Rapport

Ultimate TicTacToe

Abdelwaheb SEBA
Mohammed Ryad DERMOUCHE

2024 - 2025

Table des matières

1 Objectifs du projet

1.1 Contexte pédagogique

Ce projet a été réalisé dans le cadre de l'Unité d'Enseignement **Intelligence Artificielle**, en binôme. L'objectif principal était de concevoir, implémenter et analyser un *jeu stratégique à deux joueurs* dans lequel un **joueur humain affronte une Intelligence Artificielle (IA)**. L'accent a été mis non seulement sur le respect des règles du jeu choisi, mais surtout sur la qualité des algorithmes d'IA développés et leur performance en situation réelle.

1.2 Exigences du projet

Le cahier des charges imposait plusieurs éléments structurants :

- **Choix d'un jeu stratégique** à deux joueurs, déterministe, à information parfaite, suffisamment complexe pour exiger une véritable stratégie, mais réalisable dans le temps imparti.
- **Développement d'au moins trois IA** de difficultés différentes, intégrant des algorithmes comme Minimax, avec ou sans élagage alpha-bêta, et différentes heuristiques d'évaluation.
- **Maintien d'un temps de réponse raisonnable**, afin d'assurer une expérience utilisateur fluide et agréable.
- **Organisation de tournois entre IA**, avec au moins 50 parties par duel, pour analyser les performances relatives des différentes stratégies.
- **Rédaction d'un rapport complet** contenant une description du jeu, des algorithmes utilisés, des choix de conception, une analyse des résultats, ainsi qu'un bilan du projet.

1.3 Choix du jeu : Ultimate TicTacToe

Après analyse des différents jeux proposés (Dames, Othello, Quoridor, Puissance 4...), notre choix s'est porté sur **Ultimate TicTacToe**, pour les raisons suivantes :

- Il s'agit d'une version enrichie du morpion classique, intégrant des mécanismes de **stratégie multi-niveaux**, grâce à la structure 3×3 de sous-plateaux.
- Le jeu offre un bon compromis entre **complexité stratégique** et **faisabilité algorithmique**, ce qui le rend adapté à une implémentation en Python avec IA.
- Il permet l'utilisation de techniques classiques d'intelligence artificielle (Minimax, élagage alpha-bêta, heuristiques personnalisées), tout en rendant les parties **rapides et lisibles**.

1.4 Méthodologie suivie

Le développement a suivi une approche itérative et incrémentale, inspirée de la méthodologie proposée dans les consignes du projet :

1. **Modélisation de l'état du jeu** (*PetitPlateau*, *MorpionUltime*) avec les fonctions associées (validation, duplication, affichage...).
2. **Implémentation d'une boucle de jeu** pour deux joueurs humains, permettant de tester les règles du jeu dans un cadre neutre.
3. **Développement d'une première IA** (IAFacile) basée sur un Minimax simplifié à profondeur 1.

4. **Ajout d'une IA intermédiaire** (IAMoyenne), utilisant Minimax avec profondeur 3, et un taux d'erreur volontaire pour simuler une stratégie humaine imparfaite.
5. **Développement d'une IA avancée** (IADifficile), utilisant Minimax avec élagage alpha-bêta, profondeur 4, et heuristiques complexes (fourches, mobilité...).
6. **Création d'un module de tournoi automatique** entre IA, pour générer des statistiques et comparer les performances (victoires, nuls, efficacité...).
7. **Documenter l'ensemble du code** et rédiger un rapport analytique complet présentant les résultats.

2 Présentation du jeu

2.1 Description générale

Ultimate TicTacToe (ou Morpion Ultime) est une variante stratégique du célèbre jeu du morpion. Au lieu d'une simple grille de 3×3, cette version se joue sur une grille composée de neuf petits plateaux de morpion (soit une grille 9×9 au total, structurée en 3×3 sous-plateaux).

Chaque petit plateau fonctionne comme un jeu de morpion classique. Le but est de gagner trois sous-plateaux alignés horizontalement, verticalement ou en diagonale, exactement comme dans un morpion classique, mais à l'échelle des sous-plateaux.

