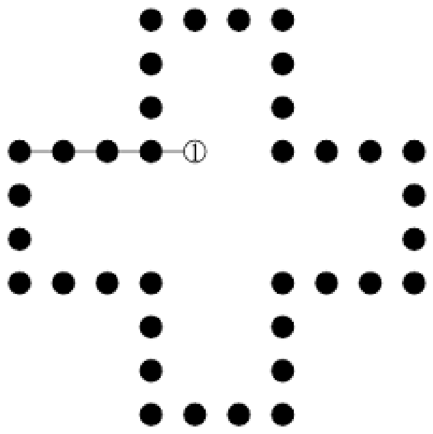
SOLITAIRE MORPION IN Q-LEARNING

June 14, 2020



SYNTHESIS PROJECT
M1 ISC

Author :
Anis MEZRAG

Tutor
Denis ROBILLIARD

Contents

Part I
Thanks

First of all, it seems appropriate to express our thanks to all those who helped us in the realisation of this project, in particular our supervisor Mr Denis ROBILLIARD, from whom we were able to benefit from a great deal of support.

1 Introduction

1.1 Principle of the game

The Morpion Solitaire is a puzzle that can be solved alone. Inspired by Morpion. It is played with paper and pencil. It consists in drawing a maximum of lines of 4 or 5 elements starting from a figure in the shape of a Greek cross.

1.2 Presentation of the project

The project consists in realizing a python algorithm that can detect all the possible lines to be played in each game. The first objective is to make it play randomly and then to connect it to an artificial neural network. The goal is to run the algorithm to optimize its score thanks to reinforcement learning.

1.3 Rules of the game

Starting configuration with a Greek cross formed by 36 points.

Example of a solitary Morpion game, 5 points in a row, after 3 moves. The game is played on a grid (supposedly of unlimited size). The starting configuration consists of a set of stitches (or small crosses) already drawn on this grid; the most common configurations place these stitches in the shape of a Greek cross whose side has 3 stitches (24 in total) or 4 stitches (36 in total).

With each move, the player places a new point on the grid so as to form a row of four or five adjacent points horizontally, vertically or diagonally (the number of points in a row is decided before the game, according to the initial configuration, and remains fixed throughout the game) and then connects this row with a pencil stroke.

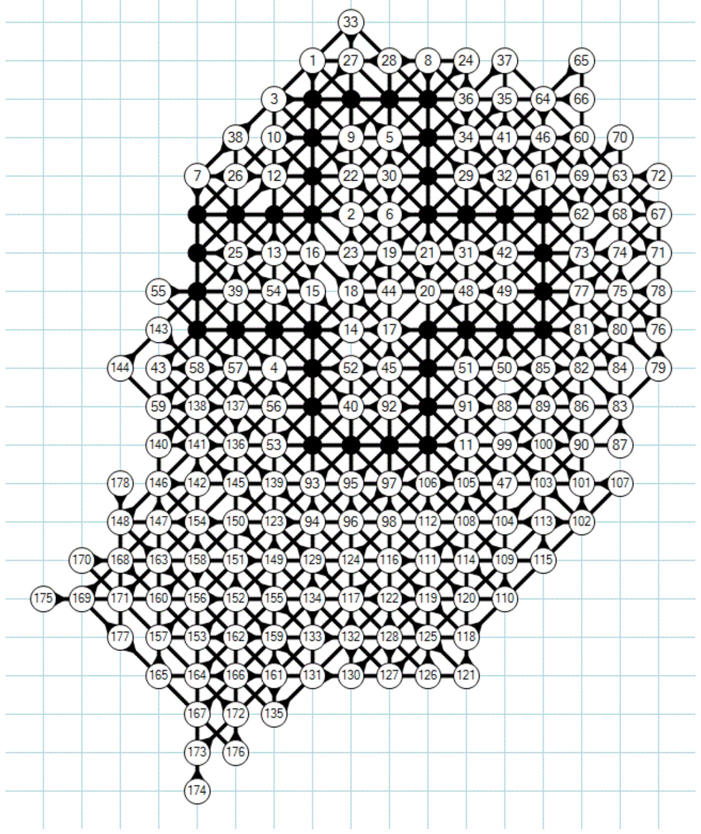
Depending on the chosen rule, a line-up may or may not be placed as an extension of a previous line-up. If the player has this possibility, the two lines of play can only have one point in common.

