Faculté des sciences fondamentales

HEMDANE Mohand Said Nassim

Programmation avancée

Programmation d'un jeu d'échecs en Python

Avril 2021

Department de mathématiques et d'informatique

Introduction

Le jeu d'échecs, ou les échecs, est un jeu de société opposant deux joueurs de part et d'autre d'un tablier appelé échiquier composé de soixante-quatre cases, 32 claires et 32 sombres nommées les cases blanches et les cases noires. Les joueurs jouent à tour de rôle en déplaçant l'une de leurs seize pièces (ou deux pièces en cas de roque), claires pour le camp des blancs, sombres pour le camp des noirs. Chaque joueur possède au départ un roi, une dame, deux tours, deux fous, deux cavaliers et huit pions. Le but du jeu est d'infliger à son adversaire un échec et mat, une situation dans laquelle le roi d'un joueur est en prise sans qu'il soit possible d'y remédier.

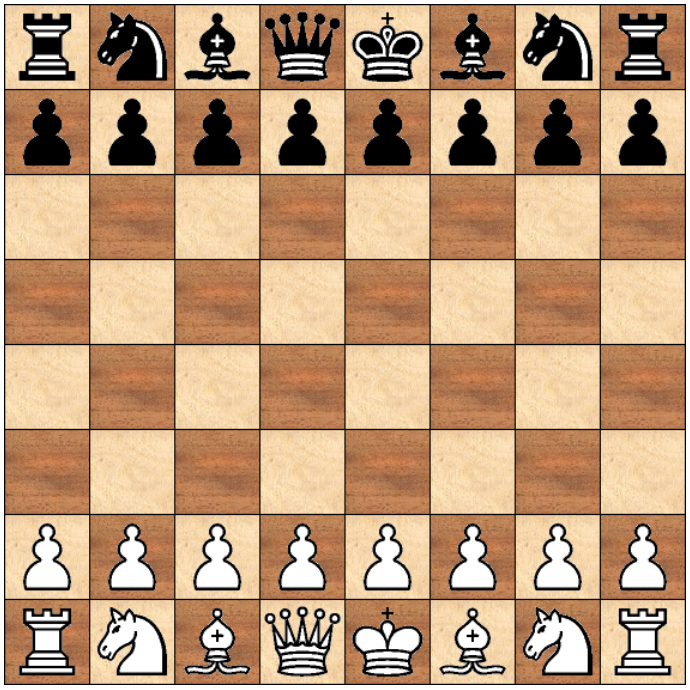

Figure 1: Jeu d'échécs

Dans le cadre de ce projet, j'ai implémenté une version simplifiée du jeu d'échecs (sans échecs et mat et sans promotions) en python.

Réalisation

J'ai réalisé ce projet en faisant de l'orienté objet. Voici un diagramme des classes représentant l'architecture que j'ai utilisé :