2.2 Règles du jeu Ultimate TicTacToe

Les règles de base sont les suivantes :

- Le jeu commence avec un joueur **X** et l'autre joueur **O**.
- Lorsqu'un joueur joue dans une case d'un sous-plateau, il envoie son adversaire jouer dans le sous-plateau correspondant à la position de la cellule choisie.
- Par exemple, si un joueur place son symbole dans la case (1,2) d'un sous-plateau, alors l'adversaire devra jouer dans le sous-plateau global situé en (1,2).
- Si le sous-plateau ciblé est déjà terminé (victoire ou nul), alors le joueur peut jouer librement dans n'importe quel sous-plateau encore actif.
- Un sous-plateau est considéré comme **gagné** s'il est remporté par un joueur selon les règles du morpion classique.
- Le jeu se termine lorsqu'un joueur remporte 3 sous-plateaux alignés ou lorsque tous les sous-plateaux sont terminés (victoire ou match nul).

3 Architecture du projet

3.1 Organisation générale

Le projet est structuré de manière claire autour d'un dossier principal nommé `code/`, contenant tous les fichiers sources Python. On y retrouve notamment :

- **`game.py`** : gestion des règles du jeu *Ultimate TicTacToe*, structures de données pour les plateaux, logique de partie, validation des coups, et affichage console.
- **`ai.py`** : définition des trois intelligences artificielles (`IAFacile`, `IAMoyenne`, `IADifficile`) et des fonctions d'évaluation heuristique.
- **`tournament.py`** : module permettant d'organiser et d'automatiser des tournois entre IA, avec statistiques de performance.

- **`main.py`** : point d'entrée principal du projet, proposant un menu interactif console permettant de jouer ou de lancer un tournoi.

À la racine du projet, on trouve également :

- **`README.md`** : document d'instructions permettant d'exécuter le projet facilement, avec explication des dépendances nécessaires et des différentes fonctionnalités disponibles.
- **Le présent rapport** : document PDF `SEBA_DERMOUCHE.pdf`, décrivant en détail le fonctionnement du projet, les algorithmes utilisés, les choix d'implémentation, les résultats des IA, et un bilan final.

3.2 Structure des fichiers

Fichier	Description
`game.py`	Structures de données du jeu (plateaux), règles, gestion des parties, interactions avec l'utilisateur.
`ai.py`	Implémentation des différentes IA avec Minimax, élagage alpha-bêta, et fonctions heuristiques.
`tournament.py`	Système de tournois entre IA, mesures statistiques (temps, nombre de coups, taux de victoire).
`main.py`	Menu principal : lance le jeu en mode humain ou tournoi.
`README.md`	Instructions d'installation, d'exécution, dépendances, explications succinctes.
`SEBA_DERMOUCHE.pdf`	Rapport de projet au format PDF.

3.3 Choix techniques

Le projet a été intégralement réalisé en **Python 3**, sans interface graphique, conformément aux consignes. Le code est organisé de façon modulaire et claire, avec :

- Une séparation stricte entre la logique du jeu, les IA, et les outils d'expérimentation.
- Une documentation interne soignée (commentaires).
- L'usage exclusif de bibliothèques standards (aucune dépendance externe nécessaire).
- Un usage du paradigme orienté objet (OO) pour modéliser les entités du jeu (plateaux, IA...).

4 Explication du code

4.1 Fichier `game.py` : logique du jeu *Ultimate TicTacToe*

Ce fichier regroupe toute la mécanique du jeu : structures de données, règles (et deux modes console pour tester rapidement).

4.1.1 Classe `PetitPlateau`

Sous-grille 3×3 indépendante. Elle encapsule la logique d'un morpion classique.

`__init__()`
Appelle `reinitialiser` pour créer un plateau vide et remettre tous les états (`vainqueur`, `jeu_termine`) à leur valeur initiale.

`reinitialiser()`
Construit une matrice 3×3 de chaînes " ␣ " et réinitialise les drapeaux `vainqueur` et `jeu_termine`. Cette méthode est réutilisable pour recommencer une partie sans recréer l'objet.

`coup_valide(ligne, colonne)`
Retourne `True` si :

1. les indices sont dans $[0, 2]$,
2. la case est vide,
3. le sous-plateau n'est pas déjà terminé.

`jouer_coup(ligne, colonne, joueur)`

- Vérifie la légalité via `coup_valide`.
- Place le symbole, puis déclenche `verifier_etat`.

