2024

Jeu Puissance 4 en Python

Nabil GUETTAF

La plateforme

02/02/2024

TABLE DES MATIÈRES

JEU PUISSANCE 4

1 PRÉSENTATION

1.1 HISTORIQUE

Puissance 4 (appelé aussi parfois **4 en ligne**) est un jeu de stratégie combinatoire abstrait, commercialisé pour la première fois en 1974 par la Milton Bradley Company, plus connue sous le nom de MB et détenue depuis 1984 par la société Hasbro.

1.2 RÈGLES DU JEU

Le but du jeu est d'aligner une suite de 4 pions de même couleur sur une grille comptant 6 rangées et 7 colonnes. Chaque joueur dispose de 21 pions d'une couleur (par convention, en général jaune ou rouge). Tour à tour, les deux joueurs placent un pion dans la colonne de leur choix, le pion coulisse alors jusqu'à la position la plus basse possible dans ladite colonne à la suite de quoi c'est à l'adversaire de jouer. Le vainqueur est le joueur qui réalise le premier un alignement (horizontal, vertical ou diagonal) consécutif d'au moins quatre pions de sa couleur. Si, alors que toutes les cases de la grille de jeu sont remplies, aucun des deux joueurs n'a réalisé un tel alignement, la partie est déclarée nulle.

2 PRÉSENTATION DES TECHNOLOGIES UTILISÉES

2.1 PYTHON

Python est un langage de programmation interprété, multiparadigme et multiplateformes. Il favorise la programmation impérative structurée, fonctionnelle et orientée objet. Il est doté d'un typage dynamique fort, d'une gestion automatique de la mémoire par ramasse-miettes et d'un système de gestion d'exceptions.

> *Notes :*
>
> *Programmation interprété : il effectue l'analyse et la traduction nécessaires à l'exécution d'un programme donné non pas une fois pour toutes, mais à chaque exécution de ce programme. L'exécution nécessite ainsi de disposer non seulement du programme, mais aussi de l'interprète correspondant.*
>
> *Multiparadigme : Le paradigme de programmation est la façon (parmi d'autres) d'approcher la programmation informatique et de formuler les solutions aux problèmes et leur formalisation dans un langage de programmation approprié.*
>
> *La programmation impérative : est un paradigme de programmation qui décrit les opérations en séquences d'instructions exécutées par l'ordinateur pour modifier l'état du programme.*

La programmation fonctionnelle : est un paradigme de programmation de type déclaratif qui considère le calcul en tant qu'évaluation de fonctions mathématiques.

La programmation orientée objet (POO) : ou programmation par objet, est un paradigme de programmation informatique. La programmation par objet consiste à utiliser des techniques de programmation pour mettre en œuvre une conception basée sur les objets. Celle-ci peut être élaborée en utilisant des méthodologies de développement logiciel objet, dont la plus connue est le processus unifié (« Unified Software Development Process » en anglais), et exprimée à l'aide de langages de modélisation tels que le Unified Modeling Language (UML).

Un ramasse-miettes : ou récupérateur de mémoire1, ou glaneur de cellules (en anglais garbage collector, abrégé en GC), est un sous-système informatique de gestion automatique de la mémoire. Il est responsable du recyclage de la mémoire préalablement allouée puis inutilisée.

Dans le contexte des langages de programmation fonctionnels et impératifs, un système de gestion d'exceptions ou SGE permet de gérer les conditions exceptionnelles pendant l'exécution du programme. Lorsqu'une exception se produit, l'exécution normale du programme est interrompue et l'exception est traitée.

Système de gestion d'exceptions : Les erreurs/exceptions les plus courantes sont probablement l'accès non autorisé à une zone mémoire (erreur de manipulation de pointeur) et la division par zéro (on ne prévoit pas le cas où le diviseur est nul).

2.2 TKINTER

De nos jours, la plupart des applications reposent sur des environnements graphiques tels que Windows ou MacOS. Tkinter est une des solutions envisageables pour coder une telle interface graphique depuis Python. Elle a l'avantage d'être intégrée à Python et donc immédiatement accessible.

Lorsque vous utilisez un outil tel que Jupyter, Google Colab ou Noteable pour entrer des commandes Python et tester votre code, vous exécutez par défaut des programmes en mode console, soit au sein d'un terminal.

Or, l'immense majorité des programmes s'exécutent au sein d'une interface graphique, que ce soit dans Windows, MacOS, ou Gnome (Linux). Donc, vous pourrez avoir à cœur de développer des programmes adaptés à de tels environnements, affichant des fenêtres, des menus déroulant, des boutons interactifs.

Toutefois, le module TKinter a un atout : il est intégré en standard à Python et il n'est donc pas nécessaire d'installer un quelconque package pour le faire fonctionner. Il suffit de l'importer.

Ainsi, la commande :

from tkinter import *

Un avantage majeur de Tkinter est qu'il est multi-plateforme. Le même code exact fonctionne avec Windows, MacOS et Linux. En d'autres termes, la commande de création d'une fenêtre va créer selon le cas, une fenêtre en mode Windows, en mode MacOS, etc. Ainsi, les applications construites avec Tkinter auront le "look" de la plate-forme sur laquelle on les exécute à un moment donné.