The object of the game is to place as many points as possible before reaching a situation where no new points can be placed. [6]

1.4 Comparative study with existing solutions

1.4.1 Highest score by Chris Rosin

The last record, on August 12, 2011, with 178 strokes! And again from Chris Rosin, a step up from his previous record announced only three months earlier, using an improved version of the method described in its article IJ-CAI. New record immediately announced by the Pour La Science website, also by Tangent N143 of Nov-Dec 2011. Chris lives in San Diego, California since 1992, and is a co-founder of Parity. Computing where he is Chief Algorithmist. His personal page is Translated with www.DeepL.com/Translator (free version) www.chrisrosin.com [2]

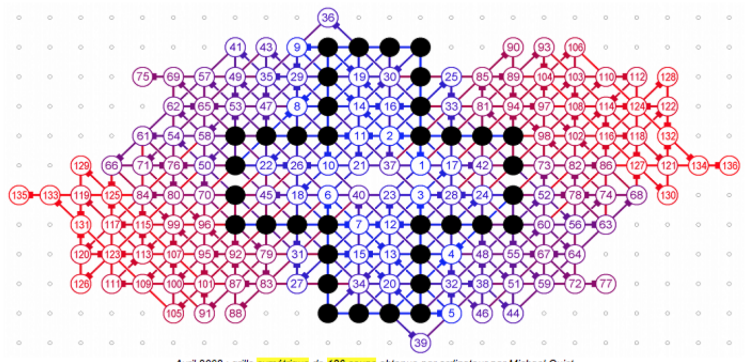


1.4.2 Analyzing symmetrical grids, Michael Quist

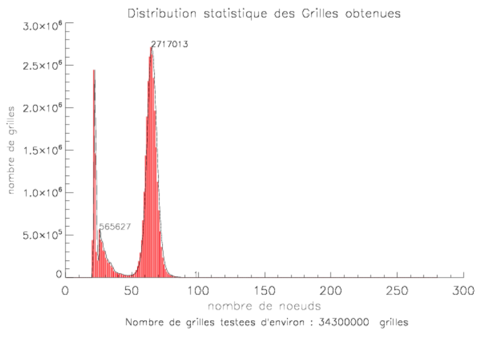
Analyzing by computer the symmetrical grids, Michael Quist, USA, found this excellent and very beautiful grid of 136 moves, so only 10 moves less than the above record! He thinks this should be the maximum score of all symmetrical 5Ts grids (symmetrical with respect to the center like this one, but also diagonally symmetrical, horizontally symmetrical, 90 rotation, ...). [1]

1.4.3 Study by Jean-Jacques Sibilla

Jean-Jacques Sibilla, IPGP (http://www.ipgp.fr/ sibilla), is carrying out an interesting research: since 2007, her computer has generated two billion grids (status as of October 2013), where the moves are played randomly. Its main results:


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

- Extremely difficult to obtain grids of more than 100 strokes, its best grid with 102 hits-two peaks at 21 and 64 hits
- Impossible to get less than 20 shots, as mentioned in the worst possible grids [2]



1.5 Interest of the game

The problem of finding long sequences of solitary tic-tac-toe can be approached in several ways. In our work, two of these approaches are of particular interest: reinforcement learning methods, and random methods based on complete searches of playable lines and playing randomly one after the other.Some of the algorithms we present search for sequences from the initial position. Some of the algorithms we present search for sequences from the initial position, while others are used to perform complete cell searches. One usefulness of these algorithms, however, is that they can be used to improve on good sequences that have already been found. [3]

Part II
Design and development environment

2 Software environment

2.1 Design and development tools

2.1.1 PyCharm

PyCharm is an integrated development environment used for programming in Python. It allows code analysis and contains a graphics debugger. It also allows the management of tests integration of version management software, the integration of unitary. [7]

The advantages of this software are numerous, in particular the free of charge and its use on several platforms such as Unix, Windows or Mac Os. All Python libraries are available in one click. It is a powerful and complete tool, particularly adapted for the computer implementation of artificial intelligence methods.

2.1.2 Anaconda

Anaconda is a free and open source distribution of the languages of Python and R programming applied to application development dedicated to data science and machine learning, which aims to simplify package management and deployment. Package versions are managed by the conda5 packages. The Anaconda distribution is used by more than 6 million users and includes more than 250 popular packages in data science suitable for Windows, Linux and MacOS. [4]

2.2 Technological choices

2.2.1 Python

For the programming language of the algorithm we started with Python because it is an interpreted, multi-paradigm and multi-platform programming language. It promotes structured, functional and object-oriented imperative programming. [8] It is also appreciated because it provides a language where the syntax, clearly separated from low-level mechanisms, allows an easy introduction to the basic concepts of programming.

2.2.2 LaTeX

For the drafting of the documents related to this project, we started on Latex. It allows us to write documents whose layout is done automatically, conforming as much as possible to typographical standards. A distinguishing feature of latex is its mathematical mode, which allows complex formulas to be composed. Latex is particularly used in the technical and scientific fields for the production of medium-sized documents (such as articles) or important documents (theses or books, for example). Nevertheless, it can be used to generate a wide variety of document types. [5]

3 Conception

3.1 Beginning of the project

3.1.1 The first steps

When we started the project we started by studying the game Snake, which was it, already connected to the neural network and perfectly functional. We also tried to understand, in a general way, what is Q learning or learning by reinforcement.

It helped us a lot in understanding the game. We understood that we had to detect all the states that change as the game progresses. We had a meeting with our tutor who helped us to better understand the problems of each project and then to choose the one that suited us best. We chose the solitaire game Morpion.

