[ RAPPORT TPS ]

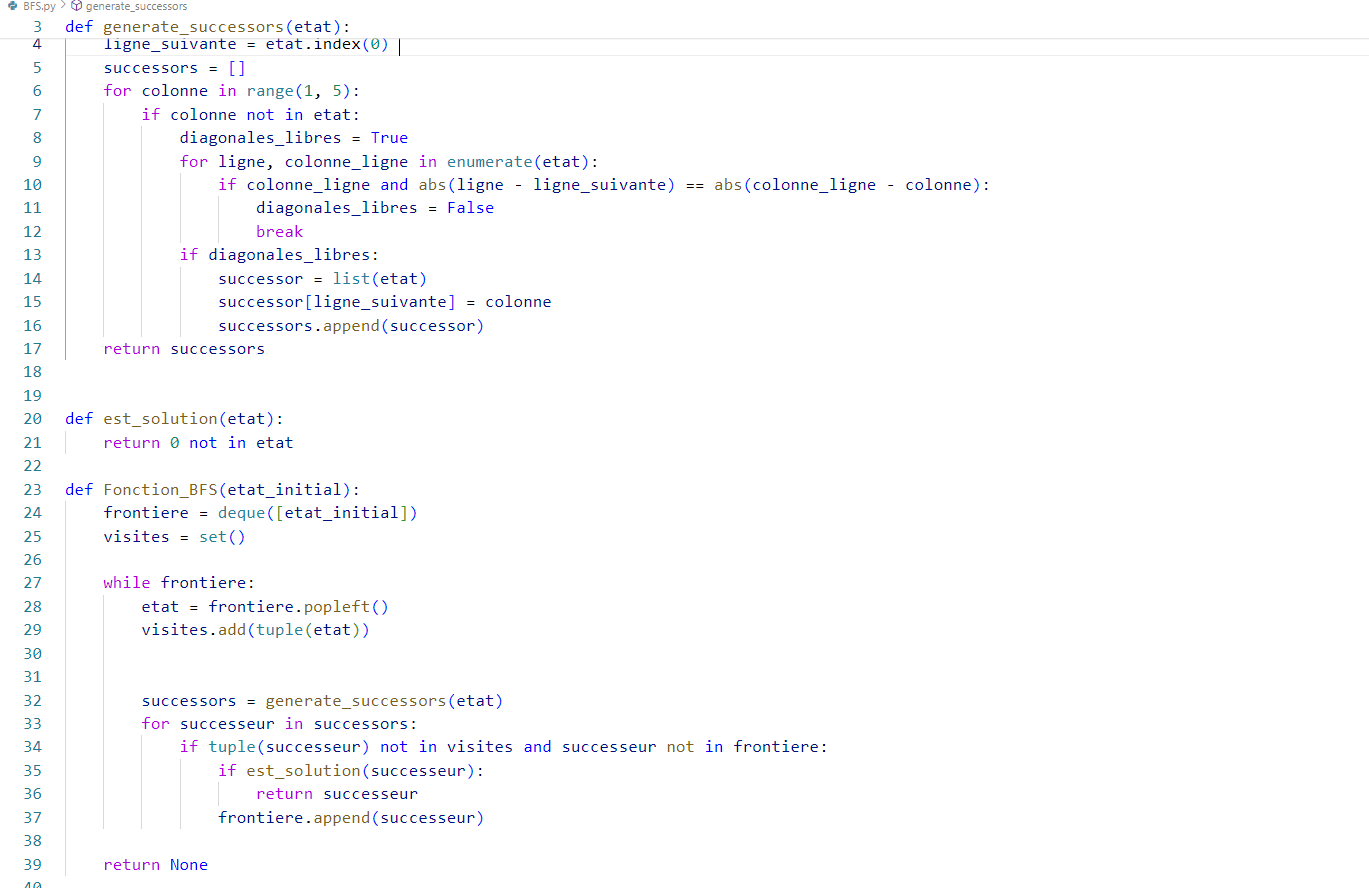
REALISE PAR : WIAM AMHAOUECH

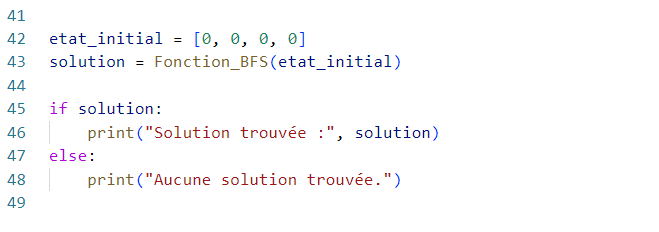
GINF2

# TP1

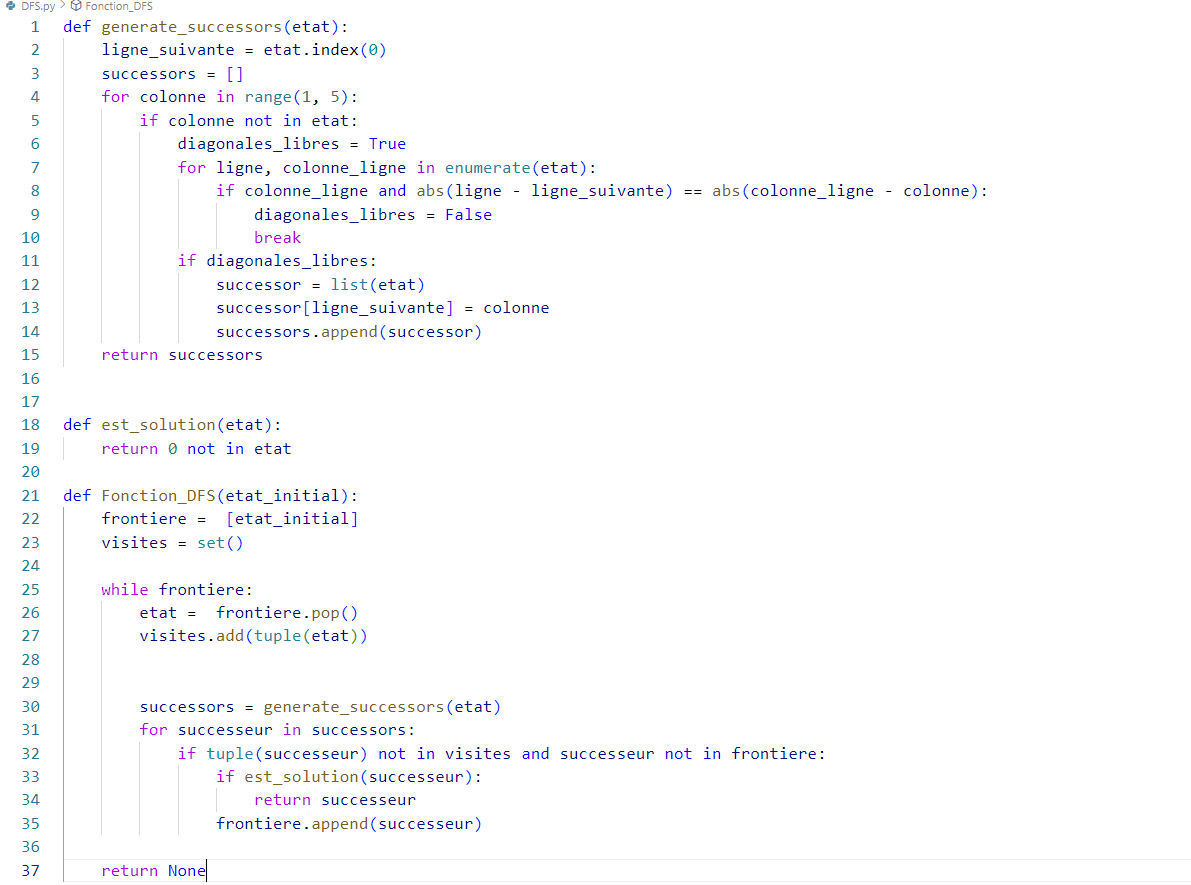
# IMPLEMENTAION DE PROBLEME DES QUATRES REINES EN BFS ET DFS

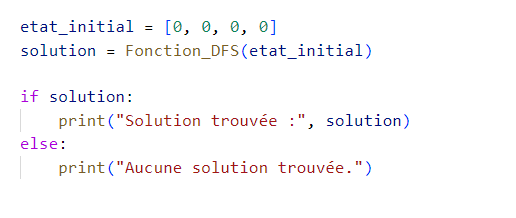
## 1.Recheche en Largueur d’abord





## 2.Recherche en Profondeur d’abord





## 3 . Explication

Le code fourni implémente deux algorithmes de recherche, à savoir la recherche en largeur d'abord (BFS) et la recherche en profondeur d'abord (DFS), pour résoudre un problème de placement de 4 Reines sur un plateau. L'objectif principal est de trouver une configuration où aucune pièce ne peut se menacer mutuellement.

Fonction generate\_successors

La fonction generate\_successors(etat) est cruciale dans la génération des états successeurs à partir d'un état donné. Elle parcourt les colonnes possibles pour la prochaine Reines, vérifie si elles sont libres, et contrôle également l'absence de menaces diagonales entre les Reines . Les successeurs générés sont des états potentiels qui sont ajoutés à une liste pour être explorés ultérieurement par les algorithmes de recherche. Cette fonction est essentielle car elle garantit que seules les configurations valides sont considérées lors de la recherche de solutions.