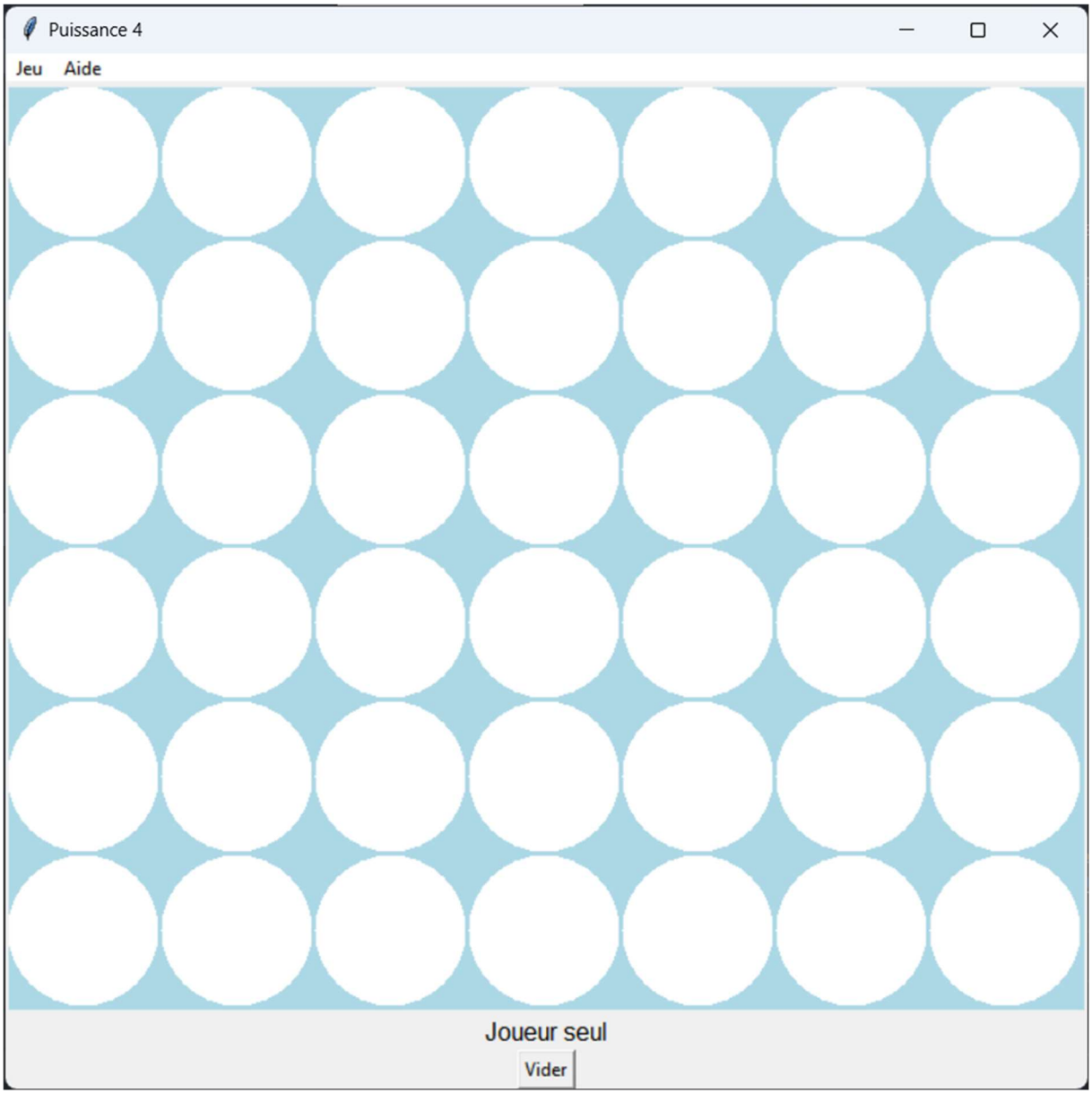
Voici quelques utilisations courantes de Tkinter :

1. Développement d'applications de bureau : Tkinter est souvent utilisé pour créer des applications de bureau avec une interface utilisateur graphique. Cela inclut des applications telles que des éditeurs de texte, des gestionnaires de fichiers, des calculatrices, etc.
2. Outils d'administration et scripts avec interfaces graphiques (GUI, Graphical User Interfaces) : Les développeurs utilisent Tkinter pour créer des outils d'administration avec une interface graphique, ou pour transformer des scripts en applications avec une interface utilisateur conviviale.
3. Applications éducatives : Tkinter est souvent utilisé dans des contextes éducatifs pour enseigner la programmation avec Python en montrant aux étudiants comment créer des interfaces graphiques interactives.
4. Prototypage rapide : Tkinter est adapté au prototypage rapide d'interfaces utilisateur, ce qui permet aux développeurs de créer rapidement des maquettes de leurs applications.
5. Outils de configuration : Vous pouvez utiliser Tkinter pour créer des outils de configuration graphique pour des logiciels ou des systèmes plus complexes.

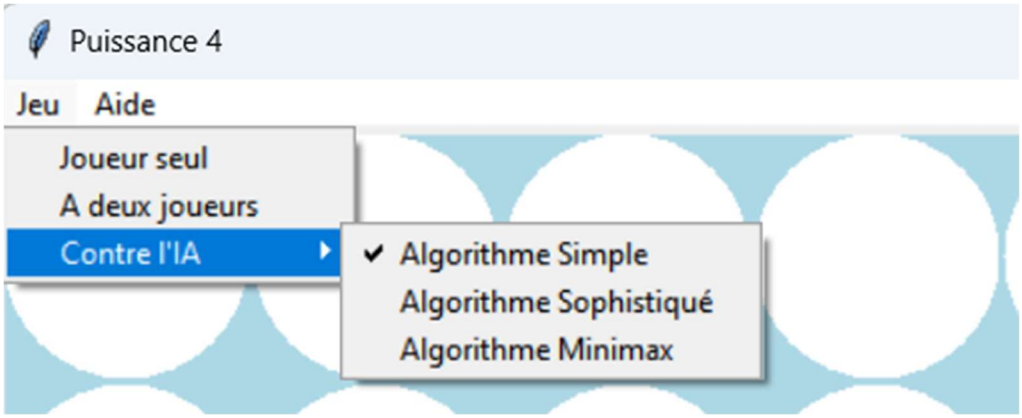
3 CONCEPTION DU JEU

La conception du jeu puissance 4 est réalisé à partir du langage python, de sa bibliothèque graphique tkinter, et de l'environnement de programmation Visual Studio Code.

