Minimax : vers l'efficacité des stratégies dans le jeu d'échecs

Houssam Eddine Touil

N° SCEI 32471

June 10, 2024

Contexte

Le domaine des jeux constitue un terrain d'étude fascinant en intelligence artificielle, exigeant des prises de décision stratégiques dans un environnement compétitif.

Figure: Le moteur d'échecs Stockfish

Figure: Le programme d'intelligence AlphaGo

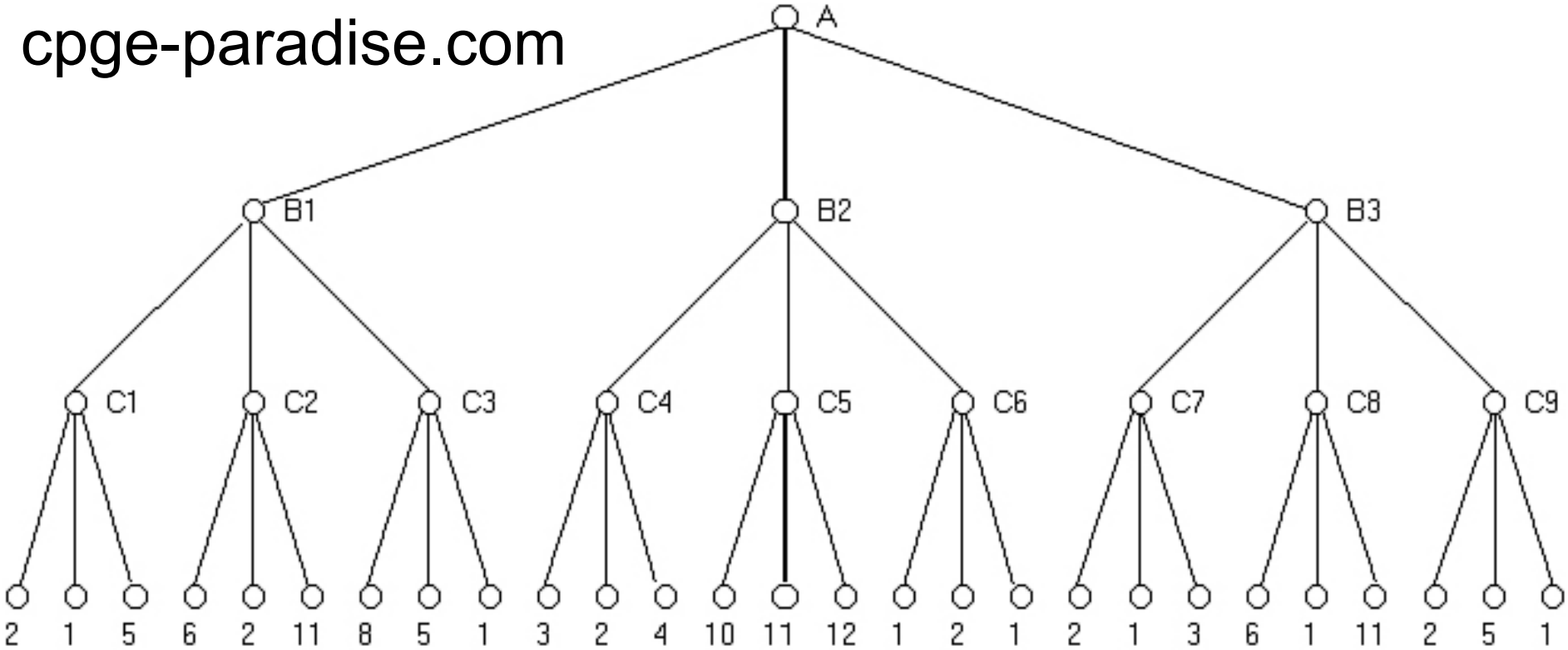


Figure: Arbre du jeu

Problématique

Comment appliquer l'algorithme Minimax dans les positions finales des échecs pour contribuer à des performances optimales ?

