Rapport du projet d'application

Réalisé par :

- **Christ Chadrak MVOUNGOU**
- **Radia MERABTENE**

Formation : **Spécialité Informatique S5**

Groupe : **TDA & TP1**

Matières évaluées :

- **Génie Logiciel**
- **Algorithmique**
- **Interface Homme-Machine**

Année universitaire 2024-2025

TABLE DES MATIERES

TABLE DES FIGURES

INTRODUCTION

Ce projet de développement d'application s'inscrit dans le cadre de l'Unité d'Enseignement Génie Logiciel et Algorithmes de notre formation (S5).

L'objectif principal est de développer un jeu de Tic Tac Toe en implémentant l'algorithme Minimax, tout en appliquant les connaissances acquises dans les cours de Bases du Génie Logiciel, Algorithmes et Structures de Données et Interfaces Hommes-Machines.

Ce projet vise à concevoir une application permettant à un joueur humain d'affronter un ordinateur (une AI) utilisant l'algorithme Minimax pour prendre des décisions optimales.

Ce rapport présente les différentes étapes du projet, depuis l'analyse du cahier des charges et la définition des spécifications, jusqu'à l'architecture logicielle, le développement des modules et leur implémentation.

I. PARTIE GENIE LOGICIELLE

1. Réalisation de la délimitation système

L'objectif de cette délimitation est de définir les frontières entre le système (notre application jeu) et son environnement. L'application Tic Tac Toe est un système interactif conçu pour un utilisateur unique qui interagit comme adversaire du joueur AI (qui utilise le MinMax).

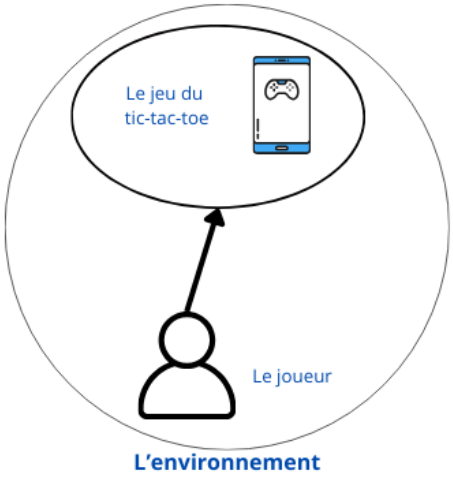

Figure 1 : Délimiration environnement-système

L'application est isolée dans le sens où elle n'interagit pas avec d'autres composants externes.

2. Modélisation UML :

Pour la modélisation nous avons fait le choix de faire deux diagrammes le diagramme des cas d'utilisation et le diagramme de séquences.

a. Diagramme des cas d'utilisation :

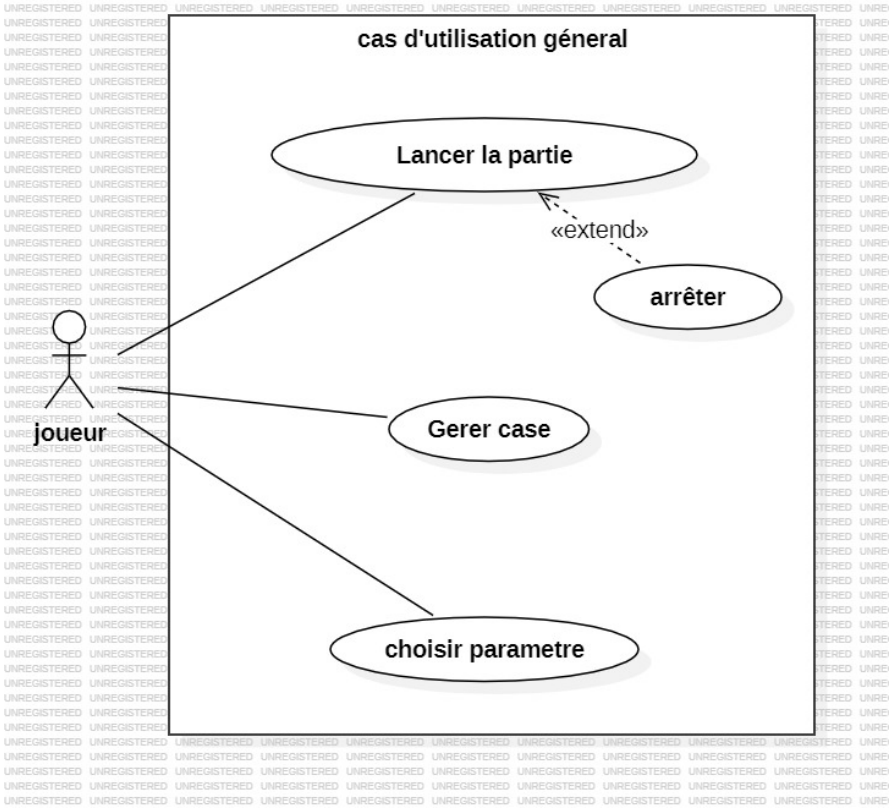

Figure 2 : diagramme de cas d'utilisation

Acteurs :

- Principal : Le joueur humain.

Précondition :

- Une partie n'est pas déjà lancée.