Fonction est\_solution

La fonction est\_solution(etat) vérifie si l'état actuel représente une solution au problème, c'est-à-dire si aucune pièce ne menace une autre. Cela est déterminé en vérifiant si la pièce vide (représentée par 0) est absente de l'état, ce qui signifie que toutes les pièces ont été correctement placées sans se menacer mutuellement. Cette fonction est utilisée pour déterminer si un état donné est la solution recherchée, ce qui est crucial pour interrompre la recherche une fois que la solution est trouvée.

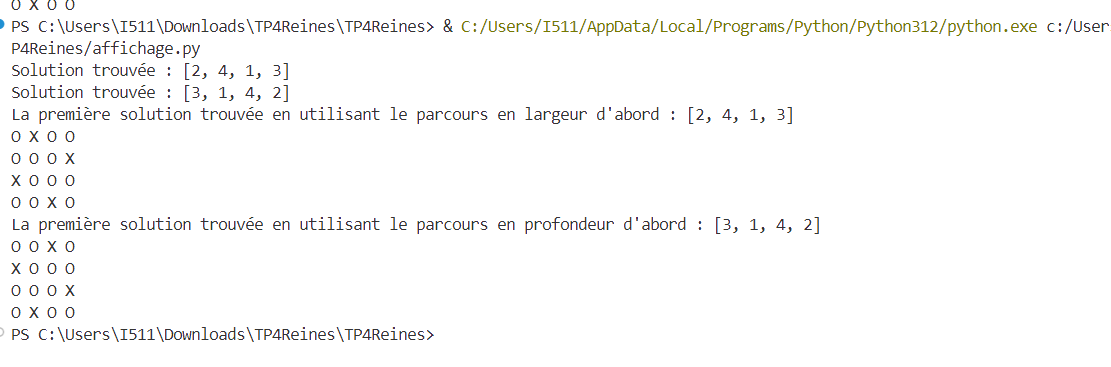
Fonction BFS

L'algorithme BFS (parcours en largeur d'abord) est implémenté dans la fonction Fonction\_BFS(etat\_initial). Il utilise une file (deque dans ce cas) pour explorer les états de manière systématique, en priorisant les états les plus récemment découverts. L'algorithme commence par l'état initial et explore ses successeurs un par un. S'il trouve une solution, il la retourne. Sinon, il continue à explorer les états jusqu'à ce que tous les états possibles soient explorés. L'algorithme BFS garantit de trouver la solution la plus courte en termes de nombre de mouvements.

Fonction DFS

L'algorithme DFS (parcours en profondeur d'abord) est implémenté dans la fonction Fonction\_DFS(etat\_initial). Contrairement à BFS, il utilise une approche plus profonde en explorant autant que possible le chemin actuel avant de revenir en arrière. Il explore en profondeur jusqu'à ce qu'il atteigne une feuille (une solution ou un état sans successeurs). Cette approche peut potentiellement explorer beaucoup plus d'états que BFS, mais elle peut également trouver une solution plus rapidement dans certains cas, notamment si la solution se trouve dans une branche plus profonde de l'arbre d'états.

