# Majeure Informatique: Jeu d'échecs

BOYER Timothé, MOURET Basile, HACINI Malik

26 Septembre 2023

## Table des Matières

# 1 Introduction

L'objectif de ce projet est de programmer (et tester) un jeu d'échecs qui permet à deux joueurs de s'affronter, chaque joueur peut être soit un humain, soit l'ordinateur via une IA (intelligence artificielle) . Le jeu se jouera dans un terminal, et si les deux joueurs sont humains, ils utiliseront le même clavier.

# 2 Collaboration

Pour ce projet, nous avons mis en place un dépôt GitHub, qui a nous a permis de chacun travailler sur sa propre version du projet avec VS Code (dépôt cloné localement) et ensuite de plus facilement mettre a jour le projet pour tout le monde. Cependant, ce projet était pour nous tous notre premier contact avec Git et Github, donc nous n'avons certainement pas utilisé l'outil à son plein potentiel. Nous avons notamment plusieurs fois du traiter manuellement des conflits de fusion, lorsque la répartition des tâches n'était pas assez bien réalisée. Dans l'ensemble, notre collaboration fut tout de même fluide et efficace.

# 3 Conception

## 3.1 Règles traitées

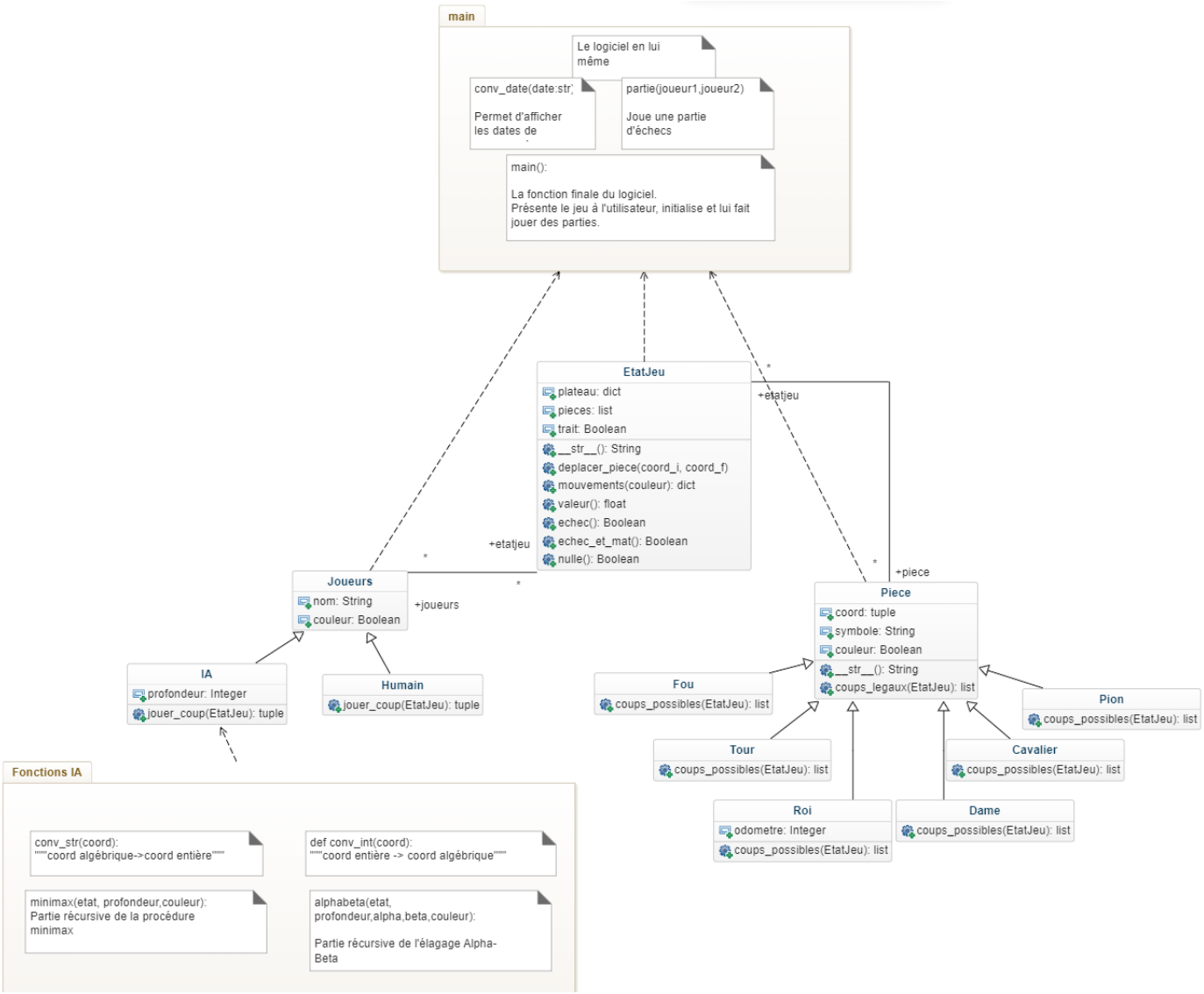
Le jeu d'échecs comporte plusieurs règles spéciales (Promotion, Roque, prise en passant...) Cependant, bon nombres d'entre elles sont plutôt fastidieuses à mettre en place, et ne sont pas forcément essentielles au déroulement du jeu. Nous avons donc seulement implémenté la Promotion.

La gestion de la nullité de la partie mérite aussi de s'y attarder. Nous avons longtemps hésité sur la méthode à mettre en place pour gérer la nullité de la partie.

## 3.2 Architecture

Ce projet a été réalisé en utilisant les outils de la programmation orientée objet (POO). Les différentes composantes d'un jeu d'échecs (joueurs, pieces) sont donc représentées par des classes.

### 3.2.1 Diagramme de classe UML




Nous avons donc divisé notre projet en trois grandes classes : EtatJeu, Joueurs et Piece. Des sous classes 'Humain' et 'IA' sont présentes dans la classe Joueurs. Ensuite, on regroupe tout dans un programme main, qui constitue la partie extérieure du logiciel.

Il y eu 2 grandes phases de développement du projet : La conception du jeu entre humains et la mise en place de l'IA. Chaque phase à donné lieu à plusieurs algorithmes principaux que nous allons détailler. Evidemment, nous avons rencontré de nombreuses difficultés, que nous détaillerons, durant chaque étape de la conception.

## 3.3 Conception du Jeu d'échecs

Nous avons commencé par créer en parallèle les 3 classes constituantes du jeu: Piece, EtatJeu et Joueurs. Ces classes sont toutes associées (CF 2.2.1), donc nous ne les avons pas réellement rédigés indépendamment.

### 3.3.1 Classe Piece

La classe Piece et ses sous-classes sont les lieux ou les règles du jeu sont définies. On représente chaque type de pièce par une sous-classe de la classe Piece. Chacune de ses pieces possède sa propre méthode (overwrite) coups-possibles, qui encode les règles du jeu qui lui sont relatives. Ces méthodes nécéssitent évidemment d'avoir accès a l'état du jeu. Enfin, la classe Piece possède une méthode coups-légaux (donc identique pour tout les types de pièces) qui trie une liste de coups possibles en enlevant ceux qui mettent en échec le roi. Ces méthodes nécéssitent évidemment d'avoir accès a l'état du jeu, que nous avons donc défini par la suite.