3.1.2 Progress in versions

The first version of the project was to make an engine calculating all the playable cells and how they are played the disadvantage of this version was that the code had more than 1700 lines with console display the compilation was very slow. a more elaborate version was then considered by removing the console display by replacing it with a much graphical interface that provides a better display a faster compilation speed.

After several exchanges with our tutor, another version has was achieved this time by transforming the 1700 lines into less than 200, thanks to the shortcut offered by the python language.

3.2 Artificial neural network design

3.2.1 Description of the Q-Learning concept

In artificial intelligence, more precisely in automatic learning, Q-learning is a learning technique by reinforcement. This technique does not require any initial model of the environment. The letter 'Q' designates the function that measures the quality of an action executed in a given state of the system. One of the strong points of Q-learning is that it allows you to compare the likely rewards of taking accessible actions without any initial knowledge of the environment. In other words, although the system is modelled by a Markovian (finite) decision process, the learning agent does not know it and the Q-learning algorithm does not use it. [9]

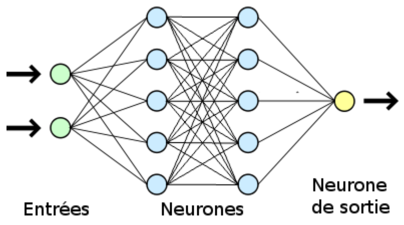


Figure 1: Example of an artificial neural network

3.2.2 Implementation of Q-Learning in our project

Taking inspiration from the snake game, we understood that the first objective was to find repetitive elements that best describe the state of the game. We started with different approaches to define these elements. We tried to work on the list of playable lines, ordering their playability. We looked for the line that would create the most playable lines and gave an indication of its position in relation to the centre, but without any concrete learning result because the game was dependent and deterministic.

Then we changed our approach. We divided the square one game into three, and o giving in the states an indication by polar coordinates, in which of the three games there is the most playable line, but in vain.

Finally, the last approach that led to concrete results was to decrease the determinism of the game, by creating a player with a distance of freedom (Its implementation and functioning are explained in Development 4.4). We are constantly changing states, we came across a formula where the game starts with random moves giving poor results and after 50 games, the player starts to see the dangés and make the right moves to play more and more lines.

Part III
Development and functionalities

4 Development

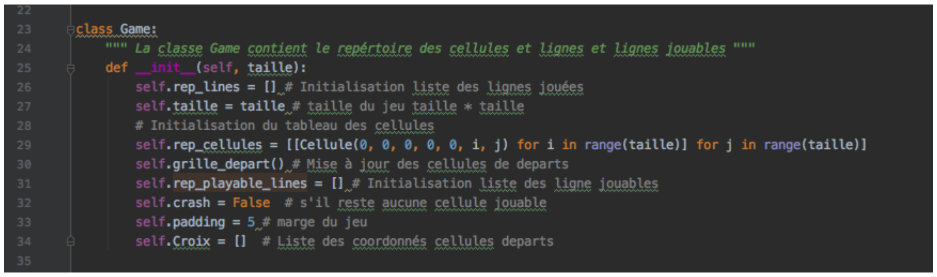
4.1 Algorithm

The main algorithm of the game quotes to scan all the cells of the game looking for playable lines. all playable lines can be detected from the first cell and this searches for 24 cells aligned click play magnet on the direction align. counts this condition is verified, the line is added to the directory of playable lines with all the cells of the line and the direction of the line.

4.2 Game Engine

4.2.1 Class Cellule

The Cell class represents a cell in the game. It contains seven attributes, five of which are Booleans. The first four are to say whether the square has already been played vertically, horizontally, diagonally left or diagonally right. The seventh ball that is "clicked" tells us whether the square and clicked or not. As for the last two, they indicate the coordinates (x, y) of the square or a cell.



