Compte-rendu Projet Python:
Création d'une intelligence artificielle pour le Tic Tac Toe 9 × 9

Pierre Eustache - Laure-Hélène Genuyt

21 mai 2018

Table des matières

1 Introduction

L'objectif de ce projet a été de créer une intelligence artificielle pour le jeu du Tic Tac Toe 9 × 9 (aussi appelé morpionception, ou Ultimate Tic Tac Toe).

Ce projet nous a permis de mieux comprendre le monde des intelligences artificielles, en en explorant différentes formes (algorithme génétique, algorithme minimax...). Il nous également permis d'apprendre à réaliser une interface graphique, en utilisant le package *tkinter* de python.

Enfin, le fait de réussir à produire une version numérique du jeu, avec une interface graphique permettant à un joueur humain de jouer contre une intelligence artificielle, nous a donné la satisfaction d'avoir réellement "créé" quelque chose grâce à la programmation.

Le projet a été organisé grâce à GitHub [1] qui a permis un partage rapide et efficace des documents lors des tests.

1. `https://github.com/Aquachoc/TTT9`

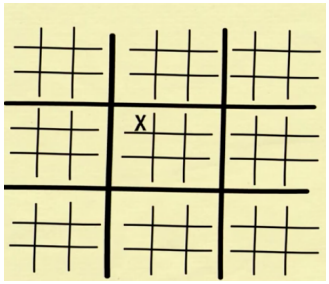

FIGURE 1 – Dans cet exemple, A a joué dans la case en haut à gauche de sa petite grille.

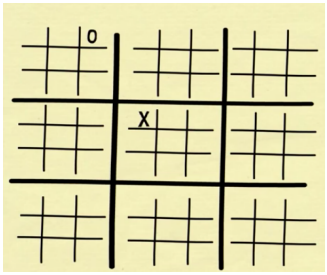

FIGURE 2 – Au tour suivant, B doit donc jouer dans la case en haut à gauche de la grande grille

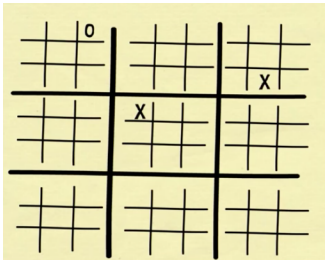

FIGURE 3 – Comme B a joué dans la case en haut à droite de sa petite grille, A doit ensuite jouer dans la case correspondante de la grande grille (etc.)

Le jeu

1. **Règles du jeu**

   Il s'agit d'un jeu de plateau, composé d'une grille de 3 cases par 3, dont chacune des cases contient elle-même une grille de 3 par 3. Dans la suite, on se référera à la "grande grille" et aux "petites grilles" pour distinguer ces deux niveaux de jeu.

   L'objectif du jeu est de remporter 3 cases de la grande grille, en y apposant son symbole (une croix ou un cercle). Pour cela, les joueurs jouent au niveau des petites grilles, et cherchent à les remporter en alignant 3 de leurs symboles. Une fois la petite grille gagnée, on ne peut plus y jouer, et elle devient elle-même une case "croix" ou "cercle".

   Le jeu présente cependant une subtilité qui le rend intéressant : chaque joueur doit jouer dans la case de la grande grille correspondant à la case de la petite grille où a joué l'autre joueur au tour précédent.

   Lorsqu'un joueur renvoie l'autre dans une case de la grande grille déjà gagnée, celui-ci peut jouer où il le souhaite.

   Le jeu se termine dès qu'un joueur a réussi à aligner 3 croix ou 3 cercles dans des cases de la grande grille.

2. **Difficultés du jeu**

   La création d'une intelligence artificielle pour ce jeu, en apparence relativement simple, pose quelques problèmes. En effet, contrairement au jeu du Tic Tac Toe classique, qui se termine par un match nul à moins d'une erreur d'inattention de la part d'un des joueurs, il est difficile de trouver une stratégie gagnante à ce jeu, et de bien en saisir tous les mécanismes.