**Difficultés** Nous avons débuté l'implémentation de la classe Piece très tôt dans la conception. Cependant, la méthode coups-légaux était très compliquée à mettre en place : nous n'avions pas encore de moyen de vérifier si un roi était un échec. Nous avons donc du mettre en pause la conception de la classe Piece. A ce stade là, voici son corps :

```
1
2  class Piece:
3  """Classe destinée à représenter les pièces du jeu d'échec. Chaque pièce
4     possède une couleur et une coordonnée (tuple).
5     Chaque pièce en particulier est une sous-classe de cette classe (on y définira notamment les v
6     et symboles des pièces).
7  """
8  def __init__(self, couleur=None, coord=None):
9      self.couleur=couleur
10     self.coord=coord
11     self.symbole=None
12
13 def __str__(self):
14     return self.symbole
15
16 #Exemple d'une piece (sous-classe)
17 class Dame(Piece):
18 """Dame du jeu d'échecs. Hérite de Piece"""
19 def __init__(self, couleur, coord=None):
20     super().__init__(couleur, coord)
21     self.nom="Dame"
22
23     if self.couleur:
24         self.symbole=" "
25         self.valeur=9
```

```
26      else:
27          self.symbole=" "
28          self.valeur=-9
29
30  def coups_possibles(self, partie):
31      coups=[]
32      x,y=self.coord
33
34      for direction in [(1,1),(1,-1),(-1,-1),(-1,1),(1,0),(0,1),(-1,0),(0,-1)]:
35          x,y=self.coord
36          x+=direction[0]
37          y+=direction[1]
38          while 0<=x<=7  and 0<=y<=7:
39              piece=partie.plateau.get((x,y),None)
40              if piece==None:
41                  coups.append((x,y))
42              else:
43                  if piece.couleur != self.couleur:
44                      coups.append((x,y))
45                      break
46                  elif piece.couleur==self.couleur:
47                      break
48              x+=direction[0]
49              y+=direction[1]
50      return coups
51
52
```

### 3.3.2 Classe EtatJeu

La classe EtatJeu représente la partie d'échecs. Elle contient le plateau, les pièces, et et est destinée à contenir toutes les méthodes relatives au jeu. (deplacer piece, calcul valeur etc)

**Plateau: Création, Affichage et Sauvegarde** Nous avons d'abord mis en place le plateau, ainsi que l'affichage de celui-ci, avec la méthode spéciale ___str___.

```
1
2   class EtatJeu:
3
4   """Partie de jeu d'échecs
5   """
6   def __init__(self, sauvegarde : str = "Plateau_base.fen"):
7       """Construit une partie d'échecs.
8       Construit le plateau et les pièces grâce au fichier de sauvegarde FEN donné (Le plateau de bas
9
```

```
10
11      Args:
12          sauvegarde (str, optional): Le nom du plateau de sauvegarde. Défault : "Plateau_base".
13      """
14
15      print("Chargement de la partie")
16      #Création des pieces
17      self.pieces=[[],[]]
18      self.plateau = dict()
19
20      def __str__(self)->str:
21      """Méthode print pour la partie. Affiche le plateau dans
22      son état actuel.
23      Args:
24          tour (bool) :  True <=> Tour aux blancs
25      Returns:
26          str: Le plateau.
27      """
28      if self.trait:
29          ordre_affichage=range(7,-1,-1)
30      else:
31          ordre_affichage=range(8)
32
33      p=""
34
35      num_ligne=[str(x) for x in range(1,9)]
36      nom_col=["A","B" ,"C",
37               "D","E" ,"F","G","H"]
38
39      p+=" "*5 +  "    ".join(nom_col) +"\n"
40
41      for i in ordre_affichage:
42
43         p+=num_ligne[i] + "  | "
44
45         for j in range(8):
46             symbole=self.plateau.get((j,i)," ").__str__()
47             p+= symbole + "  | "
48         p+=  "\n" + "   "+ "-"*41 + "\n"
49
50      p+=" "*5 +  "    ".join(nom_col)
51
52      return p
53
```

Plusieurs choix importants ont été réalisés pour la mise en place du plateau: Nous le représentons

via un dictionnaire, contenant des coordonnées (tuple) en clé et des pièces (objets de la classe Piece) en valeur. Cette réprésentation possède plusieurs avantages: Il est facile d'accéder au symbole UTF-8 des pièces pour l'affichage, à la couleur d'une pièce, ou même à ses coups possibles.

Aussi, un objet EtatJeu possède une liste de deux listes de pièces, les blanches et les noires. L'accès au pièces noires ou blanches est réalisé en accédant au bone indice de la liste pieces via le trait.

Pour la sauvegarde, nous avons commencé par en implémenter une, puis nous l'avons modifié par la suite. Notre première méthode de sauvegarde était basée sur le dictionnaire des pièces. On sauvegardait, dans un fichier texte, les éléments nécéssaires à la construction des objets Piece (type,couleur,position).

Cependant, par la suite, cette méthode nous a gênée. Pour tester le programme, nous avions besoin de pouvoir rapidement générer des plateaux avec une situation exacte. Cependant, notre sauvegarde n'étant pas pratique à manipuler, nous devions constamment jouer, sur le programme, les coups exacts pour arriver à la situation souhaitée. Cela était fastidieux, et très inefficace. Nous avons donc opté pour une sauvegarde adoptant la notation Forsyth-Edwards (FEN). Cette notation est utilisée dans la plupart des chess engine actuels. Elle nous a ensuite permis de créer les sauvegardes de tests souhaitées grâce à un outil de Lichess.

Voici l'implémentation finale de la sauvegarde:

```
1
2   #Sauvegarder la partie
3   def fen_position(self)->str:
4       """Traduit une partie en notation FEN, afin de sauvegarder.
5
6       Returns:
7           str: FEN de la partie
8       """
9       pion=["p","P"]
10      cavalier=["n","N"]
11      fou=["b","B"]
12      tour=["r","R"]
13      dame=["q","Q"]
14      roi=["k","K"]
15
16      fen=""
17      for ligne in range(7,-1,-1):
18          vides=0
19          for col in range(8):
20              piece=self.plateau.get((col,ligne),None)
21
22              if piece==None:
23                  vides+=1
24              else :
25                  if vides !=0:
26                      fen+=str(vides)
27                      vides = 0
```

```
                if isinstance(piece,Pion): fen+=f"{pion[piece.couleur]}"
                if isinstance(piece,Cavalier): fen+=f"{cavalier[piece.couleur]}"
                if isinstance(piece,Fou): fen+=f"{fou[piece.couleur]}"
                if isinstance(piece,Tour): fen+=f"{tour[piece.couleur]}"
                if isinstance(piece,Dame): fen+=f"{dame[piece.couleur]}"
                if isinstance(piece,Roi): fen+=f"{roi[piece.couleur]}"
        if vides !=0:
            fen+=str(vides)
        fen+="/"
    trait=["b","w"]

    fen+=f" {trait[self.trait]} - - 0 0"
    return fen

def sauvegarder(self,nom_fichier : str ):
    """Sauvegarde la partie dans le fichier indiqué

    Args:
        nom_fichier (str): le nom du fichier.
    """

    fichier = open("sauvegardes\\"+nom_fichier+".fen", 'w')
    fichier.write(self.fen_position())
    fichier.close()

```

On charge ensuite la sauvegarde dans le constructeur de la classe EtatJeu.

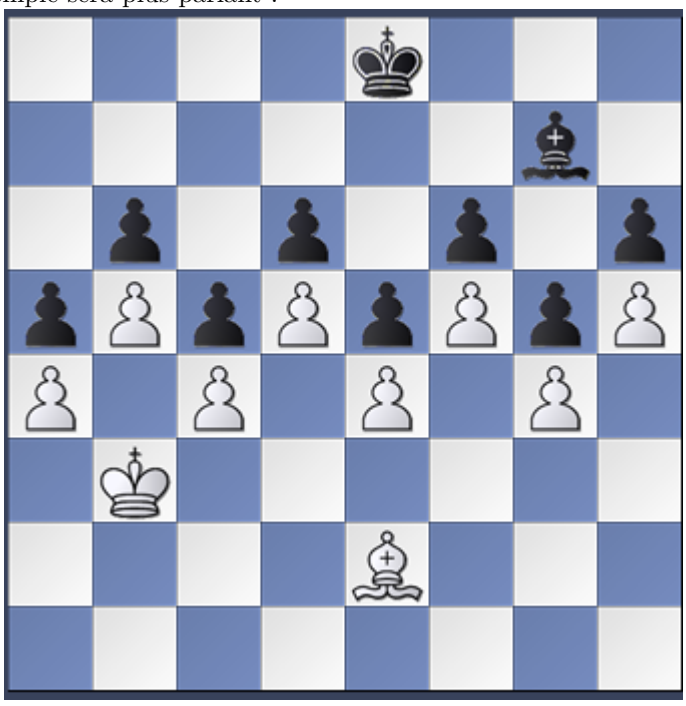
**Vérification de l'état du jeu** Nous devions donc maintenant ajouter les fonctions relatives à la vérification de l'état du jeu : échec, échec et mat,gagnant, nulle.

