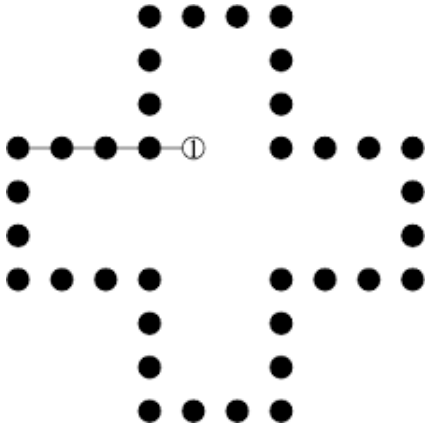
SOLITAIRE MORPION EN Q-LEARNING

June 14, 2020



PROJET DE SYNTHESE
M1 ISC

Auteurs :
Anis MEZRAG

Tuteur
Denis ROBILLIARD

Contents

Part I
Remerciements

Tout d'abord, il apparaît opportun d'adresser nos remerciements à tous ceux qui nous ont aidés pour la réalisation de ce projet, notamment notre encadrant Monsieur Denis ROBILLIARD, auprès duquel nous avons pu bénéficier d'un grand soutien.

1 Introduction

1.1 Principe du jeu

Le morpion solitaire est un casse-tête se pratiquant seul. Inspiré du morpion. Il se joue avec papier et crayon. Il consiste à tracer un maximum de lignes de 4 ou 5 éléments en partant d'une figure en forme de croix grecque.

1.2 Présentation du projet

Le projet consiste à réaliser un algorithme en python qui peut détecter toutes les lignes possibles d'être jouées à chaque partie. le premier objectif est d'arriver à le faire jouer aléatoirement pour ensuite bon le brancher sur un réseau de neurones artificiels. Le but est de faire tourner l'algorithme pour optimiser son score grâce à l'apprentissage par renforcement.

1.3 Règles du jeu

Configuration de départ avec une croix grecque formée par 36 points.

Exemple d'une partie de Morpion solitaire, 5 points en ligne, après 3 coups. Le jeu se joue sur une grille (de taille supposée illimitée). La configuration de départ comporte un ensemble de points (ou de petites croix) déjà tracés sur cette grille ; les configurations les plus courantes placent ces points en forme de croix grecque dont le côté comporte 3 points (24 au total) ou 4 points (36 au total).

À chaque coup, le joueur place un nouveau point sur la grille de façon à former un alignement de quatre ou cinq points adjacents horizontalement, verticalement ou en diagonale (le nombre de points dans un alignement est décidé avant la partie, selon la configuration initiale, et reste fixe tout au long de celle-ci) il relie ensuite cet alignement d'un trait de crayon.

Suivant la règle choisie, un alignement peut ou ne peut pas être placé dans le prolongement d'un alignement précédent. Si le joueur a cette possibilité, les deux alignements ne peuvent avoir qu'un point en commun.

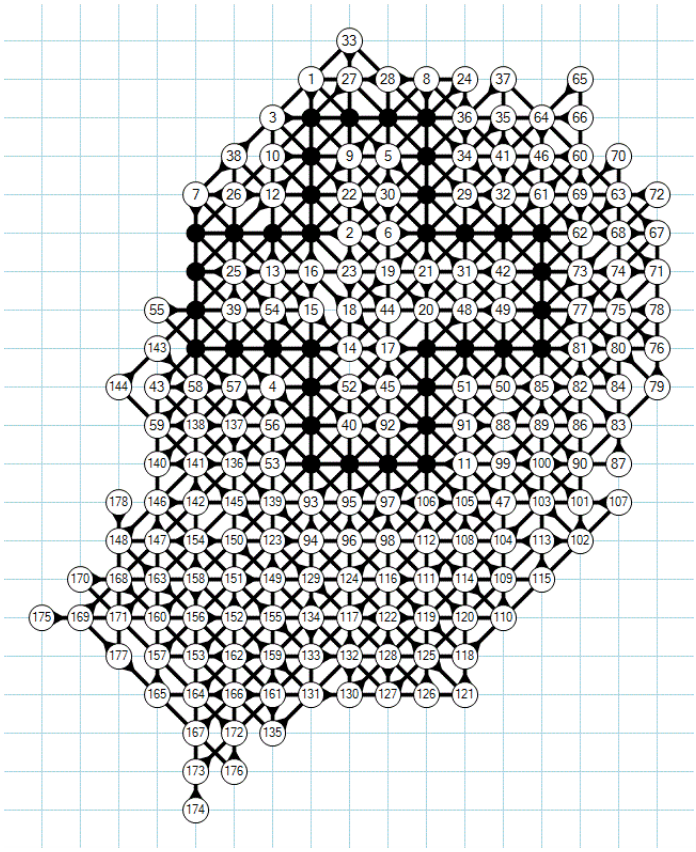
Le but du jeu est de placer le plus possible de points avant d'atteindre une situation où aucun nouveau point ne peut être placé. [6]

1.4 Etude comparative avec les solutions existantes

1.4.1 Meilleur score de Chris Rosin

Le dernier record, le 12 août 2011, avec 178 coups ! Et à nouveau de Chris Rosin, un coup de mieux que son précédent record annoncé seulement trois mois

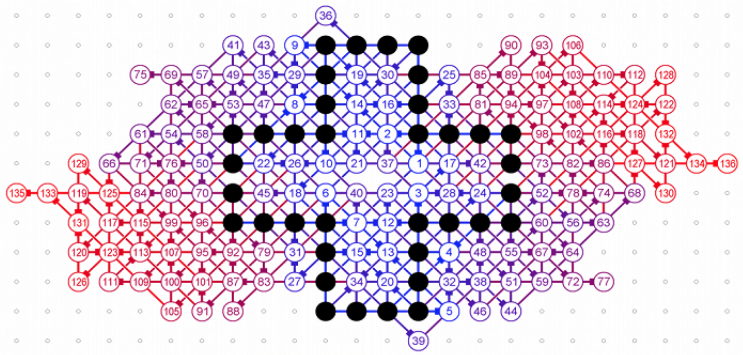
plus tôt, en utilisant une version améliorée de la méthode décrite dans son article IJCAI. Nouveau record immédiatement annoncé par le site de Pour La Science, aussi par Tangente N143 de nov-déc 2011. Chris vit à San Diego, Californie depuis 1992, et est un des cofondateurs de Parity Computing où il est Chief Algorithmist. Sa page personnelle est www.chrisrosin.com [2]