3.1 DESCRIPTION DE L'INTERFACE UTILISATEUR

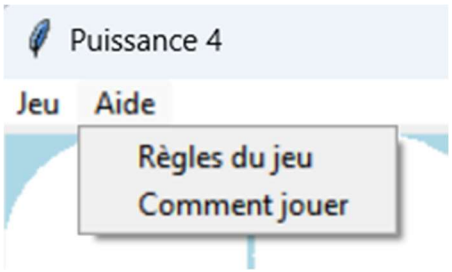


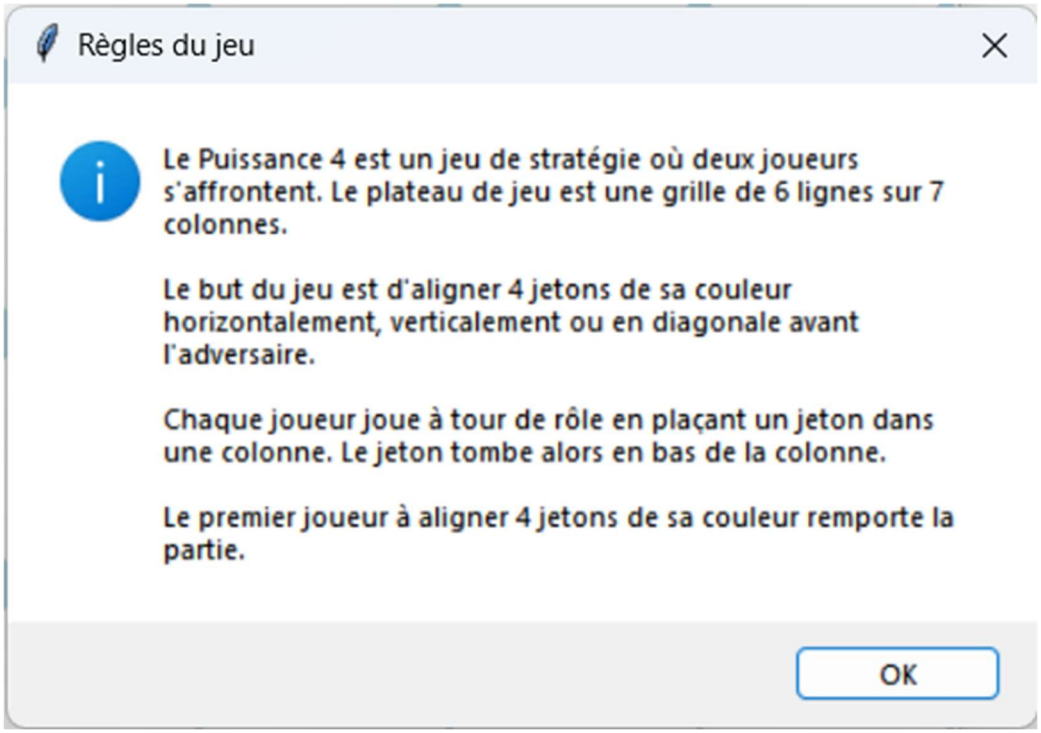
L'interface utilisateur permet de jouer seul, à deux, contre l'ordinateur (appelé IA). Le choix se fait via le menu Jeu.

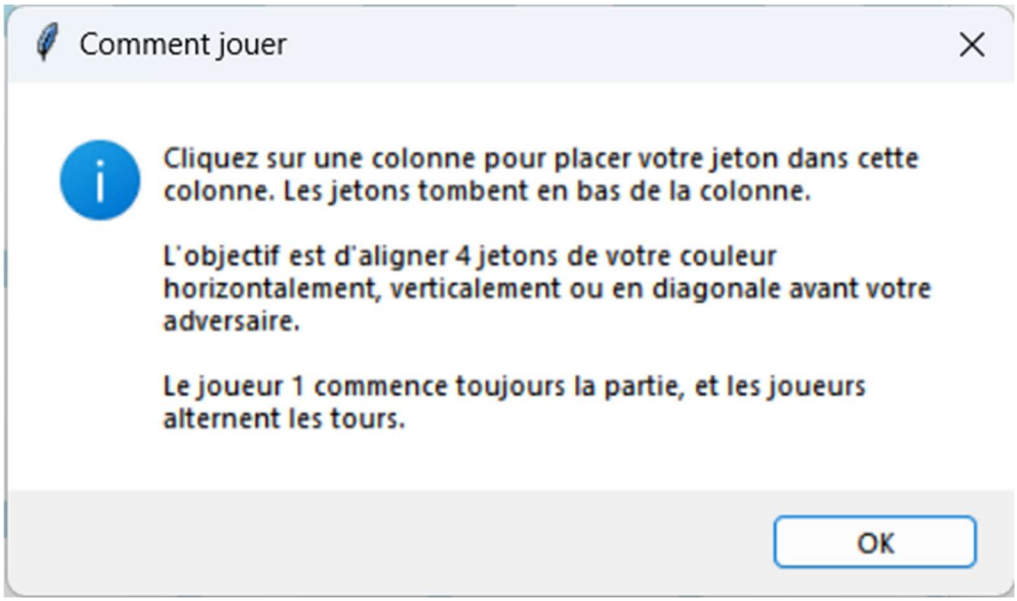


Il y a trois choix de difficulté pour jouer contre l'IA, l'algorithme Minimax est le plus performant.

Un menu d'aide existe pour présenter les règles du jeu et les instructions pour jouer.