Les fonctions échec, échec et mat et gagnant sont assez explicites :

```

def echec(self) -> bool:
    """Détermine si le roi du joueur à qui c'est le tour (trait de la partie) est en échec.

    Returns:
        bool: True <=> Roi en échec
    """
    #il faut trouver qui est le joueur à qui c'est le tour
    case_roi = None
    for piece in self.pieces[self.trait]:
        if isinstance(piece,Roi):
            case_roi = piece.coord#on récupere la case occupée par le roi
    pieces_adversaire = self.pieces[not (self.trait)]#on récupere les pieces de l'adversaire
```

```
        for piece in pieces_adversaire: #Pour les pièces de l'adversaire en jeu
            for case in piece.coups_possibles(self):# Pour chaque case controllé par l'adversaire
                if case == case_roi :#On vérifie si cette case est celle du roi
                    return True
        return False




    def echec_et_mat(self)->bool:
        """Détermine si le joueur à qui c'est le tour est en échec et mat.

        Returns:
            bool
        """
        if not self.echec() : return False #si le roi n'est pas en echec il n'y a pas mat
        #on regarde s'il existe des pièces qui ont le droit de bouger


        pieces_joueur = self.pieces[self.trait]
        for piece in pieces_joueur:
                if len(piece.coups_legaux(self))>0 :return False
        return True


    def gagnant(self)->bool:
        """Donne le gagnant de la partie, si il y en a un."""

        if self.echec_et_mat():
            return not self.trait #attention ici on ne renvoie que la couleur du trait, au main de dé
        return None
```

Pour la nulle, c'est plus compliqué. Aux échécs, il existe une multitude de manières de rendre en partie nulle. Cependant, beaucoup d'entres elles sont très complexes à mettre en place. Un exemple sera plus parlant :



Aux yeux d'un joueur d'échec, cette partie est très clairement une nulle. Cependant, il est très complexe pour un algorithme de le reconnaître. Nous ne pouvons donc pas mettre en place toutes les possibilités différentes. Notre gestion de la nulle est donc assez basique, nous ne vérifions que 2 cas particuliers.

Premièrement, nous vérifions la règle du pat : un des joueurs n'a aucun coup possible, mais il n'est pas en échec en mat. Ensuite, on annonce nulle si un des deux rois à bougé plus de 30 fois d'affilé (pour ca, on a ajouté un odomètre au rois.) Enfin, à chaque tour, les joueurs peuvent conjointement décider de terminer sur une nulle.

Voici l'implémentation des deux cas particuliers:

---

```
1
2   def nulle(self)->bool:
3       """Détermine si la partie est nulle selon les deux cas particuliers que nous avons défini : le
4       Returns:
5           bool: True <=> La partie est nulle
6       """
7       odometre=0
8       coups=[]
9       pieces_joueur = self.pieces[self.trait]
10      for piece in pieces_joueur:
11          coups+=piece.coups_legaux(self)
```

```
12          if isinstance(piece,Roi):
13              odometre=piece.odometre
14      return (not self.echec_et_mat()and len(coups)==0) or (odometre>=30)
```

### 3.3.3 Finalisation de la classe Piece

Une fois l'EtatJeu mis en place, nous pouvons écrire la méthode coups légaux.

```
1
2   def coups_legaux(self,partie)->list[tuple[int,int]]:
3       """Détermine tous les coups légaux qu'une pièce peut faire dans une partie.
4          Le principe est de simuler les coups possibles, puis de trier parmi les coups menant
5          à un échec (donc non légaux)
6       Args:
7           partie (EtatJeu): La partie
8
9       Returns:
10          list[tuple[int,int]] : Les coups
11      """
12      coups = []
13      for coord_f in self.coups_possibles(partie):
14          #sauvegarde de l'ancien plateau
15          piece_potentiellement_mangée = partie.plateau.get(coord_f,None)
16          coord_i = self.coord
17          #déplacement de la pièce
18          self.coord=coord_f
19          partie.plateau[coord_f] = partie.plateau.pop(coord_i)
20          #On retire la pièce à l'adversaire
21          if piece_potentiellement_mangée is not None:
22              partie.pieces[not self.couleur].remove(piece_potentiellement_mangée)
23
24          #On vérifie si il y a échec
25          if not partie.echec():
26              coups.append(coord_f)
27          #retrait du coups
28          self.coord=coord_i
29          partie.plateau[coord_i] = partie.plateau.pop(coord_f)
30          #On replace la pièce mangée
31          if piece_potentiellement_mangée is not None:
32              partie.plateau[coord_f] = piece_potentiellement_mangée
33              partie.pieces[not self.couleur].append(piece_potentiellement_mangée)
34      return coups
```

### 3.3.4 Classe Joueurs : Humain

A ce stade, le noyau du jeu d'échecs est terminé. Pour le rendre jouable, nous devons implémenter la classe Joueurs. C'est la classe Joueurs qui sera, durant la partie, l'interface entre le programme main et les classes Piece et EtatJeu, via la méthode jouer-coup.
En prévision de l'implémentation de l'IA, on implémente une sous classe Humain, qui a sa propre méthode jouer-coup, car celle de l'IA sera différente. Pour la classe Humain, la méthode jouer-coup doit interagir avec l'utilisateur.
On doit notamment effectuer la conversion des coordonnées du plateau en notation algébrique (il y a une fonction extérieure dédiée).

---

```
1
2   class Joueur():
3       """Représente un joueur d'échecs,qui possède un nom et une couleur.
4       Est destinée à être la super-classe des joueurs concrets (Humain, IA)
5       """
6       def __init__(self,nom : str,couleur : bool) -> None:
7           """création d'un joueur
8
9           Args:
10              nom (str): nom du joueur
11              couleur (bool): True <=> Blanc
12          """
13          self.nom = nom
14          self.couleur = couleur
15
16          class Humain(Joueur):
17          """Représente un joueur humain, qui possède un nom et une couleur.
18
19          Args:
20              Joueur (class): super classe
21          """
22          def __init__(self, nom: str, couleur: bool) -> None:
23              super().__init__(nom, couleur)
24
25
26          def jouer_coup(self,partie: dict) -> tuple[int,int]:
27              """Fait jouer un coup à l'utilisateur.
28
29              Args:
30                  partie (dict): la partie en cours
31
32              Returns:
33                  tuple[int,int]: le coup.
34              """
35              coup_jouable  = False
```

```
            while not coup_jouable:

                piece_deplacable = False

                while not piece_deplacable:
                    #On doit vérifier la validité de chaque étape du coup.
                    coord_p = input(f"{self.nom}, ou est la pièce à bouger ? \n")

                    if coord_p in ["save","nulle"]: return coord_p
                    if not len(coord_p)==2:
                        print("ce n'est pas un coup valide, veuillez respecter ce format : e2 \n")

                    elif coord_p[0] not in ("a","b","c","d","e","f","g","h") or coord_p[1] not in
                        print("Ce n'est pas un coup valide! \n")


                    else :
                        #Transformation de la position de la pièce de notation algébrique aux coo
                        #Pour la ligne, cela dépend de la couleur du joueur, l'affichage étant re
                        coord_p = conv_str(coord_p)
                        if coord_p not in partie.plateau.keys() : print("Cette case est vide.")

                        elif partie.plateau[coord_p] not in partie.pieces[self.couleur]: print("Ce
                        #vérifier que la piece peut etre bougée
                        elif partie.plateau[coord_p].coups_legaux(partie) == [] : print("Cette pi
                        #la piece peut etre déplacée
                        else : piece_deplacable = True

                #donner les coups possibles pour cette pièce
                coups_possibles=partie.plateau[coord_p].coups_legaux(partie)
                coups_a_afficher_not_alg=[conv_int(coup) for coup in coups_possibles]

                coups_a_afficher_output=""
                for coup in coups_a_afficher_not_alg:
                    coups_a_afficher_output+=f"{coup}, "


                print(f"vous pouvez déplacer votre {partie.plateau[coord_p].nom} sur les cases su
                #vérifier si le joueur veut bien jouer cette piece ou modifier son coupS
                coup_int = None
                premier_passage=True
                while coup_int not in coups_possibles:

                    if not premier_passage:
                        print("Ce coup n'est pas valide. \n")