[!h]

1.4.2 Analysant les grilles symétriques, Michael Quist

Analysant par ordinateur les grilles symétriques, Michael Quist, USA, ont trouvé cette excellente et très belle grille de 136 coups, donc seulement 10 coups de moins que le record ci-dessus ! Il pense que cela devrait être le score maximum de toutes les grilles 5Ts symétriques (symétriques par rapport au centre comme celle-ci, mais aussi diagonalement symétriques, horizontalement symétrique, rotation de 90, ...) [1]

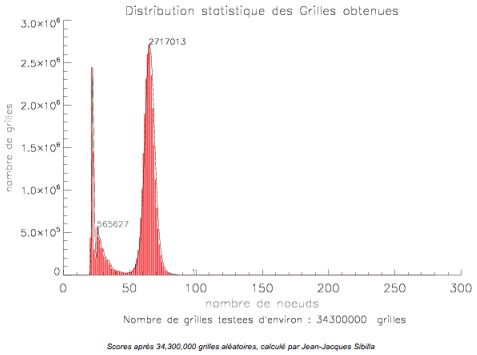


Avril 2008 : grille symétrique de 136 coups obtenue par ordinateur par Michael Quist

[!h]

1.4.3 Etude de Jean-Jacques Sibilla

Jean-Jacques Sibilla, IPGP (http://www.ipgp.fr/ sibilla), effectue une intéressante recherche: depuis 2007, son ordinateur a généré deux milliards de grilles (statut en octobre 2013), où les coups sont joués au hasard. Ses principaux résultats :
- Extrêmement difficile d'obtenir des grilles de plus de 100 coups, sa meilleure grille ayant 102 coupsdeux pics à 21 et 64 coups
- Impossible d'obtenir moins de 20 coups, comme mentionné dans les pires grilles possibles [2]



Scores après 34,300,000 grilles aléatoires, calculé par Jean-Jacques Sibilla

[!h]

1.5 Intérèt du jeu

Le problème de trouver de longues séquences au morpion solitaire peutÈtre approchés de plusieurs façons. Dans le cadre de notre travail, deux de ces approches sont particulièrement intéressantes : les méthodes d'apprentissage par renforcement, et les méthodes aléatoires basées sur des recherches complètes des lignes jouables et de jouer aléatoirement une après l'autre.Certains des algorithmes que nous présentons cherchent les séquencesà partir de la position initiale. D'autres servent à faire des recherches complètes par cellule. Une utilité de ces algorithmes, cependant, est de chercher à améliorer les de bonnes séquences déjà trouvées. [3]

Part II
Conception et environement de développement

2 Environnement logiciel

2.1 Outils de conception et développement

2.1.1 PyCharm

PyCharm est un environnement de développement intégré utilisé pour programmer en Python. Il permet l'analyse de code et contient un débogueur graphique. Il permet également la gestion des tests unitaires, l'intégration de logiciel de gestion de versions. [7]

Les avantages de ce logiciel sont nombreux notamment la gratuité et l'utilisation sur plusieurs plates-formes comme Unix, Windows ou encore Mac Os. Toutes les bibliothèques Python sont disponibles en un clique. C'est un outil puissant et complet, particulièrement adapté pour la mise en œuvre informatique des méthodes d'intelligence artificielle.

2.1.2 Anaconda

Anaconda est une distribution libre et open source des langages de programmation Python et R appliqué au développement d'applications dédiées à la science des données et à l'apprentissage automatique, qui vise à simplifier la gestion des paquets et de déploiement. Les versions de paquetages sont gérées par le système de gestion de paquets conda5. La distribution Anaconda est utilisée par plus de 6 millions d'utilisateurs et comprend plus de 250 paquets populaires en science des données adaptés pour Windows, Linux et MacOS. [4]

2.2 Choix technologiques

2.2.1 Python

Pour le langage de programmation de l'algorithme on est partis sur du Python parce que, il est à la fois un langage de programmation interprété, multi-paradigme et multiplate-forme. Il favorise la programmation impérative structurée, fonctionnelle et orientée objet. [8] Il est également apprécié parce qu'on y trouve un langage où la syntaxe, clairement séparée des mécanismes de bas niveau, permet une initiation aisée aux concepts de base de la programmation.

2.2.2 LaTeX

Pour la rédaction des documents en relation avec ce projet, on est partis sur du Latex. Il permet de rédiger des documents dont la mise en page est réalisée automatiquement en se conformant du mieux possible à des normes typographiques. Une fonctionnalité distinctive de Latex est son mode mathématique, qui permet de composer des formules complexes. Latex particulièrement utilisé dans les domaines techniques et scientifiques pour la production de documents de taille moyenne (tels que des articles) ou importants (thèses ou livres, par exemple). Néanmoins, il peut être employé pour générer des documents de types très variés. [5]

3 Conception

3.1 Début du projet

3.1.1 Les premiers pas

Quand on a débuté le projet on a commencé par étudier le jeu Snake, qui était lui, déjà branché sur le réseau de neurones et parfaitement fonctionnel. On a aussi essayé de comprendre, de façon générale, ce qu'est le Q learning ou l'apprentissage par renforcement.

Cela nous a beaucoup aidés à la compréhension du jeu. On a compris qu'il fallait détecter l'ensemble des états qui change au fur à mesure que le jeu avance. On a eu notamment de rencontre avec notre tuteur qui nous a aidés à plus comprendre la problématique de chaque projet pour ensuite choisir lequel nous convenait. On a choisi le jeu solitaire morpion.