Plan de travail

- Notions-Clés
- Présentation du théorème de von Neumann
- Application de l'algorithme Minimax pour trouver une stratégie optimale
- Conception de la fonction d'évaluation pour le jeu d'échecs
- Tests et résultats de l'algorithme Minimax pour les parties finales du jeu d'échecs

Notions-Clés

- Jeux de deux joueurs à somme nulle : Jeux où le gain total des joueurs est équilibré
- Stratégies (pures et mixtes): Une stratégie est constituée d'un vecteur $(p_1, \cdots, p_n)$ avec $p_i \geqslant 0$, où

$$\sum_{i=1}^{n} p_i = 1$$

L'idée est que le joueur A jouera la stratégie i avec une probabilité p_i. Une stratégie est dite pure si elle a la forme $(0, \cdots, 0, 1, 0, \cdots, 0)$, c'est-à-dire si elle ne met de probabilité non nulle que sur une seule action. Sinon, la stratégie est dite mixte.

Le Théorème de Von Neumann

Énoncé du théorème

Tout jeu à somme nulle et à information parfaite possède au moins une stratégie mixte optimale pour chaque joueur.

- Le théorème de Von Neumann est fondamental en théorie des jeux.
- Il garantit l'existence de stratégies optimales pour tous les jeux à somme nulle et à information parfaite.
- Ce théorème a ouvert la voie à des algorithmes d'optimisation tels que Minimax et ses variantes.

Énoncé formel du théorème

Pour m entier strictement positif, notons Δ_m l'ensemble des vecteurs colonnes comportant m coefficients réels positifs ou nuls dont la somme vaut 1.
Soit A une matrice réelle (n,k) représentant un jeu à somme nulle. On a l'identité :

$$\max_{X \in \Delta_k} \min_{Y \in \Delta_n} Y^T A X = \min_{Y \in \Delta_n} \max_{X \in \Delta_k} Y^T A X$$

Où Y est un vecteur de taille n représentant les probabilités des stratégies pures du joueur 1
X est un vecteur de taille k représentant les probabilités des stratégies pures du joueur 2.

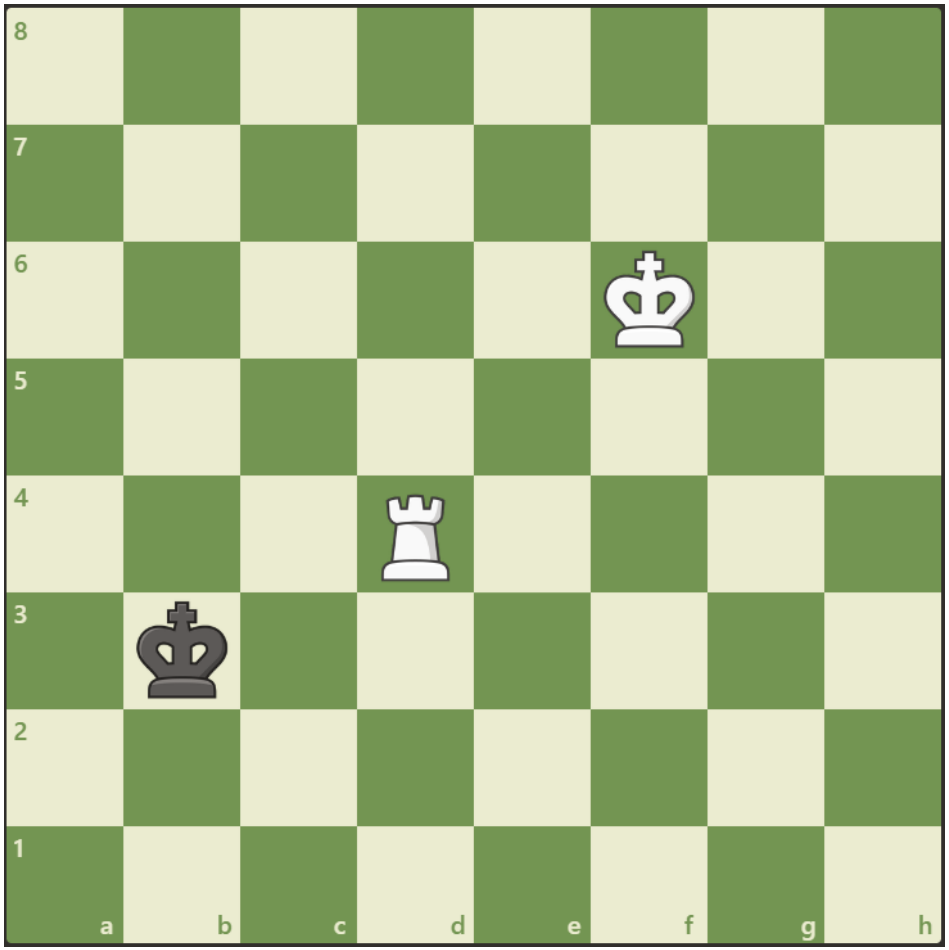
Application de l'algorithme Minimax pour trouver une stratégie optimale