`verifier_etat(joueur)`
Recherche une victoire sur :

1. les 3 lignes,
2. les 3 colonnes,
3. les 2 diagonales.

Si aucune victoire : teste `est_nul()` pour marquer "DRAW" quand le plateau est plein.

`est_nul()`
Parcours exhaustivement les 9 cases ; renvoie `True` s'il n'y a plus de case vide.

`obtenir_vainqueur()`
Accesseur simple pour savoir qui a gagné (`'X'`, `'O'`, `"DRAW"` ou `None`).

`dupliquer()`
Retourne une copie profonde (deep-copy) du sous-plateau. Indispensable pour que l'IA puisse simuler des coups sans altérer la partie réelle.

`__str__()`
Transforme le plateau en chaîne multi-ligne (affichage *debug*).

4.1.2 Classe `MorpionUltime`

Plateau principal 3×3 de `PetitPlateau`. Implémente les règles spécifiques de l'Ultimate TicTacToe.

`__init__()`
Construit directement l'objet via `reinitialiser()`.

`reinitialiser()`

- Construit la matrice 3×3 de `PetitPlateau`.
- Fixe `joueur_courant = 'X'` et remet tous les drapeaux (plateau actif, vainqueur, etc.) à leur état initial.

`coup_valide(sp_l, sp_c, cel_l, cel_c)`
Valide un coup global :

1. indices du sous-plateau dans $[0, 2]$,
2. sous-plateau non terminé,
3. respecte la contrainte du `plateau_actif` (sauf si `None`),
4. délègue enfin au `coup_valide` du sous-plateau ciblé.

`jouer_coup(sp_l, sp_c, cel_l, cel_c)`

1. Refuse si le coup est illégal ou si la partie est finie.
2. Pose le coup dans le sous-plateau.
3. Appelle `mettre_a_jour_etat_global()`.
4. Calcule le `plateau_actif` imposé à l'adversaire ; s'il est déjà terminé, le prochain coup est libre.
5. Bascule `joueur_courant` si la partie continue.

`mettre_a_jour_etat_global()`

- Construit un plateau « virtuel » de 3×3 contenant le vainqueur de chaque `PetitPlateau`.
- Recherche une ligne, colonne ou diagonale homogène (victoire globale).
- Déclare un nul global si tous les sous-plateaux sont terminés *ou* si aucune combinaison gagnante n'est mathématiquement possible.

`obtenir_coups_valides()`
Génère tous les coups légaux. Deux cas :

- `plateau_actif` imposé : on ne parcourt qu'un seul sous-plateau.
- Coup libre : on parcourt les 9 sous-plateaux non terminés.

`obtenir_vainqueur()`
Renvoie le vainqueur global (`'X'`, `'O'`) ou `None` (nulle/en cours).

`dupliquer()`
Deep-copy complète de la partie. Essentiel pour le Minimax : l'IA peut simuler sans affecter la partie réelle.

`afficher_plateau()`
Assemble les 9 sous-plateaux en une grande grille ASCII 9×9 et affiche également le prochain plateau imposé + le joueur courant.

4.1.3 Fonctions console

Deux fonctions `stand-alone` pour lancer rapidement une partie depuis le terminal :

`humain_vs_humain()`
Boucle de jeu à deux joueurs humains, saisie clavier (indices 0–2).

`humain_vs_ia(ia)`
Remplace un des joueurs par une instance d'IA (objet conforme à l'interface définie dans `ai.py`). La fonction gère :

- la saisie sécurisée côté humain ;
- l'appel à `ia.get_move()` côté machine ;
- l'annonce finale du résultat.

4.2 Fichier `ai.py` : intelligences artificielles

Trois niveaux d'IA héritent d'une même classe de base et partagent deux familles d'heuristiques.

4.2.1 Classe `IA` – socle commun

`__init__(joueur)`

- Enregistre le symbole de l'IA (`self.joueur`) et déduit celui de l'adversaire (`self.adversaire`).
- Attribue un nom par défaut (`"IA de Base"`).
- Initialise les compteurs : `self.coups_evalues` (statistiques) et `self.score_dernier_coup` (qualité du dernier coup).

`obtenir_coups_valides(partie)`
Récupère la liste des actions légales en appelant `partie.obtenir_coups_valides()`. Cette factorisation évite de dupliquer le code dans chaque sous-classe.

