Rapport Projet d'Informatique 1A :

Mathis SCHEFFLER
Baptiste FERRERE

Objectif du projet :

Nous avons souhaité programmer un Intelligence Artificielle (IA) capable de conduire une voiture dans un espace euclidien à deux dimensions. L'entrainement de l'IA avait pour objectif principal d'obtenir une intelligence artificielle qui ne commet pas d'accident et dans la mesure du possible qui va vite. Pour cela nous avons codé intégralement l'espace dans lequel évolue l'IA et en avons fait un jeu pour qu'un humain puisse y jouer et se comparer à notre IA. L'entrainement de l'ordinateur s'est fait grâce à une implémentation python de l'algorithme neat (Neuro Evolution with Advanced Topologies).

Plan du rapport :

- I) Structures du code
- II) Fonctionnement du jeu
- III) Fonctionnement de l'IA
- IV) Optimisation
- V) Indicateurs de performances de l'IA et limites du programme
- VI) Projet abandonné et idées d'amélioration

I) Structure du code :

a) Description des fichiers :

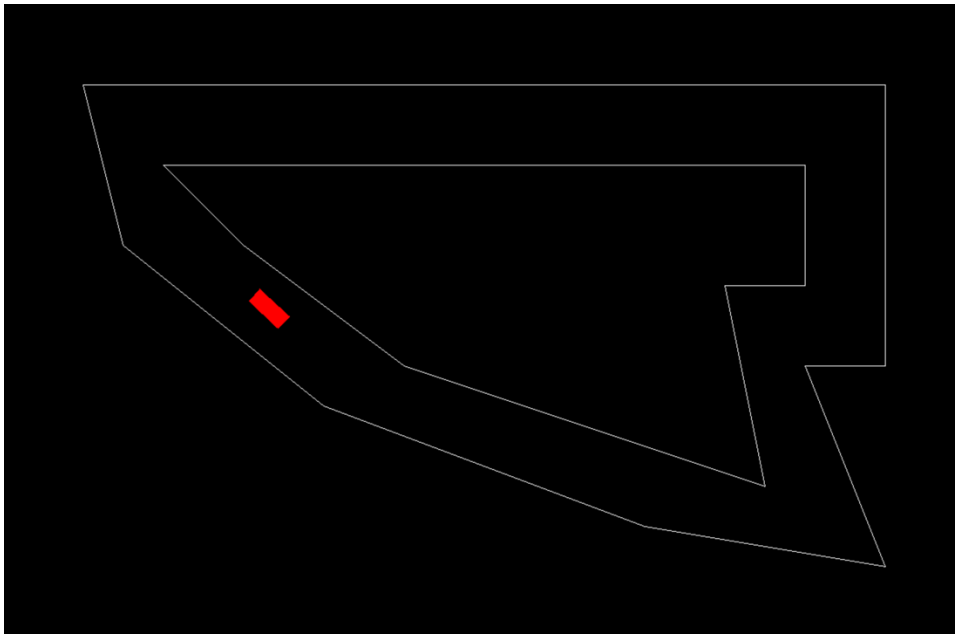
track.py : gère le circuit, les collisions et le calcul des inputs de l'IA.
player.py : gère le joueur.
geometry_class : définit les objets ligne et rectangle utiles dans le reste du code.
math_extra.py : algorithmes utiles pour le calcul des collisions.
read_stats.py : permet de visualiser les performances de l'entrainement.
unit_test.py : permet de tester l'efficacité de l'IA après un certain temps d'entrainement.
main.py : exécutable principal du code. Initie les éléments du code, leur affichage, mise à jour et interactions.

Le code repose sur la structure suivante :

- Une initialisation : Game.__init__() qui crée les objets nécessaires au code
- Une boucle qui ne s'arrête que lorsque l'utilisateur appuie sur échap qui va en permanence mettre à jour les variables, obtenir les entrées clavier du joueur et dessiner le jeu (si l'affichage est activé).

La variable Game.GAMESTATE donne le mode de jeu (jeu solo, observation de l'IA, course contre l'IA ou entrainement de l'IA).

Voici une image de l'interface en mode jeu solo :



b) Fichier test :

Le code du fichier unit_test.py permet de lancer un test sur la performance de l'entrainement de l'IA. On choisit le nombre d'entrainement que l'on souhaite effectuer, le nombre de générations à entrainer et le fitness souhaité. L'algorithme va renvoyer la proportion d'entrainement qui réussissent à entrainer une IA avec un meilleur fitness que celui entré.
Attention l'exécution de ce code est très longue. Si le nombre de générations est plus petit que 20 il faut de l'ordre de 6 secondes pour tester une génération. Si le nombre de générations est plus grand que 20, ce temps monte alors vers les 25 secondes.