L'algorithme Minimax

- Utilisé dans les jeux à somme nulle à deux joueurs
- Permet de trouver une stratégie optimale contre un adversaire rationnel
- Basé sur une recherche exhaustive de l'arbre de jeu
- Évalue les positions du jeu à l'aide d'une fonction d'évaluation
- Applique une recherche en profondeur pour explorer toutes les possibilités

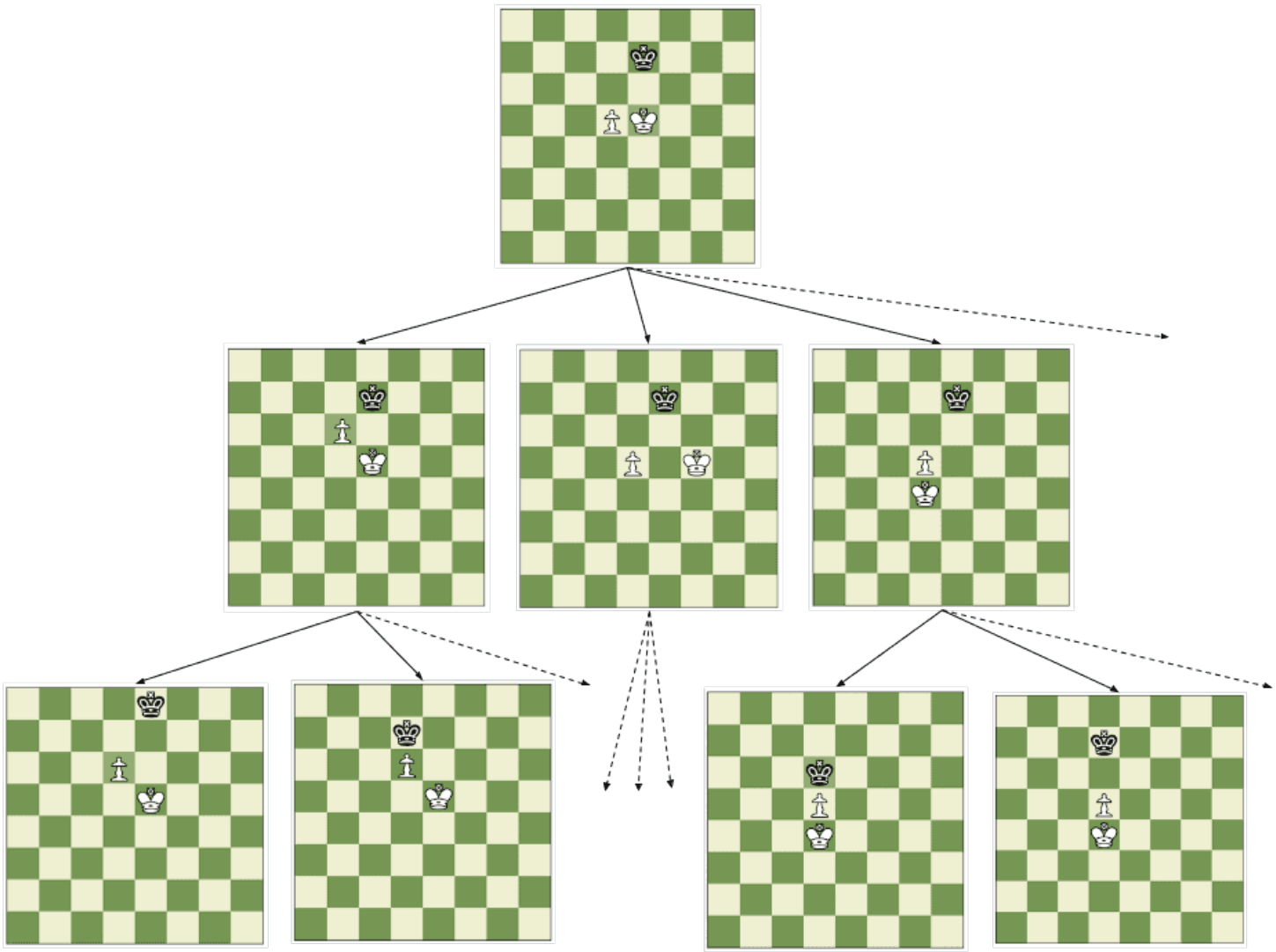
Objectif : Trouver une stratégie optimale pour maximiser les gains ou minimiser les pertes pour une finale d'échecs Roi et Tour contre Roi

