Fondamentaux de l'IA - Algorithme génétique

BARDE Alexandre

15 février 2020

Table des matières

1 Introduction **2**
1.1 The Maze Runner 2
1.2 Les algorithmes génétiques 2
1.3 Premier jet 3
1.4 La fitness ? 7

2 Annexe : le code **11**

1 Introduction

1.1 The Maze Runner

Je n'avais pas vraiment d'idées de projet ni de ce qui est était faisable en 2 semaines sachant que nous sommes en entreprise la moitié du temps. J'ai donc cherché un peu sur internet ce qui était possible, la plupart des exemples étaient soit le TSP, sac à dos ou bien des programmes basiques. J'ai relu le cours et j'ai vu qu'il y avait un exemple sur un robot qui cherchait la sortie d'un labyrinthe. Voici donc mon projet, il s'agit de trouver la sortie d'un labyrinthe mais avec quelques particularités. Il doit trouver la sortie avec le moins de déplacement possible, sur certaines cases il y a des pièges qui tuent le personnage.

1.2 Les algorithmes génétiques

C'est un type d'algorithme évolutionniste dont le but est de trouver une solution approchée à un problème d'optimisation. L'algorithme génétique est inspiré de la théorie de l'évolution qui dit que :

— Une espèce connaîtra forcément des variations aléatoires.
— Si la variation est gênante pour l'individu, il ne se reproduira pas ou peu, et cette variation disparaîtras.
— Si cette variation est avantageuse, il se reproduira plus et elle se diffusera dans les générations futures.

Dans un algorithme génétique, on retrouve toujours les composantes suivantes :

L'individu

C'est simplement une solution potentielle au problème. Un individu n'est rien de plus qu'un résultat. Quand on parle d'un individu en GA/GP, on parle en réalité d'une instance de la structure de données contenant un résultat, qu'il soit optimum ou non.

Dans notre exemple, un individu n'est rien d'autre qu'un "pion" qui doit sortir du labyrinthe.

La population

Une population est un ensemble d'individus divers. Analogiquement à la biologie, c'est une espèce.

La “fitness“

C'est une valeur associée à chaque individu, elle permet de quantifier à quel point une solution est adaptée au problème. Et des opérations permettant de faire évoluer une population vers une génération meilleure.

L'évaluation

Elle consiste à analyser tous les individus de la population pour associer à chacun une valeur de fitness.

La sélection

C'est la méthode qui permet de choisir dans la population 2 parents pour générer un individu fils. Si on fait le parallèle avec la théorie de l'évolution, cette opération représente la sélection naturelle, les individus avec les meilleures caractéristiques, et donc la meilleure fitness, on plus de chance de survivre, ce qui est matérialisé par le fait qu'ils ont plus de chance d'être sélectionnés pour se reproduire et transmettre leurs caractéristiques au générations futures.

Le croisement, ou "crossover"

Croiser 2 individus représente le processus de reproduction dans la nature. Il revient à créer un individu fils en combinant 2 parents, les caractéristiques du fils seront alors un mélange aléatoire de celles des parents.

La mutation

Un mutation est un changement aléatoire des caractéristiques d'un individu. La mutation est une étape qui consiste à introduire aléatoire des modifications dans les constituants d'un individu.

1.3 Premier jet

J'ai passé énormément de temps sur l'élaboration de la partie graphique du projet. Comment faire apparaitre une fenêtre avec le labyrinthe à l'intérieur ? Comment ajouter des "*pions*" / personnages qui vont naviguer à l'intérieur ? Etc ..

Donc pour commencer, j'ai créer un fichier texte qui contient plusieurs lignes, avec des 0 et des 1 pour les pièges ainsi que l'espace "vide" où peut aller le joueur. Il y a également un 2 et un 3 qui correspond successivement au départ des joueurs et de l'arrivée.

Voici un exemple de fichier texte :

1111111111
1020000001
1011110001
1000000001
1000000101
1000100001
1001000001
1000011001
1000000301
1111111111

Ce labyrinthe fait 10 cases par 10 cases, ce qui simplifie pour le début les algorithmes, la fenêtre quant à elle fait 400px par 400px, donc une case fait 40px par 40px.