   De plus, la création d'une intelligence artificielle suppose de pouvoir évaluer chaque coup et de déterminer la valeur d'une position. Or, cela ne va pas de soi dans ce jeu, qui se joue à plusieurs niveaux : un coup doit-il être évalué à l'aune de ce qu'il apporte au joueur dans la petite grille où il est joué ?
   Ou faut-il davantage tenir compte de la case de la grande grille où il renvoie l'adversaire ?

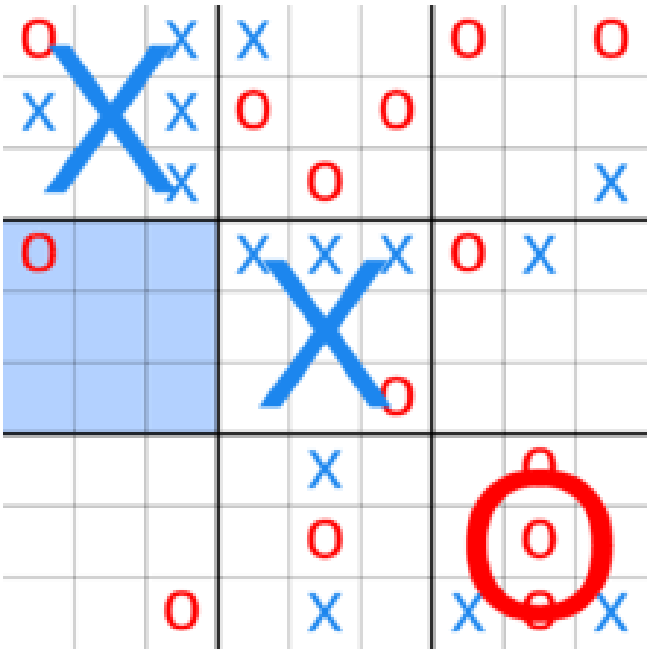

FIGURE 4 – Exemple d'une partie de Tic Tac Toe 9×9

2 Le projet

2.1 Le plan de route

— Nous avons tout d'abord réfléchi à une manière optimale de coder une partie, et plus particulièrement l'état de la grille à un instant donné. Nous avons décidé de répresenter la grande grille comme un tableau 3×3 de tableaux 3×3 (les petites grilles). Chacune des 81 cases est donc référencée grâce à un quadruple indice.

Ex : la position (1,1,3,3) fait référence à la petite case en bas à droite de la grande case en haut à gauche. Notre tableau est composé de 0, de 1 et de 2, où 0 représente une case vide, 1 une case contenant une croix, et 2 une case contenant un cercle.

Nous avons ensuite créé un algorithme permettant de coder ce tableau en binaire, afin d'optimiser l'espace pris par une grille, mais cela ne nous a finalement pas servi par la suite.

— Nous nous sommes ensuite servis du module *tkinter* pour créer une interface graphique affichant l'état de la grille à un instant donné, à partir du tableau décrit précédemment.

— Afin de créer notre intelligence artificielle, nous avons essayé d'évaluer la valeur d'un coup grâce à une analyse heuristique. En effet, contrairement à un jeu d'échecs, où la valeur d'un coup peut être assimilé à la valeur du pion qu'il permet de prendre, il est difficile de trouver une valeur objective d'un coup au Tic Tac Toe 9×9.

Nous avons donc créé une fonction prenant en compte un certain nombre d'indicateurs qui nous semblaient entrer dans l'évaluation d'un coup, et attribuant à chacun une pondération - déterminée d'après notre avis subjectif et notre expérience du jeu dans un premier temps.
Ces indicateurs sont :
— la case que l'on compte jouer se trouve-t-elle dans un coin ?
— se trouve-t-elle au centre d'une case ?
— se trouve-t-elle au bord d'une case ?

