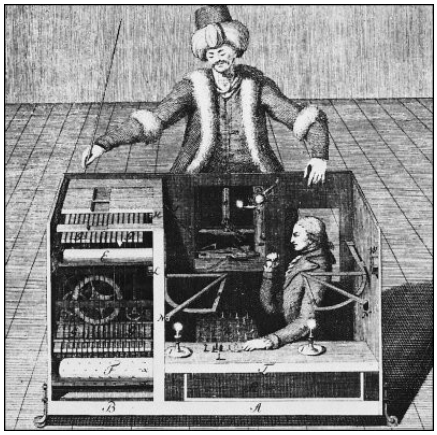


# *Chapitre 16*

# Programmer un jeu d'échecs

## 16.1. Nouveaux thèmes abordés dans ce chapitre

- Arbre de jeux
- Algorithme minimax et élagage alpha-bêta

## 16.2. Jeu d'échecs

Nous avons déjà vu le mouvement des pièces du jeu d'échecs au chapitre 14. Nous y reviendrons au § 16.3.3. Pour les règles complètes du jeu, nous renvoyons le lecteur par exemple à la page « Échecs » de Wikipédia : https://fr.wikipedia.org/wiki/Échecs.

Un des objectifs des premiers informaticiens a été de mettre au point des machines capables de jouer aux échecs, amis ce n'est qu'en 1997 que Deep Blue est devenu le premier ordinateur à battre un champion du monde en titre, dans un match qui l'opposait à Garry **Kasparov**. Cependant, ce rêve d'intelligence artificielle date de bien avant...

Le **Turc mécanique** (voir l'image en haut de cette page) est un canular célèbre construit à la fin du 18ème siècle : ce prétendu automate jouait aux échecs. Construit et dévoilé pour la première fois en 1770 par Johann Wolfgang **von Kempelen**, le mécanisme semble être en mesure de jouer contre un adversaire humain, ainsi que de résoudre le problème du cavalier (voir chapitre 14).

En réalité, le mécanisme n'était qu'une illusion permettant de masquer la profondeur réelle du meuble. Celui-ci possédait un autre compartiment secret dans lequel un vrai joueur pouvait se glisser, et manipuler le mannequin sans être vu de quiconque. L'automate était alors capable de jouer une vraie partie d'échecs contre un adversaire humain. Grâce au talent de ses joueurs cachés, le Turc mécanique remporta la plupart des parties d'échecs auxquelles il participa en Europe et en Amérique durant près de 84 ans, y compris contre certains hommes d'état tels que Napoléon **Bonaparte**, **Catherine II** de Russie et Benjamin **Franklin**.

### 16.2.1. Source du programme étudié dans ce chapitre

Programmer un jeu d'échecs de A à Z serait trop long et trop difficile pour des programmeurs peu expérimentés. Aussi, nous allons plutôt analyser un programme écrit par Jean-François **Gazet** : *Jepychess*, qui signifie à peu de chose près « JeffProd Simple Python Chess Program ». Ce programme a l'avantage d'être écrit en Python 3 et d'être disponible gratuitement et intégralement sur le site de partage « GitHub » : https://github.com/Tazeg/JePyChess.

### 16.2.2. Licence GNU GPL

Ce programme est en outre placé sous la *Licence publique générale GNU, version 2*. L'objectif de la licence GNU GPL est de garantir à l'utilisateur les droits suivants (appelés libertés) sur un programme informatique :

1. la liberté d'exécuter le logiciel, pour n'importe quel usage ;
2. la liberté d'étudier le fonctionnement d'un programme et de l'adapter à ses besoins, ce qui passe par l'accès aux codes sources ;
3. la liberté de redistribuer des copies ;
4. l'obligation de faire bénéficier des versions modifiées à la communauté.

La quatrième liberté passe par un choix : la deuxième autorisant de modifier un programme, il n'est pas tenu de publier une version modifiée tant qu'elle est pour un usage personnel ; par contre, en cas de distribution d'une version modifiée, la quatrième liberté amène l'obligation à ce que les modifications soient retournées à la communauté sous la même licence. Pour en savoir plus :
https://fr.wikipedia.org/wiki/Licence_publique_générale_GNU

## 16.3. Représentation du jeu

Les explications ci-dessous proviennent en grande partie du blog de l'auteur du programme :
http://fr.jeffprod.com/blog/2014/comment-programmer-un-jeu-dechecs.html

Avant de passer au programme, nous allons voir les différentes composantes d'un jeu d'échecs et comment les implémenter.

On peut décomposer un logiciel d'échecs en quatre parties :

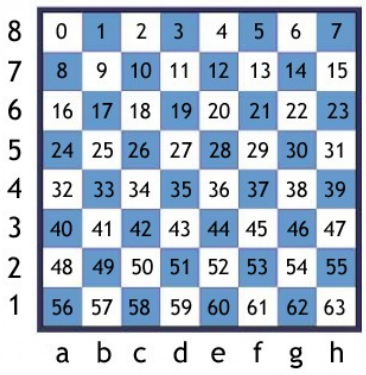
- un échiquier ;
- des pièces d'échecs ;
- un moteur de jeu ;
- une interface graphique.

Ainsi, le projet sera composé de quatre fichiers :

- **piece.py** : les attributs et les méthodes des pièces d'échecs ;
- **board.py** : les attributs et les méthodes de l'échiquier ;
- **engine.py** : l'intelligence artificielle du jeu, capable d'analyser une position de l'échiquier et déterminer le meilleur coup ;
- **main.py** : le programme principal qui affiche l'échiquier et répond aux sollicitations de l'utilisateur.

### 16.3.1. L'échiquier

Il existe plusieurs façons de représenter l'échiquier. Étant composé de 64 cases, le plus simple en programmation est d'utiliser un tableau de 64 éléments. Les indices de ce tableau vont de 0 à 63. Ainsi la case a8 correspond à 0, et h1 à 63, comme le montre le dessin ci-dessous.



```
    coord=[
        'a8','b8','c8','d8','e8','f8','g8','h8',
        'a7','b7','c7','d7','e7','f7','g7','h7',
        'a6','b6','c6','d6','e6','f6','g6','h6',
        'a5','b5','c5','d5','e5','f5','g5','h5',
        'a4','b4','c4','d4','e4','f4','g4','h4',
        'a3','b3','c3','d3','e3','f3','g3','h3',
        'a2','b2','c2','d2','e2','f2','g2','h2',
        'a1','b1','c1','d1','e1','f1','g1','h1'
        ]
```

Avec la définition du tableau ci-dessus, `coord[3]` retournera 'd8'.

Sur une case de l'échiquier sera éventuellement posée une pièce. Pour des raisons pratiques, on mettra toujours un objet-pièce sur chaque case de l'échiquier mais cet objet pourra être vide, ce qui se traduira par `Piece()`.

```
self.cases = [
    Piece('TOUR','noir'),Piece('CAVALIER','noir'),Piece('FOU','noir'),
    Piece('DAME','noir'),Piece('ROI','noir'),Piece('FOU','noir'),
    Piece('CAVALIER','noir'),Piece('TOUR','noir'),

    Piece('PION','noir'),Piece('PION','noir'),Piece('PION','noir'),
    Piece('PION','noir'),Piece('PION','noir'),Piece('PION','noir'),
    Piece('PION','noir'),Piece('PION','noir'),Piece('PION','noir'),

    Piece(),Piece(),Piece(),Piece(),Piece(),Piece(),Piece(),Piece(),

    Piece(),Piece(),Piece(),Piece(),Piece(),Piece(),Piece(),Piece(),

    Piece(),Piece(),Piece(),Piece(),Piece(),Piece(),Piece(),Piece(),

    Piece(),Piece(),Piece(),Piece(),Piece(),Piece(),Piece(),Piece(),

    Piece('PION','blanc'),Piece('PION','blanc'),Piece('PION','blanc'),
    Piece('PION','blanc'),Piece('PION','blanc'),Piece('PION','blanc'),
    Piece('PION','blanc'),Piece('PION','blanc'),

    Piece('TOUR','blanc'),Piece('CAVALIER','blanc'),Piece('FOU','blanc'),
    Piece('DAME','blanc'),Piece('ROI','blanc'),Piece('FOU','blanc'),
    Piece('CAVALIER','blanc'),Piece('TOUR','blanc')
]
```

Ainsi, en parcourant les indices (de 0 à 63) du tableau ci-dessus :

- `self.cases[1]` est un cavalier noir ;
- `self.cases[35]` est vide ;
- `self.cases[60]` est le roi blanc.