## 4.Resultat



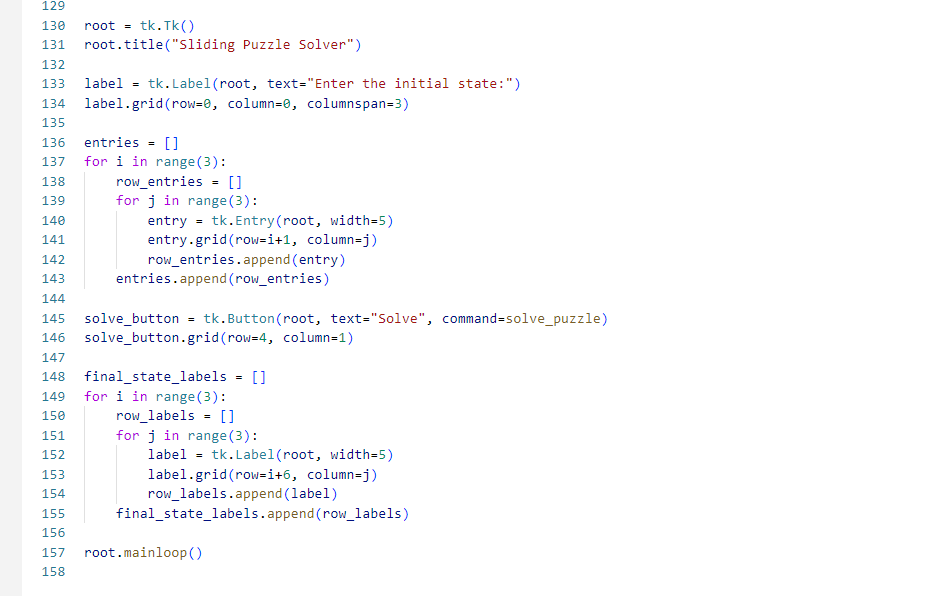
# TP2

# IMPLEMENTAION DE PROBLEME DE TAQUIN EN Algorithme A \*

## 1.Algorithme







## 2.Explication

Le code fourni implémente un solveur pour le jeu de taquin en utilisant l'algorithme A\* avec une heuristique basée sur la distance de Manhattan. Dans cet algorithme, chaque état possible du jeu est représenté par un objet PuzzleNode, qui contient des informations sur l'état du plateau, son parent (l'état précédent), le mouvement effectué pour arriver à cet état, et sa profondeur dans l'arbre de recherche.

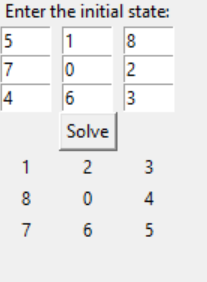
L'algorithme A\* est implémenté dans la fonction a\_star, qui utilise une file de priorité (ici, la bibliothèque heapq) pour explorer les états possibles du jeu de manière efficace. À chaque étape, l'algorithme extrait le nœud de la file de priorité qui a la plus faible valeur de coût combiné (g(n) + h(n)), où g(n) est le coût pour atteindre ce nœud depuis l'état initial et h(n) est l'estimation du coût pour atteindre l'état objectif à partir de ce nœud. Dans ce cas, la distance de Manhattan est utilisée comme heuristique, calculée par la méthode heuristic() de la classe PuzzleNode.

L'algorithme continue d'explorer les états possibles jusqu'à ce qu'il trouve l'état objectif ou qu'il n'y ait plus d'états à explorer. Si l'état objectif est trouvé, l'algorithme reconstruit le chemin de la solution en remontant les parents à partir de l'état objectif, et retourne ce chemin.

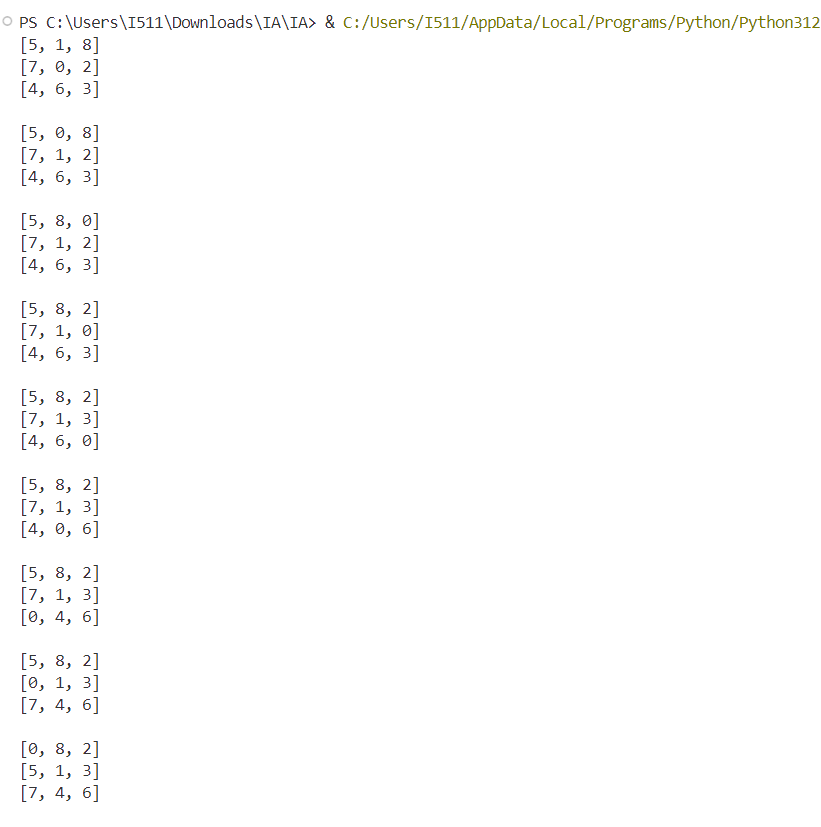
La fonction solve\_puzzle est responsable de récupérer l'état initial du jeu de taquin, de définir l'état objectif, de créer un objet PuzzleNode pour l'état initial, puis d'appeler la fonction a\_star pour résoudre le jeu. Si une solution est trouvée, elle est retournée pour être affichée ou utilisée selon les besoins.

