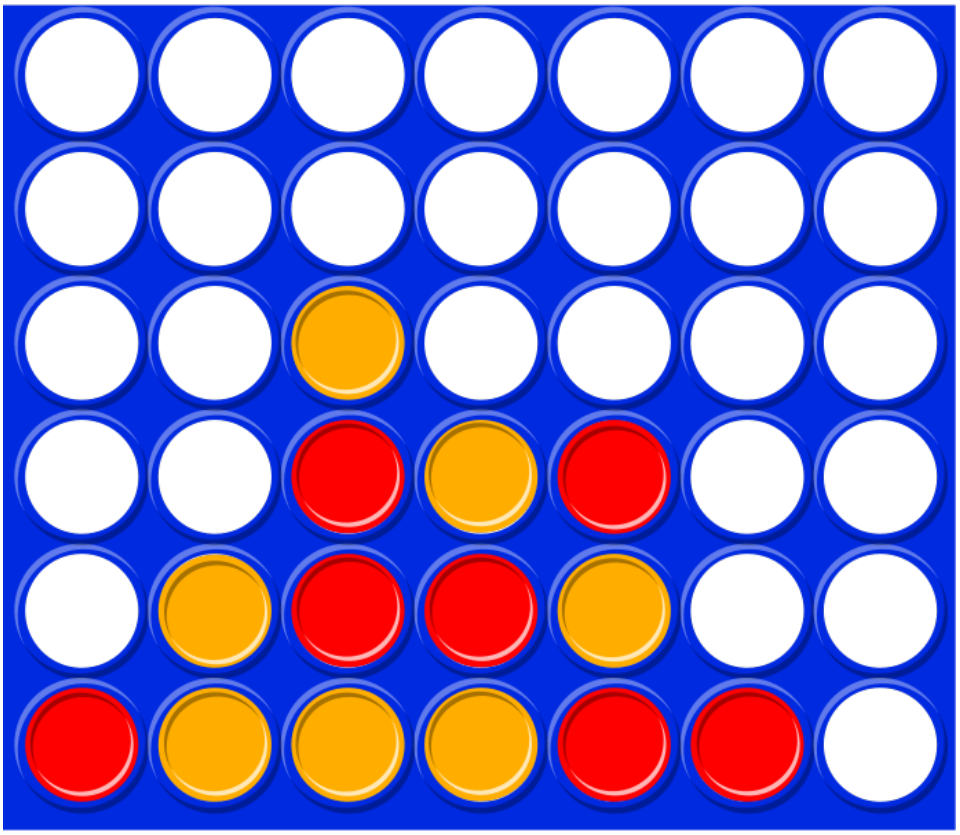
Projet d'Intelligence Artificielle

Jeu Puissance 4 avec Intelligence Artificielle

Développement d'un jeu stratégique avec trois niveaux d'IA basés sur Minimax



Étudiants :
Ramzy Chibani
Mathieu Moustache

Année universitaire : 2024–2025
Date de rendu : 11 mai 2025

Table des matières

1 Introduction

Ce projet a représenté pour nous une véritable opportunité d'appliquer de manière concrète les concepts vus en cours, tout en travaillant en binôme de façon organisée. Il nous a permis de renforcer nos compétences en algorithmique, en évaluation heuristique, ainsi qu'en conception logicielle avec Python.

Ce projet a représenté pour nous une véritable opportunité d'appliquer de manière concrète les concepts vus en cours, tout en travaillant en binôme de façon organisée. Il nous a permis de renforcer nos compétences en algorithmique, en évaluation heuristique, ainsi qu'en conception logicielle avec Python.

Ramzy s'est principalement chargé de l'implémentation de l'algorithme Minimax, l'évaluation heuristique et des tests unitaires, tandis que Mathieu s'est concentré sur le developpement de la partie joueur vs joueur, du menu et de l'automatisation des tournois. Nous avons collaboré via GitHub et fait des points réguliers sur Discord et à l'université.