```

```
                    #demander la case où le joueur veut déplacer le pion
                    coup = input("Quel coup voulez-vous jouer (None si vous voulez jouer une autre

                    coup_int = conv_str(coup)

                    premier_passage=False
                coup_jouable=True

            return coord_p,coup_int


```

## 3.4 Le Programme main

Au final, le jeu doit se joueur depuis un simple script main, qui utilise nos classes définies dans des fichiers annexes. Nous avons décidé d'implémenter ce programme avant de concevoir l'IA, afin de faciliter les tests. On définit alors une fonction partie, qui créee (à partir des choix de l'utilisateur) et fait jouer une partie à l'utilisateur. Cette fonction doit notamment gérer le vote de la nulle. On place ensuite cette fonction dans une boucle, en demandant à l'utilisateur si il veut rejouer à chaque fin de partie.

En voici les algorithmes principaux:

```


def partie(joueur1: Joueur,joueur2: Joueur):
    """Joue une partie d'échecs.

    Args:
        joueur1 (Joueur): Le joueur 1. Peut être humain ou IA
        joueur2 (Joueur): Le joueur 2. Peut être humain ou IA
    """

    save=None
    while save not in ("O","N"):

        save=input("Voulez vous charger une sauvegarde ? (O/N) \n")


    #On affiche les sauvegardes disponibles à l'utilisateur, et on lui en fait choisir une. On gè
    if save=="O":
        liste_fichiers=[fichier for fichier in os.listdir("./sauvegardes") if os.path.splitext(fic
        liste_fichiers.remove("Plateau_base.fen")
        if liste_fichiers==[]:
            start=input("""Aucune sauvegarde n'est disponible.
Appuyez sur Entrée pour démarrer une nouvelle partie.""")
```

```
            nom_save="Plateau_base.fen"
        else:
            print("Voici les sauvegardes disponibles: ")
            for index,fichier in enumerate(liste_fichiers):
                    print(index+1,".Sauvegarde du " + conv_date(fichier[5:-4]))
            fichier_valide=False
            while not fichier_valide:
                    num_save=input(f"Choisissez une sauvegarde (1-{len(liste_fichiers)}) \n")
                    if num_save.isdigit():
                        if int(num_save)-1 in list(range(len(liste_fichiers))):
                            nom_save=liste_fichiers[int(num_save) - 1]
                            fichier_valide=True
                            continue
                    print("Fichier introuvable ou invalide.")
        partie=EtatJeu(nom_save)

    else:
        partie= EtatJeu()

    print("Bonne partie ! A tout moment, entrez 'save' pour sauvegarder et quitter, et 'nulle' pou

    joueurs=[joueur2,joueur1]


    print(partie)
    nulle_votee=False
    draw_votes=0
    while partie.gagnant() is None and not partie.nulle() and not nulle_votee:


        #Si le joueur précédent a voté nulle :
        if draw_votes==1:
            vote=0
            while vote not in ("O","N"):
                vote=input(f"{joueurs[int(partie.trait)].nom}, acceptez vous la nulle ? (O/N) \n")

            if vote=="O":
                nulle_votee=True
                partie.trait = not partie.trait #On change le tour
                continue
            else: print("Nulle refusée.")

        #On demande quelle pièce bouger au joueur (il peut écrire nulle ou save)
        deplacement_valide=False
        while not deplacement_valide:
            deplacement=joueurs[int(partie.trait)].jouer_coup(partie)
```

```
            if deplacement=="nulle" and isinstance(joueurs[not partie.trait],IA):
                print("Vous ne pouvez pas voter nulle contre une IA. ")
            else:
                deplacement_valide=True

        #Cas particuliers (vote de nulle ou save)
        if deplacement=="nulle":
                    draw_votes=1
                    partie.trait = not partie.trait #On change le tour
                    continue
        else:
            draw_votes=0
        if deplacement=="save" :
            date=time.strftime("%Y%m%d-%H%M%S")
            partie.sauvegarder(f"save_{date}")
            print("Sauvegarde effectuée.")
            return "N"

        partie.deplacer_piece(deplacement[0],deplacement[1])

        print(partie)
        if partie.echec(): print("Votre roi est en échec.")

    #On affiche le résultat de la partie.
    if partie.nulle() or nulle_votee: print("Partie Nulle.")
    else: print(f"{joueurs[partie.gagnant()].nom} a gagné la partie ! \n")



def main():
    """Le jeu d'échec dans son intégralité. Initialise une partie,
    la fait jouer et répète tant que l'utilisateur veut rejouer."""

    while True:


        replay=None
        joueurs = []
        for i in (1,0):
            type_joueur=None
            if i==1: couleur="blanc"
            else: couleur="noir"

            while type_joueur not in ("1","2"):
                type_joueur=input(f"De quel type est le Joueur {couleur} ? \n 1: Humain \n 2: IA \
```

```
116
117             if type_joueur=="1":
118                 nom=input(f"Quel est le nom du Joueur {couleur} ? \n")
119                 joueurs.append(Humain(nom,i))
120             else:
121                 niveau=5
122                 while niveau not in ("0","1","2","3"):
123                     niveau=input(f"""Quel est le niveau de l'IA {couleur} souhaité ?
124 0. Novice : Joue aléatoirement.
125 1. Débutant
126 2. Intérmédiaire
127 3. Avancé \n""")
128                 nom=f"IA {couleur}"
129                 joueurs.append(IA(nom, i, int(niveau)))
130
131         replay=partie(joueurs[0],joueurs[1])
132
133         while replay not in ("O","N"):
134             replay=input("Voulez vous rejouer ? (O/N) \n")
135
136         if replay=="N":
137             print("Merci d'avoir joué ! ")
138             sys.exit()
```

## 3.5 Conception de l'IA

Notre IA sera basée sur l'algorithme minimax et l'élagage alphabeta. Pour implémenter l'IA, nous nous y sommes repris plusieurs fois. Nous avons essuyé plusieurs difficultés : valeur erronée, problèmes de récursivité, coups absurdes de l'IA etc. Notre première difficulté fut rencontré lors de l'élaboration de la fonction de calcul de valeur du jeu.