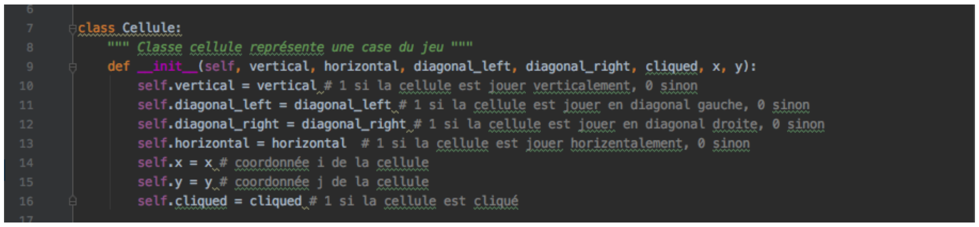
```
class Game:
    """ La classe Game contient le repértoire des cellules et lignes et lignes jouables """
    def __init__(self, taille):
        self.rep_lines = [] # Initialisation liste des lignes jouées
        self.taille = taille # taille du jeu taille * taille
        # Initialisation du tableau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

Figure 2: Code of the class Cellule

This class contains a method get attribute by index and this closed method to return the value of an attribute of the class cell with its index.

4.2.2 La classe Game

The class range has as attribute "size", this one represents the number of square or cells on one side of the square of the game. In our case we used a size which is equal to 30 which allows us to create therefore nine hundred cells.



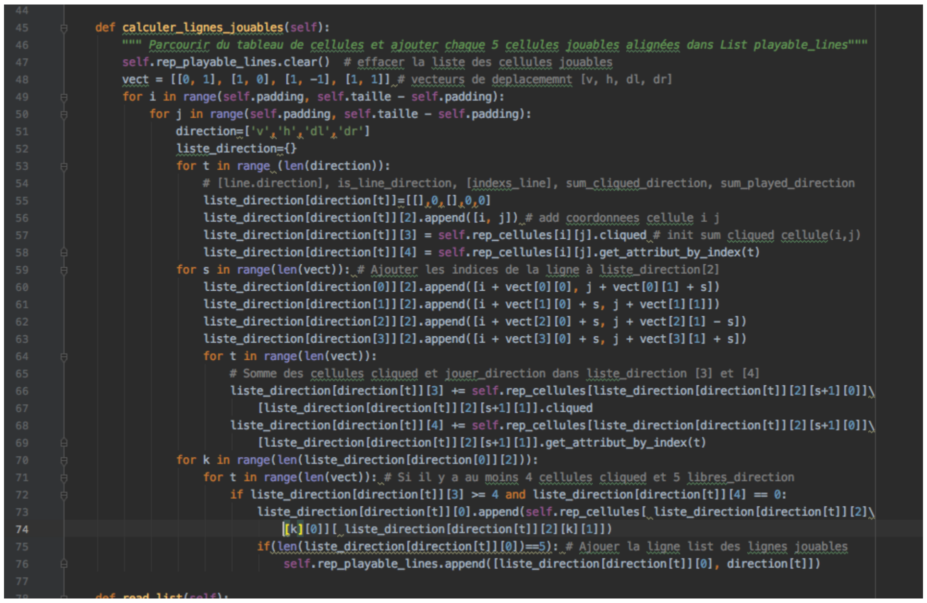
```python
class Cellule:
    """ Classe cellule représente une case du jeu """
    def __init__(self, vertical, horizontal, diagonal_left, diagonal_right, cliqued, x, y):
        self.vertical = vertical # 1 si la cellule est jouer verticalement, 0 sinon
        self.diagonal_left = diagonal_left # 1 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 horizentalement, 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 of the class Game

the method ”starting grid” allows to initialize the first cells of the game (the Greek cross), by putting them as clicked so that the differentiators of the other squares.

4.2.3 The method of detecting playable lines

Algorithm It starts by initializing the list of the directory of playable lines. Then we created a list of four vectors that summarize the index change in the four main directions, i.e. vertical, horizontal, left diagonal and right diagonal. Next, we made two nested loops that run through all the cells of the game making conditions to find at least four in five (4/5) cells that are aligned, clicked and not played in the direction of the cell. Finally, they are added to the list of playable rows, which will then be used to check whether a cell is playable or not.



```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]])
```

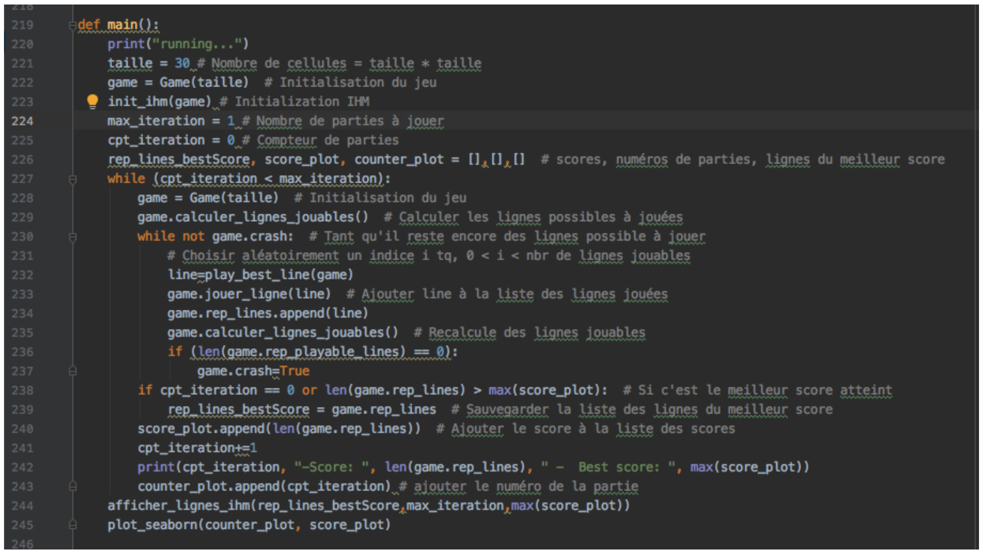
Figure 4: Code of the method that calculates playable lines

4.2.4 The method for playing a line

'Jouer-ligne' is a method that takes a line as a parameter. First, it checks if it is not empty. The line attribute is a list that contains two components. The first contains the five cells of the row. The second, on the other hand, contains the direction in which it is playable.