Scénario :

- Le joueur ouvre le jeu, il est dirigé vers la page d'acceuil où il y'aura trois boutons :
  - Lancer une partie.
  - Paramètre.
  - Quitter (ceci pour améliorer l'interaction de l'humain avec l'interface).

1. **Premier cas :** le joueur clique sur le bouton jouer directement, il aura donc la grille par défaut et commence directement à jouer en considérant les paramètres par défaut (grille 3 x 3, sans cases grisées, couleur par défaut).
2. **Deuxième cas :** Le joueur choisit d'abord les paramètres, il valide et la grille se lance selon ses paramètres et il commence à jouer. Il va alors jusqu'au bout de la partie.

Cas alternatif aux deux cas précédents :

Arrêter la partie : le joueur peut arrêter la partie quand il veut, et le jeu est terminé il est donc redirigé vers l'accueil.

b. Diagramme de séquence :

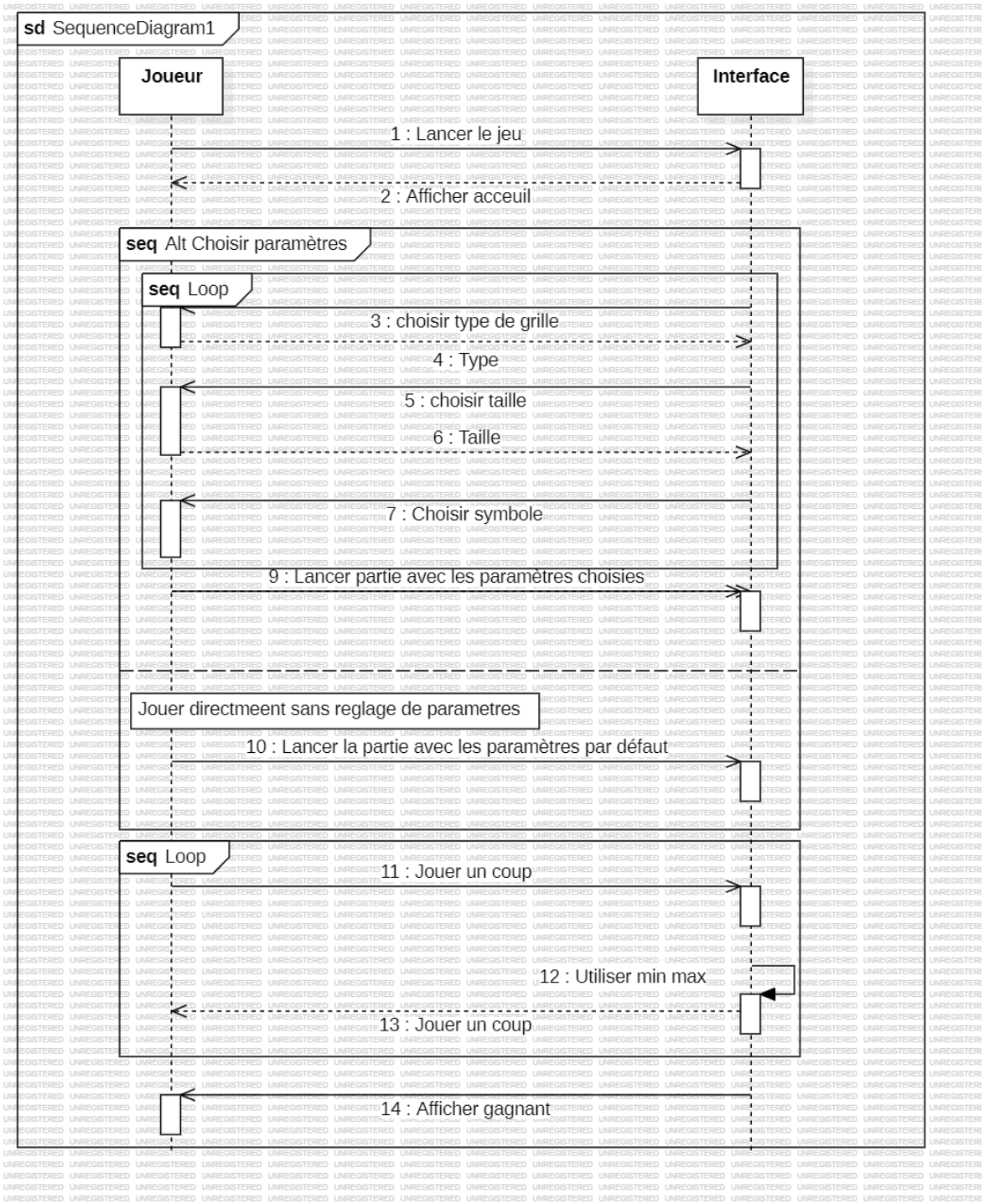

Figure 3 : diagramme de séquence

3. Architecture du projet et choix des modules

a. Architecture du projet

Nous avons adopté une approche **orientée objet** pour le développement de ce jeu. Et cela pour plusieurs raisons :

- Modélisation des entités : L'approche orienté objet permet de représenter les différentes entités du jeu de manière simple. (Joueur, représentant un joueur dans le jeu. La logique du jeu, représentant toutes les règles et la gestion du jeu…)
- Réutilisabilité : Avec cette approche, nous pouvons facilement modifier le jeu et avoir des bouts de code réutilisable.
- Encapsulation : L'approche Orientée Objet permet très facilement de cacher les détails de l'implémentation et de contrôle d'accès aux données. Cela nous permet également de pouvoir données accès aux variables par des méthodes getters/setters empêchant des modifications directes.
- Séparation des responsabilités : Cette approche nous permet de bien séparer les responsabilités entre les classes/entités du jeu rendant le code modulaire et extensible. Chaque classe ayant un rôle bien défini.