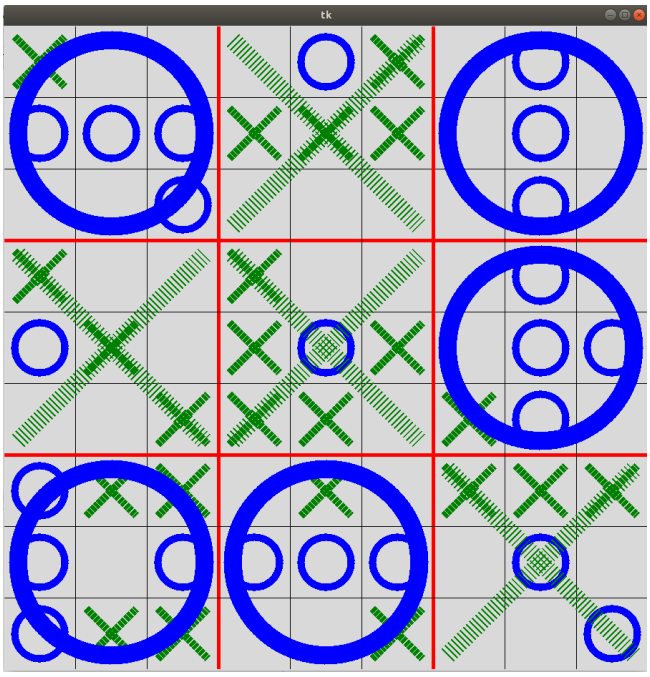

FIGURE 5 – Le programme en action

— combien de symboles a-t-on dans la grille où on renvoie l'adversaire ?
— combien l'adversaire possède-t-il de symboles dans la grille où on le renvoie ?
— la case où on renvoie l'adversaire est-elle déjà gagnée ?
— cette case est-elle vide ?
— ce coup renvoie-t-il l'adversaire dans une case qu'il peut gagner ?
— ce coup permet-il de remporter la case ?
— ce coup permet-il de gagner le jeu ?

Ces indicateurs ont une pondération positive si on juge qu'ils sont en faveur du joueur, négative si on juge le contraire.
On part par exemple du principe qu'il est plus avantageux de jouer au centre que dans un coin, et plus avantageux de jouer dans un coin que de jouer au bord.
On considère également qu'il vaut mieux renvoyer l'adversaire dans une case où il possède peu de symboles, et qu'il faut surtout éviter de l'envoyer dans une case déjà gagnée (ce qui lui permettrait de jouer n'importe où par la suite).

— Une fois cette heuristique créée, nous avons codé les règles du jeu (définition des coups possibles, enchaînement des actions, etc.), avant de créer une intelligence artificielle jouant aléatoirement selon ces règles (IA de type *dummy* dans le code).

— Nous avons ensuite implémenté un algorithme génétique dans le but d'améliorer nos pondérations, et de créer une intelligence artificielle jouant selon ces pondérations.

La performance de ces intelligences artificielles (de type *heurist* dans le code) a été évaluée en fonction de leur taux de victoire et de défaite contre l'intelligence artificielle jouant aléatoirement. L'algorithme génétique se décompose en plusieurs étapes :
— créer un lot de joueurs, jouant selon des pondérations déterminées de façon subjective dans un premier temps.
— faire jouer ces joueurs contre notre intelligence artificielle aléatoire, en enregistrant leurs scores.

- une fois le tournoi terminé, trier les joueurs selon leur score, et en garder la meilleure moitié.
- muter le reste des joueurs, en modifiant de manière aléatoire leurs pondérations.
- avec ce nouveau lot de joueurs (meilleure moitié et joueurs mutés), recommencer à partir de la deuxième étape.

— Au bout de 30 générations, nous avons obtenu les résultats suivants sur 100 matchs :
On voit donc que notre intelligence artificielle générée par cet algorithme génétique présente une très nette amélioration par rapport à notre intelligence artificielle aléatoire.

Type	Heuristique	Aléatoire
Victoires	91	35
Nuls	5	30
Défaites	4	35

— Cette intelligence artificielle heuristique présente cependant une faiblesse majeure, qui est qu'elle n'a aucune profondeur dans son évaluation des coups : elle joue simplement selon des pondérations permettant d'évaluer la valeur du coup qu'elle va jouer
Nous avons donc implémenté un algorithme minimax, permettant à l'IA de voir à une profondeur de 3 coups.

Cet algorithme, que l'on peut représenter sous forme d'un arbre, examine toutes les évolutions possibles du jeu à partir de la configuration actuelle,et attribue à chacune de ces feuilles une valeur (selon les pondérations de la meilleure intelligence artificielle générée par l'algorithme génétique).

