup les valeurs importante sur la table.
30
31             Args:
32                 tour(str): descriptif du dernier tour de jeu
33
34             Returns:
35                 self.carte = tour.split("|")[1]
36
37             if self.carte in self.piege:
38                 self.piege[self.carte]-=1
39                 self.nb_piege += 1
40                 self.historique_carte.append(self.carte)
41             elif self.carte == "R":
42                 self.nb_relique += 1
43                 self.historique_carte.append(self.carte)
44             else:
45                 self.reste += int(self.carte) % int(self.nb_joueur)
46                 self.part += int(self.carte) // int(self.nb_joueur)
47                 self.historique_carte.append(self.carte)
48
49
50     def action(self, tour : str) -> str:
51         """Appelé à chaque décision du joueur IA
52
53             Args:
54                 tour (str): descriptif du dernier tour de jeu
55
56             Returns:
57                 str: 'X' ou 'R'
58
59             #####
60             #ICI il FAUT compléter !      #
61             #####
62             self.table(tour)
63
64
65
66             if self.nb_piege == 0 and self.part < 10 :
67                 return "X"
68
69             elif self.nb_piege > 2 and self.part < 5:
70                 return "R"
71             else:
72                 return "R"
73                 print("    IA SAE reçoit tour = '" + tour + "'")
74
75
76     def fin_de_manche(self, raison : str, dernier_tour : str) -> None:
77         """Appelé à chaque fin de manche

```

Dès le début, on a créé la fonction `def table` en utilisant les données de tour pour les récupérer et les stocker dans des variables que nous avions créé dans la fonction `__init__`. Cette fonction nous permet de donner les données importantes à l'IA pour qu'elle puisse les traiter et en déduire des choix. Les premières données que nous voulons récupérer sont la valeur de la carte piocher, quand la carte est piochée nous ajoutons 1 au compteur pour savoir combien de pièges sont tombés au cours de la manche. Quand une carte relique est sortie nous ajoutons 1 au compteur du nombre de reliques piochées. Quand la carte piochée n'est pas l'un des types dit précédemment nous en déduisons que c'est une carte tresor. Si c'est une carte trésor nous calculons le reste et les parts que les joueurs reçoivent. Toutes les cartes sont inscrites dans une liste d'historiques de cartes. Nous avions pour objectif de stocker un maximum de données qui seraient possiblement intéressantes à utiliser pour notre IA.

Ensuite, nous avons fait des actions vraiment simples. Ici une fois les fonctions `__init__` et `table` configurer, la fonction `action` va appeler la fonction `table` qui nous permettra d'utiliser les variable que nous avons stocké en amont. En fonction de la carte qui sera piochée et du nombre de rubis dont le joueur dispose, le joueur va rester dans la mine ou rentrer au campement.

Deuxième version de notre IA:

Pour cette deuxième version, nous avons gardé les mêmes fonctions `__init__` et `table`. Nous avons simplement modifié les actions de l'IA en mettant des conditions plus précises, notamment en ajoutant la valeur du reste dans les conditions.

```
def action(self, tour : str) -> str:  
    """Appelé à chaque décision du joueur IA  
  
    Args:  
        tour (str): descriptif du dernier tour de jeu  
  
    Returns:  
        str: 'X' ou 'R'  
    """  
  
    #####  
    #ICI il FAUT compléter !      #  
    #####  
    self.table(tour)  
  
  
    if self.nb_piege <= 2:  
        if self.part >=7 or self.reste >=7:  
            return "R"  
        else:  
            return "X"  
  
    elif self.nb_piege > 2:  
        if self.part < 4 or self.reste <=6:  
            return "X"  
        else:  
            return "R"  
  