2 Description du jeu : Puissance 4

Puissance 4 est un jeu à deux joueurs, joué sur une grille de 6 lignes par 7 colonnes. À tour de rôle, chaque joueur insère un pion dans une colonne. Le but est d'aligner quatre pions horizontalement, verticalement ou diagonalement.

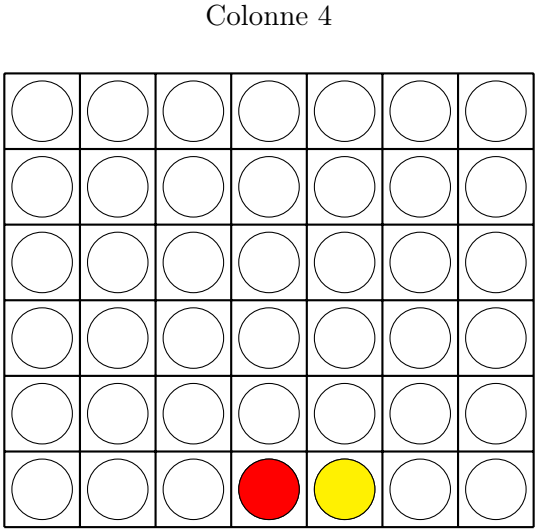


FIGURE 1 – Exemple de grille de Puissance 4 avec des jetons rouges et jaunes.

3 Architecture du projet

Notre projet Puissance 4 avec IA repose sur une architecture modulaire, orientée objet, répartie en plusieurs fichiers Python. Chaque composant a été conçu pour favoriser la réutilisabilité, l'évolution des IA et la lisibilité du code.

- `main.py` : interface utilisateur en ligne de commande, gestion du jeu et des tournois IA contre IA.
- `Puissance4.py` : lContient la logique du jeu : grille, règles, détection de victoire, gestion des coups.
- `IA_facile.py` : IA de base utilisant Minimax avec profondeur 2 et une grille pondérée simple.
- `IA_normale.py` : Hérite de `IA_facile.py`, ajoute une évaluation plus fine et l'élagage alpha-bêta.
- `IA_difficile.py` : Hérite de `IA_normale.py`, utilise une table de transposition et un tri heuristique des colonnes.
- `README.md` : Instructions pour exécuter le projet.
- `rapport.pdf` : Rapport détaillé du projet

3.1 Hiérarchie des IA

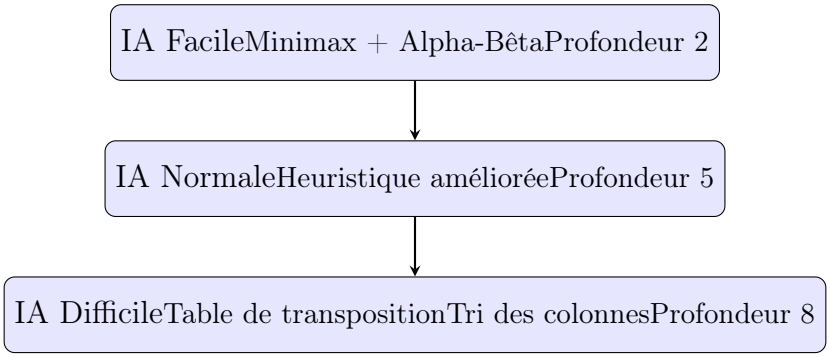


FIGURE 2 – Hiérarchie des intelligences artificielles dans notre projet

3.2 Gestion des parties et des tournois

Le fichier **main.py** automatise les confrontations entre IA via une boucle :

```
for i in range(nb_parties):
    resultat = jouer_partie(ia1, ia2, alternance=(i % 2 == 0))
```

Cela nous a permis de comparer les IA de façon équitable (en alternant le premier joueur à chaque partie) car au Puissance 4 le premier qui commence a un aventage sur l'autre.