### 3.5.1 Calcul de la valeur

Pour évaluer la partie, nous avons longtemps cherché la méthode correcte. La base de notre méthode heuristique est la fonction d'évaluation de Claude Shannon. C'est une fonction symétrique : si l'avantage est aux blancs, la valeur sera positive, si l'avantage est aux noirs, elle sera négative. Le critère principal est le barème général de la valeur des pièces aux échecs :

| Valeurs des pièces | | | | | |
|---|---|---|---|---|---|
| Roi | Pion | Cavalier | Fou | Tour | Dame |
| 1000 | 1 | 3 | 3 | 5 | 9 |

Cependant, cette base n'était pas suffisante pour obtenir une valeur reflétant correctement le jeu d'échecs. Nous avons donc pris plusieurs critères en compte :

- Si les pièces sont au centre ou au sous-centre du plateau (augmentation de la valeur)

- Le nombre de cases accessibles au prochian tour (appelées cases controlées, augmentation de la valeur)
- L'alignement des pions (baisse de la valeur)

Evidemment, cette liste n'est absolument pas exhaustive et est loin de représenter parfaitement toute la complexité du jeu d'échecs. De plus, le poids que nous avons attribué à chaque critère se base sur des méthodes déjà existantes, mais une part d'arbitraire rentre en compte. En revanche, il n'y aurait que peu d'intêret à avoir une fonction valeur extrêmemnt élaborée. En effet, notre IA ne pourra que se contenter d'une profondeur de 3 au maximum (à partir de 4, c'est trop long en pratique), et les effets d'une valeur plus précise ne se feront donc pas énormément ressentir pour la majorité de la partie. Vers la fin de la partie, l'impact est plus grand.

Nous avons testé plusieurs versions, mais voici la version finale du calcul de la valeur( dans la classe EtatJeu):

---

```
1
2   def calcul_valeur(self)->float:
3       """Fonction d'évaluation de l'état du jeu. Est basée sur JAI PAS TROUVE LE WIKI
4
5       Returns:
6           float: valeur du jeu
7       """
8       valeur=0
9       if self.echec_et_mat():
10          if self.gagnant():
11              return 1000
12          else:
13              return -1000
14
15      if self.nulle():
16          return 0
17
18
19      for pieces in [self.pieces[1], self.pieces[0]]:
20          cases_controllees=set()
21          pions=[]
22
23          for piece in pieces:
24              valeur+=piece.valeur
25
26              for centre in [(3,3),(3,4),(4,4),(4,3)]:
27                  if piece==self.plateau.get(centre,None):
28                      valeur+=(0.5)*((-1)**(not piece.couleur))
29
30              for sous_centre in [(2,2),(2,3),(2,4),(2,5),(3,5),(4,5),(5,5),(6,5),(6,4),(6,3),(6,2),
31                  if piece==self.plateau.get(sous_centre, None):
32                      valeur+=(0.1)*((-1)**(not piece.couleur))
```

```
33
34
35              if isinstance(piece,Pion):
36                  pions.append(piece)
37
38              cases_controllees |= set(piece.coups_possibles(self))
39
40          valeur+=0.05*len(cases_controllees)*((-1)**(not piece.couleur))
41
42
43          collones=[]
44          for pion in pions:
45              if pion.coord[0] not in collones:
46                  collones.append(pion.coord[0])
47              else:
48                  valeur+=0.1*((-1)**piece.couleur)
49      return round(valeur,3)
50
```

### 3.5.2 Minimax

Pour implémenter Minimax, nous nous sommes basés sur l'algorithme du cours. Il faut cependant pouvoir passer la profondeur comme un paramètre. Nous ne rédigons pas la fonction dans la classe Joueurs, du à sa nature récursive. C'est la méthode jouer coup d'un joueur IA qui y fera appel. L'algorithme minimax impose de simuler des coups. En première approche, nous avons implémenté la méthode suivante :

- On copie l'état de jeu avec un deepcopy
- On joue un coup sur ce nouvel état, et on calcule la valeur
- On renvoie la valeur

Cependant, nous nous sommes rendus compte de l'inefficacité de la méthode en la testant. La création d'un nouvel objet EtatJeu à chaque itération de la fonction était très longue, et remplissait la mémoire. Avec cette méthode (sans élagage alphabeta), une IA de profondeur 3 mettait plus de 1 min a donner un coup.

Nous avons alors modifié notre approche.

- On sauvegarde les informations qui seront modifiées (positions de la pièce déplacée, pièce mangée, odomètre du roi)
- On simule directement le coup sur l'état de jeu.
- On calcule la valeur.
- On restaure le jeu à son état initial.

Avec cette nouvelle approche, le temps d'obtention d'un coup d'une IA de profondeur 3 passe à 30 secondes.

Voici l'implémentation finale de la fonction minimax :

```
1
2  def minimax(etat, profondeur,couleur):
3  """Implémentation de l'algorithme minimax appliqué à notre exemple
4
5  Args:
6      etat (EtatJeu): etat de la partie à analyser
7      profondeur (int): nombre de profondeur restante a analyser
8      couleur (bool): couleur dont il faut identifier la valeur du jeu
9
10 Returns:
11     int: valeur du minimax
12 """
13 if profondeur==0 or etat.echec_et_mat():
14     return etat.calcul_valeur()
15 if couleur:
16     valeur = -math.inf
17     for coord_i,coords_f in etat.mouvements(etat.trait).items():
18         for coord_f in coords_f:
19             #sauvegarder les données du plateau de jeu
20             piece_retirée = etat.plateau.get(coord_f,None)
21
22             #on sauvegarde l'odometre
23             for piece in etat.pieces[not etat.trait]:
24                 if isinstance(piece,Roi):
25                     sauv_odometre = piece.odometre
26                     roi = piece
27
28             #on bouge la piece
29             etat.deplacer_piece(coord_i,coord_f)
30             #calcul recursif de la valeur
31             valeur = max(valeur,minimax(etat,profondeur-1, not couleur))
32             #retirer coup
33             #remettre la piece au bon endroit
34             etat.deplacer_piece(coord_f,coord_i)
35             if piece_retirée is not None:
36                 etat.plateau[coord_f] = piece_retirée
37                 etat.pieces[not etat.trait].append(piece_retirée)
38             #remettre l'odometre a sa valeur
39             roi.odometre = sauv_odometre
40
41     return valeur
42 else :
```

```
    valeur = math.inf
    for coord_i,coords_f in etat.mouvements(etat.trait).items():
        for coord_f in coords_f:
            #sauvegarder les données du plateau de jeu
            piece_retirée = etat.plateau.get(coord_f,None)

            #on sauvegarde l'odometre
            for piece in etat.pieces[not etat.trait]:
                if isinstance(piece,Roi):
                    sauv_odometre = piece.odometre
                    roi = piece

            #on bouge la piece
            etat.deplacer_piece(coord_i,coord_f)
            #calcul recursif de la valeur
            valeur = min(valeur,minimax(etat,profondeur-1, not couleur))
            #retirer coup
            etat.deplacer_piece(coord_f,coord_i)#remettre la piece au bon endroit
            if piece_retirée is not None:
                etat.plateau[coord_f] = piece_retirée
                etat.pieces[not etat.trait].append(piece_retirée)
            #remettre l'odometre a sa valeur
            roi.odometre = sauv_odometre

    return valeur


```

### 3.5.3 Elagage Alpha-Beta

L'élagage alpha-beta n'améliore pas la qualité des coups de l'IA, mais réduit significativement la durée des coups. Nous nous sommes basés sur l'algorithme vu en cours. En voici l'implémentation :

```

def alphabeta(etat, profondeur,alpha,beta,couleur):
"""Implémentation de l'algorithme minimax avec élagage alpha beta appliqué à notre exemple

Args:
    etat (EtatJeu): etat de la partie à analyser
    profondeur (int): nombre de profondeur restante a analyser
    couleur (bool): couleur dont il faut identifier la valeur du jeu

Returns:
    int: valeur du minimax
```

```
    """
    if profondeur==0 or etat.echec_et_mat():

        return etat.calcul_valeur()
    if couleur:
        valeur = -math.inf
        for coord_i,coords_f in etat.mouvements(etat.trait).items():
            for coord_f in coords_f:
                #sauvegarder les données du plateau de jeu
                piece_retirée = etat.plateau.get(coord_f,None)

                #on sauvegarde l'odometre
                for piece in etat.pieces[not etat.trait]:
                    if isinstance(piece,Roi):
                        sauv_odometre = piece.odometre
                        roi = piece
                #on bouge la piece

                etat.deplacer_piece(coord_i,coord_f)
                #calcul recursif de la valeur
                valeur = max(valeur,alphabeta(etat,profondeur-1,alpha,beta, not couleur))
                #retirer coup
                etat.deplacer_piece(coord_f,coord_i)#remettre la piece au bon endroit
                if piece_retirée is not None:
                    etat.plateau[coord_f] = piece_retirée
                    etat.pieces[not etat.trait].append(piece_retirée)
                #remettre l'odometre a sa valeur
                roi.odometre = sauv_odometre

                if valeur > beta :
                    break
                alpha = max(alpha,valeur)
        return valeur
    else :
        valeur = math.inf
        for coord_i,coords_f in etat.mouvements(etat.trait).items():
            for coord_f in coords_f:
                #sauvegarder les données du plateau de jeu
                piece_retirée = etat.plateau.get(coord_f,None)

                #on sauvegarde l'odometre
                for piece in etat.pieces[not etat.trait]:
                    if isinstance(piece,Roi):
                        sauv_odometre = piece.odometre
                        roi = piece