    print("    IA SAE reçoit tour = '" + tour + "'")
```

Notre idée était très simple, nous voulions maximiser nos gains de rubis en jouant de manière stratégique avec seulement des conditions précises. Nous nous sommes dit que s'il y avait plusieurs cartes pièges, nous devions évaluer si les gains potentiels étaient suffisamment importants pour continuer à jouer. Plus précisément, si l'IA peut gagner minimum 7 rubis pour sa propre part ou si le reste en jeu était de 7 rubis, nous décidons de rentrer, sinon nous continuons à jouer. Cette stratégie nous permet de prendre des risques lorsque les gains potentiels sont importants, tout en étant prudents lorsque les risques sont élevés. Cela nous permet de maximiser nos chances de gagner tout en minimisant les risques de perte.

Avec plusieurs groupes nous voulions faire combattre nos IA. Ceci nous a amenés à la modification du fichier main en ajoutant une fonction. Cette fonction nous permet de comptabiliser un point à chaque fois qu'un joueur gagne une partie. Puis nous voulions avoir un aspect vraiment précis de la probabilité de victoire des IA c'est pourquoi nous avons créé une boucle qui réalisera le nombre de parties que vous voulez simuler. Dans le cas ci-dessus nous avons réalisé 100 parties et notre IA_v2 a gagné 82 fois.

```

30     victoire= [0,0,0,0] #initialise les victoires de chaque joueur à 0
31     def calcul_winrate(victoire, score):
32         """
33             Fonction qui permet d'attribuer un point à celui qui a le plus de score.
34         """
35         gagnant = 0
36         gagnant = score.index(max(score))
37         victoire[gagnant] += 1
38
39         return victoire
40
41     if __name__ == '__main__':
42         for i in range(100): #Boucle pour faire 100 partie.
43             score = partie_diamant(5,['IA_aleatoire', 'IA_v2','IA_temeraire', 'IA_trouillarde'])
44             print(calcul_winrate(victoire,score))
45
PROBLÈMES SORTIE TERMINAL JUPYTER CONSOLE DE DÉBOGAGE
IA temeraire reçoit en fin de jeu scores = '27,67,0,7'
IA trouillarde reçoit en fin de jeu scores = '27,67,0,7'
[17, 82, 0, 1]
PS E:\SAE_2_IA_Diamants_v2> []

```

Notre IA n'étant pas basée sur des prises de décisions imprévisibles et sur le hasard comme le spécifie l'énoncé de la SAE, nous avons donc dû modifier le code de notre IA. Ce qui nous amène à développer notre version finale de notre IA.

La version finale de notre code

Dans un premier temps, nous avons gardé le principe que nous avions élaboré qui était de voir le risque. Nous avons mis ça en place grâce à une variable de risque qui pourra être augmentée ou diminuée en regardant combien de cartes pièges ont été piochées, combien de rubis ont été piochés pendant la manche, combien de joueur il reste dans la manche ou encore si une relique est en jeu pendant la manche.

Grâce à la récupération de données que nous avions mis en place dans la fonction table, nous n'avions plus besoin de nous soucier de cela durant l'ajout du taux de risque et de l'aléatoire dans notre IA. Nous avons simplement ajouté une nouvelle variable choix_joueur et total qui permet de voir le choix du joueur du tour et le total de rubis que l'IA va pouvoir récupérer s'il sort. Nous avons donc pu nous focaliser sur la réalisation de notre valeur de risque. Puis implémenter un choix aléatoire qui rend notre IA imprévisible. Bien que notre IA soit imprévisible, elle n'en est pas moins intelligente. Son intelligence est basé sur des données récupérées au fur et à mesure que la manche avance.

```

def __init__(self, match : str):
    """génère l'objet de la classe IA_Diamant
    Args:
        match (str): descriptif de la partie
    """
    self.historique_carte = [] #Liste des cartes qui tirer pendant une manche.
    self.reste = 0 #Initialisation du reste
    self.nb_joueur = match.split("|")[1] #Recup de le nombre de joueur de la manche en str
    self.piege = {"P1" : 3, "P2" : 3, "P3" : 3, "P4" : 3, "P5" : 3} #Dictionnaire qui regroupe les pièges et leur quantité
    self.nb_relique = 0 # nombre relique mis en jeu dans la manche
    self.nb_piege = 0 # nombre de piege tirer dans la manche
    self.part = 0 # nombre de rubi que peut récup l'ia
    self.total = 0 # Total des rubis que l'ia peut recuper dans la manche(ça propre part + le reste)
    self.total_rubis = 0 # Total des rubis qui sont en jeu dans la manche
    print("IA SAE reçoit match = '" + match + "'")

```

Dans la fonction `__init__` nous avons donc ajouté 2 nouvelles valeurs et une liste que nous avons donc dû initialiser dans la fonction. Elles nous permettront de stocker des valeurs qui seront importantes pour notre nouvelle manière de mettre en place une jauge de risque.

- *Total* va nous permettre d'additionner dans une variable la part et le reste que l'IA pourra gagner.
- *Total_rubis* est une variable qui comptabilise le nombre de rubis qui est sur la table pendant la manche.