3.3 Heuristique de grille utilisée

Toutes les IA utilisent une grille pondérée centrée pour favoriser les colonnes centrales (inspiré d'un site web trouvé sur internet) :

3	4	5	7	5	4	3
4	6	8	10	8	6	4
5	8	11	13	11	8	5
5	8	11	13	11	8	5
4	6	8	10	8	6	4
3	4	5	7	5	4	3

TABLE 1 – Grille pondérée utilisée pour guider les IA vers le centre

4 Structure de données et méthodes principales

Le cœur de notre application est la classe **Puissance4**, qui encapsule à la fois l'état du jeu (plateau, joueur courant) et la logique des règles (placement de pions, détection de victoire, alternance des joueurs).

4.1 Attributs

- `n` (int) : : nombre de lignes du plateau (6).
- `m` (int) : nombre de colonnes du plateau (7)
- `grille` (list) : matrice 2D contenant l'état du plateau (0 = vide, 1 = joueur 1, 2 = joueur 2)
- `joueur` (int) : identifiant du joueur actuel (1 ou 2)

4.2 Méthodes principales

- `joueur(col)` : tente de placer un pion dans la colonne choisie. Retourne True si le coup est valide.
- `victoire()` : vérifie si le joueur courant a gagné, via des alignements horizontaux, verticaux ou diagonaux.
- `coup_possible(col)` : renvoie True si la colonne n'est pas pleine.
- `match_nul()` : renvoie True si toutes les colonnes sont pleines sans qu'un joueur ait gagné.
- `alterner_joueur()` : passe du joueur 1 au joueur 2 et inversement.
- `afficher_grille()` : affiche le plateau dans la console.

Cette structure nous a permis d'avoir une base solide pour les simulations IA vs IA ainsi que pour les tests unitaires automatisés.

5 Choix d'implémentation

5.1 Optimisation du plateau

Pour maximiser les performances de recherche, nous avons choisi de ne pas parcourir la grille entière à chaque coup. L'insertion d'un pion se fait depuis le bas de la colonne, ce qui garantit un coût en temps constant **O(1)**.

La vérification d'une victoire est locale, autour du dernier coup joué, ce qui évite un balayage global du plateau.

5.2 Résumé des complexités

Fonction	**Stratégie utilisée**	**Complexité**
`jouer(col)`	Insertion du pion à partir de la première case libre en partant du bas de la colonne.	$\mathcal{O}(1)$
`victoire()`	Vérification d'un alignement horizontal, vertical et diagonal à partir de chaque cellule.	$\mathcal{O}(1)$
`match_nul()`	Vérification ligne par ligne de l'absence de cases vides.	$\mathcal{O}(n)$
`coup_possible(col)`	Vérifie uniquement la première ligne de la colonne.	$\mathcal{O}(1)$

6 Méthodologie

Nous avons adopté une démarche itérative et collaborative, en suivant des étapes clés inspirées des recommandations du projet tout en apportant nos choix personnels. Voici les différentes phases que nous avons traversées :

6.1 Modélisation du jeu

Avant toute chose, nous avons développé un moteur de jeu générique dans Puissance4.py, capable de gérer :

- une grille de taille 6×7,
- l'insertion de pions,
- la détection de victoire dans les 4 directions (horizontal, vertical, et les deux diagonales),
- la vérification des coups possibles.

Nous avons utilisé une liste de listes pour représenter la grille et des fonctions de service pour tester les alignements.