L'approche orientée objet nous a permit d'avoir un projet plus modulable et nous a évité également beaucoup de bugs.

TicTacToe/

|--- main.py : le fichier contenant la fonction main permettant d'exécuter le projet.

|--- gui.py : Interface graphique

|--- game_logic.py : Logique du jeu

|--- Player/

|--- Player.py : Classe de base pour les joueurs

|--- PlayerAI.py : Classe pour le joueur IA

|--- PlayerHumain.py : Classe pour le joueur humain

|--- color.py : Gestion des couleurs

Chaque fichier représente une classe qui gère une partie bien définie du projet.

b. Choix des modules

Notre code s'organise en classes qui s'organisent en catégories :

- **Modules type abstrait :** Les modules abstraits représente toute entité dans un application. Dans notre cas de jeu de Tic Tac Toe, on trouve dans cette catégorie les modules joueur (***Player***) qui est un module abstrait qui sera hérité par les deux autres modules plus spécifiques : ***PlayerHumain*** représentant le joueur humain et ***PlayerAI*** représentant IA/Algo MinMax/Ordinateur.
- **Modules type Action :** Les modules d'action représente un ensemble de traitement complexes dans une application. Ici on trouve principalement le module représentant la logique du jeu qui se nomme ***GameLogic***, ce module contiendra des fonctions pour vérifier si un des joueurs a remporté la partie (en vérifiant en colonnes en lignes et en diagonales), si le match est nul, et gérera toute la logique du Jeu Tic Tac Toe.
- **Modules Entrées/Sorties :** Les modules entrées/sorties contient l'ensemble des traitements et objets liés aux opérations d'entrées/sorties (cf. cours GL). Dans notre cas, l'interface utilisateur permet aux joueurs de pouvoir interagir avec notre système de la façon la plus fluide possible.

Le choix de l'architecture du projet, l'approche du développement et le choix des modules a été faite pour assurer un code de qualité, favorisant le refactoring, le jeu étant susceptible d'être modifié ou amélioré.

4. Convention de codage :

Nous avons adopté les mêmes conventions vues en cours à savoir :

- **Les variables** : chaque identifiant de variable commencera par la lettre majuscule désignant son type, puis le trigramme correspondant à la classe à qui elle appartient.
- **Les types** : chaque type commencera par la lettre T suivi du nom en minuscule.
- **Les noms de fonctions** : si la fonction appartient à une classe elle commencera donc par Le trinôme de cette classe sinon elle aura son propre nom commençant par une majuscule.

```python
from Player.color import TColor

class TPlayer:
    """
    Classe de base représentant un joueur de Tic Tac Toe.

    Attributs :
    - cPLRName (str) : Nom du joueur.
    - oPLRColor (TColor) : Couleur du joueur
    - bPLRIsAI (bool) : Indique si le joueur est une IA (True) ou un humain (False).
    """

    def __init__(self, cName: str, bIsAI=False):
        """
        Initialise un joueur avec un nom et définit s'il s'agit d'une IA ou d'un humain.

        @param cName : Nom du joueur
        @param bIsAI : Booléen indiquant si le joueur est une IA (par défaut False)
        """
        self.cPLRName = cName
        self.oPLRColor = None
        self.bPLRIsAI = bIsAI
```

Figure 4 : Exemple de code avec convention de nommage décrite

La classe se nomme **TPlayer :** le **T** pour designer que c'est un nouveau type, suivi par un nom significatif pour la classe qui est donc **Player** pour designer que cette classe va représenter des objets joueurs.

Dans le constructeur nous avons les attributs :

cPLRName : La variable commence par le type « **c** » pour chaine de caractère, suivi du trigramme de la classe Player (**PLR)** et enfin un nom significatif de l'attribut.

bPLRIsAI : le « **b** » pour booléen , puis le trigramme de la casse **PLR** et en fin un mot significatif. IsAI : on comprendra donc que cet attribut est un booléen qui confirme si l'objet Player instancié est un humain ou IA.

5. Tests unitaires :

Les tests unitaires :

Pour tester notre code nous avons décidé d'analyser la fonction **GLIcheck_column** : qui va évaluer si une colonne est remplie de façon avantageuse par rapport à un joueur (ce joueur est spécifié selon sa couleur passé en paramètre) c'est-à-dire si la partie est remportée par un joueur en analysant les colonnes de la grille du jeu.

Fonction choisie : GLIchech_column.