Méthode utilisée pour trouver le meilleur coup en commençant par une position aléatoire



- Initialisation
- Construction de l'arbre Minimax
- Évaluation des positions
- Procédure Minimax
- Mise à jour et itération

Arbre de jeu :



Fonction d'évaluation

But de la fonction d'évaluation :

- Évaluer la qualité d'une position dans le jeu
- Estimer les chances de victoire du joueur à partir de cette position

Fonctionnement de la fonction d'évaluation :

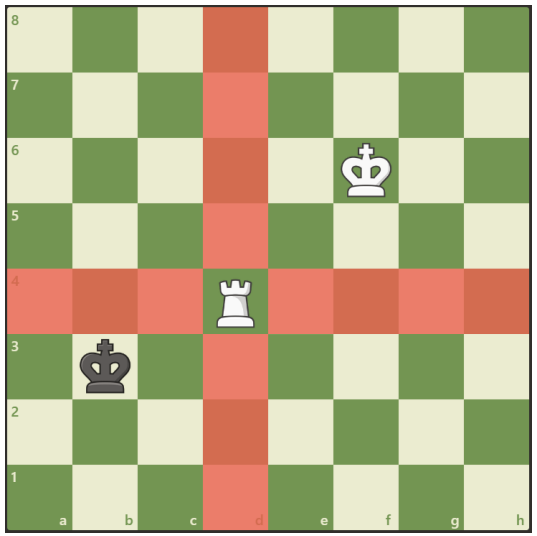
1. Analyse des caractéristiques du jeu (pions, positions, avantages tactiques, etc.)
2. Attribution d'une valeur numérique à chaque caractéristique
3. Agrégation de ces valeurs pour obtenir une valeur globale de la position