c) Commandes :
'B' : active/désactive l'affichage.
'1' : passe en mode jeu solo.
'2' : passe en mode spectateur.
'3' : passe en mode course contre l'ordinateur.
'9' : lance un entrainement sur 51 générations.
'échap' : permet de quitter le jeu
'Z' : accélère la voiture
'S' : freine
'Q' : tourne à gauche
'D' : tourne à droite

II) Fonctionnement du jeu :

a) Les déplacements du joueur :

Les déplacements du joueur se basent sur le principe de développement limité. Entre l'affichage de deux frames, se passe un temps dt de l'ordre de 1/60s (60 fps). L'accélération est calculée en fonction des inputs/entrées du joueur. On obtient la vitesse en faisant $\vec{v} = \vec{v} + dt * \vec{a}$. Dans notre programme la vitesse et l'accélération sont toujours colinéaires.
De même pour la position : $\vec{P} = \vec{P} + dt * \vec{v}$.
L'orientation de l'accélération et de la vitesse est déterminée par l'angle t.

```
def Update(self, inputs=None):
    self.player.x += dt * self.player.s * cos(self.player.t)
    self.player.y += dt * self.player.s * sin(self.player.t)

def R_turn(self):
```

```
    self.player.t += dt * self.angularspeed

def L_turn(self):
    self.player.t -= dt * self.angularspeed

def accelerate(self):
    self.player.s += dt * self.acceration

def decelerate(self):
    self.player.s = max(0, self.player.s - dt * 5 * self.acceration)
```

Pour que les mouvements de l'IA ne dépendent pas des performances de l'ordinateur, nous n'avons pas pris dt = 1/fps mais simplement dt = 1/60. Il en sera de même dans tout le code pour toute variable dt.

Les mouvements du personnage sont gérés par la classe Controller. Le joueur ne peut qu'avancer, pas reculer.

b) Les collisions

Les collisions se basent sur des calculs d'intersections de segments. On peut obtenir les segments qui constituent les bords de la voiture du joueur grâce à sa position, l'angle qui définit son orientation, sa largeur et sa longueur (la voiture du joueur étant un rectangle). Pour vérifier s'il y a une collision, on vérifie si l'un des segments qui délimite le circuit intersecte l'un des bords de la voiture du joueur. Le joueur ne pouvant pas reculer, on ne vérifie pas les collisions avec l'arrière du véhicule. S'il y a intersection, le joueur meurt.

Les calculs d'intersection entre deux segments $s1$, $s2$ sont fait de la manière suivante :

- On calcule le point d'intersection entre les droites qui prolongent $s1$ et $s2$. Si jamais $s1$ et $s2$ sont parallèles, on retourne directement que les segments ne se croisent pas.
- On vérifie si ce point d'intersection est sur les segments.

c) L'affichage :

L'affichage est effectué grâce à la bibliothèque pygame. Il est possible de désactiver l'affichage en appuyant sur la touche b. Ceci n'est utile que lors de l'entrainement de l'IA. En effet l'affichage des graphismes pendant l'entrainement de l'IA a deux défauts majeurs :

- Le dessin des graphismes a un coût non négligeable en complexité
- Lors de l'affichage, le jeu ne dépasse pas les 60 fps ce qui bride les performances.

III) Fonctionnement de l'IA :

a) Principe et fonctionnement de neat :

L'algorithme neat se base sur la structure du réseau de neurones. À l'image d'un algorithme de renforcement, cet algorithme va modifier le paramétrage du réseau de neurones. Ce qui fait la particularité de cet algorithme est qu'il joue aussi sur la structure du réseau de neurones en rajoutant/supprimant des neurones et des connections. Cet algorithme permet d'obtenir des résultats plus vite qu'avec des algorithmes plus classiques de renforcement. Les données du réseau de neurones sont stockées dans une structure appelée génome.

b) Notre utilisation de la bibliothèque neat :

- Initialisation de l'algorithme :

  Nous chargeons le fichier config.txt qui stocke les paramètres de l'algorithme (nombre d'inputs/outputs, fonction d'activation, etc…)
  Nous créons également un reporter qui va permettre d'obtenir des informations sur l'entrainement facilitant le débogage et permettant de visualiser les performances du code.
  Nous lançons ensuite l'algorithme qui va sur cet exemple entrainer sur 50 génération notre IA et retourner le meilleur réseau de neurones (sous la forme de génome).

```
population = neat.Population(config)
population.add_reporter(neat.StdOutReporter(True))
stats = neat.StatisticsReporter()
population.add_reporter(stats)
population.add_reporter(neat.Checkpointer(500))

winner = population.run(self.run_car, 50)
```

- Boucle principale :