Signature de la fonction : def GLIcheck_column(self, iRow: int, iCol: int, oColor: TColor) -> TPlayer | None:

Les variables influençant cette fonction :

- iRow : l'indice de la ligne à vérifier.
- iCol : l'indice de la colonne à vérifier.
- oColor : la couleur du joueur

A) **Variable iRow :**

a) Valeur :

- **Domaine moyen :** 0 ≤ iRow≤ taille_max de la grille (3 ou 4 ou 5) -1 → **(A)**

Car l'évaluation des indices commencent à partir de zéro.

- **Domaine limite :** iRow ≥ taille_max de la grille (3 ou 4 ou 5). → **(B)**

b) Signes :

- iRow est positif → **(C)**
- iRow est négatif → **(D)**

B) **Variable iCol :**

a) Valeur :

- **Domaine moyen :** 0 ≤ iCol≤ taille_max de la grille (3 ou 4 ou 5) -1 → **(E)**

Car l'évaluation des indices commencent à partir de zéro.

- **Domaine limite :** iRow ≥taille_max de la grille (3 ou 4 ou 5). → **(F)**

b) Signes :

- iRow est positif → **(G)**
- iRow est negatif → **(H)**

C) **Variable oColor :**

c) Valeur :

- **Domaine moyen :** oColor = { Rouge, Vert, Jaune , Bleu} → **(I)**
- **Domaine limite :** oColor ne prend aucune couleur ou une couleur non sité dans le domaine moyen → **(J)**

L'énumération des jeux de tests :

A+C+E+G+I : exemple de test : dans le cas ou la grille a pour dimension 3 * 3 :

iRow = 2

iCol= 1

Ocolor= Rouge

B+C+E+G+J :

iRow= 3

iCol= 2

Ocolor = noir

Erreur car l'utilisateur utilise une couleur en dehors des couleurs permises.

OBSERVATION :

Pour plus de sécurité : on doit typer la valeur de iCol et iRow a un **unsigned int** (pour des langages de programmation ayant ce type) au lieu d'un type **int** pour éviter des valeurs négatives dans les indices.

Ocolor : créer une énumération des couleurs possibles pour faciliter les choix des joueurs et éviter une couleur non prise en compte par l'interface.

II. PARTIE ALGORITHMIQUE :

1. L'Algorithme MinMax :

Rappel : L'algorithme **Minimax** est une méthode utilisée pour prendre des décisions optimales. Dans notre cas, nous l'utilisons pour que l'adversaire du joueur humain puisse prendre des meilleures décisions.

- **Principe :**

Nous utilisons l'algorithme de minmax pour trouver le meilleur coup pour le joueur AI/Ordinateur en simulant tous les coups possibles, tout en supposant que l'adversaire (le joueur humain) joue de manière optimale.

L'algorithme explore un arbre de décisions, où chaque nœud représente un état du jeu.

- **Min** : Correspond aux tours de l'adversaire, qui cherche à minimiser le score du joueur.
- **Max** : Correspond aux tours du joueur AI, qui cherche à maximiser son score.
  - **Étapes :**

a. **Exploration de l'arbre** : L'algorithme simule tous les mouvements possibles jusqu'à une certaine profondeur. Nous avons utilisé la profondeur dans ce jeu pour gérer le niveau de jeu. Etant donné que plus la profondeur de l'arbre est élevée, plus le joueur analyse les meilleurs coups possibles.

b. **Évaluation des états** : Chaque état du jeu est évalué donnant +10 au joueur pour un bon coup possible et un -10 pour un état où le joueur est dans la bonne situation de gagner sinon un 0 pour un match nul. Cela indiquera donc à l'AI qu'il n'y a pas trop d'intérêt de considérer cet état.

c. **Propagation des scores** : A la fin de l'exploration, arrivée au niveau des feuilles, le joueur AI remontent dans l'arbre en faisant la somme des scores.

d. **Décision finale** : Le joueur AI peut ainsi choisir le mouvement associé au meilleur score.

2. Comment s'organise le minmax dans le code ?

Pour que le joueur IA joue son tour il utilise quatre fonctions (PLRjouer, PLRMinmax, PLRget_possible_moves et simulated_game)

Roles de ces functions :

- La fonction PLRget_possibles_moves : Fais un passage sur l'état du jeu et enregistre tous les coups/mouvements possibles.
- La fonction PLRsimulate_move : A partir des coups/mouvements possibles, le joueur AI va simuler un coup et elle formera ainsi un cas possible de jeu.
- La fonction PLRMinMax : Cette fonction se rappelle récursivement avec la profondeur qui s'incrémente jusqu'à la profondeur maximale passée en paramètre pour permettre à la fin de récupérer le score le plus avantageux ainsi que les coordonnées de la case qui sera jouée par l'IA.
- La fonction PLRjouer : Peut maintenant utiliser le bon coup retourné par la fonction Minmax pour jouer.

Stratégie d'implémentation de niveau de difficulté.

Nous savons que plus la profondeur est élevée, plus le joueur AI prend la décision la plus optimale, cela nous permet donc d'incrémenter la profondeur du jeu à 1 à chaque fois que le joueur Humain qui gagne la partie.

LDA du minmax : sur papier :

L'Algorithme en LDA

① Les structures et les nouveaux types utilisés dans l'algorithme.

