SAÉ - IMPLEMENTATION D'IA

RAPPORT

Ewen GILBERT, Mehmet YUKSEL, Pierre JAUFFRES

Janvier 2023

UVSQ université PARIS-SACLAY
IUT de Vélizy-Rambouillet
CAMPUS DE VÉLIZY-VILLACOUBLAY

Introduction - DIAMANT, un jeu si compliqué ?

Diamant est un jeu assez complexe dans son fonctionnement, il prend en compte beaucoup de règles différentes et introduit des règles subtiles qu'il faut prendre en compte en tant que joueur. Un grand principe de ce jeu : **prendre des risques !**

Mais, prendre des risques, c'est bien, mais il ne s'agit pas de prendre des risques inconsidérés et de terminer dernier de la partie ! Il faut prendre des risques, mais pas trop ! Il y a donc plein de paramètres à prendre en compte.

Tout d'abord, il faut prendre en compte les cartes qui peuvent être tirées lors d'une manche. Effectivement, dans le jeu, il y a trois grand types de cartes : les trésors, qui représentent une somme de diamants à partager entre les joueurs en jeu, les dangers, qui, si rassemblés avec leur alter égau, font sortir les joueurs du temple et mettent fin à la manche, et les reliques qui ont une valeur en diamants et qui peuvent être récupérés par un et un seul joueur sortant.

Il faut donc prendre en compte ces informations, à commencer par les trésors. En effet, il existe alors plusieurs valeurs de trésor, et ces valeurs sont partagées entre les joueurs présents dans la partie. En tant que joueur, il faut prendre ce paramètre en compte, car moins il y a de joueurs présents, plus la récompense en diamants est élevée !

Ensuite, il faut prendre en compte les dangers, car ils représentent un réel risque pour le joueur. Car effectivement, si deux objets identiques sont tirés, tous les joueurs actuellement en jeu sortent du temple et perdent leurs diamants qui ne sont pas stockés ! Le joueur doit alors prendre des risques, mais pas abusivement pour ne pas perdre la partie.

Enfin, il faut également prendre en compte les reliques, car elles peuvent être récupérées seulement quand un et un seul joueur sort du temple. Il faut donc adapter son timing pour sortir seul, sinon, la relique est perdue !

Ces paramètres sont les principaux à prendre en compte en tant que joueur, mais comme on a pu le remarquer, il faut également prendre en compte le positionnement dans le classement ou les diamants laissés au sol. En effet, si notre joueur est en bas du classement, il peut se permettre de prendre plus de risques que si celui-ci est en haut du classement, et peut sortir plus tôt si le nombre de diamants au sol est avantageux.

En tant que joueur, il faut prendre en compte toutes ces informations... Mais dans cette SAÉ, **nous devons créer une IA !** Il va alors falloir prendre tous ces paramètres, et les intégrer dans notre IA, pour que celle-ci puisse prendre les meilleures décisions au meilleur moment !

I - LES BASES DE L'IA : INTERPRÉTER LE MOTEUR

1 - UTILISATION DE LA POO

Pour interpréter les IA, le moteur de jeu utilise la **Programmation Orientée Objet** (POO) qui permet alors de créer des objets, à partir de classes définies. Dans notre cas, nous avons donc une classe *`IA_a_completer`* qui va servir pour stocker toutes les fonctions et variables de notre IA dans une seule entité qui sera donc notre objet.

Dans notre IA, nous n'avons que peu d'informations sur la partie de la part du moteur. Il faut alors trouver un moyen pour que notre IA puisse avoir toutes les informations nécessaires pour pouvoir prendre une décision sur la suite de la partie.

Comme nous allons le voir un peu plus tard, le moteur de Diamant nous envoie que de petites bribes d'informations qui sont essentielles dans la compréhension du jeu par notre IA. Or, pour avoir une IA efficace qui gagne à presque tous les coups, nous avons besoin de plus d'informations que le nombre de joueurs dans la partie et la dernière carte tirée... Il faut simuler l'entièreté de la trame de la partie à l'intérieur de notre IA, celle-ci doit alors interpréter toutes les informations pour en tirer le meilleur.