Outre ses cases, l'échiquier possède les attributs suivants :

- le camp qui a le trait (à qui est-ce de jouer ?) ;
- la case d'une prise en passant ;
- le nombre de coups joués depuis le début de la partie ;
- un historique des coups joués ;
- les droits au roque.

On va créer les méthodes suivantes dans l'objet `Echiquier` :

- générer la liste de tous les coups pour un camp donné ;
- possibilité de définir une position des pièces au format FEN (voir ex. 14.13) ;
- possibilité d'exporter cette position avec la notation FEN ;
- déplacer une pièce selon les règles du jeu ;
- annuler le dernier coup joué ;
- changer le joueur qui a le trait ;
- savoir si un roi est en échec ;
- savoir si une pièce est attaquée ;
- afficher l'échiquier en mode console ;
- afficher l'historique des coups joués ;
- donner une note à une position.

## 16.3.2. Les pièces

Comme vous le savez, les pièces sont : roi, dame, tour, cavalier, fou, pion. Certaines sont plus importantes que d'autres et il peut être grave de se les faire prendre. On leur attribue donc une valeur : traditionnellement 9 points pour la dame, 5 pour la tour, 3 pour le cavalier, 3 pour le fou, et 1 pour le pion. Comme le roi ne peut pas être pris, il n'est pas utile de lui donner une valeur.

```
nomPiece=(VIDE,'ROI','DAME','TOUR','CAVALIER','FOU','PION')
valeurPiece=(0,0,9,5,3,3,1)
```

Une pièce a les attributs suivants :

- un nom (ROI, DAME, TOUR, …) ;
- une couleur (blanc, noir) ;
- une valeur (définie en points ci-dessus).

## 16.3.3. Les déplacements des pièces

Les pièces peuvent être déplacées d'une case à une autre selon les règles propres à chacune. On va donc créer les fonctions permettant de trouver l'ensemble des cases de destination d'une pièce d'après sa case de départ.

Prenons un cas concret : une tour est placée sur la case **a4** (indice 32). Si on veut la déplacer d'une case vers la droite, pas de problème : il suffit d'ajouter 1. L'indice 33 correspond bien à la case **b4**. Par contre, si on essaie de la déplacer vers la gauche, ce qui n'est pas possible, elle se retrouverait sur la case 32−1=31 soit **h5** !

Robert **Hyatt** (né en 1948) est un informaticien américain, spécialiste de la programmation des jeux d'échecs.

Pour éviter ce débordement, nous utiliserons la méthode de Robert **Hyatt** appelée « mailbox ». Il s'agit de deux tableaux de 64 et 120 éléments, ressemblant à une boîte aux lettres selon son créateur, d'où son nom :

```
tab120 = (
        -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
        -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
        -1,  0,  1,  2,  3,  4,  5,  6,  7, -1,
        -1,  8,  9, 10, 11, 12, 13, 14, 15, -1,
        -1, 16, 17, 18, 19, 20, 21, 22, 23, -1,
        -1, 24, 25, 26, 27, 28, 29, 30, 31, -1,
        -1, 32, 33, 34, 35, 36, 37, 38, 39, -1,
        -1, 40, 41, 42, 43, 44, 45, 46, 47, -1,
        -1, 48, 49, 50, 51, 52, 53, 54, 55, -1,
        -1, 56, 57, 58, 59, 60, 61, 62, 63, -1,
        -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
        -1, -1, -1, -1, -1, -1, -1, -1, -1, -1
        )
```

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | -1 |
| -1 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | -1 |
| -1 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | -1 |
| -1 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | -1 |
| -1 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | -1 |
| -1 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 | -1 |
| -1 | 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 | -1 |
| -1 | 56 | 57 | 58 | 59 | 60 | 61 | 62 | 63 | -1 |
| -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |

```
tab64 = (
        21, 22, 23, 24, 25, 26, 27, 28,
        31, 32, 33, 34, 35, 36, 37, 38,
        41, 42, 43, 44, 45, 46, 47, 48,
        51, 52, 53, 54, 55, 56, 57, 58,
        61, 62, 63, 64, 65, 66, 67, 68,
        71, 72, 73, 74, 75, 76, 77, 78,
        81, 82, 83, 84, 85, 86, 87, 88,
        91, 92, 93, 94, 95, 96, 97, 98
        )
```

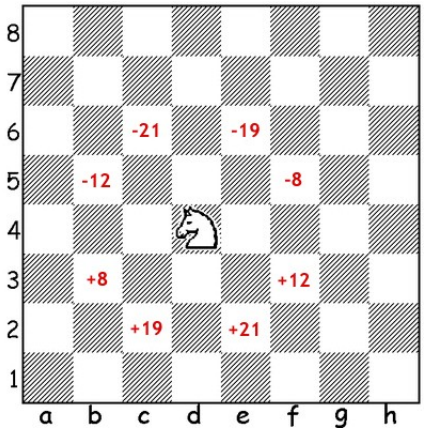
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 8 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 |
| 7 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 |
| 6 | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 |
| 5 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 |
| 4 | 61 | 62 | 63 | 64 | 65 | 66 | 67 | 68 |
| 3 | 71 | 72 | 73 | 74 | 75 | 76 | 77 | 78 |
| 2 | 81 | 82 | 83 | 84 | 85 | 86 | 87 | 88 |
| 1 | 91 | 92 | 93 | 94 | 95 | 96 | 97 | 98 |
| | a | b | c | d | e | f | g | h |

Dans le tableau « tab64 », la case **a4** correspond la valeur 61.

On déplace une tour d'une case à gauche en soustrayant 1, on obtient 60. Regardons le 60ème élément dans le tableau « tab120 » : c'est −1. Cette valeur indique une sortie de plateau.

Toujours d'après le tableau « tab64 », on peut définir les directions des déplacements possibles des pièces :

```
deplacements_tour = (-10,10,-1,1) # depl. vertical et horizontal
deplacements_fou = (-11,-9,11,9)  # depl. en diagonal
deplacements_cavalier = (-12,-21,-19,-8,12,21,19,8) # 8 depl. du cavalier
```




***Déplacements du cavalier (sur le tableau tab64)***

Inutile de spécifier des vecteurs pour la dame et le roi car ceux-ci se déplacent à la fois comme le fou et la tour, mais d'une seule case à la fois pour le roi.

## 16.3.3.1. Les déplacements du roi

(*Roque* = *Castle moves* en anglais)

Le roi est simple à déplacer (mouvement d'une case dans les huit directions possibles), sauf pour la règle du *roque*. Ce coup consiste à déplacer horizontalement le roi de deux cases vers l'une des deux tours de son camp et de placer cette tour sur la dernière case que le roi a traversée. On parle de *petit roque* pour la tour de la colonne **h** et de *grand roque*, pour la tour de la colonne **a**. Ci-contre, les blancs vont faire un grand roque et les noirs un petit roque.

Le petit roque se note 0-0 et le grand roque 0-0-0.