On considère que l'adversaire (qui joue au coup d'après), va choisir le coup qui l'amène à la situation la plus bénéfique pour lui, et donc la plus néfaste pour nous.

La valeur d'un noeud de l'adversaire correspond donc au minimum des feuilles évaluées précédemment. La valeur de chacun de nos coups possibles va donc être le maximum des valeurs des noeuds de l'adversaire, et on va jouer le coup possédant la plus haute valeur (il s'agit donc du coup qui nous permettra d'atteindre la situation de jeu la plus optimale pour nous dans 3 coups, en sachant que l'adversaire choisira toujours le coup qui lui est le plus avantageux).

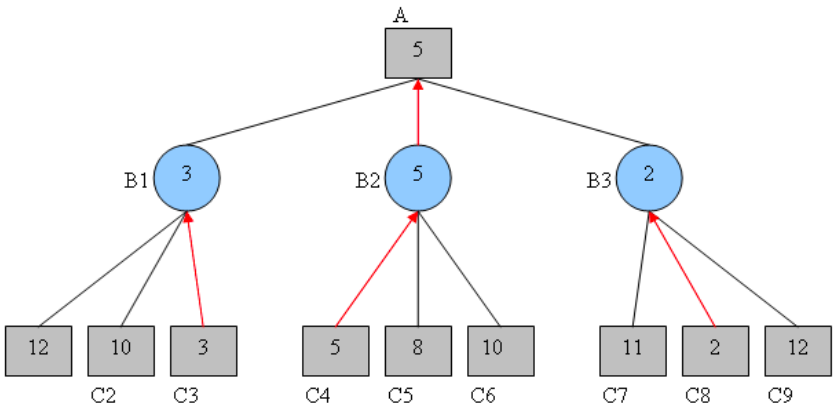

FIGURE 6
Exemple d'arbre minimax (les carrés représentent ici les noeuds joueurs et les ronds les noeuds opposants)

Cet algorithme permet à l'intelligence artificielle (définie dans le code comme étant de type *mini*) de gagner en performance.
Contre l'intelligence artificielle aléatoire, on a en effet observé, sur 200 matchs :

Type	Minimax	Heuristique	Aléatoire
Victoires	187	181	70
Nuls	10	10	60
Défaites	3	8	70

— Une fois ces étapes réalisées, nous avons complété les classes *Player* et *Game*, permettant à chacune de définir les fonctions et les propriétés d'un joueur et d'une partie.
— In fine, notre programme permet donc :
  — faire jouer deux intelligences artificielles (de type *dummy*, *eurist* ou *mini*) l'une contre l'autre
  — de jouer soi-même contre une intelligence artificielle de l'un de ces 3 types, sur une interface graphique adaptée.

$\Longrightarrow$ On a donc bien réussi à créer une intelligence artificielle pour le jeu du Tic Tac Toe 9×9, contre laquelle on peut jouer dans une interface graphique satisfaisante.
Si notre intelligence artificielle la plus satisfaisante (celle jouant selon l'algorithme minimax) est extrêmement performante contre une intelligence artificielle aléatoire, et constitue donc un réel succès sur ce point, elle s'avère cependant un peu faible face à un humain.

Cela est probablement dû à la difficulté d'évaluer simplement une situation de la grille à un moment donné, ainsi qu'à la difficulté qu'il y a à trouver une stratégie gagnante à ce jeu somme toute beaucoup moins simple qu'il n'y paraît.