(A) structure game:

cette structure déjà créé dans un autre fichier (classe).

```
Type Game_logic = structure
    player_1 : Player // une autre structure
    player_2 : Player
    taille : entier // la dimension de la grille
    condition : entier // nb de symbole à aligner pour gagner
    Grille : Type matrice = Tableau[0..taille-1] 0..taille-1] d'entiers
    liste_move // Type List de Tuple (ligne, colonne)
                ↑ List
Fin structure
```

(B) structure List:

une structure indispensable pour generer les coups joues, ceux restants.

```
Type List = structure
    head : ↑Noeud   // la Tête
    Queue : ↑Noeud  // la queue de la liste.
    Size : entier   // taille de la liste.

Fin structure
```

(C) structure Noeud

: la structure qui compose la liste:

```
Type Noeud = structure
    Suiv : ↑ Noeud.
    Valeur : ↑Tuple (coordonnées de la case)

Fin structure
```

(D) Tuple

```
Type Tuple = structure
    indice_ligne : entier (indique l'indice ligne de la case)
```

1

Figure 1 Les structures utilisées dans l'algorithme MinMax

indice_colonne : entier (indique l'indice colonne de la case)
Fin structure.

(2) LDA des fonctions:

Fonction get_possible_moves

dans le code cette fonction se trouve dans le dossier player la classe playerIA sous le nom de PLR get_possible_moves.

entrées:
game : Game_logic (instance de la partie, grille et son état)

Précondition: l'instance game doit être à l'état actuel du jeu càd doit contenir dans sa liste_move les cases déja jouées

Sortie: moves : List : la liste qui contient les cases possibles à jouer encore.

Post conditions: moves doit contenir que les case [i,j] tq ces cases ne sont jamais jouées donc de couleur grise.

L'Algorithme Get_possible_moves (game : Game_logic): List

Variables:
i , j : entier
moves : List (Tuple)

taille = game -> taille.

Début
  Pour i de 0 à taille faire :
    Pour j de 0 à taille faire.
      Si (game -> grille [i,j] == T(Colo. Vide) alors
        moves.ajouter (game -> grille [i,j])
      fin si
    fin pour
  fin pour
  Retourner moves
fin

//ajouter : est une fonction qui ajoute un noeud à la liste

2

Figure 2 : La fonction get_possible_moves

Fonction Simulate_move

Cette fonction se trouve dans le dossier Player, Classe PlayerIA sous le nom de PLR simulate_move.

//cette fonctions génère une nouvelle instance du jeu en simulant un coup

Prend une case en paramètre et la place sur une copie de la grille actuelle.

entrée :

game = Game_logic : instance du jeu, ce qui nous intéresse est la grille à l'état actuel.

row : entier // indice de ligne.

col : entier // indice de colonne.

Précondition :

$0 \leq row \leq game.taille - 1$ et $0 \leq col \leq game.taille - 1$

Sortie : une nouvelle grille avec la case simulé.

copie : Game_logic

Postcondition : la grille retournée doit être une copie pas modification réelle de la grille du jeu

Algorithme Simulate_move (game : Game_logic, row : entier, col : entier) : Game_logic :

Variables :

copie : Game_logic

copie = copie (game)

// copie : une fonction qui realise la copie de la grille passée en paramètre.

Début :

Si a qui le Tour (copie) = player 1 alors :

copie → grille [row, col] = copie → player 1. recupere couleur player 1 ()

Sinon

copie → grille [row, col] = copie → player 2. recupere couleur player 2 (

finsi

3

Figure 3 : La fonction simulate-move

```
    donne_le_Tour_au_prochain_joueur (copie)
    Retourner copie
Fin
```

Fonction min max:

Dans le code cette fonction se trouve dans le dossier Player classe .player IA

Elle utilise les 2 fonctions sitées précedemment

entrée: game : Game_logic

profondeur : entier // ce qui reflete le niveau de difficulté du jeu

maxi_mini : Booleen

// il specifie si on maximise (True) ou Minimise (False) le score

Précondition:

Sortie: Tuple (meilleur_score, meilleur coup à jouer) avec coup_à_jouer = Tuple // case à jouer

Post condition: la case designé est prise donc disponible.

Algorithme: MinMax (game: Game_logic, profondeur: entier, Maxi_mini: Booléen): Tuple

Variable: simulated_game = Game_logic
possible_moves: List best_move: Tuple best_score = entier

```
Début // cas d'arrêt.
    Si game.Verifie_gagnant(player1.couleur(1)) == // player IA / player 1 :
        Retourner 10, Null
    Sinon Si game.Verifie_gagnant(player2.couleur(1)) == player2
        Retourner -10, Null
    Fin si
    Sinon si game.Verifie_pos_gagnant( ) == True
        Retourner 0, Null
```