4.2.5 The method 'Main'

The 'main' method is the method is the entry point of the algorithm. It is where control of the program begins and ends. The first lines but this method was used to initialize the parameters of the game i.e. the size of the game, as well as the initialization of the GUI.



```python
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 of the method Main

You also initialize the maximum number of games and the game counter, the list of high score lines, high scores and games. However, as long as the game counter has not reached the maximum, the game class is initialized. Then the playable lines are calculated. Afterwards, we choose randomly among the playable lines that will serve as an indication for the agent in the states.

4.3 Graphical user interface

The graphical user interface contains a total of five main methods. These will allow you to visualize all the elements of the game, including the lines created at the end and two counters, one for the number of games played and the other to indicate the best score. The interface also allows the display of the evolution of the game with the score of each game played and this on two different panes with the display of the lines.

4.3.1 Method 'constructeur'

This method makes it possible to draw all the squares of the game and to put in bold the starting cells that are clicked.

4.3.2 Mehtod 'init-ihm'

'init-ihm' takes as scale attribute, allows to draw the square of the game as well as the two small counters, the number of games is the maximum score reached.

4.3.3 Méthode 'afficher-ligne-ihm'

This method is called at the end of the game to draw all the lines created when the game has reached its best score. It takes as an attribute a list of the lines created, the number of games played and the best score.

4.3.4 Method 'plot-seaborn'

plot-seaborn is a method that will allow us, at the end of the game, to represent each score according to the number of games played. Thus draw a line according to the method of the least squares to visualize the evolution at each game.

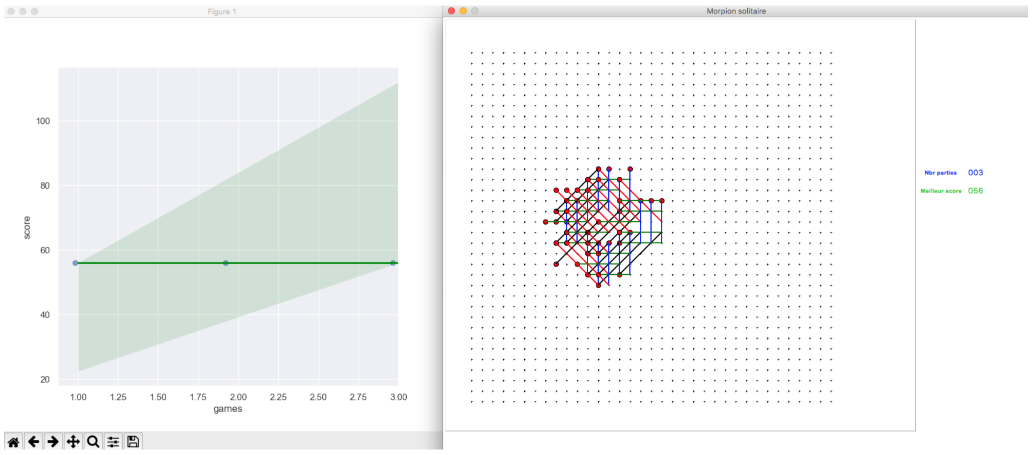
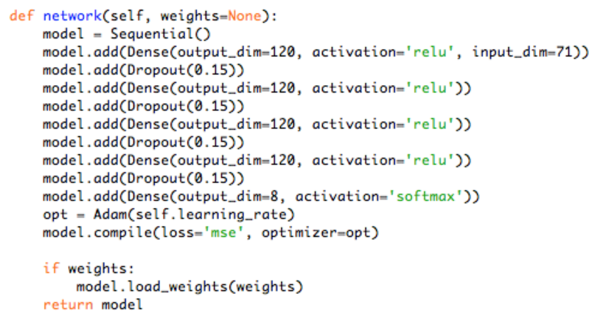


Figure 6: The graphical interface - Example with 3 games played

4.4 Network connection

The artificial neural network called 'agent' is contained in the 'DQN Py' file. This program is based for a long time on the Snake program with which we started this project. The network is composed of 3 layers, with 70 inputs and 8 outputs.