3.1.2 Avancement dans les versions

La première version du projet a été de faire un moteur calculant toutes les cellules jouable et comment elles sont jouées l'inconvénient de cette version c'était que le code avait plus de 1700 lignes avec affichage en console la compilation été très lente. une version plus élaborée a été ensuite envisagé par la suppression de l'affichage en console en le remplaçant par une interface graphique beaucoup qui procure un meilleur affichage une vitesse de compilation plus rapide.

Après plusieurs échanges avec notre tuteur, une autre version a été réalisé cette fois-ci en transformant les 1700 lignes en moins de 200. Ce c'est grâce raccourcie offerte par le langage python.

3.2 Conception du reseau de neurones artificiels

3.2.1 Déscription du concept Q-Learning

En intelligence artificielle, plus précisément en apprentissage automatique, le Q-learning est une technique d'apprentissage par renforcement. Cette technique ne nécessite aucun modèle initial de l'environnement. La lettre 'Q' désigne la fonction qui mesure la qualité d'une action exécutée dans un état donné du système. Un des points forts du Q-learning est qu'il permet de comparer les récompenses probables de prendre les actions accessibles sans avoir de connaissance initiale de l'environnement. En d'autres termes, bien que le système soit modélisé par un processus de décision markovien (fini), l'agent qui apprend ne le connait pas et l'algorithme du Q-learning ne l'utilise pas. [9]

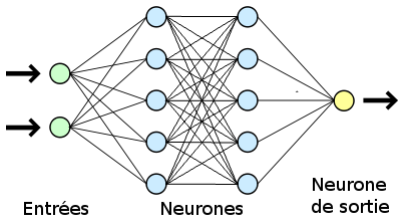


Figure 1: Exemple d'un réseau de neurones artificiels

3.2.2 Implémentation du Q-Learning dans notre projet

On s'inspirant du jeu snake, on a compris que le premier objectif était de trouver des éléments répétitifs qui décrive au mieux l'état du jeu. On est partis sur différentes approches pour définir ces éléments. On a essayé de travailler sur la liste des lignes jouables, en ordonnant leurs jouabilités. On cherchait la ligne qui créera le plus de lignes jouables et on donnait une indication de sa position par rapport au centre, mais sans résultat d'apprentissage concret parce que le jeu était dépendant et déterministe.

Ensuite, on a changé d'approche. On a divisé le carré un jeu en trois, et o donnant dans les états une indication par les coordonnées polaires, dans quelles des trois parties se trouve le plus de ligne jouable, mais en vain.

Enfin, la dernière approche qui a abouti à des résultats concrets était de diminuer le déterminisme du jeu, en créant un joueur avec une distance de liberté (Son implémentation et fonctionnement sont expliqués dans la partie Développement 4.4). On changeant constamment les états, on est tombé sur une formule où le jeu commene par des coups aléatoires en donnant des résultats médiocres et au bout de 50 parties, le joueur commence à voir les dangés et faire les bons mouvements pour jouer de plus en plus de lignes.

Part III
Développement et fonctionnalitées

4 Développement

4.1 Algorythme

L'algorithme principal du jeu cite de parcourir toutes les cellules du jeu à la recherche de ligne jouable. toutes les lignes jouables peuvent être détectées à partir de la première cellule et ceci la recherche 24 cellules alignées cliquer aimant jouer sur la direction aligner. compte cette condition est vérifiée, La ligne est ajoutée au répertoire des lignes jouables avec toutes les cellules de la ligne et la direction de cette dernière.

4.2 Moteur du jeu

4.2.1 La classe Cellule

La classe Cellule représente une case du jeu. elle contient sept attributs, cinq d'entre eux sont des booléens. Les quatre premiers c'est pour dire si la case a été déjà joué verticalement, horizontalement, diagonal gauche ou en diagonale droite. La septième boule en qui est "clique" nous informe si la case et cliquer ou pas. Quant aux deux derniers, ils indiquent les coordonnées (x, y) de la case ou d'une cellule.

```
22
23  class Game:
24      """ La classe Game contient le repértoire des cellules et lignes et lignes jouables """
25      def __init__(self, taille):
26          self.rep_lines = [] # Initialisation liste des lignes jouées
27          self.taille = taille # taille du jeu taille * taille
28          # Initialisation du tableau des cellules
29          self.rep_cellules = [[Cellule(0, 0, 0, 0, 0, i, j) for i in range(taille)] for j in range(taille)]
30          self.grille_depart() # Mise à jour des cellules de departs
31          self.rep_playable_lines = [] # Initialisation liste des ligne jouables
32          self.crash = False  # s'il reste aucune cellule jouable
33          self.padding = 5 # marge du jeu
34          self.Croix = []  # Liste des coordonnés cellules departs
35
```

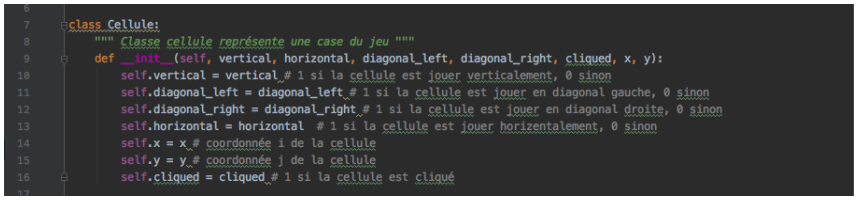
Figure 2: Code de la classe Cellule

Cette classe contient une méthode get attribute by index et cette méthode fermée de retourner la valeur d'un attribut de la classe cellule avec son indice.

4.2.2 La classe Game

La classe gamme a comme attribut "taille", celui-ci représente le nombre de case ou de cellules d'un côté du carré du jeu. dans notre cas on a utilisé une

taille qui est égale à 30 qui nous permet de créer donc neuf cents cellules.