4

Figure 4 : l'algorithme minmax debut

```
/* cette partie de l'algorithme traite le cas ou la grille reelle ou simulée est
completement rempli alors si la partie est en la faveur du joueur 1
l'IA on retourne la valeur 10 si c'est en la faveur du joueur 2
humain on retourne -10 si match Null alors une valeur 0 est
retournée, dans l'algorithme on a utilisé les fonctions
Verifie_gagnant qui retourne le player gagnant s'il y en a un.
Verifie_pas_gagnant: retourne true si le match est null */

possible_moves = get_possible_moves(game)

Si (Maxi_mini == True) alors
    best_score ← -20  // valeur tres basse
    best_move ← (0,0)
    Pour chaque couple (row, col) de possible_moves faire:
        simulated_game = simulate_move(game, row, col)
        Score, _ = min max (simulated_game, profondeur-1, false)
        // recursivité sur min max
        Si Score > best_score alors
            best_score = Score
            best_move = (row, col)
        fin si
        Retourner best_score, best_move
    fin pour
Sinon
    best_score ← -20
    best_move ← (0,0)
    Pour chaque couple (row, col) de possible_moves faire
        simulated_game = simulate_move(game, row, col)
        Score, _ = min max (simulated_game, profondeur-1, True)
        Si Score < best_score alors
            best_score = score
            best_move = (row, col)
        fin si
```

5

Figure 5 : L'algorithme minmax suite

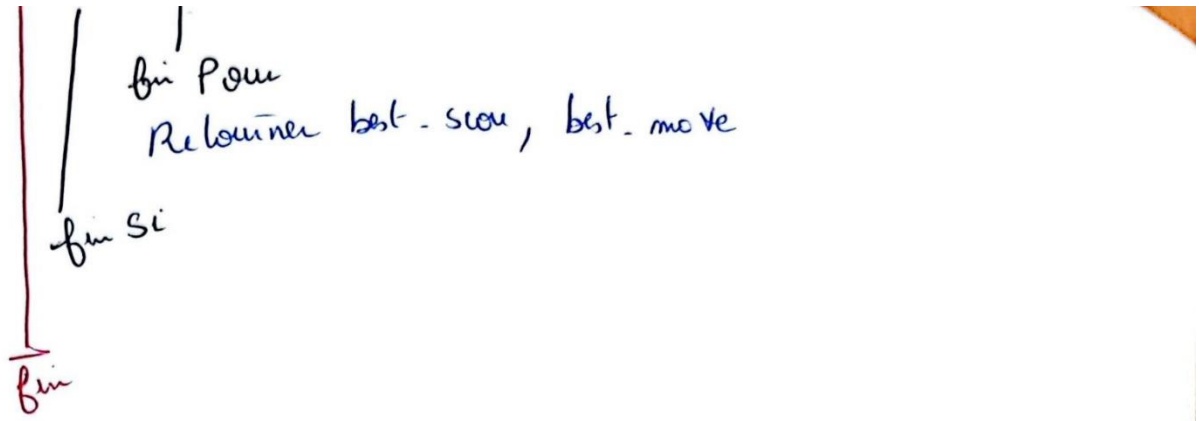

Figure 6 :Ll'algorithme minmax fin

III. PARTIE IHM

1. Interface :

a. La page d'accueil :

Quand l'utilisateur/joueur clique sur l'icone du jeu il tombera sur cette fenêtre ou deux boutons essentiels se trouveront :

- **Jouer** : En cliquant ici, une partie de jeu se lancera avec des paramètres par défaut notamment une grille de jeu 3*3 et un niveau 0.
- **Paramètre** : En cliquant ici, l'utilisateur remplira un petit formulaire lui permettant de choisir ses paramètres de jeu avant de pouvoir lancer le jeu.
- **Quitter :** Pour éteindre tout le jeu, plus facile que d'aller sur la barre de la fenêtre.

Figure 7 : Interface accueil

b. La page des paramètres du jeu :

Pour le choix des symboles, nous partons sur des jetons pour lesquelles le joueur aura le droit de choisir la couleur. Les couleurs disponibles seront sous forme d'une liste déroulée.

Nous avons également laissé le choix à l'utilisateur de choisir la dimension de sa grille