Pour afficher les différentes cases j'ai procédé comme ceci :

```
def _generate_square(self, canvas):
    chars = self._data
    for i in range(len(chars)):
        chars[i] = chars[i].replace("\n", "")
        line_tmp = list(chars[i])
        for j in range(len(line_tmp)):
            x_start = 40 * j
            y_start = 40 * i
            x_end = (40 * j) + 40
            y_end = (40 * i) + 40
            if line_tmp[j] == "1":
                self._walls.append(str(x_start) + ":" + str(y_start) + " " + str(x_e
                canvas.create_rectangle(x_start, y_start, x_end, y_end, outline="#47
            elif line_tmp[j] == "0":
                canvas.create_rectangle(x_start, y_start, x_end, y_end, outline="#47
            elif line_tmp[j] == "2":
                canvas.create_rectangle(x_start, y_start, x_end, y_end, outline="#47
            elif line_tmp[j] == "3":
                canvas.create_rectangle(x_start, y_start, x_end, y_end, outline="#47
```

Il parcourt les différentes lignes du fichier, il remplace les caractères "retour chariot" par du vide. Ensuite il parcourt chaque caractères dans la ligne et vérifie à quoi il correspond et il créer le carré en fonction de la valeur.

J'ai ensuite commencé à ajouter un rond pour essayer de le déplacer dans la grille, mais une fois que la fenêtre était lancée grâce à :

```
def _generate_app(self):
    self._window.mainloop()
```

Il n'est plus possible de faire bouger un canvas, j'ai donc essayé d'intégrer le module **Turtle** afin de pouvoir déplacer aisement un object sur mon canvas. Néanmoins il n'est pas possible de réutiliser ce même canvas, j'ai alors essayé d'exporter mon canvas en image afin de l'ouvrir avec Turtle, mais sans succès également.

J'ai alors eu l'idée d'ajouter un bouton qui fait bouger mon rond et par ce biais, cela fonctionne ! Voici alors à quoi ressemble l'interface :