```

```
58              #on bouge la piece
59              etat.deplacer_piece(coord_i,coord_f)
60              #calcul recursif de la valeur
61              valeur = min(valeur,alphabeta(etat,profondeur-1,alpha,beta, not couleur))
62              #retirer le coup
63              etat.deplacer_piece(coord_f,coord_i)#remettre la piece Nau bon endroit
64              if piece_retirée is not None:
65                  etat.plateau[coord_f] = piece_retirée
66                  etat.pieces[not etat.trait].append(piece_retirée)
67
68              #remettre l'odometre a sa valeur
69              roi.odometre = sauv_odometre
70
71              if valeur < alpha :
72                  break
73              beta = min(beta,valeur)
74      return valeur
```

Grâce à cette amélioration, une IA de profondeur 3 joue maintenant en 15 secondes en moyenne. (Nous avons établi cette valeur grâce à un test, détaillé dans la section prévue à cet effet.)

### 3.5.4 Méthode Jouer_Coup pour l'IA

Malgré sa présence à cet endroit dans le rapport, nous avons écrit la méthode jouer_coup assez tôt dans la conception de l'IA. Cela nous permettait de tester eficacement nos algorithmes alpha_beta et minimax. Sa version finale fut cependant rédigée à la fin du processus. Cette méthode met en place la version complète d'un minimax avec élagage AlphaBeta, c'est elle qui trouve et choisit le meilleur coup possible à une certaine profondeur.

La voici:

```
1
2   def jouer_coup(self,partie: dict) -> tuple[int,int]:
3   """Détermine le coup d'une IA, selon ses paramètres.
4
5   Args:
6       partie (dict): partie dans son état actuel.
7
8   Returns:
9       tuple[int,int]: coup.
10  """
11  couleurs=["noire","blanche"]
12  print(f"\n L'IA {couleurs[self.couleur]} réflechit \n")
13  meilleur_coup = None
14  alpha = -math.inf
15  beta = math.inf
16  #coups aléatoire
```

```
if self.profondeur == 0:
    coups = []
    for coord_i,coords_f in partie.mouvements(self.couleur).items():
        for coord_f in coords_f:
            coups.append((coord_i,coord_f))
    return coups[random.randint(0,len(coups)-1)]

#si le coup n'est pas aléatoire
if self.couleur :
    meilleure_valeur = -math.inf
    for coord_i,coords_f in partie.mouvements(self.couleur).items():
        #récupere les coordonnées de départ de chaque piece qui peut être déplacée, et les cases s

        #Ici, l'idée est de simuler touts les coups possibles.
        #On sauvegarde toutes les informations de la partie que nous allons modifier pendant la si
        #on les modifie pour en extraire la valeur du jeu après le coup, puis on les rétablit.


        #pour chaque coups possible dans les déplacement disponibles de la piece
        for coord_f in coords_f:
            #sauvegarder les données du plateau de jeu
            piece_retirée = partie.plateau.get(coord_f,None)

            #on sauvegarde l'odometre
            for piece in partie.pieces[not partie.trait]:
                if isinstance(piece,Roi):
                    sauv_odometre = piece.odometre
                    roi = piece
            #on déplace la piece
            partie.deplacer_piece(coord_i,coord_f)

            #calcul de la valeur avec l'algo voulu
            if self.algo == "minimax" :
                valeur = minimax(partie,self.profondeur-1, not self.couleur)
            else:
                valeur = alphabeta(partie,self.profondeur-1,alpha,beta, not self.couleur)

            #retirer coup
            #remettre la piece au bon endroit
            partie.deplacer_piece(coord_f,coord_i)
            if piece_retirée is not None:
                partie.plateau[coord_f] = piece_retirée
                partie.pieces[not partie.trait].append(piece_retirée)

            #remettre l'odometre à sa valeur de départ
            roi.odometre = sauv_odometre
```

```
63
64              #le joueur blanc le maximum
65              if valeur> meilleure_valeur:
66                  meilleur_coup = coord_i,coord_f
67                  meilleure_valeur = valeur
68
69              #modification du alpha pour le minimax avec elagage alpha beta
70              alpha = max(alpha,valeur)
71  else :
72      meilleure_valeur = math.inf
73      for coord_i,coords_f in partie.mouvements(self.couleur).items():
74          #pour chaque coups possible dans les déplacement disponibles de la piece
75          for coord_f in coords_f:
76              #sauvegarder les données du plateau de jeu
77              piece_retirée = partie.plateau.get(coord_f,None)
78
79              #on sauvegarde l'odometre
80              for piece in partie.pieces[not partie.trait]:
81                  if isinstance(piece,Roi):
82                      sauv_odometre = piece.odometre
83                      roi = piece
84
85              #on bouge une piece
86              partie.deplacer_piece(coord_i,coord_f)
87              #calcul de la valeur avec l'algo voulu
88              if self.algo == "minimax" :
89                  valeur = minimax(partie,self.profondeur-1, not self.couleur)
90              else:
91                  valeur = alphabeta(partie,self.profondeur-1,alpha,beta, not self.couleur)
92
93              #retirer coup
94              partie.deplacer_piece(coord_f,coord_i)#remettre la piece au bon endroit
95              if piece_retirée is not None:
96                  partie.plateau[coord_f] = piece_retirée
97                  partie.pieces[not partie.trait].append(piece_retirée)
98
99              #remettre l'odometre a sa valeur
100             roi.odometre = sauv_odometre
101
102             #le joueur noir veut le minimum, le joueur blanc le maximum
103             if valeur< meilleure_valeur:
104                 meilleur_coup = coord_i,coord_f
105                 meilleure_valeur = valeur
106             beta = min(beta,valeur)
107 return meilleur_coup
```

# 4 Tests

Dans cette partie du compte rendu, nous nous intéresserons aux tests réalisés sur les différentes parties de notre programme.

## 4.1 Tests de conception du jeu d'échecs

Nous avons réalisé des tests sur les différents programmes qui permettent au jeu d'échecs de fonctionner.

Pour tester le bon fonctionnement du jeu, nous avons utilisé la fonctionnalité nous permettant d'importer des sauvegardes car ainsi, on peut tester des situations précises et voir si tout fonctionne comme prévu.
On crée donc une fonction dont le but est de créer des parties à partir du nom du fichier de test. Cela nous permet de ranger les tests dans différents dossiers dédiés à chaque test.