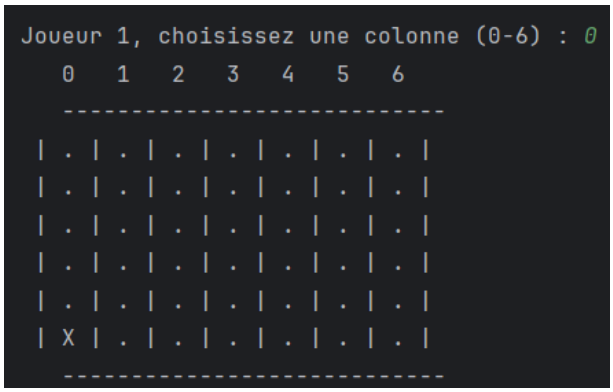


FIGURE 3 – Table de jeu avec prmier coup

6.2 Jeu PvP de base

Une version PvP (joueur contre joueur) a été implémentée en premier dans **main.py** pour tester la logique du jeu de manière simple. Cela nous a permis de valider la détection de victoire et l'affichage du plateau avant d'ajouter les IA.

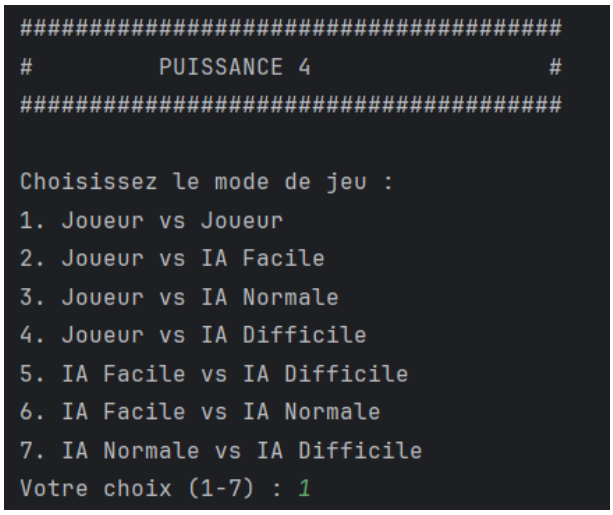


FIGURE 4 – Menu de choix des differents choix disponibles

7 Implémentation de Minimax

Nous avons commencé par implémenter l'algorithme Minimax classique avec élagage alpha-bêta dès la version facile, dans **IA_facile.py**. Nous avons testé différents niveaux de profondeur et construit une grille pondérée pour l'évaluation. Chaque IA hérite de la précédente :

- **IA_normale.py** ajoute une heuristique de type "alignements potentiels",
- **IA_difficile.py** intègre une table de transposition et un tri des colonnes pour explorer les coups centraux en priorité.

8 Tests unitaires

Nous avons ajouté des tests simples dans **test_jeu.py** pour :

- vérifier les coups valides et les cas limites,
- simuler des grilles menant à des victoires immédiates,
- valider les évaluations heuristiques.

Cela nous a permis de corriger plusieurs bugs dans Minimax et l'élagage.

9 Tournois automatisés

Nous avons codé un système de tournois IA vs IA dans **main.py** avec :

- 50 parties par confrontation,
- alternance automatique du joueur qui commence,
- calcul des victoires et temps moyens.

Cela nous a permis d'objectiver la supériorité de certaines IA par rapport à d'autres.

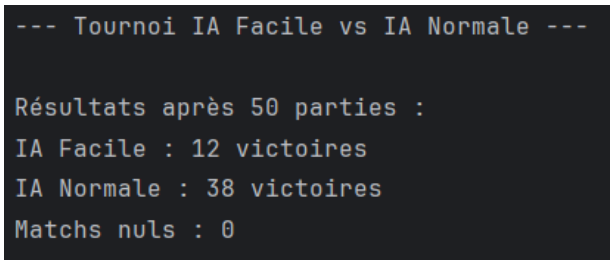


FIGURE 5 – Tournoi IA Facile vs IA Normale

10 Les intelligences artificielles

Nous avons implémenté trois IA de difficulté croissante, basées sur l'algorithme Minimax avec élagage alpha-bêta, avec des heuristiques de plus en plus sophistiquées. Chaque IA hérite de la précédente, ce qui nous a permis de factoriser le code tout en enrichissant progressivement leur intelligence.