Le roque nécessite trois conditions :

- toutes les cases entre le roi de la tour doivent être vides ;
- ni le roi, ni la tour ne doivent avoir bougé avant le roque. Par conséquent, chaque camp ne peut roquer qu'une seule fois par partie.
- aucune des trois cases (de départ, de passage ou d'arrivée) traversée par le roi lors du roque ne doit être menacée par une pièce adverse. En particulier, le roque ne permet pas d'esquiver un échec. Ci-contre, les blancs ne peuvent pas roquer puisque le fou noir en **a6** menace la case **f1** (symbolisée par la croix rouge) par laquelle le roi devrait transiter pour faire son roque. En revanche, la tour, ainsi que sa case adjacente dans le cas du grand roque, peuvent être menacées.

```
    def pos2_roi(self,pos1,cAd,echiquier,dontCallIsAttacked=False):

        """Returns the list of moves for the king :
        - which is at square 'pos1'
        - which opponent color is 'cAd' (blanc,noir)
        - dontCallIsAttacked is set to avoid recursives calls
        between is_attacked() and gen_moves_list().
        """

        liste=[]
```

```
        for i in (self.deplacements_tour+self.deplacements_fou):
            n=self.tab120[self.tab64[pos1]+i]
            if n!=-1:
                if(echiquier.cases[n].isEmpty() or echiquier.cases[n].couleur==cAd):
                    liste.append((pos1,n,''))

        if dontCallIsAttacked:
            return liste # we just wanted moves that can attack

        # The current side to move is the opposite of cAd
        c=echiquier.oppColor(cAd)

        # Castle moves
        if c=='blanc':
            if echiquier.white_can_castle_63):
                # If a rook is at square 63
                # And if squares between KING and ROOK are empty
                # And if squares on which KING walks are not attacked
                # And if KING is not in check
                # Then we can add this castle move
                if echiquier.cases[63].nom=='TOUR' and \
                echiquier.cases[63].couleur=='blanc' and \
                echiquier.cases[61].isEmpty() and \
                echiquier.cases[62].isEmpty() and \
                echiquier.is_attacked(61,'noir')==False and \
                echiquier.is_attacked(62,'noir')==False and \
                echiquier.is_attacked(pos1,'noir')==False:
                    liste.append((pos1,62,''))
            if echiquier.white_can_castle_56):
                # If a rook is at square 56
                if echiquier.cases[56].nom=='TOUR' and \
                echiquier.cases[56].couleur=='blanc' and \
                echiquier.cases[57].isEmpty() and \
                echiquier.cases[58].isEmpty() and \
                echiquier.cases[59].isEmpty() and \
                echiquier.is_attacked(58,cAd)==False and \
                echiquier.is_attacked(59,cAd)==False and \
                echiquier.is_attacked(pos1,cAd)==False:
                    liste.append((pos1,58,''))
        elif c=='noir':
            if echiquier.black_can_castle_7:
                if echiquier.cases[7].nom=='TOUR' and \
                echiquier.cases[7].couleur=='noir' and \
                echiquier.cases[5].isEmpty() and \
                echiquier.cases[6].isEmpty() and \
                echiquier.is_attacked(5,cAd)==False and \
                echiquier.is_attacked(6,cAd)==False and \
                echiquier.is_attacked(pos1,cAd)==False:
                    liste.append((pos1,6,''))
            if echiquier.black_can_castle_0:
                if echiquier.cases[0].nom=='TOUR' and \
                echiquier.cases[0].couleur=='noir' and \
                echiquier.cases[1].isEmpty() and \
                echiquier.cases[2].isEmpty() and \
                echiquier.cases[3].isEmpty() and \
                echiquier.is_attacked(2,cAd)==False and \
                echiquier.is_attacked(3,cAd)==False and \
                echiquier.is_attacked(pos1,cAd)==False:
                    liste.append((pos1,2,''))

        return liste
```

## 16.3.3.2. Les déplacements de la tour

La tour se déplace verticalement et horizontalement. Elle peut se déplacer tant qu'elle n'est pas bloquée par une autre pièce (amie ou ennemie).

```
    def pos2_tour(self,pos1,cAd,echiquier):

        """Returns the list of moves for a ROOK :
        - at square number 'pos1' (0 to 63)
```

```
        - opponent color is cAd (blanc,noir)"""

        liste=[]

        for k in self.deplacements_tour:
            j=1
            while True:
                n=self.tab120[self.tab64[pos1] + (k * j)]
                if n!=-1:  # as we are not out of the board
                    if(echiquier.cases[n].isEmpty() or echiquier.cases[n].couleur==cAd):
                        liste.append((pos1,n,''))
                            # append the move if square is empty of opponent color
                else:
                    break  # stop if outside of the board
                if not echiquier.cases[n].isEmpty():
                    break  # destination square is not empty (opponent or not)
                           # then the rook won't pass through
                j=j+1

        return liste
```

## 16.3.3.3. Les déplacements du cavalier

Paradoxalement, le cavalier, qui a les mouvements les plus « bizarres », est la pièce la plus simple à déplacer informatiquement parlant, car c'est la seule pièce qui peut passer par-dessus d'autres pièces.

```
    def pos2_cavalier(self,pos1,cAd,echiquier):

        """Returns the list of moves for a KNIGHT :
        - at square number 'pos1' (0 to 63)
        - opponent color is cAd (blanc,noir)"""

        liste=[]

        for i in self.deplacements_cavalier:
            n=self.tab120[self.tab64[pos1]+i]
            if n!=-1:
                if(echiquier.cases[n].isEmpty() or echiquier.cases[n].couleur==cAd):
                    liste.append((pos1,n,''))

        return liste
```

## 16.3.3.4. Les déplacements du fou

Le fou se déplace en diagonale. Il peut se déplacer tant qu'il n'est pas bloqué par une autre pièce (amie ou ennemie).

```
    def pos2_fou(self,pos1,cAd,echiquier):

        """Returns the list of moves for a BISHOP :
        - at square number 'pos1' (0 to 63)
        - opponent color is cAd (blanc,noir)"""

        liste=[]

        for k in self.deplacements_fou:
            j=1
            while True:
                n=self.tab120[self.tab64[pos1] + (k * j)]
                if(n!=-1): # as we are not out of the board
                if(echiquier.cases[n].isEmpty() or echiquier.cases[n].couleur==cAd):
                    liste.append((pos1,n,''))
                            # append the move if square is empty of opponent color
                else:
                    break  # stop if outside of the board
                if not echiquier.cases[n].isEmpty():
                    break  # destination square is not empty (opponent or not)
                           # then the bishop won't pass through
```

```
                j=j+1
        return liste
```

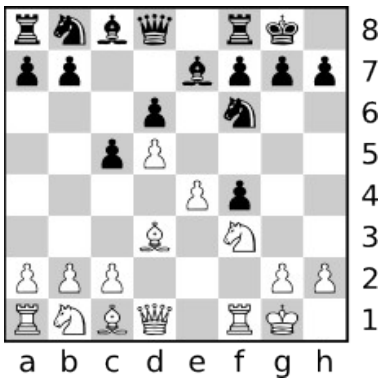
## 16.3.3.5. Les déplacements de la dame

On n'a pas besoin de programmer une fonction qui donne les coups possibles de la dame. Il suffit de combiner les cases atteignables par un fou et celles atteignables par une tour.

## 16.3.3.6. Les déplacements du pion

Le pion est assez complexe à gérer. Il peut seulement avancer devant lui et jamais reculer, en général d'une case (vide), mais éventuellement de deux cases lors de son premier déplacement.

Cependant, la prise d'une pièce adverse se fait en diagonale, et d'une case. Il existe en outre une prise spéciale, la *prise en passant*, qui obéit à des règles strictes : lorsqu'un pion se trouve sur la cinquième rangée et que l'adversaire avance un pion d'une colonne voisine de deux cases (les deux pions se retrouvent alors côte à côte sur la même rangée), le premier pion peut prendre le second. Pour effectuer la prise en passant, le joueur avance son pion en diagonale, derrière le pion à capturer (sur la sixième rangée) et ôte le pion adverse de l'échiquier. C'est le seul cas où on ne remplace une pièce adverse par la sienne. La *prise en passant* doit se faire immédiatement après le double pas du pion adverse ; elle n'est plus possible par la suite.