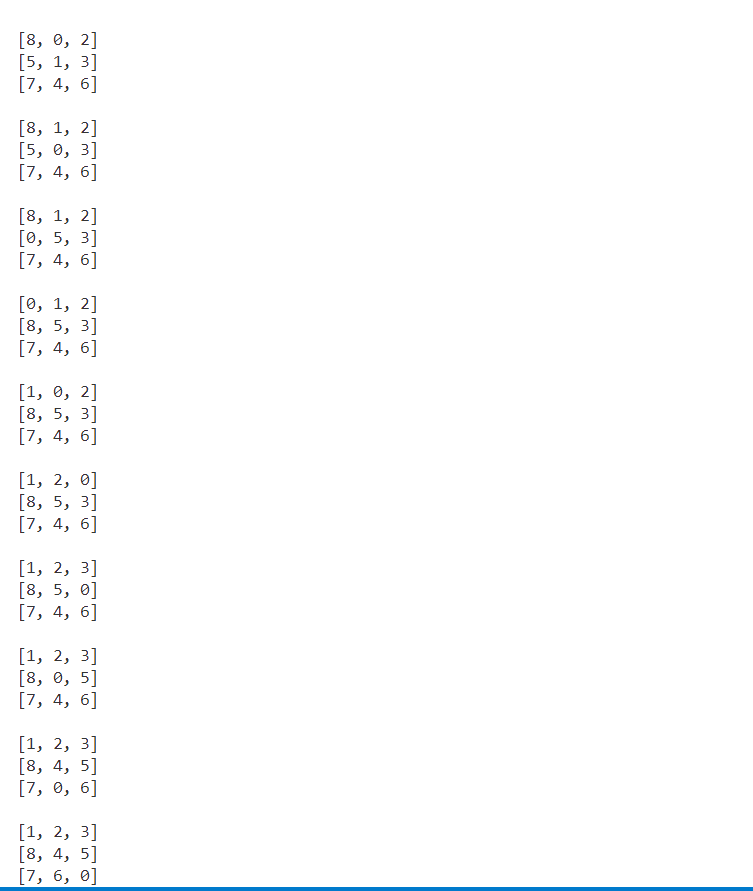
L'affichage de l'interface graphique dans ce code est réalisé à l'aide du module **tkinter** de Python. L'interface graphique comprend des entrées permettant à l'utilisateur d'entrer l'état initial du jeu de taquin, ainsi qu'un bouton "Solve" pour déclencher la résolution du jeu. Une fois que la solution est trouvée, elle est affichée dans la console pour l'utilisateur, ainsi que dans l'interface graphique en mettant à jour les étiquettes correspondantes avec l'état final du jeu de taquin. Bien que la partie principale de l'algorithme réside dans la logique de résolution du jeu, l'interface graphique fournit un moyen convivial pour l'utilisateur d'interagir avec le programme et de visualiser la solution trouvée.

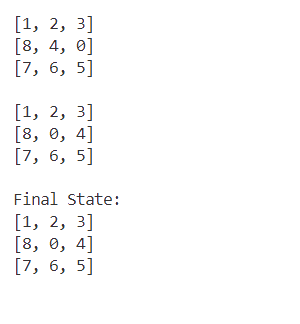
## 3.Resultat



Le chemin est :