4.2.2 Heuristiques partagées

`evaluer_ligne`

- Ligne bloquée (symboles mixtes) $\Rightarrow$ score 0.
- Ligne favorable : 10^n où $n =$ nombre de pions de l'IA.
- Ligne défavorable : -10^n pour l'adversaire.

`evaluer_sous_plateau`

- Bonus/malus ± 20 pour le contrôle du centre.
- Somme des scores de chaque ligne, colonne et diagonale via `evaluer_ligne`.

`evaluer_ultime`

1. Ajoute $\pm 10\,000$ par sous-plateau déjà gagné ou perdu.
2. Insère la contribution des sous-plateaux non terminés (`evaluer_sous_plateau`).
3. Construit un « plateau global » 3×3 ; ses lignes/colonnes/ diagonales sont évaluées et pondérées $\times 10$.

`evaluer_difficile`

- Point de départ : `evaluer_ultime`.
- Bonus de mobilité : $+1$ par coup légal restant.
- Détection des fourches : $\begin{cases} +300 & >=2 \text{ menaces IA} \\ -300 & >=2 \text{ menaces adverses} \end{cases}$

4.2.3 Classe `IAFacile`

`__init__`
Spécifie le nom et la profondeur (1 ply d'anticipation).

`get_move`

1. Pour chaque coup légal : duplique la partie, joue le coup, évalue la nouvelle position avec `evaluer_ultime`.
2. Sélectionne le coup au score maximal et mémorise ce score dans `self.score_dernier_coup` .
3. Aucun arbre de recherche : complexité linéaire dans le nombre de coups disponibles.

4.2.4 Classe `IAMoyenne`

`__init__`
Fixe une profondeur de recherche égale à 3 et un taux d'erreur stratégique de 30 % (coup aléatoire pour « humaniser »).

`minimax`

- Parcourt récursivement l'arbre des coups jusqu'à la profondeur 3 (ou jusqu'à un état terminal).
- Nœuds *Max* : l'IA maximise la valeur ; nœuds *Min* : l'adversaire la minimise.
- Pas d'élagage : toutes les branches sont explorées $\Rightarrow$ coût exponentiel $\mathcal{O}(b^p)$ (avec $b \approx 40$, $p = 3$).

`get_move`

1. Tirage aléatoire d'un coup avec probabilité 0,3.
2. Sinon, applique Minimax sur chaque coup racine ; conserve la ou les meilleures valeurs et choisit aléatoirement en cas d'égalité.
3. Enregistre la valeur retenue dans `self.score_dernier_coup`.

4.2.5 Classe `IADifficile`

`__init__`
Profondeur fixée à 4.

`minimax` (α-β élagage)

- Deux bornes sont propagées : α (au moins) pour Max, β (au plus) pour Min.
- Quand $\alpha \geq \beta$, la branche courante ne peut plus influencer le résultat global : elle est coupée.
- Les feuilles sont évaluées avec `evaluer_difficile`.

`get_move`

1. Parcourt chaque coup racine avec $\alpha = -\infty,\ \beta = +\infty$.
2. Met à jour α après chaque évaluation ; coupe les futures branches inutiles.
3. Retient le coup au meilleur score et l'inscrit dans `self.score_dernier_coup`.

Résumé. `IAFacile` : évaluation directe (1-ply). `IAMoyenne` : Minimax profondeur 3 sans élagage + erreurs volontaires. `IADifficile` : Minimax 4 avec élagage α-β et heuristique avancée (mobilité / fourches). Chaque ajout (profondeur, élagage, heuristique) augmente la puissance tout en maîtrisant le temps de calcul.

4.3 Fichier `tournament.py` : gestion des tournois

Ce module ne modifie pas les règles du jeu ; il sert exclusivement à *orchestrer* des confrontations entre IA, mesurer leurs performances et proposer une interface console minimaliste.

4.3.1 Classe `Tournoi`

Instance unique qui pilote les matchs *IA vs IA* et, si demandé, un tournoi « tous-contre-tous ».

`__init__()`
- `self.resultats` — dictionnaire qui comptera les victoires et les nuls.

`lancer_match(ia1, ia2, nb_parties=10)`

But. Faire jouer deux IA durant n manches en alternant le premier joueur afin d'annuler l'avantage du « X ».