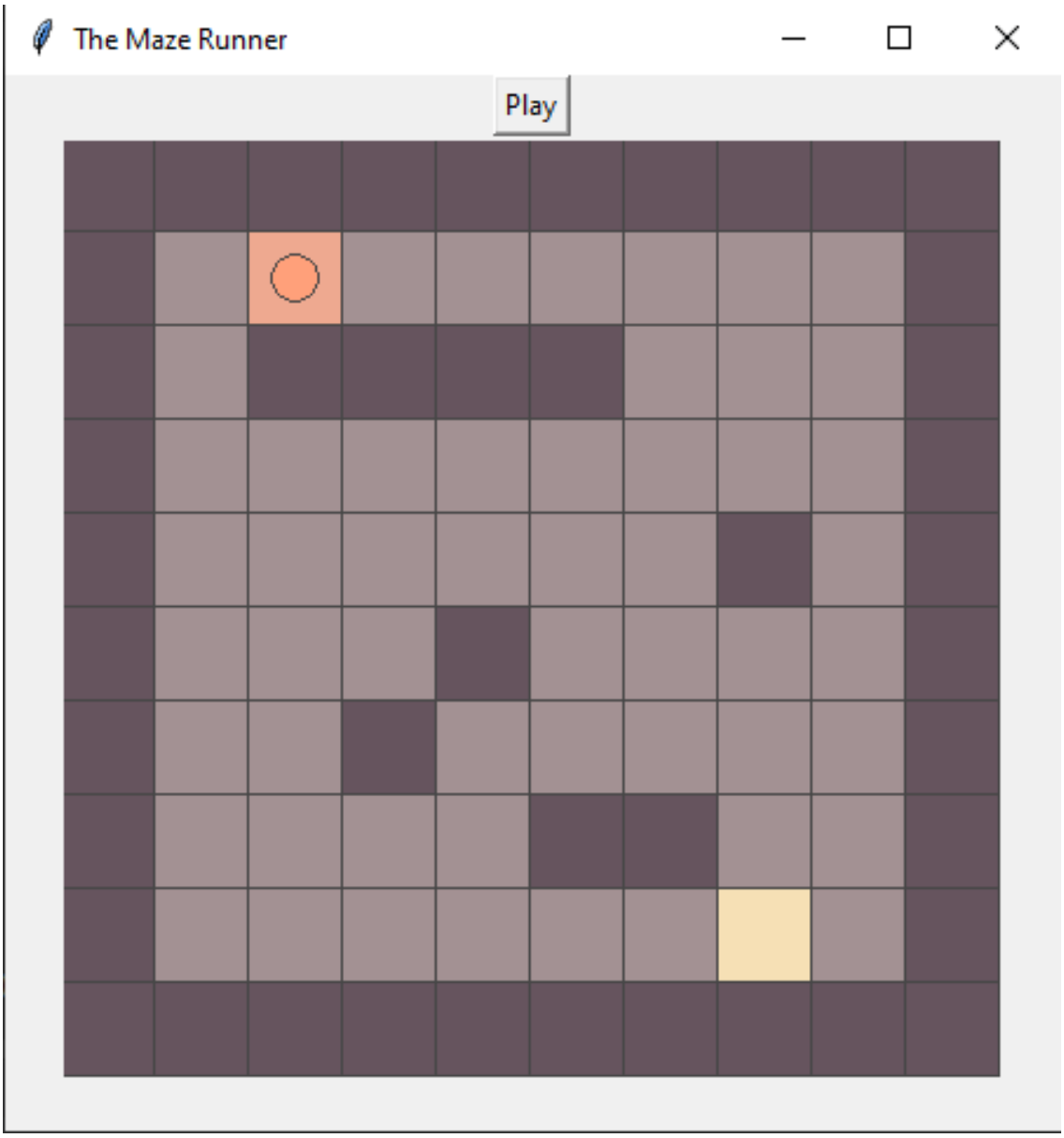

FIGURE 1 – Première version de la fenêtre de l'application

A partir de ce moment-là, je pouvais commencer à travailler sur les populations, le calcul de ma fitness, etc .. Mais j'avais déjà perdu beaucoup de temps sur la partie graphique ..

La première chose à faire était de pouvoir créer plusieurs "ronds" qui seraient plus tard mes différents individus dans ma population.

J'ai d'abord récupérer une liste d'environ 300 couleurs sur la documentation de Tkinter afin que chaque rond possède une couleur distincte (il y a une liste de couleur, j'en prend une aléatoirement et je la retire de la liste pour ne plus tomber dessus).

Voici la fonction permettant de créer x ronds (individus) dans le labyrinthe.

```
def _create_(self):
    for i in range(self._nbr_pop):
```

```
        self._pop.append(self._canvas.create_oval(90, 50, 110, 70, outline="#474747"
        self._runners.append(Runner(90, 50, 110, 70, self._walls))
```

Le programme possède une liste *pop* ainsi qu'une liste *runners*, la première stocke les canvas des ronds et la seconde stocker quant à elle les différents individus de ma population.

Après avoir ajouté le nombre souhaité d'individus, ils sont placés aux coordonnées du point de départ du labyrinthe. (cf figure 1, nous pouvons apercevoir uniquement le rond qui a été créer en dernier, les autres sont placés en dessous).

J'ai ensuite implémenté le bouton "play"qui permet à chaque *clic* de déplacer chaque individus sur les 4 axes :

— droite
— gauche
— haut
— bas

```
    def callback():
        individu_mort_pop = []
        individu_mort_runners = []
        print(self._pop)
        for i in range(len(self._pop)):
            nbr = random.randint(0, 3)
            if nbr == 0:  # à droite
                self._canvas.move(self._pop[i], 40, 0)
                self._runners[i].new_pos(self._canvas.coords(self._pop[i]))
                if not (self._runners[i]).get_is_alive():
                    individu_mort_pop.append(self._pop[i])
                    individu_mort_runners.append(self._runners[i])
            elif nbr == 1:  # en bas
                self._canvas.move(self._pop[i], 0, -40)
                self._runners[i].new_pos(self._canvas.coords(self._pop[i]))
                if not (self._runners[i]).get_is_alive():
                    individu_mort_pop.append(self._pop[i])
                    individu_mort_runners.append(self._runners[i])
            elif nbr == 2:  # à gauche
                self._canvas.move(self._pop[i], -40, 0)
                self._runners[i].new_pos(self._canvas.coords(self._pop[i]))
                if not (self._runners[i]).get_is_alive():
                    individu_mort_pop.append(self._pop[i])
                    individu_mort_runners.append(self._runners[i])
            elif nbr == 3:  # en haut
                self._canvas.move(self._pop[i], 0, 40)
                self._runners[i].new_pos(self._canvas.coords(self._pop[i]))
                if not (self._runners[i]).get_is_alive():
                    individu_mort_pop.append(self._pop[i])
                    individu_mort_runners.append(self._runners[i])
```