  Premièrement, nous traduisons tous les génomes donnés en arguments en réseau de neurones, créons les objet player et leur ajoutons un contrôleur d'IA

```
for id, g in genomes:
    net = neat.nn.FeedForwardNetwork.create(g, config)
    nets.append(net)
    g.fitness = 0

    cars.append(Player(170., 200., 50., 20., 1., 100.))
    cars[len(cars)-1].add(Ctl.IA_Controller(cars[len(cars)-1]))
```

  Ensuite et ce jusqu'à ce que toutes les IA aient perdu ou aient réalisé 10000 itérations, le programme calcul les inputs pour chaque IA, les fait conduire en fonction des inputs et les affiches si besoin.

```
            if(self.GAMESTATE == GameState.race):
        self.player.Update()
        self.track.update(self.player)
        if not self.player.alive:
            self = Game(GameState.race)

    for index, car in enumerate(cars):
        outputs = nets[index].activate(car.inputs())
        car.Update(outputs)
        self.track.update(car)

    remain_cars = 0

    for i, car in enumerate(cars):
        if(car.alive):
            remain_cars += 1
            genomes[i][1].fitness += car.get_reward()
            if(player_zone_check(car)):
                genomes[i][1].fitness += 100.
    if(remain_cars == 0):
        break
```

La fonction player_zone_check empêche les IA de faire demi-tour.

Le programme calcul également la fonction fitness qui quantifie la performance de chaque génome.

- Fin de l'algorithme :
  À la fin de l'algorithme, le meilleur génome est sauvegardé dans un fichier afin de pouvoir être observé plus tard en mode spectateur ou en faisant la course contre l'IA.
  De même, les statistiques concernant l'entrainement sont sauvegardées.

c) Les inputs et outputs :

Le réseau de neurones reçoit en arguments 6 paramètres.
Les 5 premiers indiquent à quel point la voiture est proche des murs. Ces chiffres sont obtenus en calculant les interceptions entre le circuit et des segments partant du centre du joueur et allant dans 5 directions différentes.
Le dernier input est la norme de vitesse du joueur.

Les outputs permettent à l'IA de faire tourner la voiture et de la faire accélérer. L'IA ne peut pas freiner, ce choix est expliqué en partie 4 de ce rapport.

IV) Optimisation :

Les performances ont toutes été obtenues sur le même ordinateur.
Lorsque l'affichage est désactivé, il faut de l'ordre de 6 secondes pour effectuer l'entrainement des premières générations d'IA, celles qui ont des accidents. Quand l'ordinateur arrive à conduire sans accident, ce temps passe à de l'ordre de 23 secondes. Si l'affichage est activé, ces temps sont doublés.

a) Compilation des formules mathématiques :

Dans le fichier math_extra.py, les formules sont compilées une première fois en langage constructeur. Lors des futurs appels de ces fonctions, le code déjà compilé va être appelé plutôt que le code python. Ceci fonctionne grâce à la bibliothèque numba. Cela permet de gagner quelques secondes par génération ce qui facilite grandement le débogage en plus d'entrainer plus rapidement l'IA.

b) La fonction Update de la classe Track :
Il s'agit de la fonction la plus gourmande du code vu qu'elle est appelée pour chaque voiture (150 lors d'un entrainement). Pour l'optimiser, il est important de ne plus faire les calculs de collisions avec les voitures qui ont déjà eu un accident.
Nous avons également fait attention à appeler la fonction player.rect.corner() qu'une fois.

Ces différentes optimisations ont permis de faire passer le temps de calcul de 120-180 secondes à 5-30 secondes par génération. Il faut également noter que la fonction Game.run_car() a été réécrite plusieurs fois pour obtenir un code plus efficace.

V) Indicateurs des performances de l'IA et limites du programme

a) La fonction fitness :

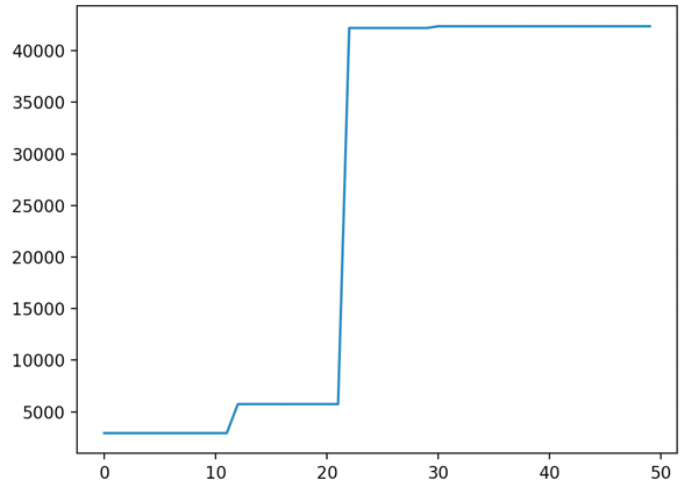
La fonction fitness choisie est la suivante :

```
def get_reward(self):
    return 2. + self.s / 50
```

Elle récompense les voitures qui survivent tout en les incitant à aller le plus vite possible.

b) Les résultats d'entrainement :

Le graphique suivant montre l'évolution du fitness du meilleur génome en fonction de la génération lors d'un entrainement sur 51 générations.