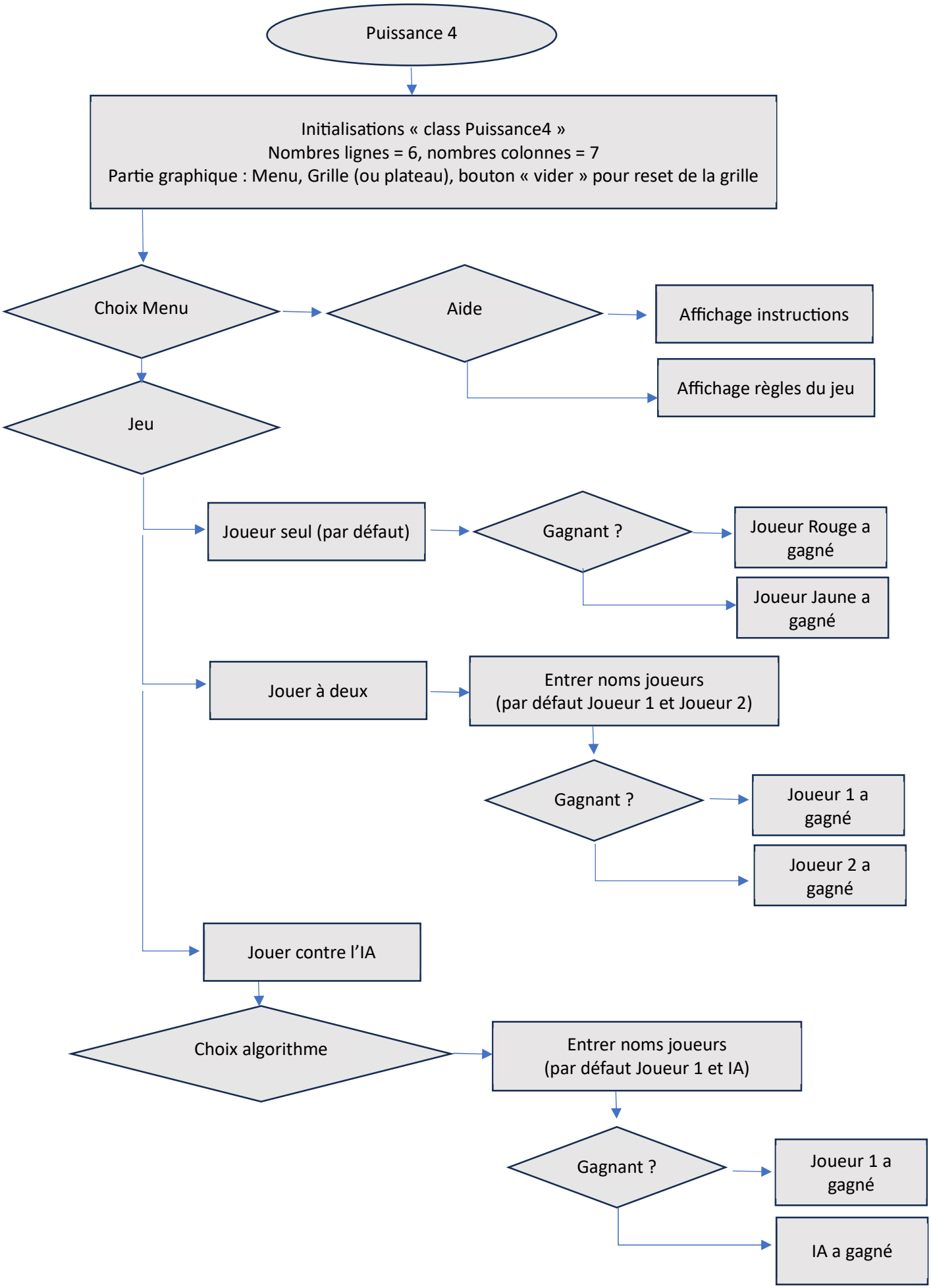






3.2 DÉVELOPPEMENT

L’organigramme suivant montre le processus suivi dans le programme en python et tkinter du jeu puissance 4 :


Puissance 4
Initialisations « class Puissance4 »
Nombres lignes = 6, nombres colonnes = 7
Partie graphique : Menu, Grille (ou plateau), bouton « vider » pour reset de la grille
Choix Menu
Aide
Affichage instructions
Affichage règles du jeu
Jeu
Joueur seul (par défaut)
Gagnant ?
Joueur Rouge a gagné
Joueur Jaune a gagné
Jouer à deux
Entrer noms joueurs
(par défaut Joueur 1 et Joueur 2)
Gagnant ?
Joueur 1 a gagné
Joueur 2 a gagné
Jouer contre l'IA
Choix algorithme
Entrer noms joueurs
(par défaut Joueur 1 et IA)
Gagnant ?
Joueur 1 a gagné
IA a gagné

Algorithme du Jeu Puissance 4:

- Initialisation du Jeu:
  Création d'une classe Puissance4 qui gère le jeu.
  Paramètres du jeu (nombre de lignes, de colonnes, joueurs, grille, etc.).
  Initialisation de l'interface graphique Tkinter.

- Menu du Jeu:
  Mise en place d'un menu avec des options pour jouer à un joueur seul, à deux joueurs, ou contre l'IA.
  Possibilité de choisir entre différents algorithmes pour l'IA (Algorithme Simple, Algorithme Sophistiqué, Algorithme Minimax).

- Plateau (grille) de Jeu:
  Création d'une grille graphique avec Canvas.
  Possibilité de réinitialiser le jeu avec un bouton "Vider".

- Jeu contre l'IA:
  L'IA peut être configurée pour utiliser l'Algorithme Simple, l'Algorithme Sophistiqué, ou l'Algorithme Minimax.
  Lorsque c'est le tour de l'IA, elle choisit une colonne en fonction de l'algorithme sélectionné.

- Déroulement du Jeu (jouer un coup):
  Chaque joueur (ou l'IA) place un jeton dans une colonne en cliquant sur la grille.
  Le jeu vérifie après chaque coup s'il y a un gagnant.
  Si un joueur gagne, le jeu affiche un message de victoire et offre la possibilité de recommencer.

- Algorithmes de l'IA:
  Algorithme Simple: Choix aléatoire d'une colonne.
  Algorithme Sophistiqué: Évite de perdre en une étape, sinon choix aléatoire.
  Algorithme Minimax: Algorithme de recherche avec évaluation de la position pour la meilleure colonne.

- Affichage des Règles et Instructions:

Possibilité d'afficher les règles du jeu et les instructions sur la manière de jouer.

- Lancement de l'Application:
  Initialisation de la fenêtre principale Tkinter et exécution de la boucle principale.

Cet algorithme permet aux joueurs de jouer entre eux, contre l'IA, et offre différents niveaux de complexité pour l'IA en fonction de l'algorithme sélectionné.

4 TESTS ET DÉBOGGAGE

unittest est un module intégré à Python qui fournit un cadre de test pour l'automatisation des tests unitaires. Les tests unitaires sont utilisés pour vérifier que chaque partie d'un programme fonctionne

correctement. Le module unittest tire son inspiration du cadre de test de Java (JUnit) et du cadre de test de Smalltalk (SUnit).

Voici quelques concepts clés associés à unittest :

1. Test Case (Cas de test) : Un cas de test est la plus petite unité de test dans unittest. Il représente un scénario de test unique. Vous créez une classe qui hérite de unittest.TestCase et y définissez des méthodes de test (méthodes commençant par test_).
2. Test Fixture (Configuration de test) : Le test fixture est l'environnement dans lequel les tests s'exécutent. Cela peut inclure la création de données, l'initialisation d'objets, etc. setUp et tearDown sont des méthodes spéciales qui sont appelées avant et après chaque méthode de test.
3. Assertion (Assertion) : Les assertions sont des déclarations qui vérifient si une condition est vraie. Si la condition est fausse, l'assertion génère une exception, indiquant que quelque chose ne va pas dans le code testé.
4. Test Runner (Lanceur de tests) : Le test runner est responsable de l'exécution des tests. Il peut être utilisé via la ligne de commande (python -m unittest) ou intégré dans des outils de développement.