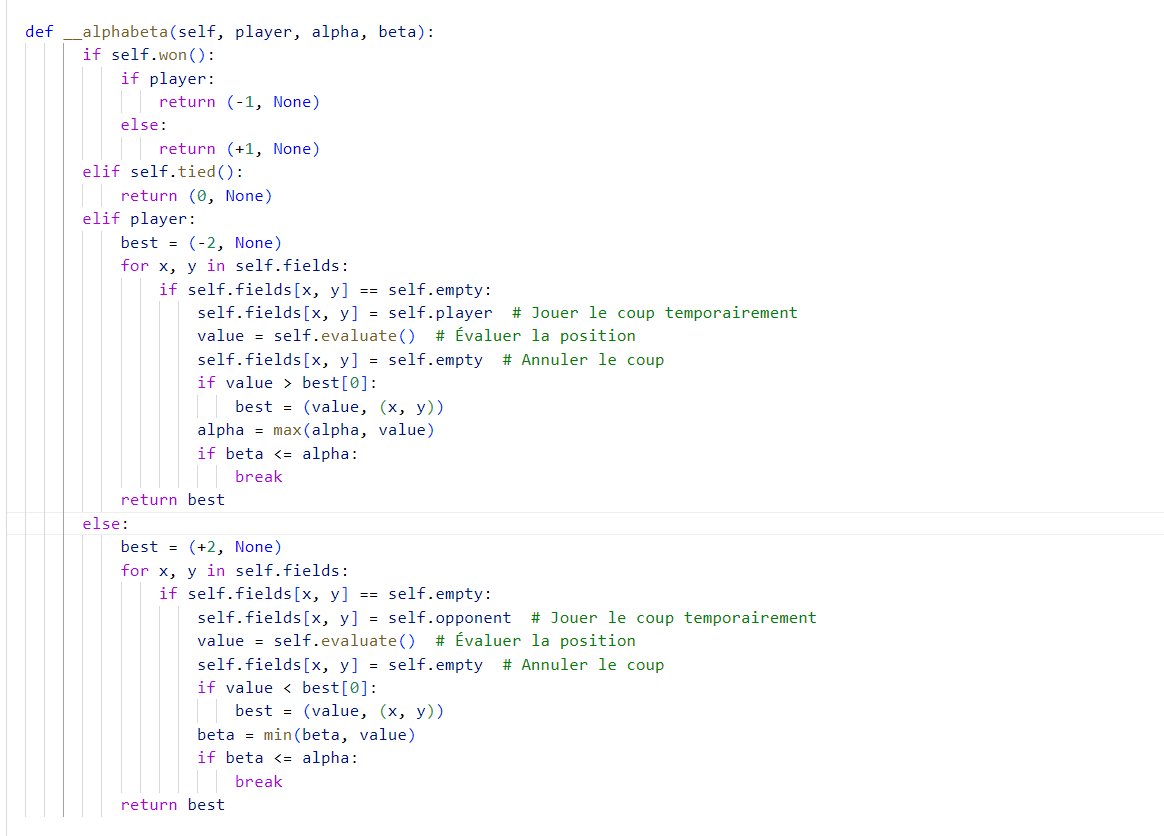


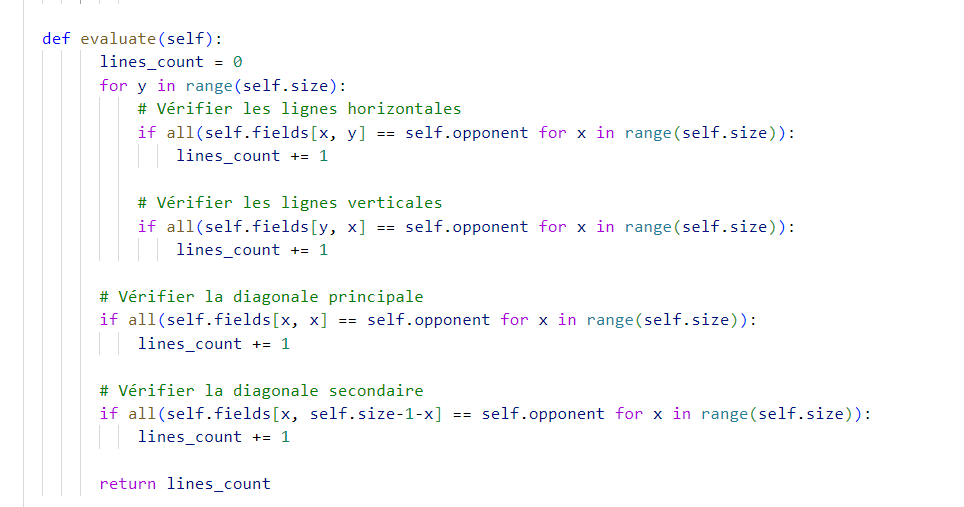


# TP3

# IMPLEMENTAION DE JEU X-O EN Algorithme ALPHA-BETA

## Implementaion de L’algorithme





## 2.Explication

Le code fourni implémente un jeu de Tic Tac Toe (X-O) avec une fonction d'évaluation utilisant l'algorithme Minimax avec élagage alpha-bêta. L'algorithme Minimax est une méthode d'exploration d'arbres de décision utilisée dans les jeux à somme nulle à deux joueurs, comme le Tic Tac Toe. L'objectif est de choisir le meilleur coup possible pour l'ordinateur en minimisant les chances de victoire du joueur et en maximisant ses propres chances de gagner.

Dans ce jeu, chaque état possible du tableau de Tic Tac Toe est représenté par un nœud dans l'arbre de décision. L'algorithme explore récursivement cet arbre jusqu'à une certaine profondeur, en évaluant chaque état du tableau à l'aide d'une fonction d'évaluation. Cette fonction d'évaluation, appelée evaluate(), attribue une valeur à chaque état du jeu en fonction de sa propension à mener à la victoire pour l'ordinateur.