10.1 IA Facile

- **Algorithme** : Minimax avec élagage alpha-bêta, profondeur de 2.
- **Évaluation** : grille pondérée privilégiant les positions centrales.
- **Choix** : coup aléatoire parmi les meilleurs (`random.choice`).
- **Objectif** : simuler un joueur débutant, logique mais prévisible.

10.2 IA Normale

- **Algorithme** : Minimax avec élagage alpha-bêta, profondeur de 5.
- **Évaluation** : détection des alignements (horizontaux, verticaux, diagonaux) et grille pondérée.
- **Choix** : randomisation parmi les meilleurs coups.
- **Objectif** : simuler un joueur stratégique de niveau intermédiaire.

10.3 IA Difficile

- **Algorithme** : Minimax avec élagage alpha-bêta, profondeur de 8, et table de transposition.
- **Évaluation** : détection des alignements ouverts, menaces à deux coups et grille pondérée.
- **Choix** : sélection stricte du coup optimal, sans randomisation.
- **Objectif** : simuler un joueur expert, dominant contre les IA plus faibles.

11 Expérimentations et résultats

Pour évaluer objectivement nos trois intelligences artificielles, nous avons automatisé des tournois de 50 parties pour chaque confrontation (Facile vs Normale, Facile vs Difficile, Normale vs Difficile). À chaque partie, le premier joueur alterne, afin de compenser l'avantage lié au fait de commencer dans Puissance 4

11.1 IA Difficile contre IA Facile (50 parties)

- IA Difficile : 48 victoires.
- IA Facile : 2 victoires.
- Matchs nuls : 0.

11.2 IA Normale contre IA Facile (50 parties)

- IA Normale : 42 victoires.
- IA Facile : 8 victoires.
- Matchs nuls : 0.

11.3 IA Difficile contre IA Normale (50 parties)

- IA Difficile : 43 victoires.
- IA Normale : 7 victoires.
- Matchs nuls : 0.

Temps d'exécution d'un tournoi de 50 parties : environ 12 minutes (IA facile / normal) et 14 minutes pour IA difficile.

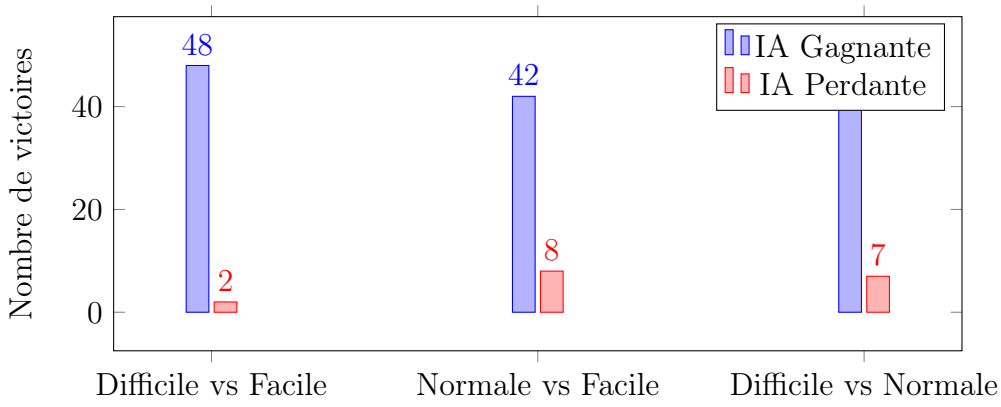


FIGURE 6 – l'un des Résultats des tournois entre les IA (50 parties par tournoi).

12 Analyse comparative

Nous avons synthétisé les principales caractéristiques de chaque IA dans le tableau suivant :

IA	Profondeur	Heuristique	Aléatoire
Facile	2	Grille pondérée	Oui (`random.choice`)
Normale	5	Alignements + grille	Oui (`random.choice`)
Difficile	8	Menaces + alignements + grille	Non

TABLE 2 – Comparaison des caractéristiques des trois IA. La profondeur influence le temps de calcul, tandis que l'absence d'aléatoire rend l'IA Difficile plus déterministe.