Nous allons donc commencer par créer dans notre classe *`IA_a_completer`*, une classe *`Joueur`* qui va alors nous permettre de créer un objet différent par joueur, pour pouvoir stocker leur statut, nombre de diamants, ou le fait qu'ils soient notre IA ou non. Cette classe est primordiale dans la simulation de notre jeu, car pour être efficace, notre IA doit connaître déjà son nombre de diamants à elle, mais aussi celui de ses adversaires, pour prendre de meilleures décisions.

```
class Joueur:
  def __init__(self, est_ia):
    self.etat = 'X'
    self.diamant_poche = 0
    self.diamant_campement = 0
    self.est_ia = est_ia
```

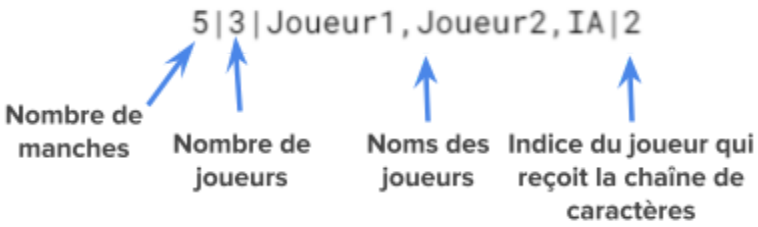
Dans notre classe, les variables *`diamant_poche`* et *`diamant_campement`* comptent alors les diamants que les joueurs possèdent, et permettent alors grâce à une fonction de déterminer un classement des joueurs, et de positionner notre IA à l'intérieur grâce à notre variable booléenne *`est_ia`* qui est en *`True`* si le joueur concerné se trouve être notre IA. La variable *`etat`*, elle sert de stockage pour la décision de chaque joueur, et permet alors d'interpréter tour par tour la meilleure décision à prendre pour notre IA.

Nous allons également récupérer un historique des cartes tirées pour nous permettre de socker et de compter le nombre de pièges, pour savoir le niveau de risque pour notre IA, stocker et partager le nombre de diamants, pour connaître le classement en temps réel et le potentiel de diamants que notre IA peut ramasser en sortant du temple, mais aussi les reliques, en stockant directement leur valeur pour l'ajouter à celle des diamants restés au sol.

2 - LIRE LES INFORMATIONS DU MOTEUR

Tout d'abord, pour créer notre IA, il faut déjà interpréter les données envoyées par le moteur. Ici, le moteur du jeu envoie à notre IA 3 chaînes de caractères différentes au cours de la partie. On va ici se concentrer sur les deux premières.

La première chaîne de caractères est celle qui correspond à l'initialisation de la partie, et à l'initialisation de notre IA. Cette chaîne se nommant "`match`" est celle qui va donner à notre IA les premières informations importantes. Elle est composée de 4 données :



Ces données servent d'initialisation de toutes les variables primordiales pour l'IA. Ce qui va nous intéresser ici va surtout être le nombre de joueurs, mais aussi l'indice de l'IA dans la liste des joueurs utilisé par le moteur.

Le nombre de joueurs va nous aider à créer une variable *`nbjoueurs`* et *`nbjoueurs_plateau`*. Comme on peut le voir sur l'image à droite, nous récupérons le nombre de joueurs dans la chaîne en cherchant de caractère 'I' qui sert de séparateur dans la chaîne, ensuite, on récupère l'information dans *`nbjoueurs`*

```
for i in range(len(match)):
  if match[i] == '|':
    self.nbjoueurs = int(match[i + 1])
    self.nbjoueurs_plateau = self.nbjoueurs
    break
```

La première va servir de base pour tous les calculs et les autres fonctions comme celle de classement. La seconde va comptabiliser le nombre de joueurs présents dans le temple pour les calculs de partage de diamants par exemple, avec les fonctions *`partage_diamants()`* et *`partage_diamants_fin()`*.

```
self.id_mon_ia = int(match[-1])

for i in range(self.nbjoueurs):
  if i == self.id_mon_ia:
    self.joueurs.append(Joueur(True))
  else:
    self.joueurs.append(Joueur(False))
```

C'est aussi avec *nbjoueurs* que nous allons utiliser une boucle pour créer les objets des joueurs grâce à la classe *Joueur* avec comme paramètre le booléen qui nous dit si notre objet doit être défini comme notre IA ou non avec l'indice du joueur donné dans le dernier chiffre de la chaîne de caractères. On va donc, à l'aide d'une boucle, créer les joueurs avec le paramètre en *False*, jusqu'au moment où la boucle va atteindre la valeur de la variable *id_mon_ia* où on va créer un objet avec comme paramètre *True* pour définir le joueur qui est considéré comme notre IA par le moteur.