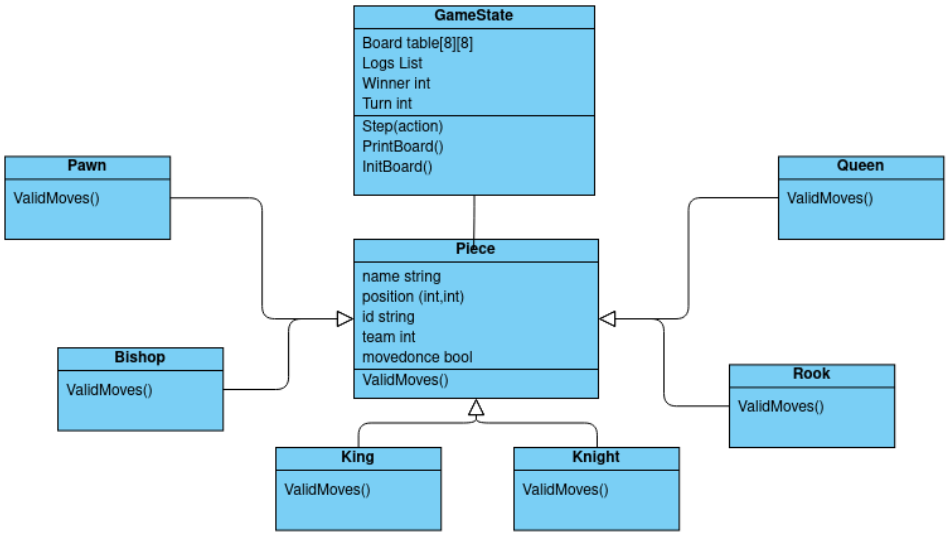

Figure 2: Diagramme des classes

J'ai d'abords créé une classe "GameState" qui décrit l'état du jeu et donc du board et j'ai implémenté une fonction "step" qui se charge de faire avancer le jeu en faisant un mouvement (tout en vérifiant que celui-ci est permit).

J'ai représenté le board avec une matrice 8*8 d'objects que j'initialise en plaçant les pièces au bon endroit (les cases vides sont représentées par des 0).

J'ai ensuite créé une classe "Piece" dont hérite les classes "King", "Pawn", "Bishop", "Queen", "Rook" et "Knight". Je représente une pièce avec son nom, son id et l'équipe à laquelle elle appartient (noir ou blanc).

Pour chaque pièce (Roi, Reine, Pion, Fou, Tour et Chevalier) Je crée une fonction "ValidMoves" qui, en fonction de sa position, retourne un ensemble de position possibles.

Exemple avec "ValidMoves" du Chevalier:

```
class Knight(Piece):
  def valid_moves(self):
      moves=[]

      pos_moves=[(1,2),(-1,2),(1,-2),(-1,-2),(2,1),(-2,1),(-2,-1),(2,-1)]
      for t in pos_moves:
          bpos =self.position
          bpos = (bpos[0]+t[0],bpos[1]+t[1])
          if(not (bpos[0]<0 or bpos[1]<0 or bpos[0]>7 or bpos[1]> 7)):
              pot_object = self.gs.get_table()[bpos[0],bpos[1]]
              if(pot_object==0):
                  moves.append(bpos)
              else:
                  if(pot_object.get_team()==self.team):
                      pass
                  else:
                      moves.append(bpos)
      return moves
```

Figure 3: ValidMoves du Chevalier

Le principe est simple, le chevalier a un ensemble de transitions par défaut, il suffit de vérifier pour toutes ces transitions, si elles sont possible. C'est à dire qu'elle ne dépassent pas les limites du board et qu'elle n'écrasent pas une pièce alliée.

Il important de préciser que pour les pièces qui bougent de manières continue (Reine, Tour et fou), L'algorithme est différent. Je fais une boucle dans chacune des directions possibles et j'ajoute les mouves jusqu'à ce qu'ils ne soient plus possibles.

Vous trouverez ci dessous un exemple de "ValidMoves" pour le fou:

```
class Bishop(Piece.Piece):

    def valid_moves(self):
        moves = []
        moves = moves + self.right_top() + self.left_top() + self.right_down() + self.left_down()
        return moves

    def left_top(self):
        ltmoves=[]
        pos = self.position
        possible = True
        while(possible):
            if(pos[0]==0 or pos[1]== 0):
                possible = False
                break
            pos=(pos[0]-1,pos[1]-1)
            pot_object = self.gs.get_table()[pos[0],pos[1]]
            if(pot_object==0):
                ltmoves.append(pos)
            else:
                if(pot_object.get_team()==self.team):
                    possible=False
                    break
                else:
                    ltmoves.append(pos)
                    possible=False
```

Figure 4: ValidMoves Fou

Pour ce qui est du programme principal, il consiste en une boucle qui se finit une fois l'un des

deux roi mort. A chaque itération on demande à l'utilisateur d'insérer la case de départ et la case vers laquelle il veut déplacer sa pièce. Le GameState s'occupe automatiquement d'assigner les tours et de vérifier si les actions sont possibles.