2.2 Un point de design : la structure de données

Le programme s'articule essentiellement autour de l'utilisation de deux classes : *Player()* et *Game()*.

```
class Game:
    def __init__(self,p1,p2):
        self.p1=p1 #ref the two players
        self.p2=p2
        self.mgrid=np.zeros((3,3),dtype=int)
        self.grid=np.zeros((3,3,3,3),dtype=int)
        self.winner=0
        self.actual=1 #actual player
        self.last=[3,3] #3,3 = no constrains
        self.dispo=np.ones((3,3,3,3),dtype=int)
        self.hist=[]   #keep an history
#         self.gagnable=np.ones((2,3,3),dtype=int)
        self.p1.tmpId=1 #sets for the player their order of turn
        self.p2.tmpId=2
        self.p1.gAct=self.p2.gAct=self #creates a ref to the game for the player
        self.turn=-1 #turn count
        self.mHist=np.zeros((3,3),dtype=int)-1 #hist for mGrid
        if self.p1.IA*self.p2.IA: self.auto_play() # plays
        elif self.p1.IA:
            self.versus(self.p2, self.p1)
        else: self.versus(self.p1, self.p2)
```

```
class Player:
    name=0 #name by order of creation
    def __init__(self,pond=[3,2,1,1,1,1,1,1,1,1,1, math.inf],typ="heuri",IA=1):
        self.score=0
        self.IA=IA
        self.name=Player.name
        Player.name+=1
        self.wins=0
        self.pond=np.array(pond) #heurisitic
        self.tmpId=0
        self.gAct=None
        self.typ=typ
        self.survival=0

    def move(self): #generic move function
        if self.typ=="dummy":
            return self.move1(self.gAct.grid, self.gAct.last, self.gAct.mgrid)
        if self.typ=="heuri":
            return self.move2()
        return self.move3(4)
```

Les méthodes sont pour beaucoup passées sous silence ici.

La partie est une instance de *Game* qui a pour attribut les deux joueurs. C'est également elle qui détient l'état du jeu au sens large, et l'historique des coups. Elle attribue à sa création une ID temporaire qui sert à distinguer les deux instances de joueurs de manière simple dans l'instance de chaque joueur. Elle enclenche ensuite une des deux méthodes de jeu en fonction du fait que les instances passées en argument soient ou non des joueurs ou des IA, puis gère les scores.

Le joueur quant à lui reçoit par le constructeur de *Game()* une référence vers la partie ce qui lui permet d'accéder facilement aux différents paramètres.

Cette structure doublement chaînée permet d'utiliser toute la puissance de l'accès à tous les paramètres à tout moment tout en gardant deux type de structures différents ce qui permet par exemple dans l'algorithme génétique de bénéficier d'une écriture et flexible.

2.3 L'algorithme génétique

Description

- L'algorithme génétique consiste à répéter plusieurs fois les opérations suivantes :
  - créer une génération de joueurs
  - les faire jouer plusieurs fois contre l'intelligence artificielle aléatoire
  - garder la meilleure moitié
  - "muter" l'autre moitié, en modifiant aléatoirement les pondérations auxquelles ces joueurs obéissent
  - recommencer avec cette nouvelle génération

Une étape de cet algorithme (soit une *génération*, correspond à la fonction suivante :

```
def generation(n=30,N=[],nb=100):
    if len(N)==0:
        N=create_batch(n)
        randomize(N)
    dummy_tournament(N,nb)
    mutate(N)
    return N
```

Cette fonction *génération* fait elle-même appel à plusieurs autres fonctions, que nous allons détailler maintenant.

— La fonction *génération* prend plusieurs arguments :
  — n, qui correspond à la taille de l'échantillon de joueurs que nous allons considérer, et qui est ici fixé à 30
  — nb, qui correspond au nombre de matchs que chacun de ces joueurs va faire contre l'intelligence artificielle aléatoire, et qui est ici fixé à 100
  — N, qui correspond à la liste des joueurs

— À la première génération, N est encore vide. On crée donc un lot de joueurs, en faisant appel à la fonction *create batch* :

```
def create_batch(n):
    L=[]
    for i in range(n):
        L.append(Player())
    return L
```

Cette fonction prend en argument n, la taille du lot de joueurs, et renvoie une liste contenant n éléments de la classe *Player*, la classe qui définit les propriétés et les fonctions d'un joueur.

Une fois ce lot créé, on fait appel à la fonction *randomize* pour générer des pondérations aléatoires :

```
def randomize(players):
    n=len(players[0].pond)-1
    for i in players:
        i.pond[:-1]=np.array([random_range*random()-
              random_range*0.5 for _ in range(n)])
```

Cette fonction prend en argument une liste de joueurs et, pour chacun des joueurs de cette liste, leur attribue une tableau de pondérations générées aléatoirement.