Calcul de la fonction d'évaluation

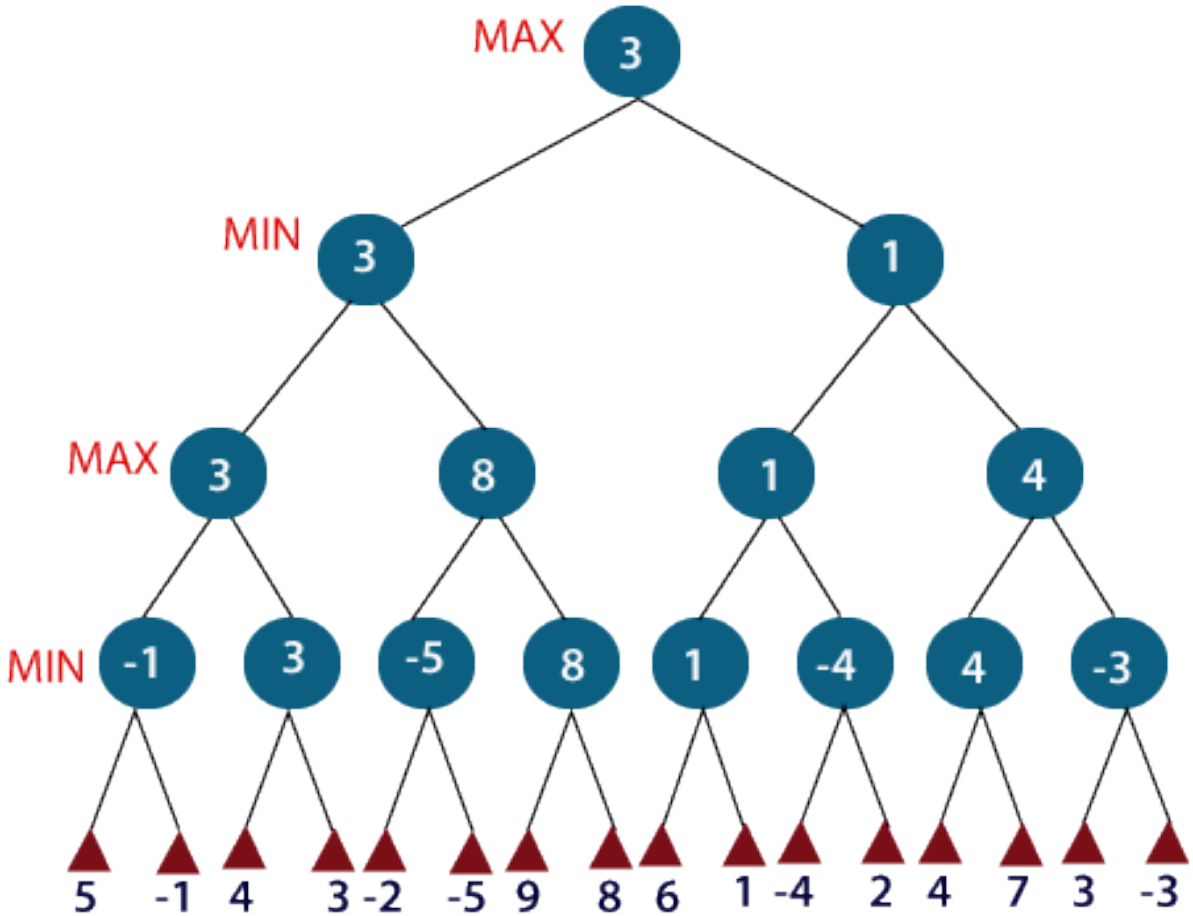
Cette fonction combine deux composants principaux :

- **freeSquares**
- **endgameValues**

$$Valeurtotale = freeSquares + endgameValues$$



Application de l'algorithme Minimax (Exemple)



Principe de l'algorithme :

L'algorithme Minimax est utilisé pour trouver la meilleure stratégie dans le jeu d'échecs. Voici les étapes principales de l'algorithme

- Création de l'arbre de jeu : génération de tous les coups possibles à partir de la position actuelle du jeu.