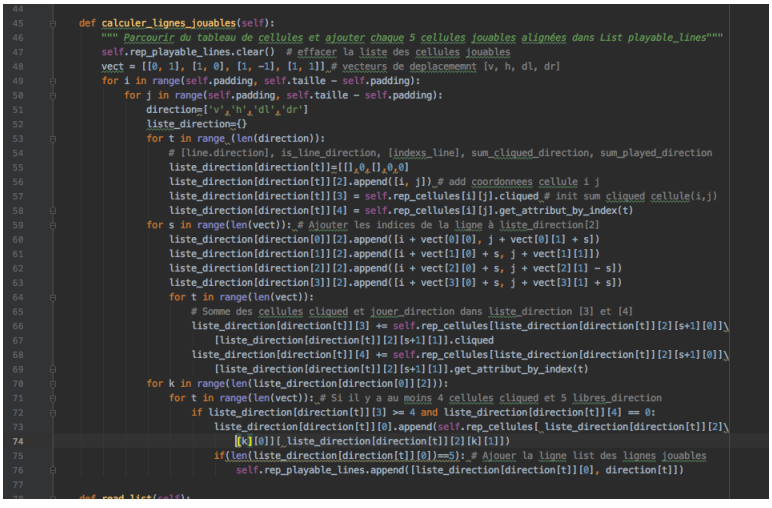
```python
class Cellule:
    """ Classe cellule représente une case du jeu """
    def __init__(self, vertical, horizontal, diagonal_left, diagonal_right, cliqued, x, y):
        self.vertical = vertical # 1 si la cellule est jouer verticalement, 0 sinon
        self.diagonal_left = diagonal_left # 1 si la cellule est jouer en diagonal gauche, 0 sinon
        self.diagonal_right = diagonal_right # 1 si la cellule est jouer en diagonal droite, 0 sinon
        self.horizontal = horizontal  # 1 si la cellule est jouer horizontalement, 0 sinon
        self.x = x # coordonnée i de la cellule
        self.y = y # coordonnée j de la cellule
        self.cliqued = cliqued # 1 si la cellule est cliqué
```

Figure 3: Code de la classe Game

la méthode "grille-départ" permet d'initialiser les premières cellules du jeu (la croix grecque), en les mettant comme cliquées pour ainsi les differenciers des autres cases.

4.2.3 La méthode de détéction des lignes jouables

la méthode "calculait lignes jouables", une des méthodes les plus importantes du jeu. Elle commence par initialiser la liste du répertoire des lignes jouables. Ensuite on a créé une liste de quatre vecteurs qui résument le changement d'indice dans les quatre directions principales, soit vertical, horizontal, diagonal gauche et diagonale droite. Ensuite, on a fait deux boucles imbriquées qui parcourent toutes les cellules du jeu en faisant des conditions pour trouver au moins quatre dans cinq (4/5) cellules qui sont alignées, cliquées et non joué dans la direction de celle-ci. Enfin, on les rajoute dans la liste des lignes jouables qui sera utilisée ensuite pour vérifier si une cellule est jouable ou pas.



```python
    def calculer_lignes_jouables(self):
        """ Parcourir du tableau de cellules et ajouter chaque 5 cellules jouables alignées dans List playable_lines"""
        self.rep_playable_lines.clear()  # effacer la liste des cellules jouables
        vect = [[0, 1], [1, 0], [1, -1], [1, 1]] # vecteurs de deplacememnt [v, h, dl, dr]
        for i in range(self.padding, self.taille - self.padding):
            for j in range(self.padding, self.taille - self.padding):
                direction=['v','h','dl','dr']
                liste_direction={}
                for t in range (len(direction)):
                    # [line.direction], is_line_direction, [indexs_line], sum_cliqued_direction, sum_played_direction
                    liste_direction[direction[t]]=[[],0,[],0,0]
                    liste_direction[direction[t]][2].append([i, j]) # add coordonnees cellule i j
                    liste_direction[direction[t]][3] = self.rep_cellules[i][j].cliqued # init sum cliqued cellule(i,j)
                    liste_direction[direction[t]][4] = self.rep_cellules[i][j].get_attribut_by_index(t)
                for s in range(len(vect)): # Ajouter les indices de la ligne à liste_direction[2]
                    liste_direction[direction[0]][2].append([i + vect[0][0], j + vect[0][1] + s])
                    liste_direction[direction[1]][2].append([i + vect[1][0] + s, j + vect[1][1]])
                    liste_direction[direction[2]][2].append([i + vect[2][0] + s, j + vect[2][1] - s])
                    liste_direction[direction[3]][2].append([i + vect[3][0] + s, j + vect[3][1] + s])
                    for t in range(len(vect)):
                        # Somme des cellules cliqued et jouer_direction dans liste_direction [3] et [4]
                        liste_direction[direction[t]][3] += self.rep_cellules[liste_direction[direction[t]][2][s+1][0]]\
                            [liste_direction[direction[t]][2][s+1][1]].cliqued
                        liste_direction[direction[t]][4] += self.rep_cellules[liste_direction[direction[t]][2][s+1][0]]\
                            [liste_direction[direction[t]][2][s+1][1]].get_attribut_by_index(t)
                for k in range(len(liste_direction[direction[0]][2])):
                    for t in range(len(vect)): # Si il y a au moins 4 cellules cliqued et 5 libres_direction
                        if liste_direction[direction[t]][3] >= 4 and liste_direction[direction[t]][4] == 0:
                            liste_direction[direction[t]][0].append(self.rep_cellules[_liste_direction[direction[t]][2]\
                                [k][0]][_liste_direction[direction[t]][2][k][1]])
                            if(len(liste_direction[direction[t]][0])==5): # Ajouer la ligne list des lignes jouables
                                self.rep_playable_lines.append([liste_direction[direction[t]][0], direction[t]])

    def read_list(self):
```

Figure 4: Code de la méthode qui calcule les lignes jouables

4.2.4 La méthode pour jouer une ligne

'Jouer-ligne' est une méthode qui prend comme paramètre une ligne. En premier, il vérifie si elle n'est pas vide. l'attribut ligné est une liste qui contient deux composantes. La première contient les cinq cellules de la ligne. La deuxième, par contre contient la direction dans laquelle elle est jouable.