— L'étape précédente n'est réalisée qu'à la première génération, si N est vide. Sinon, on passe directement à l'étape suivante : le tournoi entre chacun des joueurs de N et l'intelligence artificielle aléatoire. Pour cela, on fait appel à la fonction *dummy tournament* :

```
def dummy_tournament(players,n=60):
    b=Player(typ="dummy")
    j=0
    l=0
    w=0
    for i in players:
       print(j)
       j+=1
       for k in range(n):
            if l%(n//10):
               print(l)
            if n%2:
                main=Game(i,b)
```

```
            else:
                main=Game(b,i)
            main.auto_play()
            if main.winner==i.tmpId:
                w+=1
    return w20
```

Cette fonction prend deux arguments :
— players, une liste de joueurs
— n, le nombre de fois que chaque joueur va jouer contre l'intelligence artificielle, et qui est ici fixé à 60.

On crée tout d'abord un joueur b, de type *dummy*, et qui correspond à l'intelligence artificielle aléatoire contre laquelle nos joueurs vont jouer.
On crée également j, un compteur du nombre de joueurs qui ont déjà effectué leurs matchs, afin de pouvoir suivre l'avancement du programme.

Ensuite, pour chaque joueur :
— On augmente j de 1 et on l'affiche, pour suivre l'avancée du programme
— On réalise les opérations suivantes n fois :
  — si n est impair, on décide que l'intelligence artificielle aléatoire jouera en deuxième
  — si n est pair, elle commencera à jouer.
  — on joue ensuite la partie
  — si le vainqueur est notre joueur étudié, on augmente son score de 1

  La fonction renvoie enfin le score de chaque joueur.

— Une fois ce tournoi effectué et les scores des joueurs conservés en mémoire, on fait appel à la fonction *mutate* pour créer une nouvelle génération de joueurs.

```
def mutate(players):
    n=len(players)
    k=n//2
    podium(players)
    reset_scores(players)
    p=[]
    m=[]
    for i in range(k):
        p=players[-i-1].pond
        m=mutation(p)
        players[i]=Player(pond=p+m)
```

Cette fonction prend en argument une liste de joueurs, et la "mute" pour créer une nouvelle génération de joueurs.
— On commence par couper la liste de joueurs en deux, en les triant selon leurs scores. Une fois ce tri effectué, on remet les scores à zéro.
— Ensuite, pour chaque joueur de la première moitié de la liste (donc les joueurs ayant obtenu les moins bons scores), on effectue les opérations suivantes :
  — on attribue à p, au départ un tableau vide, le tableau des pondérations de ce joueur
  — on applique la fonction *mutation* à ce tableau de pondérations :

```
def mutation(pond):
    n=len(pond)
    a=np.array([mut_range*random()-0.5*mut_range for _ in range(n)])
    return a
```

  Cette fonction prend en argument un tableau de pondérations, le remplace par un tableau de même taille généré aléatoirement, et renvoie ce nouveau tableau.
  — dans la liste des joueurs, on met à la place du joueur considéré un joueur dont les pondérations sont égales à la somme de ses pondérations de départ et des pondérations mutées.

On obtient alors une nouvelle liste de joueurs, dont une moitié à vu ses pondérations modifiées, et on peut appliquer à nouveau notre fonction *generation* à cette liste.

Lorsqu'on observe les scores des joueurs de chaque génération, on s'aperçoit que chaque génération est plus efficace (i.e. obtient de meilleures scores face à l'intelligence artificielle aléatoire).

Coût de l'algorithme génétique

Bien qu'il est difficile d'estimer le coût d'une partie en elle-même, on peut calculer le coût d'une génération entière. On note dans la suite :

- N le nombre d'individus
- m le nombre de matchs
- f le nombre de facteurs pris en compte dans l'heuristique
- d la profondeur de calcul du minimax

Le calcul demeure assez simple. On suppose dans la suite que l'accès aux éléments d'une liste en python est linéaire de sa taille, et qu'une partie est jouée en temps constant. Calculons les participations individuelles de chaque élément.