Pour exécuter les tests, on utilise la commande « python -m unittest nom_du_fichier.py » ou dans notre cas « py -3 -m nom_du_fichier_test.py ».

Les fichiers tests unitaires se trouvent en annexes.

5 CONCLUSION ET PERSPECTIVES

5.1 CONCLUSION

Les difficultés sont de s'approprier des syntaxes python et les syntaxes tkinter qu'ils n'ont pas été vu en formation, bien que l'on peut trouver de l'aide auprès de sites web et chatgpt qui est d'un grand secours.

Comme par exemple :

Dans l'expression `_, best_col`, `_` est une convention utilisée en Python pour indiquer que la valeur n'est pas utilisée dans le corps de la boucle ou de la fonction. En d'autres termes, le nom `_` est généralement utilisé pour des variables "jetables" ou des valeurs qui ne sont pas utilisées.

Dans le contexte de votre test, `_, best_col = puissance4.minimax(...)` est une manière de dire que vous n'utilisez pas la première valeur renvoyée par la méthode `minimax`, mais seulement la deuxième (dans ce cas, `best_col`). Cela peut être utile lorsque la fonction renvoie plusieurs valeurs, mais vous n'avez besoin que de certaines d'entre elles.

Voici un exemple simple pour illustrer cela :

```
# Supposons une fonction qui renvoie deux valeurs
def exemple_fonction():
```

```
    return 10, 20

# Utilisation avec _
_, y = exemple_fonction()

# Dans cet exemple, la valeur 10 est ignorée et la valeur 20 est assignée à
la variable y
```

Cela rend le code plus clair en indiquant explicitement que la première valeur n'est pas utilisée intentionnellement.

5.2 PERSPECTIVES

Voici quelques suggestions :

- Interface utilisateur améliorée :
  il est possible d'envisager d'améliorer l'interface utilisateur en ajoutant des éléments graphiques, des animations ou des effets visuels pour rendre le jeu plus attrayant.
- Options de personnalisation :
  Ajoutez des fonctionnalités permettant aux utilisateurs de personnaliser le jeu, telles que la possibilité de choisir la couleur des jetons, le thème graphique, etc.
- Gestion des scores :
  Implémentez un système de gestion des scores pour suivre les performances des joueurs au fil du temps. Cela pourrait inclure la sauvegarde des scores dans un fichier ou l'utilisation d'une base de données simple.
- Mode de jeu en réseau :
  Explorez la possibilité d'ajouter un mode de jeu en réseau, permettant à deux joueurs de s'affronter en ligne.
- Optimisation de l'IA :
  Pour améliorer l'IA, vous pouvez explorer des algorithmes plus avancés que le simple algorithme Minimax, tels que l'algorithme Alpha-Beta Pruning pour réduire le nombre de nœuds évalués.
- Tests unitaires :
  Ajoutez des tests unitaires pour garantir la robustesse de votre code, en particulier pour les parties logiques telles que l'IA et les vérifications de victoire.
- Documentation
- Gestion des erreurs :
  Améliorez la gestion des erreurs pour rendre le jeu plus robuste face à des situations inattendues.
- Interface multilingue :
  Ajoutez la possibilité de changer la langue de l'interface utilisateur pour rendre le jeu accessible à un public plus large.
- Refactoring :
  Passez en revue votre code et effectuez éventuellement un refactoring pour rendre le code plus lisible, réutilisable et maintenable.