La seconde chaîne de caractères envoyée par le moteur est reçue entre chaque tour de la partie par notre IA, elle contient deux informations importantes : La décision des joueurs, qui est une suite de lettres séparées par des virgules, et la dernière carte tirée, définie par les caractères positionnés après "|".

La décision de chaque joueur s'illustre par une lettre parmis 3 lettres avec une signification précise, elle est lue par la fonction `lecture_choix_joueurs(choix)` :

Pour interpréter les données, nous allons donc créer une liste *choix_joueurs* qui va récupérer les lettres, et les séparer grâce aux virgules en différentes cases d'une même liste. Chacune des trois lettres a une signification :

- **"R" pour "Retrer"** : Le joueur décide de rentrer au campement et de quitter le temple, il sauvegarde alors les diamants présents dans la poche et récupère/partage les trésors éventuels laissés au sol. Si celui-ci est le seul à sortir, et qu'une relique est au sol, il la récupère. On éxecute la fonction `partage_diamants_fin()` qui va contrôler la situation pour distribuer le butin comme le fait le moteur.
- **"X" pour "Explorer"** : Le joueur décide de continuer d'explorer et de ne pas sauvegarder les diamants qu'il a sur soi. Il passe au prochain tour.
- **"N" pour "Néant"** : Le joueur n'est déjà plus dans le temple, l'état "Néant" ne va pas changer la trame de la partie.

Enfin, nous allons récupérer les caractères qui définissent la dernière carte tirée, puis nous allons éxecuter la fonction `comptabilise_carte(carte)` qui va prendre en paramètre ces caractères. C'est ici que notre IA va pouvoir prendre ses premières décisions.

3 - INTERPRÉTER, LA BASE DES DÉCISIONS

Maintenant que nous savons interpréter les informations données par le moteur, que tout est en place au niveau de la récupération de données, nous allons commencer à prendre des informations pour que notre IA puisse **prendre des décisions basées sur les chances de perdre ou non**. Dans cette logique, nous devons alors stocker et calculer des probabilités basées sur les cartes piochées pendant la partie.

Comme dit plus tôt, nous avons déjà stocké notre carte dans une liste `historique` qui va garder en mémoire toutes les cartes qui ont été piochées lors de la manche. Ensuite, avec notre carte, nous allons exécuter la fonction `compatbilise_carte(carte)` qui va compter trois conditions, qui vont déterminer le type de la carte. Si *carte* est un nombre, on considérera que c'est un trésor, si *carte* est la lettre "R", on considérera que c'est une relique, et si *carte* commence par la lettre "P", on considérera que c'est un piège.

```
def comptabilise_carte(self, carte):
  self.historique.append(carte)
  try:
    carte = int(carte)
  except:
    pass
  if type(carte) == int:
    self.tresor_partage = carte // self.nbjoueurs_plateau
    self.tresor_sol += carte % self.nbjoueurs_plateau
    self.partage_diamants()
  elif carte[0] == 'P':
    self.liste_pieges[carte] += 1
    if self.liste_pieges[carte] == 2:
      self.pieges_sortie[carte] -= 1
  elif carte == 'R':
    self.nbreliquesdecouvertes += 1
    self.valreliquesol += self.valeurs_reliques[self.nbreliquesdecouvertes -1]
    self.relique_en_jeu -= 1
```

Le premier facteur de décision serait alors de savoir la probabilité qu'un piège tombe au prochain tour. Pour celà, nous créons un dictionnaire `liste_pieges` contenant chaque piège en clé, et le nombre d'occurences qui ont été piochées en valeur. On va donc, à chaque fois qu'un piège apparaît, incrémenter 1 au piège correspondant.

En dessous, sur l'image, on peut voir qu'il y a une condition qui regarde, si le piège qui vient d'être tiré a déjà été tiré. En principe, quand ce cas arrive, la manche se termine car on a deux pièges du même type, mais notre IA va enregistrer cette donnée dans la fonction `pieges_sortie(carte)` sur laquelle nous nous concentrerons un peu plus tard. Cette fonction est une base dans la prise de décisions de notre IA, et la comptabilisation des pièges est la chose la plus influente dans la décision de sortir ou non du temple.