Entrées.
- `ia1`, `ia2` : instances héritant de `IA`.
- `nb_parties` : nombre de manches (défaut 10).

Sortie. Tuple (*resultats*, *perf*) où :
- *resultats* = {`ia1.nom` : n, `ia2.nom` : m, `'nuls'` : d} ;
- *perf* = {temps total, temps moyen, scores cumulatifs, score moyen par coup}.

Algorithme.

1. Initialise compteurs, lance un chronomètre.
2. Pour chaque partie :
   (a) Crée un `MorpionUltime` vierge.
   (b) Inverse les rôles 'X'/'O' une fois / deux pour supprimer le biais du premier coup.
   (c) Tour à tour, l'IA courante :
      - choisit un coup via `get_move` ;
      - le joue dans la partie réelle ;
      - incrémente les compteurs :
        - **nœuds explorés** (`coups_evalues`),
        - **score heuristique** du coup (`score_dernier_coup`).
3. En fin de manche, met à jour la clé `iaX.nom` ou `'nuls'`.
4. Après les n parties : calcule le temps total, le temps moyen, les sommes et moyennes de score par coup.

`tournoi_complet(nb_parties=10)`

But. Lancer un round-robin entre `IAFacile`, `IAMoyenne` et `IADifficile`.
Logique. Double boucle $i < j$ sur la liste des IA pour éviter les doublons. Chaque duel appelle `lancer_match` puis affiche :

- % de victoires et de nuls ;
- temps total et temps moyen ;
- **nouveaux indicateurs** : score total et score moyen par coup de chaque IA.

Retour. Une liste `res_global` contenant, pour chaque duel, un dictionnaire : `{duel, resultats, performance}`. Cela permet d'exporter facilement vers un graphique ou une annexe.

4.3.2 Fonction `menu_principal()`

Interface texte « tout-en-un » destinée à la démonstration :

- Six entrées : Humain/Humain, Humain/IA (3 niveaux), tournoi automatique, quitter.
- Appelle la fonction correspondante (`humain_vs_ia`, `tournoi_complet`, ...).
- Après l'action, propose de revenir au menu pour enchaîner plusieurs tests.

4.3.3 Point d'entrée

```
if __name__ == "__main__":
menu_principal()
```

Permet d'exécuter `tournament.py` directement : `python tournament.py` lance aussitôt le menu.

En résumé : `tournament.py` ajoute une *couche expérimentale* au projet : il généralise les statistiques (temps / scores heuristiques / nœuds), raffine l'analyse des IA et offre un lanceur rapide pour le correcteur.

4.4 Fichier `main.py` : point d'entrée du projet

Ce fichier ne contient qu'une seule responsabilité : déclencher le menu principal du projet, accessible en console. Il centralise l'accès aux différentes fonctionnalités du programme sans implémenter de logique métier.

4.4.1 Importation du menu principal

```
from tournament import menu_principal
```

Cette ligne importe depuis le fichier `tournament.py` la fonction `menu_principal()`, qui gère toutes les interactions en ligne de commande permettant à l'utilisateur de :

- jouer une partie à deux joueurs humains ;
- jouer contre l'IA de difficulté choisie (Facile, Moyenne ou Difficile) ;
- lancer un tournoi automatique entre IAs.

4.4.2 Bloc `if __name__ == "__main__"`

```
if __name__ == "__main__":
menu_principal()
```

Ce bloc permet de lancer le menu uniquement si le script `main.py` est exécuté directement (et non importé comme module). Il constitue le point d'entrée classique d'un projet Python.

En résumé, `main.py` agit comme un **lanceur unifié**, qui redirige l'utilisateur vers l'interface console développée dans `tournament.py`. Il n'implémente aucune logique de jeu ni d'intelligence artificielle.

5 Évaluation expérimentale des IA : résultats finaux

Les trois agents ont été confrontés deux-à-deux sur **50 manches**. En plus des taux de victoire, `tournament.py` enregistre :

- le *score heuristique total* cumulé par chaque IA (`score_dernier_coup`) ;
- la *moyenne* de ce score par coup réellement joué — mesure plus fine de la « qualité positionnelle » au fil d'une partie.