**Exemple :** les noirs viennent de jouer c7-c5.
Les blancs peuvent prendre le pion en **c5** en jouant **d5-c6 e.p.**, mais seulement ce coup-ci.
(e.p. est l'abréviation de « en passant »).

Pour finir, lorsqu'un pion atteint la dernière rangée, il est promu et peut se transformer en n'importe quelle pièce, sauf le roi. Après la promotion d'un pion blanc, il peut très bien y avoir deux dames blanches sur l'échiquier (ou plus).

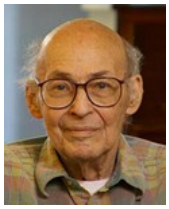
```
    def pos2_pion(self,pos1,couleur,echiquier):

        """Returns the list of moves for a PAWN :
        - at square number 'pos1' (0 to 63)
        - opponent color is cAd (blanc,noir)"""

        liste=[]

        # White PAWN ----------------------------------------------------
        if couleur=='blanc':

            # Upper square
            n=self.tab120[self.tab64[pos1]-10]
            if n!=-1:
                if echiquier.cases[n].isEmpty():
                    # If the PAWN has arrived to rank 8 (square 0 to 7)
                    if n<8:
                        # it will be promoted
                        liste.append((pos1,n,'q'))
                        liste.append((pos1,n,'r'))
                        liste.append((pos1,n,'n'))
```

```
                            liste.append((pos1,n,'b'))
                        else:
                            liste.append((pos1,n,''))

                # 2nd square if PAWN is at starting square
                if echiquier.ROW(pos1)==6:
                    # If the 2 upper squares are empty
                    if echiquier.cases[pos1-8].isEmpty() and echiquier.cases[pos1-16].isEmpty():
                        liste.append((pos1,pos1-16,''))

                # Capture upper left
                n=self.tab120[self.tab64[pos1]-11]
                if n!=-1:
                    if echiquier.cases[n].couleur=='noir' or echiquier.ep==n:
                        if n<8 # Capture + promote
                            liste.append((pos1,n,'q'))
                            liste.append((pos1,n,'r'))
                            liste.append((pos1,n,'n'))
                            liste.append((pos1,n,'b'))
                        else:
                            liste.append((pos1,n,''))

                # Capture upper right
                n=self.tab120[self.tab64[pos1]-9]
                if n!=-1:
                    if echiquier.cases[n].couleur=='noir' or echiquier.ep==n:
                        if n<8:
                            liste.append((pos1,n,'q'))
                            liste.append((pos1,n,'r'))
                            liste.append((pos1,n,'n'))
                            liste.append((pos1,n,'b'))
                        else:
                            liste.append((pos1,n,''))

            # Black PAWN ------------------------------------------------
            else:

                # Upper square
                n=self.tab120[self.tab64[pos1]+10]
                if n!=-1:
                    if echiquier.cases[n].isEmpty():
                        # PAWN has arrived to 8th rank (square 56 to 63),
                        # it will be promoted
                        if n>55 :
                            liste.append((pos1,n,'q'))
                            liste.append((pos1,n,'r'))
                            liste.append((pos1,n,'n'))
                            liste.append((pos1,n,'b'))
                        else:
                            liste.append((pos1,n,''))

                # 2nd square if PAWN is at starting square
                if echiquier.ROW(pos1)==1:
                    # If the 2 upper squares are empty
                    if(echiquier.cases[pos1+8].isEmpty() and echiquier.cases[pos1+16].isEmpty()):
                        liste.append((pos1,pos1+16,''))

                # Capture bottom left
                n=self.tab120[self.tab64[pos1]+9]
                if n!=-1:
                    if echiquier.cases[n].couleur=='blanc' or echiquier.ep==n:
                        if n>55:
                            liste.append((pos1,n,'q'))
                            liste.append((pos1,n,'r'))
                            liste.append((pos1,n,'n'))
                            liste.append((pos1,n,'b'))
                        else:
                            liste.append((pos1,n,''))

                # Capture bottom right
                n=self.tab120[self.tab64[pos1]+11]
                if n!=-1:
                    if echiquier.cases[n].couleur=='blanc' or echiquier.ep==n:
```

```
                        if n>55:
                            liste.append((pos1,n,'q'))
                            liste.append((pos1,n,'r'))
                            liste.append((pos1,n,'n'))
                            liste.append((pos1,n,'b'))
                        else:
                            liste.append((pos1,n,''))

            return liste
```

Toutes ces fonctions retournent la liste des coups possibles pour chaque pièce. Il s'agira ensuite de choisir parmi tous ces coups le meilleur. C'est le sujet du paragraphe suivant.

John **McCarthy**
(1927-2011)



Marvin Lee
**Minsky**
(1927-2016)

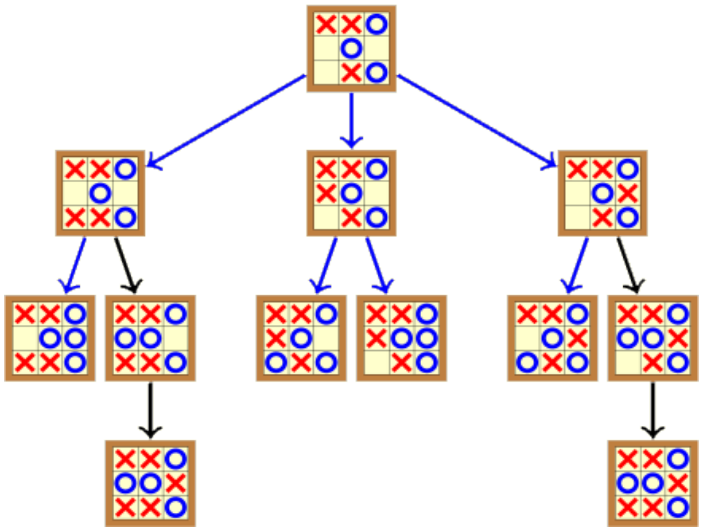
# 16.4. Intelligence artificielle

Le terme « intelligence artificielle », créé par John **McCarthy**, est souvent abrégé par le sigle « I. A. » (ou « A. I. » en anglais, pour *Artificial Intelligence*). Il est défini par l'un de ses créateurs, Marvin Lee **Minsky**, comme « la construction de programmes informatiques qui s'adonnent à des tâches qui sont, pour l'instant, accomplies de façon plus satisfaisante par des êtres humains, car elles demandent des processus mentaux de haut niveau tels que : l'apprentissage perceptuel, l'organisation de la mémoire et le raisonnement critique ».

On y trouve donc le côté « artificiel » atteint par l'usage des ordinateurs ou de processus électroniques élaborés et le côté « intelligence » associé à son but d'imiter le comportement. Cette imitation peut se faire dans le raisonnement, par exemple dans les jeux ou la pratique des mathématiques, dans la compréhension des langues naturelles, dans la perception : visuelle (interprétation des images et des scènes), auditive (compréhension du langage parlé) ou par d'autres capteurs, dans la commande d'un robot dans un milieu inconnu ou hostile.