Si un individu tombe dans un piège, il meurt, si la taille de la population vaut 0, une nouvelle génération commence.

```
            if not self._runners[i].get_is_alive():  # si l'individu est mort
                self._nbr_alive = self._nbr_alive - 1
```

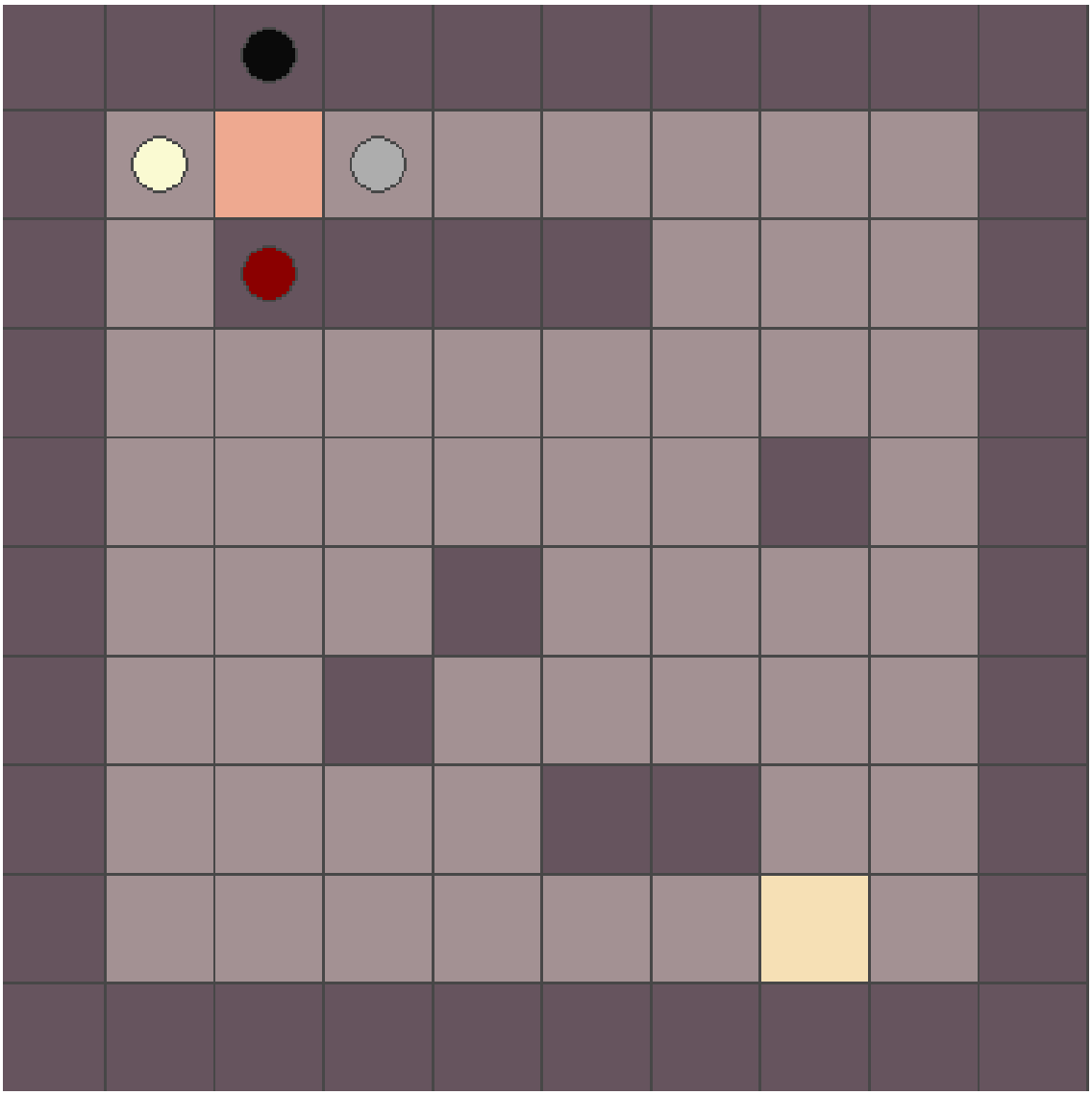

FIGURE 2 – Implémentation basique de la population

1.4 La fitness ?

Pour calculer la fitness je vais utiliser la méthode des dominants. C'est à dire que nous gardons uniquement les individus qui possèdent les meilleurs fitness.