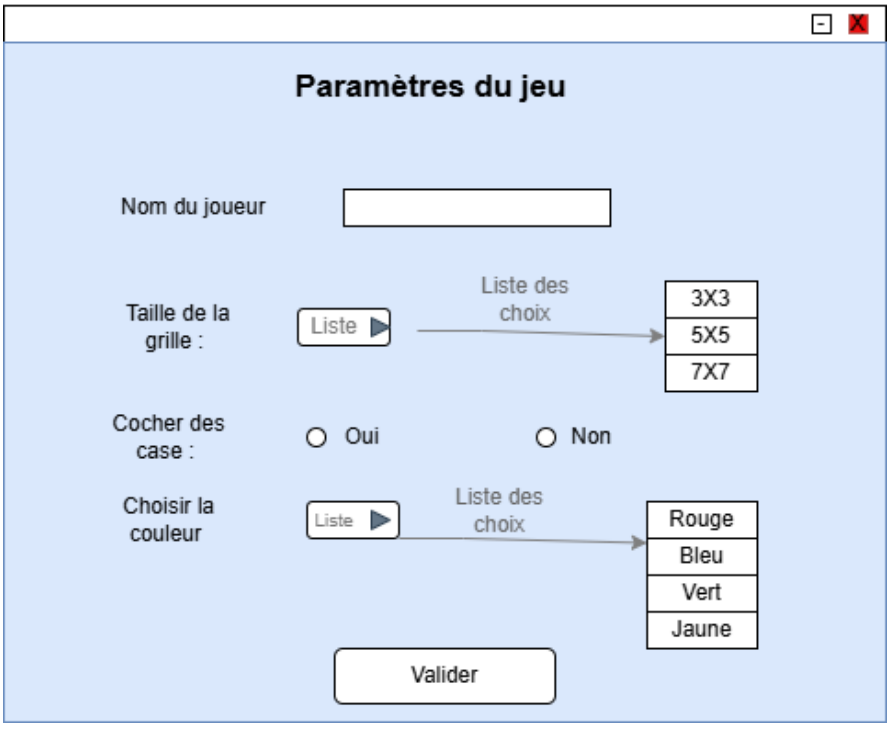

Figure 8 : Interface paramètres

c. La grille du jeu :

D'abord un menu qui va permettre de terminer la partie, un undo (en option) pour décocher une case (la dernièrement cliquée) et puis un bouton pour réinitialiser une partie et permette de recommencer à nouveau de jouer.

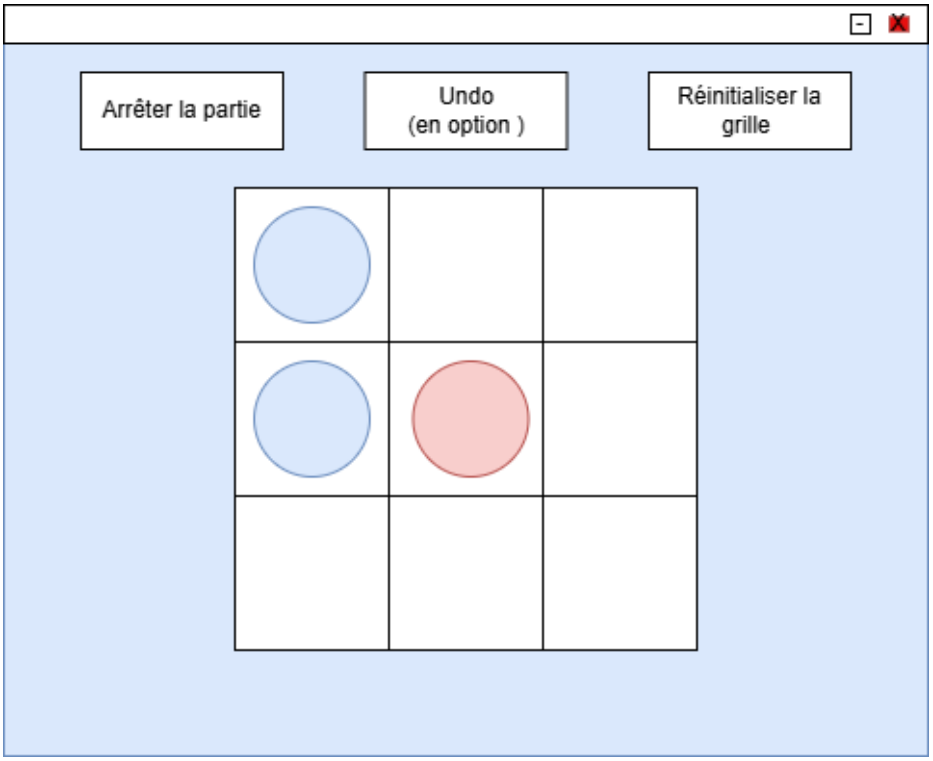

Figure 9 : Interface lancement d'une partie du jeu

Concernant les options :

Nous avons décidé d'aller avec la fonction undo qui va permettre à l'utilisateur de revenir sur le dernier choix de case effectué.

IV. FEEDBACK DU PROJET

Dans cette partie nous allons revenir sur notre expérience avec le projet, nos difficultés et comment at-on pu gérer nos problèmes.

Le projet est une bonne mise en situation pour l'application des normes du génie logiciel dans le développement, la création d'une interface efficace convenable pour tout type d'utilisateur, ainsi que la production de code à partir d'un bon algorithme facilitant ainsi l'exercice de la programmation dans un langage précis.

Nous avons débuté le projet par la phase modélisation où nous avons pensé à la meilleure architecture à suivre dans le développement et avons finalement convenu de travailler

avec l'approche objet, en parallèle nous avons aussi réalisé (la maquette) le premier prototype de l'interface pour essayer de voir les liens ainsi que les modules que nous devons développer.

Ensuite nous avons commencé à réfléchir sur l'algorithme minimax qui représente le noyau du projet, nous avons donc produit un premier prototype de l'algorithme que nous avons intégré dans la classe PlayerIA et adopter pour convenir aux contraintes du langage python.

Une fois les composantes essentielles du projet établi, nous avons commencé à développer l'interface utilisateur (qui a déjà était pensé dans la phase modélisation et la production de la maquette ci-haut **(cf. figure 7, 8 et 9)** en parallèle avec la logique du jeu pour identifier simplement les bugs.