6 ANNEXES

6.1 CODE SOURCE DU FICHIER « PUISSANCE4.PY »

```
# Importation des modules Tkinter nécessaires
import tkinter as tk
from tkinter import messagebox, simpledialog
import random

# Définition de la classe Puissance4 qui gère le jeu
class Puissance4:
    # def __init__(self, root):
    def __init__(self, root=None): # Modification pour unittest
        # Initialisation de l'interface graphique et des paramètres du jeu
        # self.root = root
        self.root = root or tk.Tk()  # Utilisez root s'il est fourni, sinon
créez un nouveau Tkinter Tk()

        self.root.title("Puissance 4")
        self.lignes = 6
        self.colonnes = 7
        self.joueurs = ['', '']
        self.joueur_actuel_indice = 0
        self.grille = [['' for _ in range(self.colonnes)] for _ in
range(self.lignes)]
        self.algorithme_ia = None
        self.profondeur_minimax = 3  # Choisir une profondeur appropriée pour
l'algorithme Minimax

        # Initialisation du menu, du plateau de jeu et du bouton de
réinitialisation
        self.init_menu()
        self.init_grille()
        self.bouton_vider()
        self.demarrer_jeu(False)

    # Ajoutez cette méthode pour permettre l'initialisation avec une
configuration spécifique
    # pour unittest
    @classmethod
    def init_with_config(cls, config):
        root = config.get('root', None)
        instance = cls(root=root)
        # Initialisez d'autres attributs de l'instance si nécessaire
        return instance
```

```
    # Méthode pour initialiser le menu du jeu
    def init_menu(self):
        menu = tk.Menu(self.root)
        self.root.config(menu=menu)

        jeu_menu = tk.Menu(menu, tearoff=0)
        menu.add_cascade(label="Jeu", menu=jeu_menu)

        jeu_menu.add_command(label="Joueur seul", command=lambda:
self.demarrer_jeu(False))
        jeu_menu.add_command(label="A deux joueurs", command=lambda:
self.demarrer_jeu(True))

        self.algo_menu = tk.Menu(jeu_menu, tearoff=0)
        jeu_menu.add_cascade(label="Contre l'IA", menu=self.algo_menu)
        self.init_algorithme_options()

        help_menu = tk.Menu(menu, tearoff=0)
        menu.add_cascade(label="Aide", menu=help_menu)
        help_menu.add_command(label="Règles du jeu",
command=self.affiche_regles)
        help_menu.add_command(label="Comment jouer",
command=self.affiche_insctructions)


    def init_algorithme_options(self):
        algorithmes = ["Algorithme Simple", "Algorithme Sophistiqué",
"Algorithme Minimax"]
        self.selected_algo = tk.StringVar()
        self.selected_algo.set(algorithmes[0])

        for algo in algorithmes:
            self.algo_menu.add_radiobutton(label=algo,
variable=self.selected_algo, value=algo, command=self.update_selected_algo)

    def update_selected_algo(self):
        if self.selected_algo.get() == "Algorithme Simple":
            self.algorithme_ia = self.algorithme_simple
        elif self.selected_algo.get() == "Algorithme Sophistiqué":
            self.algorithme_ia = self.algorithme_sophistique
        elif self.selected_algo.get() == "Algorithme Minimax":
            self.algorithme_ia = self.algorithme_minimax

        self.demarrer_jeu_contre_ia()


    # Méthode pour initialiser le plateau de jeu
    def init_grille(self):
```

```
        self.Canvas = tk.Canvas(self.root, width=self.colonnes * 100,
height=self.lignes * 100, bg='#add8e6')
        self.Canvas.pack()

        self.mode_jeu_label = tk.Label(self.root, text="", font=("Helvetica",
12))
        self.mode_jeu_label.pack()

        self.dessin_jeu()

    # pour unittest
    def afficher_grille(self, grille):
        for row in grille:
            print(" ".join(row))

    # Méthode pour dessiner le plateau de jeu
    def dessin_jeu(self):
        for ligne in range(self.lignes):
            for col in range(self.colonnes):
                x1 = col * 100
                y1 = ligne * 100
                x2 = x1 + 100
                y2 = y1 + 100

                self.Canvas.create_oval(x1, y1, x2, y2, outline='#add8e6',
fill='white', width=2)

        self.Canvas.bind("<Button-1>", self.detection_clic)

    # Méthode pour initialiser le bouton de réinitialisation
    def bouton_vider(self):
        reset_button = tk.Button(self.root, text="Vider",
command=self.vider_init)
        reset_button.pack()

    def vider_init(self):
        self.jeu_reset()
        self.demarrer_jeu(False)

    # Méthode appelée lorsqu'un clic est effectué sur le plateau de jeu
    def detection_clic(self, event):
        col = event.x // 100
        self.placer_jeton(col)

    # Méthode pour placer un jeton dans la colonne spécifiée
    def placer_jeton(self, col):
        for ligne in range(self.lignes - 1, -1, -1):
            if self.grille[ligne][col] == '':
                self.grille[ligne][col] = self.joueur_actuel()
```

```
                self.jeton(ligne, col)
                if self.verif_gagnant(ligne, col):
                    self.affiche_gagnant()
                    self.jeu_reset()
                    self.demarrer_jeu(False)
                else:
                    self.joueur_suivant()
                    if self.joueur_actuel() == 'IA':
                        self.joueur_ia()
                break

    # Méthode pour dessiner un jeton sur le plateau de jeu
    def jeton(self, ligne, col):
        x1 = col * 100 + 10
        y1 = ligne * 100 + 10
        x2 = x1 + 80
        y2 = y1 + 80

        color = 'red' if self.joueur_actuel() == self.joueurs[0] else 'yellow'
        self.Canvas.create_oval(x1, y1, x2, y2, outline=color, fill=color)

    # Méthode pour permettre à l'IA de jouer
    def joueur_ia(self):
        available_colonnes = [col for col in range(self.colonnes) if
self.grille[0][col] == '']
        if available_colonnes:
            col = self.algorithme_ia(available_colonnes)
            self.placer_jeton(col)

    # Méthode pour vérifier s'il y a un gagnant après chaque coup
    def verif_gagnant(self, ligne, col):
        # Vérification horizontale
        if self.verif_ligne(ligne, col, 0, 1) or self.verif_ligne(ligne, col,
0, -1):
            return True

        # Vérification verticale
        if self.verif_ligne(ligne, col, 1, 0):
            return True

        # Vérification diagonale (/)
        if self.verif_ligne(ligne, col, -1, 1) or self.verif_ligne(ligne, col,
1, -1):
            return True

        # Vérification diagonale (\)
        if self.verif_ligne(ligne, col, -1, -1) or self.verif_ligne(ligne,
col, 1, 1):
            return True
```

```
        return False

    # Méthode pour vérifier une ligne dans une direction spécifique
    def verif_ligne(self, ligne, col, ligne_modif, col_modif):
        count = 1
        player = self.joueur_actuel()

        # Vérification dans une direction
        for i in range(1, 4):
            new_ligne = ligne + i * ligne_modif
            new_col = col + i * col_modif

            if 0 <= new_ligne < self.lignes and 0 <= new_col < self.colonnes
and self.grille[new_ligne][new_col] == player:
                count += 1
            else:
                break

        # Vérification dans l'autre direction
        for i in range(1, 4):
            new_ligne = ligne - i * ligne_modif
            new_col = col - i * col_modif

            if 0 <= new_ligne < self.lignes and 0 <= new_col < self.colonnes
and self.grille[new_ligne][new_col] == player:
                count += 1
            else:
                break

        return count >= 4

    # Méthode pour afficher le gagnant de la partie
    def affiche_gagnant(self):
        nom_gagnant = self.joueurs[0] if self.joueur_actuel_indice == 0 else
self.joueurs[1]
        messagebox.showinfo("Gagnant", f"{nom_gagnant} a gagné !")

    # Méthode pour réinitialiser le jeu
    def jeu_reset(self):
        self.joueur_actuel_indice = 0
        self.joueurs = ['', '']
        self.grille = [['' for _ in range(self.colonnes)] for _ in
range(self.lignes)]
        self.Canvas.delete("all")
        self.dessin_jeu()

    # Méthode pour passer au joueur suivant
    def joueur_suivant(self):
```

```
        self.joueur_actuel_indice = 1 - self.joueur_actuel_indice

    # Méthode pour obtenir le nom du joueur actuel
    def joueur_actuel(self):
        return self.joueurs[self.joueur_actuel_indice]

    # Méthode pour commencer une nouvelle partie avec un ou deux joueurs
    def demarrer_jeu(self, deux_joueurs):
        self.jeu_reset()
        if deux_joueurs:
            self.entrer_noms_joueurs()
            mode_jeu = "Jeu à deux joueurs"
        else:
            self.joueurs = ["Joueur Rouge", "Joueur Jaune"]
            mode_jeu = "Joueur seul"

        self.mode_jeu_label.config(text=mode_jeu)

    # Méthode pour demander les noms des joueurs
    def entrer_noms_joueurs(self):
        for i in range(2):
            # Mettre la boîte de dialogue en premier plan
            # self.root.focus_force()
            # self.root.grab_set()
            nom_joueur = simpledialog.askstring("Nom du Joueur", f"Entrez le
nom du Joueur {i + 1}")

            if nom_joueur:
                self.joueurs[i] = nom_joueur
            else:
                self.joueurs[i] = f"Joueur {i + 1}"

            # Libérer la prise sur la fenêtre
            # self.root.grab_release()


    # Méthode pour commencer une partie contre l'IA
    def demarrer_jeu_contre_ia(self):
        self.jeu_reset()
        mode_jeu = "Jeu contre l'IA"
        self.mode_jeu_label.config(text=mode_jeu)
        self.entrer_noms_joueurs()

        self.joueurs[1] = 'IA'  # Définir le deuxième joueur comme l'IA
        if self.joueur_actuel() == 'IA':
            self.joueur_ia()


    # Méthode pour afficher les règles du jeu
```

```
    def affiche_regles(self):
        regles_text = "Le Puissance 4 est un jeu de stratégie où deux joueurs
s'affrontent. Le plateau de jeu est une grille de 6 lignes sur 7
colonnes.\n\nLe but du jeu est d'aligner 4 jetons de sa couleur
horizontalement, verticalement ou en diagonale avant l'adversaire.\n\nChaque
joueur joue à tour de rôle en plaçant un jeton dans une colonne. Le jeton
tombe alors en bas de la colonne.\n\nLe premier joueur à aligner 4 jetons de
sa couleur remporte la partie."

        messagebox.showinfo("Règles du jeu", regles_text)

    # Méthode pour afficher les instructions sur la façon de jouer
    def affiche_insctructions(self):
        insctructions_text = "Cliquez sur une colonne pour placer votre jeton
dans cette colonne. Les jetons tombent en bas de la colonne.\n\nL'objectif est
d'aligner 4 jetons de votre couleur horizontalement, verticalement ou en
diagonale avant votre adversaire.\n\nLe joueur 1 commence toujours la partie,
et les joueurs alternent les tours."

        messagebox.showinfo("Comment jouer", insctructions_text)


    ######################################################################
    # Algorithme simple :
    # Choix d'une colonne aléatoire
    ######################################################################
    # Debut
    def algorithme_simple(self, available_colonnes):
        return random.choice(available_colonnes)
    # Fin

    ######################################################################
    # Algorithme sophistiqué
    ######################################################################
    # Debut
    def algorithme_sophistique(self, available_colonnes):
        for col in available_colonnes:
            # Tester si en jouant dans cette colonne, le joueur actuel gagnera
            copie_grille = [ligne.copy() for ligne in self.grille]
            self.simuler_coup(col, copie_grille, self.joueur_actuel())
            if self.check_victoire_simulee(copie_grille,
self.joueur_actuel()):
                return col

        for col in available_colonnes:
            # Tester si en jouant dans cette colonne, l'adversaire gagnera
            copie_grille = [ligne.copy() for ligne in self.grille]
            adversaire = self.joueurs[1] if self.joueur_actuel() ==
self.joueurs[0] else self.joueurs[0]
```

```
            self.simuler_coup(col, copie_grille, adversaire)
            if self.check_victoire_simulee(copie_grille, adversaire):
                return col

        # Aucune menace détectée, jouer dans une colonne aléatoire
        return random.choice(available_colonnes)

    def simuler_coup(self, col, copie_grille, joueur):
        for ligne in range(self.lignes - 1, -1, -1):
            if copie_grille[ligne][col] == '':
                copie_grille[ligne][col] = joueur
                break

    def check_victoire_simulee(self, copie_grille, joueur):
        for ligne in range(self.lignes):
            for col in range(self.colonnes):
                if copie_grille[ligne][col] == joueur:
                    if self.check_victoire_simulee_direction(copie_grille,
ligne, col, 0, 1):
                        return True
                    if self.check_victoire_simulee_direction(copie_grille,
ligne, col, 1, 0):
                        return True
                    if self.check_victoire_simulee_direction(copie_grille,
ligne, col, -1, 1):
                        return True
                    if self.check_victoire_simulee_direction(copie_grille,
ligne, col, 1, -1):
                        return True
        return False

    def check_victoire_simulee_direction(self, copie_grille, ligne, col,
ligne_modif, col_modif):
        count = 1
        joueur = copie_grille[ligne][col]

        for i in range(1, 4):
            new_ligne = ligne + i * ligne_modif
            new_col = col + i * col_modif

            if 0 <= new_ligne < self.lignes and 0 <= new_col < self.colonnes
and copie_grille[new_ligne][new_col] == joueur:
                count += 1
            else:
                break

        for i in range(1, 4):
            new_ligne = ligne - i * ligne_modif
            new_col = col - i * col_modif
```

```
            if 0 <= new_ligne < self.lignes and 0 <= new_col < self.colonnes
and copie_grille[new_ligne][new_col] == joueur:
                count += 1
            else:
                break

        return count >= 4
    # Fin

    ######################################################
    # Algorithme Minimax
    ######################################################
    # Debut

    def algorithme_minimax(self, available_colonnes):
        # Appel de la fonction Minimax pour trouver la meilleure colonne
        _, best_col = self.minimax(self.grille, self.profondeur_minimax, True)

        return best_col

    def minimax(self, grille, profondeur, joueur_maximisant):
        score = self.evaluer_grille(grille)

        if profondeur == 0 or score == 100 or score == -100:
            return score, None

        if joueur_maximisant:
            max_eval = float('-inf')
            best_col = None

            for col in range(self.colonnes):
                if grille[0][col] == '':
                    copie_grille = [ligne.copy() for ligne in grille]
                    self.simuler_coup(col, copie_grille, self.joueur_actuel())
                    eval, _ = self.minimax(copie_grille, profondeur - 1,
False)

                    if eval > max_eval:
                        max_eval = eval
                        best_col = col

            return max_eval, best_col

        else:
            min_eval = float('inf')
            best_col = None

            for col in range(self.colonnes):
```

```
                if grille[0][col] == '':
                    copie_grille = [ligne.copy() for ligne in grille]
                    self.simuler_coup(col, copie_grille, self.joueurs[1] if
self.joueur_actuel() == self.joueurs[0] else self.joueurs[0])
                    eval, _ = self.minimax(copie_grille, profondeur - 1, True)

                    if eval < min_eval:
                        min_eval = eval
                        best_col = col

            return min_eval, best_col

    def evaluer_grille(self, grille):
        if self.verif_gagnant_simulee(grille, self.joueur_actuel()):
            return 100
        elif self.verif_gagnant_simulee(grille, self.joueurs[1] if
self.joueur_actuel() == self.joueurs[0] else self.joueurs[0]):
            return -100
        else:
            return 0

    def verif_gagnant_simulee(self, grille, joueur):
        for ligne in range(self.lignes):
            for col in range(self.colonnes):
                if grille[ligne][col] == joueur:
                    if self.check_victoire_simulee_direction(grille, ligne,
col, 0, 1):
                        return True
                    if self.check_victoire_simulee_direction(grille, ligne,
col, 1, 0):
                        return True
                    if self.check_victoire_simulee_direction(grille, ligne,
col, -1, 1):
                        return True
                    if self.check_victoire_simulee_direction(grille, ligne,
col, 1, -1):
                        return True
        return False

    def check_victoire_simulee_direction(self, grille, ligne, col,
ligne_modif, col_modif):
        count = 1
        joueur = grille[ligne][col]

        for i in range(1, 4):
            new_ligne = ligne + i * ligne_modif
            new_col = col + i * col_modif
```

```
            if 0 <= new_ligne < self.lignes and 0 <= new_col < self.colonnes
and grille[new_ligne][new_col] == joueur:
                count += 1
            else:
                break

        for i in range(1, 4):
            new_ligne = ligne - i * ligne_modif
            new_col = col - i * col_modif

            if 0 <= new_ligne < self.lignes and 0 <= new_col < self.colonnes
and grille[new_ligne][new_col] == joueur:
                count += 1
            else:
                break

        return count >= 4

    # Fin


# Lancement de l'application
if __name__ == "__main__":
    root = tk.Tk()
    app = Puissance4(root)
    root.mainloop()
```