Avant de se lancer tête baissée dans le développement de la fitness, j'ai essayé de déterminer comment j'allais pouvoir calculer ma fitness. J'ai pensé à calculer en fonction des

individus qui arrivent à l'arrivée, le problème c'est que c'est fort possible que les individus meurent avant de l'atteindre. Donc ce n'est pas la bonne solution. La seconde option est de garder les individus qui ont la durée de vie la plus petite car ça veut dire qu'ils ont réussi à atteindre l'arrivée le plus rapidement possible (pas par le chemin le plus rapide). Néanmoins avec cette métohde ceux qui vont dans un piège et qui meurent ont une durée de vie très courte aussi il faut alors couper ce critère avec la distance qui les sépares de l'arrivée.

Cela fait donc :

> Pour un *individu lambda* ll existe au moins un dominant si un *individu lambda* à une distance à l'arrivée et une durée de vie plus faible.

Voici le code python pour le calcul de la fitness :

```
def get_fitness(self):
    move = self._nbr_move
    diff_x = self._pos_x_2 - self._pos_x_1
    diff_y = self._pos_y_2 - self._pos_y_1

    pos_x = diff_x + self._pos_x_1
    pos_y = diff_y + self._pos_y_1

    diff_x_end =  self._end[2] - self._end[0]
    diff_y_end = self._end[3] - self._end[1]

    pos_x_end = diff_x_end + self._end[0]
    pos_y_end = diff_y_end + self._end[3]

    distance = round(((pos_x - pos_x_end) ** 2 + (pos_y - pos_y_end) ** 2) / 100)

    self._fitness = distance * move
```

Je récupère le nombre de déplacements ainsi que les coordonnées du rond et les coordonnées de l'arrivée. Ensuite vient la distance et je calcul la fitness par la multiplication des deux termes.

Après plusieurs l'implémentation de la fitness, j'ai essayé l'application, mais j'ai eu plusieurs erreurs à cause de la fonction récursive (max recursion depth exceeded while calling a python object). Après plusieurs recherche j'ai du ajouté ces lignes sur tous les scripts python :

```
sys.setrecursionlimit(99999999)
threading.stack_size(200000000)
```

Lorsqu'une génération se termine, je modifie la dernière direction de chaque individu (car vu qu'il est mort, c'est qu'il est tombé dans un piège).

Une fois la solution trouvée, plusieurs graphiques sont générés :