4.2.5 La méthode 'Main'

La méthode 'main' est la méthode est le point d'entrée de l'algorithme. C'est l'endroit où le contrôle du programme commence et se termine. Les premières lignes mais cette méthode permettait d'initialiser les paramètres du jeu c'est-à-dire la taille du jeu, ainsi que l'initialisation de l'interface graphique.

```
def main():
    print("running...")
    taille = 30 # Nombre de cellules = taille * taille
    game = Game(taille)  # Initialisation du jeu
    init_ihm(game) # Initialization IHM
    max_iteration = 1 # Nombre de parties à jouer
    cpt_iteration = 0 # Compteur de parties
    rep_lines_bestScore, score_plot, counter_plot = [],[],[]  # scores, numéros de parties, lignes du meilleur score
    while (cpt_iteration < max_iteration):
        game = Game(taille)  # Initialisation du jeu
        game.calculer_lignes_jouables()  # Calculer les lignes possibles à jouées
        while not game.crash:  # Tant qu'il reste encore des lignes possible à jouer
            # Choisir aléatoirement un indice i tq, 0 < i < nbr de lignes jouables
            line=play_best_line(game)
            game.jouer_ligne(line)  # Ajouter line à la liste des lignes jouées
            game.rep_lines.append(line)
            game.calculer_lignes_jouables()  # Recalcule des lignes jouables
            if (len(game.rep_playable_lines) == 0):
                game.crash=True
        if cpt_iteration == 0 or len(game.rep_lines) > max(score_plot):  # Si c'est le meilleur score atteint
            rep_lines_bestScore = game.rep_lines  # Sauvegarder la liste des lignes du meilleur score
        score_plot.append(len(game.rep_lines))  # Ajouter le score à la liste des scores
        cpt_iteration+=1
        print(cpt_iteration, "-Score: ", len(game.rep_lines), " -  Best score: ", max(score_plot))
        counter_plot.append(cpt_iteration) # ajouter le numéro de la partie
    afficher_lignes_ihm(rep_lines_bestScore,max_iteration,max(score_plot))
    plot_seaborn(counter_plot, score_plot)
```

Figure 5: Code de la méthode Main

On initialise aussi le nombre de parties maximum et le compteur de parties, la liste des lignes du meilleur score, les meilleurs scores et les parties. Cependant, tant que le compteur de parties n'a pas atteint le maximum, on initialise la classe game. Ensuite, on calcule les lignes jouables. Après, on choisit ligne au hasard parmi les lignes jouables qui nous servira d'indication pour l'agent dans les états.

4.3 Interface graphique

L'interface graphique, contient aux totales cinq méthodes principales. Ces dernières permettront de bien visualiser tous les éléments du jeu, que sa soient les lignes créées à la fin ainsi que deux compteurs, un pour le nombre de parties jouées et l'autre pour indiquer le meilleur score. L'interface permet aussi

l'affichage de l'évolution du jeu avec le score de chaque partie joué et ceci sur deux volets diffients avec l'affichage des lignes.

4.3.1 Méthode 'constructeur'

Cette méthode permet de dessiner toutes les cases du jeu et de mettre en gras les cellules de départ qui sont cliquées.

4.3.2 Méthode 'init-ihm'

'init-ihm' prends comme attribut gamme, permet de dessiner le carré du jeu ainsi que les deux petits compteurs, le nombre de parties est le score maximum atteint.

4.3.3 Méthode 'afficher-ligne-ihm'

Cette méthode est appelée à la fin du jeu pour dessiner toutes les lignes créées lorsque le jeu a atteint son meilleur score. elle prend comme attribut une liste des lignes créées, le nombre de parties jouées ainsi que le meilleur score.

4.3.4 La méthode 'plot-seaborn'

plot-seaborn est une méthode qui va nous permettre, à la fin du jeu, de représenter chaque score selon le nombre de parties jouées. Ainsi dessiner une droite selon la méthode des moindres carrées pour visualiser l'évolution à chaque partie.

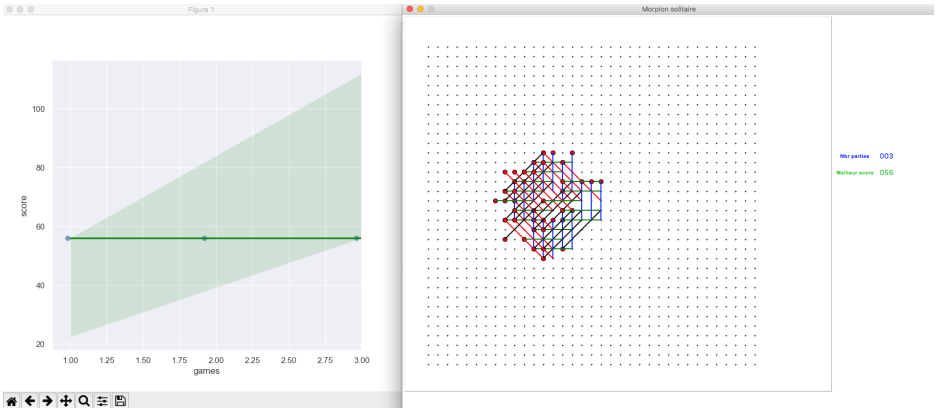


Figure 6: L'interface graphique - Exemple avec 3 parties jouées

4.4 Branchement au réseau

Le réseau de neurones artificiels qu'on a appelé 'agent' est contenu dans le fichier "DQN Py'. Ce programme est longuement inspiré de celui du programme Snake avec lequel on a commencé ce projet. Le réseau est composé d'une couche de 80 entrées, une sortie de 8 bits ainsi que deux couches intermédiares.

```
def network(self, weights=None):
    model = Sequential()
    model.add(Dense(output_dim=120, activation='relu', input_dim=71))
    model.add(Dropout(0.15))
    model.add(Dense(output_dim=120, activation='relu'))
    model.add(Dropout(0.15))
    model.add(Dense(output_dim=120, activation='relu'))
    model.add(Dropout(0.15))
    model.add(Dense(output_dim=120, activation='relu'))
    model.add(Dropout(0.15))
    model.add(Dense(output_dim=8, activation='softmax'))
    opt = Adam(self.learning_rate)
    model.compile(loss='mse', optimizer=opt)

    if weights:
        model.load_weights(weights)
    return model
```

Figure 7: Code de la création des couches du réseau

On a ajouté au programme un joueur, qui représente un point qui se déplace d'une distance d'un dans huit directions différentes dans le but de trouver une cellule qui représente le début d'une ligne jouable. Le joueur ou le point de déplacement est doté avec des coordonnées x et y (x-player, y-player).