- La création est $\mathcal{O}(N)$.
- Le calcul sur une couche de l'heuristique est $\mathcal{O}(f)$ car la grille est de taille constante.
- Le tri de la liste se fait en $\mathcal{O}(N.ln(N))$
- Parcourir la liste en faisant jouer chaque joueur est $\mathcal{O}(N.m.P)$ avec P le temps d'une partie.
- La dépendance selon d est exponentielle, et dépend du nombre de facteurs à vérifier sur chaque niveau $\mathcal{O}(f^d)$
- Enfin la mutation se fait en temps linéaire selon f et N

On en déduit un total de

$$\boxed{\mathcal{O}(N.m.f^d + N.ln(N))}$$

2.4 Problèmes rencontrés

Le principal algorithme utilisé est l'algorithme génétique. Bien que celui-ci soit en théorie efficace, la réalité et les restrictions en terme de temps d'exécution acceptables nous ont forcé à déterminer expérimentalement ses paramètres principaux : combien de joueurs ? quels conditions initiales ? faut il mettre en place de petites ou grosses mutations ?

Après plusieurs tests, nous avons déterminé que même en convergeant vers des IA relativement différentes, les différents paramètres faisaient évoluer les pondérations dans une même direction globale (notre heuristique est à considérer à une constante multiplicative près). Cependant, les paramètres qui se sont avérés importants ont été l'intensité de la mutation, et le nombre d'individus fixés respectivement à $mut\_range = 6, n = 30$ afin de concilier le meilleur de la performance en un temps raisonnable.

3 Exécution du programme (usage)

Le programme est implémenté prêt à jouer contre l'utilisateur. En effet, l'IA par défaut est déjà correctement paramétrée en ayant en mémoire l'une des pondérations les plus plus prometteuses trouvée durant notre algorithme génétique.

Cependant, le programme permet à l'utilisateur de manipuler à loisir les différentes générations et d'ainsi pouvoir voir se dérouler des parties entre différentes IA de son choix.

usage :

- *Player(pond = [float], typ=str)* avec *pond* les pondérations de l'IA, et *typ* pouvant prendre les valeurs :
  - *"dummy"* → aléatoire
  - *"heuri"* → simple heuristique
  - *"mini"* → heuristique + minimax
- *Game(p1=Player(), p2=Player())* avec *p1, p2* respectivement le premier et le second joueur à jouer.
- *Game.hist()* affiche un historique de la partie.

Le programme a également été optimisé pour rendre automatique l'algorithme génétique, si l'utilisateur désire tenter de développer sa propre colonie.

— $training(nb = int, gen = int, nb\_games = int) = list[Player]$
Génère une couvée de *nb* joueurs, sur *gen* générations, en leur faisant vivre *nb_ games* pour les départager.
— $mut\_range = 6$ Détermine l'ampleur de la mutation à chaque génération
— $random\_range = 10$ Détermine à quelle distance de 0 se trouvent les initialisations de l'algorithme génétique.

4 Améliorations et perspectives

Le programme peut encore être amélioré.

1. **Interface graphique**

   L'interface graphique actuelle est très sommaire car elle n'a été développée que dans des perspectives de test et de debug. Dans cette optique, il serait possible d'implémenter une interface plus complexe qui permettrait de commander les opérations implémentées facilement et lancer des parties contre différentes IA, de difficulté croissantes.

   Il serait également possible de faire l'exact exemple et d'incorporer un éditeur d'IA qui permettrait de défier ses amis en développant une IA battant la leur, à la manière de LeakWars [2].

2. **Intelligence artificielle**

   L'étape suivante dans la quête d'une IA serait d'adopter le paradigme du réseau de neurone afin de remplacer l'heuristique. En effet, il serait intéressant d'entraîner un réseau de neurones qui noterait sa position d'arrivée et permettrait de choisir un coup de manière à maximiser sa valeur d'arrivée.

   Cela nécessiterait de passer de la méthode actuelle de notation qui se fait en fonction du coup proposé, et de la situation actuelle, à une notation d'une grille seule.

3. **Applications**

   Le morpionception est un jeu dont l'arbre de décision est extrêmement complexe. Le résoudre pourrait se faire de manière partielle avec l'implémentation d'une IA qui gagne à tous les coups. Sa mise au point serait alors équivalent à résoudre le jeu. Il peut également être intéressant d'utiliser d'autres langages qui permettent une meilleure gestion de la mémoire comme le C++ afin d'essayer de générer l'arbre de jeu entier.

2. `https://leekwars.com/`