Input:

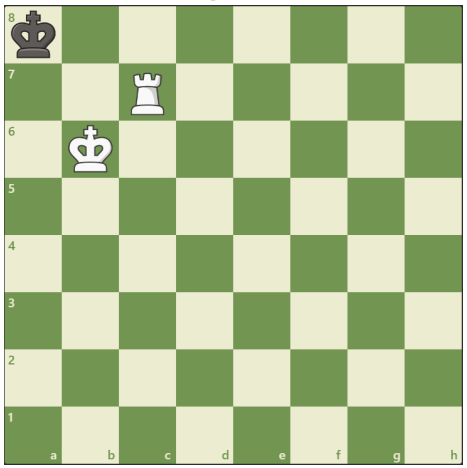


Figure: Echiquier initial du jeu

Output:

```
>>> (executing file "chess minimax work.py")
Meilleure évaluation: -500
Meilleur coup: [0, 2, 17]
```

Figure: Meilleur Coup et son évaluation

Test et résultats : Jeu d'échecs (Annexe)


```python
def entiersToCoordonnees(pos):
    return divmod(pos, 8)

def coordonneesToEntiers(ligne, colonne):
    return ligne * 8 + colonne

def is_attacked(ligne, colonne, WR_pos, WK_pos):
    WR_ligne, WR_colonne = entiersToCoordonnees(WR_pos)
    WK_ligne, WK_colonne = entiersToCoordonnees(WK_pos)

    if ligne == WR_ligne or colonne == WR_colonne:
        return True
    if abs(ligne - WK_ligne) <= 1 and abs(colonne - WK_colonne) <= 1:
        return True
    return False
```

Test et résultats : Jeu d'échecs (Annexe)


```
def generer_coups(pos_actuelle, joueur):
    BK_ligne, BK_colonne = entiersToCoordonnees(pos_actuelle[0])
    WR_ligne, WR_colonne = entiersToCoordonnees(pos_actuelle[1])
    WK_ligne, WK_colonne = entiersToCoordonnees(pos_actuelle[2])

    # Vérification des conditions d'échec et mat
    if BK_ligne == BK_colonne == WR_ligne == 0 and WR_colonne > 1 and
WK_colonne == 1 and WK_ligne == 2:
        return []

    if BK_ligne == BK_colonne == WR_ligne == 7 and WR_colonne < 6 and
WK_colonne == 6 and WK_ligne == 5:
        return []

    if BK_ligne == WR_ligne == 7 and BK_colonne == 0 and WR_colonne >
1 and WK_colonne == 1 and WK_ligne == 5:
        return []

    if BK_colonne == 7 and BK_ligne == WR_ligne == 0 and WR_colonne <
6 and WK_ligne == 2 and WK_colonne == 6:
        return []

    if BK_ligne == WR_ligne == 0 and WK_colonne == BK_colonne and
WR_colonne != BK_colonne + 1 and WR_colonne != BK_colonne - 1 and
WK_ligne == 2:
        return []

    if BK_ligne == WR_ligne == 7 and WK_colonne == BK_colonne and
WR_colonne != BK_colonne + 1 and WR_colonne != BK_colonne - 1 and
WK_ligne == 5:
        return []
```

Test et résultats : Jeu d'échecs (Annexe)


```
    if BK_colonne == WR_colonne == 7 and WK_ligne == BK_ligne and
WR_ligne != BK_ligne + 1 and WR_ligne != BK_ligne - 1 and WK_colonne
== 5:
        return []

    if BK_colonne == WR_colonne == 0 and WK_ligne == BK_ligne and
WR_ligne != BK_ligne + 1 and WR_ligne != BK_ligne - 1 and WK_colonne
== 2:
        return []

    coups_possibles = []

    if joueur == 'B':
        # Générer les coups pour le roi noir
        for dr in [-1, 0, 1]:
            for dc in [-1, 0, 1]:
                if dr != 0 or dc != 0:
                    new_ligne, new_colonne = BK_ligne + dr, BK_colonne
+ dc
                    if 0 <= new_ligne < 8 and 0 <= new_colonne < 8:
                        new_pos = coordonneesToEntiers(new_ligne,
new_colonne)
                        if new_pos not in pos_actuelle[1:] and not
is_attacked(new_ligne, new_colonne, pos_actuelle[1], pos_actuelle[2]):
                            coups_possibles.append([new_pos,
pos_actuelle[1], pos_actuelle[2]])
```