FIGURE 3 – Fitness au cours du temps

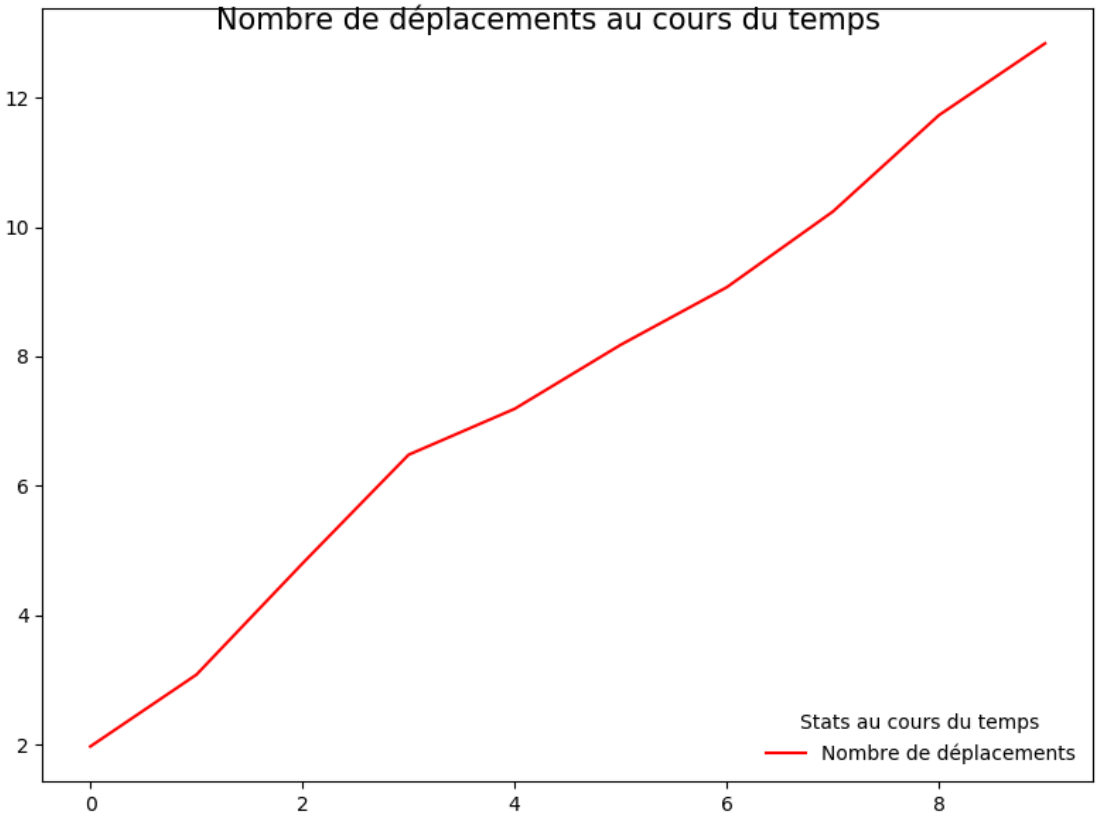

FIGURE 4 – Nombre de déplacements au cours du temps

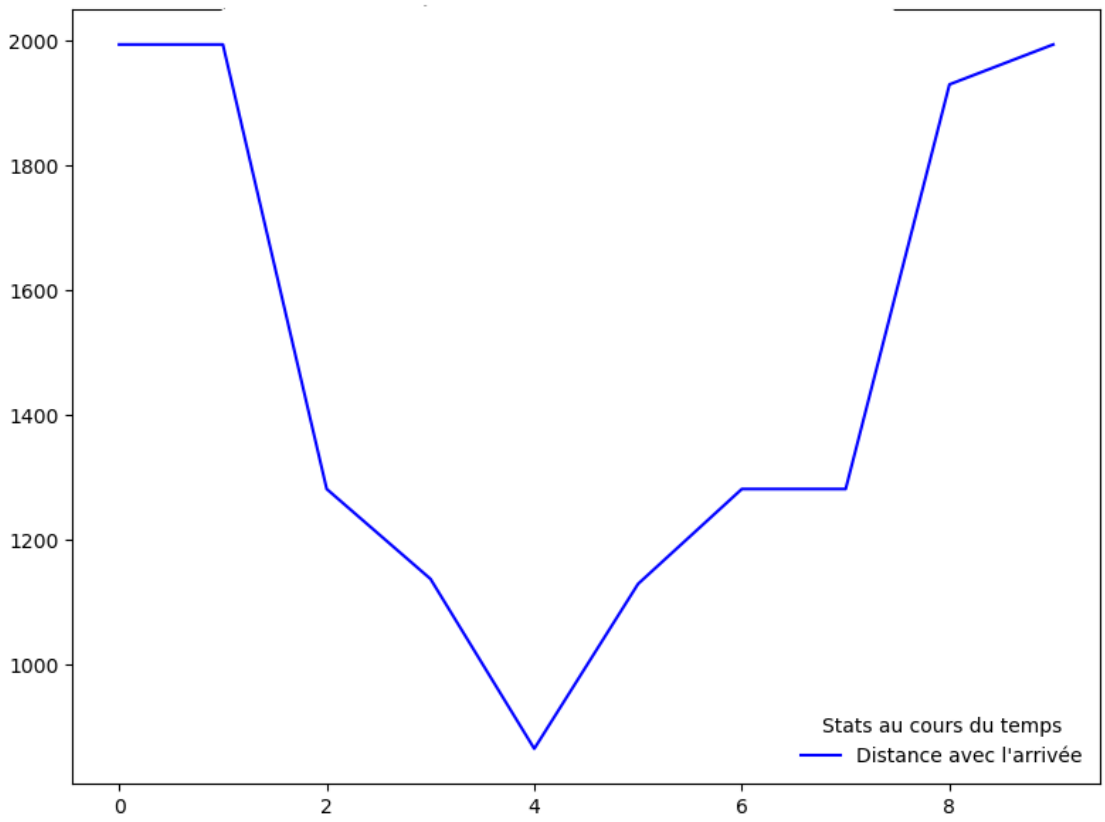

FIGURE 5 – Distance avec l'arrivée

2 Annexe : le code

Le code source est aussi disponible ici : `https://github.com/AlexandreBarde/TheMazeRunner`