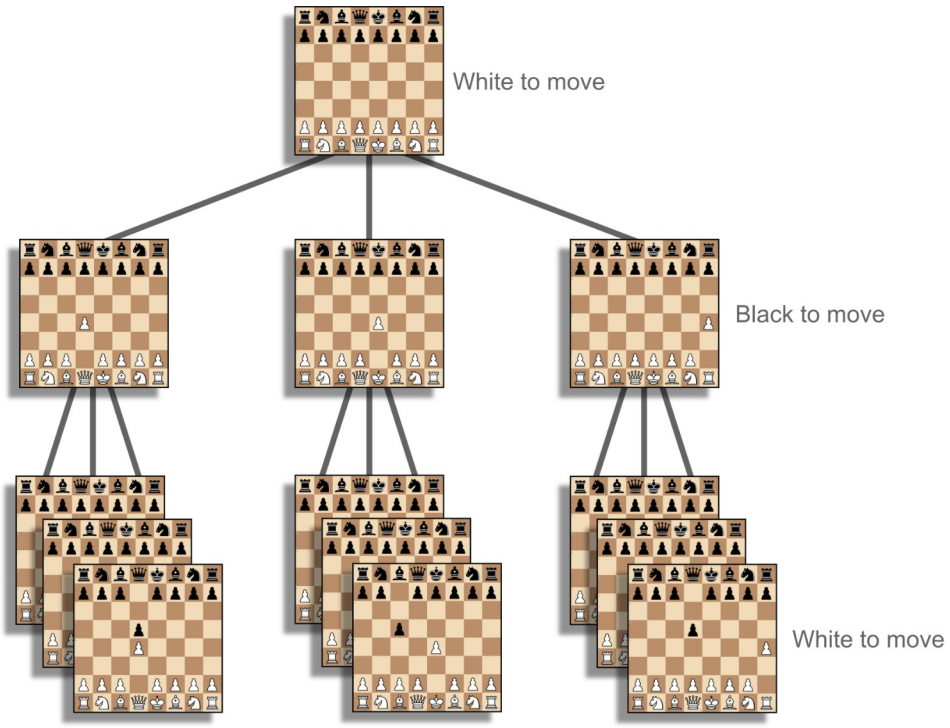
## 16.4.1. Arbre de jeu

Un arbre de jeu est une représentation imagée de toutes les positions que l'on peut atteindre à partir de celles du coup précédent. Voici l'arbre de jeu d'une fin de partie de jeu de morpion.



Selon le jeu, cet arbre devient gigantesque à partir de peu d'étages déjà. Aux échecs par exemple, les blancs ont le choix entre 20 coups (16 coups de pions et 4 coups de cavaliers) pour débuter la partie. Cela fait donc déjà 20 branches qui partent du sommet. Ensuite, les noirs ont aussi 20 coups à disposition. Cela fait déjà 400 positions possibles après 2 coups, puisque 20 branches partent des 20 positions précédentes ! Autant dire, que, sauf pour des jeux très simples, il est impossible de dessiner l'arbre de jeu complet. Aussi doit-on élaguer l'arbre pour éliminer les positions manifestement mauvaises pour le joueur. Nous verrons plus tard comment s'y prendre.

Voici ci-dessous un arbre partiel pour un début de partie d'échecs :




Insistons sur le fait que cet arbre ne devra pas être implémenté tel quel. Il est utilisé par l'homme pour (essayer de) se représenter toutes les suites de coups possibles, mais l'ordinateur n'en a pas besoin.

## 16.4.2. L'évaluation

Pour « évaluer » une position, c'est-à-dire savoir si elle est bonne ou mauvaise, il faut lui attribuer une valeur. Chaque position de l'échiquier est analysée selon plusieurs critères. Une note est calculée en fonction des pièces présentes et de nombreux bonus/malus. Voici la plus simple des fonctions d'évaluation, car elle ne prend en compte que le matériel :

```
def evaluer(self):

    WhiteScore=0
    BlackScore=0

    # Parsing the board squares from 0 to 63
    for pos1,piece in enumerate(self.cases):

        # Material score
        if(piece.couleur=='blanc'):
            WhiteScore+=piece.valeur
        else:
            # NB : here is for black piece or empty square
            BlackScore+=piece.valeur

    if self.side2move=='blanc':
        return WhiteScore-BlackScore
    else:
        return BlackScore-WhiteScore
```

On pourrait compléter cette fonction d'évaluation avec les bonus et malus suivants :

- pion passé : pion qui ne rencontrera plus de pion adverse et qui peut donc aller en promotion ;

- tour en 7ème rangée ;
- roi dans une colonne ouverte ;
- pions du roque avancés ;
- le roi a été déplacé et ne peut plus roquer ;
- pions doublés (deux pions dans la même colonne) ;
- pion isolé : pion qui ne peut plus être protégé par un autre pion.

On peut également accorder des bonus/malus selon l'avancement de la partie :

- en début de partie, il est préférable de roquer pour protéger le roi. Au contraire, en fin de partie, il faut placer le roi au centre pour éviter un mat sur un bord ou dans un coin de l'échiquier ;
- un cavalier peut atteindre un plus grand nombre de cases s'il est centré ;
- les pions prennent de la valeur s'ils approchent la 8ème rangée.

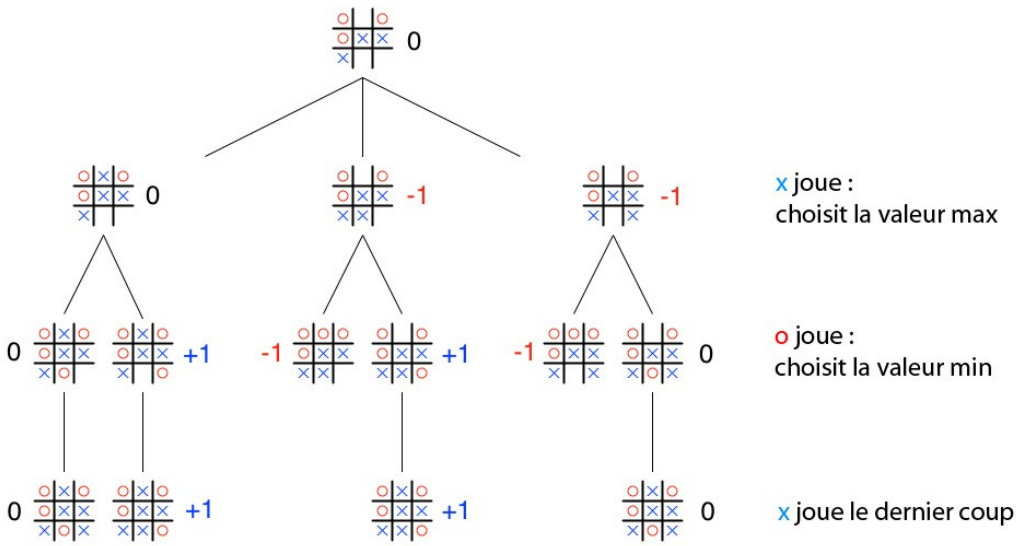
## 16.4.3. La recherche du meilleur coup

C'est à l'ordinateur de jouer. Il génère la liste de tous les coups possibles. Il déplace une pièce. Il attribue une note à la position. Il se met à la place de l'adversaire et il fait exactement la même chose, et ainsi de suite. Il faut donc préciser au moteur de recherche une profondeur fixe (par exemple 5 déplacements) ou un temps donné de réflexion. Le programme va donc élaborer un arbre de recherche. Il construit une arborescence de positions de l'échiquier où chacune est notée.

Cet arbre fait penser à un arbre généalogique : le père est en haut, il a un certain nombre de fils, qui ont eux-mêmes des fils, etc.

Prenons comme exemple un arbre de jeu de morpion. On descend dans l'arbre le plus possible, jusqu'à trouver une fin de partie : soit un match nul (auquel on attribue la valeur 0), une victoire des croix (valeur de +1) ou une victoire des ronds (valeur de -1).

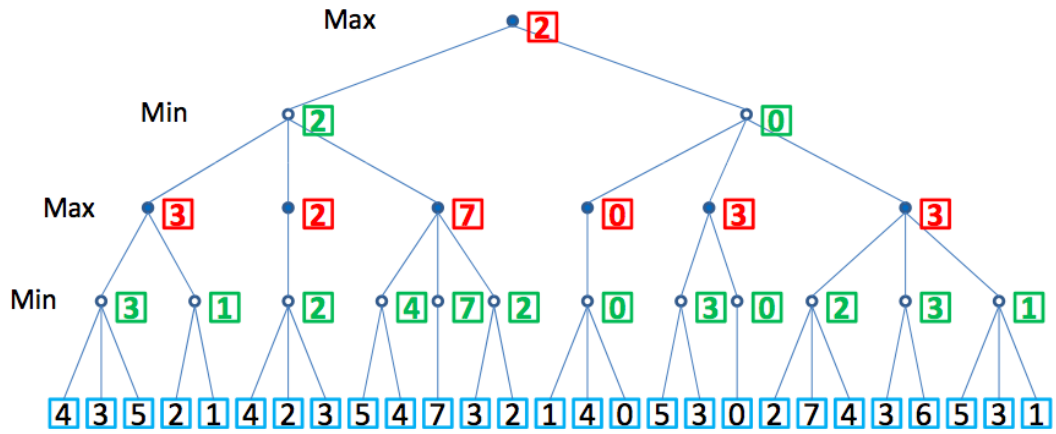
On « remonte » ensuite ces valeurs vers le haut de l'arbre en donnant au père la valeur minimale de ses fils si c'est les ronds qui jouent et la valeur maximale si c'est les croix qui jouent (d'où le nom d'algorithme minimax).