1. Difficultés rencontrées :

Les difficultés rencontrées lors du développement de ce projet était minime ceci sans doute grâce au bon démarrage ou 6h en était passé à la modélisation avant de se lancer dans le code ou nous avons décidé de suivre une approche orienté objet dont les raisons était évoqué dans un chapitre précédent.

2. Les taches non accomplies :

Une des tâches que nous n'avons pas pu réaliser est l'option « aligné et non aligné », en raison de la contrainte de temps. Comme cette fonctionnalité n'est pas essentielle et que le jeu a été conçu pour permettre des factorisations, nous avons décidé de la laisser comme une piste d'amélioration.

Le Random marche mais si le joueur remporte la partie le niveau de profondeur augmente et la grille se réinitialise sans garder les cases grisées.

Voici le rapport d'évaluation des autres étudiants vis-à-vis du code lors de la dernière séance :

NB : Nous avons essayé de prendre en considération les remarques qui nous ont été adressé et donc des changements dans l'interface ainsi que le code sont possible.

Mini-formulaire d'évaluation

Nom du binôme développeur : MERA BTENE
M VOUNGOU

Partie du questionne concernant l'utilisateur (qui il est)

Informations personnelles (pour des raisons de confidentialité entre étudiants, elles sont enlevées, mais on peut demander AGE, GENRE, etc)

Votre statut (cocher les cases) :

Etudiant	Prof	Autre
X O		

Votre niveau d'étude :

PEIP	DI3	DI4	DI5	Thèse
	X O			

Vous jouez aux jeux vidéo :

1 fois par jour	1 fois par semaine	1 fois par mois	1 fois par an	Jamais
X	O			

Vous avez déjà programmé un jeu :

Non	Oui
X	O

Tâche à réaliser : faire jouer l'utilisateur (2 parties) avec les paramètres suivants :

- 7 colonnes, 6 lignes, 4 pions a aligner, profondeur 4 pour le minmax, en jouant un asset
- 5 colonnes, 5 lignes, 3 pions à aligner, profondeur 3, pyramide, ne pas aligner

Pas de phase d'accueil ou d'apprentissage : l'utilisateur joue directement.

L'interface proposée est globalement (cocher une seule case) :

	1	2	3	4	
Lente				O X	Rapide
Illisible			O	O X	Lisible
Laide			X O		Belle

Figure 10 : Fiche d'évaluation étudiants

	1	2	3	4	
Ennuyeuse		X	O		Stimulante

	1	2	3	4	
Difficile à comprendre				OX	Facile à comprendre

L'algorithme IA est :

	1	2	3	4	
Lent		O		X	Rapide

	1	2	3	4	
Facile à battre		O	X		Difficile à battre

Autres points sur votre expérience :

Le réglage des paramètres a été :

	1	2	3	4	
Difficile		XO			Facile

L'asset (atout) proposé dans le jeu était :

	1	2	3	4	
Pas intéressant			XO		Rigolo

Si vous deviez jouer régulièrement à ce jeu, vous choisiriez :

	1	2	3	4	
Plutôt une autre interface		X	O		La même interface

Figure 11 : Fiche d'évaluation Etudiant suite

3. Guide d'utilisation du jeu :

Ici nous revenons sur les fonctionnalités qui peuvent tromper l'utilisateur :

- **Fonction Undo** : Pour essayer la fonction undo alors il faudrait faire deux undo(s) un après l'autre c'est-à-dire annuler le dernier coup du joueur et celui de l'IA pour pouvoir continuer sinon la grille devient injouable.
- **Profondeur :** Pour l'implémentation de l'algorithme minmax , sur l'interface des paramètres nous n'évoquons pas la possibilité de la difficulté du jeu qui est relié donc à la profondeur du minmax, car nous avons décidé que la profondeur s'incrémente à chaque partie remportée par le joueur donc le niveau 1 est facilement jouable et gagnable pour l'humain mais au fur à mesure des niveaux augmentant la profondeur du minmax augmente le rendant plus puisant et donc minimisant les chances de remporter la partie pour l'humain.
- **Random** : il faut choisir une valeur inferieure à la moitié de nombre de cases de la grille, si l'utilisateur ne prend pas ceci en considération une note s'affichera quand il cliquera sur « Jouer » pour lui rappeler de ne pas dépasser un certain nombre de cases à griser.

Cas sans cases grises : Le joueur devrait cocher Non pour l'option case grisée et aussi mettre la valeur 0 dans le nombre de case à griser sinon le jeu ne se lancera pas.

Pyramid : dans cette option puisque les dimensions de grilles créent des matrices carrées visualiser la pyramide est un peu problématique.

CONCLUSION

Ce projet de développement d'application qui était une mise en situation de développement d'un projet de bout en bout nous a permit d'appliquer beaucoup de connaissances accumulées en Cours Magistraux, en Travaux dirigés et en Travaux pratiques.

Niveau soft-skills, le développement de ce projet ce en binome, nous a également poussé à la collaboration et à la communication. Ces compétences, très utiles pour un ingénieur.

Niveau technique, le développement de ce projet nous a poussé aussi à effectuer des recherches poussées et avancées, à lire des documentations pour plus d'éclaircissement quant à ce qui concerne des concepts assez complexes.