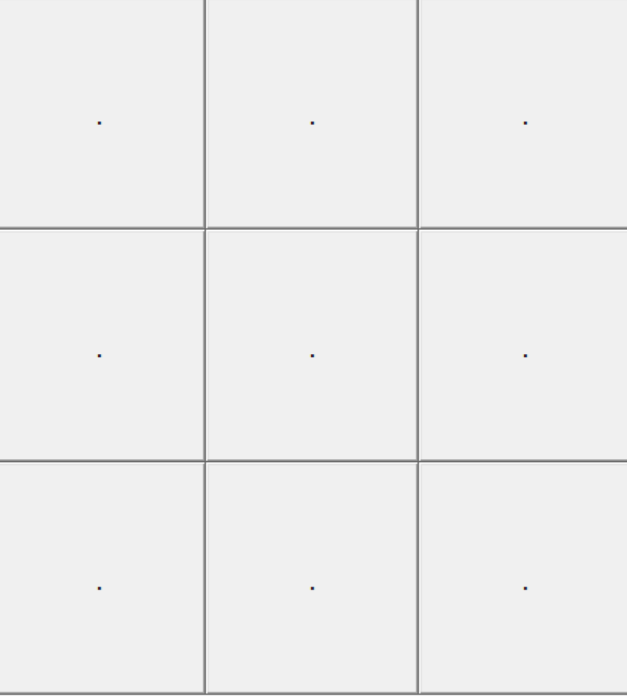
L'algorithme utilise également l'élagage alpha-bêta pour réduire le nombre de nœuds à explorer, ce qui permet d'optimiser les performances de l'algorithme. L'élagage alpha-bêta élimine certaines branches de l'arbre de décision qui ne sont pas susceptibles de produire de meilleurs résultats que les branches déjà explorées.

En combinant Minimax avec élagage alpha-bêta et une fonction d'évaluation efficace, l'ordinateur est capable de jouer de manière stratégique et compétitive contre le joueur humain dans le jeu de Tic Tac Toe. L'algorithme analyse les différentes possibilités de jeu et choisit le coup optimal qui maximise les chances de victoire de l'ordinateur tout en minimisant les chances de victoire du joueur.

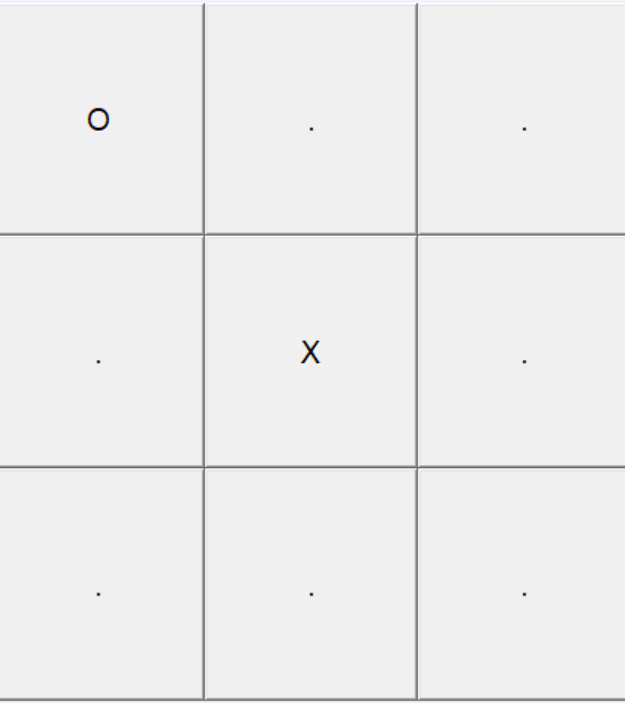
## 3.Resultat

Exemple 1 :

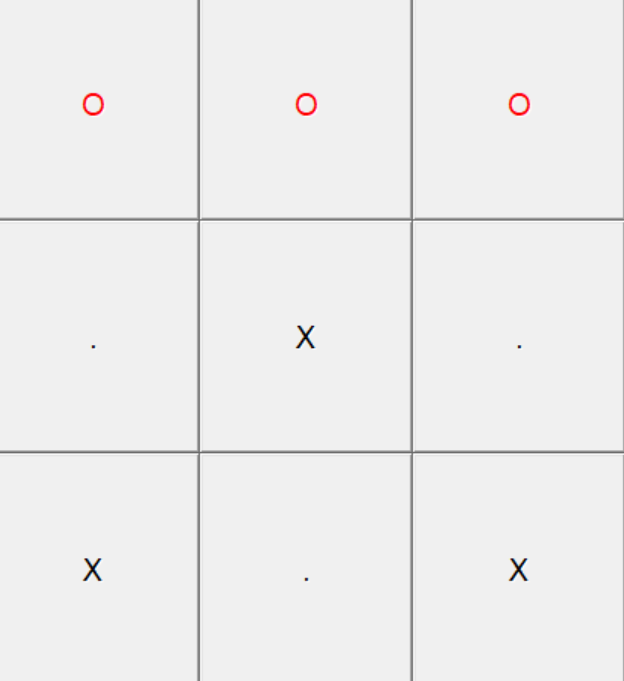
Etat0 :