La valeur obtenue en haut de l'arbre est 0. Cela signifie que si les deux joueurs jouent parfaitement, l'issue sera un match nul (la position finale sera celle tout à gauche, en bas de l'arbre).

Aux échecs, contrairement au morpion, il est impossible d'explorer toutes les branches. L'élagage alpha-bêta permet de « couper » des branches.

Prenons un arbre de jeu quelconque dont les valeurs ont été calculées selon l'algorithme minimax.

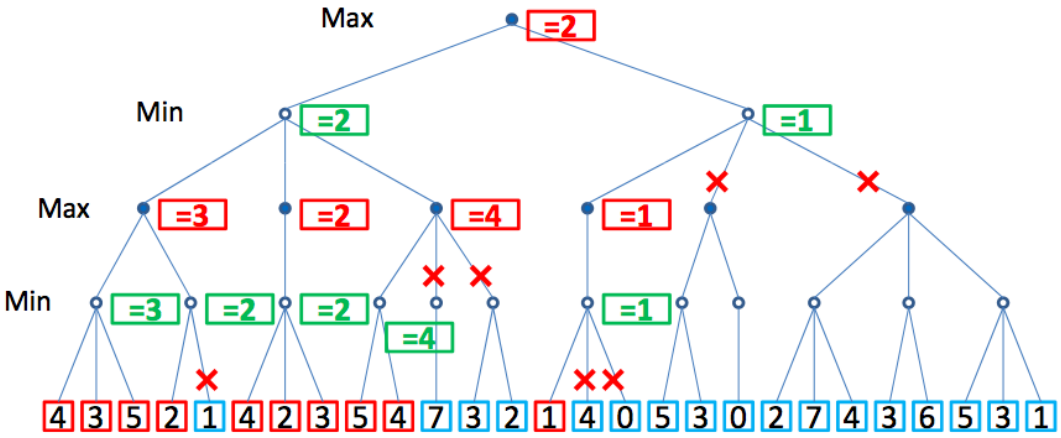



On pourrait se demander s'il est vraiment nécessaire d'évaluer toutes les positions. La réponse est non. Certaines branches peuvent être élaguées, car elles ne changeront pas la valeur attribuée au père.

Les positions finales sont examinées de gauche à droite.

Observons la 5ème position, évaluée à 1. Le père des 3 premières positions est évalué à 3 (le minimum de ses 3 fils). Pour l'instant le grand-père est évalué à 3 et le père des positions 4 et 5 vaut 2. Comme ce père va choisir la valeur minimale de ses fils, sa valeur ne pourra pas dépasser 2. Et comme le grand-père prend la valeur maximum de ses deux fils, sa valeur de 3 ne pourra pas changer, quelle que soit l'évaluation de la position 5. Il est donc inutile d'explorer cette branche.

Autre cas. La 10ème position a été explorée. On a remonté la valeur 4 (min(5,4)) vers le père et le grand-père. La valeur du grand-père ne pourra pas descendre en dessous de 4, puisque l'on va prendre la valeur maximale des fils. Mais comme l'arrière-grand-père va prendre la valeur minimale de ses fils, il est inutile d'explorer d'avantage cette branche.




En bleu, les 17 positions (sur 27) qui n'ont pas dû être explorées grâce à l'élagage alpha-bêta.

Voici l'algorithme alpha-bêta de JePyChess. Le meilleur coup est enregistré dans un tableau nommé « Triangular PV-Table » (PV pour *Principal Variation*). Il s'agit d'un tableau d'éléments indexés par le numéro de profondeur par rapport au début de la recherche. Il a pour but de collecter la meilleure ligne de coups successifs. Les éléments sont les meilleurs coups trouvés et enregistrés dans alpha-bêta. Il est donc recommandé de suivre cette ligne de coups afin de permettre un élagage efficace, puis les coups avec prise.

```
    def alphabeta(self,depth,alpha,beta,b):

        # We arrived at the end of the search : return the board score
        if depth==0:
            return b.evaluer()
            # TOTO : return quiesce(alpha,beta)

        self.nodes+=1
        self.pv_length[b.ply] = b.ply

        # Do not go too deep
        if b.ply >= self.MAX_PLY-1:
            return b.evaluer()

        # Extensions
        # If king is in check, let's go deeper
        chk=b.in_check(b.side2move) # 'chk' used at the end of func too
        if chk:
            depth+=1

        # Generate all moves for the side to move. Those who
        # let king in check will be processed in domove()
        mList=b.gen_moves_list()

        f=False # flag to know if at least one move will be done
        for i,m in enumerate(mList):

            # Do the move 'm'.
            # If it lets king in check, undo it and ignore it
            # remind : a move is defined with (pos1,pos2,promote)
            # i.e. : 'e7e8q' is (12,4,'q')
            if not b.domove(m[0],m[1],m[2])):
                continue

            f=True # a move has passed

            score=-self.alphabeta(depth-1,-beta,-alpha,b)

            # Unmake move
            b.undomove()

            if score>alpha:

                # this move caused a cutoff,
                if(score>=beta):
                    return beta
                alpha = score

                # Updating the triangular PV-Table
                self.pv[b.ply][b.ply] = m
                j = b.ply + 1
                while j<self.pv_length[b.ply+1]:
                    self.pv[b.ply][j] = self.pv[b.ply+1][j]
                    self.pv_length[b.ply] = self.pv_length[b.ply + 1]
                    j+=1

        # If no move has been done : it is DRAW or MAT
        if not f:
            if chk:
                return -self.INFINITY + b.ply # MAT
            else:
                return 0 # draw

        return alpha
```

## 16.5. Améliorations possibles

Le code a été écrit au plus simple. Il est possible par exemple de l'optimiser pour augmenter la vitesse de calcul.

Il reste également à implémenter :

- move ordering : suivre la variation principale et les captures de pièces en premier pour la recherche des meilleurs coups ;
- la règle des 50 coups sans prise (partie nulle) ;
- la règle des 3 répétitions de position (partie nulle) ;
- la gestion du temps (partie en blitz, recherche par temps fixe…) ;
- détecter les parties nulles pour matériel insuffisant ;
- bibliothèque d'ouvertures ;
- base de données de fins de partie ;
- recherche « quiescent » : au lieu de stopper net sur la note d'une position, si une pièce vient d'être prise, il peut s'avérer utile de continuer la recherche des prises uniquement, ceci afin de limiter l'effet d'horizon.


***Exercice 16.1***

Pouvez-vous répondre aux questions suivantes ?

1. Qu'est-ce que la « mailbox » ? À quoi sert-elle ?
2. Qu'est-ce qu'un arbre de jeu ?
3. Qu'est-ce que l'algorithme minimax ?
4. À quoi sert l'élagage alpha-bêta et comment cela fonctionne-t-il ?
5. Qu'est-ce qu'une fonction d'évaluation ? Quelle est la fonction utilisée dans le programme étudié ?

# 16.6. Interface graphique

L'interface du programme créé par Jean-François **Gazet** est des plus rudimentaires : tous les déplacements se font par l'intermédiaire du clavier et l'échiquier est affiché à l'écran sous forme de chaînes de caractères.

Résumons les fonctions qui existent dans les différents modules avant de nous lancer dans la réalisation d'une interface graphique.