Extrait de la sortie console

```
=== Lancement du tournoi avec 50 parties par duel ===

Match : IA Facile vs IA Moyenne
Résultats :
  IA Facile victoires : 6 (12.0%)
  IA Moyenne victoires : 41 (82.0%)
  Nuls : 3 (6.0%)
Performance :
  Temps total : 202.11s
  Moyenne par partie : 4.0421s
  IA Facile score total : -9427820.00
  IA Moyenne score total : 11804190.00
  IA Facile score moyen par coup : -11697.05
  IA Moyenne score moyen par coup : 14325.47

Match : IA Facile vs IA Difficile
Résultats :
  IA Facile victoires : 0 (0.0%)
  IA Difficile victoires : 50 (100.0%)
  Nuls : 0 (0.0%)
Performance :
  Temps total : 232.14s
  Moyenne par partie : 4.6427s
  IA Facile score total : -5899500.00
  IA Difficile score total : 7984975.00
  IA Facile score moyen par coup : -11799.00
  IA Difficile score moyen par coup : 15209.48

Match : IA Moyenne vs IA Difficile
Résultats :
  IA Moyenne victoires : 3 (6.0%)
  IA Difficile victoires : 41 (82.0%)
  Nuls : 6 (12.0%)
Performance :
  Temps total : 134.27s
  Moyenne par partie : 2.6854s
  IA Moyenne score total : -4882920.00
  IA Difficile score total : 6205750.00
  IA Moyenne score moyen par coup : -4117.13
  IA Difficile score moyen par coup : 5145.73
```

FIGURE 1 – Log brut du tournoi (`nb_parties = 50`).

Tableau de synthèse

Match	% Vic. IA_1	% Vic. IA_2	% Nuls	Temps (s)	t/partie (s)	Score moyen IA_1	Score moyen IA_2
Facile vs Moyenne	12.0	82.0	6.0	202.11	4.04	−11 697	14 325
Facile vs Difficile	0.0	100.0	0.0	232.14	4.64	−11 799	15 209
Moyenne vs Difficile	6.0	82.0	12.0	134.27	2.69	−4 117	5 146

TABLE 1 – Tournois de 50 parties par duel (Intel i5-10210U CPU @ 1.60GHz, Python 3.11). Les deux dernières colonnes indiquent la moyenne du score heuristique par coup réel joué.

Analyse détaillée et justification des résultats

Rappel des chiffres clés. Pour chaque duel de 50 parties, les pourcentages *victoires / défaites / nuls* sont directement corrélés :

- à la **profondeur effective** explorée par Minimax ;
- à la **qualité de l'heuristique** (pondération à $\pm 10\,000$ par sous-plateau, bonus fourches, etc.) ;
- au **bruit stratégique** (30 % de coups aléatoires pour l'IA Moyenne).

IA Facile vs IA Moyenne

- **82 % de victoires pour la Moyenne.** Étant un Minimax profondeur 3, l'IA Moyenne voit *jusqu'à deux réponses* de la Facile avant de jouer, ce qui suffit à exploiter les erreurs 1-ply systématiques de son adversaire.
- **6 % de nuls.** Les nuls apparaissent lorsque la Moyenne, pénalisée par son `taux_erreur_strategique=0.3`, choisit un coup aléatoire qui rompt son propre plan de victoire et aboutit à un plateau global saturé (`mettre_a_jour_etat_global` → état « nulle anticipée »).
- **12 % de victoires pour la Facile.** Elles proviennent presque exclusivement des coups aléatoires *malchanceux* de la Moyenne : un seul mauvais placement dans un sous-plateau déjà critique vaut $-10\,000$ points à l'heuristique globale et suffit à inverser la partie.

IA Facile vs IA Difficile

- **100 % de victoires pour la Difficile.** L'ajout d'un demi-coup de profondeur (Minimax 4) *et* des coupures α-β élimine toute branche conduisant à une égalité défavorable. Chaque fois que la Facile laisse une double-menace (« fourche »), la fonction `evaluer_difficile` augmente son score de $+300$, ce qui dirige le moteur vers la ligne gagnante.
- **0 % de nuls.** Contrairement au duel précédent, aucun *bruit stratégique* n'est présent du côté de la Difficile ; l'arbre de recherche ne conserve donc pas d'issues neutres.