```
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: Network Layer Creation Code

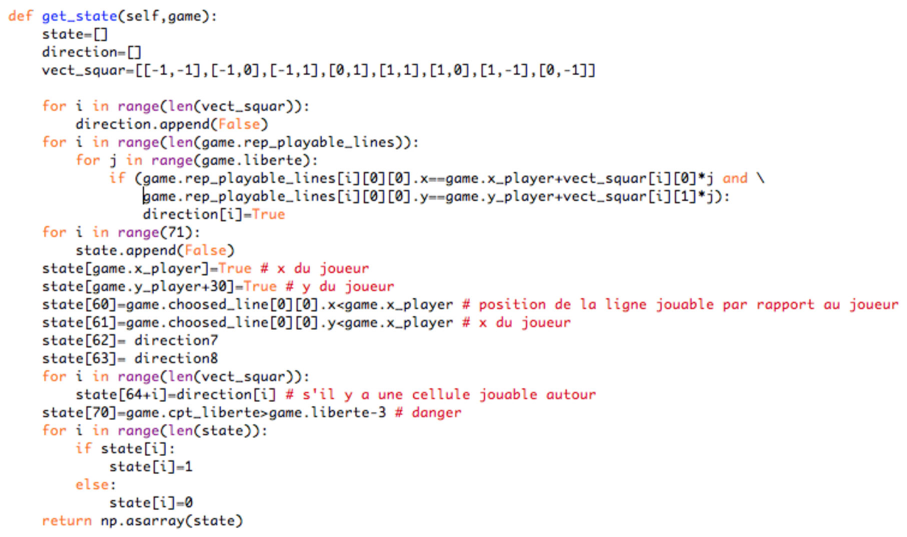
A player has been added to the program, representing a point that moves a distance of one in eight different directions in order to find a cell that represents the beginning of a playable line. The player or the moving point is provided with x and y coordinates (x-player, y-player).

The degree of freedom of the player is indicated in the range class and this indicates the maximum number of moves he is allowed to make according to a given output, to find a playable line otherwise he will crash. The agent contains an epsilon attribute that also indicates his freedom and this gives him the possibility, up to game 120 (chosen according to the best scores) to make random moves, so he can discover paths that could not have been explored with just a prediction and his game is deterministic.

The game states are chosen to give the agent as much information as possible. There are 80 states, in the first 60 are used to indicate the position of the player.

Exemple: (x-player,y-player)=(3,4) This is equivalent in the states to [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]