L'algorithme neat étant basé sur des modifications aléatoires d'un réseau de neurones, ce graph peut varier grandement d'un entrainement à un autre. Il n'est pas rare d'avoir dès la génération 2 des voitures qui peuvent effectuer un tour de circuit (finess ≥ 10 000).

Les grandes améliorations du génome sont généralement et dans l'ordre d'apparition : la voiture apprend à tourner à gauche pour éviter de foncer dans le mur (fitness ≥ 2000), la voiture apprend à tourner à droite pour éviter les obstacles, elle peut alors conduire souvent jusqu'à un tour complet (fitness ≥ 6000) ensuite l'IA n'a plus d'accident (fitness ≥ 40 000). Après ces grandes étapes, le fitness maximum augmente plus lentement lorsque l'IA apprend à avoir de meilleures trajectoires et à accélérer sans avoir d'accident ce qui lui donne une suite de petites améliorations comme ici à la génération 30.

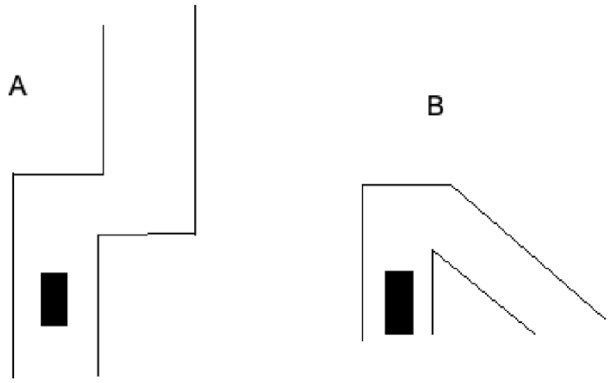
c) Notre IA :

L'IA pré-enregistrée des modes *spectator* et *race* a été entrainée en 120 générations et a un fitness de 74426.5.

Elle présente un défaut majeur. En effet, elle n'accélère pas assez dans la ligne droite. Cela pour deux raisons :
Premièrement, le paramètre player.rect.sight_reach vaut 300 ce qui signifie que l'IA ne se rend pas compte de l'espace laissé devant elle pour accélérer. Ce paramètre a été laissé à 300 car augmenter sa valeur rend l'entrainement significativement plus long. De plus, des essais avec une valeur de 800 ont montré que l'IA se retrouve dans des situations ou ses inputs étaient les

mêmes mais les actions que la voiture devait faire étaient différentes. Ce genre de phénomène ne se retrouve pas avec la valeur de 300.
Deuxièmement, l'IA ne peut pas freiner. Cela a été également choisi car l'IA se retrouvait dans des situations ou ses inputs étaient les mêmes mais les actions qu'elle devait prendre étaient différentes. Ce choix a permis de grandement accélérer l'entrainement au prix de performances.

Exemple :



Dans ces deux situations, les inputs des voitures sont les mêmes. Pourtant la voiture A devrait accélérer et la voiture B devrait freiner.

Ainsi la principale limite de notre code concerne les inputs. Il serait plus efficace de prendre comme input l'image du jeu. Cependant le temps de calcul et d'entrainement nécessaire à l'entrainement de cette IA nous a découragé.

d) Jeu contre :

En une centaine de parties contre l'IA, jouées par 5 personnes, personne n'a réussi à battre l'IA sur une course de 5 tours bien qu'elle ne soit pas parfaite. Nous pouvons donc conclure que les résultats de notre IA, sont, contre toute toutes premières attentes, largement satisfaisants.

VI) Projet abandonné et idées d'amélioration:

a) La première version de ce code n'utilisait pas pygame. Nous utilisions la bibliothèque pyglet. Cependant le fonctionnement de la librairie neat et du moteur pyglet ne permettaient pas d'observer et d'entrainer les IA dans une même fonction de code. Nous avons essayé de visualiser l'entrainement en

lançant pyglet en programmation asynchrone. Malheureusement pyglet ne supporte pas d'être lancé en asynchrone. Nous avons donc réécrit notre code en utilisant pygame.

b) Idées d'amélioration :

- Segmenter le circuit en plusieurs zones. N'effectuer que certaines vérifications de collisions en fonction de la zone dans laquelle le joueur se trouve. (Pas besoin de vérifier si le joueur touche un mur en bas à droite s'il se trouve en haut à gauche)
- Entrainer l'IA sur l'image du jeu et la laisser freiner.
- Entrainer l'IA sur différents circuits en même temps en prenant comme fitness le minimum de ses scores sur chaque circuit afin d'obtenir une IA plus polyvalente.