Le degré de liberté du joueur est indiqué dans la classe gamme et ceci indique le nombre de mouvements maximum qu'on lui donne la possibilité de faire selon une sortie donnée, pour trouver une ligne jouable sinon il va crasher. L'agent contient un attribut epsilon qui lui indique aussi sa liberté et ceci en lui donnant la possibilité, jusqu'à la partie 120 (Choisis d'après les meilleurs scores) d'avoir des mouvements un peu aléatoires, ainsi il pourra découvrir des chemins qui n'auraient pas pu explorer avec juste une prédiction et son enpeèche le jeu d'être déterministe.

Les états du jeu sont choisis de façon à donner le maximum d'informations à l'agent. Il y a 80 États, dans les 60 premiers servent à indiquer la position du joueur.

Exemple: (x-player,y-player)=(3,4) Cela est équivalent dans les états à [0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, 0,0,0,0,0,1,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]

La logique est la suivante, les 30 premiers bits indiquent le x du joueur (x-player) et du 30e bit jusqu'à 60e, il indique la position d' y du joueur(y-player).

```
def get_state(self,game):
    state=[]
    direction=[]
    vect_squar=[[-1,-1],[-1,0],[-1,1],[0,1],[1,1],[1,0],[1,-1],[0,-1]]

    for i in range(len(vect_squar)):
        direction.append(False)
    for i in range(len(game.rep_playable_lines)):
        for j in range(game.liberte):
            if (game.rep_playable_lines[i][0][0].x==game.x_player+vect_squar[i][0]*j and \
                game.rep_playable_lines[i][0][0].y==game.y_player+vect_squar[i][1]*j):
                direction[i]=True
    for i in range(71):
        state.append(False)
    state[game.x_player]=True # x du joueur
    state[game.y_player+30]=True # y du joueur
    state[60]=game.choosed_line[0][0].x<game.x_player # position de la ligne jouable par rapport au joueur
    state[61]=game.choosed_line[0][0].y<game.x_player # x du joueur
    state[62]= direction7
    state[63]= direction8
    for i in range(len(vect_squar)):
        state[64+i]=direction[i] # s'il y a une cellule jouable autour
    state[70]=game.cpt_liberte>game.liberte-3 # danger
    for i in range(len(state)):
        if state[i]:
            state[i]=1
        else:
            state[i]=0
    return np.asarray(state)
```

Figure 8: Codage des états

Les bits de 60 à 64 permettent de données une indication de position d'une cellule (début d'une ligne) jouable choisie aléatoirement à chaque partie, par rapport au joueur.

Les bits du 64e au 72e, on indique, s'i existe une ligne jouable autour de la position du joueur.

Du 72e au 79e bits e sortie, on mentionnent le dérniers mouvemement réalisé par le joueur dans l'une des huits directions qui lui sont possibles.

Le dernier bit d'entrée sert à indiquer un danger de crash. Il est mis à un quand le joueur n'a pas trouvé de ligne jouable et lui reste deux mouvements (2 distances de libérées) avant de crasher.

D'après les pas du jeu on aura une prédiction qui est une sortie de huit bits, dans un seul, égale à un. Cela permettra d'indiquer la direction pour le joueur.

Exemple: Une sortie du réseau = [1,0,0,0,0,0,0,0]

Cela est équivalent à appliquer le vecteur [-1, -1] aux coordonnées du joueur (x-player, y-player), ce qu'il veut dire d'aller en diagonale droite vers l'arrière avec une distance de 1.
Le joueur perd, s'il fait plus de dix mouvements est trouvée une ligne jouable. À chaque crash, on vérifie si le score atteint est le meilleur depuis le début du jeu, on enregistre les lignes jouées dans une liste qu'on utilisera à la fin pour l'affichage.

Part IV
Réalisations

5 Étude des réalisations

5.1 Mode aléatoire

5.1.1 Spécification fonctionnelle

En utilisant la fonction "random" qu'offre Python pour la génération de nombres aléatoires, on produit un nombre qui sera l'indice d'une ligne dans la liste des lignes jouables. Cela des fois, avec des tests entre 10 parties et 100 parties peut nous conduire en erreur en remarquant parfois des améliorations au fur et à mesure quelles parties se succèdent. On a démontré le contraire en faisant des tests avec des grands nombres de parties (figure 3 et 4). Ceci, nous montre bien une ligne constante qui est adéquate avec le mode aléatoire.