The logic is as follows, the first 30 bits indicate the x of the player (x-player) and from the 30th bit to the 60th bit, it indicates the y position of the player (y-player).



```python
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: Coding of states

The bits from 60 to 64 allow data an indication of the position of a playable cell (beginning of a line) randomly chosen at each game, in relation to the player.

The bits from 64th to 72nd indicate, if there is a playable line around the player's position.

From the 72nd to the 79th exit bit, we mention the last move made by the player in one of the eight possible directions.

The last input bit is used to indicate a crash hazard. It is set to one when the player has not found a playable line and has two moves left (2 clear distances) before crashing.
According to the steps of the game we will have a prediction that is an eight-bit output, in one, equal to one. This will indicate the direction for the player.

Exemple: An exit from the network = [1,0,0,0,0,0,0,0]

This is equivalent to applying the vector [-1, -1] to the player's coordinates (x-player, y-player), which means going diagonally right backwards with a distance of 1.
The player loses, if he makes more than ten moves is found a playable line. At each crash, we check if the score reached is the best since the beginning of the game, we save the played lines in a list that we will use at the end for the display.

Part IV
Achievements

5 Review of Achievements

5.1 Random mode

5.1.1 Functional Specification

By using the "random" function that Python offers for random number generation, we produce a number that will be the index of a line in the list of playable lines. This sometimes, with tests between 10 games and 100 games can lead us into error by sometimes noticing improvements as games follow one another. The opposite has been demonstrated by doing tests with large numbers of games (figure 3 and 4). This shows us a constant line that is adequate with the random mode.

5.1.2 Testes

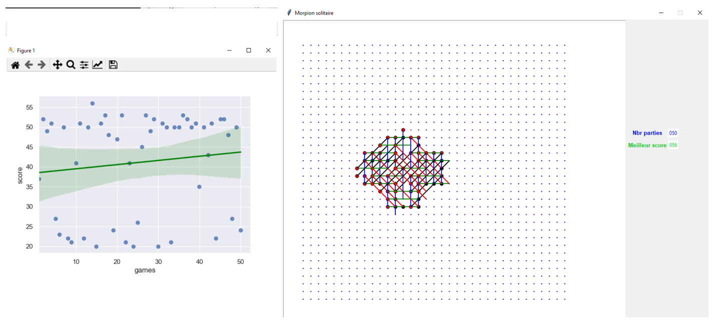


Figure 9: Random test result - 50 games

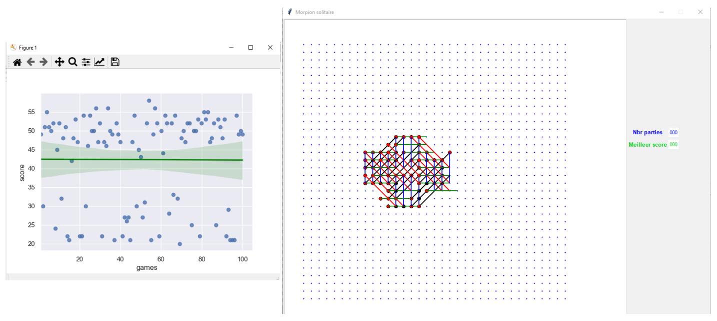


Figure 10: Random test result - 100 games

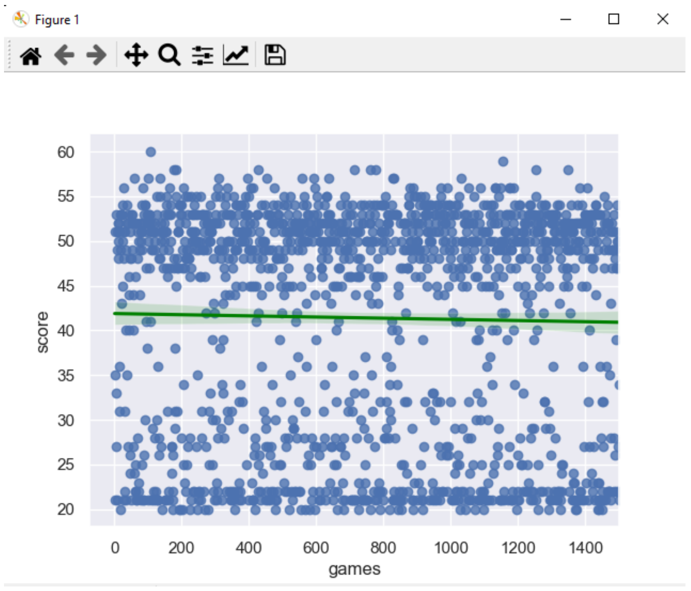


Figure 11: Random test result - 1000 games

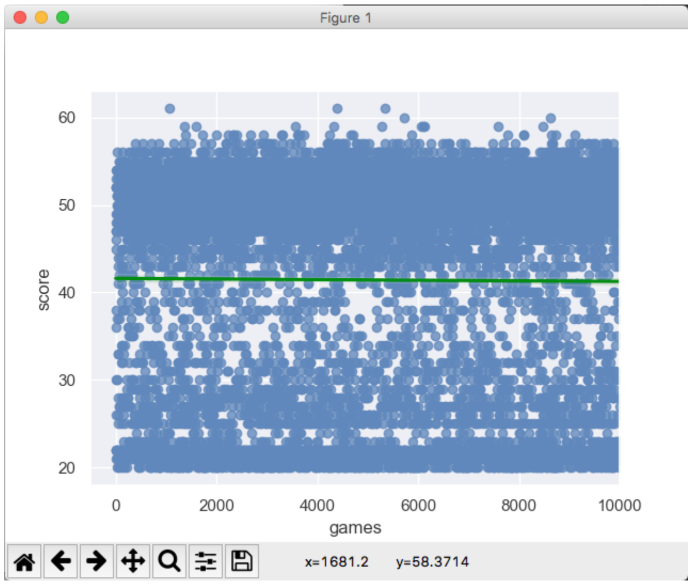


Figure 12: Random test result - 10 000 games

5.2 Learning mode

5.2.1 Functional Specification

In figures 8 and 9, we can see the progression of the game. We begin to see a real improvement only after the 100th game. In the first games, the player starts by discovering the field and thanks to the indications and his experience, he starts to distinguish the moves in which he accumulates the most reward. The best score reached after 400 games was 48 lines. In notice that the learning is done but it doesn't reach higher scores than the random mode.