```

def table(self, tour : str):
    """
    Recup les valeurs importante sur la table.
    Args:
        tour(str): descriptif du dernier tour de jeu
    """
    self.carte = tour.split("|")[1] # recup la valeur de la carte
    self.joueur_partant = 0 # initialise la variable joueur partant
    self.choix_joueur= tour.split("|",2) # Dans un premier temps separe en deux à partir du caractere "|" et le stock dans la variable choix_joueur
    self.choix_joueur.pop(1) # Supprime la valeur à l'indice 1 car il representait la carte du tour
    self.choix_joueur=self.choix_joueur[0].split(",") # Separe la chaine de caractere qui se trouve à l'indice 0 à chaque virgule
    for i in self.choix_joueur: # boucle pour regarder chaque choix de joueur
        if i in ['N', 'R']: # si le joueur est rentré on ajoute un au compteur joueur_partant
            self.joueur_partant +=1
    if self.carte in self.piege: #Si la carte tirer est une carte piege
        self.nb_piege += 1 # ajoute un au compteur du nombre de piege tirer
        self.historique_carte.append(self.carte) #Ajoute la carte dans l'historique
    elif self.carte == "R": # Si la carte est une relique
        self.nb_relique += 1 #ajoute 1 au nombre de carte relique tirer.
        self.historique_carte.append(self.carte) #Ajoute la carte dans l'historique
    else:
        self.reste += int(self.carte) % (int(self.nb_joueur)-self.joueur_partant) # Calcule pour ajouter le reste que peux avoir l'IA avec les joueurs encore en jeu
        self.part += int(self.carte) // (int(self.nb_joueur)-self.joueur_partant) # Calcule la part que peut avoir l'IA avec les joueurs encore en jeu
        self.total_rubis += int(self.carte) # ajoute les rubis à chaque tour pour avoir le total de rubis de la manche
        self.historique_carte.append(self.carte) #Ajoute la carte dans l'historique
    self.total += self.part + self.reste # Total que peux recevoir l'IA en comptant la part + le reste

```

Dans la fonction `table` nous avons ajouté deux variables qui nous permettront de stocker de nouvelle valeur qui seront utilisés pour le calcul du risque.

- *choix_joueur* est une liste qui stock le choix de chaque joueur. Elle nous permettra d'augmenter le compteur de *joueur_partant* grâce à la boucle `for i in self.choix_joueur` qui permettra de parcourir la liste nous ajoutons une condition si le status est N ou R alors nous additionnons 1 au compteur des *joueur_partant*.

- *joueur_partant* est une variable qui compte le nombre de joueurs qui ont arrêté de jouer la manche. Cette variable nous permettra de les retirer du reste et des parts cela nous donnera un nombre beaucoup plus précis à notre IA pour sa prise de décision dans la suite du programme.