Ensuite, nous interprétons également le nombre de diamants et de reliques au sol, car même s'il n'y a pas de danger, si le butin est intéressant pour notre IA, elle peut choisir de sortir. Nous regarderons alors plus tard les valeurs de `tresor_sol` et `valreliquesol` qui déterminent la valeur du butin au sol. La variable `tresor_partage` sera importante pour obtenir le nombre de diamants possédés par chaque joueur, qui permettra plus tard de générer un classement qui déterminera aussi la prise de risque que peut se permettre notre IA.

II - COMMENT GAGNER LA PARTIE ?

1 - DE L'ALÉATOIRE... MAIS PAS TROP

Voilà, maintenant, notre IA est capable de comprendre le jeu, elle peut lire les décisions des autres, peut lire la dernière carte, en résumé, elle peut simuler les données importantes de la partie. Mais maintenant, elle doit pouvoir interpréter toutes ces informations, et prendre une décision... Doit elle rester dans le temple sans trop de risque, rester mais prendre de gros risques, ou rentrer au campement pour sauvegarder son butin ? C'est là que les choses sérieuses commencent.

Pour qu'elle puisse prendre des décisions, notre IA va avoir besoin d'une fonction de choix. A partir d'une variable nommée ***facteurdesortie***, qui représente un pourcentage, l'IA va tirer un nombre entre 1 et 100 grâce au module *random* et va regarder si ce nombre est inférieur à ce facteur. Si c'est le cas, notre IA va renvoyer au moteur "R", pour signifier son désir de rentrer au campement, sinon elle renvoie "X" pour continuer la partie.

Ce facteur est un **pourcentage qui va déterminer les chances de sortir de notre IA**, elle va varier selon plusieurs facteurs. Son principal facteur est le pourcentage de chances de perdre à la prochaine manche, qui va être calculé à l'aide du dictionnaire *liste_pieges*. On va donc calculer la probabilité qu'un piège identique à un des pièges présents dans l'historique puisse sortir, pour déterminer le pourcentage de chance pour notre IA de perdre la manche au prochain tour.

```
def risque_piege(self):

  piege_probable = 0
  for k in self.pieges_sortie.keys():
    piege_probable += (self.pieges_sortie[k] - 1) * self.liste_pieges[k]

  self.facteurdesortie = (piege_probable / self.carte_dans_le_deck) * 100
```

Pour que l'IA puisse calculer ce facteur, on va avoir besoin de plusieurs fonctions. Pour commencer, on a besoin d'une fonction qui calcule la probabilité qu'un piège réapparaisse, pour cela on utilise la fonction *risque_piege()* qui va modifier la valeur de *facteurdesortie* pour qu'elle soit égale au pourcentage de chance qu'un piège réapparaisse. Dans cette fonction, nous allons initialiser une variable *piege_probable* à 0 qui va représenter le nombre total de pièges sur lequel on peut retomber. On va ensuite, à l'aide d'une boucle, parcourir la liste *pieges_sortie* qui répertorie les pièges présents dans la pioche de la manche. En parcourant cette liste, on va ajouter à *piege_probable* la chance de tomber sur chacun des 5 pièges du jeu. Une fois ces chances additionnées, on va diviser ce nombre par le nombre de cartes dans la pioche, et le multiplier par 100 pour créer la probabilité de perdre au prochain tour en pourcentage.

2 - DIFFÉRENTS MOYENS DE VARIATION

On arrive maintenant à calculer le facteur de sortie grâce au risque de retomber sur un piège, mais il faut aussi prendre en compte d'autres paramètres comme le classement ou les diamants.