6.2 CODE SOURCE DU FICHIER TEST_PUISSANCE4_1.PY (AVEC UNITTEST)

```
# Tests pour l'interface graphiquejeu_menu
# exécuter les tests à l'aide de la commande py -3 -m unittest
test_puissance4_1.py

import unittest
from unittest.mock import patch
from puissance4 import Puissance4
import tkinter as tk

class TkinterTestCase(unittest.TestCase):
    def test_minimax(self):
        root = tk.Tk()  # Créer une instance de Tk
        game = Puissance4(root)
        available_cols = [0, 1, 2, 3, 4, 5, 6]
        col = game.algorithme_minimax(available_cols)
        self.assertIn(col, available_cols)

    def test_algorithme_simple(self):
        root = tk.Tk()  # Créer une instance de Tk
        game = Puissance4(root)
        available_cols = [0, 1, 2, 3, 4, 5, 6]
        col = game.algorithme_simple(available_cols)
        self.assertIn(col, available_cols)

    def test_algorithme_sophistique(self):
        root = tk.Tk()  # Créer une instance de Tk
        game = Puissance4(root)
        available_cols = [0, 1, 2, 3, 4, 5, 6]
        col = game.algorithme_sophistique(available_cols)
        self.assertIn(col, available_cols)

    def test_entrer_noms_joueurs(self):
        # Créer une instance factice de tk.Tk
        root = tk.Tk()

        # Créer une instance de Puissance4 avec la racine factice
        game = Puissance4(root)

        # Définir la sortie attendue de la boîte de dialogue
        with patch("tkinter.simpledialog.askstring", return_value="Nom du
Joueur 1"):
            game.entrer_noms_joueurs()
            self.assertEqual(game.joueurs[0], "Nom du Joueur 1")
```

```
        with patch("tkinter.simpledialog.askstring", return_value="Nom du
Joueur 2"):
            game.entrer_noms_joueurs()
            self.assertEqual(game.joueurs[1], "Nom du Joueur 2")

if __name__ == '__main__':
    unittest.main()
```