5.2.2 Tests

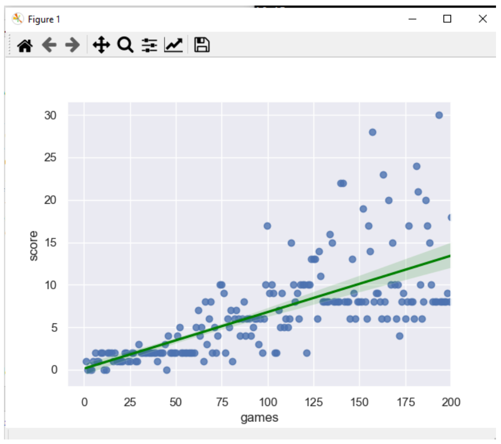


Figure 13: Result Q-Learning test - 200 games

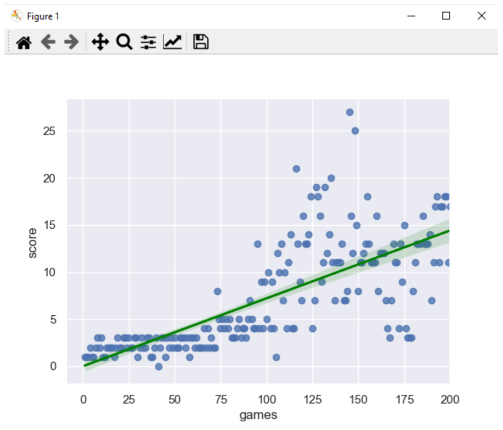


Figure 14: Result Q-Learning test - 200 games

5.3 Best random mode and learning parts

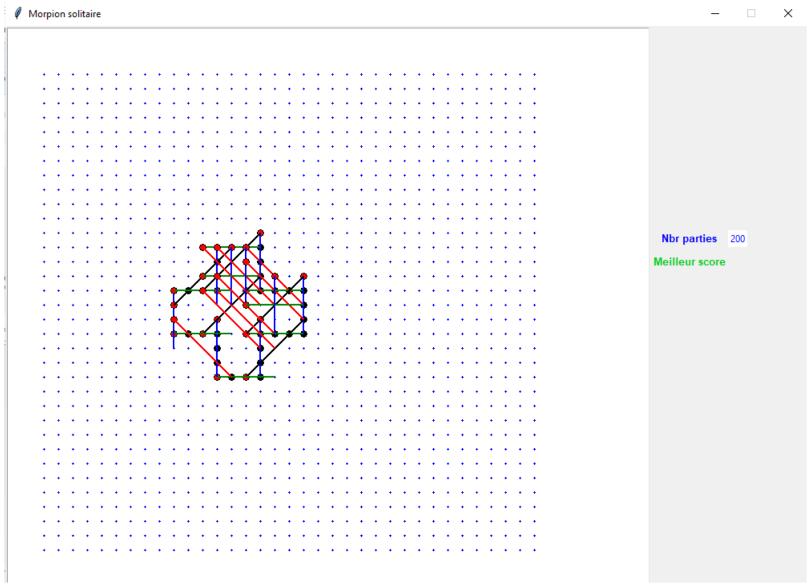


Figure 15: One of the best games in Q-Learning mode - 200 games

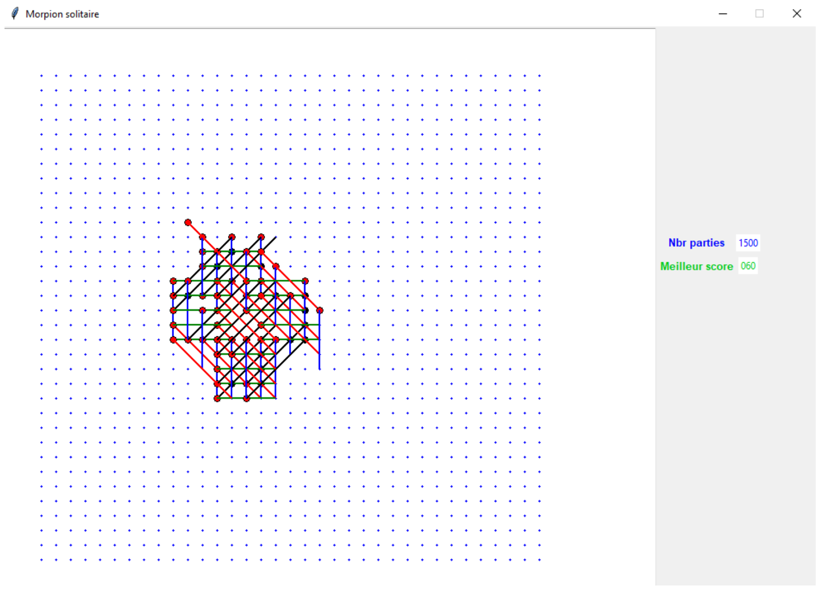


Figure 16: Best game in random mode - 1000 parties

6 Conclusion

As part of our synthesis project, we designed and developed an algorithm for the Solitaire Morpion reinforcement learning game in order to optimize its score. Our solution offers a clean code and an ergonomic, simple and coherent interface while ensuring ease of use for a preliminary research team wishing to push the research for a better score. This report covers the study, design and implementation of the solution.

First, we began by studying the general context of the game. Then, we prepared a work schedule according to the priorities of our needs. However, we started studying a solution for the game Snake, which could be similar to ours.

Throughout this period, we spent more time optimizing the code and learning how an artificial neural network works. Indeed, we encountered difficulties in configuring the Tensorflow environment, finding free slots to meet with our tutor and the current health crisis which did not help the situation either. In addition, we spent a lot of time coding the algorithm. The contribution of this work has been very important because it has allowed us: to follow a well-studied working method, to deepen my knowledge in the field of artificial intelligence and to master the shortcuts offered by the Python language. On the technical side, this project was first of all to ensure an optimized algorithmic code of the game and this thanks to various exchanges with our tutor Denis ROBILLIARD who gave us the benefit of his experience despite the problems encountered during this crisis. Indeed, taking charge of such a project in this context certainly allows us to develop our analytical, thinking and decision-making skills.

In the end, our work is not finished there. We aim at optimizing the neural network with new tests that will determine the number of layers to which the algorithm is sensitive. Finally, we can extend the algorithm by adding new optimizations that we did not consider in the first study. For example, we can add another artificial intelligence that will take into account the best games reached in this game.

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.