Pour le classement c'est plutôt simple : on dit que plus le joueur est bas, plus il prend de risques, car il n'a rien à perdre. Il faut aussi tenir compte de l'écart du nombre de diamants avec les autres joueurs, car plus on a d'écart dans les scores avec le joueur qui est devant nous, plus il faut prendre de risques. Alors qu'à l'inverse plus on a d'écart avec le joueur qui est en dessous de nous dans les scores, moins on doit prendre de risques, car celui-ci a moins de chances de nous rattraper. Il ne nous reste plus qu'à mettre tout ça dans une fonction que l'on va appeler `ecart_diamants` qui va modifier une liste `ecart` de deux éléments, où le premier est l'écart avec le joueur juste derrière notre IA dans les scores et où le second est l'écart avec le joueur juste devant notre IA.

```
def ecart_diamants(self):

  if self.indice_ia == 0:
    self.ecart[0] = 0 #s'il n'y a personne apres il n'y a pas de diamants d'avance
    self.ecart[1] = self.classement_diamants[self.indice_ia][0] - self.classement_diamants[self.indice_ia+1][0]
  elif self.indice_ia == 3:
    self.ecart[0] = self.classement_diamants[self.indice_ia][0] - self.classement_diamants[self.indice_ia-1][0]
    self.ecart[1] = 0 #s'il n'y a personne avant il n'y a pas de diamants de retard
  else:
    self.ecart[0] = self.classement_diamants[self.indice_ia][0] - self.classement_diamants[self.indice_ia-1][0]
    self.ecart[1] = self.classement_diamants[self.indice_ia][0] - self.classement_diamants[self.indice_ia+1][0]
```

Il ne nous reste plus qu'à faire évoluer les chances en fonction du nombre de diamants au sol. Pour simplifier on va tout rassembler dans une même fonction qui va modifier `facteurdesortie` avec les nouveaux facteurs que l'on va appeler `set_coeffs` puis on va faire un fonction qui si on génère un nombre aléatoire entre 0 et 100 et s'il est inférieur à `facteurdesortie` renvoi True donc l'IA s'arrête.

```
  def set_coeffs(self,boost = 0):

    self.ecart_diamants()
    plus = self.indice_ia * 10 + self.ecart[1]*0.5 + (self.tresor_sol/self.nbjoueurs_plateau) - self.ecart[0] * 0.75
    self.facteurdesortie += plus

  def doit_continuer(self):

    self.risque_piege()
    self.set_coeffs()
    r = random.randint(0,100)
    if self.joueurs[self.id_mon_ia].diamant_poche == 0 or self.tour == 0:
      return False
    return r <= self.facteurdesortie
```

3 - DES TESTS EN MASSE

Pour affiner notre IA, une fois toutes les fonctions de simulation et de décision terminées, il fallait alors faire quelques petits changements pour que celle-ci soit la plus optimisée possible. Bien entendu, il reste une part d'aléatoire, mais en changeant quelques chiffres, nous pouvions obtenir de meilleures performances.

Pour faire ces changements, il fallait faire des tests, beaucoup de tests. Des tests en masse qui permettent de mesurer de grands volumes de données. Ce qui nous interessait ici était alors de savoir **quelle IA réalisait le plus de victoires**.

Les changements prennent forme avec des listes de coefficients multiplicateurs, appliqués sur différents calculs, comme ceux du facteur de sortie ou encore ceux de la mesure de l'écart. Ces petits paramètres, bien qu'à première vue insignifiants, ont une grande influence dans le calcul du choix de l'IA. Chaque IA avait donc son duo de coefficients, ayant une influence sur ses performances.

Pour réaliser ces tests, nous avions alors apporté quelques modifications temporaires au moteur et à notre IA pour intégrer ces facteurs. Nous avons ensuite créé une boucle qui réalise 100 parties à la suite, pour ensuite récupérer celle des quatre IA qui aura réalisé le meilleur score. Cela nous a donc permis d'affiner nos calculs pour faire, selon nous, une IA qui réalise de belles performances.

```
if __name__ == '__main__':
  scoremax = 0
  for i in range(100):
    classement = partie_diamant(5,['IA_a_completer',
'IA_a_completer','IA_a_completer', 'IA_a_completer'],coeffs)
    max = classement[0]
    indice = 0
    for j in range(len(classement)):
        if classement[j] > max:
            max = classement[j]
            indice = j
    if max > scoremax:
      scoremax = max
    nbvictoires[indice] += 1


print(nbvictoires)
print(scoremax)
```

III - UN PLAN QUI NE SE DÉROULE PAS SANS ACCROC

1 - LA POO, UN MONDE NOUVEAU

Cette SAE à été programmée en POO. Pour cela nous devions tout programmer avec cette méthode. Au début nous avions eu quelques difficultés car la POO à été vue brièvement au lycée ou jamais vue pour certains. C'est pour cela que nous avons dû revoir et apprendre les bases de la POO pour réussir cette SAE et faire une IA efficace. C'était un tout nouveau défi à relever.