Test et résultats : Jeu d'échecs (Annexe)


```
    else:
        # Générer les coups pour la tour blanche
        for r in range(8):
            if r != WR_ligne:
                new_pos = coordonneesToEntiers(r, WR_colonne)
                if new_pos not in pos_actuelle:
                    coups_possibles.append([pos_actuelle[0], new_pos,
pos_actuelle[2]])
        for c in range(8):
            if c != WR_colonne:
                new_pos = coordonneesToEntiers(WR_ligne, c)
                if new_pos not in pos_actuelle:
                    coups_possibles.append([pos_actuelle[0], new_pos,
pos_actuelle[2]])

        # Générer les coups pour le roi blanc
        for dr in [-1, 0, 1]:
            for dc in [-1, 0, 1]:
                if dr != 0 or dc != 0:
                    new_ligne, new_colonne = WK_ligne + dr, WK_colonne
+ dc
                    if 0 <= new_ligne < 8 and 0 <= new_colonne < 8:
                        new_pos = coordonneesToEntiers(new_ligne,
new_colonne)
                        if new_pos not in pos_actuelle and not
((BK_ligne == new_ligne and (BK_colonne == new_colonne + 1 or
BK_colonne == new_colonne - 1)) or
                                                                (BK_c
olonne == new_colonne and (BK_ligne == new_ligne + 1 or BK_ligne ==
new_ligne - 1)) or
                                                                (BK_c
olonne == new_colonne + 1 and BK_ligne == new_ligne + 1) or
```

Test et résultats : Jeu d'échecs (Annexe)


```
olonne == new_colonne + 1 and BK_ligne == new_ligne + 1) or
                                                          (BK_c
olonne == new_colonne + 1 and BK_ligne == new_ligne - 1) or
                                                          (BK_c
olonne == new_colonne - 1 and BK_ligne == new_ligne + 1) or
                                                          (BK_c
olonne == new_colonne - 1 and BK_ligne == new_ligne - 1)):
                        coups_possibles.append([pos_actuelle[0],
pos_actuelle[1], new_pos])

    return coups_possibles
```

Test et résultats : Jeu d'échecs (Annexe)


```python
def generer_arbre(arbre_depart, niveau):
    if niveau >= 0:
        coup, joueur, fils = arbre_depart
        L = generer_coups(coup, joueur)
        for e in L:
            if joueur == 'N':
                A = [e, 'B', []]
                fils.append(A)
                generer_arbre(A, niveau - 1)
            else:
                A = [e, 'N', []]
                fils.append(A)
                generer_arbre(A, niveau - 1)
    return arbre_depart
```

Test et Résultats : Jeu d'échecs (Annexe)

- Fonction d'évaluation : définition d'une fonction qui attribue des valeurs aux positions terminales en fonction du résultat du jeu.

```
def creerFeuilles(Arbre, L):
    if Arbre[2] == []:
        L.append(Arbre[0])
    else:
        for fils in Arbre[2]:
            creerFeuilles(fils, L)
    return L

def evaluate_position(BK_pos, WR_pos, WK_pos):
    free_squares = count_free_squares(BK_pos, WR_pos, WK_pos)
    endgame_value = calculate_endgame_value(BK_pos, WR_pos, WK_pos)
    return free_squares + endgame_value

def count_free_squares(BK_pos, WR_pos, WK_pos):
    free_squares = 0
    BK_ligne, BK_colonne = entiersToCoordonnees(BK_pos)

    for dr in [-1, 0, 1]:
        for dc in [-1, 0, 1]:
            if dr == 0 and dc == 0:
                continue
            new_ligne, new_colonne = BK_ligne + dr, BK_colonne + dc
            if 0 <= new_ligne < 8 and 0 <= new_colonne < 8:
                new_pos = coordonneesToEntiers(new_ligne, new_colonne)
                if not is_attacked(new_ligne, new_colonne, WR_pos,
WK_pos):
                    free_squares += 1

    return free_squares
```