Ci-dessous vous trouverez le code de la fonction main :

```
import sys
from GameState import GameState
def main():
    if(len(sys.argv)>1):
        f = open(sys.argv[1],"w")
    gs = GameState()
    while(not gs.get_finished()):
        sturn = ""
        if(gs.get_turn() == 0):
            sturn = "Black moves"
        else:
            sturn = 'White moves'
        pos1= input(sturn + "\n" + "Enter piece position: ")
        pos2 = input("Enter destination: ")
        if ((pos1[0] in gs.get_alphabetical()) and (pos1[1] in gs.get_numerical())
        and (pos2[0] in gs.get_alphabetical()) and (pos2[1] in gs.get_numerical())):
            p1 = (gs.get_numerical().index(pos1[1]),gs.get_alphabetical().index(pos1[0]))
            p2 = (gs.get_numerical().index(pos2[1]),gs.get_alphabetical().index(pos2[0]))
            if(not gs.step(p1,p2)):
                print("Impossible move")

            print(gs.print_board())
        else:
            print("error in input")
    if(gs.get_winner ==0 ):
        print("gg wp for the player 0!")
    else:
        print("gg wp for the player 1!")
    for i in gs.get_logs():
        f.write(i)
    f.close
```

Figure 5: Programme principal

Notons que le déroulement de la partie est enregistré au fur et à mesure du jeu dans une variable (logs) du GameState et qu'une fois la partie finie, c'est enregistré dans un fichier donné en paramètre du programme.

Pour finir voici un exemple de l'exécution du jeu:

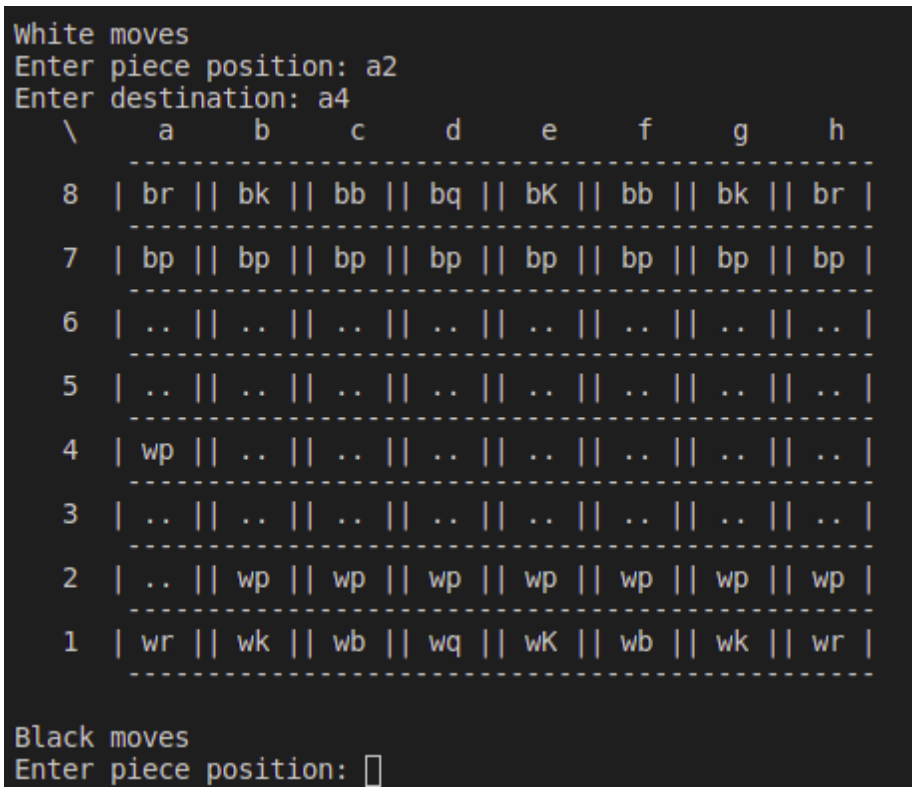

Figure 6: Exemple d'exécution