### 4.1.1 Tests de la classe pièce

Pour tester les pièces, nous avons d'abord testé la méthode coups_possibles pour chacune des pièces existantes. Pour cela, nous avons d'abord créé une situation avec la pièce que l'on veut tester. On note à la main toutes les cases accessibles par cette pièce et on vérifie si la méthode coups_possibles de cette pièce renvoie la même liste de coups.

```
1  # Déplacement des pièces
2  def test_deplacement_pion():
3      partie=init_partie_test("test_deplacement_pion")
4      partie.pieces[1][1].premier_coup=False
5      assert partie.pieces[1][0].coups_possibles(partie)==[(3, 4), (4, 4)]
6      assert partie.pieces[1][1].coups_possibles(partie)==[]
7      assert partie.pieces[1][2].coups_possibles(partie)==[(1, 2), (1, 3)]
8
9  def test_deplacement_fou():
10     partie=init_partie_test("test_deplacement_fou")
11     assert partie.pieces[1][0].coups_possibles(partie)==[(5,5),(6,6),(7,7),(5,3),(3,3),(2,2),
12     (1,1),(3,5),(2,6),(1,7)]
13
14 def test_deplacement_tour():
15     partie=init_partie_test("test_deplacement_tour")
16     assert partie.pieces[0][1].coups_possibles(partie)==[(5, 4), (6, 4), (7, 4), (4, 5),
17     (3, 4), (2, 4), (1, 4), (4, 3), (4, 2), (4, 1)]
18
19
20 def test_deplacement_reine():
21     partie=init_partie_test("test_deplacement_reine")
22     assert partie.pieces[0][0].coups_possibles(partie)==[(5, 5), (6, 6), (5, 3), (3, 3),
23     (2, 2), (1, 1), (0, 0), (3, 5), (2, 6), (1, 7),
24     (5, 4), (6, 4), (7, 4), (4, 5), (4, 6), (4, 7), (3, 4),
```

```
25      (2, 4), (4, 3)]
26
27  def test_deplacement_cavalier():
28      partie=init_partie_test("test_deplacement_cavalier")
29      assert partie.pieces[1][1].coups_possibles(partie)==[(5, 5), (3, 5), (2, 4), (2, 2),
30      (3, 1), (5, 1), (6, 2)]
```

Il faut ensuite vérifier si il est bien impossible de déplacer une pièce en mettant le roi en échec. Cette fonctionnalité correspond à la méthode coups_legaux de la classe Pièce. Pour vérifier si ça marche, on fait donc un plateau avec une pièce qui est clouée (si elle bouge, le roi est en échec) et on vérifie qu'elle n'a pas de coups légaux.

```
1   def test_coups_legaux():
2       partie=init_partie_test("test_coups_legaux")
3       assert partie.pieces[1][0].coups_possibles(partie)==[(5, 3), (3, 3), (2, 2), (2, 0),
4       (6, 0), (6, 2)]
5       assert partie.pieces[1][0].coups_legaux(partie)==[]
```

**Validation des tests**

```
collected 6 items

test_pièce.py ......

============================== 6 passed in 0.37s ===============================================
```

### 4.1.2 Tests de l'ÉtatJeu

**Tests des échecs** Nous avons fait différents tests pour voir si les méthodes echec et echec_et_mat fonctionnent correctement. Pour cela, on crée 3 situations :

- Une avec le joueur qui doit jouer en échec
- Une sans échec et donc sans mat
- Une autre où le joueur qui doit jouer est en échec et mat

```
1   def test_pat():
2       # 1ère situation de nulle, il n'y a pas d'échec et mat et aucun coup n'est possible
3       partie=init_partie_test("pat_1")
4       assert partie.pat()
5       # 2ème situation de nulle, seul les rois bougent
6       partie=init_partie_test("pat_2")
7       partie.pieces[1][0].odometre=40
8       assert partie.pat()
```

```
9
10  def test_echec():
11      partie=init_partie_test("avec_echec")
12      assert partie.echec()
13      partie=init_partie_test("sans_echec")
14      assert not partie.echec()
15
16  def test_mat():
17      partie=init_partie_test("avec_mat")
18      assert partie.echec_et_mat()
19      partie=init_partie_test("sans_mat")
20      assert not partie.echec_et_mat()
```

**Test de la promotion**

```
1   def test_promotion():
2   partie=init_partie_test("promotion")
3   partie.deplacer_piece((7,6),(7,7))
4   for piece in partie.pieces[not partie.trait]:
5       assert not isinstance(piece, Pion)
6   assert isinstance(partie.plateau[(7,7)], Dame)
```

**Validation des tests**

```
collected 4 items

test_EtatJeu.py ...

============================== 4 passed in 0.47s ===================================
```

## 4.2 Tests relatifs à l'IA

### 4.2.1 Tests de la valeur

Pour tester le calcul de la valeur, on a créé des plateaux simples où il est possible de calculer à la main facilement la valeur du plateau.

```
1   # Test valeur
2   def test_valeur():
3       # Test de situation finale
4       partie=init_partie_test("avec_mat")
5       assert partie.calcul_valeur()==-1000
6       partie=init_partie_test("pat_1")
7       assert partie.calcul_valeur()==0
```

```
    # Test centre et sous-centre
    partie=init_partie_test("controle_centre_et_ss_centre")
    assert partie.calcul_valeur()==0.4
    # Test pions alignés
    partie=init_partie_test("pions_allignes")
    assert partie.calcul_valeur()==-0.1
```

### 4.2.2 Tests d'existence des coups

Pour cette partie, on regarde si l'IA pour différentes profondeurs fait bien des coups existants.

```

# Test jouer un coup
# Test IA aléatoire (niveau == 0)
def test_IA_0():
    partie, bot1, bot2 = initialiser_plateau_bots(profondeur_blanc=0, profondeur_noir=0)
    # Blanc
    coup = bot1.jouer_coup(partie)
    assert coup[0] in partie.mouvements(True).keys()
    assert coup[1] in partie.mouvements(True)[coup[0]]
    partie.deplacer_piece(coup[0], coup[1])
    # Noir
    coup = bot2.jouer_coup(partie)
    assert coup[0] in partie.mouvements(False).keys()
    assert coup[1] in partie.mouvements(False)[coup[0]]

def test_minimax_profondeur_1():
    partie, bot1, bot2 = initialiser_plateau_bots(profondeur_blanc=1, profondeur_noir=1)
    # Blanc
    coup = bot1.jouer_coup(partie)
    assert coup[0] in partie.mouvements(True).keys()
    assert coup[1] in partie.mouvements(True)[coup[0]]
    partie.deplacer_piece(coup[0], coup[1])
    # Noir
    coup = bot2.jouer_coup(partie)
    assert coup[0] in partie.mouvements(False).keys()
    assert coup[1] in partie.mouvements(False)[coup[0]]

def test_minimax_profondeur_2():
    partie, bot1, bot2 = initialiser_plateau_bots(profondeur_blanc=2, profondeur_noir=2)
    # Blanc
    coup = bot1.jouer_coup(partie)
    assert coup[0] in partie.mouvements(True).keys()
    assert coup[1] in partie.mouvements(True)[coup[0]]
    partie.deplacer_piece(coup[0], coup[1])
```

```
    # Noir
    coup = bot2.jouer_coup(partie)
    assert coup[0] in partie.mouvements(False).keys()
    assert coup[1] in partie.mouvements(False)[coup[0]]


def test_minimax_profondeur_3():
    partie, bot1, bot2 = initialiser_plateau_bots(profondeur_blanc=3, profondeur_noir=3)
    # Blanc
    coup = bot1.jouer_coup(partie)
    assert coup[0] in partie.mouvements(True).keys()
    assert coup[1] in partie.mouvements(True)[coup[0]]
    partie.deplacer_piece(coup[0], coup[1])
    # Noir
    coup = bot2.jouer_coup(partie)
    assert coup[0] in partie.mouvements(False).keys()
    assert coup[1] in partie.mouvements(False)[coup[0]]