5.1.2 Testes

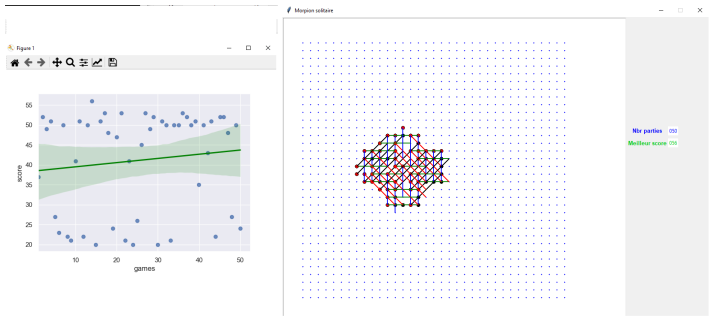


Figure 9: Résultat teste aléatoire - 50 parties

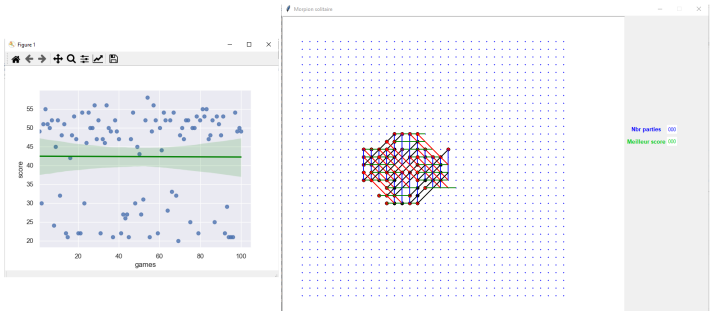


Figure 10: Teste aléatoire - 100 parties

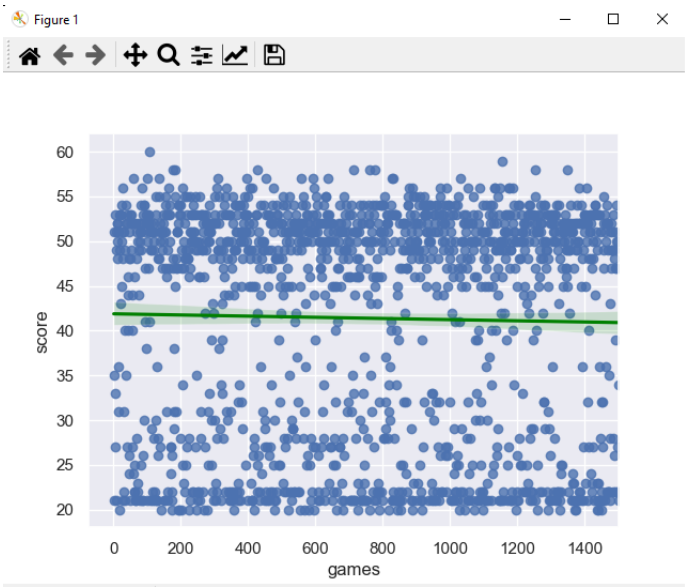


Figure 11: Résultat teste aléatoire - 1000 parties

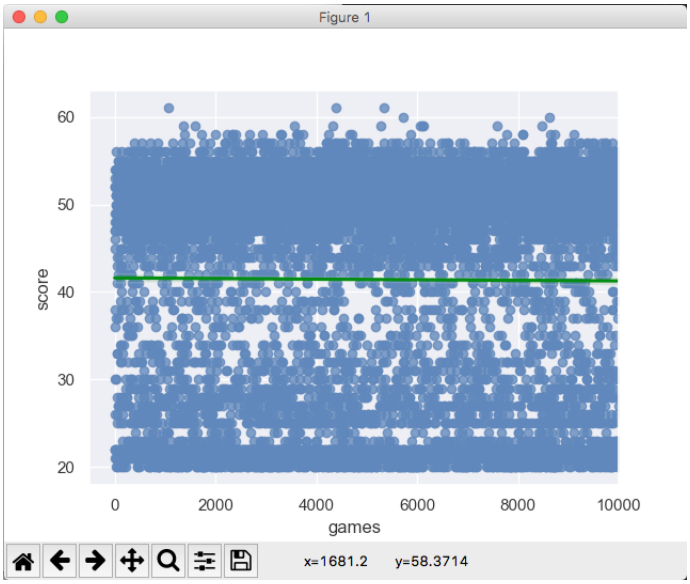


Figure 12: Résultats teste aléatoire - 10 000 parties

5.2 Mode apprentissage

5.2.1 Spécification fonctionnelle

Dans les figures 8 et 9, on voit bien la progression du jeu. On commence à voir une vrai amélioration qu'après la 100e partie. Les premières parties, le joueur commence par découvrir le terrain et grace aux indications et son expérience, il commence à distinguer les mouvements dans lesquels il accumule le plus de récompense. Le meilleur score atteint après 400 parties était de 48 lignes. En remarque que l'apprentissage se fait mais ça n'atteint pas des scores plus importants que le mode aléatoire.