6.3 Code source du fichier test_puissance4_2.py (avec unittest, spécifique algorithme minimax)

```
# Tests pour l'algorithme Minimax
# exécuter les tests à l'aide de la commande py -3 -m unittest
test_puissance4_2.py

import unittest
from puissance4 import Puissance4

class TestPuissance4(unittest.TestCase):
    def setUp(self):
        # Initialisation pour les tests
        self.puissance4 = Puissance4.init_with_config({'root': None})

    def test_minimax(self):
        # Test avec une grille vide, le résultat doit être une colonne
aléatoire
        grille_vide = [['' for _ in range(self.puissance4.colonnes)] for _ in
range(self.puissance4.lignes)]
        _, best_col_vide = self.puissance4.minimax(grille_vide,
self.puissance4.profondeur_minimax, True)
        self.assertIn(best_col_vide, range(self.puissance4.colonnes))

        # Test avec une grille gagnante pour le joueur actuel
        grille_gagnante = [
            ['X', 'O', 'X', 'O', '', '', ''],
            ['O', 'X', 'X', '', '', '', ''],
            ['', '', '', '', '', '', ''],
            ['', '', '', '', '', '', ''],
            ['', '', '', '', '', '', ''],
            ['', '', '', '', '', '', '']
        ]
        _, best_col_gagnante = self.puissance4.minimax(grille_gagnante,
self.puissance4.profondeur_minimax, True)
        self.assertEqual(best_col_gagnante, 4)  # Colonne gagnante

        # Test avec une grille où l'adversaire a déjà gagné, l'algorithme ne
doit pas choisir la colonne gagnante
        grille_adversaire_gagnant = [
            ['O', 'X', 'O', 'X', '', '', ''],
            ['X', 'O', 'O', '', '', '', ''],
            ['', '', '', '', '', '', ''],
            ['', '', '', '', '', '', ''],
            ['', '', '', '', '', '', ''],
            ['', '', '', '', '', '', '']
        ]
```

```
        _, best_col_adversaire_gagnant =
self.puissance4.minimax(grille_adversaire_gagnant,
self.puissance4.profondeur_minimax, True)
        self.assertNotEqual(best_col_adversaire_gagnant, 3)  # Colonne
gagnante

if __name__ == '__main__':
    unittest.main()
```