```

### 4.2.3 Tests de cohérence

On vérifie maintenant si les algorithmes alphabeta et minimax proposent le même coup.

```

# Test si minimax et alphabeta jouent le même coup (profondeur 2)
def test_alphabeta():
    plateaux = ["Sauvegarde_Test_IA\\Plateau_base","Sauvegarde_Test_IA\\suicide"]
    for plateau in plateaux:
        partie_ab, bot_blanc_ab, bot_noir_ab = initialiser_plateau_bots(plateau)
        partie_mm, bot_blanc_mm, bot_noir_mm = initialiser_plateau_bots(plateau)
        bot_blanc_mm.algo = "minimax"
        bot_noir_mm.algo = "minimax"
        if partie_ab.trait:
            assert bot_blanc_mm.jouer_coup(partie_ab) == bot_blanc_ab.jouer_coup(partie_ab)
            print(bot_blanc_mm.jouer_coup(partie_ab) == bot_blanc_ab.jouer_coup(partie_ab))
        else :
            assert bot_noir_mm.jouer_coup(partie_ab) == bot_noir_ab.jouer_coup(partie_ab)
            print(bot_noir_mm.jouer_coup(partie_ab) == bot_noir_ab.jouer_coup(partie_ab))

```

### 4.2.4 Tests de durées

Nous avons aussi vérifié si l'algorithme alphabeta était plus optimisé que minimax et que donc le temps d'exécution de chaque coup était plus court pour alphabeta que pour minimax et ce pour plusieurs profondeurs.

```
1
2  # Test différence de temps minimax alphabeta, différentes profondeurs
3  def test_duree_1():
4      plateaux = ["Sauvegarde_Test_IA\\Plateau_base","Sauvegarde_Test_IA\\suicide"]
5      rapports = []
6      for plateau in plateaux:
7          partie_ab, bot_blanc_ab, bot_noir_ab = initialiser_plateau_bots(plateau,1,1)
8          partie_mm, bot_blanc_mm, bot_noir_mm = initialiser_plateau_bots(plateau,1,1)
9          bot_blanc_mm.algo = "minimax"
10         bot_noir_mm.algo = "minimax"
11         if partie_ab.trait:
12             debut = time.time()
13             bot_blanc_mm.jouer_coup(partie_ab)
14             fin_mm = time.time()
15             bot_blanc_ab.jouer_coup(partie_ab)
16             rapport = (fin_mm-debut)/(time.time()-fin_mm)
17             print("alphabeta est "+str(round(rapport,3))+" plus rapide que minimax")
18             rapports.append(rapport)
19         else:
20             debut = time.time()
21             bot_noir_mm.jouer_coup(partie_ab)
22             fin_mm = time.time()
23             bot_noir_ab.jouer_coup(partie_ab)
24             rapport = (fin_mm-debut)/(time.time()-fin_mm)
25             print("alphabeta est "+str(round(rapport,3))+" plus rapide que minimax")
26             rapports.append(rapport)
27     print("moyenne profondeur 1 : "+str(round(average(rapports),3)))
28
29 def test_duree_2():
30     plateaux = ["Sauvegarde_Test_IA\\Plateau_base","Sauvegarde_Test_IA\\suicide"]
31     rapports = []
32     for plateau in plateaux:
33         partie_ab, bot_blanc_ab, bot_noir_ab = initialiser_plateau_bots(plateau,2,2)
34         partie_mm, bot_blanc_mm, bot_noir_mm = initialiser_plateau_bots(plateau,2,2)
35         bot_blanc_mm.algo = "minimax"
36         bot_noir_mm.algo = "minimax"
37         if partie_ab.trait:
38             debut = time.time()
39             bot_blanc_mm.jouer_coup(partie_ab)
40             fin_mm = time.time()
41             bot_blanc_ab.jouer_coup(partie_ab)
42             rapport = (fin_mm-debut)/(time.time()-fin_mm)
43             print("alphabeta est "+str(round(rapport,3))+" plus rapide que minimax")
44             rapports.append(rapport)
45         else:
```

```
46              debut = time.time()
47              bot_noir_mm.jouer_coup(partie_ab)
48              fin_mm = time.time()
49              bot_noir_ab.jouer_coup(partie_ab)
50              rapport = (fin_mm-debut)/(time.time()-fin_mm)
51              print("alphabeta est "+str(round(rapport,3))+" plus rapide que minimax")
52              rapports.append(rapport)
53      print("moyenne profondeur 2 : "+str(round(average(rapports),3)))
54
55
56  def test_duree_3():
57      plateaux = ["Sauvegarde_Test_IA\\Plateau_base","Sauvegarde_Test_IA\\suicide"]
58      rapports = []
59      for plateau in plateaux:
60          partie_ab, bot_blanc_ab, bot_noir_ab = initialiser_plateau_bots(plateau,3,3)
61          partie_mm, bot_blanc_mm, bot_noir_mm = initialiser_plateau_bots(plateau,3,3)
62          bot_blanc_mm.algo = "minimax"
63          bot_noir_mm.algo = "minimax"
64          if partie_ab.trait:
65              debut = time.time()
66              bot_blanc_mm.jouer_coup(partie_ab)
67              fin_mm = time.time()
68              bot_blanc_ab.jouer_coup(partie_ab)
69              rapport = (fin_mm-debut)/(time.time()-fin_mm)
70              print("alphabeta est "+str(round(rapport,3))+" plus rapide que minimax")
71              rapports.append(rapport)
72          else:
73              debut = time.time()
74              bot_noir_mm.jouer_coup(partie_ab)
75              fin_mm = time.time()
76              bot_noir_ab.jouer_coup(partie_ab)
77              rapport = (fin_mm-debut)/(time.time()-fin_mm)
78              print("alphabeta est "+str(round(rapport,3))+" plus rapide que minimax")
79              rapports.append(rapport)
80      print("moyenne profondeur 3 : "+str(round(average(rapports),3)))
81
82
```

---

#### 4.2.5 Tests de niveau de l'IA

Tester le niveau de notre IA est une tâche difficile.
Nous avons d'abord vérifié que l'IA (profondeur 3) voyait bien les échecs et mat disponibles à sa profondeur. Pour cela, nous avons crée 3 plateaux (avec des doublons pour chaque couleur) :

- L'IA peut être mise en échec et mat en 2 coups (elle doit l'éviter)
- L'IA peut mettre en échec et mat en 1 coup

- L'IA peut mettre en échec et mat en 1 coup, grâce à une promotion.

Voici les tests associés:

```python
# Test si minimax ne se suicide pas
def test_eviter_mat():
    # Initialisation d'un plateau où les blancs peuvent être mis en échec et mat en 1, on vérifie
    partie, bot_blanc, bot_noir = initialiser_plateau_bots("Sauvegarde_Test_IA\\suicide.fen")
    coup = bot_blanc.jouer_coup(partie)
    partie.deplacer_piece(coup[0], coup[1])
    coup = bot_noir.jouer_coup(partie)
    partie.deplacer_piece(coup[0], coup[1])
    assert not partie.echec_et_mat()

def test_mat_blanc():
    plateaux = ["Sauvegarde_Test_IA\\mat_en_1_b.fen","Sauvegarde_Test_IA\\mat_promotion_b.fen"]
    for plateau in plateaux:
        partie,bot_blanc,bot_noir = initialiser_plateau_bots(plateau)
        coup = bot_blanc.jouer_coup(partie)
        partie.deplacer_piece(coup[0],coup[1])
        assert partie.echec_et_mat()

def test_mat_noir():
    plateaux = ["Sauvegarde_Test_IA\\mat_en_1_n.fen","Sauvegarde_Test_IA\\mat_promotion_n.fen"]
    for plateau in plateaux:
        partie,bot_blanc,bot_noir = initialiser_plateau_bots(plateau)
        coup = bot_noir.jouer_coup(partie)
        partie.deplacer_piece(coup[0],coup[1])
        assert partie.echec_et_mat()

```

### Validation des tests

```
collected 11 items

test_IA.py ......

============================== 11 passed in 14.5s ==============================
```

Finalement, après avoir joué plussieurs parties, et testé l'IA contre des IA de niveaux établis, nous estimons l'IA de profondeur 2 ayant un elo de 700, et 750-800 pour la profondeur 3. La profondeur 3 n'améliore pas grandement le niveau, car elle ne permet pas de prévoir 2 réponses de l'adversaire.