5.2.2 Testes

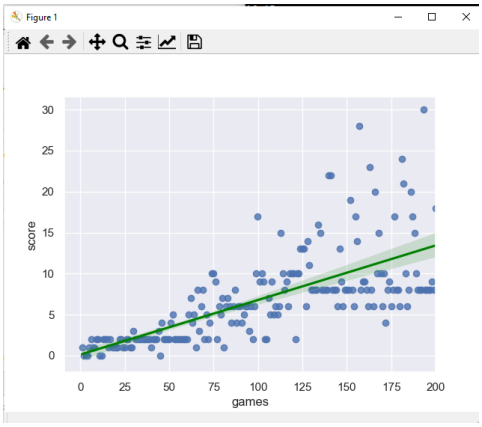


Figure 13: Résultat teste Q-Learning - 200 parties

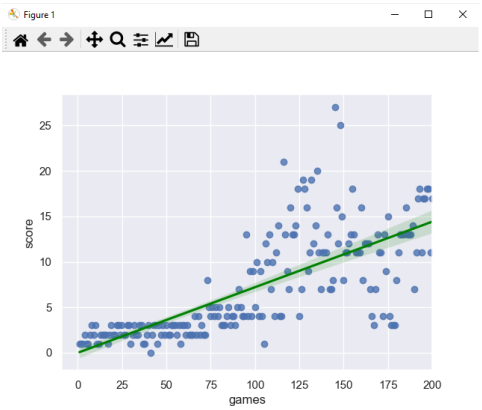


Figure 14: Résultat teste Q-Learning - 200 parties

5.3 Meilleurs parties mode aléatoire et apprentissage

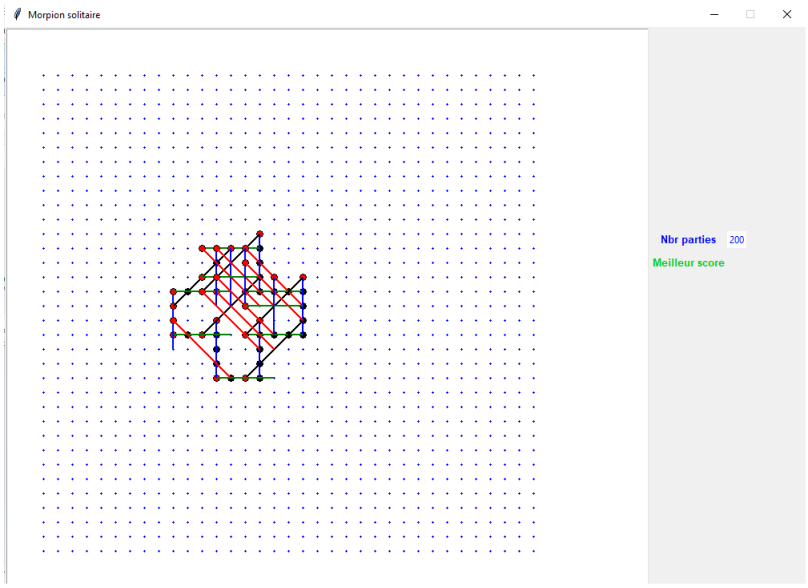


Figure 15: l'une des meilleure partie en mode Q-Learning - 200 parties

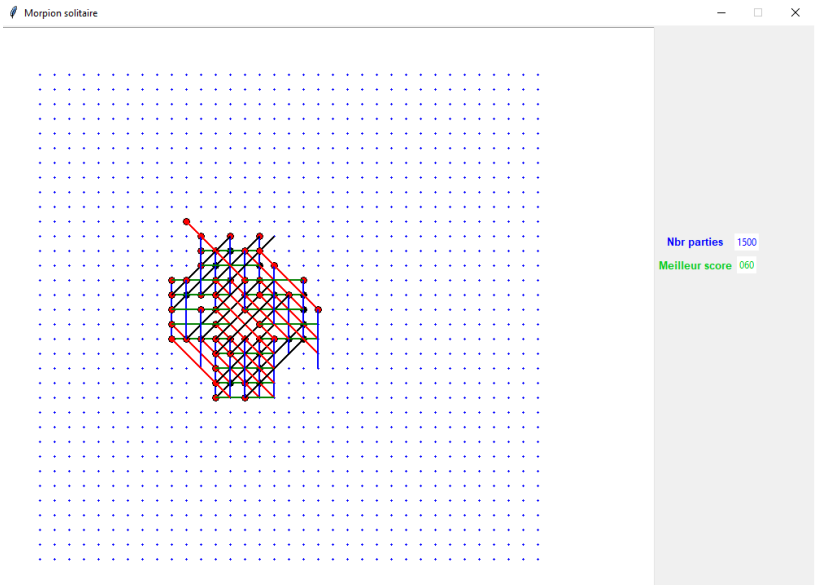


Figure 16: Meilleure partie en mode aléatoire - 1000 parties

6 Conclusion

Dans le cadre de notre projet de synthèse, nous avons conçu et développé un algorithme du jeu Solitaire Morpion d'apprentissage par renforcement dans le but d'optimiser son score. Notre solution offre un code propre et une interface ergonomiques, simples et cohérentes tout en assurant la facilité d'utilisation pour une préalable équipe de recherche souhaitant pousser la recherche pour un meilleur score. Le présent rapport couvre l'étude, la conception et l'implémentation de la solution.

Dans un premier lieu, nous avons commencé par étudier le contexte général du jeu. Ensuite, nous avons préparé un planning de travail en respectant les priorités de nos besoins. Cependant, on a commencé l'étude d'une solution apportée au jeu Snake, qui pouvait être similaire à la nôtre.

Tout au long de cette période, nous avons consacré plus de temps à optimiser le code et apprendre le fonctionnement d'un réseau de neurones artificiels. En effet, nous avons rencontré des difficultés à configuré l'environnement Tensorflow, à trouver des créneaux libres de rencontre avec notre tuteur ainsi que la crise sanitaire actuelle qui n'a pas arrangé non plus la situation. En outre, nous avons passé beaucoup de temps à coder l'algorithme. L'apport de ce travail a été d'une importance très considérable puisqu'il nous a permis: de suivre une méthode de travail bien étudié, d'approfondir mes connaissances dans le domaine de l'intelligence artificielle et de bien maîtriser les raccourcis qu'offre le langage Python. Au niveau de l'aspect technique, ce projet a été d'abord d'assurer un code algorithmique optimisé du jeu et ce grâce aux différents échanges avec notre tuteur Denis ROBILLIARD qui nous a fait bénéficier de son expérience malgré les problèmes rencontrer pendant dans cette crise. En effet, prendre en charge un tel projet dans ce cadre nous permet assurément de développer notre esprit d'analyse, de réflexion et de décision.

Finalement, notre travail n'est pas achevé à ce niveau. Nous visons, d'optimiser le réseau de neurones par des nouveaux testés qui vont déterminés le nombre de couche auquel l'algorithme est sensible. Enfin, nous pouvons étendre l'algorithme par l'ajout de nouvelles optimisations que nous n'avons pas considérées lors de la première étude. Par exemple, nous pouvons ajouter une autre intelligence artificielle qui elle prendra en compte les meilleures parties atteintes dans ce jeu.

References

[1] Christian Boyer. les grilles symétriques, michael quist, 2020.

[2] Christian Boyer. Morpion solitaire - grilles records (jeu 5t), 2020.

[3] Bernard Helmstetter. Analyses de dépendances et méthodes de monte-carlo dans les jeux de réflexion, 2020.

[4] wikipedia. Anaconda, 2020.

[5] wikipedia. Latex, 2020.

[6] wikipedia. Morpion_solitaire, 2020.

[7] wikipedia. Pycharm, 2020.

[8] wikipedia. Python, 2020.

[9] wikipedia. Q-learning, 2020.