```

def action(self, tour : str) -> str:
    """Appelé à chaque décision du joueur IA
    Args:
        tour (str): descriptif du dernier tour de jeu
    Returns:
        str: 'X' ou 'R'
    """
    self.table(tour)
    risque = 0 #initialise le risque à 0
    risque += self.nb_piege*33 #multiplie le nombre de piege par 33. Si 3 cartes tirer le risque sera à 99% donc dangereux.
    risque += (self.total_rubis - 35)*3 #regarde le total de rubis en jeu pendant la manche soustrait par une moyenne de
    #rubis total d'une manche et multiplier par 3.Cela permet de baisser ou non le risque.
    risque += (int(self.nb_joueur)-self.joueur_partant)//100 # Permet de rajouter du risque moins il a de joueur restant.
    #Regarde si une carte relique a été piocher et si le nombre de rubis total est inférieur ou supérieur à 7
    #Cette suite de condition permet d'ajouter du risque en fonction de plusieurs situations.
    #Plus le risque est élevé plus il a la proba de sortir
    if self.carte == "R" and self.total_rubis < 7:
        if risque > 66:
            risque += self.nb_relique * 25
        else:
            risque += self.nb_relique * 5
    elif self.carte == "R":
        if risque > 50:
            risque += self.nb_relique * 35
        else:
            risque += self.nb_relique * 20
    # Tire un nombre entre 0 et 100.
    # Si le nombre est inférieur au risque alors l'IA rentre sinon il reste
    if random.randint(0,100) < risque:
        return "R"
    else:
        return "X"
    print("    IA SAE reçoit tour = '" + tour + "'")

```

Notre fonction *action* est celle qui a le plus changé comparé à nos anciennes versions. C'est dans cette fonction que nous avons ajouté notre variable *risque*. Celle-ci est initialisée dès le début à zéro et sera augmentée ou diminuée grâce à aux valeurs des variables *nb_piege*, *total_rubis* mais aussi si le nombre de joueur restant diminue ou si une carte relique est tirée. Nous avons procédé avec des coefficients qui augmentent ou diminuent le risque comme par exemple avec le nombre de cartes pièges. Nous avons choisi comme coefficient 33 car si trois cartes pièges sont piochées, nous estimons que le jeu devient vraiment dangereux. Le risque sera minimum à 99 ce qui est vraiment beaucoup. Une fois que le risque a été fixé nous avons une condition qui procède à un random d'un chiffre entre 0 et 100 et si cette valeur est inférieure à notre valeur de risque alors nous rentrons sinon l'IA continue l'exploration. Chaque coefficient que nous avons choisi pour chaque valeur de risque est une vision des choix que nous avions pris en binôme en jouant au jeu. En faisant plusieurs tests nous en avons conclu que nos choix étaient justifiés et efficaces.

```

def fin_de_manche(self, raison : str, dernier_tour : str) -> None:
    """Appelé à chaque fin de manche
    Args:
        raison (str): 'R' si tout le monde est un piège ou "P1","P2",... si un piège a été déclenché
        dernier_tour (str): descriptif du dernier tour de la manche
    """
    print(" IA SAE reçoit en fin de manche raison = '" + raison + "' et dernier_tour = '" + dernier_tour + "' ")
    # Réinitialise toutes les listes et variables que nous devons réutiliser
    # pour le bon fonctionnement de la prochaine manche
    self.nb_relique = 0
    self.nb_piege = 0
    self.part = 0
    self.reste = 0
    self.historique_carte = []
    self.total_rubis=0

```

La fonction `fin_de_manche` nous permet de réinitialiser toutes les variables et la liste que nous utilisons pendant une manche. Au cours du projet, nous avions totalement oublié de réinitialiser les variables ce qui nous avait posé des problèmes car les valeurs étaient totalement erronées et inutilisables.

Conclusion du projet:

Pour conclure ce projet de création d'IA pour le jeu diamant s'est avéré être très enrichissant d'un point de vu enseignement car nous avons pu voir pour la première fois certaines notions comme par exemple les class ou encore `self`. Nous avons aussi pu découvrir les méthodes spéciales comme `__init__`. Ce projet nous a donc permis d'enrichir nos connaissances tout en les appliquant dans un domaine totalement nouveau qui est la création d'une IA. La mise en pratique de ces nouveaux concepts tout au long du projet ont permis une meilleure compréhension de la programmation orientée objet en python. C'est un aspect que nous n'aurons peut-être pas explorer mais qui c'est trouvé très ludique et intéressant. Ce projet nous a permis de découvrir encore plus l'étendu des possibilités que nous apporte le langage python. Notre avis personnel sur cette SAE dans l'ensemble est que nous l'avons trouvé plus intéressante que la précédente.