Test et résultats : Jeu d'échecs (Annexe)


```
def calculate_endgame_value(BK_pos, WR_pos, WK_pos):

    BK_ligne, BK_colonne = entiersToCoordonnees(BK_pos)
    WR_ligne, WR_colonne = entiersToCoordonnees(WR_pos)
    WK_ligne, WK_colonne = entiersToCoordonnees(WK_pos)
    # Check for stalemate
    if count_free_squares(BK_pos, WR_pos, WK_pos) == 0 and not
is_attacked(BK_ligne, BK_colonne,WR_pos,WK_pos):
        return 500



    if BK_ligne == BK_colonne == WR_ligne == 0 and WR_colonne > 1 and
WK_colonne == 1 and WK_ligne == 2:
        return -500

    elif BK_ligne == BK_colonne == WR_ligne == 7 and WR_colonne < 6
and WK_colonne == 6 and WK_ligne == 5:
        return -500

    elif BK_ligne == WR_ligne == 7 and BK_colonne == 0 and WR_colonne
> 1 and WK_colonne == 1 and WK_ligne == 5:
        return -500

    elif BK_colonne == 7 and BK_ligne == WR_ligne == 0 and WR_colonne
< 6 and WK_ligne == 2 and WK_colonne == 6:
        return -500

    elif BK_ligne == WR_ligne == 0 and WK_colonne == BK_colonne and
WR_colonne != BK_colonne+1 and WR_colonne != BK_colonne-1 and WK_ligne
== 2:
```

Test et résultats : Jeu d'échecs (Annexe)


```
        return -500

    elif BK_ligne == WR_ligne == 7 and WK_colonne == BK_colonne and
WR_colonne != BK_colonne+1 and WR_colonne != BK_colonne-1 and WK_ligne
== 5 :
        return -500

    elif BK_colonne == WR_colonne == 7 and WK_ligne == BK_ligne and
WR_ligne != BK_ligne+1 and WR_ligne != BK_ligne-1 and WK_colonne == 5:
        return -500

    elif BK_colonne == WR_colonne == 0 and WK_ligne == BK_ligne and
WR_ligne != BK_ligne+1 and WR_ligne != BK_ligne-1 and WK_colonne == 2:
        return -500

    #checks

    elif WR_ligne == BK_ligne == WK_ligne and (BK_colonne < WR_colonne
< WK_colonne or WK_colonne < WR_colonne < BK_colonne or
WK_colonne<BK_colonne < WR_colonne or
WR_colonne<BK_colonne<WK_colonne) :
        return -250

    elif WR_colonne == BK_colonne == WK_colonne and (BK_ligne <
WR_ligne < WK_ligne or WK_ligne < WR_ligne < BK_ligne or WK_ligne <
BK_ligne < WR_ligne or WR_ligne<BK_ligne<WK_ligne):
        return -250

    elif WR_ligne == BK_ligne and WK_ligne != WR_ligne :
        return -250
```

Test et résultats : Jeu de morpion (Annexe)

- Algorithme Minimax : application récursive de l'algorithme Minimax pour évaluer toutes les positions non terminales de l'arbre.

```
        elif WR_colonne == BK_colonne and WK_colonne != WR_colonne :
            return -250

        # No special endgame condition met
        return 0

    def minimax(Arbre, depth, is_maximizing):
        if depth == 4 or Arbre[2] == []:
            return evaluate_position(Arbre[0][0], Arbre[0][1], Arbre[0]
[2]), Arbre[0]

        if not is_maximizing:
            max_eval = float('-inf')
            best_move = None
            for fils in Arbre[2]:
                eval, _ = minimax(fils, depth + 1, True)
                if eval > max_eval:
                    max_eval = eval
                    best_move = fils[0]
            return max_eval, best_move
        else:
            min_eval = float('inf')
            best_move = None
            for fils in Arbre[2]:
                eval, _ = minimax(fils, depth + 1, False)
                if eval < min_eval:
                    min_eval = eval
                    best_move = fils[0]
            return min_eval, best_move
```