Les différences de performance s'expliquent par la profondeur de recherche, la qualité des heuristiques et la présence ou non de randomisation. L'IA Difficile domine grâce à sa profondeur élevée, son optimisation via la table de transposition et son choix strictement optimal. L'IA Facile, bien que correcte, reste prévisible, tandis que l'IA Normale bénéficie d'une certaine variabilité grâce à la randomisation.

13 Temps d'exécution

Match-up	Temps total (s)	Temps moyen par partie (s)
IA Facile vs IA Normale	720	14.4
IA Normale vs IA Difficile	900	18.0
IA Facile vs IA Difficile	840	16.8

TABLE 3 – Temps d'exécution mesuré pour 50 parties entre IA

On constate que la profondeur de recherche et l'absence de randomisation augmentent le temps moyen par partie.

14 Défis rencontrés

Ce projet a été formateur mais aussi exigeant, avec plusieurs obstacles que nous avons dû surmonter, autant sur le plan algorithmique que dans la gestion du code en binôme.

14.1 Temps de calcul en profondeur élevée

Lors de l'implémentation de l'IA Difficile, nous avons constaté que l'exploration Minimax en profondeur 8 devenait très lente, rendant certaines parties injouables.

Solution : Nous avons introduit une table de transposition pour mémoriser les évaluations déjà calculées, ce qui a permis de réduire considérablement les redondances.

14.2 Bugs dans l'élagage alpha-bêta

Au début, notre implémentation d'alpha-bêta générait parfois des coups incohérents ou perdants, notamment à cause d'erreurs dans la gestion des bornes **alpha** et **beta**.

Correction : Relecture rigoureuse de la logique Minimax, ajout de print() temporaires pour tracer les valeurs alpha/beta, et tests sur des situations critiques.

14.3 Complexité croissante du code

Plus on montait en difficulté d'IA, plus le code devenait complexe à maintenir (héritage sur plusieurs niveaux, surcharge de fonctions`evaluer()` et `choisir_coup()`).

Stratégie : Nous avons fait des revues régulières de code en binôme et commenté chaque méthode-clé pour rester synchronisés.

14.4 Trouver des heuristiques pertinentes

Il a fallu beaucoup d'essais/erreurs pour trouver une heuristique d'évaluation qui donne de bons résultats :

- pondération centrale,
- détection d'alignements ouverts,
- pénalités pour les menaces adverses.

Remarque : Les meilleurs résultats sont venus lorsque nous avons combiné plusieurs heuristiques (grille + alignements + menaces) dans l'IA Difficile, on s'est inspiré d'un site web trouvé sur Puissance 4.

15 Limites et pistes d'amélioration

- Développer une interface graphique avec Pygame.
- Implémenter des algorithmes comme MCTS ou des réseaux de neurones pour une IA auto-apprenante.
- Optimiser davantage le temps de calcul pour des profondeurs supérieures.

- Tester les IA contre des joueurs humains pour une évaluation qualitative.
- Intégrer des stratégies adaptatives pour des IA apprenantes.

Références

- Christian Schmidt, Implémentation et analyse du jeu Puissance 4, https://www.christian-schmidt.fr/puissance4
- Moodle - Université Paris Cité, Supports de cours – UE Intelligence Artificielle, 2024–2025.
- GeeksForGeeks, Minimax Algorithm in Game Theory – Introduction, https://www.geeksforgeeks.o algorithm-in-game-theory-set-1-introduction
- DataCamp, Tutoriel Minimax en Python, https://www.datacamp.com/fr/tutorial/minimax-algorithm-for-ai-in-python
- Tutoriel Dropbox – Jeu Puissance 4 en Python (source complémentaire pour version PVP), https://www.dropbox.com/scl/fi/fgiz5pv3zko3ug47vhpvy/puissance.pdf