## 16.6.1. Le module « piece »

Le fichier **piece.py** ne contient que la classe `piece`, qui gère la création et les déplacements des pièces.

Les variables :

- `VIDE='.'`
- `nomPiece=(VIDE,'ROI','DAME','TOUR','CAVALIER','FOU','PION')`
- `valeurPiece=(0,0,9,5,3,3,1)`
- les déplacement des pièces selon la description du § 16.3.3
  - `tab120` et `tab64`
  - `deplacements_tour=(-10,10,-1,1)`
  - `deplacements_fou=(-11,-9,11,9)`
  - `deplacements_cavalier=(-12,-21,-19,-8,12,21,19,8)`

Les fonctions :

```
    def __init__(self,nom=VIDE,couleur=''):

        """Creating a piece object, with its attributes :
        - 'nom' as name (ROI, DAME...);
        - 'couleur' as color (blanc,noir);
        - 'valeur' as its value"""
```

```
    def isEmpty(self):

        """Returns TRUE or FALSE if this piece object is defined,
        As any square on board can have a piece on it, or not,
        we can set a null piece on a square."""
```

```
    def pos2_roi(self,pos1,cAd,echiquier,dontCallIsAttacked=False):

        """Returns the list of moves for the king :
        - which is at square 'pos1'
        - which opponent color is 'cAd' (blanc,noir)
        - dontCallIsAttacked is set to avoid recursives calls
        between is_attacked() and gen_moves_list()."""
```

```
    def pos2_tour(self,pos1,cAd,echiquier):

        """Returns the list of moves for a ROOK :
        - at square number 'pos1' (0 to 63)
        - opponent color is cAd (blanc,noir)"""
```

```
    def pos2_cavalier(self,pos1,cAd,echiquier):

        """Returns the list of moves for a KNIGHT :
        - at square number 'pos1' (0 to 63)
        - opponent color is cAd (blanc,noir)"""
```

```
    def pos2_fou(self,pos1,cAd,echiquier):

        """Returns the list of moves for a BISHOP :
        - at square number 'pos1' (0 to 63)
        - opponent color is cAd (blanc,noir)"""
```

```
    def pos2_pion(self,pos1,couleur,echiquier):

        """Returns the list of moves for a PAWN :
        - at square number 'pos1' (0 to 63)
        - opponent color is cAd (blanc,noir)"""
```

## 16.6.2. Le module « board »

Le fichier **board.py** ne contient que la classe `board`, qui gère l'échiquier : mouvements possibles des pièces, promotion, prise en passant, alternance des joueurs, etc.

La variable :

```
# Names of the 64 squares
    coord=[
        'a8','b8','c8','d8','e8','f8','g8','h8',
        'a7','b7','c7','d7','e7','f7','g7','h7',
        'a6','b6','c6','d6','e6','f6','g6','h6',
        'a5','b5','c5','d5','e5','f5','g5','h5',
        'a4','b4','c4','d4','e4','f4','g4','h4',
        'a3','b3','c3','d3','e3','f3','g3','h3',
        'a2','b2','c2','d2','e2','f2','g2','h2',
        'a1','b1','c1','d1','e1','f1','g1','h1',
        ]
```

Les fonctions :

```
    def init(self):

        "Init the chess board at starting position"

        # Chessboard has 64 squares, numbered from 0 to 63 (a8 to h1)
        # Placing pieces ('cases' is 'square' in french :)
```

```
    def gen_moves_list(self,color='',dontCallIsAttacked=False):

        """Returns all possible moves for the requested color.
        If color is not given, it is considered as the side to move.
```

```
            dontCallIsAttacked is a boolean flag to avoid recursive calls,
            due to the actually wrotten is_attacked() function calling
            this gen_moves_list() function.
            A move is defined as it :
            - the number of the starting square (pos1)
            - the number of the destination square (pos2)
            - the name of the piece to promote '','q','r','b','n'
              (queen, rook, bishop, knight)
            """

        def setboard(self,fen):

            """Set the board to the FEN position given. i.e. :
            rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - - 0
            Returns TRUE or FALSE if done or not.
            If not : print errors.
            """

        def getboard(self):

            """Returns the FEN notation of the current board. i.e. :
            rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - - 0
            """

        def domove(self,depart,arrivee,promote):

            """Move a piece on the board from the square numbers
            'depart' to 'arrivee' (0..63) respecting rules :
            - prise en passant
            - promote and under-promote
            - castle rights
            Returns :
            - TRUE if the move do not let king in check
            - FALSE otherwise and undomove is done.
            """

        def undomove(self):

            "Undo the last move in history"

        def changeTrait(self):

            "Change the side to move"

        def oppColor(self,c):

            "Returns the opposite color of the 'c' color given"

        def in_check(self,couleur):

            """Returns TRUE or FALSE
            if the KING of the given 'color' is in check"""

        def is_attacked(self,pos,couleur):

            """Returns TRUE or FALSE if the square number 'pos' is a
            destination square for the color 'couleur'.
            If so we can say that 'pos' is attacked by this side.
            This function is used for 'in check' and for castle moves."""

        def render(self):

            "Display the chessboard in console mode"

        def caseStr2Int(self,c):

            """'c' given in argument is a square name like 'e2'
            "This functino returns a square number like 52"""

        def caseInt2Str(self,i):

            """Given in argument : an integer between 0 and 63
            Returns a string like 'e2'"""
```

```
    def showHistory(self):

        "Displays the history of the moves played"
```

```
    def evaluer(self):

        """A wonderful evaluate() function
        returning actually only the material score"""
```

### 16.6.3. Le module « engine »

Le fichier **engine.py** ne contient que la classe `engine`, qui implémente l'intelligence artificielle du programme. C'est notamment là que l'on parcourt l'arbre de jeu avec l'algorithme alpha-bêta.

```
    def __init__(self):

        self.MAX_PLY=32
        self.pv_length=[0 for x in range(self.MAX_PLY)]
        self.INFINITY=32000
        self.init()
```

```
    def usermove(self,b,c):

        """Move a piece for the side to move, asked in command line.
        The command 'c' in argument is like 'e2e4' or 'b7b8q'.
        Argument 'b' is the chessboard.
        """
```

```
    def chkCmd(self,c):

        """Check if the command 'c' typed by user is like a move,
        i.e. 'e2e4','b7b8n'...
        Returns '' if correct.
        Returns a string error if not.
        """
```

```
    def search(self,b):

        """Search the best move for the side to move,
        according to the given chessboard 'b'
        """
```

```
    def alphabeta(self,depth,alpha,beta,b):
```

```
    def print_result(self,b):

        "Check if the game is over and print the result"
```

```
    def clear_pv(self):

        "Clear the triangular PV table containing best moves lines"
```

```
    def setboard(self,b,c):

        """Set the chessboard to the FEN position given by user with
        the command line 'setboard ...'.
        'c' in argument is for example :
        'setboard 8/5k2/5P2/8/8/5K2/8/8 w - - 0 0'
        """
```

```
    def setDepth(self,c):

        """'c' is the user command line, i.e. 'sd [x]'
        to set the search depth.
        """
```

```
    def perft(self,c,b):

        """PERFformance Test :
        This is a debugging function through the move generation tree
        for the current board until depth [x].
        'c' is the command line written by user : perft [x]
```

```
            """

        def legalmoves(self,b):

            "Show legal moves for side to move"

        def getboard(self,b):

            """The user requests the current FEN position
            with the command 'getboard'"""

        def newgame(self,b):

        def bench(self,b):

            """Test to calculate the number of nodes a second.
            The position used is the 17th move of the game :
            Bobby Fischer vs. J. Sherwin, New Jersey State
            Open Championship, 9/2/1957 :
            1rb2rk1/p4ppp/1p1qp1n1/3n2N1/2pP4/2P3P1/PPQ2PBP/R1B1R1K1 w - - 0 1
            The engine searches to a given depth, 3 following times.
            The kilonodes/s is calculated with the best time.
            """

        def undomove(self,b):

            "The user requested a 'undomove' in command line"

        def get_ms(self):

        def init(self):
```


***Exercice 16.2***

Créez une interface graphique du jeu d'échecs proposé dans ce chapitre. Il faudra principalement modifier le fichier **main.py**. Il faudra avoir bien compris les trois autres modules pour pouvoir les réutiliser, peut-être en les modifiant légèrement : **piece.py**, **board.py** et **engine.py**.