IA Moyenne vs IA Difficile

- **82 % de victoires pour la Difficile.** L'écart de victoire est moindre qu'avec la Facile, car la Moyenne voit quand même trois demi-coups dans l'arbre : elle neutralise certaines menaces immédiates, d'où **12 % de nuls**. Toutefois, sa profondeur insuffisante ne détecte pas toutes les fourches placées un coup plus loin.
- **6 % de victoires pour la Moyenne.** Celles-ci s'expliquent presque exclusivement par son *erreur stratégique volontaire*. Un coup aléatoire peut, dans de rares cas, sortir la Difficile d'une ligne d'évaluation locale optimale et lui imposer un sous-plateau déjà résolu.

Interprétation des métriques de score

- Les **scores moyens par coup** suivent la hiérarchie Facile $<$ Moyenne $<$ Difficile ; ils quantifient la *qualité intrinsèque* des positions choisies, indépendamment du résultat final. Ainsi, la Moyenne obtient $\approx -4\,100$ contre la Difficile : elle joue régulièrement des coups jugés perdants par l'heuristique avancée, mais tient parfois le nul.
- Les **scores totaux**, très élevés en valeur absolue ($\pm 8\times 10^6$), s'expliquent par la pondération $\pm 10\,000$ par sous-plateau (`evaluer_ultime`) multipliée par le nombre de coups : verrouiller tôt plusieurs petits plateaux fait exploser le cumul positif.
- Le **temps moyen** tombe à $2.7\,\text{s}$ dans le duel Moyenne / Difficile car les deux IA referment rapidement le plateau global : la Difficile force des victoires locales alors que la Moyenne, coincée, n'a plus qu'un choix restreint de sous-plateaux, réduisant le facteur de branchement.

Synthèse

1. La **profondeur de recherche** reste le déterminant principal du taux de victoire ; l'élagage α-β permet à l'IA Difficile d'explorer plus loin sans surcharge temporelle.
2. L'**heuristique fourche + mobilité** ajoute un gain qualitatif ($+300$ ou -300 par double-menace) qui se reflète dans le score moyen par coup.
3. Le **bruit aléatoire** intégré à l'IA Moyenne augmente la variance : il offre une petite chance de victoire surprise mais détériore le score moyen et, au final, le taux de victoire.

En définitive, les valeurs « victoires / défaites / nuls » et les métriques de *score heuristique* convergent : elles valident la progression pédagogique souhaitée des trois niveaux d'intelligence et démontrent l'impact mesurable de chaque amélioration (augmentation de profondeur, élagage, heuristique enrichie).

6 Bilan du projet

Ce que nous retenons.

- **Choix d'une représentation compacte.** Un état de jeu tient dans deux matrices 3×3 (sous-plateaux + vainqueurs). Cette structure s'avère suffisante pour parcourir efficacement l'arbre Minimax.
- **Impact direct des optimisations.** *Minimax* $+\,\alpha$-β $+$ *heuristique « fourche »* $\Rightarrow$ taux de victoire multiplié par 6 face à la version sans élagage, tout en divisant par 2 le nombre de positions évaluées.
- **Importance du paramétrage.** Un simple *taux d'erreur volontaire* (30 d'un style "parfait" à un style plus humain, sans toucher au noyau Minimax : utile pour calibrer la difficulté.

Difficultés rencontrées.

- Trouver une profondeur raisonnable : au-delà de 4 plis, le temps explose malgré l'élagage.
- Détecter les *double-menaces* sans alourdir l'heuristique ; plusieurs essais ont été nécessaires pour éviter les faux positifs.

Conclusion. Les trois niveaux d'IA proposés démontrent clairement l'effet des classiques de l'IA décisionnelle : *recherche en profondeur*, *élagage*, *fonction d'évaluation pertinente*. Le cadre d'Ultimate TicTacToe, bien que simple, s'est révélé idéal pour illustrer et mesurer ces techniques dans un temps d'exécution compatible avec une démonstration courte.

7 Sources

- Tutoriels IA Minimax :
  - https://dilipkumar.medium.com/game-theory-for-coding-440643d74e04
  - https://en.wikipedia.org/wiki/Minimax
  - https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-1-introduction/
- Wikipédia – Ultimate TicTacToe : https://en.wikipedia.org/wiki/Ultimate_tic-tac-toe