Etat1 :

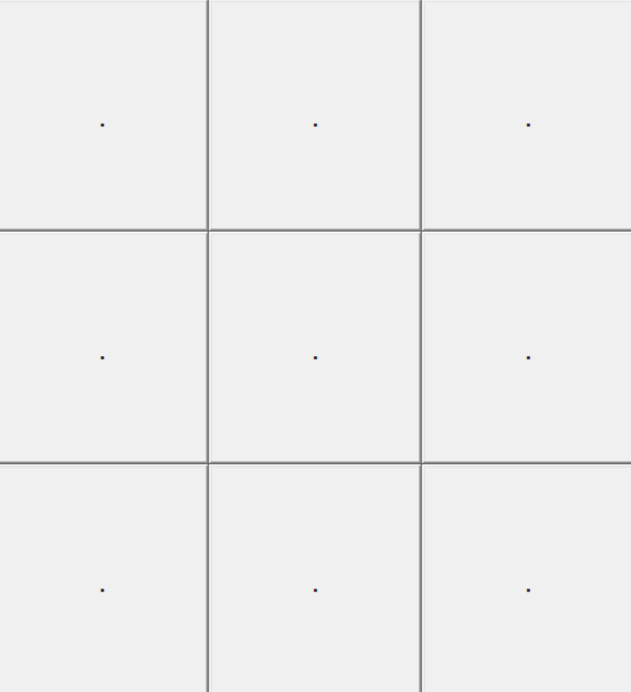


Etat2 : Etat3 :(l’ordinateur gagne)

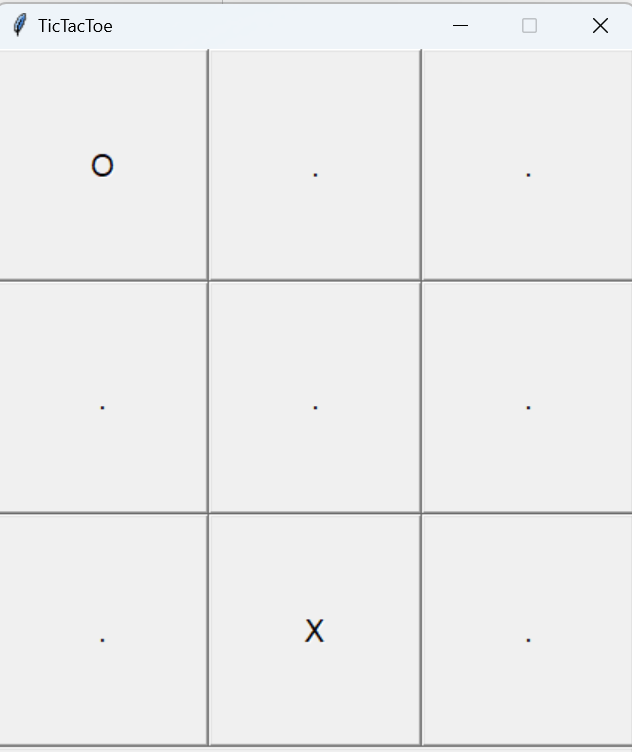
 

Exemple 2:

Etat1:



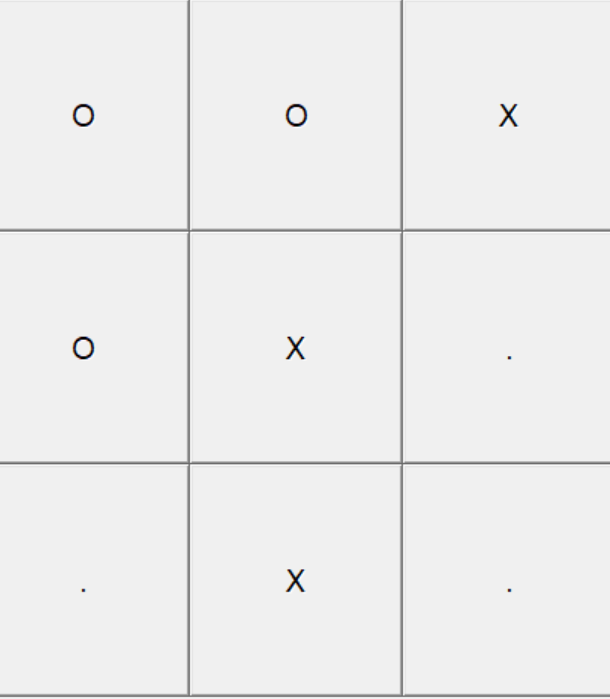
Etat1:



Etat2:



Etat3:



Etat4:(moi qui gagne)