C'est un exercice long et difficile qui vous prendra probablement plusieurs dizaines d'heures.

Conseil : procédez par étapes et n'essayez surtout pas de tout faire en même temps !

## 16.7. Bibliothèque d'ouvertures

En jouant contre le programme que nous avons étudié, on constate, si l'on est un joueur d'échecs, que l'ordinateur joue des débuts de parties très étranges (et en plus toujours les mêmes). Cela n'est pas étonnant : vu que la fonction d'évaluation est uniquement basée sur l'avantage matériel, et que généralement les prises interviennent plus tard dans la partie, tous les coups sont jugés plus ou moins équivalents par l'intelligence artificielle.

La solution est de doter ce programme d'une bibliothèque d'ouvertures (*opening book* en anglais). Il s'agit d'un fichier où sont répertoriées des suites de premiers coups classiques (il y a notamment l'ouverture italienne, l'espagnole, la défense Pirc, la nimzo-indienne, etc.). Tout joueur d'échecs se doit de bien connaître ces ouvertures et l'ordinateur devra faire de même.

On va donc procéder ainsi pour améliorer le début de partie :

- on va donner à l'ordinateur quelques centaines d'ouvertures ;
- au début, le programme sera en mode « ouverture » ;
- s'il commence la partie, le programme jouera le premier coup d'une de ces ouvertures prise au hasard ;
- le programme ira fouiller dans la liste d'ouvertures pour trouver celle qui est en train de se jouer, et il se contentera de jouer les coups de cette ouverture tant qu'il le pourra ;
- quand la liste de coups est terminée ou lorsque l'adversaire ne jouera plus le coup « classique », le programme quittera le mode ouverture et utilisera son I.A.


### Exercice 16.3

Vous trouverez sur www.apprendre-en-ligne.net/pj/echecs/ un fichier nommé **book.txt** qui contient plusieurs centaines d'ouvertures données sous la forme :

**g1f3 g8f6 c2c4 b7b6 g2g3**

Chaque ligne correspond à une ouverture. Le nombre de coups est variable d'une ligne à l'autre.
Cette bibliothèque provient de la page de Tom **Kerrigan** (www.tckerrigan.com/Chess/TSCP).

Implémentez cette bibliothèque d'ouvertures comme expliqué dans le § 16.7. Modifiez le fichier **main.py.**


### Exercice 16.4

Le programme de l'ex. 16.3 n'est pas encore terminé (loin s'en faut, voir § 16.5). Il y a encore beaucoup de fonctionnalités qui ne sont pas implémentées. Par exemple, lorsqu'un pion est promu, il n'est pas automatiquement remplacé par une dame. Il est parfois plus intéressant de le remplacer par une tour ou un cavalier (ou même un fou).

Modifiez le programme de l'ex. 16.3 pour que le joueur puisse choisir la pièce qui remplacera le pion promu.

## 16.8. Stockfish

Il existe évidemment bien d'autres programmes d'échecs open source. Avant 2018, le plus performant d'entre eux s'appelait *Stockfish 8*. Il a en effet remporté la finale 2016 du *Thoresen Chess Engine Championship*, considéré comme le championnat du monde non-officiel des programmes d'échecs.

Il est écrit en C++ et est disponible à l'adresse https://stockfishchess.org/. Le programme provient de *Glaurung*, un moteur de jeu d'échecs open source créé par Tord **Romstad** et sorti en 2004. Marco **Costalba** a créé *Stockfish* 1.0 en novembre 2008 en reprenant la version 2.1 de *Glaurung* en corrigeant quelques bugs et en incluant des améliorations.

## 16.9. AlphaZero

En décembre 2017, le monde des échecs a été secoué lorsque Stockfish a été vaincu lors d'un match à sens unique. Il a été battu par un programme informatique inconnu qui semblait venir d'un autre monde - AlphaZero.

AlphaZero a été développé Demis **Hassabis** de la société de recherche et d'intelligence artificielle DeepMind, rachetée par Google. Il s'agit d'un programme informatique qui a atteint un niveau de jeu pratiquement impensable en utilisant uniquement l'apprentissage par renforcement pour entraîner ses réseaux neuronaux. En d'autres termes, on ne lui a donné que les règles du jeu, puis il a joué contre lui-même plusieurs millions de fois (44 millions de parties au cours des neuf premières heures, selon DeepMind).

AlphaZero utilise ses réseaux neuronaux pour effectuer des évaluations extrêmement poussées des positions, ce qui lui évite de devoir examiner plus de 70 millions de positions par seconde (comme le fait Stockfish). Selon DeepMind, AlphaZero a atteint les critères nécessaires pour vaincre Stockfish en seulement quatre heures.

AlphaZero utilise un hardware spécifique souvent qualifié de « supercalculateur de Google », bien que DeepMind ait depuis précisé qu'AlphaZero fonctionnait sur quatre unités de traitement du tenseur (TPU) dans ses parties.

En décembre 2017, DeepMind a publié un rapport de recherche qui annonçait qu'AlphaZero avait facilement vaincu Stockfish dans un match de 100 parties. AlphaZero a ensuite dominé Stockfish dans un second match composé de 1000 parties ; les résultats ont été publiés dans un article à la fin de l'année 2018.

# 16.10. Références

## 16.10.1 Sources utilisées pour ce chapitre


- Chess.com : AlphaZero, <https://www.chess.com/fr/terms/alphazero-echecs>
- Jean-François Gazet, *Comment programmer un jeu d'échecs*, <http://fr.jeffprod.com/blog/2014/comment-programmer-un-jeu-dechecs.html>
- *Tom Kerrigan's Simple Chess Program*, <http://www.tckerrigan.com/Chess/TSCP>
- Wikipédia : *Le Turc mécanique*, <https://fr.wikipedia.org/wiki/Turc_mécanique>
- Wikipédia : *Licence publique générale GNU*, <https://fr.wikipedia.org/wiki/Licence_publique_générale_GNU>
- Wikipédia : *Intelligence artificielle*, <https://fr.wikipedia.org/wiki/Intelligence_artificielle>
- Wikipédia : *Top Chess Engine Championship*, <https://fr.wikipedia.org/wiki/Top_Chess_Engine_Championship>
- Wikipédia : *Stockfish (programme d'échecs)*, <https://fr.wikipedia.org/wiki/Stockfish_(programme_d'échecs)>


## 16.10.2 Pour aller plus loin


- *Computer Games I & II*, édité par David N. L. Levy, 1988
- *Échecs et C : Initiation à l'analyse et à la programmation du jeu d'échecs*, Yann Takvorian, 1993
- *Techniques de programmation des jeux*, David Levy, Editions du PSI, 1983
- *How Computers Play Chess,* David N. L. Levy & Monty Newborn, Ishi Press, 2009
- *Intelligence artificielle : une approche ludique*, Tristan Cazenave, Ellipse, 2011


*Chess IBM 704 (1957)*


# 16.11. Ce que vous avez appris dans ce chapitre

- Nous avons fait dans ce chapitre ce que tout programmeur sera amené à faire un jour : reprendre et modifier le programme d'un autre. Ce n'est pas un exercice facile, car chaque programmeur a son style d'écriture, un peu comme un écrivain ou un dessinateur. Il faut donc d'abord comprendre comment le programme fonctionne avant de pouvoir le modifier. Cela montre bien la grande importance de la documentation fournie avec le programme et des commentaires placés dans le code.
- Vous avez vu les grandes idées qui se cachent dans un programme de jeu d'échecs.