En effet, dans notre premier semestre de BUT, nous n'avions fait que très peu de POO, et nous savions que c'était tout de même un concept importanour le futur, pour apprendre des langages comme Java. Il a donc fallu découvrir cette dimension du programme, qui était une découverte pour certains d'entre nous. Nous nous sommes alors entraidés, pour que tout le monde comprenne bien le concept et pour le programmer efficacement.

Mais cet apprentissage n'était pas sans aléas. Beaucoup d'erreurs étaient présentes dans notre code au début, comme des variables non déclarées en self, ou des fonctions mal appelées. Mais finalement, grâce au groupe, nous avons réussi à passer cette difficulté.

2 - LES PIÈGES, UNE HISTOIRE ÉPINEUSE

Pour pouvoir faire fonctionner notre IA nous avions eu besoin de connaître le nombre de pièges distincts apparus et le nombre de pièges restants, car celui-ci diminuait si jamais la partie se finissait par un piège.

Pour cela on a utilisé un dictionnaire *liste_pieges* qui

```
self.liste_pieges = {'P1': 0, 'P2': 0, 'P3': 0, 'P4': 0, 'P5': 0}
self.pieges_sortie = {'P1': 3, 'P2': 3, 'P3': 3, 'P4': 3, 'P5': 3}
```

compte le nombre de pièges apparus pendant la manche et *pieges_sortie* qui donne pour chaque piège différent le nombre de fois qu'il se trouve dans le jeu. Pour compter le nombre d'apparitions de chaque piège durant la manche, il faut ajouter 1 dans le dictionnaire *liste_pièges* à la clé du piège quand celui-ci apparaît. Pour pouvoir en connaître à chaque manche le nombre exact il suffit juste, dans la fonction *fin_de_manche* de retirer, si la raison de la fin de manche est un piègen, 1 dans le dictionnaire *pièges_sortie* à la clé du piège. Cela paraît simple mais l'idée d'utiliser deux dictionnaire ne nous est pas venue tout de suite, c'était en cherchant comment savoir exactement combien de pièges il nous restait et comment calculer le nombre de pièges de chaque type qu'il y avait, que l'idée d'utiliser deux dictionnaires, un avec des valeurs qui

```
if not raison == 'R':
    self.pieges_sortie[raison] -= 1
```

augmente l'autre avec des valeurs qui diminues, nous est venue. Cela nous a grandement simplifié la tâche pour calculer avec la fonction `risque_piège()` la probabilité de retomber sur un piège déjà apparu auparavant.

3 - UNE LISTE DE JOUEURS... MAIS QUI EST L'IA ?

Une IA ! Mais où est-elle ?... Nous avons une liste de joueurs, qui nous permet de faire un classement, mais comment reconnaître notre IA ?

En effet, sans connaître la position de notre IA dans les calculs du moteurs et dans les nôtres, impossible de définir un quelconque classement qui permettrait d'affiner les décisions de l'IA. Il fallait absolument **trouver l'indice de notre IA** pour la placer et connaître son nombre de diamants et son état pour simuler correctement la partie.

Et c'est à ce moment là qu'un souvenir nous revient en tête : Le moteur nous donne l'indice de notre IA ! Un petit nombre caché dans la première chaîne de caractères nous permettait alors de connaître l'indice exact de notre IA dans la liste de joueurs du moteur.

Il manquait plus qu'à ajouter un booléen `est_ia` dans la classe `Joueur`, pour localiser efficacement notre IA parmis les autres joueurs de la partie. Comme ça, dans l'établissement de notre classement, il faudra juste consulter cette variable pour calculer l'écart de notre IA par rapport aux autres, comme expliqué plus tôt.

Ce booléen est pris en paramètre dans la création de notre objet avec la classe `Joueur`. On va donc initialiser le joueur à l'indice de la liste de joueurs correspondant à l'indice donné par le moteur (`id_mon_ia`) avec le booléen `est_ia` en True.

```
for i in range(self.nbjoueurs):
  if i == self.id_mon_ia:
    self.joueurs.append(Joueur(True))
  else:
    self.joueurs.append(Joueur(False))
```

Ensuite, pour retrouver notre IA dans le classement, nous devions juste utiliser une boucle, s'exécutant jusqu'au moment où celle-ci tombait sur le joueur ayant `est_ia` en True. Maintenant, notre IA peut être localisée dans le classement et utilisée dans une fonction comme `ecart_diamants()`.